

PPIG 2022

Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group

5-9 September 2022

The Open University, Milton Keynes & Online



Edited by
Simon Holland, Marian Petre, Luke Church, Mariana Marasoiu

Editor's Preface

PPIG 2022: Turing, Piaget, Radiohead: PPIG and the Muse

This year's theme was 'Turing, Piaget, Radiohead: PPIG and the Muse', aimed at sparking insights drawing on programming, abstractions, notations, psychology, and music.

This was the first PPIG to be held physically since 2019, following the two online-only PPIGs in 2020 and 2021, both during the Covid pandemic. It was also the first PPIG conference to be designed specifically for hybrid attendance. Reflecting the theme, it was hosted by Music Computing Lab at the Open University in Milton Keynes, and held a mere ten minutes from Bletchley Park, where Alan Turing and colleagues triumphed algorithmically in WWII.

Doctoral Consortium

Thanks to Marian Petre and Clayton Lewis for running the Doctoral Consortium on the first day of the conference (Sunday), and thanks to all the DC attendees: Andrea Bolzoni, Olli Kiljunen, Julia Busuttill-Crossley, Michael Moorhouse and Jude Nzemeke.

PPIG 2022 Award Organisation

Thanks to Clayton Lewis and Brett Becker for organising the games that spanned the whole conference, and judging and presenting the awards. Games included:

- non-secret words: allowing presenters to score style points by including implausible organiser-selected words in their presentation,
- secret words: challenging co-located participants to identify secret words known only to remote participants, sprinkled in online comments,
- games testing the limits of human collaborative cognition; GPT-3 prompt games,
- emergent games: retrospective crowd-sourced games/prizes.

Thanks to Marian Petre for organising the physical awards, which everyone won. Thanks also to Marian for the musician pig designs and to Daniel Gooch for laser cutting facilities.

Hybrid Art Competition

Separate from the above games, and to reflect the hybrid nature of the conference, an art competition was held on Monday and Tuesday which challenged participants to create artworks on the theme of the psychology of programming. Any medium was allowed, but the key constraint was that artworks should be presentable on Zoom in 30 seconds or less. To support aesthetic ideation, four drop-in workshops for physical attendees ran concurrently on three different floors of the Venables building, offering inspiration, and algorithmic artistic support. Thanks to the organisers of the four drop-in workshops:

Owen Green	Making Music with Fluid Corpus Manipulation
Marcel Valový and Brett Becker	Dall-E Prompt hacking for art's sake
Jason Carvalho and Enrico Daga	Haptic Bracelet guided Samba rhythms
Clayton Lewis	GPT-3 Prompt hacking & epistemics

Showings/performances of artworks were held after the main presentations in the demos & reflections section, in later parts of the week.

The Art of Fugue Contrapunctus XIX

After the main presentations on Monday, attendees took part in a hybrid experiment using musical harmony as a vehicle to evoke reflection on the representation of parallel processes. This involved listening to, watching, and responding to an animated visualisation of JS Bach's The Art of Fugue, Contrapunctus XIX, as played by the Danish String Quartet. The visualisation was created by MIRAGE, a comprehensive AI-based system for Advanced Music Analysis. We thank Olivier Serge Gabriel Lartillot of RITMO, Oslo, Norway for making the visualisation available and running the experiment remotely, providing participants online and in person with an opportunity to reflect on Bach's inspired but tightly rule-bound weaving of multi-stranded temporal patterns.

Putting the Rhythm in Algorithm

On Tuesday morning, two invited drop-in workshops focusing on insights from musical rhythm were run for physical attendees concurrently on two floors of the Venables building. These were designed to be enjoyable and rewarding in their own right, and to promote reflection on issues of foundational interest to PPIG. Thanks to the workshop devisers and organisers:

Alex McLean: Algorithmic drumming circle

An algorithmic drumming circle was kindly designed and led by invited speaker Alex Mclean. Participants were invited to bring their own laptops to collectively investigate and make rhythms with Alex's celebrated pattern weaving, manipulating and visualising systems, the Haskell-based Tidal, and the Web-based Strudel.

Noam Lederman: drum conversations with a real-time improvisatory drum agent

An improvisational drumming circle was organised by international-level session drummer, and music educator Noam Lederman. Participants were invited to improvise drum patterns using various hand drums in musical conversations with Boom Ka, the real time drumming agent designed and programmed by Noam as part of his PhD work in-progress at the Music Computing Lab. Boom Ka has been highly rated to date by professional drummers using it in shedding sessions – complementing this, the workshop demonstrated that dialogues with Boom Ka could be strongly engaging for beginners as well.

Local Volunteers

Thanks to Jonathan Bailey, Audrey Ekuban, Lida Shamiri, Caroline Holland and Michael Holland for valuable and greatly appreciated onsite assistance.

Technical assistance

Thanks to Andrea Bolzoni for work way beyond the call of duty in extended technical trouble-shooting over several days to support the aims of a convivial hybrid conference.

Thanks to Sam Hazell for calm, thorough, top class audiovisual support.

Spirit of PPIG

Thanks to Mariana Marasoiu and Luke Church for making sure the spirit of PPIG never got lost.

Onsite organisation

Particular thanks to Lesley Simpson for good humouredly and efficiently liaising with estates, security, catering, two hotels, taxis, the Plough, Bletchley Park and the National Museum of Computing, as well as firefighting and helping with registration and setup throughout the conference.

Keynote Speakers

The Psychology of Programming and the Psychology of Mathematics

Henry Lieberman (work with Warren Robinett)

MIT Computer Science & AI Lab

Many fundamental results in mathematical logic and set theory, among other mathematical fields, were developed long before the advent of computers and modern computer science. Many of these proofs depend crucially on symbol-manipulation procedures, originally described in the proofs with prose or with equations that specify constraints on the procedures. We recast some of these procedures with modern computer science concepts, in modern programming languages, and discover (perhaps not surprisingly), that... some of them have bugs. One reason that these bugs may have gone unnoticed for so long may be due to differences in how mathematicians and programmers think about procedures. Programmers think about control structures and temporal relationships, whereas mathematicians want to abstract away from time. The equation is ubiquitous in mathematical language, whereas programmers have discovered the "=" can have different roles: definition, equality testing, and assignment. Programmers think computationally, whereas mathematicians are comfortable with "there exists" descriptions, which may prove uncomputable. Finally, the two fields have different ideas about the notion of abstraction, which affect how their concepts are introduced and how they are used.

Making program analysis useful

Emma Soderberg

Senior Lecturer and Reader in Computer Science at Lund University, Sweden

Program analysis is meant to aid software developers in creating software, but often fails at this task. Why is this and what can we do about it?

Live coding and the 'what-if' paradigm

Alex McLean

UKRI Future Leaders Fellow and Live Coding pioneer

Live coding is an 'end-user programming' community of musicians and other performing artists, which has developed rather separately from the world of software engineering over the past 20 years. As a result, it has some peculiarities. In particular, improvisation is strongly promoted across the community, supported through technological developments such as pure functional reactive programming, in-code visualisation, and algorithmic approaches to pattern-making informed by heritage practices. Through this talk, I'll try to argue that this improvisatory approach offers a third paradigm in programming, combining the 'what' of declarative programming, and the 'how' of imperative programming, to offer an alternative: 'what if?'. I'll try to sketch out the difference, why it's needed, and how we might support its development. In the end, the question is how such a formal, explicit approach to notation as computer programming can help us explore what we know, but can't explain.

And finally, many thanks to all the attendees, contributors and referees, who collectively bought PPIG 2022 to life. We hope to see you, and diverse newcomers at PPIG 2023.

Simon Holland

December 2022



Drumming circle led by Noam Lederman's computational drumming agent, Boom Ka.



Live trace from Noam's real-time drum agent seen scrolling in background.



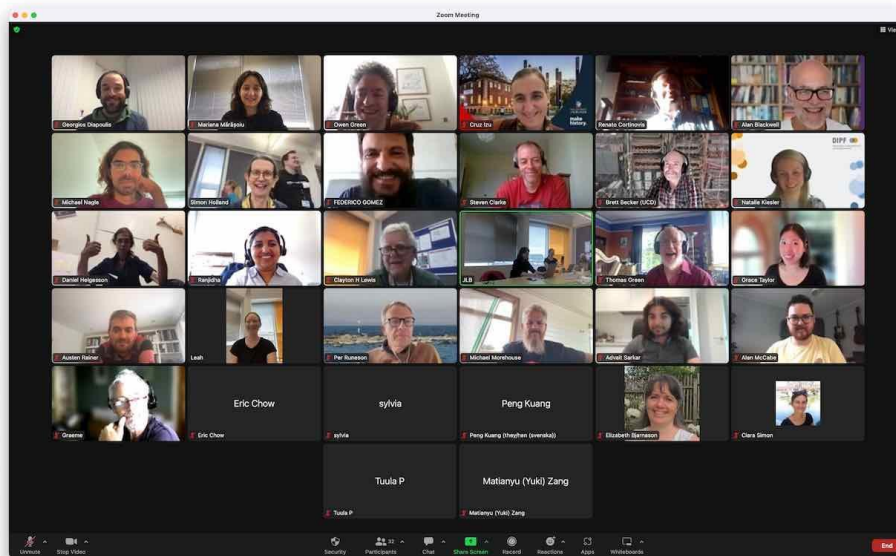
Strapping on a haptic bracelet to participate in collective samba drumming.



Algorithmic laptop drumming circle led by invited speaker Alex Mclean.



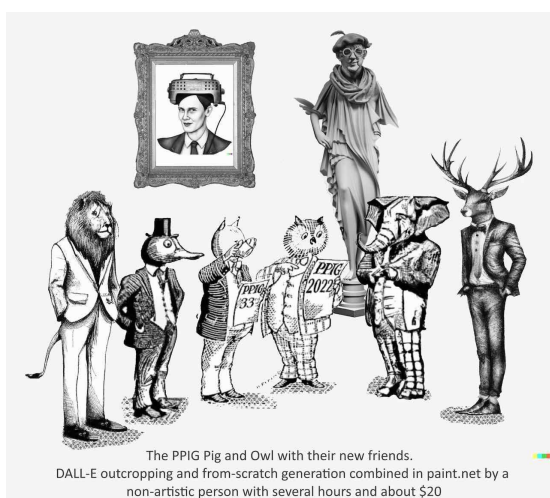
Participants empirically investigating whether Milton Keynes is all Turing, Piaget, and Radiohead.



Some of the 34 online participants.



Some of the 28 physical participants (including workshop leaders, doctoral consortium and volunteers).



Artwork created by Brett Becker with some help from Dall-E, and prior PPIG art by W. Rudling. Looking on are Alan Turing with a radio on his head, and Jean Piaget.



Still from an animation by Mariana Marasoiu, (winner of the PPIG 2022 Oliver Postgate Animation Award).

These are two representatives of the many diverse artworks created.

PPIG 2022

Programme & Proceedings Index

Monday, 5th September

9:30 - 10:00	<i>Coffee and registration</i>	
10:00 - 12:00	<i>In-person</i> Multi-media art competition & drop-in workshops	
12:00 - 14:00	Lunch	
14:00 - 16:00	<i>In-person</i> Multi-media art competition & drop-in workshops	
16:00 - 16:10	<i>Conference open - hybrid</i> PPIG 2022 Open and Welcome Doctoral Consortium 1 min intros	
16:10 - 17:10	<i>Presentations session - hybrid</i> Architecture about Dancing: Creating a Cross Environment, Cross Domain Framework for Creative Coding Musicians Owen Green, Pierre Alexandre Tremblay, Ted Moore, James Bradbury, Jacob Hart and Gerard Roma	14
	Visual cues in compiler conversations Alan McCabe, Emma Söderberg, Luke Church and Peng Kuang	25
	Intuition-enhancing GUI for visual programming Vasile Adrian Rosian	39
17:10 - 17:20	<i>Break</i>	
17:20 - 18:20	<i>Presentations session - hybrid</i> A Reflection on Distributed Cognition for Teamwork (DiCoT) framework for studying in Software Engineering Marjahan Begum, Helen Sharp and Yvonne Dittrich	
	Pilot Study: Validation of Stimuli for Studying Mental Representations Formed by Parallel Programmers During Parallel Program Comprehension Leah Bidlake, Eric Aubanel and Daniel Voyer	47
	Coding or AI? Tools for Control, Surprise and Creativity Alan Blackwell	57
18:20 - 18:40	<i>Hybrid</i> Demos & reflections The Art of Fugue Contrapunctus XIX, Olivier Lartillot	
19:30	<i>In-person</i> PPIG Dinner at The Plough Address: Simpson Rd, Simpson, Milton Keynes MK6 3AH	

Tuesday, 6th September

9:30 - 10:00	<i>Coffee and registration</i>	
10:00 - 12:00	<i>In-person (Venables Building)</i>	
	Algorithmic Drum Circle with Alex McLean (GR Complexity Lab - 2nd Floor)	
	Conversation with an expert drumming agent with Noam Lederman (Music Computing Lab - 1st Floor)	
12:00 - 12:45	<i>Lunch</i>	
12:45 - 15:15	<i>In-person</i>	
	National Museum of Computing visit	
16:00 - 17:00	<i>Presentations session - hybrid</i>	
	<i>Invited talk:</i>	
	Live coding and the ‘what-if’ paradigm	67
	Alex McLean	
	Story-thinking, computational-thinking, programming and software engineering	68
	Austen Rainer and Catherine Menon	
17:00 - 17:10	<i>Break</i>	
17:10 - 18:30	<i>Presentations session - hybrid</i>	
	Interactive Bayesian Probability for Learning in Diverse Populations	77
	Zainab Attahiru, Rowan Maudslay and Alan Blackwell	
	An agent for creative development in drum kit playing	88
	Noam Lederman, Simon Holland and Paul Mulholland	
	Would a Rose by any Other Name Smell as Sweet? Examining the Cost of Similarity in Identifier Naming	91
	Naser Al Madi and Matianyu Zang	
	Experimental Pair Programming Study Design (Work in Progress)	107
	Marcel Valový	
18:30 - 18:40	<i>Break</i>	
18:40 - 19:00	<i>Hybrid</i>	
	Demos & reflections	
	presentation of PPIG 2022 art challenge entries	

Wednesday, 7th September

10:00 - 12:00	<i>In-person</i> Bletchley Park visit	
around 12:00	Get your own lunch in Hut 4	
around 13:00	Free/travel home time	
	NB: From this point onwards, the rest of the conference is online only. (although the conference room at the OU will continue to be available for communal online participation)	
16:00 - 17:00	<i>Presentations session - online</i> Intro to today's session	
	Keynote: The Psychology of Programming and the Psychology of Mathematics Henry Lieberman	113
	A Tour Through Code: Helping Developers Become Familiar with Unfamiliar Code Grace Taylor and Steven Clarke	114
17:10 - 17:20	<i>Break</i>	
17:20 - 18:30	<i>Presentations session - online</i> What is it like to program with artificial intelligence? Advait Sarkar, Carina Negreanu, Ben Zorn, Sruti Srinivasa Ragavan, Christian Poelitz and Andrew Gordon	127
	POGIL-like learning and student's impressions of software engineering topics: A qualitative study Bhuvana Gopal, Ryan Bockmon and Steve Cooper	154
	On Writing Workshops for Programming Michael Nagle	164
18:30 - 18:50	<i>online</i> Demos & reflections Get GPT-3 to say...	

Thursday, 8th September

16:00 - 17:00	<i>Presentations session - online</i> Intro to today's session	
	Mastery Learning and Productive Failure: Examining Constructivist Approaches to teach CS1 Cruz Izu, Daniel Ng and Amali Weerasinghe	168

	The construction of knowledge about programs Sylvia Da Rosa and Federico Gómez	179
	Evaluating and improving the Educational CPU Visual Simulator: a sustainable Open Pedagogy approach Renato Cortinovis and Ranjidha Rajan	189
17:10 - 17:20	<i>Break</i>	
17:20 - 18:00	<i>Presentations session - online</i>	
	A Grounded Theory of Cognitive Load Drivers in Novice Agile Software Development Teams Daniel Helgesson, Daniel Appelquist and Per Runeson	197
	Livecode me: Live coding practice and multimodal experience Georgios Diapoulis	216
18:00 - 18:15	<i>online</i> Demos & reflections Parallel programming an analog game of Chinese whispers	

Friday, 9th September

16:00 - 17:00	<i>Presentations session - online</i>	
	Intro to today's session	
	Keynote: Making program analysis useful Emma Soderberg	225
	Mental Models of Recursion: A Secondary Analysis of Novice Learners' Steps in Java Exercises Natalie Kiesler	226
17:10 - 17:20	<i>Break</i>	
17:20 - 18:00	<i>Presentations session - online</i>	
	The impact of POGIL-like learning on student understanding of software testing and DevOps: A qualitative study Bhuvana Gopal, Ryan Bockmon, Steve Cooper and Justin Olmanson	241
	<i>Doctoral Consortium presentations:</i>	
	Tutorials Embedded in an IDE: A Feasible Way for CS Students to Learn Debugging? – A Study Design Olli Kiljunen	254
	Sound-based music style modelling, for a free improvisation musical agent Andrea Bolzoni	261
18:00 - 18:15	<i>online</i> PPIG Prizes and conference close	

PPIG 2022 Doctoral Consortium

Programme & Proceedings Index

Sunday, 4th of September

13:30	opening, introductions	
13:45	Julia Crossley - <i>Do mathematical proof skills in continuous areas of Maths improve algorithmic thinking in CS students in HE?</i>	251
14:10	Olli Kiljunen - <i>Tutorials Embedded in an IDE: A Feasible Way for CS Students to Learn Debugging? – A Study Design</i>	254
14:35	break	
15:00	Jude Nzemeke - <i>An Investigation of Student Learning in Computing Education Research</i>	257
15:25	Andrea Bolzoni - <i>Sound-based music style modelling, for a free improvisation musical agent</i>	261
15:50	AMA (ask me/us anything) – open discussion	
around 16:30	break for dinner	

Architecture about Dancing: Creating a Cross Environment, Cross Domain Framework for Creative Coding Musicians

Owen Green, Pierre Alexandre Tremblay,
Ted Moore, James Bradbury,
Jacob Hart, Alex Harker
Centre for Research in New Music
University of Huddersfield
{o.green, p.a.tremblay, t.moore,
j.bradbury, j.hart2, a.harker}@hud.ac.uk

Gerard Roma
School of Computer Science
Leeds Trinity University
g.roma@leedstrinity.ac.uk

Abstract

In this paper, we offer reflections on the (still-forming) outcomes of a five-year project, situated in a context of artistic research around music technology, that seeks to facilitate the use of machine listening and machine learning techniques for creative coding musicians working in the Max, Pure Data and Supercollider environments. We have developed a suite of software extensions and learning materials, and, unusually, we have included community development efforts in our work. Whilst the project has no doubt differed in aims, methods and knowledge claims to how PPIG researchers may approach these topics, we feel there is significant common interest in a number of the emerging themes. We focus here on continuing attempts, by user-programmers and library programmers alike, to navigate various tensions thrown up by ambitions for the project's *accessibility*, *community* and *continuity*. Among these tensions are: cross environment legibility *vs* cross domain legibility between music and data-science *vs* environment idioms *vs* (unknown) idiosyncratic working patterns *vs* quick iterative design *vs* maintainability and longevity *vs* low cost of entry *vs* penetrability (Clarke & Becker, 2003).

1. Introduction

This paper offers some reflections from the late stages of a five-year project, *Fluid Corpus Manipulation* (FluCoMa), that has sought to make a toolkit of signal processing and data science extensions available to creative coding musicians who work in the Max (Zicarelli, 2002; Puckette, 2002), Pure Data (Puckette, 1997) or Supercollider (McCartney, 2002) languages. Our goal here is to bring into focus how a range of different priorities interacted to produce some specific design choices, and some consequences these priorities have had for the usability of the toolkit, based on the experiences of users.

We will briefly introduce the project and its disciplinary background. We place our efforts in the context of some earlier work in this area that serves to help explain how the priorities that steered the design and implementation process took root. We then present four vignettes describing aspects of the toolkit that continue to present difficulties to users, and point towards how our design priorities played a part in producing these issues. A takeaway is that whilst these priorities have served us well in getting this far, they may be coming to the end of their usefulness, both for dealing with remaining usability issues, and as the project transitions (we hope) into community development.

2. Background

We will sketch out the background context to the FluCoMa project here to the extent that we can illustrate how a particular set of design priorities came to inform development. For a fuller account of the toolkit's foundations and eventual form, interested readers can consult Green, Tremblay, and Roma (2018) and Tremblay, Roma, and Green (2022).

Whilst the bulk of computer based music making still happens in the context of digital audio workstations (DAWs) that mimic the functionality of mixing consoles and multitrack recorders, a long-running thread of creative work and research has focused on environments that are more open-ended in how they allow musicians and other artists to work with sound. As computing power has increased, musical

languages have augmented symbolic data processing capabilities with ways of working directly with audio, and with that has come an interest in getting and working with data *from* audio, ranging from very simple analyses (like tracking the energy of a signal) to the much more complex (like ‘un-mixing’ the voices of a polyphonic sound). Meanwhile, as machine learning techniques have become more tractable and accessible, there has been interest in how these can be used to help facilitate or organise some of this work.

Two particular strands of prior work in machine listening and learning for music had a formative influence on the types of task we wished to enable and explore with our toolkit. The first of these, pioneered by Rebecca Fiebrink’s *Wekinator*, uses simple machine learning models as an alternative paradigm for programming *mappings* between input data and controls for synthesisers or other processes (Fiebrink, 2019). Crafting such mappings plays a very large role in the programming of interactive and generative music systems. Doing this work by hand can be tedious and unintuitive, as well as frequently producing brittle results, especially when using data derived from audio. The second strand, exemplified by Diemo Schwarz’s *CataRT* (Schwarz, Beller, Verbrugghe, & Britton, 2006) uses audio feature analysis as a tool to discover and play with relationships between sounds in a corpus. Typically this involves an interface that uses a 2D scatter plot of segments of sound, arranged according to some audio descriptors or, in more recent manifestations, according to some dimensionality reduction process (Roma, Green, & Tremblay, 2019; Garber, Ciccola, & Amusatogui, 2020)¹.

The currently dominant languages for creative-coding musicians—Max, Pure Data and SuperCollider—have a shared emphasis on working in real-time, making them useful not only for composing, but also for building custom instruments, installations or even co-players. All are also able to host extensions via plugins written in C or C++. Despite this common focus on real-time work, they are very different environments (see Nash, 2015, for a fuller comparison). Max and Pure Data are superficially similar data-flow-like languages that share the same inventor. A ‘box-and-arrow’ type patching idiom promotes a focus on processes rather than data, and the facilities for manipulating data structures are quite limited (though different) in each. However, Max and Pure Data have diverged somewhat from their common roots, both in terms of what is possible or easy to achieve in each, but also in terms of what is idiomatic to each.

SuperCollider, meanwhile, is a very different environment with a completely different architecture where the language (‘sclang’) runs in a separate process to real-time audio processing (the server), leading to very different usage experiences and constraints. sclang is a textual, dynamically-typed language, with some similarities to Smalltalk, and sophisticated facilities for manipulating data and expressing the timing and patterning of musical data. It communicates with the server (which actually does the sound production) via a network protocol (using UDP by default). The server is primarily focused on the playback of graphs of sound generators that have been specified in the client language, although there are some limited facilities for adding new ‘commands’ for offline processing. Facilities for communicating back from the server to the language are limited and awkward. Whilst there is a C-based SDK for SuperCollider, this applies only to the server (for writing new sound generators or commands), and not to the language, so that all FluCoMa components need supporting scaffolding in sclang.

Whilst the built-in support for analysing and working with audio data has been quite limited in these environments, there have been various extensions and packages available for each language, such as the *MuBu and Friends* package for Max (Schnell et al., 2009). An impetus for the FluCoMa project was that, despite much promise, existing extensions in this area left a number of things to be desired from the point of view of supporting widespread uptake and long-term artistic research, such as:

1. Imposing a whole new language to learn on top of the host environment.

¹For an impression of how our toolkit looks and works in practice, a video tutorial showing how to construct a CataRT-style exploration interface in Max can be found at <https://learn.flucoma.org/learn/2d-corpus-explorer/>. To compare the toolkit’s feel in different languages, these videos demonstrate equivalent examples in Max (<https://youtu.be/cjk9oHw7PQg>) and SuperCollider (<https://youtu.be/Y1cHmtbQPSk>)

2. Having only sparse or expert-level documentation, whilst also introducing many new concepts.
3. Not releasing source code and / or being unmaintained after a short time.
4. Only being available for a single creative coding environment, inhibiting portability and cross-communication.

From this background, the FluCoMa project had three overarching thematic preoccupations that would crystallise into a set of design priorities as work progressed:

Accessibility Whilst recognising that we would be addressing a reasonably advanced subset of creative coding musicians with this toolkit, there is no particular reason to believe that being an expert user of one of these environments translates into an appetite for finding one's own documentation or for learning more new languages. Furthermore, we knew from experience that such appetite can vary quite markedly depending on where in the progress of a creative project one is.

Community A key rationale of the project is that it should deliver enabling conditions for more and better artistic research around data-driven music making, meaning that we are concerned not just with augmenting the possibilities of our own artistic practices but with establishing a community of interest around this topic. It was crucial for us that this community forming took place *alongside* technical work. This is because we were conscious that developing in isolation could shape the affordances of the toolkit too strongly around our own musical proclivities or ways of thinking, and result in more exclusive and less interesting work. Moreover, evaluating artistic research *requires* community: it is by nature discursive and qualitative, and poorly served by proxy measures.

Continuity Artistic research takes time: pieces take a long while to complete, instruments require many hours of practice and performance, and typically research questions and concrete musical goals solidify and emerge over the course of practical investigation rather than being clear at the outset of a project. The timescales involved will very often be far in excess of funding periods, so artistic researchers are justifiably nervous of tools that may stop working, or simply disappear.

These themes played a structuring role in the overall approach taken to development, as well as coming to inform (more or less explicitly) design choices made along the way. A core commitment of the project, serving both accessibility and community, was that the toolkit should target more than just a single host language, the better to reach a wide range of practitioners and benefit from the distinct ways of thinking about music and code that working in different languages might bring (McPherson & Tahiroğlu, 2020).

2.1. Project Phases

The early phase of the project was structured by two waves of professional commissions with public performances, ensuring that we would have some committed input from fellow experts prepared to endure using software in a state of flux whilst pushing some work through to a developed state over the course of some months². As the toolkit matured towards public release, the later stages of the project have been marked by wider participation, both via our forum at <https://discourse.flucoma.org/>, and over 30 workshops delivered to a mixture of researchers, students and independent artists, mostly in Europe and North America.

Expanding our user pool in this later phase has also allowed us to dramatically improve our documentation based on being able to observe and discuss where people experienced problems. As well as platform-specific help files and reference material, we developed a web site of learning resources at <https://learn.flucoma.org/>, focussing on concepts and usage patterns rather than the specifics of particular components.

²Documentation of these commissions can be found at <https://www.flucoma.org/commissions/> along with some deeper analysis amongst the articles at <https://learn.flucoma.org/explore/>

2.2. Design Priorities

Certain design priorities fell out of this project structure quite naturally:

Rapid Development To make sure that we could be responsive to our commissioned artists’ experiences with early versions of the tools, we tried to arrange matters to produce and trial new components rapidly, across each of our host environments. To support this way of working we made heavy use of C++ templates to establish a host-agnostic way to specify the form and behaviour of a component (a ‘client’) that would yield a well-formed extension in each of Max, Pure Data or SuperCollider (see Figure 1).

Maintainable Code Other priorities, meanwhile, needed to be observed *despite* this fast-paced, iterative way of working. Trying to ensure continuity for the project, for instance, has meant always bearing in mind what impact on future maintenance adding features might have, especially considering that we would ideally like this maintenance to be a communal affair in the future.

Idiomatcity A key feature of accessibility is whether an API respects established idioms for the language it targets, which may well not be enforced by the language, but reflect common working practices that in turn serve to support sharing and discussing code. At the same time, people pick up these particular languages in quite distinct ways and there may well be multiple idioms emerging from different enclaves or traditions. Furthermore, we are addressing artists, many of whom might well have playful and idiosyncratic ways of working, which we would not wish to hinder.

Legibility In the interests of accessibility and community, we have tried to ensure that both the functionality and form of the APIs provided are consistent between each of our target languages, the better to enable communication and comparison of practices, so that code using our toolkit in one language should be legible to users of another language. We are also, of course, trying to make legible concepts from outside the musical domain, imported from signal processing and data science. Both of these exist in tension with *idiomaticity*.

Complexity Accessibility is also served by trying to ensure that the cost of participation is low: it should be possible to do interesting things with a few components, with little tweaking and not too much recourse to documentation. Meanwhile, however, both community and continuity require that there is scope to do new and powerful things, and that more complex experimentation is both feasible and rewarding. The process of bringing a creative project to fruition always involves moving between different types of programming practices, and it is important that we can reward ‘sketching’ and ‘tinkering’ whilst also supporting more orthodox programming, as one ‘toughens’ a patch ready for performance or distribution ([Bergström & Blackwell, 2016](#)).

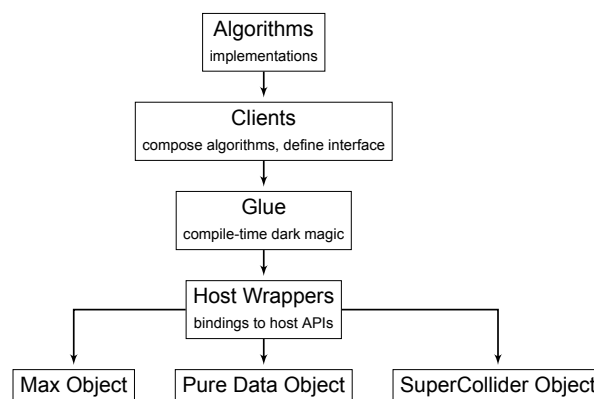


Figure 1 – The Supporting C++ Architecture

Clearly, it would be impossible to satisfy all these principles all the time because they pull against each other. Furthermore, we have to operate within the engineering constraints of what are actually pretty different languages and supporting platforms. This is especially true with respect to SuperCollider's distinctive architecture: because it is only possible to write C++ extensions for the server side, coming up with ways of working that satisfy both a sense of what is locally idiomatic and recognisably 'FluCoMa' is a delicate balancing act.

These tensions notwithstanding, we are reasonably happy that an approach of pragmatically balancing between these factors has served us well so far. The toolkit and its associated resources have been received well: people are using it in their work unprompted, and contributions of discussion, feedback, and code from new people are coming in. Nevertheless, the ways in which we have been weighing between these factors will probably have to change. The following section details four areas where we have noticed that people experience particular difficulty in using the tools (this is not an exhaustive list, mind you), and we reflect upon how these points of friction materialised.

3. Four Vignettes of Difficulty

We will present here four brief vignettes examining aspects that users have found it hard to learn or that are cumbersome in practice. We discuss how some of our original design decisions brought these about, and how / why it is hard to arrive at solutions that continue to balance the principles described in Section 2.2.

3.1. Components With Many Parameters

Dealing with an abundance of parameters is a routine problem in designing an API in any language. Many of the algorithms we have implemented in the toolkit have a great many and, when in doubt, we have often opted to expose things rather than hide them on the basis that any encapsulation can prematurely foreclose possibilities for creative experimentation. However, this can cause problems for users in knowing what might be most useful to reach for at first, or in grasping how different parameters might be related. In addition, parameters are exposed to users quite differently across our three environments, and the available mechanism to help users make sense of large sets of these things also vary (see Figure 2).

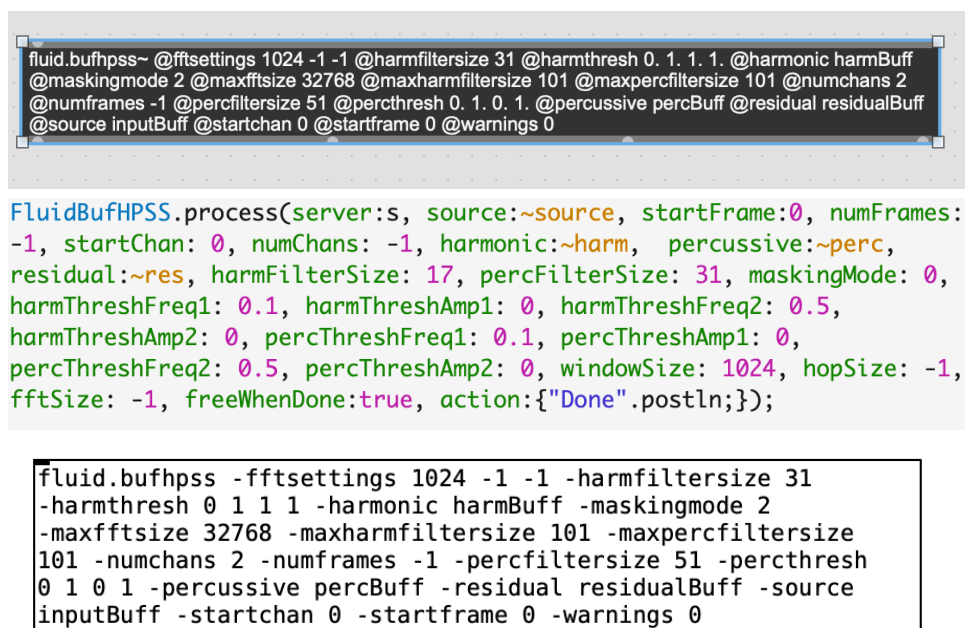


Figure 2 – Views of the same component with many parameters in Max, SuperCollider and Pure Data.

In principle, there are different tactics we could investigate to alleviate this. However, because of the varied ways the environments expose parameters, these tactics push against our desire to keep things consistent, and reduce maintainability by demanding environment specific approaches. For instance, we could try to group together parameters so that the ‘primary’ and ‘tweaky’ controls for an algorithm are clearly separated, or so that parameters addressing a given aspect are encapsulated together. In SuperCollider this could be done by bundling parameters into new slang classes that signal their relatedness. In Max and Pure Data the mechanism isn’t so clear. An alternative would be to encapsulate whole components into simpler interfaces for newcomers, and expose fewer moving parts. This could be done through abstractions in Max / PD and wrapper classes in SuperCollider, but at the cost of a great deal of duplication across environments that becomes hard to maintain as the number of components increases. A more tenable solution would involve returning to our C++ APIs and working on making it possible to compose ‘clients’ at this level.

3.2. Buffers as Universal Containers

Early on in development we realised that as well as having real-time processes, we needed processes that worked offline on stored pieces of audio data. Some algorithms only work this way (because they are not causal), but such a facility also enables one to process data in batches, often faster than real-time. This raised a question about what the output of such offline processors should be contained in when the result wasn’t audio data (e.g. some kind of analysed feature like pitch or loudness). We settled on using the same containers that the host environments use to store audio, variously called ‘buffers’ (Max, SuperCollider) or ‘arrays’ (Pure Data). It seemed a straightforward decision: all three environments had such a component with which users would already be familiar, and they were also scalable up to very large sizes (unlike the ‘list’ types in Max and Pure Data).

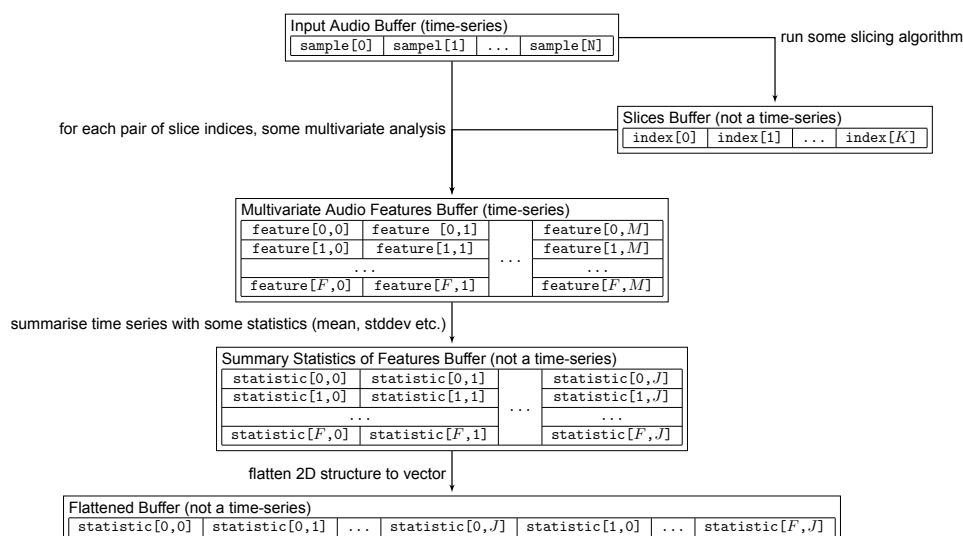


Figure 3 – Various different structures we impose on the same generic container, at different points in an audio analysis workflow

However, when we also encountered the need to output things that weren’t simple time series, we stuck with buffers, primarily in the interests of being able to keep moving. This has contributed to a number of difficulties we see people having:

- The very fact of using buffers ‘off-label’ by putting non-audio data into them causes trouble for some users, though fortunately seldom long-lasting. Clearly, we violate a principle of least astonishment at some level for these users.
- More stubborn difficulties arise when people need to manipulate data in these buffer objects. These are two-fold. First is that reshaping operations, such as flattening a two-dimensional matrix

of values to a vector, can be hard to conceptualise in the absence of Matlab / NumPy-like facilities for inspecting the effects of such moves.

- All this is exacerbated by the extent to which we have ‘overloaded’ the usage of buffers to also contain things that either aren’t time series (vectors of statistics or slice points), or try to express more dimensions than two in what is an inherently 2D structure.
- Slice point buffers are especially vexing, because there’s not a nice way to deal with them across platforms. Typically, we wish to iterate over pairs of points to analyse sections of an audio buffer, and this currently always pushes error-prone boilerplate on to the user: iteration in Max and Pure Data is famously awkward. Meanwhile, in SuperCollider, the same iteration involves bringing the buffer back to the language to be able to iterate over it, and then launching a stream of processes back on the server. Because this is all asynchronous, it is all too easy to end up with code full of nested callbacks that are hard to reason about.

A big part of the problem here is that the environments’ native buffer / array components don’t expose much direct machinery for doing things beyond playing back audio. In Max and Pure Data we can copy into a native ‘list’, which allows for more direct manipulation, but is limited in both size and representative power (flat and one-dimensional only). In SuperCollider, we can talk to Buffers in the context of Synths on the server (which is awkward), or stream them back to the language (which is slow and awkward).

3.3. Iterative Workflows in Machine Learning

An important set of components in our toolkit provides building blocks for some basic machine learning tasks, such as supervised and unsupervised learning, and some data pre-processing. Figure 4 shows an example pipeline that combines some of these techniques. The shape of our API is heavily based on scikit-learn (Buitinck et al., 2013), a machine learning toolkit for Python, which gave us something solid and proven to go on quickly, and meant that there was a useful documentation resource available whilst our own learning resources caught up.

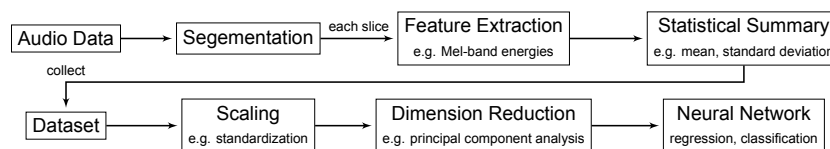


Figure 4 – An example machine learning pipeline

Whilst this building-block approach works well for being able to quickly gather data and assemble chains of algorithms, it has become apparent that it doesn’t really communicate a core expectation of machine learning development practices: the process of building up a useful model is always *iterative*, both in the senses of tweaking or perhaps swapping out each stage of a pipeline, but also that one will return frequently to one’s data to experiment with its composition and analysis. In this sense there is a moderately normative aspect of the practice that our API wishes to support that is left implicit by the interface, and the costs of it not being apparent might be that newcomers simply get discouraged by poor initial results.

Our target environments contribute different sorts of difficulty to this sort of iterative experimentation, which suggests that solutions that maintain a consistent API might not be forthcoming. In the patching languages (Max and Pure Data), assembling a pipeline is easy enough, but repeatedly re-assembling or re-organising can be a drag as things tend to get messy, and there are a great many mouse clicks involved. In SuperCollider, by contrast, slang makes it much easier to reorganise things and also has more powerful tools for manipulating data structures. However, the data and algorithms are all on the server, meaning that people can end up having to manage a great deal more traffic between language and server than is desirable.

For many musical users who are used to working in digital audio workstations or synthesisers, the basic idea of such an iterative approach to finding a sweet spot would be familiar enough from the practice of designing sounds through chains of audio generators and processors. However, what is conspicuously different here is the immediacy of feedback and, crucially, the ease with one can make sense of it. This leads us on to the issue of how one can evaluate results in machine learning pipelines and the question of interpretation.

3.4. Making Sense: Evaluation and Interpretability

Sometimes all that is needed to verify that one is happy with how a model is performing in a musical context is to play with it and decide if one likes the results. However, if one isn't happy, then it can be hard to know what to do about it. So far, we haven't built in typical supporting machinery for model evaluation, such as mechanisms for performing cross-validation or comparing performance on training data to performance on held-out test data. In part this is because of the usage and implementation complexity they would add, but also because the results of these processes still require careful interpretation. Where this is most evident is with our neural network components, where we have noticed that new users can become overly focused on isolated values for the training error, which signals—at the very least—that we need to better document what limited sense can be made from this number.

More generally, we notice that people struggle with the increasing abstraction away from perceptually explicable quantities as they move through pipelines like that shown in Figure 4. Supervised processes, like neural networks, are hard to look 'into', whereas the output of unsupervised algorithms like dimension reduction can be hard (or impossible) to interpret in sonic terms. Even certain audio analysis features can be difficult to relate directly to aural experience, such as Mel frequency cepstral coefficients (MFCCs), which are commonly used but rarely well explained³.

Because all the components of such pipelines interact, difficulties in interpretation become compounded and innocuous-seeming steps, like whether or how to normalise the ranges of input data, have presented obstacles to some users. The 'correct' choice for such a step cannot be given *a priori* because it depends both on the condition of the input data and often also on the assumptions of algorithms further downstream. For instance, any algorithm that involves a distance calculation is likely to do better if all input dimensions are uniformly scaled, and other algorithms may also depend on the data being centred. Using the sample mean and standard deviation for this centring and scaling is very much standard practice in many machine learning workflows, but whether it actually makes sense to use these statistics depends on the data itself. What if the distribution is not Gaussian, for instance? Again, we see implicit conditions on what effective usage patterns might be that are not directly expressed by the API, and whose explanations can get very technical quite quickly.

Established and emerging mechanisms for evaluating and interpreting machine learning models tend to be numerical or visual. So far, we've hesitated to introduce more than quite minimal visual facilities into the toolkit because, besides being hard to program well, they are non-portable between our hosts and therefore represent a real maintenance headache.

4. Discussion

With these four vignettes our intention has been to develop a critical assessment of our toolkit from a user perspective, and to illustrate some concrete challenges that arose from trying to balance contradictory priorities over the course of development. In ending up having components with a great many parameters, presented homogeneously, we opted to err on the side of exposing more moveable parts in the interests of not imprinting the API with whatever we happened to think was musically important, and to leave room for experimentation. Ways of mitigating this that also preserve consistency between the APIs exposed in different hosts have so far eluded us. Opting to use our hosts' audio buffer components in a variety of ways unintended by their original authors served the interests of rapid development and of accessibility, insofar as users didn't have to learn a whole new container, but nevertheless is a persistent

³We have had a go: <https://learn.flucoma.org/reference/mfcc/>.

source of confusion for new users that, again, resists easy fixes that would also preserve the consistency of experience between environments.

Meanwhile, taking a building block approach to making some data science facilities available was motivated by wishing to balance immediate usability against scope for deeper experimentation, whilst not foreclosing unforeseen musical outcomes by making advance judgements about what is important or desirable. What we have ended up with resembles quite closely standard machine learning toolkits like Python's scikit-learn. On the plus side, this provides a proven basis of an effective API, and a smooth route for users wishing to experiment in a different setting. However, there are aspects of common practice that remain implicit in our implementation, and this can be confusing. Both dimensions of this that we described—the expectation of an iterative workflow and the difficulties in interpreting what's happening—point not just at the possible fruitfulness of offering some alternative abstractions but, more nebulously, at a need to try and make the statistical languages and practices of data science more legible for musicians, perhaps even aurally.

While none of these challenges are insurmountable, addressing them may well mean relaxing adherence to our guiding principles. In particular, striving to maintain consistency of both functionality and form across three quite different environments finds itself in increasing tension with the ambition that the tools should slot neatly into the idioms of each host, and that they should be accessible both to less experienced (or, indeed, less patient) users whilst also offering a rich palette for experimentation. Of course, a further difficulty with this ambition for cross-host consistency is that individual users may well not care: if something's difficult to use, it's possibly not of much comfort to learn that this difficulty stems from a lofty principle, however well-intentioned.

Furthermore, as the project ends its initial phase of development and moves into what we hope will be a more communal mode of maintenance and enhancement, it seems clear that the balance of priorities and principles would have to shift in any case. The idea of cross-host consistency has been possible to try and stick to as long as we were a small group of developers working full-time. Insisting that any and all future community contributions place the same level of focus on this is unrealistic, especially if we don't wish to deter people from getting involved. Similarly, our ideas of what is and is not maintainable will probably shift. At present, these estimations are very much conditioned by the very fact that we are so few people, as well as what kinds of change the existing C++ framework makes more or less difficult, itself a function of how we have balanced priorities to date. One especially pressing body of work left to address *before* the project moves into community development is the maintainability of the C++ code itself, parts of which are suffering from repeated duct-tape solutions and over-obscurity, generally made in the interests of trying to keep development iterations reasonably rapid. Crucially, just as we have tried to create scope for progressively deeper engagement with the toolkit, we want something similar for potential contributors of new algorithms and components in C++, in the form of a entry-level API that tries to minimise the barriers to contribution.

If we are to be optimistic (and why not?), we may well hope that a wider pool of contributors will be enabling in other ways. For instance, in this initial phase we have been wary of encapsulating *too* much for fear that we would imprint our own, limited musical priorities on the toolkit too strongly. However, a more distributed mode of development opens up more scope for a variety of different encapsulations and ways-in that can address different musical practices and technical proclivities. In this way, we might also hope that some of the problems of legibility pointed at in sections 3.3 and 3.4 could be tackled in varied ways. If one dimension of the problems described is that there are implicit workflow expectations that aren't clearly illustrated by having all these separate building blocks, another is that the tools we have so far borrowed from data science remain generic and that we don't yet have a vocabulary for articulating what is characteristic of different types of musical data, or for relating our aural experiences of these data to statistical explanations.

5. Conclusion

Over the course of five years of research and development, the Fluid Corpus Manipulation project has put together a toolkit for creative coding musicians that consolidates and builds upon a range of prior work that provides functionality for audio decomposition, analysis and basic machine learning workflows in Max, Pure Data and SuperCollider. The toolkit has been successfully used to produce a number of complete pieces, has been enthusiastically received by user communities, and successfully taught to newcomers in over 30 workshops, which gives us some confidence that our ambitions for the learnability, utility, performance and robustness of the tools are being fulfilled. This has been achieved in part by following a pattern of (relatively) rapid, iterative software development, combined with efforts to produce expansive learning resources, and to bring forth a community of interest around the project's topic.

We have highlighted some wrinkles in usability, and explored how these have their roots in the interaction between some principles that guided development. In particular, we have tried to balance the accessibility and flexibility of the tools, whilst also making efforts to keep a sense of consistency across three quite distinct target languages, and ensure a basis for continued development once funding has finished.

Whilst this work has taken place in an artistic research setting, with motivations and methods that may be unfamiliar to some readers, we feel that this account points to questions and areas for further work that are possibly interesting to the PPIG community. On one front, it charts the conduct of a moderately-sized software project in the wild that has had to contend with the affordances of a range of languages simultaneously, and reveals some of the dance between pragmatism and principle involved in making this work. Meanwhile, answers to questions about how creative coding musicians might relate to and play with concepts from data science might well be usefully approached in the future by some combination of the release-it-and-see-what-happens approach we take here, and more controlled experimentation.

Acknowledgement

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725899).

6. References

- Bergström, I., & Blackwell, A. F. (2016). The practices of programming. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 190–198). IEEE Computer Society. doi: 10.1109/VLHCC.2016.7739684
- Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., ... Varoquaux, G. (2013). API design for machine learning software: Experiences from the scikit-learn project. In *ECML PKDD workshop: Languages for data mining and machine learning* (pp. 108–122).
- Clarke, S., & Becker, C. (2003). Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In *Psychology of Programming Interest Group (PPIG) 2003*.
- Fiebrink, R. (2019). Machine Learning Education for Artists, Musicians, and Other Creative Practitioners. *ACM Transactions on Computing Education*, 19(4), 1–32. doi: 10.1145/3294008
- Garber, L., Ciccola, T., & Amusatogui, J. C. (2020). AudioStellar, an Open Source Corpus-Based Musical Instrument for Latent Sound Structure Discovery and Sonic Experimentation. In *Proceedings of the ICMC 2020*. Santiago, Chile: Pontificia Universidad Católica de Chile.
- Green, O., Tremblay, P. A., & Roma, G. (2018). Interdisciplinary Research as Musical Experimentation: A case study in musicianly approaches to sound corpora. In *Electroacoustic Studies Network Conference*. Florence, Italy.
- McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4), 61–68. doi: 10.1162/014892602320991383
- McPherson, A., & Tahiroğlu, K. (2020). Idiomatic Patterns and Aesthetic Influence in Computer Music Languages. *Organised Sound*, 25(1), 53–63. doi: 10.1017/S1355771819000463
- Nash, C. (2015). The cognitive dimensions of music notations. In *International Conference on Tech-*

- nologies for Music Notation and Representation (TENOR)* (pp. 191–203).
- Puckette, M. (1997). Pure Data: Another Integrated Computer Music Environment. In *International Computer Music Conference* (pp. 224–227).
- Puckette, M. (2002). Max at Seventeen. *Computer Music Journal*, 26(4), 31–43. doi: 10.1162/014892602320991356
- Roma, G., Green, O., & Tremblay, P. A. (2019). Adaptive Mapping of Sound Collections for Data-driven Musical Interfaces. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 313–318).
- Schnell, N., Röbel, A., Schwarz, D., Peeters, G., Borghesi, R., & Pompidou, I. C. (2009). Mubu & Friends- Assembling Tools for Content Based Real-Time Interactive Audio Processing in Max/Msp. In *ICMC*.
- Schwarz, D., Beller, G., Verbrugghe, B., & Britton, S. (2006). Real-Time Corpus-Based Concatenative Synthesis with CataRT. In *Digital Audio Effects (DAFx)* (pp. 279–282). Montreal, Canada.
- Tremblay, P. A., Roma, G., & Green, O. (2022). Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit. *Computer Music Journal*, 45(2), 9–23. doi: 10.1162/comj_a_00600
- Zicarelli, D. (2002). How I Learned to Love a Program That Does Nothing. *Computer Music Journal*, 26(4), 44–51. doi: 10.1162/014892602320991365

Visual Cues in Compiler Conversations

Alan T. McCabe

Lund University

alan.mccabe@cs.lth.se

Emma Söderberg

Lund University

emma.soderberg@cs.lth.se

Luke Church

University of Cambridge

luke@church.name

Peng Kuang

Lund University

peng.kuang@cs.lth.se

Abstract

When people are conversing, a key non-verbal aspect of communication is the direction in which the participants are looking, as this may convey where each person’s attention is focused. In a programming context, for instance an integrated development environment (IDE), the interaction design frequently directs the programmer’s gaze towards specific locations on-screen. For example, syntax highlighting and error messaging may be used to draw attention towards problematic sections of code. However, error messages frequently direct the user towards the compiler’s point of discovery as opposed to the actual source of an error. Previously we have applied a conversational lens considering the interaction between the programmer and the compiler as a conversation, in this work we refine that into an “attentional lens”. We consider via a prototype and small exploratory user study the difference between where a developer chooses to spend their attention, where the tooling directs it, and how the two might be aligned through the use of visualisation techniques.

1. Introduction

For programmers, the act of programming is a primarily one-way relationship: the programmer writes code, most commonly in an IDE; executes it through a compiler; and receives limited feedback in the form of error messages. These error messages are often obtuse and difficult to read (Beneteau et al., 2019), and may even mislead the reader into looking at the wrong sections of code altogether (Becker et al., 2019; Kats, de Jonge, Nilsson-Nyman, & Visser, 2009). This “feedback” is not only limited in form, but is also delivered in a purely binary format - an error occurs, or it does not.

In our previous work, “Breaking down and making up - a lens for conversing with compilers” (Church, Söderberg, & McCabe, 2021), we explored the consequences of expanding on this interaction to allow for a more complete two-sided relationship between developer and environment. This was achieved by analysing the interaction in the context of a conversation between two participants, inspired by the work of Dubberly & Pangaro (Dubberly & Pangaro, 2009) and Pask (Pask, 1976), thereby applying a “conversational lens” to the activity of programming. For instance, under the conversational lens, the participants of the conversation can be said to be the programmer and their development environment, including IDE, virtual machine, compiler, and various other software components. As stated by Dubberly & Pangaro (Dubberly & Pangaro, 2009), a key aspect of conversation is the mutual construction of meaning and convergence upon agreement - when applied to a programming activity, the meaning of the conversation can be said to be agreed upon when the programmer expects their code to execute in a certain manner, and the compiler performs this execution as desired. This activity helped to highlight areas where programming as an interaction diverged significantly from a familiar human interaction, and instances where the development environment lacks the ability to properly engage in the conversational activity as an equal partner. Based on these findings, a prototype development tool, Progger, was created (McCabe, Söderberg, & Church, 2021), which will be described in more detail in Section 2.

In this paper, we present the second iteration of our exploration of applying a conversational lens to interactions with compiler error messages. In this iteration we narrow our conversational lens to that of *attention*. In a situation where the conversation between programmer and compiler breaks down, the standard response comes in a textual form. By contrast, in normal human interactions other factors come in to play, such as facial expression, gaze, voice, and body language. These can be used to introduce additional information into the interaction, such as a participant’s disposition, or their focus of attention.

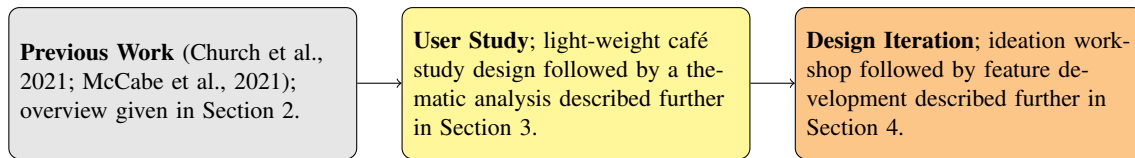


Figure 1 – **Overview** of the work presented in this paper and its disposition.

With this in mind, we present the results of our exploration of attention in the form of visual cues in the interaction with the compiler. Figure 1 gives an overview of the different components of the iteration presented in this paper. We present results of a user study evaluating the Progger prototype (Section 3), the results of a follow-up ideation workshop, building on insights from the user study (Section 4), and finally we end the paper with a discussion of design implications of this work (Section 5).

2. Background and Related Work

In this section, we cover related work connected to the use of attention, either as focus in an empirical study or as part of an intervention (Section 2.1). We also provide a brief summary of our past work on the Progger tool where we applied a conversational lens to the interaction with error messages (Section 2.2).

2.1. Attention in Software Development Tools

Software development is a complex task that requires high cognitive load (CL) (Gonçalves, Farias, da Silva, & Fessler, 2019). It also involves a wide variety of tooling, which further adds to that. One CL-intensive activity during this process is reading and understanding code. This often happens in a development environment, with some assistance from the underlying tool(s). The assistance can manifest in multiple representations, for example, textual (e.g., error messages) and visual (e.g., coloring and alignment). In order to digest such multi-modal information, attention is needed from developers.

Based on the assumption that attention resting on code reflects the visual effort developers take to read and understand it, several studies have focused on code comprehension; e.g., (Busjahn et al., 2015; Storey, 2005); contrasting the behaviours of experienced and novice programmers. Below we selectively elaborate on some work we deem more relevant to this study.

Crosby et al. (Crosby, Scholtz, & Wiedenbeck, 2002) employed eye tracking to examine the roles that *beacons* play among programmers. A beacon could, for instance, be in the form of a comment beacon or a line of code that contains a hint about program functionality. Experienced programmers were more aware of and inclined to make use of beacons in code to facilitate their reading. Novice programmers were less capable of distinguishing between beacons and other areas of code, and thus made little use of them.

Bednarik (Bednarik, 2012) analysed the temporal development of visual attention strategies between novices and experts during debugging in a multi-representational development environment. The study found that experts and novices exhibited similar gaze behaviours in the beginning but diverged in the later phases. Experts were more resourceful with the available information while novices monotonously stuck to one strategy. For challenging bugs, experts more actively related the output to code.

The presented work by Crosby et al. and Bednarik indicate that: 1) efficient utilisation of visual cues elevates code comprehension and debugging, and 2) expert and novice programmers need to be treated differently when designing tools for them; in particular, novices may be those who need help most.

Attention-based interventions have been explored in a couple of studies. For instance, Ahrens et al. (Ahrens, Schneider, & Busch, 2019) visualised developers’ attention in the form of heat maps and coloured class names in Eclipse. In the context of software maintenance tasks, they reported that these two mechanisms provided little aid in orientation and code finding for developers, although the heat map slightly alleviated the cognitive demand. Most developers, especially experienced ones, did not find them helpful. Instead, they found the visualisations to be a distraction from understanding the code

quickly and clearly.

Another example is the work by Cheng et al. (Cheng, Wang, Shen, Chen, & Dey, 2022). They empirically evaluated the usefulness of a tool that captures developers' shared gaze in real time. They found that shared visualisation mechanisms such as gaze cursor, area of interest (AOI) border, grey shading, and connected lines between AOIs helped improve the efficiency of code review. Assisted by these visual features, especially the cursor and border, developers found it easier to identify where their collaborator looked and focused. Based on that, they could adopt either a follow- or separated-strategy to find the bugs more quickly.

These two studies by Ahrens et al. and Cheng et al. demonstrate various ways of approaching attention visualisation in a software development context and the possible granularity of how attention can be visualised. Further, we gather that there appears to be no existing best practises for the time being and the design remains a largely open space.

2.2. The Evolution of Progger

In conversational theory, participants work collaboratively to create a meaningful shared mental model of the on-going conversation (Pask, 1976). Frequently, however, the understanding of the participants becomes divergent for some reason. For example, something may be misheard or misunderstood by one actor, or an incorrect assumption may be made about implicit knowledge (Beneteau et al., 2019). When this occurs, it is said that there is a "breakdown" in the conversation, at which point a meta-conversation must be entered into in order to repair the fault (Dubberly & Pangaro, 2009).

When applying the conversational lens to the activity of programming, the participants become the programmer and their development environment, which may contain several tools such as a compiler, IDE etc. In this context, it can be said that when a program does not perform as expected by the author then a breakdown occurs (Church et al., 2021). When such a breakdown occurs, a common form of feedback to the developer is that of compiler error messages. However, unlike in a natural conversation, the compiler error message is the end of the interaction - if it is not understood by the programmer, there is no mechanism for further exploring the breakdown. At this point the onus of repairing the conversation falls entirely on the human participant.

In an attempt to bridge this gap between programmer and compiler, a research tool was created in the form of a simple web-based Java IDE (McCabe et al., 2021). This tool, Progger, consists of a Dart¹ front-end communicating via a REST API with a Java compilation server. The compilation service contains a small extension to the extendable Java compiler ExtendJ² (Ekman & Hedin, 2007a), which itself is based on the JastAdd³ meta-compilation system (Ekman & Hedin, 2007b). The decision to base the tool on ExtendJ was made for a number of reasons, the most significant being the fact that it is a compiler that makes use of reference attribute grammars (Hedin, 2000). An explanation of this formalism is beyond the scope of this paper, however a detailed description of reference attribute grammars and their significant within the Progger system may be found in our previous work, "Progger: Programming by Errors (Work In Progress)" (McCabe et al., 2021).

By use the tracing system inherent in JastAdd (Söderberg & Hedin, 2010), the evaluation of attributes is tracked by Progger. When a compiler error occurs, this evaluation tree is logged and returned to the front-end for display to the user. This tree is then displayed in the development environment as shown in Figure 2, with the error message augmented by a button with which the user can ask the compiler to "tell me more". As many nodes in the attribute tree are directly related to tokens in the text of the code, this information is used to highlight the relevant code sections as the user mouses over the tree. In this way, the user is able to follow the "thought process" of the compiler as it looked at various parts of the code in an attempt to validate the line at which the error occurred.

¹The Dart programming language, <https://dart.dev/>.

²The Extensible Java Compiler ExtendJ, <https://extendj.org>.

³The meta-compilation system JastAdd, <https://jastadd.org>.

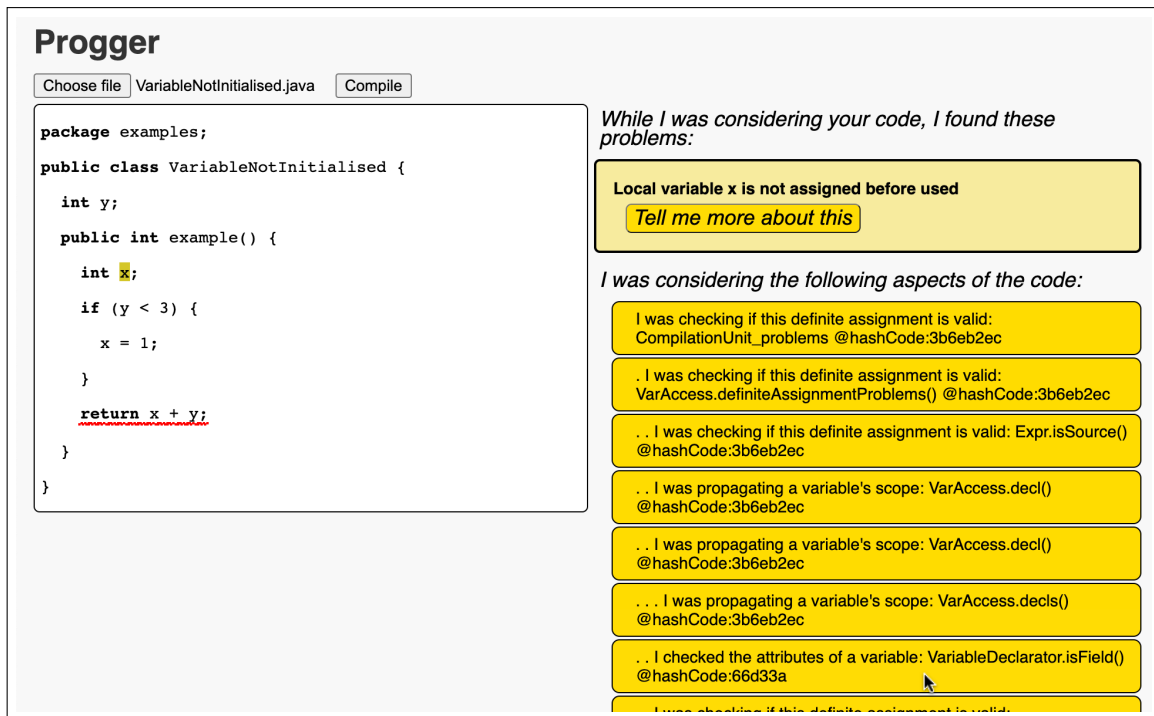


Figure 2 – Screenshot of Progger version 1.0. The left side shows a small Java code snippet with an error in it pointed out with a red squiggly line. The right side shows the error with a "Tell me more about this" button which has been clicked here to expand an attribute trace tree shown in the bottom right.

3. User Study

In initial testing of the Progger research tool, the development team found the results to be of interest. Despite this interest, it was understood that the tool itself worked on a very simple assumption: providing more information to the user is inherently useful. Much of this information, however, came in a form which was only intelligible to someone with a reasonable understanding of the internal workings of RAG-based compilers. In contrast to this, we felt that the target audience who stood to benefit most from a tool like Progger was that of relatively inexperienced programmers. Due to this disconnect, it was decided to undertake a user study in order to determine the usefulness of the tool in its current state when presented to the target audience.

3.1. Study Design

A light-weight exploratory study, here named a “café study”, was undertaken in an effort to obtain initial design feedback for a relatively low time investment. As a general target audience of fairly novice programmers was selected, the café study was conducted on the grounds of Lund University. This took the form of a booth set up in the foyer of the computer science building, as shown in Figure 3, where students were offered the opportunity to complete a short task within Progger in exchange for a lunch coupon.

Students deciding to participate in the study were asked to complete an informed consent form, after which they were presented with a single-class Java program, chosen randomly from a set of 3 pre-defined programs, each of which contained several compiler errors. These programs were selected randomly from a public repository⁴ of solutions to Kattis⁵ programs, with a selection of errors manually inserted into the code by the first author. The errors are representative of a small set of semantic error patterns, selected to provide instances where the prototype was found to be particularly helpful in

⁴Provided with permission by Pedro Contipelli: <https://github.com/PedroContipelli/Kattis>, visited at commit 30884ba

⁵Kattis problem archive: <https://open.kattis.com/>



Figure 3 – The user study booth.

initial testing, and instances where it was not. For example, uninitialised variable error rendered a set of localities that were deemed to be interesting, while missing import statements did not return any locations within the class file. Figure 8 includes an example of a Java code snippet used in the study. Participants were then asked to attempt to fix the errors, to the best of their abilities, while the screen and verbal interactions were recorded. This method yielded a set of 13 recordings over a two day period. Of the set of participants, none of them had industrial programming experience, with most of their exposure to programming coming in an academic context of between 0 and 4 years of higher education. This information was obtained via a follow-up survey distributed by e-mail, which yielded a relatively low response rate. In retrospect, an immediate follow-up survey, to be completed on-site at completion of the study task, would have been a more rigorous method of acquiring this data, a factor that will be taken into consideration in future iterations of the café study methodology.

Ultimately, the study setup was considered to be a success by the authors. Despite the low level of commitment required to set up and conduct the experiment - in the region of hours of total work - the aforementioned assumption, that more information is inherently useful, was effectively challenged by the study findings. These findings will be discussed in detail in the following section.

3.2. Data Analysis

After an initial transcribing exercise, the transcripts were read through independently by the first and second author, with the aim of completing a thematic analysis. This entailed the identification of comments and interactions that were deemed to be of interest. Depending on the content of the highlighted excerpts, a number of codes were coalesced during this process. Once the initial reading and codifying of the transcripts was completed, the first and second author met to compare the annotated transcripts, whereupon areas of overlap were identified and used to inform a combined set of codes, shown as boxes in Figure 4.

Codes represented specific features like highlighting, capturing both positive (e.g., *"It's very good with the highlight system because you know exactly where you want to look initially"*) and negative aspects (e.g., *"It's highlighting everything, it's too much"*), as well as whether the error messages were deemed to be helpful (e.g., *"It's putting human sentences instead of just like error codes and [...] more explained I would say, absolutely more explained"*) or unhelpful (e.g., *"It didn't describe the error very well"*). Another example of a feature represented by a code was the attribute trace tree and, for instance, when it was found to not be helpful (e.g., *"All this text, it doesn't really say anything to me"*). Other codes captured broader aspects such as how beginner friendly the tool was (e.g., *"if you're a beginner programmer and you would get this kind of [...] feedback on your code, it would be very much easier to [...] fix it"*),

Code	Participants												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Beginner friendly	5.5	23.9	25.9				6.4			3.1		7.6	20.1
Helpful High-lighting	20.4			30.3	5.7	17.4	14.6	24.2		14.1	8.7	10.6	11.9
Helpful Message		12.5	4.1	20.8			12.0						7.9
Unhelpful Trace Tree			16.3		4.0		9.1			6.8	19.8		
Unhelpful High-lighting				4.2			3.7	23.2			14.7		
Unhelpful Message	7.5							2.8	4.6				
Prototype Bugs	2.6					10.1				4.0			
Comprehension	42.8		19.7	30.7	44.8	49.3	6.0	22.0	49.1	15.3	8.1	59.1	43.1
Helpful Experience	7.5				12.9				24.1	9.8			
Relation to Debugger									3.3	21.5	17.6	7.6	
Suggestions							26.1	13.1				9.0	
Instructions				5.8			5.3		1.9	5.5			

Table 1 – Overview of occurrence of codes per participant. Colour coding represents the theme which each code ultimately fell into (green for positive, red for negative, and grey for neutral), and intensity in terms of percentage of total words dedicated to each code by a participant (with color intensity increasing with percentage).

or comments made when encountering prototype bugs ("This isn't showing anything").

Furthermore, a number of codes were introduced for the purpose of categorising interactions that were deemed to be less interesting, such as: instances where the interview subject is verbalising their process of comprehension (e.g., "So, I have a couple of 'if' statements, and if none of these are fulfilled I will returned T"); asking the interviewer for instruction (e.g., "Do you want me to recompile it?"); relating Progger to their experience of conventional debuggers (e.g., "Some debuggers are [...] scary because they have an overwhelming amount of functions that you're not really accustomed to [...] while this one [...] generalises it more by giving you the simple fact of 'this is where we think the problem is'"); making suggestions for future improvements (e.g., "One step further could be [...] to make a suggestion how to fix it"); and relating that the tool may become more useful with experience (e.g., "If you [...] get to know this tool I think it becomes easier").

These codes were then applied to two of the interview transcripts in a collaborative exercise involving the first and second authors, in order to come to an agreement upon interpretation. After completing this exercise, the first author applied the combined code set to the rest of the transcripts individually. Certain codes occurred frequently across all participants, with a breakdown presented in Table 1.

Across the codes, various loose themes began to emerge: neutral discussion, positive comments about the tool, and negative comments about the tool. Within the positive and negative themes, two sub-themes, helpful and unhelpful respectively, were also constructed. The final theme map is presented in Figure 4 with themes shown as circles connected to codes in boxes.

3.3. Discussion

Through reading of the transcripts and the thematic analysis exercise, two main take-aways arose: 1) that the trace tree showing the internal workings of the compiler as it traversed the attribute tree was largely deemed to be unhelpful, with no positive comments made regarding it, and 2) that the highlighting of localities within the code was of particular interest to participants - when it worked well it was praised highly, when it did not work well it was criticised as distracting. The prevalence of these sentiments

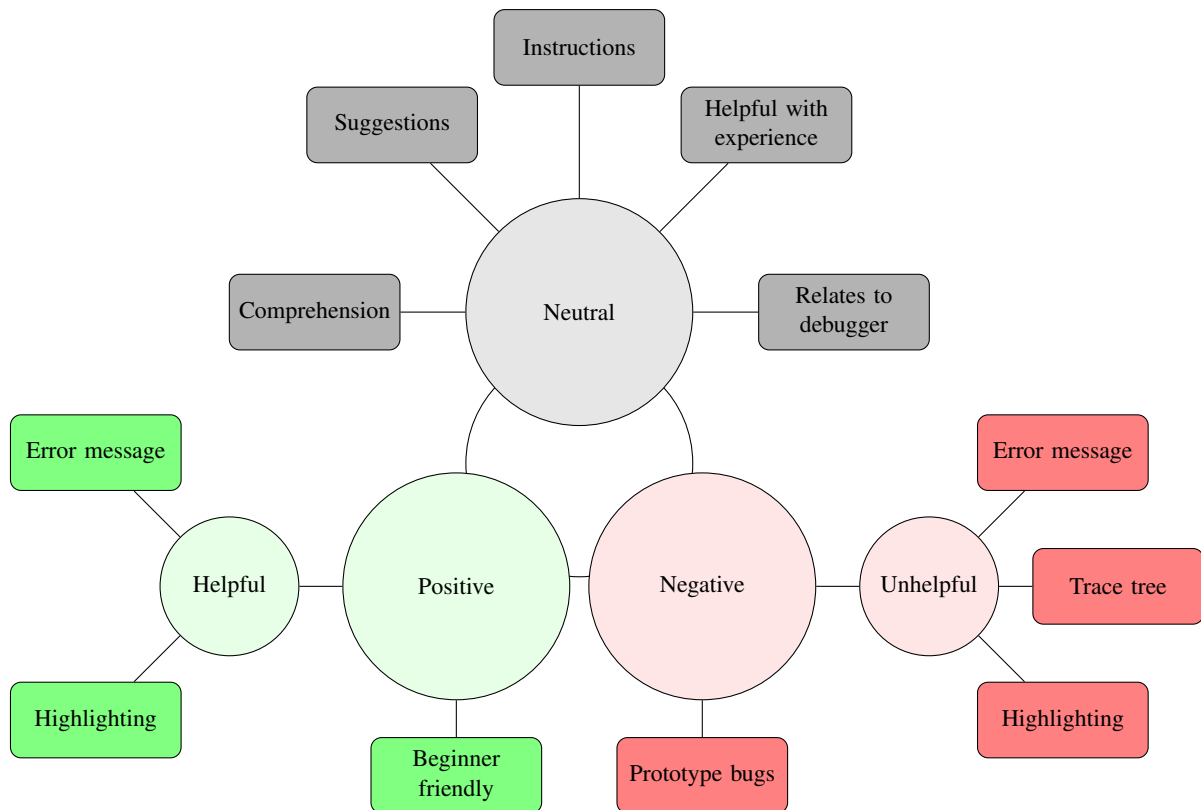


Figure 4 – Identified **themes**, represented by circles, and associated **codes**, represented by boxes.

is lent credence by the incidence of the relevant codes across the transcripts: out of 13 interviews, 11 participants made mention of the highlighting feature in a positive light, with 5 containing a negative comment. Similarly, the trace tree was mentioned in an unfavourable light in 6 out of the 13 transcripts - the highest incidence of any one negative code - while it was not spoken of positively by any participant.

In light of the related work on attention, the positive feedback regarding *highlighting* drew comparisons to the results of the study by Crosby et al. (Crosby et al., 2002). In this study, it was found that experts were more aware, and made more use, of beacons in code, while novices were found to not be as adept at distinguishing beacons from other less-relevant code. We hypothesise that the use of highlighting may help to even out this distance by drawing the attention of novices to areas of the code which may act as beacons for experts.

Regarding the dominantly negative feedback about the attribute trace tree, we did not see anything close to an effect where a participant expressed that they understood the compilers "train of thought" by using the trace tree. How we implemented the feature together with the limited user study may be two reasons for this. We further speculate that the effect we saw here may be related to Norman's gulf of evaluation (Norman, 2013).

The consequences of these findings were discussed and a design ideation workshop scheduled in order to iterate upon the design of the prototype. The starting point for the design iteration was ultimately decided to be a new version where the trace tree was entirely removed, and where *locality* became the primary means of interaction with the user.

4. Design Iteration

In order to operationalise the above data into a design process, we elected to perform a divergent design phase (Cross, 2005; Dubberly, 2004; Pugh, 1981). During this phase we attempted to generate as many broadly differing designs as possible in order to create ideas that we could curate. These were done in the context of the original design, but with the explicit intention of not being literally driven by it.

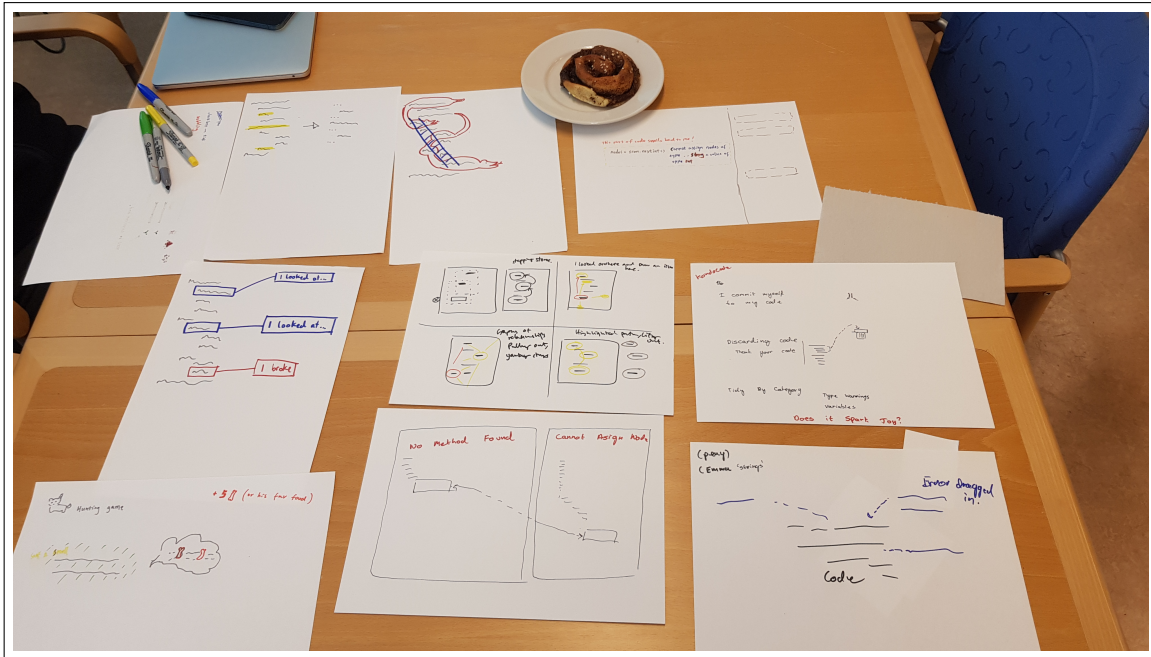


Figure 5 – A selection of design concepts as produced by the ideation workshop.

The design workshop was run in two phases with all the authors creating one series of new designs, then a short presentation where each of the authors described their ideas to each other, followed by another iteration to allow cross-pollination of ideas, followed by a final discussion and wrap up.

The process was successful in generating a wide range of different designs and interaction metaphors, which can be seen in Figure 6.

4.1. Data Analysis

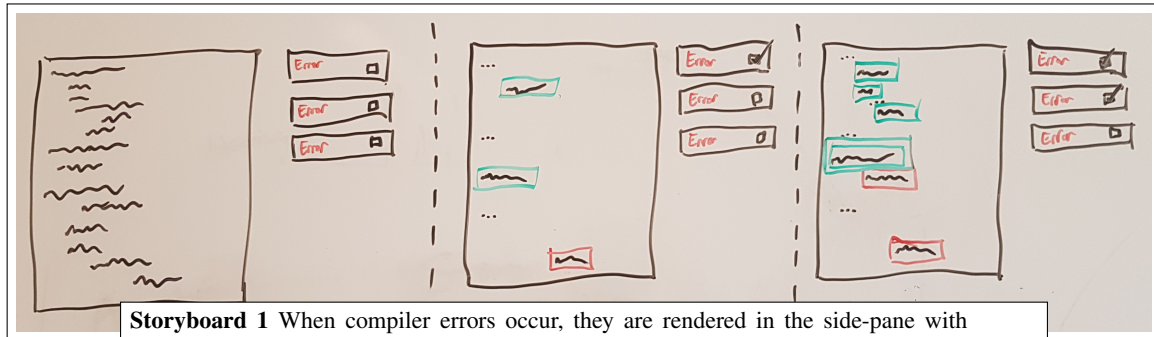
After the main ideation workshop session, a period of time was allowed for the participants to consider the proposed ideas. Ultimately, the first author selected three concepts for further development, based on both novelty and technical feasibility. From these three design concepts, storyboards were drawn up to further illustrate the design interaction. These storyboards are presented in Figure 7.

Following the creation of these storyboards, a final session was held between the first three authors to narrow the selection down to a single final concept for further development. Locality, one of the main takeaways from the user study, was found to be a strong theme in each of the concepts, taking the form of obfuscating non-relevant code in Storyboards 1 and 3 and the physical separation of relevant code from the main body in Storyboard 2. Storyboard 2, however, also introduced a physical relationship between the elements of the code. As described in Figure 7, it was conceptualised that the information collected during the evaluation of an error, specifically the token locations in the code, could be used to introduce a "weight" to each considered element. This "weight" would be based on the number of times the compiler passed over each token - in essence, a compiler "heatmap" - and led to a discussion amongst the authors about what this information might reveal. Ultimately, three theories emerged:

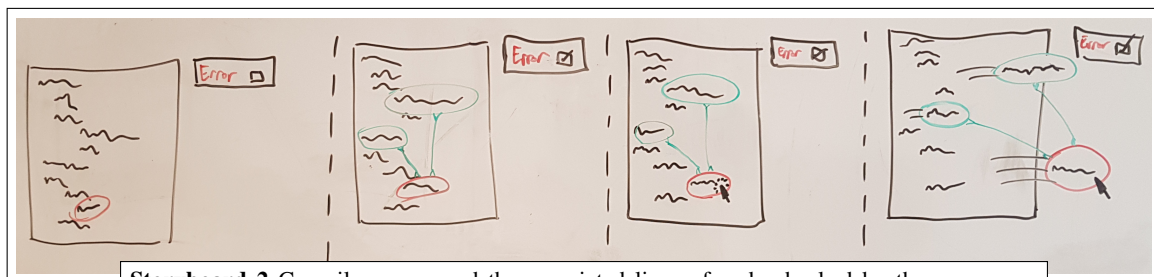
1. Locations that are *frequently* revisited by the compiler may indicate sections of code that are critical to the calculation of an error.
2. Locations that are *less-frequently* visited by the compiler may indicate sections of code that the compiler struggles to parse correctly, leading to it being visited very few times before an error is thrown.
3. Frequency of visitation of the compiler may have no significance when considering the source of an error.



Figure 6 – A collage of the ideas conceptualised during the ideation session.



Storyboard 1 When compiler errors occur, they are rendered in the side-pane with check-boxes. When the box for a particular error message is checked, all code not directly checked by the compiler is obfuscated. Multiple errors may be checked, with lines of common interest highlighted.



Storyboard 2 Compiler errors and the associated lines of code checked by the compiler are highlighted with ellipses, with "elastic bands" connecting them. When the error node is clicked and dragged, the associated code sections are pulled out of the code pane alongside it. The number of times a code fragment is checked by the compiler is used to assign "weight" to the elements, with more frequently checked elements "sticking" to the code pane.



Storyboard 3 A permutation of story board 1. When the error box is checked, a "post-it note" element is added to the display, showing only the code that is directly checked by the compiler. These post-it notes may be moved around the screen at will.

Figure 7 – Storyboards of expanded concepts from the ideation session.

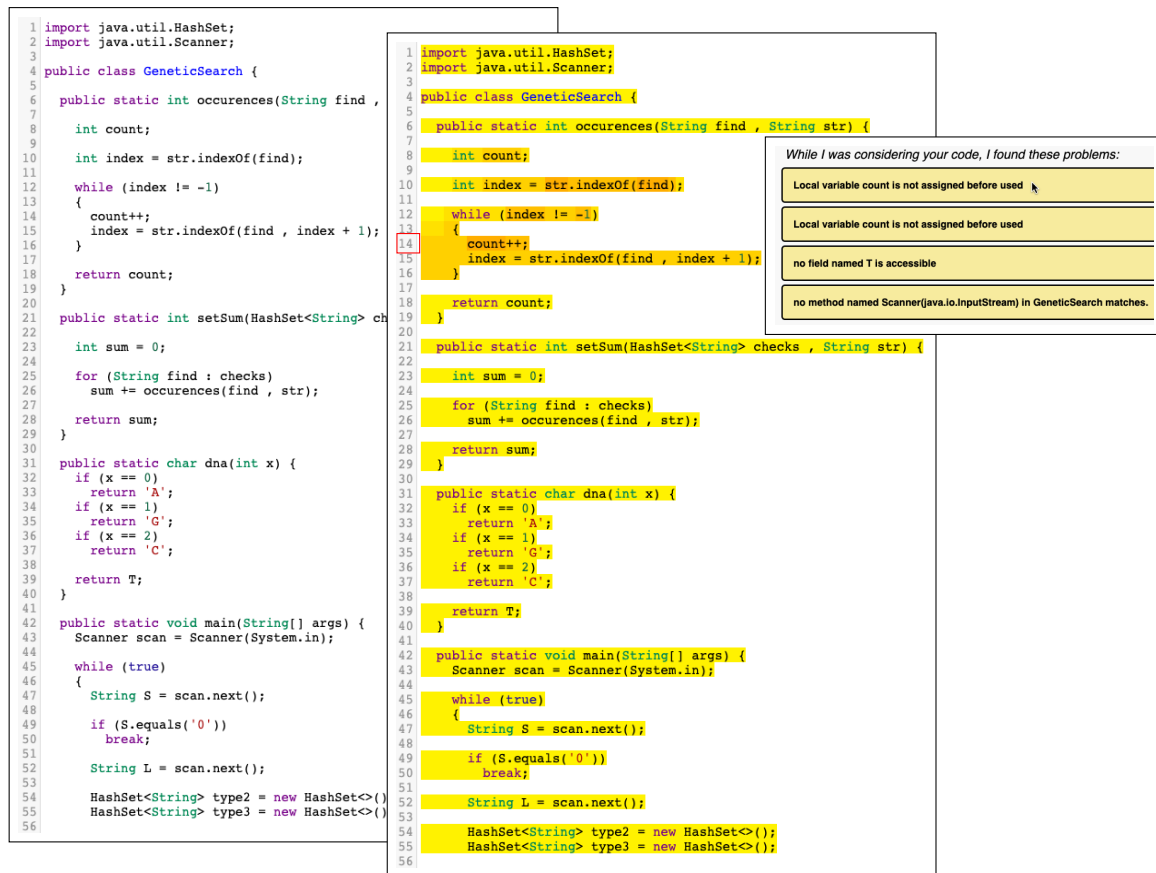


Figure 8 – An example of a heat map generated on a one of the Java code snippet that was presented in the user study. As the mouse pointer is hovering over the first error ("local variable count is not assigned before use" on line 14) the attention of the compiler when finding that error is shown is visualized as a line-based heat map.

As no consensus could be arrived at which of these three theories was most likely to prove correct, it was decided that Storyboard 2 offered the most interesting opportunity for exploration. For this reason, Storyboard 2 was selected for further development.

4.2. Implementation

Building on the previous Progger work, a new version of the prototype was developed. The initial version of Progger, as described in Section 2, uses the JastAdd tracing system to track the evaluation of attributes at the time of an error occurring. From this, an evaluation tree is constructed, with many of the nodes directly related to token locations in the code. As token locality was the primary focus of the most recent iteration, this previous design meant that no further information was required to be extracted from the compiler in order to determine the heatmap.

On reception of the attribute tree from the compiler, Progger v2.0 iterates through the tree and logs each instance of a "location" attribute occurrence. These location attributes come as a range, e.g: 14, 0–15, 12, which is of the format `startLine, startColumn–endLine, endColumn`. From this list of ranges, a map is calculated where the key is a *single location*, e.g. 14, 0, and the value is the number of times this token is visited across all location ranges. From this map, highlighting can be applied to the code pane, with the darkness of the highlighted code calculated from the number of times the token has been visited by the compiler. An example of this is found in Figure 8.

5. Discussion

In this paper, we have presented the results of a user study evaluating the approach implemented in the Progger prototype (McCabe et al., 2021). We used a light-weight café style study, combined with a thematic analysis of transcripts, to gather design input for an ideation workshop. The results from the user study played down the utility of the attribute trace tree (which we had some hope for) while the utility of the companion highlighting was brought to the surface. As a consequence, our focus was geared toward that of attention and the role it plays in the kind of "compiler conversations" we are considering in this work. With input from the sketches generated in the ideation workshop, we explored one design direction focusing on incorporating visual cues into Progger in the form of a "compiler attention heat map" laid out on visual tokens in the source code.

Heatmaps & Attention As mentioned in Section 2, Ahrens et al. (Ahrens et al., 2019) have also explored the use of heat maps but with the goal of visualizing the attention of other developers. They found some issues in their method due to imprecision between the generated heat maps and the mapping to the source code, distorting the locality of the attention, causing some of the experienced programmers in their study to find the visualisation technique distracting. In our explored setup, we consider the "thought-process" of the compiler where the heat map weights are calculated and assigned based on the number of code element visits by the compiler during analysis. We speculate that we would not see the same distortion of attention as when eye-tracking data is mapped to code lines as in the case with the work by Ahrens et al., but we may on the other hand see distortions amounting from the structure of the abstract syntax tree modelling the code.

Collaboration & Attention We find the work by Cheng et al. (Cheng et al., 2022), which explores visualisation of other developers' gaze in a collaborative setting, inspiring. It may be worthwhile to explore a combination of the conversational lens, as we are applying here, in a similar setting. For instance, questions like *"how can conversations within one group help another group?"* or *"how can the compiler's knowledge about experts enhance its communication with novices?"* could be considered. As a possible exploration, we can imagine the compiler as a host that is able to store all programming mistakes made, and visual attention given by, developers. When a new actor enters the environment, the most frequently looked parts of code or the most possible problematic code regions are already marked out. In that sense, there is a historical component to the conversation where past actors remain present in the new conversation.

Programmers' Attention Earlier work on Attention Investment (Blackwell, 2002) within the PPIG community has explored the way in which programmers considered the likely costs and rewards of expenditure of their attention with a notational system. In starting to investigate effective mechanisms by which this attention can be directed, we are seeking to understand how the broad strokes of the attention investment model emerge. This could be helpful in exploring whether or not there are places where this could be done more efficiently - but doing so might also generate information about the details of the Attention Investment framework; for example, does the misdirection of attention play a significance role in the way in which programmers perceive risk and reward?

The design of the Progger system also allows the possibility of integrating analyzers to further direct the programmers' attention. Such analyzers may be used to, for instance, facilitate a conversational-style interaction about considerations such as control flow. This may be explored in future work, however one key distinction to note between analyzers and the compiler is the inherent uncertainty of analysis results. Where a compiler error is indicative of a critical error that prevents the program from being executed, analysis tools are susceptible to false positives, and as such may direct the programmer's attention to an area of code that ultimately does not require fixing. False positives have previously been related by analysis tool users as one of the biggest factors in their low usage statistics, therefore the benefits of

introducing features prone to false positives into the Progger system would need to be weighed carefully against the risks.

Concluding Remarks More widely our results indicate that in the context of programming in the face of errors, it is difficult to build a general conversational bridge between the programmers and compiler authors via the crude medium of error messages. However, whilst error messages can be problematic, the compiler directing the programmers attention to areas of the code, and the programmer being able to ask "*what were you looking at when you did x*" seem to be effective. It feels counter intuitive at first to abandon the richer communicative possibilities of error message text to focus only on the direction of attention, but it may prove a productive route for further exploration. Sometimes in conversations, it seems that less is more, especially when one of the participants (the compiler) does not really know what they are trying to say, and can not empathise effectively with the other.

Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research under Grant No. FFL18-0231 and the Swedish Research Council under Grant No. 2019- 05658.

6. References

- Ahrens, M., Schneider, K., & Busch, M. (2019). Attention in software maintenance: An eye tracking study. , 2-9. doi: 10.1109/EMIP.2019.00009
- Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., ... others (2019). Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the working group reports on innovation and technology in computer science education* (pp. 177–210).
- Bednarik, R. (2012, feb). Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *Int. J. Hum.-Comput. Stud.*, 70(2), 143–155. doi: 10.1016/j.ijhcs.2011.09.003
- Beneteau, E., Richards, O. K., Zhang, M., Kientz, J. A., Yip, J., & Hiniker, A. (2019). Communication breakdowns between families and alexa. In *Proceedings of the 2019 CHI conference on human factors in computing systems* (pp. 1–13).
- Blackwell, A. F. (2002). First steps in programming: A rationale for attention investment models. In *Proceedings ieee 2002 symposia on human centric computing languages and environments* (pp. 2–10).
- Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... Tamm, S. (2015). Eye movements in code reading: Relaxing the linear order. In *2015 ieee 23rd international conference on program comprehension* (p. 255-265). doi: 10.1109/ICPC.2015.36
- Cheng, S., Wang, J., Shen, X., Chen, Y., & Dey, A. (2022, 06). Collaborative eye tracking based code review through real-time shared gaze visualization. *Frontiers of Computer Science*, 16. doi: 10.1007/s11704-020-0422-1
- Church, L., Söderberg, E., & McCabe, A. (2021). Breaking down and making up-a lens for conversing with compilers. In *Psychology of programming interest group annual workshop 2021*.
- Crosby, M., Scholtz, J., & Wiedenbeck, S. (2002, 07). The roles beacons play in comprehension for novice and expert programmers. In *Psychology of programming interest group annual workshop 2002*.
- Cross, N. (2005). *Engineering design methods: strategies for product design*. John Wiley & Sons.
- Dubberly, H. (2004). How do you design. *A compendium of models*, 10.
- Dubberly, H., & Pangaro, P. (2009). What is conversation? how can we design for effective conversation. *Interactions Magazine*, 16(4), 22–28.
- Ekman, T., & Hedin, G. (2007a). The jastadd extensible java compiler. In *Proceedings of the 22nd annual acm sigplan conference on object-oriented programming systems, languages and applications* (pp. 1–18).

- Ekman, T., & Hedin, G. (2007b). The jastadd system—modular extensible compiler construction. *Science of Computer Programming*, 69(1-3), 14–26.
- Gonçales, L., Farias, K., da Silva, B., & Fessler, J. (2019). Measuring the cognitive load of software developers: A systematic mapping study. In *2019 IEEE/ACM 27th international conference on program comprehension (icpc)* (p. 42-52). doi: 10.1109/ICPC.2019.00018
- Hedin, G. (2000). Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 301–317.
- Kats, L. C., de Jonge, M., Nilsson-Nyman, E., & Visser, E. (2009). Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-lr parsing. *ACM SIGPLAN Notices*, 44(10), 445–464.
- McCabe, A. T., Söderberg, E., & Church, L. (2021). Progger: Programming by errors (work in progress). In *Psychology of programming interest group annual workshop 2021*.
- Norman, D. (2013). *The design of everyday things: Revised and expanded edition*. Basic books.
- Pask, G. (1976). Conversation theory. *Applications in Education and Epistemology*.
- Pugh, S. (1981). Concept selection: a method that works. In *Proceedings of the international conference on engineering design* (pp. 497–506).
- Söderberg, E., & Hedin, G. (2010). Automated selective caching for reference attribute grammars. In *International conference on software language engineering* (pp. 2–21).
- Storey, M.-A. (2005). Theories, methods and tools in program comprehension: past, present and future. In *13th international workshop on program comprehension (iwpc'05)* (p. 181-191). doi: 10.1109/WPC.2005.38

Intuition-enhancing GUI for visual programming

Vasile Adrian Rosian

Department of Computer Science
"Babes-Bolyai" University Cluj-Napoca
adrian.rosian@gmail.com

Abstract

While searching for a mechanism for combining free constructs and cofree interpreters with monad-comonad adjunctions for GUIs ((Freeman, 2017) and (Xavier, Da, Bigonha, & Freeman, 2018)) to permit transitioning the state cursor of a GUI application and constructing a DSL using free monads for this state cursor transition that is later on given Haskell semantics through string diagrams (Coecke & Kissinger, 2018) we arrived at the problem of enhancing usability of the solution beyond the advantages given by composability as a means of reducing complexity.

The semantics of string diagrams are limited in the degrees of freedom available to them to enhance the "intuition" of the user because of their 2D structure and lack of semantics for positioning. We explore here similar semantics in a 3D space using the idea that data types are "circles" positioned at some sort of "distance" from a conscious observer that gravitate towards a "center" given by an "input-output" tensor, much like magnetic fields around a coil. The combination gives birth to a torus shape where programs are closed transversal loops that merge into "vortexes" (or "spirals") on the surface of the torus. Enhancement the intuition is then available when giving semantics to size, rotation speeds, traversal speed, color, sound - in a consistent manner - in a VR environment.

1. GUIs with monads and comonads

1.1. State, store, pairing

In category theory an adjunction is a relaxation of the definition of an equivalence of categories. Seen from the perspective of hom-sets an adjunction is a bijection of those sets. Formally:

$$\begin{aligned} F : \mathcal{C} &\rightarrow \mathcal{D}; \mathcal{C}, \mathcal{D} \in Cat \\ U : \mathcal{D} &\rightarrow \mathcal{C} \\ (\eta, \varepsilon) : F &\vdash U \\ \varepsilon : F \circ U &\rightarrow 1_{\mathcal{D}} \\ \eta : 1_{\mathcal{C}} &\rightarrow U \circ F \\ or \\ hom_{\mathcal{C}}(X, UY) &\cong hom_{\mathcal{D}}(FX, Y) \end{aligned}$$

The η and ε natural transformations between the free (F) and forgetful (U) functors are the familiar monadic unit and co-monadic co-unit.

Intuitively, an adjunction specifies to the maximum extent possible that two categories are quasi-identical in structure (with determined transforms). Freeman makes explicit such a comparison between state and the store of interfaces of a program.

As such, transitioning state in a state monad is akin to switching screens in the space of all screens of an application, effectively representing reactive applications through a pairing (Kmett, 2011) of functors that "cancel" each other's contexts.

Xavier et al. makes use of the pairing under the shape of Kmett's *zapWithAdjunction* morphism to express the connection between switching the screens of a comonadic interface and the transitioning of application state through monadic actions.

1.2. Combining GUIs

Xavier et al. gives two examples fo combining GUIs expressed as comonads: a binary-driven sum type for interfaces that are switched in place and the Freeman-inspired Day convolution of two functors to express adjoint UIs. For hierarchical components, however, the solution given by Xavier et al. is still comonad transformers.

A hierarchy of components (like a modal inside a parent window) is given by:

```
data StoreT s w a = StoreT
    (w (s -> a)) s

instance Comonad w
=> Comonad (StoreT s w) where
    extract (StoreT wtrans s)
        = extract wtrans s
    duplicate (StoreT wtrans s) =
        StoreT (extend StoreT wtrans) s
```

A side-by-side joining of two components is represented by a *Day* convolution (Kmett, 2016) :

```
data Day f g a = forall b c .
    Day (f b) (g c) (b -> c -> a)
```

and a pair of interfaces where one replaces the other based on a binary toggle is given by (Xavier et al., 2018):

```
data Sum f g a = Sum Bool (f a) (g a)
```

where the datatype *a* is the type of the value written to the component state once the actions in the interface have been executed.

To be noted that language limitations in Haskell do not allow the compiler to discriminate on types and thus provide a correct interpretation in the case where DSL branches are not complete on the DSL continuation type, which is often the case for complete programming languages (yao Xia, 2019).

A good approach based on the GADT technique is implemented based on Wu and Schrijvers and is available in the Polysemy library <https://github.com/polysemy-research/polysemy>.

1.3. State transition function boundaries

With these primitive combinators one can build a comonadic tree that will pair with a *Co* monad obtained using Kmett:

```
co :: Functor w => (forall r .
    w (a -> r) -> r) -> Co w a
runCo :: Functor w =>
    Co w a -> w (a -> r) -> r
```

If *a* and *r* are state values then the transition function is reduced to a function of modifying state while optionally running some effects.

2. Representing state transforms

String diagrams (Coecke & Kissinger, 2018) are a way of graphically representing strict monoidal categories that ensures that the graphical operations on the wires and boxes representing typed values and morphisms make sense in the underlying category.

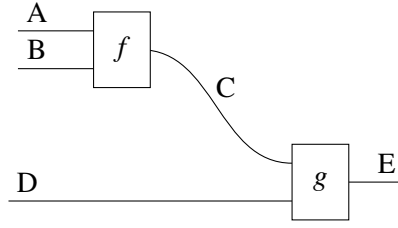


Figure 1 – Wires are typed values, boxes are morphisms

2.1. Standard transforms

In string diagrams values are typed wires and boxes are morphisms.

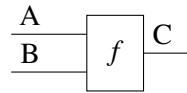


Figure 2 – Morphism $f : A \otimes B \rightarrow C$

Identities are represented by "ghost" boxes on the wires, effectively "not transforming" the value on the wire. In (2) any of the wires might as well be transformed by an identity.

Composition (3) is represented by connecting outputs to inputs and parallel composition represents the monoidal product \otimes .

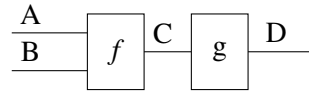


Figure 3 – $g \circ f : A \otimes B \rightarrow D$

2.2. Compositions

As mentioned, simple composition can be performed like in (3) but the output of a box can be connected to part of the input of another box like in (1).

However, we can also split the monoidal product of a box output to provide partial inputs to subsequent functions, like in (4).

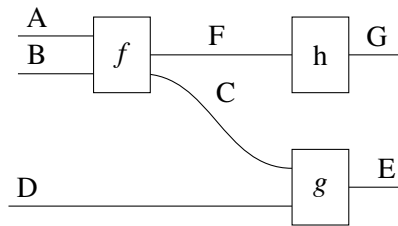


Figure 4 – Monoidal product splitting

Composition helps prove the expressiveness of string diagrams by showing easy equivalences:

2.3. Transform resolution

The graphical language allows collapsing regions of the graph into boxes by mapping input morphisms of the region into inputs to the collapsed box and outputs to box outputs, effectively allowing to "zoom" out of a graphical view while retaining typing information. The approach can be found in the graphical language Enso Luna (Team, 2020).

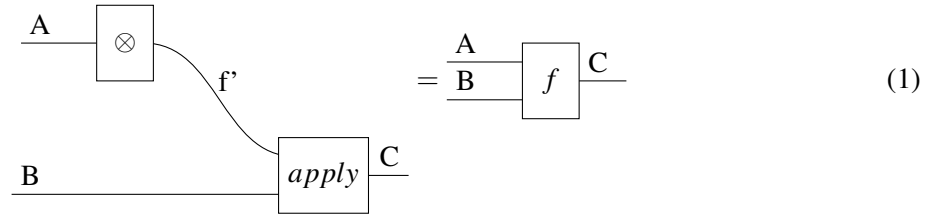


Figure 5 – Morphism $f : A \otimes B \rightarrow C$ expressed as curry-ing

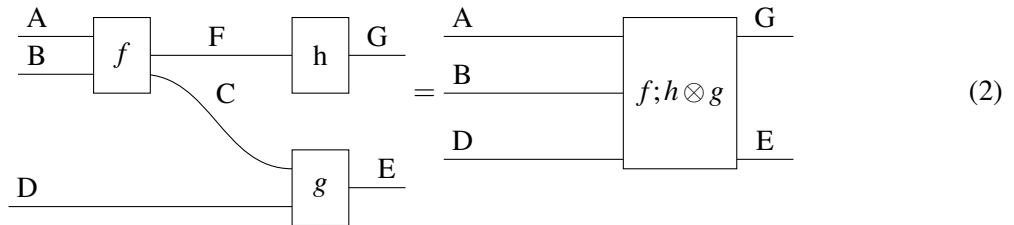


Figure 6 – It is possible to "zoom out" of a diagram

3. Representing effects

Effects in pure functional programming languages such as Haskell are achieved via monads (Moggi, 1991). However, Moggi's definition is not helpful in identifying the right string diagram candidate for the monad representation.

It is Launchbury and Peyton Jones' implementation of the IO monad for Haskell using state threads that points us in the right direction. In the transformer diagram the state thread transformer is represented as taking in the "world" state in addition to normal inputs and returning the modified world alongside regular outputs.

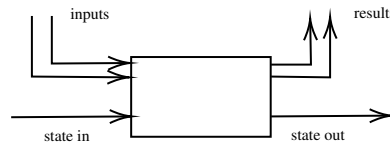


Figure 7 – The ST transformer used as a basis for IO in Haskell

This is consistent with the categorical view of a monad when an adjunction is involved. The ϵ and η natural transformations are morphisms in their respective categories that get their semantics from the functorial relation with the exterior of the category. To be noted that also in this representation we are talking about functors.

The round trip provided by an adjunction provides a nice way of interpreting two functorial relations (the "right" and the "what is left" directions) as one relation in the original category through the means of a natural transformation (a monad is, after all, a monoid for endofunctors - with "endo" being key here). In other words, if category \mathcal{D} is planet Earth and there is some distant, M-class planet discovered called \mathcal{C} , we would be able to describe the lifestyle of the inhabitants of \mathcal{C} using regular actions from the Earth only thanks to the adjunction between the two planets. Regular actions on Earth get new meanings based on the discovery of \mathcal{C} and analogies we can make.

When we talk about some communication (transmission) between Earth and \mathcal{C} we are only interested of either the failure of the transmission or the possible effects on Earth of a successful communication (say we instructed the inhabitants of \mathcal{C} to setup a video feed - we are only interested if we managed to get a new TV channel where we can see the strange new world of \mathcal{C}). Which is to say that an "effect" is only interesting in terms of the results it produces in the originating process.

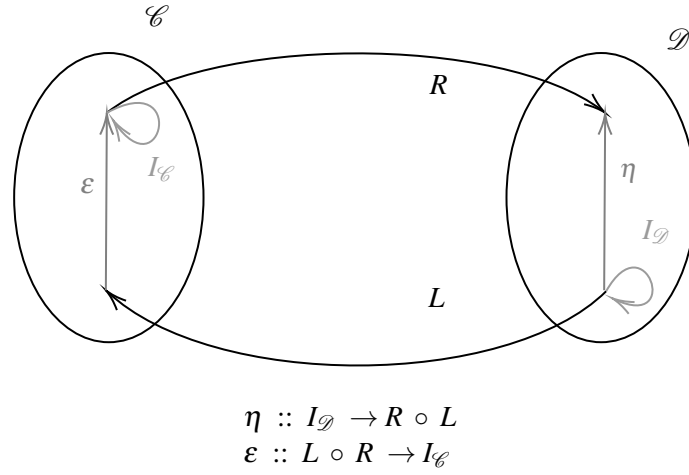


Figure 8 – Monads from adjunctions in category theory

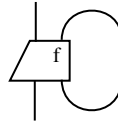


Figure 9 – A monadic effect candidate is the partial trace of a process

As such, the right candidate for monad representation and thus handling of effects in the string diagram language is a partial trace of a process, represented by the process box with the regular part of input-s/outputs and the traced "world" loop that synchronizes the effect on the world with the linked state obtained.

4. Intuition-enhancing representation

4.1. The torus

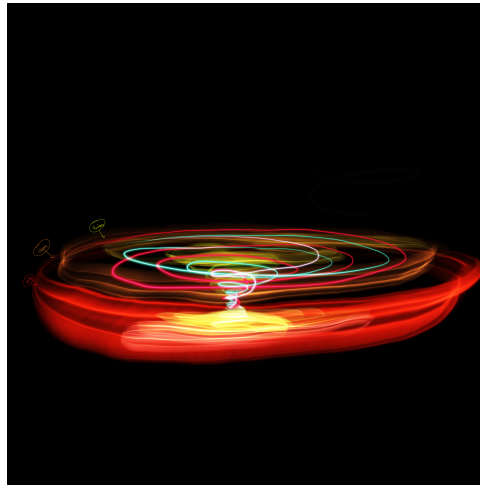


Figure 10 – The program torus attempt at artistic rendering

The semantics of string diagrams are limited in the degrees of freedom available to them to enhance the "intuition" of the user because of their 2D structure and lack of semantics for positioning. We propose similar semantics in a 3D space using the idea that data types are "circles" positioned at some sort of "distance" from a conscious observer that gravitate towards a "center" given by an "input-output" tensor, much like magnetic fields around a coil. The combination gives birth to a torus shape where programs are closed transversal loops that merge into "vortexes" (or "spirals") on the surface of the torus.

Enhancement the intuition is then available when giving semantics to size, rotation speeds, traversal speed, color, sound - in a consistent manner - in a VR environment.

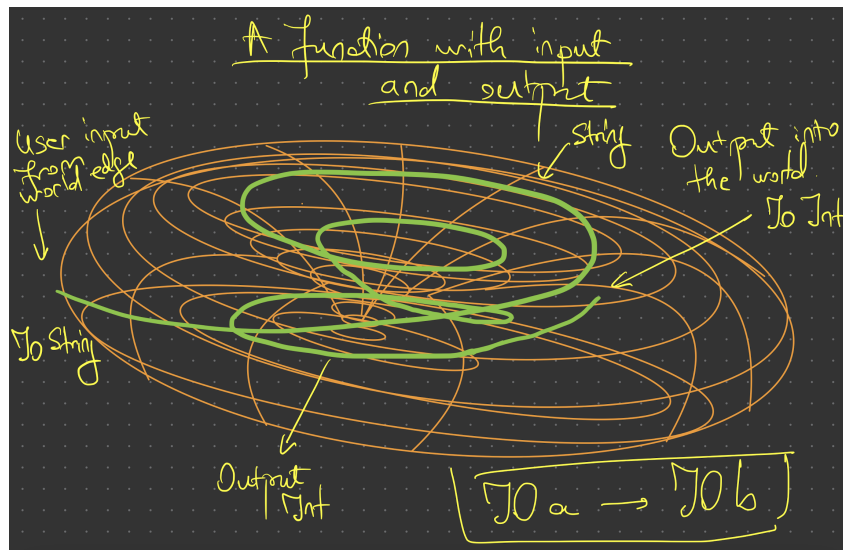


Figure 11 – Function with IO

The semantics of the string diagrams are conserved across the representation, and, even better, asynchronous executions of programs, in terms of crossing the "World edge" are better visualized.

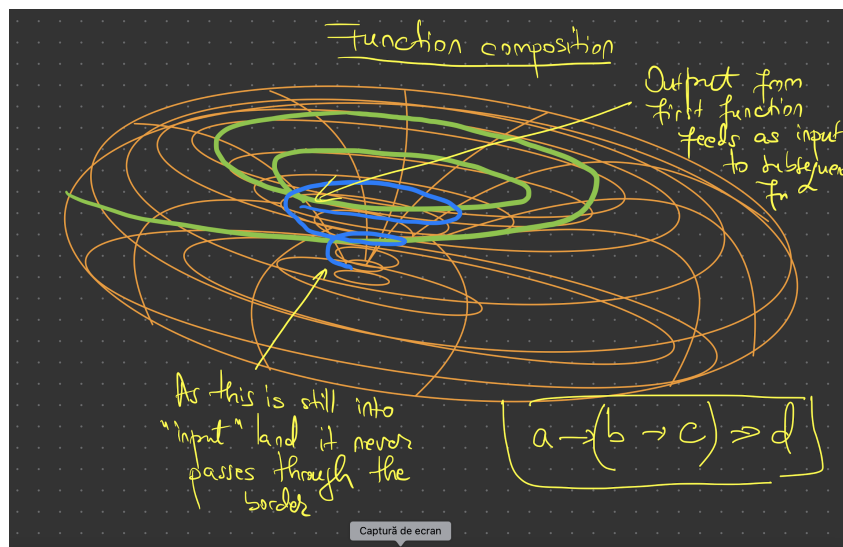


Figure 12 – Function composition

4.2. Visuals

Similar to 3D game worlds, vibration is the key concept used here for conveying and enhancing intuition. When objects in the world vibrate in the same color spectrum they are intuitively semantically close.

As such, the torus is color-coded from exterior "World edge" to interior vortex center to represent basic data types. The colors are conveyed using particle effects (diffuse luminescent smoke) and are presented as "bands" where function "loops" can take place.

Functions are light lines following the surface of the torus. If they start at the world edge they will start as dark red light, otherwise they start as the color of the starting torus band. The light within pulsates at the rate of the starting color. Function lines loop in the band of the data-type where the input/output is. Function lines can join other lines or can split. Joining and splitting modifies both pulse frequency and

color blending.

Pulses of light give the overall impression of spiral rotations which, together with golden-ratio spiral sizes and steps should create a hypnotic effect on the user, stimulating the building instinct.

During construction of the program elements of the construction will either harmonize pulsation and color or will cause electric-like (or light saber-like) sparkles if they do not match in position.

All operations of the VR environment are available, such as resizes, rotations, zoom-in and out, navigation.

4.3. Sound

When it comes to sound there is an intrinsic vibration of the space surrounding the torus that will give the impression, especially around torus world edges, of "generated" light. It is ethereal and omnipresent, keeping the attention and focus on the light torus in front of the user.

Type bands on the torus will also match sound harmonics according to some (hopefully) coherent mapping. Approaching the type band would intensify the sound while distancing oneself from it would reduce the volume of the band.

Function spirals would have matching sounds, but in a higher range and with an added electric buzzing effect. Function dis-harmonies on joins would cause the corresponding electric crackle of a short-circuit.

Sound would be spatial, causing pass-by Doppler effects and would be localized as source for each element of the program.

4.4. Haptics

Normal VR haptics would indicate matching of functions at specific locations (or lack of thereof). Depending on zoom level and position of navigation in the program space, gyroscopic resistance can be factored in for moving spirals around the program space.

5. Future work

The presented work is theoretical and very early-stage, leaving plenty of room for development.

Although intuitively the semantics of the work match string diagrams, equivalence still needs to be demonstrated, especially around partial traces and equivalent matrix transforms. Completeness of the representation is also a field of concern, especially around the four horsemen of IO, Exceptions, Continuations and Randomness. Special attention needs to be paid to bounding mechanics, as outlined by the work of Statebox team (The Statebox Team, 2018).

In the interface part, GUI considerations for the user interface, such as library of components access, layout building mechanics, Day convolution function supply and improvements to intuitive clues still can benefit from a lot of attention.

6. Acknowledgements

A sizable thank you to my coordinating professor, dr. Horia F. Pop for constructive feedback and patience. A special thanks also goes to Radu Ometita for encouraging me to better understand comonads, Claudiu Ceia and George Cosma for taking the time to hear and criticize my musings as well as to Iuliana Rosian for her continuous support.

7. References

- Coecke, B., & Kissinger, A. (2018). Picturing quantum processes: A first course on quantum theory and diagrammatic reasoning. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)*. doi: 10.1007/978-3-319-91376-6_6
- Freeman, P. (2017). *Declarative UIs are the Future — And the Future is Comonadic!* Retrieved from <https://functorial.com/the-future-is-comonadic/main.pdf>
- Kmett, E. (2011). *Monads from comonads*. Retrieved from <http://comonad.com/reader/>

- 2011/monads-from-comonads/
- Kmett, E. (2016). *Kan extensions package*. Retrieved from <https://hackage.haskell.org/package/kan-extensions-5.0.1>
- Launchbury, J., & Peyton Jones, S. L. (1994, June). Lazy functional state threads. *SIGPLAN Not.*, 29(6), 24–35. Retrieved from <https://doi.org/10.1145/773473.178246> doi: 10.1145/773473.178246
- Moggi, E. (1991, July). Notions of computation and monads. *Inf. Comput.*, 93(1), 55–92. Retrieved from [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4) doi: 10.1016/0890-5401(91)90052-4
- Team, T. E. (2020). *Enso: The Syntax*. <https://github.com/luna/enso/blob/master/doc/syntax/specification/syntax.md>. ([Online; accessed 15-May-2020])
- The Statebox Team. (2018). *The Mathematical Specification of the Statebox Language* (Tech. Rep.). Retrieved from <https://statebox.io>
- Wu, N., & Schrijvers, T. (2015). Fusion for free efficient algebraic effect handlers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9129, 302–322. doi: 10.1007/978-3-319-19797-5_15
- Xavier, A., Da, R., Bigonha, S., & Freeman, P. (2018). *A Real-World Application with a Comonadic User Interface*. Retrieved from <https://arthurxavierx.github.io/RealWorldAppComonadicUI.pdf>
- yao Xia, L. (2019). *Free monads of free monads*. <https://blog.poisson.chat/posts/2019-06-09-free-monads-free-monads.html>. ([Online; accessed 14-May-2020])

Pilot Study: Validation of Stimuli for Studying Mental Representations Formed by Parallel Programmers During Parallel Program Comprehension

Leah Bidlake

Eric Aubanel

Daniel Voyer

Faculty of Computer Science, Faculty of Computer Science, Department of Psychology

University of New Brunswick

leah.bidlake@unb.ca, aubanel@unb.ca, voyer@unb.ca

Abstract

Research on mental representations formed by programmers during program comprehension has not yet been applied to parallel programming. The goals of the pilot study were to validate a stimulus set, consisting of 80 programs written in C using OpenMP 4.0 directives, that will be used in subsequent studies on mental representations formed by expert parallel programmers and to serve as a resource for researchers who want to replicate or expand the research on program comprehension to include the parallel programming paradigm. The task used to stimulate the comprehension process was determining the presence of data races. Responses to the data race question were analyzed to determine the validity of the stimuli.

The results of the pilot study indicate that the level of difficulty of the stimuli (accuracy rate of .65) and the time limit for exposure to the stimuli are both appropriate and do not need to be adjusted for the main study. Participants' self-perceived level of expertise correlated with their accuracy indicating this is a reasonable measure of expertise. Given the disparity of responses when asking participants what cues or program components they used to determine whether or not there was a data race, the main study will also include specific questions about components of the code to determine the type of information that is included in their mental representations.

1. Introduction

During the comprehension process, programmers form mental representations of the code they are working with (Détienne, 2001). Understanding these representations is important for developing programming languages and tools that enhance and assist programmers in the comprehension process and other tasks. The cognitive component of program comprehension that is of interest here is the abstract mental representations that are formed during program comprehension. These mental representations, often referred to as mental models, are founded in the theories of text comprehension (Pennington, 1987a). The mental model approach to program comprehension is based on the propositional or text-based model and the situation model that were first developed to describe text comprehension (Détienne, 2001).

Parallel programming has introduced new challenges including bugs that are hard to detect, making it difficult for programmers to verify correctness of code. For example, data races are a type of bug that can occur only in parallel programming and their detection often require close consideration of the code. Data races occur when multiple threads of execution access the same memory location without controlling the order of the accesses and at least one of the memory accesses is a write (Liao, Lin, Asplund, Schordan, & Karlin, 2017). Depending on the order of the accesses, some threads may read the memory location before the write and others may read the memory location after the write, which can lead to unpredictable results and incorrect program execution. Data races are difficult to detect and verify as they will not appear every time that the program is executed. To detect data races, programmers must understand how a program executes in parallel on the machine and the memory model of the programming language.

In parallel programming, there is a significant lack of theory to inform the development of programming languages, instructional practices, and tools (Mattson & Wrinn, 2008). Empirical research on

mental representations formed by programmers during program comprehension has been predominately conducted using sequential code. The comprehension of parallel code requires programmers to mentally execute multiple timelines that are occurring in parallel at the machine level. Therefore, parallel program comprehension may require additional dimensions to construct a mental representation.

2. Background

Empirical research on program comprehension has a direct impact on the development of programming languages and tools. For example, tools have been developed to assist programmers with maintaining, debugging, and documenting code using principles of program comprehension. For instance, Boshernitsan, Graham, and Hearst (2007) developed iXj, a tool that uses a visual language to allow programmers to specify and execute code changes. The design of iXj was guided by the Cognitive Dimensions framework developed by T. R. G. Green (1989) to provide programmers with visual representations that reflect their own mental model of the source code. Empirical research on programming knowledge and plans was used by Tubaishat (2001) to develop a theoretical model, Conceptual Model for Software Fault Localization (CMSFL). The CMSFL model was then used as the basis for developing the BUG-DOCTOR, an Automated Assistant Fault Localization (AASFL) tool that assists programmers with software fault localization. Another example is a tool developed by Arab (1992) for formatting and documenting Pascal programs to assist programmers to write more readable and easier to understand programs. The development of this tool was influenced by empirical research that identified formatting and documenting as important factors in program comprehension.

In terms of task parameters, program comprehension studies have used a wide variety of code constructs ranging from code snippets with as few as eight lines of code (Gilmore & Green, 1988) to complete industrial code (von Mayrhauser & Vans, 1998). There are a number of studies that have used programs with fewer than 20 lines of code (Davies, 1990; Furman, 1998; Gilmore & Green, 1988; Soloway & Ehrlich, 1984), and between 20-30 lines of code (Barfield, 1997; Bateson, Alexander, & Murphy, 1987; Bergantz & Hassell, 1991; Pennington, 1987b; Ramalingam, LaBelle, & Wiedenbeck, 2004; Shargabi, Aljunid, Annamalai, & Zin, 2020; Teasley, 1994; Wiedenbeck & Ramalingam, 1999). Time limits ranging from 60 to 120 seconds have been used in program comprehension experiments with programs that range between eight and 28 lines of code (Gilmore & Green, 1988; Pennington, 1987b; Ramalingam et al., 2004; Teasley, 1994; Wiedenbeck & Ramalingam, 1999). Research in program comprehension has also been conducted using larger programs and software systems that more accurately reflect real life situations involving thousands of lines of code (Abbes, Khomh, Guéhéneuc, & Antoniol, 2011; Bavota et al., 2013; Nosál' & Porubán, 2015). These studies tend to use very few stimuli and often involve a small number of participants providing little power.

3. Research Goals

To date, no empirical research on program comprehension or mental representations of parallel programmers has been conducted (Bidlake, Aubanel, & Voyer, 2020). Because of the considerable differences between parallel and sequential programming, it is impossible to determine if the findings of the empirical research using sequential code would resemble the comprehension process and mental representations of parallel programmers. To inform the development of tools and languages that specifically support parallel programmers, it is important to analyze the mental representations formed by parallel programmers during program comprehension. Therefore, empirical research on program comprehension needs to be expanded to include parallel programming. The lack of research in this programming paradigm also means that there are no existing data sets or stimuli to draw from. The research goal of the pilot study was to validate the stimulus set we created as producing a reasonable level of difficulty.

The long term research goal is to develop a model for parallel program comprehension that is based on the abstract mental representations formed by parallel programmers during program comprehension. Our future studies to investigate these models will make use of the stimuli developed here. The first step in realizing this goal will be to conduct a study to investigate the progression of mental models formed by programmers during instruction on parallel programming.

The stimuli we developed would be relevant for replication studies and incremental research that builds on previous work in the psychology of programming field. The stimuli would also be useful for those who want to expand the research on program comprehension to include the parallel programming paradigm.

4. Method

We conducted an online pilot study with eight participants. The results of the pilot study were analyzed to determine if any of the parameters need to be adjusted for the main study including exposure duration and difficulty of stimuli.

4.1. Participants

Participants had to have experience programming in C and using OpenMP 4.0 directives to implement parallelization. To recruit participants, university instructors emailed the advertisement to students who had completed their parallel programming course and colleagues that would have the appropriate background to complete the study. In the end, a final sample of eight participants completed the experiment. Five participants were professionals and three participants were students. The mean age of participants was 30 years. Their mean amount of programming experience was 8 years. Participants could choose to receive a \$10 e-gift card as an incentive. Participants were informed in the consent form that the incentive is only available in select countries. Participation was voluntary and the protocol was approved by the research ethics board at UNB.

4.2. Materials

The programs from the DataRaceBench 1.2.0 benchmark suite (Liao et al., 2017) were used as inspiration for the programs written by the first two authors, who are computer science instructors. The stimuli were all programs written in C using OpenMP 4.0 directives with no comments or documentation. The selection of OpenMP directives for the stimuli was based largely on the set of directives referred to as The Common Core (Mattson, Koniges, He, & Chapman, 2018).

Research in program comprehension has shown that programmers possess programming plan knowledge consisting of typical solutions to problems, such as searching for a value in a data structure (Soloway & Ehrlich, 1984). The programs for the stimuli were written using unplan-like code so that participants would have to construct their mental representation without relying on prior plan knowledge. We selected the task of detecting data races as it requires programmers to mentally execute the code at the machine level and consider how the execution occurs in multiple timelines in parallel. This additional layer of complexity that requires considering multiple timelines of execution and how they interact will likely result in mental representations that differ from the program and situation models formed during program comprehension of sequential code.

There were multiple considerations when determining the number of stimuli and the length of each stimulus. Given that we are sampling a very specific population of programmers that must have experience using OpenMP directives in C, we did not expect to be able to recruit a large number of participants for the main study. To ensure adequate power with a potentially small number of participants, we had to have a large number of stimuli to meet the minimum recommendation proposed by Brysbaert and Stevens (2018). We also wanted to make sure that the time to complete the experiment was approximately one hour so that it would be a reasonable request for busy professionals. Another consideration was that our stimuli would be used in future experiments using an eye tracking device. To reduce the complexity of eye tracking, we wanted to ensure that the stimuli would fit on a single screen. Given the extensive research that has been done in program comprehension using small scale programs, between eight and 30 lines of code (see section 2), we developed stimuli that ranged between 17 and 24 lines of code to meet the aforementioned needs of our study. We also wanted to use a variety of OpenMP directives including those from the common core to account for differences in participant background knowledge (i.e.: participants may be more familiar with some directives and not others). Using 13 different directives and writing multiple programs using each directive with an equal number of programs with and without a data race, we were able to create 80 unique programs all containing a parallel region.

We felt this was an adequate number of stimuli to draw from for our pilot and main study so that if it was found that some of the stimuli were too difficult or if participants performed poorly on particular directives, we would still have a substantial data set for future research.

The purpose of a time limit for exposure was to determine a reasonable amount of time for participants to complete the task without giving them additional time to read the code for other purposes. To study the mental representations of participants, specific questions about components of the code can be used to determine the type of information they have in working memory and would likely be part of their mental model. Pennington (1986) found that the task (e.g.: study, modification) influenced programmers' mental models. Our concern is that once participants are asked questions pertaining to specific parts of the code that may not be part of their model they may then use additional time to study the code in order to prepare for these questions, creating a practice effect. The time limits used in program comprehension studies, with both novice and expert participants, have ranged between 60 and 120 seconds using programs of similar length to ours (see section 2). Given that our target population is experts, we selected a time limit of 60 seconds.

To reduce the mental strain of tracing code, variable names used in the stimuli match typical programming conventions (e.g.: variables *i*, *j*, and *k* are used for loop counters) (Beniamini, Gingichashvili, Orbach, & Feitelson, 2017) and the variable names were consistent between stimuli to reduce the mental load (e.g.: variables used for arrays were *a*, *b*, and *c*, the variables used for the size of the arrays were *n* and *m*).

Four of the stimuli were used as practice and one practice stimulus was followed by the question asking the participant what cues or components they used to determine whether or not there was a data race. The practice allowed participants to familiarize themselves with all aspects of the interface (e.g.: use of the visual analogue scale and entering text) and the ratio of stimuli followed by the question in the practice (25%) is similar to that of the experiment (26%). For the 76 stimuli used for the experiment, 38 of the stimuli contained a data race and 38 of the stimuli did not (see Figure 1). The length of the stimuli was measured by the number of lines of code. Although we recognize that the lines of code metric does not necessarily reflect the complexity of the code, especially in parallel programming, it is a metric that is commonly used for just that (Bhatia & Malhotra, 2014). Therefore, it is reasonable to expect that programmers would also initially judge the complexity of the programs based on their length. In order to mitigate this we made our programs similar in length. When counting the lines of code we only excluded blank lines, and therefore included all lines that contained any text or symbols. The length of the stimuli with a data race (mean = 20.95, SD = 1.52) and the length of the stimuli without a data race (mean = 20.95, SD = 1.29) did not differ significantly from each other ($p = 1$).

4.3. Procedure

Participants completed the tasks online. The experiment was developed using PsychoPy 3 (Peirce et al., 2019), an open source software package, and Pavlovica was used to host the experiment online. Qualtrics was used to administer the consent form at the beginning of the experiment, the questionnaire at the end of the experiment, and to collect participants' emails if they chose to receive an incentive.

In the advertisement for the study a link to the consent form was provided. The consent form described the background required, the tasks, questionnaire, and compensation (including the list of countries where compensation is not available). After consent, participants were redirected to the experimental portion of the study. The experiment began with a set of instructions asking the participant to determine as quickly and accurately as possible if each program contained a data race and respond by pressing the 'y' or 'n' key on their keyboard. The participants were instructed that the first four stimuli were practice and they would not be included in the results of the study. For each stimulus, participants were given up to 60 seconds to view the stimulus and respond; if they exceeded the time limit exposure to the stimulus ended and they were asked to decide if the stimulus contained a data race or not. The practice set contained two stimuli with a data race and two without, and one stimulus was followed by a question asking participants what cues or program components they used to determine whether or not there was

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]){

    int n = 10;
    int a[n], i, x = 0;

    #pragma omp parallel firstprivate(x)
    {
        #pragma omp for
        for (i = 0; i < n; i++){
            if(i%2 == 0){
                x = x + n;
            }
            a[i] = x + i;
        }

        #pragma omp for
        for(i = 0; i < n; i++){
            a[i] = a[i/2] + x;
        }

        printf("%d\n", x);
    }

    return 0;
}

```

(a) Stimuli containing a data race.

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]){

    int n = 10;
    int a[n], i, z = 2;

    #pragma omp parallel
    {
        #pragma omp for
        for(i = 0; i < n; i++){
            a[i] = z + i;
        }

        #pragma omp sections
        {
            #pragma omp section
            a[n-1] = a[n-1] * 3;

            #pragma omp section
            z = z + 10;
        }

        printf("%d %d\n", z, a[n-1]);
    }

    return 0;
}

```

(b) Stimuli that does not contain a data race.

Figure 1 – Sample stimuli.

a data race. After the practice set, the instructions were repeated, and the 76 experimental stimuli were presented to the participants in random order. Participants were asked after the data race question to rate their level of confidence in their answer using a visual analogue scale that ranged from “Not Confident” to “Very Confident”. For 20 of the stimuli, 10 with a data race and 10 without, participants were asked what cues or program components they used to determine whether or not there was a data race. The data collected from the experiment consisted of the correctness of their response, response time, level of confidence in their response, and the answer describing the program components they used to determine whether or not there was a data race. Correctness of their response refers to whether participants submit a correct answer to the task of identifying the presence of a data race. After completing the experiment portion, participants were redirected to the questionnaire documenting their level of education, programming experience, their perceived level of programming expertise, and age (Feigenspan, Kastner, Liebig, Apel, & Hanenberg, 2012). The questionnaire also solicited feedback on the experiment. Participants were then asked if they would like to receive the e-gift card and, lastly, were redirected to the debriefing.

4.4. Results

For each trial, the accuracy (1 = correct, 0 = incorrect), level of confidence (0 = not confident to 100 = very confident), response time, and whether each trial had a race condition (y) or no race condition (n) were recorded and analyzed for all eight participants in the study using the statistical software program R (R Core Team, 2021). Experiments that were aborted in Pavlovia and consent forms that had no corresponding experimental results from Pavlovia were discarded.

The mean accuracy was .65 (SD = .48) and the mean response time was 24.20 seconds (SD = 17.58). The participants with the greatest mean accuracy (mean = .96 and mean = .79) also had the longest mean response times (mean = 46.57 and mean = 42.16), whereas the participant with the lowest mean accuracy (mean = .46) had the shortest mean response time (mean = 5.65). A one sample t-test was performed to test the null hypothesis that the sample accuracy was equal to chance ($\mu = .50$) with a 95% confidence interval. The results indicated that the accuracy of participants was significantly higher than chance, $t(7) = 2.73, p < .05$.

The data were analyzed using a mixed linear model that was fitted with the **lme4** package (Bates, Mäch-

Table 1 – Spearman Correlation Coefficient Between Accuracy, Confidence, and the Measures of Self-Perceived Expertise

	Accuracy
	r
Confidence	.47
Self-estimation of expertise in programming	.45
Self-estimation of expertise in parallel programming	.59
Self-estimation of expertise in programming compared to their peers	.48
Self-estimation of expertise in parallel programming compared to their peers	.37

ler, Bolker, & Walker, 2015) in R. The design used confidence as a continuous predictor, race condition as a repeated measures factor, and accuracy as the dependent variable. Using generalized linear mixed model with the **glmer** procedure from the **lme4** package, we first determined the best fitting model by comparing the fixed slope model and the random slope model. The results of comparing the models show that the random slope model improved fit significantly ($p < .001$). We also compared the random slope model with participant as the random factor to the random slope model with participant and stimuli as random factors. The results of comparing the models show that the random slope model using both participant and stimuli as random factors improved fit significantly ($p < .05$). Likelihood ratio values were then obtained with the **Anova** procedure from the **car** package (Fox & Weisberg, 2019), using the best fitting model with participant and stimuli as random factors with participants treated as random over the intercept, race condition as the random slope component, and accuracy as the dependent variable for confidence and race condition. Results did not show a significant effect of confidence, $LR = 3.35$, $p = .067$ or race condition, $LR = 1.57$, $p = .211$.

Spearman rank correlations were computed to examine the relationship between accuracy, confidence, and the measures of self-perceived expertise. The strength of the correlations was determined using the scale for r values: $r = .1$ is weak, $r = .3$ is moderate, $r \geq .5$ is strong (Cohen, 1988). The correlation coefficients are listed in Table 1. There were moderate positive correlations between accuracy and confidence, accuracy and self-estimation of expertise in programming, accuracy and self-estimation of expertise in programming compared to their peers, and accuracy and self-estimation of expertise in parallel programming compared to their peers. There was a strong positive correlation between accuracy and self-estimation of expertise in parallel programming.

The participants with the highest accuracy (mean = .96, .79, .63) provided responses for all 20 stimuli that asked what cues or program components they used to determine whether or not there was a data race. These participants wrote more detailed responses that included specific references to OpenMP directives from the stimuli, and either specific variables, data structures, or programming structures in the stimuli or descriptions of concurrency issues (data sharing, concurrent regions). The participants with lower accuracy did not provide responses to all the questions pertaining to elements relevant to a data race; two of these participants did not provide any responses. The participants with lower accuracy provided shorter, less detailed responses, and some responses were unrelated to the question.

The participant with a mean accuracy of .96 responded within the time limit for 71% of the stimuli. This participant had three incorrect responses, none of which were responses given after the time limit expired. The participant with a mean accuracy of .79 responded within the time limit for 76% of the stimuli. This participant had 16 incorrect responses, and only 5 were given after the time limit expired.

4.5. Discussion

The result of the t-test showing that the accuracy of participants is significantly higher than chance suggests that the level of difficulty of the stimuli is appropriate. The mean response times for all participants were within the 60 second exposure limit implying that the time exposure limit was appropriate.

Although the mean number of lines of code for the stimuli is 20.95 this also includes sequential code (e.g.: declaring and initializing variables, printing), lines containing only an opening or closing brace, and pre-processor directives (i.e.: `#include`). Given that the task is to determine the presence of data races, it is likely that the high performing participants were able to respond with high accuracy before the time limit expired since the lines of code within parallel regions would be of most importance to the task. Therefore, we hypothesize that participants focused on only select lines of code they felt were critical to the task. For example, in Figure 1a, the lines of code including the directive for the parallel region that contains more than just an opening or closing brace is 10, compared to the total 22 lines of code.

Although the correlations between accuracy, confidence, and self-perceived expertise are not statistically significant it could be attributed to the lack of statistical power given the small sample size. Despite the lack of power, the moderate to strong correlations between accuracy and measures of self-perceived expertise imply that these measures are relevant and worth preserving.

Confidence based assessment which combines accuracy and confidence levels provides four regions of knowledge: uninformed (wrong answer with low confidence), doubt (correct answer with low confidence), misinformed (wrong answer with high confidence), and, mastery (correct answer with high confidence) (Maqsood & Ceravolo, 2018). The moderate positive correlation between confidence and accuracy indicates that higher performing participants had greater confidence in their answers and therefore had higher levels of mastery of the programming language and parallelization directives and were less likely guessing. Lower performing participants tended to have lower confidence which would indicate they lacked knowledge of either the programming language, the parallelization directives, or both, and may have employed more guessing. This indicates that the higher performing participants were from our target population (expert parallel programmers) and that for this target population the level of difficulty and time limit for exposure are appropriate.

4.6. Threats to Validity

In the development of the stimuli, bias may have been introduced due to our background knowledge and familiarity with particular OpenMP directives. To reduce this bias, the majority of the directives used in the stimuli were selected from the common core, considered by Mattson et al. (2018) to be the most prevalent directives used by programmers. The types of errors that we introduced to create data races may have also been biased towards the types of mistakes we most commonly make ourselves. To reduce this bias, we used some of the same types of errors that were found in the DataRaceBench benchmark suite. Bias that may have been introduced by participants would be their prior experiences with data races that may have caused them to look for mistakes they commonly make. Participants may also have more familiarity with some directives than others. Another threat to validity is our lack of control over the experiment environment. Because the study was conducted online, participants may have been in a very distracting environment with other people around them and access to their personal devices.

The small sample size we ended up with was likely the biggest limitation of our work as it greatly reduced statistical power. In an effort to recruit participants, we contacted colleagues at other institutions that shared the advertisement with their colleagues and students who would have the appropriate background. We also emailed the advertisement to past and current students who had studied parallel programming at our institution and to contacts working in the area of high performance computing. A power analysis was performed on the pilot study using the **powerSim** procedure from the **simr** package (P. Green & MacLeod, 2016). The result of the analysis was a power estimate of 46%; this is well below the threshold of 80% power that is considered adequate (P. Green & MacLeod, 2016). Using the power analysis proposed by Brysbaert and Stevens (2018), the pilot study, having 304 observations (38 stimuli x 8 participants), would also not meet the minimum recommendation of 1600 observations. To determine the number of participants for the main study, a simulation using the **powerSim** procedure was performed for 40 participants. The result of the simulation was a power estimate of 98%. Although this exceeds the recommended threshold of 80%, 40 participants would only provide 1520 observations.

We aim to recruit 60 participants (2280 observations) for the main study to ensure that we have adequate power.

5. Conclusion

The results of the pilot study indicate that the time exposure to the stimuli and the level of difficulty of the stimuli are both appropriate and do not need to be adjusted for the main study. The results also indicate that the measures of self-perceived expertise are relevant and should be included in the main study.

We found that for the 20 stimuli that participants were asked what cues or program components they used to determine whether or not there was a data race, the responses varied greatly ranging in level of detail and also in the number responses they provided making it challenging to analyse these data. We speculate that the reason for the variation in responses is due to the open-ended nature of this question. In the main study we will use an approach similar to Burkhardt, Détienne, and Wiedenbeck (2002) and include questions about specific components of the code to determine the type of information that is included in the mental representations formed by participants. We predict that by asking more specific questions we may be able to gain a better understanding of the participants' mental models and that they will be more likely to provide answers to more direct questions. Therefore, in the main study, 12 of these 20 stimuli, six with a data race and six without, will be given more specific questions regarding the data structures in the programs. For eight of the stimuli, four with a data race and four without, participants will still be asked what cues or program components they used to determine whether or not there was a data race.

In adding questions for the main study, we also considered that, in addition to the program and situation model, an execution model has been proposed that includes the behaviour of data structures (Aubanel, 2020). The behaviour of data structures is considered an important part of parallel program comprehension. Specifically, how the data structures are accessed and changed by one or more threads and the relationship between data structures, is particularly important for determining if a data race exists. The importance of data structures is also evident by the responses in the pilot study when asked what program components were used to determine if there was a data race or not. Of the 60 responses from the highest performing participants, 11 responses referred to data structures, either by name or to the elements within a data structure. We propose that although the data structure behavior is important, the contents of the data structures and the size of the data structures will not necessarily be part of the participants' mental models as those details are not critical to the task. The two types of questions that will be asked of participants are questions related to the contents and size of data structures, and the other is related to accessing and changing of the contents of the data structures and relationships between data structures. Six stimuli, three with a data race and three without, will be selected for each type of question.

The main study has been preregistered with Open Science Framework (OSF) (Bidlake, 2022) and the stimuli, data, and R code will be made available upon completion of the project (pilot and main study).

6. References

- Abbes, M., Khomh, F., Guéhéneuc, Y.-G., & Antoniol, G. (2011, Mar). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th european conference on software maintenance and reengineering* (p. 181–190). doi: 10.1109/CSMR.2011.24
- Arab, M. (1992, 2). Enhancing program comprehension: Formatting and documenting. *ACM SIGPLAN Notices*, 27(2), 37–46. doi: 10.1145/130973.130975
- Aubanel, E. (2020). Parallel program comprehension. In *31st Annual Workshop of the Psychology of Programming Interest Group (PPIG 2020)*.
- Barfield, W. (1997, 12). Skilled performance on software as a function of domain expertise and program organization. *Perceptual and Motor Skills*, 85(3, Pt 2), 1471–1480. doi: 10.2466/pms.1997.85.3f.1471

- Bates, D., Mächler, M., Bolker, B., & Walker, S. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1), 1–48. doi: 10.18637/jss.v067.i01
- Bateson, A. G., Alexander, R. A., & Murphy, M. D. (1987). Cognitive processing differences between novice and expert computer programmers. *International Journal of Man-Machine Studies*, 26(6), 649–660. doi: 10.1016/S0020-7373(87)80058-5
- Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., & De Lucia, A. (2013, May). An empirical study on the developers' perception of software coupling. In *2013 35th international conference on software engineering (icse)* (p. 692–701). doi: 10.1109/ICSE.2013.6606615
- Beniamini, G., Gingichashvili, S., Orbach, A. K., & Feitelson, D. G. (2017, May). Meaningful identifier names: The case of single-letter variables. In *2017 IEEE/ACM 25th international conference on program comprehension (icpc)* (p. 45–54). doi: 10.1109/ICPC.2017.18
- Bergantz, D., & Hassell, J. (1991). Information relationships in prolog programs: How do programmers comprehend functionality?. *International Journal of Man-Machine Studies*, 35(3), 313–328. doi: 10.1016/S0020-7373(05)80131-2
- Bhatia, S., & Malhotra, J. (2014, Aug). A survey on impact of lines of code on software complexity. In *2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014)* (p. 1–4). doi: 10.1109/ICAETR.2014.7012875
- Bidlake, L. (2022, Feb). *Validation of stimuli for studying mental representations formed by parallel programmers during parallel program comprehension*. OSF. Retrieved from <https://osf.io/fcnyx/> doi: 10.17605/OSF.IO/FCNYX
- Bidlake, L., Aubanel, E., & Voyer, D. (2020, July). Systematic literature review of empirical studies on mental representations of programs. *Journal of Systems and Software*, 165, 110565. doi: 10.1016/j.jss.2020.110565
- Boshernitsan, M., Graham, S. L., & Hearst, M. A. (2007). Aligning development tools with the way programmers think about code changes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 567–576). doi: 10.1145/1240624.1240715
- Brysbaert, M., & Stevens, M. (2018). Power analysis and effect size in mixed effects models: A tutorial. *Journal of Cognition*, 1(1). doi: 10.5334/joc.10
- Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(22), 115–156.
- Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed ed.). Hillsdale, N.J.: L. Erlbaum Associates. Retrieved from <http://www.gbv.de/dms/bowker/toc/9780805802832.pdf>
- Davies, S. (1990). The nature and development of programming plans. *International Journal of Man-Machine Studies*, 32(4), 461–481. doi: 10.1016/S0020-7373(05)80143-9
- Détienne, F. (2001). *Software design-cognitive aspect*. Springer Science & Business Media.
- Feigenspan, J., Kastner, C., Liebig, J., Apel, S., & Hanenberg, S. (2012, Jun). Measuring programming experience. In *20th IEEE International Conference on Program Comprehension (ICPC)* (pp. 73–82). IEEE. doi: 10.1109/ICPC.2012.6240511
- Fox, J., & Weisberg, S. (2019). *An R companion to applied regression* (Third ed.). Thousand Oaks CA: Sage. Retrieved from <https://socialsciences.mcmaster.ca/jfox/Books/Companion/>
- Furman, S. M. (1998). *Improving software comprehension* (Unpublished doctoral dissertation). ProQuest Information & Learning.
- Gilmore, D. J., & Green, T. R. G. (1988). Programming plans and programming expertise. *The Quarterly Journal of Experimental Psychology A: Human Experimental Psychology*, 40(3-A), 423–442. doi: 10.1080/02724988843000005
- Green, P., & MacLeod, C. J. (2016). simr: an r package for power analysis of generalised linear mixed models by simulation. *Methods in Ecology and Evolution*, 7(4), 493–498. Retrieved from <https://CRAN.R-project.org/package=simr> doi: 10.1111/2041-210X.12504
- Green, T. R. G. (1989). Cognitive dimensions of notations. In *Proceedings of the Fifth Conference*

- of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V (pp. 443–460). New York, NY, USA: Cambridge University Press.
- Liao, C., Lin, P.-H., Asplund, J., Schordan, M., & Karlin, I. (2017). Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 11:1–11:14). ACM. doi: 10.1145/3126908.3126958
- Maqsood, R., & Ceravolo, P. (2018, Jul). Modeling behavioral dynamics in confidence-based assessment. In *2018 IEEE 18th International Conference on Advanced Learning Technologies (ICALT)* (p. 452–454). doi: 10.1109/ICALT.2018.00112
- Mattson, T., Koniges, A., He, Y. H., & Chapman, B. (2018). *The OpenMP Common Core: A hands on exploration*. SC.
- Mattson, T., & Wrinn, M. (2008). Parallel programming: Can we please get it right this time? In *Proceedings of the 45th Annual Design Automation Conference* (pp. 7–11). New York, NY, USA: ACM. doi: 10.1145/1391469.1391474
- Nosál', M., & Porubán, J. (2015, Jun). Program comprehension with four-layered mental model. In *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)* (p. 1–4). doi: 10.1109/EMES.2015.7158420
- Peirce, J., Gray, J. R., Simpson, S., MacAskill, M., Höchenberger, R., Sogo, H., ... Lindeløv, J. K. (2019, Feb). Psychopy2: Experiments in behavior made easy. *Behavior Research Methods*, 51(1), 195–203. doi: 10.3758/s13428-018-01193-y
- Pennington, N. (1986). *Stimulus structures and mental representations in expert comprehension of computer programs*.
- Pennington, N. (1987a). Empirical studies of programmers: Second workshop. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), (pp. 100–113). Norwood, NJ, USA: Ablex Publishing Corp.
- Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3). doi: 10.1016/0010-0285(87)90007-7
- R Core Team. (2021). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from <https://www.R-project.org/>
- Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004, Jun). Self-efficacy and mental models in learning to program. In *Proceedings of the 9th annual SIGCSE conference on Innovation and Technology in Computer Science Education* (p. 171–175). New York, NY, USA: Association for Computing Machinery. Retrieved from <http://doi.org/10.1145/1007996.1008042> doi: 10.1145/1007996.1008042
- Shargabi, A. A., Aljunid, S. A., Annamalai, M., & Zin, A. M. (2020, Jul). Performing tasks can improve program comprehension mental model of novice developers: An empirical approach. In *Proceedings of the 28th International Conference on Program Comprehension* (p. 263–273). New York, NY, USA: Association for Computing Machinery. Retrieved from <http://doi.org/10.1145/3387904.3389277> doi: 10.1145/3387904.3389277
- Soloway, E., & Ehrlich, K. (1984, Sep). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), 595–609. doi: 10.1109/TSE.1984.5010283
- Teasley, B. (1994). The effects of naming style and expertise on program comprehension. *International Journal of Human - Computer Studies*, 40(5), 757–770. doi: 10.1006/ijhc.1994.1036
- Tubaishat, A. (2001). A knowledge base for program debugging. In *AICCSA '01 Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications* (Vol. 2001-January, pp. 321–327). doi: 10.1109/AICCSA.2001.934005
- von Mayrhauser, A., & Vans, A. M. (1998, 6). Program understanding behavior during adaptation of large scale software. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)* (pp. 164–172). doi: 10.1109/WPC.1998.693345
- Wiedenbeck, S., & Ramalingam, V. (1999, 07). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies*, 51(1), 71–87. doi: 10.1006/ijhc.1999.0269

Coding or AI? Tools for Control, Surprise and Creativity

Alan F. Blackwell
Computer Laboratory
University of Cambridge
afb21@cam.ac.uk

Abstract

This essay presents an argument, accessible to non-specialists, for creative tools that are conceived as programming languages rather than as creative artificial intelligences (AI). The computational implications of creative experience are explained (again, for non-specialists) in terms of fundamentals of information theory. Using analogies to musical instruments, and drawing on recent advances in generative large language models, the paper explains the potential of aleatoric and stochastic elements in programming languages for live coding, and contrasts these with the opportunity for generative language models to be incorporated in programming tools.

1. Introduction

This paper relates to interactive systems that mix elements of programming and machine learning, including program synthesis algorithms and design strategies for mixed initiative interaction. The common theoretical basis spanning these fields, quite familiar to long-term delegates at PPIG (the Psychology of Programming Interest Group), are the decisions and actions that are taken by users on the basis of attention investment in abstraction use (Blackwell 2002a, 2002b), and the implications of notational aspects of the system as discussed in the Cognitive Dimensions of Notations framework (Green 1989, Green & Petre 1996) and subsequent variants such as Patterns of User Experience (Fincher 2002, Blackwell & Fincher 2010).

This issue is explored, in recognition of this 2022 workshop's theme "PPIG and the Muse," in relation to questions of creativity, and illustrated with some simple musical examples. To the extent that machine learning research continues the long-term agenda of AI, this combination of machine learning and creative programming engages directly with the longstanding concern of whether an AI system can or cannot be creative. That topic has been discussed at great length, in particular through the work of Margaret Boden (1998, 2010), and in many collections and scholarly contributions that draw on and extend her framework.

In this paper, I do not present any ideas that are fundamentally different to Boden's analysis, but I emphasise a perspective from programming, with an information theoretic account of creative experience that is intended to engage with the literature on end-user programming (where many users might have programming-like experiences that would not necessarily be described as programming languages in the conventional understanding – Blackwell 2002a, Blackwell, Robinson et al 2002). The specific genre of programming languages to which this analysis is especially relevant is live coding languages, and in particular those used for coding music (Collins et al 2003, Blackwell & Collins 2005, Blackwell et al 2023 in press). In relation to other research on end-user programming languages, this paper is more relevant to those kinds of end-user programmers who are motivated by creative and artistic questions rather than practical ones (Aghaee et al 2015, Blackwell 2017).

2. Programming Languages vs. AI design strategies

A central concern of the Psychology of Programming field over many years has been about giving users greater control, so that they are able to explain to computers what the user actually wants to do. In contrast to the field of natural language processing, which aims to have computers understand the way people *naturally* talk to other people, psychology of programming considers the alternative that an *artificial* language (or perhaps 'un-natural language') might be a more effective design approach in the longer term.

The more familiar research agenda of natural language processing can be described in terms of the Turing Test, in which a computer interacting via a typewriter keyboard would become indistinguishable

from a human (Turing 1950). However there are two ways to win the Turing Test. The hard way is to make computers more and more intelligent, until computers and humans are as smart as each other. The easy way is to construct software systems, businesses, and social networks that cause humans to act more and more stupidly, until computers and humans are as stupid as each other.

Psychology of programming therefore provides necessary fightback against the companies and organisations that push “AI” algorithms onto us, perhaps because they hope that making their customers and citizens more stupid might result in better business and less inconvenience (for the company). Critical analysis of machine learning algorithms from an HCI perspective (e.g. Blackwell 2015, 2019) has suggested some of the problems that result, and recent advances in Human-Centred AI (e.g. Shneiderman 2022) argue that instead of creating artificial intelligences, what we need to create is intelligent super-tools. Such super-tools, if they are to be created, can certainly benefit from the insights of psychology of programming that were discussed in the introduction.

3. Creativity and AI

Despite the familiar human-centric design emphasis on giving users greater control (and at PPIG, the possibility of more powerful and general control through programming), this paper considers the (relatively rare) occasions when we enjoy having a machine doing something unexpected. That is, the machine does something beyond, or other than, what the user had specifically asked for. This is related to the questions that AI philosophers ask about whether a machine can be creative, although I am going to argue that “creativity” is the wrong word, both philosophically and technically. What I’m actually going to talk about is when a machine does something *surprising*.

This distinction follows the well-established writing of Boden on the question of surprise (2010). As Boden analyses in far greater detail, there is an everyday relationship between creativity and surprise. For example when we see somebody make a creative solution to a problem, this seems creative precisely because we hadn’t thought of that solution ourselves. If we had already thought of the same solution, it wouldn’t be surprising, and would seem less creative as a result, despite the fact that the inventor might have engaged in exactly the same cognitive process, whether or not anybody was watching.

Much recent philosophical discussion about whether machines can be creative has focussed on analysis of a single incident – move 37 of the Go match between AlphaGo and Lee Sedol, a move which it seems none of the professional Go players or commentators had considered before. This single move has been described as if it introduced the dawn of a new age of machine creativity (Curran et al 2020). But in this argument of this paper, I’m going to argue that move 37, although clearly surprising to those directly involved, was not really very creative in the sense that humans find interesting.

4. Information Theory and Aesthetic Experience

To understand this important distinction, we need to use the principles of information theory developed as a theory of communication by Claude Shannon at the telephone company research institute Bell Labs (Shannon 1948, Shannon & Weaver 1949). In case this paper is read by non-specialists, I will note that information theory is the fundamental operating principle of all information and communication technologies. Shannon’s discovery and formalisation of information theory is perhaps the most significant practical advance in mathematical physics since Isaac Newton’s laws of gravity. Information theory is not yet taught in high schools, but certainly will be before long. And as far as Go players, or philosophers of creativity, are concerned, I will suggest that trying to explain a creative move without using the mathematics of information theory is like attempting a philosophical explanation of why balls move the way that they do on a pool table, but without using Newton’s equations.

In its simplest form, information theory gives us a quantitative measure of the amount of information that has been sent over some communication channel. Although this seems straightforward, there are some subtle points, so subtle that they might even seem paradoxical. After more than 70 years since Shannon first started to define principles of digital communication over telephone lines, we have all become very familiar with measuring amounts of data in terms of megabits or gigabits needed to stream a movie or email a photograph. The paradoxical aspect for many is that not all *data* counts as *information* in the theoretical sense developed by Shannon. If you were to send me an email message, and then immediately send the same email message again, the second message might use a lot of data,

but it hasn't given me any more information – it is *redundant* in the technical jargon of information theory. On the other hand, if you sent the second email to a different person, it's not redundant, because *that person* hasn't seen it before. Although it is easy to measure the number of bits being transmitted along a cable, the information content is not necessarily easy to measure objectively, because it depends on the person who receives the message. If the receiver already knows what the message will contain, no information has been transferred. Information theory is a measure of that amount of information provided by a message, that the receiver does not already know. Information theory is a measure of surprise!¹.

I've followed the argument of Boden and others that what we call "creativity", when done by a machine, is more precisely a measure of how much we are surprised by what the machine does. So information theory can be used as a measure of creativity. When a Go-playing robot is being observed by an expert human Go-player, and every move is exactly the one expected, there are no surprises, no subjective impression of creativity, and no information. Another person, with less expertise in Go, might be surprised by every move they see – but this is not because the robot has become more creative, just that we have found an audience who is easily impressed. In many fields of human creativity, we call an ignorant person who is easily impressed *undiscriminating*, and an expert *discriminating*. In fact, this is another technical term in information theory, and perhaps a mathematical formalisation of the proverbial understanding that creativity, like beauty, is in the eye of the beholder.

We see this all the time in human art forms, and especially in music, which is probably the easiest of art forms to describe mathematically. Some sequences of notes are very predictable. Perhaps the most predictable is when I have a favourite song, and listen to the exact same recording many times. Once I get to know it, it can be comforting to listen to a piece of music that I know really well, although also a little boring to listen to it over and over again. For most people listening to music, there is a sweet spot between things that are comforting, familiar (or possibly boring), and things that are interesting and surprising. For example, I like to listen to blues music, which often follows a 12-bar pattern, with each chord determined by its place in that sequence. Because the sequence of chords is so familiar, it's satisfying to listen to. The chords themselves, and the tune of the vocal part, are likely to follow one of the familiar patterns of Western music, such as the minor scale. For someone who listens to a lot of blues, it is possible to hum along and make a good guess at what note will come next, or even improvise your own new part to follow the rules you have learnt.

If you aren't a musician, but have access to a piano keyboard, you can get an understanding of this with a simple experiment. Just play up and down the white keys, playing one note at a time, and changing direction whenever you feel like it. You are now playing a tune in the key of C Major. After a minute or so, press one black key. You will hear this as a kind of surprise, different from the comforting (and possibly boring) routine of the C Major tune. Pretty much all Western music is built on managing the amount of surprise, including just the right number of black notes, and at appropriately surprising moments, to maintain the interest of the listener. In contrast, if you play any old combination of white and black notes at random, the result will not sound very pleasant, or even like a coherent tune. In the mid 20th-century, "serialist" composers set themselves the challenge to use all 12 white and black notes before repeating any of them. The result has not caught on, and doesn't seem to have become the basis of any familiar pop songs or comforting lullabies.

Great musical performers, in blues or any other genre, follow the rules of that genre, but also add a little surprising spice by using a different note from time to time. Pop music has its own set of rules, and every pop song sounds more or less like another song, because that's what makes pop music comforting and familiar. If you listen to heavy metal, there is a different set of conventions, and we each learn to like what we like. Heavy metal fans like stuff that might not seem appropriate to play in a restaurant, and think they have extreme tastes, but a soprano coming out to sing *O mio babbino caro* in the middle

¹ That exclamation point is not really very surprising, or creative – perhaps another example of redundancy.

of a concert by the notorious death metal band *Vile Demons of Excrement*² would be even more surprising than a blood-curdling scream.

The lesson from this is that we like certain kinds of surprise, but not others, even within art forms that apparently invite disruptive alternatives to mainstream culture. This points to another paradox of information theory, which is that the greatest amount of information you can send over a communication channel is a completely random sequence of numbers. Imagine a message that is composed of zeros and ones, where every bit is chosen by flipping a coin. The person receiving this message would have no way to predict what the next bit is going to be, meaning that every bit that arrives is a complete surprise. This message is as surprising as it could possibly be, but we don't perceive it as being creative. On the contrary, a sequence of random bits is described as *noise* (another technical term in information theory).

A theory of creativity in machines must make a distinction between the information theory categories of *signal* and *noise*. The signal is the message we want to hear (a meaningfully creative novelty), and the noise is random stuff that is not interesting. We are interested in messages that relate to our expectations, and to things that are familiar, whether Go, 12-bar blues, or photographs of our children. When someone jumps in with something completely random, for example noise corrupting a digital photograph, it is annoying and upsetting, not an act of creativity on the part of the camera. We do like to receive surprising messages, for example when an old friend contacts us out of the blue. But we don't like very "surprising" messages such as a loud burst of random static. The kinds of surprise that we recognise as being creative depend on two things – somebody who wants to tell us something, and our expectation of what we might hear.

5. Machine learning and surprise

Now, let's consider how these principles of information theory, which we perceive as expectation and surprise in aesthetic experiences, relate to creativity in machines.

Machine learning systems, by definition, learn to replay what they have seen before. At the simplest level, predictive text of the kind we have on our phones has been trained with a *language model* – a dictionary, and perhaps a few hints about what common word might come next. This model is used to predict what comes next according to a principle of information theory – the principle of least surprise. The word that your phone predicts is always the least surprising word, that is, the one that is most expected to come after the letters you have just entered. According to information theory as discussed in the previous section, the least surprising word is also the least creative. Indeed, who would want a more creative predictive text algorithm on their phone? A *creative* "AI" phone, which surprised you with words completely different to what you were trying to say, would be a pain in the arse.

The latest generation of large language models, based on deep neural networks that can predict whole sentences or paragraphs at a time, follow exactly the same information theoretic principle. They have been trained with far more than a simple dictionary – including wikipedia pages, online forums, and whatever other text their developers can find online for free (Bender et al 2021). Sometimes the results can be surprising, but only to people who haven't spent much time on the internet. It's somewhat surprising, but probably more horrifying, when these models produce large quantities of offensive, racist and violent text. In fact it's not that surprising that the internet is full of violent and racist text, because we seem to have designed it to be that way (Monteiro 2019).

This is a sad fact of modern life, and certainly suggests some precautions we might take before creating more powerful predictive text systems for our phones. However, none of this is evidence that the machine is being creative, and none of the resulting text is surprising in the information theory sense. It is just a communication channel, where the things that come out are the product of the things we put in (GIGO, or garbage in, garbage out, according to the old software developer's saying).

Just like music, surprises come from the random bits of noise that we didn't expect because they didn't follow the pattern. We could train a neural network language model with lots of beautiful poetry, instead

² Not a real death metal band.

of forum posts by violent racists, and that network would produce something that was (hopefully) more poetic. But this result would not be surprising, and it still wouldn't be creative. According to information theory, creative surprise needs to involve some kind of random decision, if it is going to be genuinely unexpected by anybody.

This is exactly how the generative language models work. They have a built-in random number generator, that produces less expected outcomes by essentially tossing a coin, as if your predictive text system occasionally threw a completely random word into one of your SMS messages, just for fun. On your phone, most of the suggestions it gives are actual words, rather than random letters, and in a large language model, most of the suggestions are actual sentences. It just that as the randomness "temperature" is turned up, those sentences become more surprising, further away from the more likely things that any person would say next.

6. Surprise as a creative tool

I've experimented with large language models trained to plagiarise my own work, by feeding 10 years of my academic papers into a neural network. If the temperature is very low, the results aren't very surprising. In fact, they are very likely to produce outright plagiarism, just repeating the kinds of sentences and phrases that I've often used myself (indeed, just as I have done in writing this paper for PPIG 2022). As the temperature is turned up, more random stuff can happen, and it becomes more entertaining to look at the results. These sentences do look kind of like things that I would write, but often with weird twists or gaps in logic. Sometimes, these even seem like creative opportunities – saying things that I've talked about before, but suddenly combined in a new way that I haven't thought of (Alexander et al 2050/2021).

This process – using coin tosses or throws of a dice to generate new ideas - is a common strategy used by music composers and other artists when they would like to develop their ideas in a new and unexpected direction. 20th century composers like John Cage were famous for "aleatoric" compositions starting from a random process (Leong et al 2006). Brian Eno's "oblique strategies" (Eno & Schmidt 1975) include all kinds of disruptive suggestions, intended to help a creative artist break out of their comfortable habits of thought and try something completely new. Painters and sculptors also like to surprise themselves through conversations with their material, where random splashes, unexpected turns in the wood grain, or slips of the chisel might help them to explore possible ideas beyond their conscious intentions.

We need to be very clear about the difference between the use of randomness as a compositional strategy, and the presentation of randomness as an artwork in itself. I've already explained that the most surprising message (in an information theory sense) is a sequence of completely random numbers or coin tosses, where there is no way to predict any number from the ones that came before. Completely random messages are very surprising, but also very uninteresting, because they communicate nothing at all. There is no hidden message in a series of coin tosses or dice throws, no matter how much we might want to find one. And of course, the human desire to find meaning has resulted in many superstitious practices where people do look for meaning in coin tosses, dice throws, or cards drawn from a deck. Tarot readings, gambling, and other entertaining performances, just like the compositions of John Cage, use random information as a starting point for human creativity.

But what we need to remember is that random information is not communicating anything. Random information is perceived as surprising precisely because there is no actual message that could have been anticipated. We can enjoy the performance of a Tarot reader, but the idea that random events have meaning in themselves is nonsense. We also need to understand that the same is true of the random processes that cause us to attribute "creativity" to AI systems. An AI system can produce surprising output if it includes random elements. But this is not creative in the human sense, because it is not a message from anywhere. In general, random elements within a digital sequence are not signal, but noise. In terms of philosophy of mind, a random message has no meaning because there is no *intention* behind it. All of these things can be directly described in terms of information theory. Philosophical speculations about creativity, when applied to surprising outputs of a randomising system, are no better than mystical explanations of why there might be meaning hidden in a Tarot card deal.

These are the simple reasons why digital machines may be surprising, but not creative. As I said earlier, Shannon's principles of information theory are as fundamental as Newton's principles of gravity, so searching for a magical basis of creativity in AI that would be contradicted by information theory is as productive as research into anti-gravity machines. Possibly entertaining in science fiction or theatrical magic performances, but not a serious basis for science or engineering.

None of this is to say that randomness is not a valuable aid to human creativity. Just as with John Cage's aleatoric composition methods, it can be exciting to experience art works with the right mix of comfort and surprise. We don't like music (or any other artwork) to be too comforting, because it is just boring. Small amounts of surprise, at the right time, add spice to our mental worlds. Sometimes, a random event from our own computer, or even an unexpected suggestion from a predictive text keyboard, might be a happy accident that we enjoy responding to, and perhaps even repeating to our friends, just as when John Cage saw a dice throw that he liked, and included that note in his composition.

In future, I expect that I will use more powerful generative language models to save myself typing. The ones I use today can already complete full sentences. Usually in a very boring way, because these sentences reflect the most expected, least surprising, thing I could say. That's OK, because I do quite often need to write a lot of boring and repetitive text, for example in this paper. I expect that I will also enjoy turning up the temperature, for more surprising randomness, on a day when I've bored myself, and trying to think of something new to say.

These kinds of tools are all great, and I look forward to using them, but we can't confuse the dice throw inside the neural network with the creative decision that I make myself when I decide just when to throw the dice, or when I choose which one of the various random suggestions is most interesting to use. At that point, *I'm the one being creative*, because I have a message that I want to send - in this essay, a message to you, the reader. If you were to throw a dice to generate a sequence of random words and letters instead of reading this paper, or if you were to read the randomised output from a chatbot, you might find it an entertaining game, but there is literally nobody sending you a message, any more than if you chose words from a dictionary by throwing a dice.

7. How to design surprising tools

To bring this back to the PPIG theme, I now turn to the kinds of programming languages that artists use. Because artists appreciate opportunities to incorporate random surprises in their work, they are unusually interested in programming languages that sometimes behave in random ways. In other circumstances, this is *not* what we want from a programming language. Certainly not a program that is managing the cruise control on a car, or the thermostat on an oven, or a nuclear power station.

In the hands of an expert user, slightly unpredictable tools can be surprisingly valuable. Aeronautics engineer Walter Vincenti (later director of the Stanford university program in Science, Technology and Society), described the long-term research effort to create more stable and predictable airplanes, which although technically successful, resulted in planes that pilots hated flying (Vincenti 1990). It turned out that an expert pilot needs a plane that is just slightly unstable, just as a Formula 1 driver performs best in a car that is always on the edge of going out of control (which would be a car that a normal person is likely to crash within seconds).

I spent several years working with a team of expert violin researchers, analysing the acoustic vibration and auditory perception of that surprisingly complex object (Fritz et al 2012). It turns out that violins, like aircraft and racing cars, are designed to be unpredictable. The mode of vibration for a string being driven by a rosined bow has an incredibly complex variety of frequency components, amplified by a wood body that is designed to vibrate in many different directions at the same time. The resulting rich timbre can be controlled by an expert player to create a huge range of different sounds. In the hands of a beginner, the surprising range of random noises that can come out of a violin make it one of the more unpleasant choices for a child's instrument. It is possible to "tweak" a string instrument to emphasise some kinds of vibration more than others, and our research team interviewed a specialist adjuster who makes a living just by moving around small internal components of the instruments played by top-flight cellists. He showed me how he could adjust a cello to make a more reliably beautiful and comforting sound, which sounded good even in my bass-player's hands (Blackwell 2022). But as he explained, this

adjustment, however much I liked it myself, is the opposite of what an expert player wants. Professional music is only interesting with that element of surprise, and it has to be available at the times the player needs it. And to return briefly to the argument earlier in this paper, it is not the violin being creative here – the random elements of surprise are a creative resource, no more than a tool for the human artist.

So how does this relate to computer-based tools? Just as with violin players and airplane pilots, professional computer artists appreciate tools that have the capacity to surprise them. I have spent a lot of time working in the musical genre of “live coding” (Collins et al 2003, Blackwell & Collins 2005, Blackwell, McLean et al 2014, Blackwell, Cocker et al 2023 in press), where music is synthesised by an algorithm that the performer creates on stage, writing the code in front of an audience. In its most popular incarnation, this is the “algorave” (Collins and McLean 2014), where a nightclub full of people may be dancing to the algo-rhythms.

Live coder Sam Aaron was working in our research group when he created his popular Sonic Pi language (Aaron et al 2013, 2014)³. When Sam and I were discussing the features that would be needed in Sonic Pi, it was clear that some kind of randomising function would be needed, since it is often useful in performance. At a micro-level, a small amount of randomness can make a repeated drumbeat sound less robotic, or the frequency components of a synthesised sound richer and more complex in the same way as for a bowed instrument like a violin. Random walks can also be used to choose the notes of a tune, just as in the experiment I suggested earlier where a non-musician might wander backwards and forwards over the white notes of a piano. In Sonic Pi, a straightforward approach to creating a tune-generating program might be to generate the scale of C-major, and then proceed up and down that scale at random. To make the tune slightly more interesting, the program might occasionally jump two notes rather than one, or add the occasional black note, but do those things at unexpected moments, which would be determined by another random variable.

Sonic Pi programmers certainly do these things, and most live coding performers use similar strategies, making use of the “random” keyword that is in almost all programming languages. Careful choice of the amount of randomness, just like choosing the temperature in more creative uses of a large language model, can be a source of interesting inspiration to the human artist.

However, Sam noticed an interesting thing after a year or two performing and composing with Sonic Pi, and talking to the other musicians and school students who it was designed for. Although random surprising notes can be interesting in moderation, these random tunes pretty quickly get boring, even if they are following the rules of western harmony scales and chords, because there is no artistic intention. Even worse, a randomising program would occasionally produce something that sounded really beautiful, but would then vanish, never to be heard again. Sam therefore changed the random function in Sonic Pi so that it is not really random at all. Most programming languages have very sophisticated random number generators, to guarantee (for example) that there is no way a code-breaker would be able to predict the random key to an encrypted message. The new Sonic Pi randomising function is an artistic tool that will produce an unexpected result when first used, but then do the same thing again if you ask for it. All music plays with the expectation systems of the human brain, leading us to expect one thing, surprising us with something else, then perhaps comforting the listener by repeating the same thing again. The verse and chorus in pop songs, the repeated passages in a Vivaldi concerto, and the thematic motifs of a Mahler symphony all draw us in by repeating variations of an intriguing idea until it becomes familiar.

These are also the ways that we can have creative experiences with AI systems. Surprise plays an important part in aesthetic experience, and tools that sometimes behave randomly can produce surprises in a way that is described by information theory. Tools for expert creative artists do need these kinds of capability, but the tool itself has nothing to communicate, because a random event is not a message from anywhere. AI systems are not creative, but if we can program them to manipulate comfort and surprise, they certainly become creative tools.

³ If any readers would like to try this for themselves, just download the Sonic Pi package (free for Windows, Mac, Linux and Raspberry Pi), and experiment with its built-in tutorials and examples to make your own music synthesis code.

Insights from psychology of programming are desperately needed, in the study of machine learning in art, literature and music. It is clear from the discussion in this paper that creativity will play a part, and that surprise will play a part. As the broader population becomes more familiar with the implications of generative neural network architectures, for example through the recent popularity of the DALL-E Mini project, it seems likely that these will become more widely used as an element of creative process. The selection of suitable prompts to generate interesting output from such text-to-image systems is becoming known as *prompt engineering*, or even *prompt programming*, suggesting that insights from psychology of programming and end-user software engineering may be useful. There are certainly immediate opportunities for insight from the application of Cognitive Dimensions in this domain, since the language interfaces currently being used are woefully inadequate as a programming notation, even for creative and exploratory design activities, on many relevant dimensions.

8. Creating novel source code


Does this essay have any implications for the use of generative networks to produce program source code, including well known demonstrations such as Copilot Labs from GitHub? We might certainly imagine that aleatoric processes for source code synthesis could be a component of an artistic project or programme, perhaps alongside other practices of programming identified by Bergstrom (2016). There have also been interesting experiments, over many years, in the random synthesis and perturbation of source code to generate useful programs automatically, for example using genetic algorithms to stochastically explore the space of alternative programs that might produce a desired output. But it is widely expected that systems like Copilot would have more practical utility – “AI pair programmer,” as current promotional material from GitHub suggests. The empirical literature on pair programming, for example as presented at PPIG in the past, suggests that this is not a plausible model of how such tools might work.

Nevertheless, there is certainly opportunity to improve the language models and prediction performance of the code completion algorithms currently used in leading programming editors. Anticipating welcome modes of interaction in code completion might benefit more from consideration of natural language text completion interfaces (otherwise known as predictive text) than by making unlikely analogies to pair programming. Advances along those lines should become useful and welcome, but the generative model in which natural language text input is used to generate source code output more closely resembles a stochastic compiler. If such a “compiler” from natural language to source code interprets the natural language prompt without original “creative” (i.e. random) information, then the output will be a kind of pastiche or plagiarism of the source code that the model was trained on. The distinction between pastiche and plagiarism, in this whole generation of models, is really an outcome of the temperature setting – the amount of random variation. Aside from artistic applications, it seems unlikely that many programmers will be interested in random variations on source code that would otherwise have been expected to work. As a result, the stochastic generation temperature is likely to be kept low, and such programmer assistance algorithms will therefore be closer to plagiarism (or “software reuse”, as it is known when done with permission). Rather than a translation interface, the input prompt in this case is not actually a specification, but rather a query for retrieval of an appropriate piece of code (hopefully) known to work in the past.

Shneiderman (2022) makes the case that any predictive text entry interface can also be regarded as a recommender system, where the recommendation being offered relates to opportunities for time saving, improved textual “productivity” (for those paid by the word), or for improved attention savings, in the analytic terms of the attention investment framework. This perspective is likely to offer a helpful approach to the design of programming assistants that incorporate generative models. Ideally, such models will interact on a mixed initiative basis, for most effective augmentation of the human programmer, taking account of attention investment-based design strategies such as Wilson and Burnett’s Surprise-Explain-Reward (Wilson, Burnett et al 2003), and Horvitz’s advice for mixed-initiative interaction (1999). As with programming by example, the notational choices for explaining the proposed additions to the code are likely to be critical, as few programmers will want to accept automatically generated code whose function they do not understand (Blackwell 2001).

Overall, there is a bright future for the use of machine learning techniques to enhance programming, both for creative purposes and more mundane data processing automation. Nevertheless, meaningful input, if we avoid the temptation toward large-scale institutional plagiarism, will come from the programmers.

9. References

- Aaron, S. and Blackwell, A.F. (2013). From Sonic Pi to Overtone: Creative musical experiences with domain-specific and functional languages. *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pp. 35-46.
- Aaron, S., Blackwell, A.F. and Burnard, P. (2016). The development of Sonic Pi and its use in educational partnerships: co-creating pedagogies for learning computer programming. *Journal of Music, Technology and Education* 9(1), 75-94.
- Aghaee, S., Blackwell, A.F., Kosinski, M. and Stillwell, D. (2015). Personality and Intrinsic Motivational Factors in End-User Programming. *Proceedings of IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2015)*, pp. 29-36.
- Alexander, A., Bassett, C., Blackwell, A. and Walton, J. (2050/2021). *Ghosts, Robots, Automatic Writing: an AI Level Study Guide*. Cambridge Digital Humanities: Cambridge, PREA.
- Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021, March). On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? . In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (pp. 610-623).
- Bergstrom, I. and Blackwell, A.F. (2016). The Practices of Programming. In *Proceedings of IEEE Visual Languages and Human-Centric Computing (VL/HCC 2016)*, pp. 190-198.
- Blackwell, A.F., Cocker, E., Cox, G., McLean, A. and Magnusson, T. (2023, in press). *Live Coding: A user's manual*. MIT Press
- Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of PPIG 2005*, pp. 120-130.
- Blackwell, A.F. and Fincher, S. (2010). PUX: Patterns of User Experience. *interactions* 17(2), 27-31.
- Blackwell, A.F., McLean, A., Noble, J. and Rohrerhuber, J. (2014). Collaboration and learning through live coding. *Dagstuhl Reports* 3(9), 130-168. Edited in cooperation with Jochen Arne Otto.
- Blackwell, A.F., Robinson, P., Roast, C. and Green, T.R.G. (2002). Cognitive models of programming-like activity. *Proceedings of CHI'02*, 910-911.
- Blackwell, A.F. (2001). SWYN: A Visual Representation for Regular Expressions. In H. Lieberman (Ed.), *Your wish is my command: Giving users the power to instruct their software*. Morgan Kauffman, pp. 245-270.
- Blackwell, A.F. (2002a). What is programming? In *Proceedings of PPIG 2002*, pp. 204-218.
- Blackwell, A.F. (2002b). First steps in programming: A rationale for Attention Investment models. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 2-10.
- Blackwell, A.F. (2015). Interacting with an inferred world: The challenge of machine learning for humane computer interaction. In *Proceedings of Critical Alternatives: The 5th Decennial Aarhus Conference*, pp. 169-180.
- Blackwell, A.F. (2017). End-user developers - what are they like? In F. Paternò and V. Wulf (Eds). *New Perspectives in End-User Development*. Springer, pp. 121-135.
- Blackwell, A.F. (2019). Objective Functions: (In)humanity and Inequity in Artificial Intelligence. *HAU: Journal of Ethnographic Theory* 9(1), 137-146.

- Blackwell, A.F. (2022). Too Cool to Boogie: Craft, culture and critique in computing. In J. Impett (Ed.) *Sound Work: Composition as Critical Technical Practice*. Leuven University Press / Orpheus Institute, pp. 15-33.
- Boden, M. A. (1998). Creativity and artificial intelligence. *Artificial intelligence*, 103(1-2), 347-356.
- Boden, M. A. (2010). *Creativity and art: Three roads to surprise*. Oxford University Press.
- Collins, N., McLean, A., Rohrerhuber, J., and Ward, A. (2003). Live coding techniques for laptop performance. *Organised Sound*, 8(3), 321-330.
- Collins, N., & McLean, A. (2014). Algorave: Live performance of algorithmic electronic dance music. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 355-358.
- Curran, N.M., Sun, J. & Hong, J.W. (2020). Anthropomorphizing AlphaGo: a content analysis of the framing of Google DeepMind's AlphaGo in the Chinese and American press. *AI & Society* 35, 727–735. <https://doi.org/10.1007/s00146-019-00908-9>
- Eno, B., & Schmidt, P. (1975). *Oblique strategies*. Opal. (Limited edition, boxed set of cards.).
- Fincher, S. (2002). Patterns for HCI and Cognitive Dimensions: two halves of the same story? In *Proc. 14th Workshop of the Psychology of Programming Interest Group (PPIG 14)*, pp.156-172.
- Fritz, C., Blackwell, A.F., Cross, I., Woodhouse, J. and Moore, B. (2012). Exploring violin sound quality: Investigating English timbre descriptors and correlating resynthesized acoustical modifications with perceptual properties. *Journal of the Acoustical Society of America* 131(1), 783-94.
- Green, T. R. G. (1989). Cognitive Dimensions of Notations. In L. M. E. A. Sutcliffe (Ed.), *People and Computers V*. Cambridge: Cambridge University Press.
- Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7, 131-174.
- Horvitz, E. (1999). Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pp. 159-166.
- Leong, T.W., Vetere, F. and Howard, S. (2006). Randomness as a resource for design. In *Proceedings of the 6th conference on Designing Interactive systems (DIS '06)*, pp. 132–139.
- Monteiro, M. (2019). *Ruined by Design: How Designers Destroyed the World, and What We Can Do to Fix It*. Mule Books.
- Shannon, C.E. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal*. 27 (3), 379–423.
- Shannon, C.E. and Weaver, W. (1949). *The Mathematical Theory of Communication*. University of Illinois Press.
- Shneiderman, B. (2022). *Human-Centered AI*. Oxford University Press.
- Turing, A.M. (1950). Computing Machinery and Intelligence, *Mind* 59(236), 433–460, <https://doi.org/10.1093/mind/LIX.236.433>
- Vincenti, W. G. (1990). *What engineers know and how they know it*. Baltimore: Johns Hopkins University Press.
- Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M. and Rothermel, G. (2003). Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 305-312.

Keynote

Live coding and the 'what-if' paradigm

Alex McLean

UKRI Future Leaders Fellow and Live Coding pioneer

Live coding is an 'end-user programming' community of musicians and other performing artists, which has developed rather separately from the world of software engineering over the past 20 years. As a result, it has some peculiarities. In particular, improvisation is strongly promoted across the community, supported through technological developments such as pure functional reactive programming, in-code visualisation, and algorithmic approaches to pattern-making informed by heritage practices. Through this talk, I'll try to argue that this improvisatory approach offers a third paradigm in programming, combining the 'what' of declarative programming, and the 'how' of imperative programming, to offer an alternative: 'what if?' I'll try to sketch out the difference, why it's needed, and how we might support its development. In the end, the question is how such a formal, explicit approach to notation as computer programming can help us explore what we know, but can't explain.

Story-thinking, computational-thinking, programming and software engineering

Austen Rainer

School of Electronics,
Electrical Engineering
& Computer Science
Queen's University Belfast
a.rainer@qub.ac.uk

Catherine Menon

School of Physics,
Engineering
& Computer Science
University of Hertfordshire
c.menon@herts.ac.uk

Abstract

Working with stories and working with computations require different modes of thought. We call the first mode “story-thinking” and the second “computational-thinking”. The aim of this paper is to explore the nature of these two modes of thinking, and to do so in relation to programming, including software engineering as programming-in-the-large. We use two stories as illustrative examples: a famous six word story and the Byzantine Generals Problem. With these two stories, we explore one fundamental problem, i.e., the problem of “neglectful representations”. We briefly suggest ways in which this problem might be tackled, and briefly summarise our ongoing investigations.

1. Introduction

The term “story” is widely used in software engineering, e.g., the “user story” (Lucassen, Dalpiaz, Van Der Werf, & Brinkkemper, 2015) and the “job story” (Lucassen et al., 2018). Related concepts are also used, e.g., the scenario (Carroll, 2003). But *story* – in the fullest sense of that word – and algorithm are very different things. We use the term “story-thinking” to refer to the ways in which we conceptualise and engage with stories, e.g., how we create them, how we formally analyse and evaluate them, and how and why we appreciate them. We use the term “computational-thinking” to refer to the ways in which we think about computations, e.g., how we create algorithms, how we formally analyse and evaluate them, and how we simulate their execution on real or nominal machines. The contrasts between story-thinking and computational-thinking stimulate a range of questions, e.g., what are the ways in which story and algorithm might relate? how might story – its writing, telling and re-telling – complement computational thinking? and what do we gain, and what do we lose, with these contrasting ways of thinking?

The aim of this paper is to explore the nature of these two modes of thinking, and to do so in relation to programming, including software engineering as programming-in-the-large. The paper contributes the following:

1. A conceptualisation of some of the issues, through the application of these two modes of thinking to two illustrative examples.
2. The identification of a fundamental problem: the problem of “neglectful representations”.
3. Brief proposals for how these problems might be tackled.

It is helpful, at the outset, to briefly review definitions of computational thinking. Aho (2012) defines computational thinking as, “...the thought processes involved in *formulating problems* so their *solutions* can be *represented as computational steps and algorithms*.” (emphasis added). Denning (2009) recognises the *representation* as more fundamental than the algorithm, since the representation needs to be computable (Erwig, 2017) or, on other words, manipulatable by or through computation. Heineman, Pollice, and Selkow (2008) write that, “Designing efficient algorithms often starts by selecting *the proper data structures* in which to represent the problem to be solved.” (emphasis added).

Priami (2007) highlights a particular feature of the representation: “... the basic feature of computational thinking is abstraction [representation] of reality in such a way that the *neglected details* in the model make it executable by a machine.” (emphasis added). This concept of negative selection – i.e., of what is

removed from the model *in order to* allow or support or enable computation – is particularly interesting because the focus with abstraction tends to be on what is retained, i.e., on *retaining* only those features of the thing to be modelled that are *essential* (Starfield, Smith, & Bleloch, 1994).

The remainder of the paper is organised as follows. We start our exploration with an illustrative example, a six word story, in Section 2. In Section 3, we model the example computationally. In Section 4 we introduce and discuss a second illustrative problem, the Byzantine Generals Problem. In Section 5, we then consider computational thinking in the context of software engineering, as programming-in-the-large. In Section 6, we then summarise the problems and propose ways forward. Finally, with Section 7, we briefly conclude.

2. First illustrative example: the sale of shoes

Consider the following short story (which has been intentionally modified slightly from a well-known story in the literary world):

“For sale. Baby’s shoes. Never worn.”

What is your reaction to this story? In his book, *Once upon an if*, Worley (2014) describes his wife’s first reaction to this story. “Oh, no!” he says she responded, an indication that she interpreted the story as one of sadness or tragedy, presumably relating to the life of a baby. As the reader, you might, similar to Worley’s wife, interpret the story as a tragedy. Or you might, as Worley says his wife’s friend did, interpret the story as about someone who buys things for others but those things are not wanted. And, of course, you might have other reactions to the story. In a recent online presentation of this story to software engineering academics, the audience was asked for their reactions. Responses included, “love, compassion”, “sadness”, “cute”, and “nostalgic – story made me reflect on that episode of my life”. Conversely, in a recent teaching activity with undergraduate students, the students were asked to brainstorm possible explanations for the events described in the six words. One suggested explanation was that the six words are simply an advert, albeit a little oddly phrased, for the sale of new shoes. One might imagine this advert in the window of a shop, with the message now “de-particularised.” There is no longer a protagonist (unless you count the shop keeper) nor a plot. There are just shoes for sale.

Furthermore, the reference to a *baby* means the phrase “never worn” carries different connotations compared to, for example, an adult having “never worn” the shoes. The connotations of a baby’s shoes never being worn are tragic because babies are seen as more susceptible to, and “fitting” (in story-terms) for, tragedy. An adult’s “never worn” shoes may imply the shoes have been worn at least once, for example to try them on, but then not worn again, e.g., perhaps they didn’t fit.

Whatever your interpretation of the story, and your reaction to it, the story-teller (the story is based on a story allegedly written by Ernest Hemingway, though this is disputed) has provoked an experience and done so very efficiently. In just six words, the story-teller creates characters (e.g., a baby, and possibly a parent), a plot (e.g., perhaps someone has lost a baby, or not been able to have a baby) and therefore an unfolding of events over time, one or more goals and a struggle (e.g., the goal of having a baby, with the struggle of not having a baby, or of surviving the loss of a baby), an outcome (e.g., the goal was not attained) and an emotional experience for the reader (e.g., sadness).

The entire story is shorter in length than Cohn’s (2004) well-accepted *template* for a *single* user story in requirements engineering: As a <type of user> , I want <goal>, [so that <some reason>]. One reason we chose this story as an illustrative example is because it is concise and therefore easy to present completely in an academic publication with restrictions on page length. But we also chose this story because it is efficient: an extraordinary amount of information and emotion is evoked in only a few words. This efficiency contrasts with the user story. The contrast – between the six-word story and the template for a single user-story – suggests fundamental differences in the way that stories model the world and the way that typical software engineering and programming constructs model the world; and also suggests fundamental differences in the ways that story-thinking and computational-thinking

require us, or encourage us, to *attend* to the world.

3. Representing the story computationally

An alternative way to think about the six-word story is computationally. And it seems that as soon as we start to attend to the six-word story computationally it is no longer a *story* but instead becomes a text. To explore this point we present and consider two forms of text associated with computational thinking: user stories and software designs.

3.1. Representing the story as user stories

Table 1 presents examples of *possible* user stories, first re-stating Cohn’s (2004) template for user stories. These user stories can only be speculative because they depend on how one interprets the six words.

ID	Example
N/A	As a <type of user> , I want <goal>, [so that <some reason>]
1	As a user, I want to be able to sell items, so that I can make some money.
2	As the purchaser of an unwanted item of clothing, I want to sell the item, so that I can recover my costs.
3	As a grieving parent, I want to sell my baby’s shoes, so that I can reduce my financial losses.

Table 1 – Example user stories for the thought experiment

The examples in Table 1 illustrate some of the tensions, strengths and weaknesses of computational-thinking and story-thinking, such as:

1. The *story* tells us very little explicitly about the actual protagonist, other than the protagonist wants to sell a pair of baby’s shoes. We are left to infer the characteristics of the protagonist, including the basis of their goal. In terms of story-thinking, this is an effective rhetorical device. In terms of computational-thinking, this is problematic. There is a tension between story-thinking’s use of evocative connotation and computational-thinking’s standards of specification and denotation.
2. There is also a tension between story-thinking’s particularity and computational-thinking’s abstraction. The users, or personas, defined in User Stories 2 and 3 can be abstracted to the user defined in User Story 1. Conversely, whilst the emotional aspects of the user in User Story 3 (e.g., sadness, nostalgia, not wanting something) might be *representable* in a software system, it is not clear what gain there is *for the software system* with such a representation. Or in other words, it is not clear how representing such states in the system helps the user with what they want to *do*.

This distinction between particularity and generality can be further illustrated by comparing the version of the story presented at the beginning of Section 2 with the well-known, original story. In the original version, the story reads, “For sale. Baby shoes. Never worn.” Adding one apostrophe together with one letter, the letter *s*, helps to particularise the story, e.g., by adding a new character, a specific baby. There is no longer the abstract category, baby shoes; nor is there a collection of babies’ shoes. Instead there is “[a] baby’s shoes”. There is another, perhaps more subtle, particularising effect too: the word “Baby” can often be used as a name, e.g., “Have you fed Baby yet?” or, “Don’t wake Baby.”

3. In the act of preparing User Stories to model the story, we begin to reshape our conception of the story. We specify, and possibly also abstract, and by doing so we change the story, limit it and reduce its effect as a story.

3.2. Representing the story as a software design

A second way to think computationally about the story is in terms of software designs. Figure 1 presents a simple UML object diagram (Figure 1a) and a database table (Figure 1b) for the sale of a product, in this case a pair of baby’s shoes. We can of course think with and about these models, but notice how the *kind* of thinking we do with these models is different to the kind of thinking we do with the *story*.

With the object diagram we can, for example, infer that the object is an instance of the class `Product`, and we can see the `forSale` attribute is a boolean variable. We can also see that no private or public methods are stated for this object, at least in the diagram, and we might examine the class `Product` for such methods. The database table shows that we can easily begin to design a database for persistent storage of information taken from the story. Being outputs of our thinking, these models provide insights into the nature of our computational-thinking.

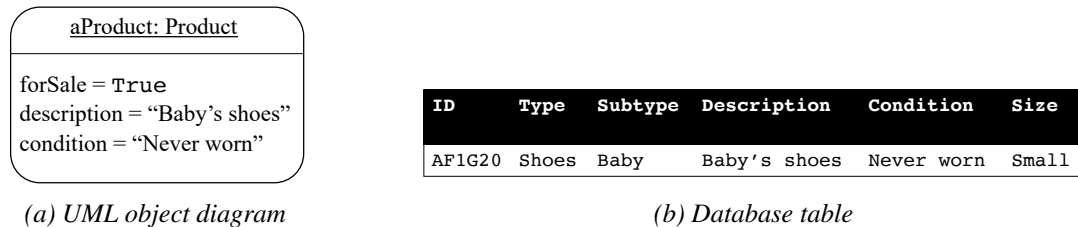


Figure 1 – UML object diagram and database table for the sale of a product

And we can, of course, revise the models. We might change the `condition` attribute to become an enumerated variable, perhaps using the value of 1 to represent the condition of “Never worn.” An enumerated variable would help us implement other features, e.g., with an SQL database, it becomes easier to select products that are never worn, e.g., `SELECT * WHERE Condition == 1`. An enumerated variable also improves the efficiency of computational processes, e.g., an `if` test of a numeric variable requires less computational resource than an equivalent test of a string value.

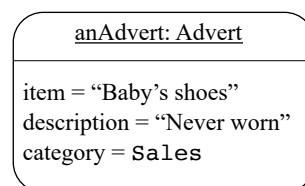


Figure 2 – A UML object diagram for an item to advertise

The object model presented in Figure 1 is not the only model that might be constructed. Figure 2 suggests a different object model. Comparing Figure 1 and Figure 2, the same *text* is present but the way we have *structured* the text is different. Our interpretation of the six words leads to different computational representations and those representations then support different thinking, e.g., we might enumerate the `category` attribute and, with the appropriate database design, select only the sale adverts: the SQL statement, `SELECT * WHERE category == 1`, now means something different. Though the model has changed, the fundamental nature of the model hasn’t, and the kind of thinking – computational thinking – hasn’t changed either.

The UML examples also help to illustrate that one feature removed from the story is not the literal words but the *order* to the words. Removing the order “dismantles” the story. Haven (2007) presents an example to illustrate how simply varying the placement of a word effects the meaning of a sentence. Compare the following examples, taken from Haven’s book, *Story Proof* (Haven, 2007):

John will marry Elise.
Even John will marry Elise.
 John will *even* marry Elise.
 John will marry *even* Elise.

Notice not just that the meaning of the sentence changes but also, if we dwell on each sentence, we might start to wonder about the context and the motivation for John, e.g., what might be going on for John to marry *even* Elise?

As well as “dismantling” the story, removing the order of the words so as to make a computable representation also ‘de-means’ – i.e., reduces the very meaning of – the story for a human whilst, conversely, formalising the representation to enable computation. At the same time, details are *added* to the computational model, e.g., data types, methods, class-object structures.

4. Second illustrative example: the Byzantine Generals Problem

Removing information in order to enable computation can be illustrated through another story, the Byzantine Generals Problem (BGP). There are several versions of the BGP reported in the computing literature. We take what is perhaps the most commonly-cited version, published within the safety engineering community by Lamport, Shostak, and Pease (1982).

... several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, **they must decide upon a common plan of action**. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that the following two conditions are met:

#1: All loyal generals **decide on the same plan of action** [...]

#2: A small number of traitors cannot cause the loyal generals to adopt a bad plan.

(emboldened emphasises added)

The BGP was formulated by computer scientists to illustrate a particular *computational* problem, i.e., ensuring reliable communication in the presence of faulty components. In a previous publication, we (Menon & Rainer, 2021) critically evaluated the BGP story, concluding that the story ‘fails’ *as a story*. For example, the story lacks any *humanly-meaningful* objective. Generals are expected to agree on a common plan of action, but it does not matter on what they agree. They can attack or retreat or, in principle, do anything else they agree on. Their objective is arbitrary. This is odd for a story because usually in a story a character would have some motive for their objective.

It is conceivable that, for some stories, there are characters who just want everyone to agree, but don’t care what they agree on. A good example is a waiter trying to take an order for a shared pizza from a family of four. But in this example, there are other characters – those in the family – who do want particular outcomes. And also, generals arguing over strategy don’t fit this kind of story.

To make a more effective story for the reader (a *reader*, not a software engineer or a programmer or, more generally, a computational-thinker), what is missing from the sentence is a final clause, i.e., decide to do what? Furthermore, since the loyal generals’ objective is arbitrary, the traitors’ objectives are also arbitrary: the traitors are only concerned with preventing an arbitrary legitimate agreement. But note too that not only must the traitors’ objectives be arbitrary relative to the loyal generals, *each* of the traitor’s objectives must be arbitrary relative to all other traitors. In this context, each traitor might be better understood as an agent of chaos.

The BGP therefore *risks* misrepresenting the computational problem through the way it presents a story in terms of human agents. To prevent this risk, to “square the circle” between story-thinking and computational-thinking, the human agents in the BGP are given odd (for a human) intention. Safety engineers possess the technical knowledge needed to interpret the story “correctly” *for the computational problem* being explored. For safety engineers, as computational-thinkers, the arbitrariness of the objectives is not just acceptable but essential. This is because the arbitrariness of the objectives allows for an algorithmic solution that has wide applicability: the algorithmic solution would apply for situations where generals agree to attack and for situations where generals agree to retreat and for situations where generals (arbitrarily) agree to do something else. The BGP abstracts the problem so as to provide a generalised solution (compare with the abstraction of User Story 2 and 3 to Use Story 1, in Table 1). And in abstracting the problem it must necessarily remove a humanly-meaningful quality of the story, i.e., meaningful human intention.

Overall then, the BGP acts as a kind of mirror example to the six-word story. To go from the six-word story to the computational representation, we remove something essential. To go from the computational problem to a BGP *story* we would need to add something essential, i.e., consistent characterisation and motivation, and the failure to do so contributes to its failure as a story. In contrasting ways, both neglect. The BGP is not *meaningless* – we can still understand the story – but it is not *meaningful*, in human terms, as a “good” story.

5. Software engineering: programming-in-the-large

For conciseness, we use Johnson and Ekstedt’s (2016) Tarpit Theory of Software Engineering as our reference for discussing the nature of software engineering. As part of the summary of their theory, Johnson and Ekstedt write, “The goal of software development is to create programs that, when executed by a computer, result in behavior that is of utility to some stakeholder.”

Drawing on the discussion of computational thinking, we might say that software is *useful* – of utility – to a stakeholder when the software behaves in a way that solves a *representation* of a problem experienced by that stakeholder. Solving a representation of a problem is not the same as solving the problem. And inferring from Priami’s (2007) assertion, in Section 1, as well as from our discussion of the BGP, some problems must be *neglected* in order to make the resulting model computable.

Johnson and Ekstedt (2016) also write that, “Much of software engineering concerns translations; design specifications are translated into source code. . . , source code is translated into machine code. . . , etc. In fact, the whole process of software engineering can be considered as a series of translations. . . (Johnson and Ekstedt (2016), p. 187). They define *language* as “A set of specifications”, *translation* as, “An activity that preserves the semantic equivalence between source and target languages [specifications],” and *semantic equivalence* as, “Two specifications are semantically equivalent if, when translated to a common language, they are syntactically identical.” (Notice how *semantic equivalence* is being defined in terms of *syntactic identity*.)

We have shown, with the six-word story and the BGP story, that it is not necessarily feasible to translate the story from the source language to a target language *and* maintain semantic equivalence. More than that, it is unreasonable to *expect* a translation, in the way that Johnson and Ekstedt define it, from the English language of the six-word story to, as examples, the User Story, the object diagram or the database table. But that is the point. The language used with story-thinking, and therefore the representations used for story-thinking, are not semantically equivalent to the language and representations used for computational-thinking.

Johnson and Ekstedt (2016) seem to recognise this problem of non-equivalence, when they write:

This leads us to *the major challenge of software engineering*, that programs – these formal, static, syntactic compositions – are very different from the oftentimes **fluid and intangible stakeholder experiences** they are intended to **evoke**. A significant feat of **imagination** is thus required on the part of the developers to predict the effects of a given syntactic modification to the program code on the end-users’s experiences. An even greater challenge, which we propose as the *core task of software* endeavors, is **to determine which syntactic manipulation will cause a specified stakeholder experience**, and then to perform the appropriate informed manipulations, i.e. to make design decisions.

(In the original text, the *san serif* typeface denotes formal constructs of the theory, whilst the *italicised* text are the original authors’ emphasise. **Emboldened** emphasis are ours. We recognise these variations in typeface and font make the text more confusing, visually, but retain them to be faithful to the original.)

Clearly, Johnson and Ekstedt (2016) are aware of the “gap” between stakeholders’ experiences and executable programs, and of the challenge of bridging this gap.

6. A summary of the problems and suggested ways forward

Drawing on the preceding discussion, we summarise one specific problem, briefly suggest ways in which this problem might be tackled, and briefly describe ongoing investigations we are conducting in this area.

6.1. The problem of neglectful representations

As Priami (2007) observes, computational thinking neglects. More than that, computational thinking must neglect. It does this in order to arrive at a representation that is computable. These “neglectful representations” inevitably “de-mean”, i.e., reduce humanly-meaningful qualities. Neglectful representations have implications for the impact of computational-thinking, programming and software engineering on society, the economy and the environment. For example, economic impact is much easier to align with computational-thinking because economic thinking appears to be a way of thinking similar in kind to computational-thinking. But humanly-meaningful qualities of society, as well as the qualities of the environment, are much harder to represent, a point that is increasingly recognised, e.g., with work on values in computing (Ferrario et al., 2017), kind computing (Alrimawi & Nuseibeh, 2022), compassionate computing (Pomputius, 2020) and responsible software engineering (Schieferdecker, 2020).

This problem of neglectful representation and, by comparison, the complementary use of story as a possible solution, appear to be implicitly recognised by others in software engineering. Strøm (2006, 2007) investigates the role and value of stories that include emotions and conflicts in software engineering. Bailin introduced design stories (Bailin, 2003) and also discussed how features need stories to convey the “missing semantics” that address “fine-grained questions of context, interface, function, performance, and rationale” (Bailin, 2009). In an unpublished paper, Stubblefield et al. (2002) observe that, “In our experience, most software development projects do not fail for technical reasons. They fail because they do not engage users at the fundamental level of value, meaning and practice. They solve the wrong problem, or fail to support deeper patterns of work and social interaction.” (Stubblefield & Rogers, 2002). Finally, Yilmaz et al. (2016) report a preliminary study to demonstrate the benefits of story-driven software development: “... it is important to capture and store the excessively valuable tacit knowledge using a rich story-based approach.”

6.2. Suggested ways forward

We suggest two directions for future research:

1. Developing methodology that connects stories with algorithms. In a previous paper, Rainer (2017), drawing on prior work in law and legal reasoning, proposed a methodology for identifying arguments and stories from blog articles, extracting them, and then graphically combining them. This methodology might be extended or, alternatively, might provide an example for a potential methodology to integrate representations used for story-thinking with representations used for computational-thinking.
2. Changing software practice. With the agile methodology, software practitioners return to the user story, e.g., its acceptance criteria, to evaluate whether a user story has been delivered. We suggest that practitioners might return to the *story*, and not just the user story. As one example, a software engineering (SE) team developing a software system for managing information about children in publicly-funded care homes might analyse Sissay’s memoir, *My Life is Why* (Sissay, 2019), to gain insights into the experiences of children in care, and therefore into the meaningful user stories for such a persona or stakeholder.

6.3. Ongoing investigations

We have four ongoing investigations in this area:

1. A closer study of the Byzantine General Problem, in which the Problem is represented as user stories and one then attempts to implement those user stories in executable code.

2. A study of Sissay's memoir, *My Life is Why* (Sissay, 2019), in which, again, the memoir is represented as user stories, as well as other requirements, and one then attempts to implement the user stories into database designs and then executable code.
3. A study of how writers think about theirs, and others', short stories, and how formal techniques from software engineering might support their thinking.
4. A study of how GPT-3 responds to prompts about stories such as the six-word story.

7. Conclusion

In this paper we explored two modes of thinking: story-thinking and computational-thinking. Using two examples – a six-word story and the Byzantine Generals Problem – we show how story-thinking and computational-thinking attend to these stories in different ways. We considered how software engineering, as programming-in-the-large, recognises, at least implicitly, these different modes of thought, as well as the challenges of integrating these modes. We identified the problem of neglectful representations, briefly suggested ways in which these problems might be tackled, and briefly summarised our ongoing investigations.

8. Acknowledgements

We thank the reviewers and the conference attendees for their constructive and supportive feedback. We also thank participants of the online presentation for permission to quote them, and the students of the brainstorming session.

9. References

- Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal*, 55(7), 832–835.
- Alrimawi, F., & Nuseibeh, B. (2022). Kind computing.
- Bailin, S. C. (2003). Diagrams and design stories. *Machine Graphics and Vision*, 12(1), 17–38.
- Bailin, S. C. (2009). Features need stories. In *International conference on software reuse* (pp. 51–64).
- Carroll, J. M. (2003). *Making use: scenario-based design of human-computer interactions*. MIT press.
- Cohn, M. (2004). *User stories applied: For agile software development*. Addison-Wesley Professional.
- Denning, P. J. (2009). The profession of it beyond computational thinking. *Communications of the ACM*, 52(6), 28–30.
- Erwig, M. (2017). *Once upon an algorithm: how stories explain computing*. MIT Press.
- Ferrario, M. A., Simm, W., Whittle, J., Frauenberger, C., Fitzpatrick, G., & Purgathofer, P. (2017). Values in computing. In *Proceedings of the 2017 chi conference extended abstracts on human factors in computing systems* (pp. 660–667).
- Haven, K. (2007). *Story proof: The science behind the startling power of story*. Greenwood Publishing Group.
- Heineman, G. T., Pollice, G., & Selkow, S. (2008). *Algorithms in a nutshell*. O'Reilly Media.
- Johnson, P., & Ekstedt, M. (2016). The tarpit—a general theory of software engineering. *Information and Software Technology*, 70, 181–203.
- Lamport, L., Shostak, R., & Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 382–401.
- Lucassen, G., Dalpiaz, F., Van Der Werf, J. M. E., & Brinkkemper, S. (2015). Forging high-quality user stories: towards a discipline for agile requirements. In *2015 ieee 23rd international requirements engineering conference (re)* (pp. 126–135).
- Lucassen, G., van de Keuken, M., Dalpiaz, F., Brinkkemper, S., Sloof, G. W., & Schlingmann, J. (2018). Jobs-to-be-done oriented requirements engineering: a method for defining job stories. In *International working conference on requirements engineering: Foundation for software quality* (pp. 227–243).
- Menon, C., & Rainer, A. (2021). Stories and narratives in safety engineering. In *Proceedings of the 30th safety critical systems symposium, volume: Scsc-170*. Retrieved from <https://scsc.uk/scsc-170>

- Pomputius, A. (2020). Compassionate computing in the time of covid-19: Interview with laurie n. taylor. *Medical Reference Services Quarterly*, 39(4), 399–405.
- Priami, C. (2007). Computational thinking in biology. In *Transactions on computational systems biology* viii (pp. 63–76). Springer.
- Rainer, A. (2017). Using argumentation theory to analyse software practitioners' defeasible evidence, inference and belief. *Information and Software Technology*, 87, 62–80.
- Schieferdecker, I. (2020). Responsible software engineering. In *The future of software quality assurance* (pp. 137–146). Springer, Cham.
- Sissay, L. (2019). *My name is why*. Canongate Books.
- Starfield, A. M., Smith, K. A., & Bleloch, A. L. (1994). *How to model it: Problem solving for the computer age*. Interaction Book Company.
- Strøm, G. (2006). The reader creates a personal meaning: A comparative study of scenarios and human-centred stories. In *People and computers xix—the bigger picture* (pp. 53–68). Springer.
- Strøm, G. (2007). Stories with emotions and conflicts drive development of better interactions in industrial software projects. In *Proceedings of the 19th australasian conference on computer-human interaction: Entertaining user interfaces* (pp. 115–121).
- Stubblefield, W. A., & Rogers, K. S. (2002). *The micro traveler and the hero's journey*. Retrieved from <https://wmstubblefield.com/wp-content/uploads/2019/06/narrativePaper.pdf>
- Worley, P. (2014). *Once up an if: the storythinking handbook*. Bloomsbury.
- Yilmaz, M., Atasoy, B., O'Connor, R. V., Martens, J.-B., & Clarke, P. (2016). Software developer's journey. In *European conference on software process improvement* (pp. 203–211).

Interactive Bayesian Probability for Learning in Diverse Populations

Zainab Attahiru
Computer Laboratory
University of Cambridge
zia21@cantab.ac.uk

Rowan Hall Maudslay
Computer Laboratory
University of Cambridge
rh635@cam.ac.uk

Alan F. Blackwell
Computer Laboratory
University of Cambridge
afb21@cam.ac.uk

Abstract

This paper builds on work presented at PPIG 2021, “Visualising Bayesian Probability in the Kalahari” (Blackwell et al., 2021). Here, we describe a proposed interface for visualising Bayesian probability in a non-Western educational setting—in this case, a school in Nigeria. We evaluate the approach in a school workshop combining didactic and discovery methods of learning. The results offer insights for future curricula in high-school level probability, as well as an agenda for Computer Science education research, exploring how a programming perspective (through model building) might improve reasoning and intuition about probability.

1. Introduction

A paper at PPIG 2019 set out an agenda for the usability of probabilistic programming languages, including a “furthest-first” agenda that would develop new languages for use in schools of Sub-Saharan Africa (Blackwell et al., 2019). At PPIG 2021, a progress report on that programme of work described an investigation toward visualising Bayesian probability in the Kalahari (Blackwell et al., 2021)

In this paper, we extend that programme of work with a report on visualising Bayesian models in Nigeria, with education and learning as primary considerations in the design process. This builds from related work that have attempted to simplify and improve probabilistic reasoning using visualisations by employing storytelling structures (Erwig & Walkingshaw, 2013), including diagrams in live programming environments (Gorinova, 2015), and incorporating causality in the language of probability and statistics (Pearl, 1995). Pearl’s work in particular, provides the foundation of the visualisation presented in this paper.

This work is motivated by previous work in computer science education research that has explored the efficacy of diagrams and storyboards in learning (Waite et al., 2016) as well as the success of visual languages such as Scratch (Resnick et al., 2009) and Sonic Pi (Aaron & Blackwell, 2013) as first programming languages. Our work investigates the potential of diagrams and graphical objects to produce equally successful outcomes in other areas such as probability education in non-Western contexts.

2. Background

2.1. Visualisations in Child Learning

Exploring external representations as a learning methodology for children goes back to education pioneers such as Froebel in the 19th century (Manches et al., 2010). Piaget for example, highlights the importance of the manipulation of concrete objects in learning via his constructivist theory (Von Glasersfeld, 1982). Papert (1980), inspired by Piaget, developed a theory of constructionism where he argues that these manipulable objects provide an effective learning tool as they are embedded in a child’s cultural environment. Papert’s insights have been influential in later works including those concerned with the development of digital interfaces and development environments for kids (Resnick et al., 2009; Stead, 2016). Scratch (Resnick et al., 2009) for instance, uses graphical objects in the form of legos as constructs to allow younger audiences explore programming and art creativity. This has been a successful endeavour boasting a record of 42 million users in 2021. Manches et al. (2010) have posited that the success of external representations is attributable to their ability to offload cognition by mapping processes whilst providing conceptual metaphors that allow children to relate concepts to real world experiences. This helps with knowledge transfer, an important design goal for building educational tools

(Stead, 2016).

Despite the success of Scratch, the use of legos as the foundation of the graphical representation signifies Western preferences. Thus, the transferability of the success of such visual representations to kids that are not exposed to Western culture is unknown. This provides an avenue for exploring the effect of external representations on children from varying backgrounds and the possibility of a “general” visualisation that could be discovered from a non-Western culture.

2.2. Explaining Probabilistic Models

The ubiquity of intelligent tools raises questions relating to user interaction. User in this context is not limited to the scope of the generic end-user but includes other stakeholders such as programmers and system designers. A prominent design approach to this problem has been to improve the ‘explainability’ of these tools by helping users build accurate mental models of system behaviour. As the underlying models of these intelligent tools are probabilistic in practice, some of the work done has sought to explain the inference process in probabilistic programs.

Erwig & Walkingshaw (2013) for example, designed a visual metaphor based on the causal structure of stories to breakdown the stages of a probabilistic modelling process. This representation serves as an explanation that maps the discrete values of a distribution to the probability of their outcome. The authors use the Monty Hall problem to buttress the intuitive nature of their representation and further provide insights into how their work could improve certain programmer-based tasks such as debugging. A related work has defined and discussed debugging principles that could guide the design of explanations (Kulesza et al., 2015). Gorinova (2015) on the other hand focused on how visual explanations could aid the probabilistic programming process. The author developed a live multiple representation development environment (MRE) that shows the Bayesian network of variables based on an infer.NET probabilistic model. Participants using the MRE were found to provide high-level descriptions of underlying probabilistic dependencies compared to those using a conventional development environment.

This has led to a promising view of graphical visualisations as an effective means of improving probabilistic reasoning. Blackwell et al. (2021) have especially discussed the potential of interactive visualisations for teaching probability in schools whilst looking at a furthest-first design outlook that is inclusive of underserved communities.

2.3. Causal Inference

Causal thinking is a cognitive principle used by humans and its deductive nature plays a significant role in scientific thought and discovery. The domain of statistics and probability, as exemplified by its popular mantra “Correlation does not imply Causation” discounts this principle, resulting in non-intuitive language for communicating concepts and paradoxes. The science of Causal Inference propounded by Pearl (1995) seeks to provide tools that mathematise the concept of causality by building on current statistical methods. This approach is gaining support in the AI community as it provides a pathway to solving learning problems using less data as well as tackling ethical problems in algorithmic design such as fairness (Kusner et al., 2017).

Pearl asserts that a purely statistical approach to AI is inadequate as it neglects the causal assumptions and beliefs that are necessary for producing intelligent behaviour. This leads him to formulate a hierarchical division of intelligence referred to as the Ladder of Causation. Each rung of the ladder has additional expressiveness compared to the rung below it, which determines the type of questions it can answer (Bareinboim et al., 2022). The rungs and their assessment of likelihood of outcomes are presented below;

- Association: The likelihood of an outcome is assessed by observed evidence. Most of traditional statistics and data-only machine learning solutions lie in this rung. An example of a question on this rung is “What is the likelihood that a toddler will be vaccinated?”.
- Intervention: The likelihood of an outcome is assessed by introducing an intervention and mea-

suring its effects. This is not the same as an assessment by conditioning on an observed value of a variable but rather by a forced manipulation of the variable. This is particularly useful in healthcare research where randomised trials may not be feasible or are ethically sensitive (Zhang et al., 2021). An example of a question on this rung is “How many people will die if the vaccine dose is doubled?”.

- **Counterfactual:** The likelihood of an outcome is assessed based on a hypothetical scenario. The rung alludes to imagining and Pearl argues that imagining is responsible for human intelligence and thus essential for achieving artificial general intelligence. AI research has employed counterfactuals to proffer solutions in computational advertising (Bottou et al., 2013) and algorithmic fairness (Kusner et al., 2017). An example of a question on this rung is “What will happen if a person who took a vaccine decides not to take it?”.

Causal Inference as a technical discipline specifically deals with answering interventional and counterfactual questions. To effectively do this, one has to be able to represent what they know (assumptions) and what they want to know (queries). Causal inference provides tools for representing these knowledge. Assumptions are expressed using two methods; mathematically using Structural Causal Models (SCMs) and diagrammatically using Causal Diagrams. These representations are complementary. Queries are expressed using Do-Calculus. This paper will be focused on providing a brief overview of SCMs and Causal Diagrams. For an in-depth coverage of Do-Calculus, refer to Glymour et al. (2016).

2.3.1. Structural Causal Models

When defining SCMs, one has to be cognisant of the two variable types in causal modelling; exogenous and endogenous variables. Exogenous variables are background variables whose causes are determined by factors outside of the model. Endogenous variables are those variables whose causes or values are known and defined within the model. Exogenous variables cannot be descendants of other variables as their causes are unknown while endogenous variables can be descendants of exogenous or endogenous or both types of variables. The relationship between child and parent variables are mapped by functions. This in turn induces strong assumptions about the nature of the relationship but provides a convenient syntax for how the distribution of these variables will change in response to external interventions (Pearl, 1995).

Definition 2.1 *A causal model can be defined as a triple of sets $\langle \mathcal{U}, \mathcal{V}, \mathcal{F} \rangle$, where \mathcal{V} is a set of endogenous (observed) variables, \mathcal{U} is a set of exogenous (latent) variables, and \mathcal{F} is a set of functions (each one corresponding to a variable $V_i \in \mathcal{V}$), such that*

$$V_i = f(pa_i, U_{pa_i})$$

where $f \in \mathcal{F}$, $pa_i \subseteq \mathcal{V}$ are the parents of V_i , and $U_{pa_i} \subseteq \mathcal{U}$ are some independent noise variables and can be identified as the “causes” of V_i .

2.3.2. Causal Diagrams

These are directed acyclic graph visualisations of SCMs. Solid nodes in the graph structure represent endogenous variables while hollow nodes represent exogenous variables. The functional relationship between variables are represented using directed edges that connect variables. Causal diagrams aid in answering causal queries and computing causal effects in the absence of empirical data (Forney & Mueller, 2021). In addition, they provide an appropriate visual for structures that could result in statistical paradoxes such as confounding, mediation and collision.

Causal diagrams bear a close resemblance to Bayesian networks and are indeed built from the same foundation. By employing a causal structure in their construction, we propose that these diagrams could provide an alternative visualisation for representing and explaining Bayesian probabilistic models that is closer to human causal thinking. The abstract form of the graph provides a more universal visual approach as opposed to established norms of dices and coins in probability explanations that could have cultural connotations. In addition, the basic elements of causal diagrams - nodes and directed arrows

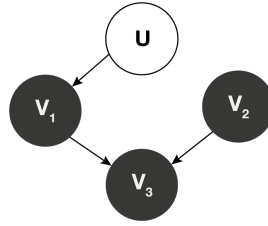


Figure 1 – A simple model represented as a causal diagram

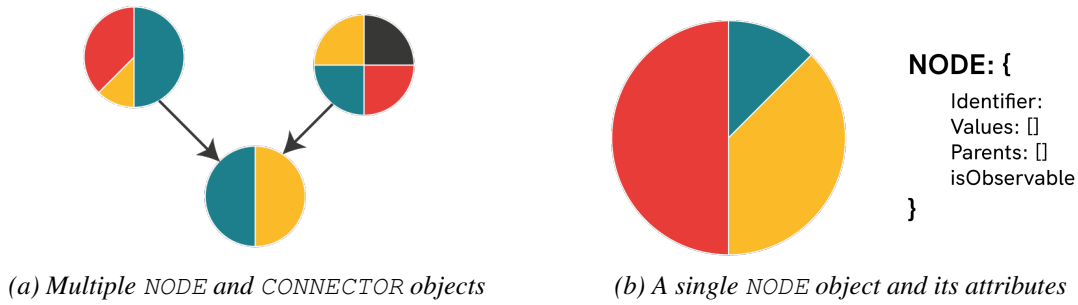


Figure 2 – A visual notation for causality

- provide basic building blocks that are similar to those available in graphics software such as Adobe Illustrator. These can be converted to manipulable graphical objects that could transform the model building process into an interactive experience.

3. Work

3.1. A Modified Visual of Causal Diagrams

Building on our decision to adopt causal diagrams to visualise Bayesian probability, we modify the graph to accommodate this objective. In our modified visual formalisation (as shown in Figure 5), we treat both endogenous and exogenous variables as a single graphical object, *NODE*. This represents any random variable of a typed discrete finite probability distribution. The visualisation is in the form of a pie chart, with sectors of the chart representing value-likelihood pairs. Functions are represented using a *CONNECTOR* object, a directed arrow between nodes similar to Figure 1 (see Figure 2a).

A *NODE* is defined using four attributes (as shown in JSON in Figure 2b right) : an identifier, values, parents, and a flag which marks whether it is observable. Below is a breakdown of what each of these attributes represents:

- **Identifier:** The identifier holds a string name assigned to a random variable. In basic formulations, the identifier could be prove to be risky for users as it places a variable naming burden. This could be simplified by including predefined options in the naming interface or alternatively, by inferring the variable name from the content of the values attribute as done in spreadsheets (Sarkar et al., 2022).
- **Values:** The values attribute holds the different values that a random variable can take: it is an array of tuples, each containing a name attribute (of the value) paired with a likelihood attribute.¹ In the visualisation (Figure 2b), each value corresponds to a sector of the pie-chart. The likelihood attribute specifies how likely the user assesses that an outcome is to occur and therefore holds their prior belief about the random variable. This is represented using floating point numbers (rounded to two decimal places).

In defining this attribute, a burden is placed on users by prompting them to identify possible out-

¹Since we are dealing with discrete probability distributions, these are countable.

comes of a random variable and their likelihood. This induces a form of premature commitment and could be classified as a hard cognitive task in instances where users don't have a solid understanding of what a random variable constitutes.²

- **Parents:** This attribute holds the parent variables of a random variable: it is an array of tuples, each containing a parent's identifier and an array specifying the effect of the parent on the variable's distribution. This defines the relationship between child and parent variables, and is represented visually using the `CONNECTOR` graphical object (a directed arrow, see Figure 2a).
- **isObservable:** This attribute specifies the latency of a random variable and is used to determine which variables are exogenous and which are endogenous. This allows us to represent both variable types using the `NODE` object. Setting this attribute to false places a restriction on adding parent nodes, since exogenous variables' causes are unknown (owing to the nature of the variable or the modeller's choice).

3.2. An Interface for Defining the Diagrams

To produce the visual formalisation presented in the previous section, we sketch out the design of interactive interface components that can be used to create expressions of causal dependencies. This interface induces the following interaction flow: declare variable, specify value distribution, and define relationships between variables.

3.2.1. Declaring Variables

Figure 3a shows a variable declaration component used to create “pseudo” `NODE` objects (this only specifies the identifier and observability; values and parents are left to later components). The observability of the variable is posed as a question “Can you observe it”. An earlier iteration considered the question “Can you see it” as a more natural form, but this proved to be an oversimplified and narrow depiction of observation. To accommodate for the more complex idea of observation, a tooltip is included to express the different ways of observing.

In our discussion on the identifier and values attributes (§3.1), we mentioned the potential cognitive strain imposed on the user during naming. While we referred to selecting from a predefined set of variables (based on a particular domain of application such as health as per the presented design) and inferring from the values attributes as mitigations, another alternative could be the use of the “Variable Description” field to serve as prompt for the variable's identifier (denoted as “Variable” in the presented design). This serves as a guide and teaching aid whilst allowing users to retain a level of autonomy in determining what variable they want to represent.

3.2.2. Specifying Value Distributions

Figure 3b presents the distribution builder component used specify the values of a variable while ensuring that the sum of their likelihoods is equal to one. Distributions are defined from selecting between binary and range options and value names can be edited based on the user's preferences. The value names in Figure 3b serve as placeholder text and a simple explanation of how value names can be defined. The distribution of the values is represented using bar charts with draggable bars. The length of each bar corresponds to a value's likelihood. Likelihood is assigned by dragging the bar which provides both a spatial visualisation and relative depiction (to other values). Additionally, likelihoods are presented in percentages to provide an alternative representation for understanding the concept of likelihood.

Each bar is responsive to changing values in the other bars (dragging a bar results in a change of value in other bars) as a means of enforcing the sum rule of discrete probability distributions. This provides a convenient modal of interaction for binary outcomes but could be cumbersome for distributions with three or more values. This can be bypassed by disabling the responsive dragging and performing validation checks before saving. Additionally, values can be added and removed. A constraint of the number of values allowed per variable can be reinforced to allow for clarity in visualisation.

²This could once again be simplified using predefined options.

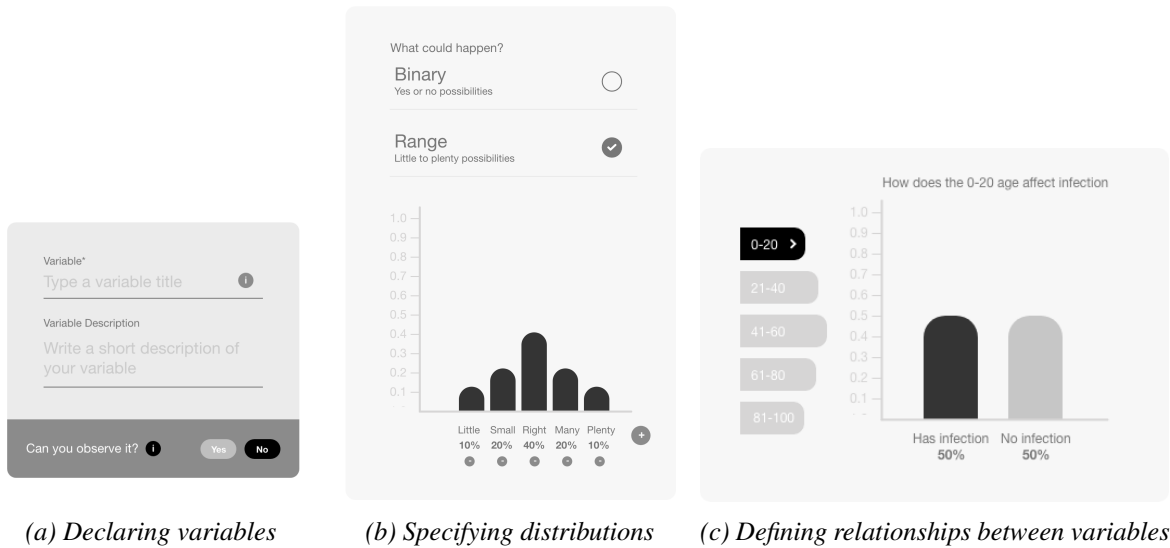


Figure 3 – Prototypical UI components

3.2.3. Defining Relationships

Defining relationships between parent and child variables in causal diagrams is a difficult task and requires assumptions based on knowledge of probability distributions such as normal, poisson or binomial distributions. As our objective is to employ a graphical approach suitable for younger users, we define these relationships not as functions between distributions but rather by conditioning.

In the relationship builder component (Figure 3c), we combine two bar chart representations with different interaction constraints. Each horizontal bar (to the left) represents a parent variable's values with their width corresponding to their likelihood. These bars are selectable and selecting a bar re-renders the child variable's values (denoted by the vertical bar chart). These values are vertical draggable bars that can be used to specify the effect of a parent variable's value on the child variable's values. The text with the following template - "How does [parent variable's value] affect [child variable]" reinforces the causal notion that the parent is a determinant factor in the values of the child.

The component's construction on only defining the effect of a single parent results in the marginalisation of other parent variables. When specifying prior distributions, this is adequate. However, when posterior distributions are specified (in the form of edits), this could lead to an inference problem.

4. Workshop

To determine the efficacy of our modified visual of causal diagrams as a tool for explaining Bayesian probabilistic models, we performed a short user study with secondary school students in Nigeria.

4.1. Study Design

A web application consisting of four separate modules, representing four tasks was used as the study environment. Rather than tackle the question of the explainability of our modified causal diagrams directly, we opted to answer the more concrete question of measuring the influence of the visualisation on the participant's decision making. This goal was broken down into the following research questions:

1. Are secondary school students in Nigeria using Bayesian reasoning in their decisions?
2. Can Bayesian reasoning be introduced through an interaction with the modified causal diagram?

With these questions, we designed the study as follows - the first and fourth tasks were decision-making tests while the second and third tasks served as a short lesson and tool interaction session respectively. By comparing the results from the decision-making tests, we can investigate the effect of the visualisation on the answers provided by the participants.

To determine what type of questions will feature in the tests, we considered domains of application that will be culturally relevant to the participant group. For the questions in first decision-making test as well as the content of short lesson, we chose the COVID-19 pandemic and further narrowed the scope to a “COVID in the Market” scenario. The global impact of the pandemic and its depictions within public discourse provides a familiar space where participants can reason about events and their causal structure. For the second decision-making test, we opted for a more localised domain - tree climbing.

The inclusion of a short lesson as opposed to direct exposure (and exploration) of the visualisation is intended to create a structure that is familiar to the participant group, who are conversant with learning environments that have a didactic component in contrast to their Western counterparts. Sfard (1998) has also asserted that a mixture of both didactic and discovery methods are essential in learning.

4.2. Participants

Eight participants were recruited from Government Secondary School Wuye in Abuja, Nigeria. Their prior exposure to probability is based on rote method computation of mutually exclusive events as opposed to extensive understanding of concepts such as randomness and conditioning.

4.3. Procedure

We used a within-subjects experiment design for the study. The study lasted for approximately an hour. Each participant was provided with a desktop computer to access the study environment online. The participants were required to complete the four tasks in order. During the first and fourth tasks, participants were not allowed to interact with each other in order to preserve the validity of the answers they provide. The second and third tasks were in the form of an open engagement session. Below is a breakdown of the details of each task;

Participants completed four tasks, detailed below. In the first and last tasks, students make decisions in hypothetical scenarios. In the middle tasks, they are introduced to Bayesian reasoning. The aim is to observe the influence of introducing the notions of randomisation and conditioning on the certainty of the decisions made by the participants.

1. Task A: Participants were required to make decisions based on three hypothetical scenarios placed within the “COVID in the Market” domain. The questions include:
 - (a) *The market stall in your neighbourhood was just reopened since the COVID pandemic started. Will you go to buy sweets or not?*
 - (b) *Your friend has eyes that can see COVID. She went to the stall at 10 in the morning and said half the people she saw had COVID. Will you still go at 2 in the afternoon?*
 - (c) *It has been three (3) days since your friend went to the stall and she seems fine. Will you go to the stall today?*
2. Task B: Participants read through ten short explanations (including both textual and visual imagery) on Bayesian reasoning and causal diagrams.
3. Task C: Participants interact with a tool based on our modified causal diagram visualisation with emphasis on the NODE object. This was achieved by presenting a predefined variable representation and allowing students to run sampling simulations. Interactive discussions during the session prompted students to reason about changing distributions of random variables when placed in different contexts (for example when a background variable changes). The complete version of the Figure 3 was not developed for this study.
4. Task D: Participants were required to make decisions based on three hypothetical scenarios in a tree climbing scene. The questions include:
 - (a) *You belong to a group of people who love to climb one particular tree. You have been doing this for the past two years. How likely are you to break your leg from climbing a tree?*

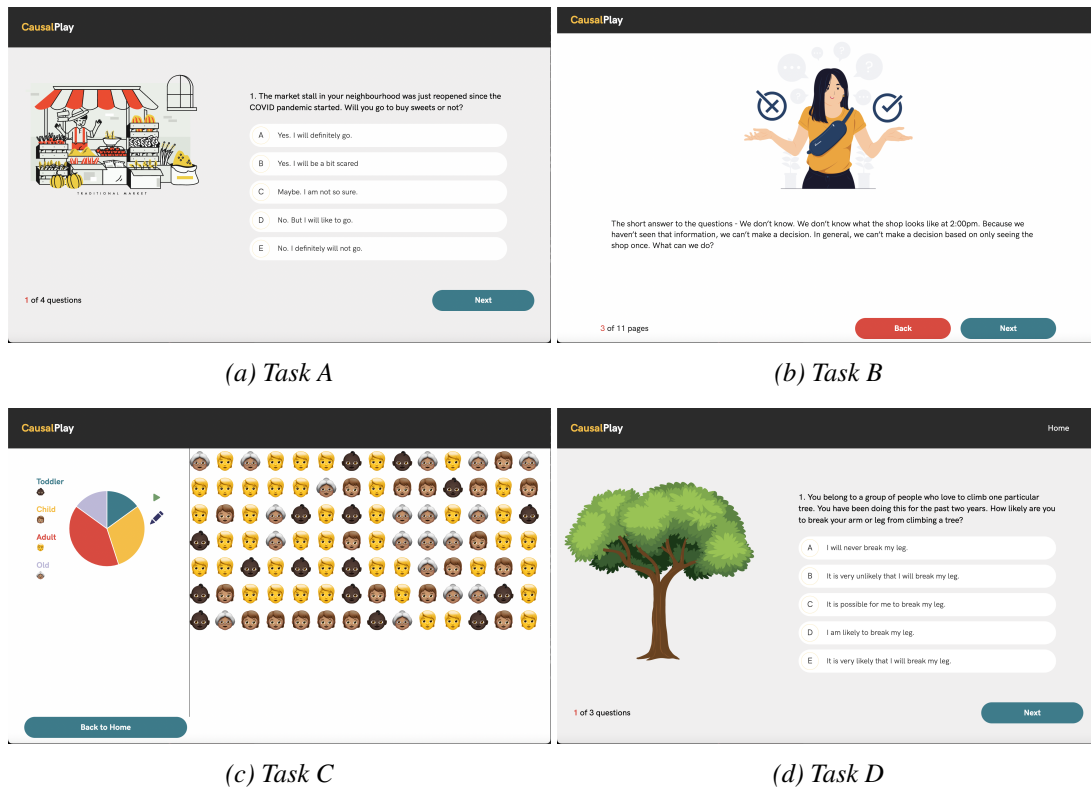


Figure 4 – Study Environment Screenshots

- (b) *Your best friend recently broke both legs climbing this tree. Will you join the others to climb the tree tomorrow?*
- (c) *Fatima just moved into your neighbourhood. She can run very fast. How good will she be at climbing trees?*

4.4. Findings and Discussion

The responses to the two decision-making tasks were collected and analysed. For each question, participants were required to give a response from five options. See Figure 5a for the map of a participant's decisions during the tasks. The table below provides a scheme for each option;

Option	Content	Scale	Description
A	Yes, I will definitely go to the market	5	Positive
B	Yes, I will be scared	4	Semi-positive
C	I am not sure	3	Uncertain
D	No, but I want to go	2	Semi-skeptic
E	No, I will absolutely not go	1	Skeptic

The standard deviation of both sets of responses was calculated to determine their distribution in order to investigate the questions posed in the study's design. The expectation is that the employment of Bayesian reasoning will induce a normal distribution of responses with most responses in the uncertain region (corresponding to 3 on the scale). This is because the hypothetical scenarios provide little evidence to cause a decision shift towards the extreme ends of the range. We provide discussions to the study research questions below:

Are secondary school students in Nigeria using Bayesian reasoning in their decisions? The data shows that this is not the case. Responses to the first decision-making test were skewed to extreme ends of the

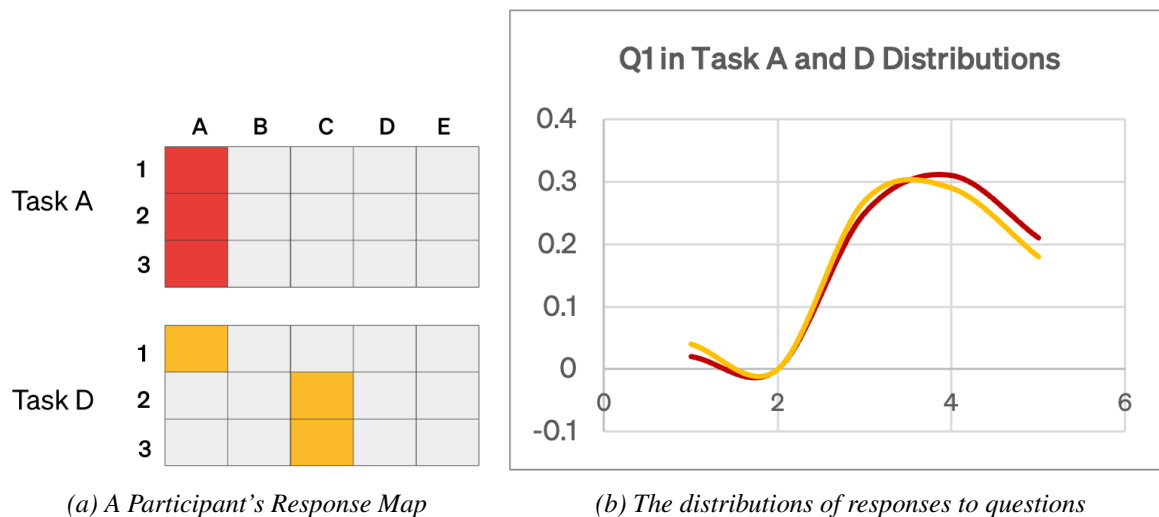


Figure 5 – Analysing study findings

range across all questions. In the second question for instance, the uncertain (“I and skeptic choice were not selected by any of the participants. However, this result is to not surprising. There is a wide range of research that has been undertaken that shows that people find it difficult to think in terms of probabilistic quantities.

Can Bayesian reasoning be introduced through an interaction with the modified causal diagram? The data once again shows that this is not the case. Similar to the first decision-making test, the second test’s responses are also skewed towards extreme ends. The distribution of the responses to the first questions in both tests exhibit a striking resemblance as depicted by Figure 5b. This result could have arisen due to size of the participant group, which represents a small (and perhaps biased) sample of the target population. Furthermore, research in learning are in the format of studies that measure effects over a longer period of time. The expectation that a short study could produce a similar effect is unrealistic.

These results raise new questions not the least of which are those surrounding the validity of the decision-making tests. One worth noting is how the effect of different biases on different decision scenarios could be taken into account when designing future user studies.

5. Conclusion and Future Work

In this paper, we argue that representations and demonstrations of Bayesian probability can be redesigned to tackle problems related to inclusivity in education. We highlight previous work undertaken on the use of visualisations in child learning and in representing probability models. This lead us to modify Pearl’s causal diagrams to include additional handles for specifying Bayesian probability models with interactivity as a key design consideration.

A study was conducted in Nigeria using a simplified form of the modification and the explainability of the visualisation was assessed via its effect on the participant’s decisions. Findings from this study were limited by a variety of factors including the scale of the software implementation, study length, and participant sample size. Future work on this project will focus on expanding the functionality of the software environment to allow for more dynamic model building, taking inspiration from graphics editing software such as Adobe Illustrator. This will be accompanied by longitudinal user studies to determine the efficacy of the visualisation and interaction environment as learning tools, as well as the selection of adequate metrics for measures of interest such as retention and knowledge transfer.

References

Aaron, S., & Blackwell, A. F. (2013). From sonic pi to overtone: Creative musical experiences with domain-specific and functional languages. In *Proceedings of the first acm sigplan workshop on func-*

- tional art, music, modeling amp; design* (pp. 35–46). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2505341.2505346> doi: 10.1145/2505341.2505346
- Bareinboim, E., Correa, J. D., Ibeling, D., & Icard, T. F. (2022). On pearl’s hierarchy and the foundations of causal inference. *Probabilistic and Causal Inference*.
- Blackwell, A. F., Bidwell, N. J., Arnold, H. L., Nqeisji, C., Kunta, K., & Ujakpa, M. M. (2021). Visualising bayesian probability in the kalahari. *Psychology of Programming Interest Group (PPIG)*.
- Blackwell, A. F., Church, L., Erwig, M., Geddes, J., Gordon, A., Gorinova, M., ... others (2019). Usability of probabilistic programming languages. *Psychology of Programming Interest Group (PPIG)*.
- Bottou, L., Peters, J., Quiñonero-Candela, J., Charles, D. X., Chickering, D. M., Portugaly, E., ... Snelson, E. (2013). Counterfactual reasoning and learning systems: The example of computational advertising. *Journal of Machine Learning Research*, 14(65), 3207–3260. Retrieved from <http://jmlr.org/papers/v14/bottou13a.html>
- Erwig, M., & Walkingshaw, E. (2013). A visual language for explaining probabilistic reasoning. *Journal of Visual Languages Computing*, 24(2), 88-109. Retrieved from <https://www.sciencedirect.com/science/article/pii/S1045926X13000025> doi: <https://doi.org/10.1016/j.jvlc.2013.01.001>
- Forney, A., & Mueller, S. (2021). Causal inference in ai education: A primer.
- Glymour, M., Pearl, J., & Jewell, N. P. (2016). *Causal inference in statistics: A primer*. John Wiley & Sons.
- Gorinova, M. I. (2015). Interactive development environment for probabilistic programming.
- Kulesza, T., Burnett, M., Wong, W.-K., & Stumpf, S. (2015). Principles of explanatory debugging to personalize interactive machine learning. In *Proceedings of the 20th international conference on intelligent user interfaces* (p. 126–137). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2678025.2701399> doi: 10.1145/2678025.2701399
- Kusner, M. J., Loftus, J., Russell, C., & Silva, R. (2017). Counterfactual fairness. In I. Guyon et al. (Eds.), *Advances in neural information processing systems* (Vol. 30). Curran Associates, Inc. Retrieved from <https://proceedings.neurips.cc/paper/2017/file/a486cd07e4ac3d270571622f4f316ec5-Paper.pdf>
- Manches, A., O’Malley, C., & Benford, S. (2010). The role of physical representations in solving number problems: A comparison of young children’s use of physical and virtual materials. *Computers Education*, 54(3), 622-640. Retrieved from <https://www.sciencedirect.com/science/article/pii/S0360131509002632> (Learning in Digital Worlds: Selected Contributions from the CAL 09 Conference) doi: <https://doi.org/10.1016/j.compedu.2009.09.023>
- Papert, S. A. (1980). *Mindstorms: Children, computers, and powerful ideas*.
- Pearl, J. (1995). Causal diagrams for empirical research. *Biometrika*, 82(4), 669–688. Retrieved 2022-06-03, from <http://www.jstor.org/stable/2337329>
- Resnick, M., Maloney, J., Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... Kafai, Y. (2009, 11). Scratch: Programming for everyone. *Communications of the ACM*, 52, 60-67.

- Sarkar, A., Ragavan, S. S., Williams, J., & Gordon, A. D. (2022). End-user encounters with lambda abstraction in spreadsheets: Apollox2019;s bow or achillesx2019; heel? In *2022 ieee symposium on visual languages and human-centric computing (vl/hcc)* (p. 1-11). doi: 10.1109/VL/HCC53370.2022.9833131
- Sfard, A. (1998). On two metaphors for learning and the dangers of choosing just one. *Educational researcher*, 27(2), 4–13.
- Stead, A. G. (2016). Using multiple representations to develop notational expertise in programming. *University of Cambridge, Computer Laboratory, Technical Report*(UCAM-CL-TR-890).
- Von Glasersfeld, E. (1982). An interpretation of piaget's constructivism. *Revue internationale de philosophie*, 612–635.
- Waite, J., Curzon, P., Marsh, W., & Sentance, S. (2016). Abstraction and common classroom activities. In *Proceedings of the 11th workshop in primary and secondary computing education* (pp. 112–113). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2978249.2978272> doi: 10.1145/2978249.2978272
- Zhang, W., Ramezani, R., & Naeim, A. (2021). *Causal inference in medicine and in health policy, a summary*. arXiv. Retrieved from <https://arxiv.org/abs/2105.04655> doi: 10.48550/ARXIV.2105.04655

An agent for creative development in drum kit playing

Noam Lederman

Music Computing Lab
The Open University
noam.lederman@open.ac.uk

Simon Holland

Music Computing Lab
The Open University
s.holland@open.ac.uk

Paul Mulholland

Knowledge Media Institute
The Open University
p.mulholland@open.ac.uk

Abstract

The aim of this research is to design, implement and evaluate a conversational agent for drum kit players using a call and response model in order to develop their own drumming vocabulary and creativity. Evaluation of the system and analysis of resulting expert drummer behaviour will be used to illuminate various factors: the process of vocabulary development in drum kit players; ways in which technology can assist drum kit players to achieve their creative goals, and more broadly how this approach could support the practices of professional drummers. Although previous research has been conducted in music computing focusing on creativity development in melodic instruments, for example, electronic keyboard (Addessi, 2014) and guitar (Manaris et al., 2011), little is known whether this approach can be applied in rhythmic instruments such as the drum kit. Therefore, the main contribution to knowledge is exploring a gap in human-computer interaction tools for creativity development in drum kit playing.

1. Introduction

Bruford (2018) explains that a drum performance can be creative at different levels: i) by developing certain musical phrases ii) by gluing the performance together iii) and, most ambitiously, by redefining the possible future of the instrument. This research focuses on the first two of these: developing musical phrases and gluing the performance together. Both of these involve a process of fluid vocabulary expansion: learning new drumming vocabulary, combining new vocabulary with existing vocabulary, and using different applications of vocabulary. Leman (2016) refers to these transformations of vocabulary as an *enactment process*, which encourages music performers to experiment with transformations of intentions to sounds, and sounds to intentions, aiming to produce improvised, individual and novel work.

Drumming functions and develops in some of the ways that spoken language does. For example, when learning a spoken language, the focus may be vocabulary expansion through its use in various contexts. Practically, this is generally rehearsed in the form of a conversation, encouraging the learner to use the vocabulary in real-time interaction. Such a conversational learning process involves quick responses, appropriate use of vocabulary and adequate aural and auditory skills. Similarly, in a musical context, expanding drumming vocabulary and developing its practical use might be rehearsed with another drummer. However, in practice drummers have limited opportunities to play with other drummers. This is primarily because i) most musical groups have a single drummer ii) many drummers focus on serving a supportive role rather than developing their improvised vocabulary iii) on a practical level, it is more complicated to arrange a drumming interaction between two acoustic drummers, primarily because of the size of the drum kit and the amount of noise pollution two drum kits can create.

As previously outlined, this research aims to explore how conversational interactions with an agent can support creativity development in drum kit players. The research questions focus on: How can a drumming agent be designed to support drummers? What musical transformations are needed to support drummers? What user controls are required to support drummers? How can the drumming agent be adopted and potentially used in real-life scenarios of drummers? In order to understand which principles should guide the conversational design, to what extent can these principles be enacted by a system, and what effect do these principles have on drummers, we plan to collect quantitative and qualitative data from professional drum kit players who interact with a drumming agent. By collecting both types of data we hope to gain more understanding of improvised drumming conversations and explore the relationship between professional drummers and the drumming agent. Our methodology adopts a mixed

methods approach (Tashakkori & Creswell, 2007) consisting of interviews, surveys, and audio data from the drumming interactions.

2. Related work

Computer technology can be used to enhance live performance (Tanaka et al., 2005) or offer support in the process of learning skills and developing creative aspects of musical performance (Pachet, 2004). Several systems designed to develop creativity in music have taken a conversational approach between the human and the system. For example, *Controlling Interactive Music* (Brown, 2018) focused on improvised piano conversations between a system and a human pianist. *Monterey Mirror* (Manaris et al., 2011) explored a conversational agent to support creativity in jazz guitarists. However, to our knowledge, there is currently no conversational interaction system to foster creativity in expert drum kit players.

3. Conceptual design

In the preliminary phase of this research, an initial conceptual design of a drumming agent was created. Under this design model, the system generates new phrases in the conversation by applying *transformations* of a given *core phrase*. The core phrase is a rhythmic phrase of accented and unaccented notes which forms the starting point for a particular drumming interaction. The transformations involve adapting the core phrase in various ways, for example, by making changes to the rhythms or drum voices used. However, before we explain what each transformation means within our conceptual design, it is important to clarify the origin of the transformations we chose. The drumming language can be learnt by listening, copying, and developing grooves, fills and other rhythmic patterns around the drum kit. Once a certain rhythmic phrase is learnt, there are clear methods for expanding this vocabulary. Although the methods of expanding drumming vocabulary can differ according to the drumming style, there are generic methods that can be used effectively in many styles as well as in our design. The first transformation is *reduction*, which leads to a drumming phrase where only some of the notes are played. This will be useful for drummers who want to create more space in the interaction or simplify the core phrase. The next transformation relates to the drum voices that are used. Therefore, we will refer to this transformation as *orchestration*, where vocabulary expansion is supported by alternating the drum voices. For example, changing the accented notes from the snare drum to the floor tom. The next transformation involves changes in *rhythm*, more specifically, the agent generates subdivisions for some of the notes. This will support and challenge the technical abilities of the human drummers. The final transformation involves a *groove* based on a specific core phrase by using these drum voices: snare drum, bass drum and hi-hat. Practically, the agent will use one of these three voices to play the accented notes while maintaining a consistent groove pattern with the other two voices. This basic architecture aims to offer system prompts which are relevant, varied and sufficiently convincing in the *conceptual space* (Boden, 2009) of professional drummers. This provisional design was tested with seven professional drummers, using pre-recorded system phrases sequenced by the author. The participants of the preliminary study responded positively to the drumming interaction suggesting that the system offers a good basis for exploring our research questions outlined above. Following the analysis of the data from the preliminary study, we progressed with software implementation of this drum-specific system design. Supercollider was chosen due to the extensive facilities for representation and manipulation of musical patterns (Harkins, 2009).

4. Implementation

There are three consecutive phases to the process of implementing our drumming agent design in Supercollider. Phase 1, which has already been achieved, is discussed immediately below, while phases 2 and 3 are discussed in the *Future Work* section. In phase 1, a drumming agent was created based on our conceptual design. At this phase the agent was able to generate phrases in real-time based on a database of core phrases and the following transformations: *reduction* plays only the accented notes; *orchestration* alternates the drum voices used for the accented or unaccented notes; *rhythm* subdivides every unaccented note to two even notes; *Groove* focuses on the snare, bass drum and hi-hat, transforms the core phrase into a groove. To facilitate the interaction, we created a framework of common time, i.e. 4/4, where the agent continuously generates one bar of drumming and schedules a one-bar gap between its generated transformations for the human drummer to respond. Following the data from the

preliminary study, we added three additional elements to our design. These include rhythmic rotations, randomised rests, and a *human* variable. The rhythmic rotations assist the agent in generating more interesting phrases by changing the starting point of the core phrase. The randomised rests schedule occasional rests within the transformations and the *human* variable offers a subtle element of playing some notes slightly ahead or behind the beat. Both the rests and *human* variable assist the agent transformations sound more fluid. In the next section, we present our future work plans which include adding user controls and machine listening elements so the agent can adapt to each individual drummer and provide better long-term support for their development as well as evaluating the drumming agent with professional drummers in real-life scenarios.

5. Future work

Phase 2 will focus on adding user controls so the human drummer can adapt the behaviour of the agent during real-time interaction. This phase will build on the prototype developed in phase 1, implement some of the cues detected in the analysis from phase 1 and give the tool steerable capabilities. The user controls will give drummers the ability: (i) to focus on specific transformations and explore certain areas of their playing (ii) to record the interactions in order to listen back and evaluate (iii) to adapt the agent volume (iv) to adapt the tempo (v) to add or remove a metronome. In addition, we will explore integrating machine listening elements in Supercollider with BeatTrack, DrumTrack or Onsets (Wilson et al, 2011).

Phase 3 will focus on refining the user controls and machine listening elements. In addition, we will evaluate the agent in real-life scenarios with professional drummers. Likely scenarios include: the practice room, education settings such as lessons or masterclasses, recordings, and live performances. The participants will maintain a documented log of the drumming interactions. The evaluation process will involve thematic analysis (Braun & Clarke, 2012) of the logs, interviews and surveys from all phases aiming to understand the potential of this type of tool in drum kit playing practices. The focus on creativity in drum kit playing, vocabulary expansion, user controls and use of the technology in real-life scenarios might provide opportunities for future research and interest not only in the music computing community but also for music educators, music performers and human-computer interaction researchers designing for support of real-time interactions with an agent.

6. References

- Adnessi, A. R. (2014). Developing a theoretical foundation for the reflexive interaction paradigm with implications for training music skill and creativity. *Psychomusicology: music, mind, and brain*, 24(3), 214.
- Boden, M. A. (2009). Computer models of creativity. *AI Magazine*, 30(3), 23-23.
- Braun, V., & Clarke, V. (2012). *Thematic analysis*. American Psychological Association.
- Brown, A. R. (2018). Creative improvisation with a reflexive musical bot. *Digital Creativity*, 29(1) 518.
- Bruford, B. (2018). *Uncharted: Creativity and the expert drummer*. University of Michigan Press.
- Harkins, Henry James. "A Practical Guide to Patterns." SuperCollider 3.3 Documentation, 2009.
- Leman, M. (2016). *The expressive moment: How interaction (with music) shapes human empowerment*. MIT press.
- Manaris, B., Hughes, D., & Vassilandonakis, Y. (2011, June). Monterey mirror: combining Markov models, genetic algorithms, and power laws. In *Proceedings of 1st Workshop in Evolutionary Music, 2011 IEEE Congress on Evolutionary Computation (CEC 2011)* (pp. 33-40). New York, NY: IEEE.
- Pachet, F. (2004). On the design of a musical flow machine. *A Learning Zone of One's Own*, 111-134.
- Tanaka, A., Tokui, N., & Momeni, A. (2005, November). Facilitating collective musical creativity. In *Proceedings of the 13th annual ACM international conference on Multimedia* (pp. 191-198).
- Tashakkori, A., & Creswell, J. W. (2007). The new era of mixed methods. *Journal of mixed methods research*, 1(1), 3-7.
- Wilson, S., Cottle, D., & Collins, N. (2011). *The SuperCollider Book*. The MIT Press.

Would a Rose by any Other Name Smell as Sweet? Examining the Cost of Similarity in Identifier Naming

Naser Al Madi
Computer Science
Colby College
nsalmadi@colby.edu

Matianyu Zang
Computer Science
Brown University
matianyu_zang@brown.edu

Abstract

Background: Identifier naming is one of the main sources of information in program comprehension, where the majority of software development time is spent. When reading natural language texts or code, readers perform lexical access to retrieve the orthography (word shape), phonology (pronunciation), and semantic (meaning) representations of words from memory. The successful retrieval of these representations is vital for success in comprehension and subsequent code maintenance and evolution.

Objective: This paper examines the cost of identifier similarity in orthography, phonology, or semantic representation and how that affects debugging performance and programmer workload. By recognizing common identifier naming combinations that hinder code comprehension, we can discover new programming best-practices and create automated tools that flag problematic naming combinations.

Method: Through a human experiment (n=43), we explore the impact of orthographic, phonological, and semantic similarity on debugging success, time, and workload. In our experiment, participants worked on debugging three programs, each of which has two versions that are identical except for one pair of identifiers, with either similar identifier names (e.g. i and j) or dissimilar names (e.g. row and column). Participants were randomly assigned a version of the code, and their performance was recorded to measure debugging success and time. At the end of each trial, they reported the subjective workload through NASA Task Load Index (NASA-TLX).

Results: We found some differences in debugging success and duration between similar and dissimilar identifiers with advanced programmers, but the differences are not statistically significant.

Conclusion: The results call for further investigation of identifier similarity and its influence on code comprehension. The study of identifier similarity can shed light on new linguistic anti-patterns that could potentially hinder code comprehension.

1. Introduction

Software developers spend the majority of their time reading source code for program comprehension, debugging, modifying, or learning (Maalej et al., 2014; Shneiderman and Mayer, 1979; Von Mayrhauser and Vans, 1995). In fact, several studies revealed that developers spend more than 50% of their time searching for information (Ko et al., 2007; Murphy et al., 2006; Maalej et al., 2014; Von Mayrhauser et al., 1997; Standish, 1984; Tiarks, 2011). Therefore, program comprehension is a critical component of the software development process, and even a small improvement in comprehension time can lead to significant gains in software development time and programmer workload. In addition, the association between identifier naming and code quality has been well established on the class and method levels (Butler et al., 2010). This is especially important when we consider the economic standpoint that software maintenance is the biggest factor in the cost of a software system (Barry et al., 1981; Siegmund and Schumann, 2015).

In programming, comprehension depends on a number of factors such as code comments and the names programmers choose for variables, classes, functions, modules, parameters, and constants, also known as identifier naming. In fact, identifier naming is regarded as one of the primary sources of information that programmers use to understand source code (Lawrie et al., 2006; Newman et al., 2020; Maletic and Marcus, 2001). Multiple studies have been conducted on identifier naming, and we group these previous works under two categories that are relevant to our current work: First, works that focused

on the influence of identifier naming on comprehension. Second, works that focused on the impact of identifier naming on workload and difficulty.

Under the first category, an early study by Takang et al. (1996) focused on the effects of comments and identifier names on code comprehension, and found that "programs that contain 'full' identifier names are more understandable than those with abbreviated identifier names." The study used multiple-choice questions and subjective code assessment to measure program comprehension, and one of the study's recommendations was to use a better assessment method of program comprehension in controlled experiments. Similarly, Lawrie et al. (2007) compared the influence of single letters, abbreviations, and full word identifiers on comprehension. The study found evidence that full word identifiers lead to better code comprehension than single letters and abbreviations. Another study on identifier abbreviations by Scanniello et al. (2017) compared full-word identifiers to abbreviations in a bug fixing task, and found no significant difference in debugging duration or effectiveness between the two. Moreover, Binkley et al. (2013) reported multiple experiments comparing camel case and underscore identifier naming styles. The study demonstrated that beginner programmers benefited from the use of camel case in comprehension and effort, while experienced programmers were not affected by identifier styles.

Under the second category, a key study by Arnaoudova et al. (2013) presented linguistic anti-patterns, which described common poor naming and commenting choices in code. Linguistic anti-patterns highlight inconsistencies between identifier names and their behaviors. A recent study by Fakhoury et al. (2020) focused on the influence of linguistic anti-patterns on developers' cognitive load during bug localization tasks. The study revealed that linguistic anti-patterns significantly increased developers' workload.

In this paper, we focus on the influence of identifier similarity on code comprehension and programmer workload. Identifier similarity could occur in one of three lexical dimensions, orthography (word shape), phonology (word pronunciation), and semantics (word meaning). When programmers use identifiers that are similar in one of the three dimensions, it is possible for confusing identifier pairs to hinder code comprehension. An example of orthographically similar identifiers that are often used in the same scope are 'i' and 'j' in nested loops. The two identifiers are similar in shape, easily causing programmers to substitute one for another or use them interchangeably, and that leads to logical errors that are difficult to localize and fix.

A previous study Aman et al. (2021) indicated that such confusing identifier naming combinations do occur in production software, yet an empirical study focusing on the effect of similarity in identifier naming on comprehension is still needed. The potential influence of identifier similarity on comprehension could uncover new linguistic anti-patterns that impede code comprehension.

The foundations of this study come from Cognitive Science, where the role of lexical access in reading natural language texts has been studied extensively (Rayner, 1998). Lexical access describes the retrieval of word shape, pronunciation, and meaning from memory during reading for comprehension. Typically, readers retrieve meaning and sound from word form, but occasionally top-down processing takes place in predictable contexts, for example, where the word is recognized from its meaning before the eyes make a fixation on it (Rayner, 1998). The presence of this top-down phenomena is usually associated with skilled readers who are more likely to skip predictable and frequent words (Rayner et al., 2006).

Despite sharing many cognitive processes with reading natural language texts, reading source code differs fundamentally in purpose, syntax, semantics, and viewing strategy from natural texts (Busjahn et al., 2015; Liblit et al., 2006; Busjahn et al., 2014; Schulte et al., 2010). This motivates the study of identifier naming and lexical similarity in code further, especially with the important implications on code comprehension and workload discussed earlier.

In this study, to avoid the methodological limitations of multiple choice questions, we focus instead on debugging (fixing logical errors in code) as an indicator of code comprehension. This approach allows us to measure debugging success probability and debugging duration as objective ways to evaluate code

comprehension under different conditions. In addition, we use NASA Task Load Index (NASA-TLX) to assess subjective workload during the debugging process. We aim to answer the following research questions:

- RQ1: Does identifier similarity influence success in debugging source code?
- RQ2: Does identifier similarity influence source code debugging time?
- RQ3: Does identifier similarity influence source code debugging difficulty?

Studying identifier similarity in source code helps uncover new best practices that enhance programmer productivity and comprehension. Such best practices can be integrated in automated tools for enhancing code readability, maintainability, and quality.

2. Pilot Study

In this section, we present the details of our first **IRB approved** pilot study and the sources of data we used to answer our research questions. Our within-subject pilot study consisted of 12 participants in total.

2.1. Objective

Our goal in the pilot study was to empirically verify the hypothesis that identifier similarity influences debugging success and time. We examined this hypothesis with orthographic, phonological, and semantic similarity in identifier naming, with the assumption that similarity could potentially hinder lexical access and in turn comprehension. We state two research questions to clarify our objective in the pilot study:

- RQ1: Does identifier similarity influence success in debugging source code?
- RQ2: Does identifier similarity influence source code debugging time?

By providing some initial results for the two questions in the pilot study, we aimed to gain insights for a larger experiment with additional sources of data and additional research questions (experiment one).

2.2. Participants

The 12 participants in our study were Computer Science students with experience ranging from one to three years. All participants had completed their second Computer Science course (CSII or equivalent) and were thus familiar with Java programming language. All participants were above the age of 18, and 5 were Female and were 7 Male. Participation was voluntary, and participants were awarded a \$15 gift card after the experiment.

```
for(int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        if (this.myGrid[j][j] == ""){
            empty++;
        }
    }
}
```

(a) Similar

```
for(int row = 0; row < 3; row++){
    for (int column = 0; column < 3; column++){
        if (this.myGrid[column][column] == ""){
            empty++;
        }
    }
}
```

(b) Dissimilar

Figure 1 – Pilot study orthographic similarity material (identifiers highlighted).

2.3. Material

As stimuli for our pilot study, we prepared three programs in Java that included a pair of identifiers similar in orthography, phonology, or semantics. The three programs had one logical defect (bug) each

that inhibited the code from working as intended when running. Nonetheless, programs could be compiled successfully without any syntactic errors. As a control, we modified the programs to use identifiers that were dissimilar, with all other aspects of the programs remaining identical to the first three programs. Therefore, each of the three programs had two versions, one with similar identifiers and one with dissimilar identifiers.

For orthographic similarity, we prepared a text-based Tic-Tac-Toe program in Java. The program consisted of 200 lines of code, and it included a logical defect in the check for a tie method. Figure 1 shows the two versions of similar and dissimilar identifiers presented. The similar version uses the identifiers ‘i’ and ‘j’ which are orthographically similar (in form), and the dissimilar version of the same code uses ‘row’ and ‘column’ instead. The rest of the code is identical. Both programs included the same logical error, seen at line three in Figure 1, where the grid was indexed incorrectly. Orthographic similarity was determined by programmatically comparing the shape of letters or words to determine the percentage of overlap.

<pre>public static String join(String separator, String[] input) { ... public static String connect(String str1, String str2){ (a) Similar</pre>	<pre>public static String listToString(String separator, String[] input) { ... public static String AddSpace(String str1, String str2){ (b) Dissimilar</pre>
---	---

Figure 2 – Pilot study semantic similarity material (identifiers highlighted).

For semantic similarity, we prepared a phonebook program that stored data in a comma-separated file. One functionality of the program was allowing adding data from a new comma-separated file to the phonebook. The program was 149-line-of-code long. Figure 2 shows the similar and dissimilar versions of the program. The identifiers in this task were method names, and the logical error was using the incorrect method later in the code. The identifiers ‘join’ and ‘connect’ were semantically similar, while ‘listToString’ and ‘addSpace’ were dissimilar, and they performed the same job in the code. Semantic similarity was determined using Wordnet (Miller, 1995).

<pre>for(int numIter=0; numIter<100; numIter++) { monitor.roll(); if ((numIter / 10) == 0) { monitorTens.add(monitor.getSideUp()); } }</pre> <p>(a) Similar</p>	<pre>for(int iter=0; iter<100; iter++) { die.roll(); if ((iter / 10) == 0) { monitorTens.add(die.getSideUp()); } }</pre> <p>(b) Dissimilar</p>
---	--

Figure 3 – Pilot study phonological similarity material (identifiers highlighted).

For phonological similarity, we prepared a text-based Die simulation where a die was thrown one hundred-times, and every tenth throw was added to a list of results. Figure 3 shows similar and dissimilar identifiers in the same context in the program, where ‘numIter’ and ‘monitor’ are phonologically similar and ‘iter’ and ‘die’ are not. Both programs included the same logical error in the way results were added to the results list. Phonological similarity was determined using Wordnet (Miller, 1995).

2.4. Pilot Study Procedure

This experiment was conducted remotely using Zoom video conferencing and screen sharing. One member of the research team met with one participant on Zoom at a time, and the general procedure of the experiment was explained along with an online consent form. If the participant agreed to proceed

with the experiment, the online form collected the participant's programming experience in years along with demographic information.

The participant shared their screen with the member of the research team, and the participant was provided the experiment material. Each experiment included three programs in a random order to eliminate the order effects, and the programs were randomized in terms of similar and dissimilar identifiers as well. This means that each participant was given either two programs with similar identifiers and one that was dissimilar, or two programs with dissimilar identifiers and one similar. This accounted for the experience effect, where participants perform better in a repeated task, and also guaranteed that each participant worked on debugging programs with similar and dissimilar identifiers.

Each of the three debugging tasks had a maximum time of 15 minutes, and comments at the top of each program explained the intended function of the program, an example of the intended output, and the time limit of each task, but no information of the number of bugs was provided. Participants worked on writing and running the code in a natural software development environment using the VS Code Integrated Development Environment, and their behavior was recorded in terms of debugging time and whether they were successful in debugging or not. The experiment took approximately 50 minute in total, and after each task the code was saved and uploaded back to the research team.

2.5. Pilot Study Design

To answer our questions on the influence of identifier similarity on debugging success and time, we compared the debugging time and success probability of both similar and dissimilar versions of the same program. Since the only variable was identifier similarity, this could give us evidence in the form of increased or reduced time and success probability in the similar condition. In addition, we could focus on a specific type of similarity, such as orthographic, phonological, or semantic and inspect debugging success probability and time in each type separately.

2.6. Pilot Study Results

RQ1: Does identifier similarity influence success in debugging source code?

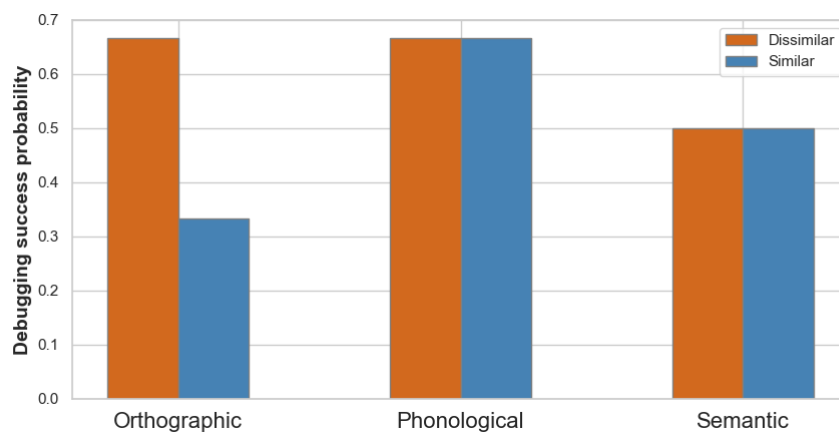


Figure 4 – Debugging success probability in the pilot study.

We measured debugging success probability as the proportion of participants who correctly fixed the logical error in a trial within 15 minutes. Figure 4 compares the success probability of similar and dissimilar identifiers. The figure shows that identifier similarity in phonology and semantics do not appear to influence debugging success, as similar and dissimilar identifiers result in the same debugging success probability. On the other hand, presenting orthographically similar identifiers to programmers decreased their aggregate debugging success from 66% to 33%. Yet, a Fisher's test of independence was performed to examine the relation between identifier orthographic similarity and debugging success. The relation between these variables was not significant, $p = .56$. Fisher's test was chosen since the sample size is small (i.e., $n < 50$).

RQ2: Does identifier similarity influence source code debugging time?

We measured the amount of time needed to successfully debug code with similar and dissimilar identifiers, in order to quantify the influence of identifier similarity on debugging time. Figure 5 shows debugging time (in seconds) of code with similar and dissimilar identifiers. The white dot is the median, the upper and lower limits are the maximum and minimum values respectively, and the shape is a histogram of the data points. The results demonstrate less debugging time for code with orthographically and phonologically dissimilar identifiers compared to the same code with similar identifiers. There is no evidence suggesting the same effect for semantically similar identifiers. In fact, the average debugging time is slightly lower for semantically similar identifiers.

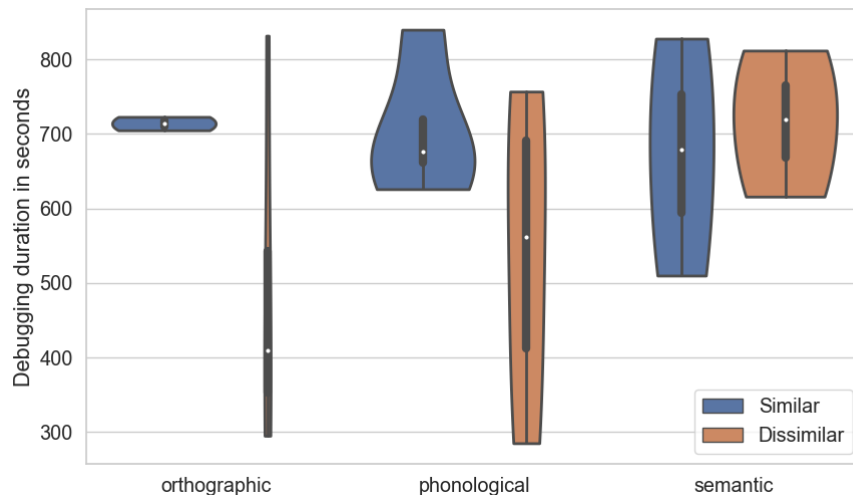


Figure 5 – Debugging time analysis in the pilot study.

On average, debugging code with orthographically similar identifiers takes 714 seconds, and debugging the same code with dissimilar identifiers takes 486 seconds. The difference is approximately 3.8 minutes, which may be substantial when scaled to bigger programs and projects. Nonetheless, there is no significant effect for orthographic similarity on debugging duration, $t(10) = -1.27$, $p = .27$, despite similar identifiers ($M = 714$, $SD = 12$) attaining higher debugging time than dissimilar identifier ($M = 486$, $SD = 238$).

Similarly, debugging code with phonologically similar identifiers on average takes 704 seconds, and dissimilar identifiers takes 541 seconds. Yet, there is no significant effect for phonological similarity on debugging duration, $t(10) = -1.4$, $p = .20$, despite similar identifiers ($M = 704$, $SD = 93$) attaining higher debugging time than dissimilar identifier ($M = 541$, $SD = 212$).

The results of the pilot study called for further investigation of the influence of identifier similarity on code comprehension and debugging specifically. On one hand orthographic and phonological similarity seem to increase debugging time, and on the other hand similarity in semantics does not seem to have any effect on debugging time and success. Also, the differences between similar and dissimilar identifiers, although pronounced with orthographic and phonological similarity, does not appear to be statistically significant, possibly due to the small sample size. Therefore, we tried to address some of these issues in the main experiment, which we present next.

3. Experiment

In this section, we present the details of our main experiment which incorporated elements from the pilot study, in addition to new sources of data that we used to answer our research questions. Our controlled experiment consisted of 31 participants in total.

3.1. Objective

The pilot study provided no statistically significant evidence in support of the hypothesis that identifier similarity influences debugging success and time. Our goal in the main experiment was to improve our stimuli and incorporate additional sources of data to quantify the effect of identifier similarity on code comprehension and programmer workload.

- RQ1: Does identifier similarity influence success in debugging source code?
- RQ2: Does identifier similarity influence source code debugging time?
- RQ3: Does identifier similarity influence source code debugging difficulty?

With a larger sample size and an enhanced stimulus, we intended to provide better evidence to answer our research questions.

3.2. Participants

The 31 participants in our study were Computer Science students with experience ranging from one to eight years. Based on the k-means classification, 9 were classified as advanced programmer with four or more years of programming experience, and the remaining were classified as beginner. All participants had completed their second Computer Science course (CSII or equivalent) and were thus familiar with Java programming language, while some students had taken upper-level courses and had completed several internships as professional programmers. All participants were above the age of 18, and 17 were Female, 13 Male, and 1 Non-binary. Participation was voluntary, and participants were awarded a \$15 gift card after the experiment.

3.3. Material

To enhance our experiment stimuli, we used a more systematic way of searching for similar identifier names in popular open-source projects. We used 10 popular Java repositories to mine for identifier names. We used repositories that fulfilled the guides of selecting meaningful repositories sets (Munaiah et al., 2017). The repositories represent approximately 3.9 million lines of code, after removing comments from code. Only Java files were processed; code in other languages was excluded. Table 1 shows the details of the selected Java repositories and the number of tokens in each repository.

Table 1 – Java repositories used to search for similar identifier names.

Repository	Files	Lines
Ant	1,314	304,957
Batik	1,651	353,516
Cassandra	2,673	586,451
Eclipse	154	25,914
Log4J	309	60,078
Lucene	8,467	1,874,373
Maven2	378	60,775
Maven3	834	113,384
Xalan-J	958	348,769
Xerces2	833	261,312
Total	17,571	3,979,251

From the repositories, we collected 170,823 unique identifiers. Using Wordnet (Miller, 1995) and a custom tool for calculating word-form overlap, we calculated the orthographic, phonological, and semantic similarity among the 1,000 most frequent identifiers in the code. These identifiers represent common identifier names that programmers use, and therefore they represent more realistic identifier names than the pilot study. From the resulting list of similar identifiers, the research team selected similar identifiers that could be integrated in realistic code snippets for use in our experiment.

<pre>for(int count=0; count<100; count++){ myDie.roll(); int number = myDie.getSideUp(); if ((number / 10) == 0) { results.add(myDie.getSideUp()); } }</pre>	<pre>for(int iteration=0; iteration<100; iteration++){ myDie.roll(); int sideUp = myDie.getSideUp(); if ((sideUp / 10) == 0) { results.add(myDie.getSideUp()); } }</pre>
(a) Similar	(b) Dissimilar

Figure 6 – Experiment semantic similarity material (identifiers highlighted).

We repeated the same structure as the pilot study with three programs and two versions of each program, one with similar and one with dissimilar identifiers. The orthographic similarity stimuli remained the same, since ‘i’ and ‘j’ are commonly used identifiers. For semantic similarity, we prepared a program with similar and dissimilar identifiers as shown in Figure 6. The semantically similar identifiers that we found in the Java repositories were ‘count’ and ‘number’ and the dissimilar identifiers were ‘iteration’ and ‘sideUp’. Again, both versions included the same logical error.

<pre>String right = stack.pop(); String operand = stack.pop(); String left = stack.pop(); write = left + " " + right + " " + operand; stack.push(right);</pre>	<pre>String right = stack.pop(); String operand = stack.pop(); String left = stack.pop(); postfix = left + " " + right + " " + operand; stack.push(right);</pre>
(a) Similar	(b) Dissimilar

Figure 7 – Experiment phonological similarity material (identifiers highlighted).

For phonological similarity, we prepared a program that converted an infix expression to postfix. Figure 7 shows the two versions of the program with phonologically similar and dissimilar identifiers. The identifiers ‘write’ and ‘right’ were identical in pronunciation (homophones), while ‘postfix’ and ‘right’ were dissimilar.

3.4. Study Design

To answer our research questions, we collected three sources of data in our experiment: debugging time, success probability, and workload surveys. To answer our questions on the influence of identifier similarity on debugging success and time, we compared the debugging time and success probability of both similar and dissimilar versions of the same program, as we did in the pilot study. The order of the tasks was randomized, and the version of each program (similar/dissimilar) was also randomized.

Subjective Workload: To answer the research question focusing on the influence of identifier similarity on debugging difficulty and workload, we utilized NASA-Task Load Index (NASA-TLX) (Hart and Staveland, 1988). NASA-TLX has a wide range of applications (Grier, 2015) from aviation, physical activity, cognitive tasks, to software development (Fritz et al., 2014; Al Madi et al., 2022). The survey uses a multidimensional scale to represent six factors that contribute to workload and difficulty during various tasks. We used a simplified version of NASA-TLX which reported the magnitude of the experienced workload component on a scale from 0 to 100. The components are (Hart and Staveland, 1988):

- **Mental Demand:** How much mental and perceptual activity was required?
- **Physical Demand:** How much physical activity was required?

- **Temporal Demand:** How much time pressure did you feel due to the pace at which the tasks or task elements occurred?
- **Overall Performance:** How successful were you in performing the task?
- **Effort:** How hard did you have to work (mentally and physically) to accomplish your level of performance?
- **Frustration Level:** How irritated, stressed, and annoyed versus content, relaxed, and complacent did you feel during the task?

The magnitude of experienced components was reported on each scale in increments of 5, where zero represented the lowest magnitude and 100 represented the highest. The overall score described the perceived workload in performing the task and is often dubbed TLX. This final score had no unit nor an upper limit. We used NASA-TLX after each trial to measure the subjective workload experienced by participants when debugging source code with similar/dissimilar identifier naming.

3.5. Experiment Procedure

First, the participant was entered into the experiment lab, and the general procedure of the experiment was explained along with an online consent form. If the participant agreed to proceed with the experiment, the online form collected the participant's programming experience in years, and demographic information. The first debugging task was presented and timed from the start until the participant fixed the bug in the code or 15 minutes passed. The comments in the file informed programmers of the main functionality of the code, but no information of number of bugs were provided. During the task, participants were free to modify and run the codes. If the 15-minute limit was reached, the debugging task was considered unsuccessful. The debugging tasks were presented in randomized order, and the identifiers in each task were presented in either similar or dissimilar condition. After each debugging task, the participant filled an online version of NASA-TLX to report the workload experienced in that task.

3.6. Experiment Results

RQ1: Does identifier similarity influence success in debugging source code?

Here we attempt to provide more evidence on the influence of identifier similarity on debugging success. We approach this by examining the debugging success data of advanced programmers in contrast to data from beginners. Figure 8 shows debugging success probability for advanced programmers when presented with programs containing similar and dissimilar identifiers. The results demonstrate decreased success when programmers were presented with similar identifiers in orthography or phonology, with no difference in semantic similarity.

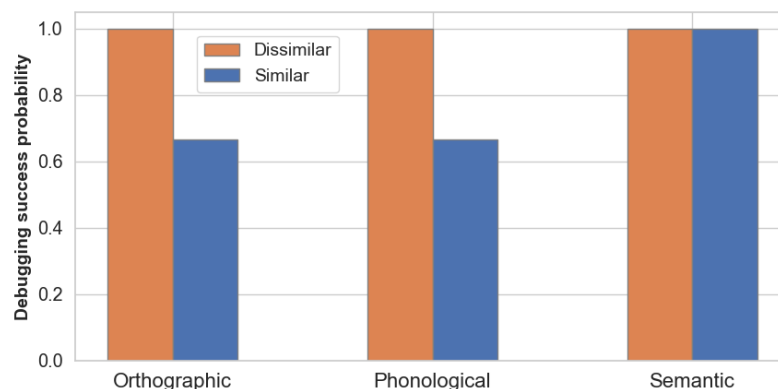


Figure 8 – Advanced programmers debugging success probability in the main experiment.

Looking at the data from beginners, we observe in Figure 9 the lower success probability in debugging for beginners, evident by the lower scores when compared to advanced programmers in figure 8. More

importantly, the differences between similar and dissimilar identifiers appear reversed in orthography and semantics.

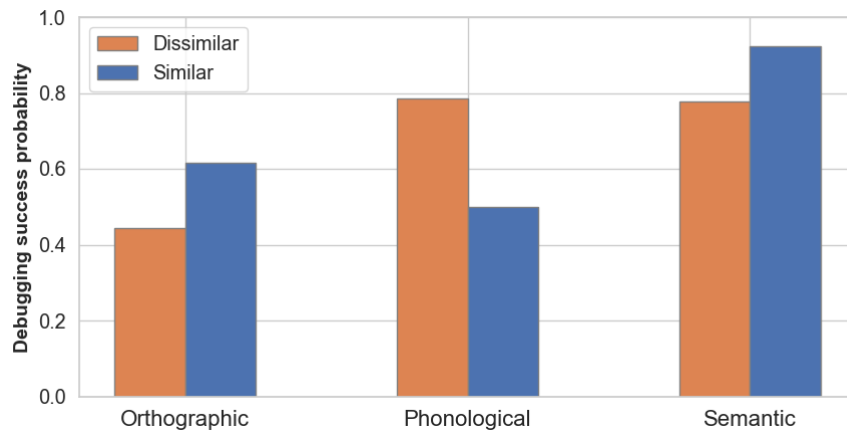


Figure 9 – Beginner programmers debugging success probability in the main experiment.

Comparing the results of beginners and advanced programmers, it appears that beginners have lower debugging success probability in general (comparing Figure 8 and Figure 9). Also, advanced programmers appear to be hindered by identifier similarity in orthography and phonology, while beginners only show reduced success with phonologically similar identifiers. Surprisingly, beginners seem to be advantaged by identifier similarity in orthography and semantics. Yet, none of the differences are statistically significant according to Fisher's test of independence. For advanced programmers Fisher's test results are $p = .33$ for orthography and $p = .49$ for phonology. For beginner programmers, Fisher's test results are $p = .66$ for orthography, $p = .34$ for phonology, and $p = .54$ for semantic similarity.

RQ2: Does identifier similarity influence source code debugging time?

Similar to the previous question, we examine the debugging duration of similar and dissimilar identifiers contrasting data from beginners to advanced programmers. Akin to the pilot study, we focus on the data from participants who successfully debugged the code.

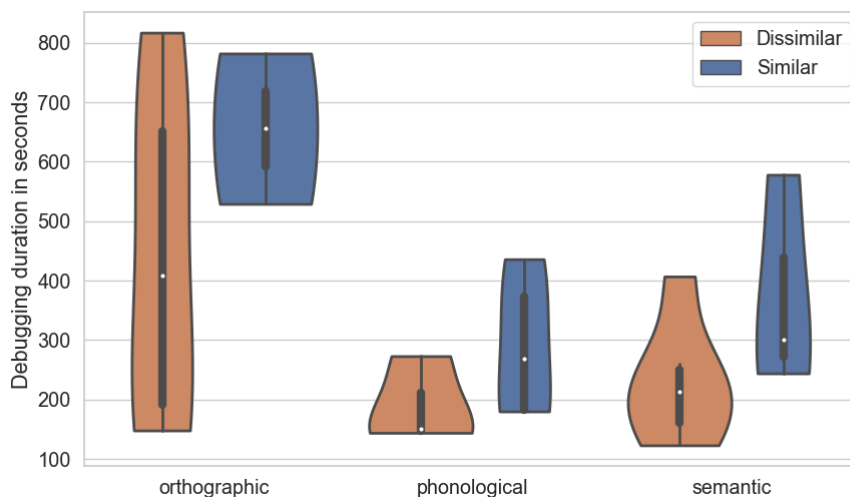


Figure 10 – Advanced programmers debugging time analysis in the main experiment.

Starting with Figure 10, which shows debugging duration for advanced programmers across similar and dissimilar programs, we notice a consistent pattern of similar identifiers leading to longer debugging time. This pattern is present in orthographic, phonological, and semantic similarity, and it is most

clear in orthographic similarity. On average, programmers takes 655 seconds to debug code with orthographically similar identifiers, and 439 seconds to debug the same code with dissimilar identifiers. The difference is approximately 3.6 minutes between similar and dissimilar identifiers. Nonetheless, there is no significant effect for orthographic similarity on debugging duration, $t(7) = -0.98$, $p = .36$, despite similar identifiers ($M = 655$, $SD = 178$) attaining higher debugging time than dissimilar identifier ($M = 439$, $SD = 284$).

For phonological similarity, programmers takes 288 seconds to debug code with similar identifiers, and 189 seconds to debug the same code with dissimilar identifiers. The difference is approximately 1.6 minutes between similar and dissimilar identifiers. Nonetheless, there is no significant effect for phonological similarity on debugging duration, $t(7) = -1.19$, $p = .28$, despite similar identifiers ($M = 288$, $SD = 127$) attaining higher debugging time than dissimilar identifier ($M = 189$, $SD = 72$).

For semantic similarity, programmers take 374 seconds to debug code with similar identifiers, and 227 seconds to debug the same code with dissimilar identifiers. The difference is approximately 2.45 minutes between similar and dissimilar identifiers. Nonetheless, there is no significant effect for semantic similarity on debugging duration, $t(7) = -1.27$, $p = .14$, despite similar identifiers ($M = 374$, $SD = 178$) attaining higher debugging time than dissimilar identifier ($M = 277$, $SD = 101$).

Examining the debugging duration data of beginners, Figure 11 presents a comparison between similar and dissimilar code debugging times. Unlike advanced programmers, the amount of time taken by beginners to debug programs with similar and dissimilar identifiers appears to be comparable, and in some instances code with dissimilar identifiers took longer to debug.

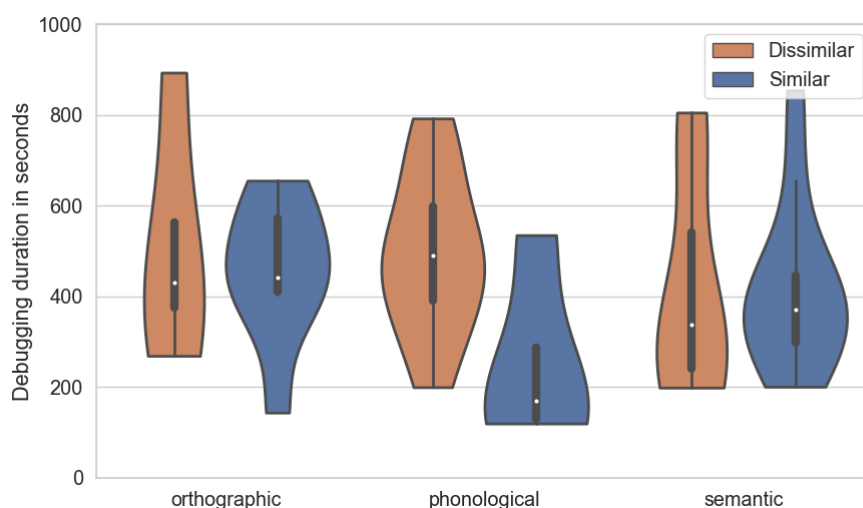


Figure 11 – Beginner programmers debugging time analysis in the main experiment.

For orthographic similarity, beginners take 627 seconds to debug code with similar identifiers, and 725 seconds for dissimilar identifiers. The difference is approximately 1.6 minutes, with a surprising advantage to similar identifiers. For phonological similarity, beginners takes 574 seconds to debug code with similar identifier, and 579 seconds for dissimilar identifier. The difference is 5 seconds on average. For semantic similarity, beginners takes 453 seconds to debug code with similar identifiers, and 522 seconds for dissimilar identifiers. The difference is approximately 1 minute, with an advantage to similar identifiers. None of the differences are statistically significant.

In summary, the results suggest that the influence of identifier similarity on debugging time is seen as an increase in debugging time in the data of experienced programmers, yet identifier similarity has no influence or reversed influence on beginner programmers. None of the differences appear to be significant on a statistical level.

RQ3: Does identifier similarity influence source code debugging difficulty?

To understand debugging difficulty, we measure participants' subjective workload via NASA-TLX. Figure 12 shows the Task Load Index (TLX) of advanced programmers in debugging similar and dissimilar programs. The results show increased workload and difficulty when programmers encounter similar identifiers in phonology, and lower workload debugging code with similar identifiers in orthography and semantics.

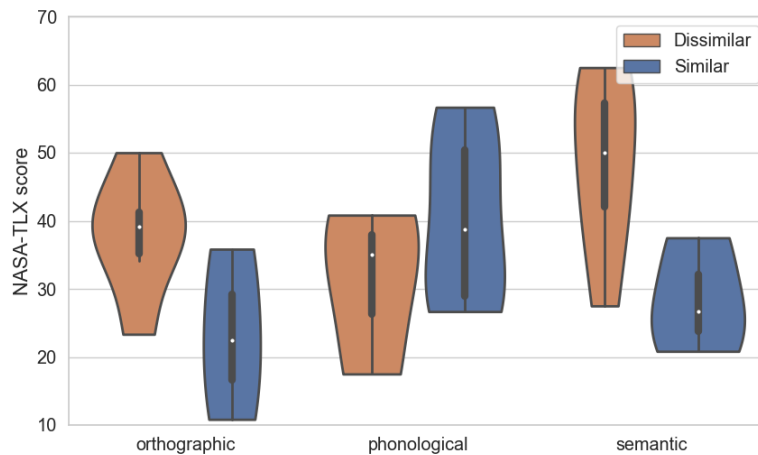


Figure 12 – Advanced programmers NASA-TLX in the main experiment.

For orthographic similarity, advanced programmers have an average NASA-TLX score of 23.1 (SD = 12.5) with similar identifiers and 38.0 (SD = 8.8) with dissimilar ones. Despite the huge difference of 14.9 with similar identifiers acquiring less workload than dissimilar identifiers, there is no significant effect for semantic similarity on NASA-TLX score, $t(7) = 2.10$, $p = .07$. For phonological similarity, advanced programmers have an average score of 40.1 (SD = 12.9) with similar identifiers, and 31.1 (SD = 12.1) with dissimilar identifiers. The difference between similar and dissimilar identifiers is approximately 9. In this case, similar identifiers increased debugging difficulty more than dissimilar identifiers. However, there is no significant effect for semantic similarity on NASA-TLX score, $t(7) = -1.00$, $p = .34$. Lastly for semantic similarity, advanced programmers have an average NASA-TLX score of 28.3 (SD = 8.5) with similar and 48.2 (SD = 5.3) with dissimilar ones. The gap goes to approximately 9.9 between similar and dissimilar identifiers, where similar identifiers make the debugging less challenging. However, we again fail to detect significant effect for semantic similarity on debugging duration as $t(7) = -1.62$, $p = .15$.

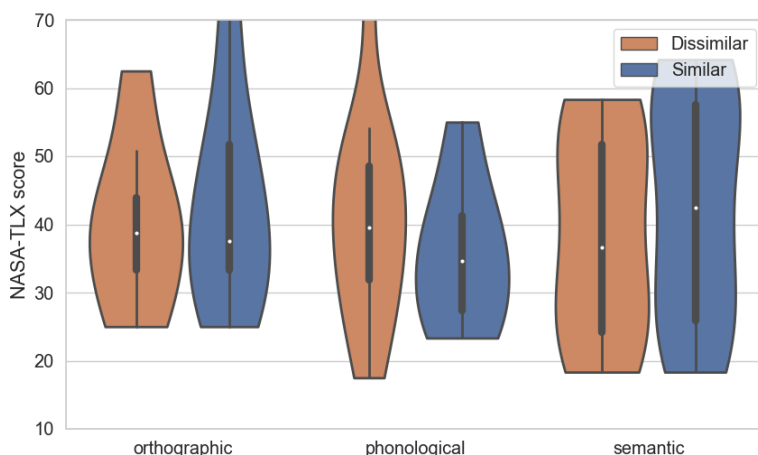


Figure 13 – Beginner programmers NASA-TLX in the main experiment.

On the other hand, Figure 13 illustrates the NASA-TLX data for beginners, where we observe no dif-

ference in the workload of debugging similar and dissimilar programs. For orthographic similarity, beginner programmers have an average NASA-TLX score of 42.7 (SD = 15.06) with similar identifiers and 40.0 (SD = 12.5) with dissimilar identifiers. Similar identifiers add slightly more difficulty to the debugging task as the mere difference of 2.7 indicates. For phonological similarity, beginner programmers have an average NASA-TLX score of 35.6 (SD = 10.8) with similar identifiers and 40.6 (SD = 14.8) with dissimilar identifiers. The difference is approximately 4 between similar and dissimilar identifiers, and debugging codes with similar identifiers is easier in this case. In addition, beginner programmers have an average NASA-TLX score of 42.6 (SD = 16.7) with similar identifiers and 38.6 with dissimilar ones (SD = 15.4) for semantic similarity. The difference is also 4, but this time programs with similar identifiers appear to be harder. Conducting statistical tests on the data, we find that there is no significant effect for all three types of similarity, with $t(19) = -0.44$, $p = .67$; $t(19) = 0.83$, $p = .42$; $U = 59.5$, $p = .97$ for orthographic, phonological, and semantic similarity respectively. Note that we use the Student T-test for orthographic and phonological similarity and the Mann-Whitney U Test for semantic similarity.

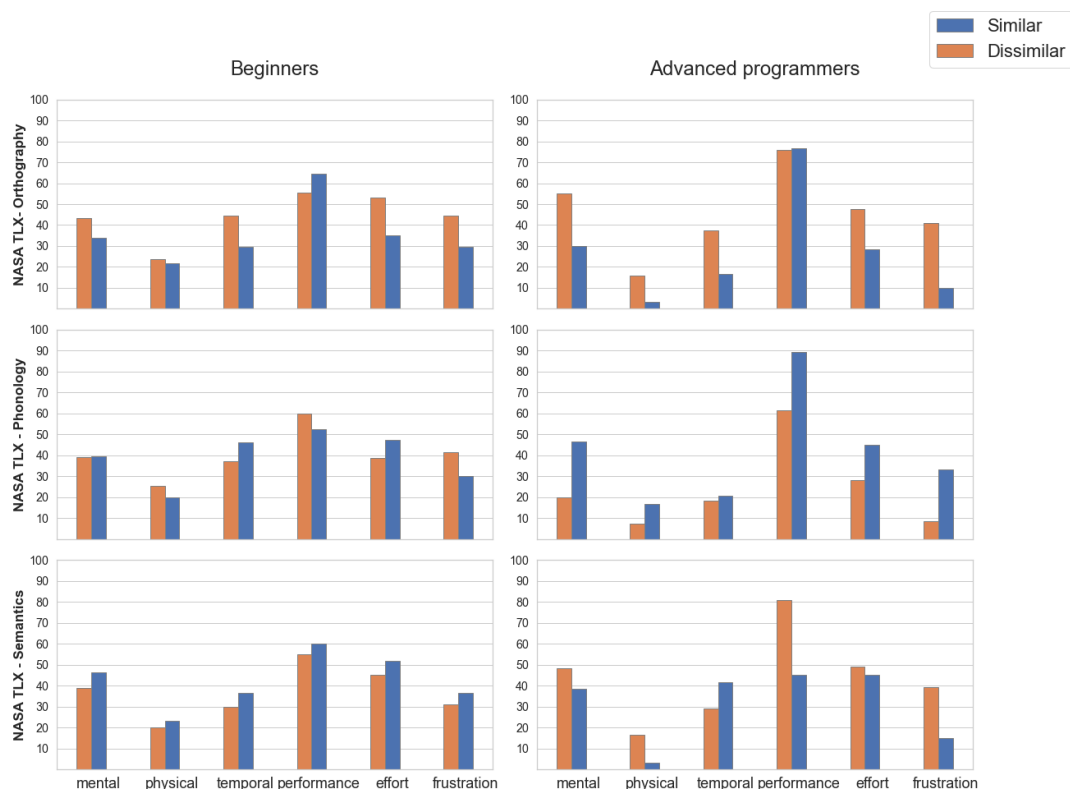


Figure 14 – Beginner programmers NASA-TLX in the main experiment.

Figure 14 further showcases breakdown scores of six components per task for both beginners and advanced programmers. All six graphs follow a general pattern: physical and temporal demand is less than mental demand, effort needed has a higher score than the frustration, and performance achieves the highest score among all six components. Intuitively comparing the NASA-TLX scores between beginners and advanced programmers, we state that experts experience less mental, physical, temporal demand, pay less effort to attain a higher performance in debugging, and meanwhile suffer from less frustration. However, after we conduct the Student T-test on the whole data, we argue that the difference is not a significant effect with $t(90) = -1.01$, $p = .32$, though advanced programmers report less overall workload ($M = 37.2$, $SD = 13.2$) than beginners ($M = 40.4$, $SD = 14.3$) do.

4. Discussion

Despite finding differences between similar and dissimilar identifiers with advanced programmers, the differences were not statistically significant, and therefore further research is needed to confirm/refute

that identifier similarity has an influence on debugging success, duration, or workload.

One of the takeaways from this paper is that comprehension is difficult to measure, and we use debugging as a “better” method of evaluating program comprehension when compared to multiple-choice questions or self-assessment. Nonetheless, it is possible that other factors, such as programming experience and debugging skill, play a bigger role in the overall success and performance in debugging compared to identifier naming. The previously mentioned study by Scanniello et al. (2017) highlights the significance of examining the different strategies programmers use when using debugging as an indicator of code comprehension.

Another important takeaway from this study is that only advanced programmers took longer and were less successful in debugging similar identifiers. Possibly, experience and repeated exposure to familiar programming structure such as ‘i’ and ‘j’ cause programmers to skip predictable code blocks where the error is located. Skipping predictable words is a well studied phenomena in natural language reading (Rayner, 1998). This possibility calls for a future experiment utilizing eye tracking to examine the skip probability of advanced and beginner programmers in relation to predictable and unpredictable identifier names.

An additional takeaway is that the three forms of similarity that we considered can potentially cause confusion in lexical access during reading. Lexical access describes the retrieval of word form, pronunciation, and meaning from memory during reading. Lexical access is usually measured on the level of millisecond (Rayner, 1998), and it is possible that the scale of our experiment (15 minutes) could mask the millisecond differences between similar and dissimilar identifiers. This idea, in combination with our results, calls for further investigation of similarity in identifier naming and its influence on code comprehension. Researchers might use a form of lexical decision task to accurately measure the difficulty or confusion that might occur on the millisecond level. It is also possible for eye tracking to shed light on hidden aspects of identifier similarity in reading code.

5. Conclusion and Future Work

Through a controlled human experiment (n=43), we explored the impact of orthographic, phonological, and semantic similarity on debugging success, time, and workload. We found that the observed differences in debugging success, duration, and workload were not statistically significant. These results and the details of the experiments we present in this paper call for further investigation of identifier similarity, with more focus on lexical access and potentially eye movement in code reading. Studying identifier similarity in code can shed light on new linguistic anti-patterns that could potentially hinder code comprehension.

Our future work includes extending several aspects of the current work. First, we would like to supplement the debugging task with other tasks to track programmers’ true understanding of codes, such as code tracing and lexical decision tasks. In addition, we can explore other comprehension-impacting factors that we did not consider in this study. One example lies in our orthographic identifier pairs. While ‘row’ and ‘col’ are orthographically dissimilar, they convey more meaning than ‘i’ and ‘j’ as an orthographically similar pair. In addition, the physical distance between the identifiers in the code is another issue. It is possible that if a pair of identifier names is hundreds of lines apart, their similarity is unlikely to impact code understanding. Last but not least, we hope to collect more data from advanced and professional programmers and to affirm/refute the results of this study with more evidence.

References

Naser Al Madi, Siyuan Peng, and Tamsin Rogers. Assessing workload perception in introductory computer science projects using nasa-tlx. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 668–674, 2022.

Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. An investigation of compound variable names toward automated detection of confusing variable pairs. In *2021 36th*

- IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 133–137. IEEE, 2021.
- Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 187–196. IEEE, 2013.
- Boehm Barry et al. Software engineering economics. *New York*, 197, 1981.
- Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2): 219–276, 2013.
- Teresa Busjahn, Roman Bednarik, and Carsten Schulte. What influences dwell time during source code reading?: analysis of element type and frequency as factors. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 335–338. ACM, 2014.
- Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265. IEEE, 2015.
- Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.
- Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. Measuring the impact of lexical and structural inconsistencies on developers’ cognitive load during bug localization. *Empirical Software Engineering*, 25(3):2140–2178, 2020.
- Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th international conference on software engineering*, pages 402–413. ACM, 2014.
- Rebecca A Grier. How high is high? a meta-analysis of nasa-tlx global workload scores. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 59, pages 1727–1731. SAGE Publications Sage CA: Los Angeles, CA, 2015.
- Sandra G Hart and Lowell E Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52, pages 139–183. Elsevier, 1988.
- Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE’07)*, pages 344–353. IEEE, 2007.
- Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What’s in a name? a study of identifiers. In *14th IEEE international conference on program comprehension (ICPC’06)*, pages 3–12. IEEE, 2006.
- Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.
- Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *PPIG*, page 11. Citeseer, 2006.
- Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–37, 2014.

- Jonathan I Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001*, pages 103–112. IEEE, 2001.
- George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.
- Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.
- Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE software*, 23(4):76–83, 2006.
- Christian D Newman, Reem S AlSuhaibani, Michael J Decker, Anthony Peruma, Dishant Kaushik, Mohamed Wiem Mkaouer, and Emily Hill. On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software*, 170:110740, 2020.
- Keith Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.
- Keith Rayner, Erik D Reichle, Michael J Stroud, Carrick C Williams, and Alexander Pollatsek. The effect of word frequency, word predictability, and font difficulty on the eye movements of young and older readers. *Psychology and aging*, 21(3):448, 2006.
- Giuseppe Scanniello, Michele Risi, Porfirio Tramontana, and Simone Romano. Fixing faults in c and java source code: Abbreviated vs. full-word identifier names. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(2):1–43, 2017.
- Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H Paterson. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports*, pages 65–86. ACM, 2010.
- Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3): 219–238, 1979.
- Janet Siegmund and Jana Schumann. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering*, 20(4):1159–1192, 2015.
- Thomas A Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, (5): 494–497, 1984.
- Armstrong A Takang, Penny A Grubb, and Robert D Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.
- Rebecca Tiarks. What maintenance programmers really do: An observational study. In *Workshop on Software Reengineering*, pages 36–37. Citeseer, 2011.
- Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.
- Anneliese Von Mayrhauser, A Marie Vans, and Adele E Howe. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, 9(5): 299–327, 1997.

Experimental Pair Programming: A Study Design And Preliminary Results

Marcel Valový

Department of Information Technologies
Prague University of Economics and Business
xvalm00@vse.cz

Abstract

In the context of facing a period in time where traditional work settings have been challenged, we require exploring new ways of increasing motivation in software teams. Pair programming is one of the agile techniques that might increase the motivation of software professionals and, as a result, also their productivity and creativity. This paper presents a work-in-progress design for conducting mixed-methods controlled experimental research on assessing, evaluating, and exploring the meaning behind the relations between personality, motivation, and programming roles.

Keywords: pair programming, experimental design, software engineering, intrinsic motivation, big five.

1. Introduction

This paper presents an experimental mixed-methods research design for establishing and measuring the strength of links between the agile software development practice of pair programming and the motivation of software professionals. Why is this important? First, the topic of pair programming is now actively discussed in both scientific and industrial communities in relation to remote work settings. Secondly, traditional motivation techniques are now less effective but, more than ever, motivated and cooperating team members are crucial for any software project's success.

During the pandemic, software engineers writing computer programs in solitude might be suffering from the frustration of the triad of basic psychological needs identified by Ryan & Deci (2017): competency, relatedness, and autonomy. A study based on factor analysis of 2225 reports from 53 countries indicates the pandemic had put stress on the psychological needs and, in turn, significantly hurt developers' wellbeing and productivity (Ralph et al., 2020). Pair programming could be employed as a remedy to satisfy the needs and boost intrinsic motivation, but it is not guaranteed to be helpful. Based on the meta-analysis of previous reports, it possibly carries both positive and adverse effects on the motivation and performance of software developers (Hannay et al., 2009a). But what effects do its roles have *separately* and do they affect programmers of different *personalities* the same? That was *not* examined by any of the existing studies, and this paper sets to present an initial work-in-progress query framework for that, together with initial results, and discuss possible extensions to the initial framework.

The author proposes a nomological network to represent the constructs of interest in this experimental design, their observable manifestations, and the inter-relations between them. The core constructs are programming role (independent variable), intrinsic motivation (dependent variable), and personality (moderating variable). To perform statistical tests, motivation and personality must be operationalized into observable variables. The former will be modeled within the Self-determination framework (Ryan & Deci, 2017) and the latter within the Big Five model (Goldberg, 1993). Data collection ought to be executed in controlled experiments and semi-structured interviews.

Research problem:

How to effectively measure the effect of pair programming on motivation in software teams.

Research questions:

RQ1: Do distinct pair programming roles affect programmers' motivation differently?

RQ2: Can psychometric tests improve the assignment of pair programming roles?

The five initially tested hypotheses were drawn on the synthesis of knowledge in multiple disciplines, including psychology, organizational behavior, and software engineering, followingly:

H1: Programmers have distinct personalities.

- H2: Distinct personality types prefer different pair programming roles.*
H3: Openness positively moderates motivation in the pilot role.
H4: Extraversion and agreeableness are essential for a motivated navigator.
H5: Neuroticism and introversion are detrimental to both pilot and navigator roles.

2. Preliminary Experimental and Methodological Design

During the winter semester of 2021, the author created a methodological and experimental design and employed it in university classrooms. The context overview with a brief explanation of methods used follows first, then the preliminary experimental design is illuminated.

2.1 Context and methods

The proposed research questions revolve around the research problem of effectively increasing motivation using agile development techniques, such as pair programming. To answer them, we opted for an experimental, mixed-methods research strategy.

In the context of a 2nd-year IT undergraduate software engineering course, three rounds of a controlled experiment were carried out. Afterward, the quantitative data from the experiments were analyzed using contemporary statistical methods to establish empirical links between personality dimensions and software engineers' attitudes (Feldt et al., 2010; Graziotin et al., 2021). Additionally, we conducted semi-structured interviews with twelve of the experiment participants after the experiments finished and evaluated them using qualitative methods such as thematic analysis (Braun & Clarke, 2006), following the precise method of theme construction proposed by Vaismoradi et al. (2016).

2.2 Experimental design

The subjects were students who signed up for an advanced software engineering course at the undergraduate university level during the winter semester of the academic year 2021/22.

The author ran three laboratory sessions of 60-minutes net programming time each, during which the ($N = 40$) subjects took a break every 10 minutes to self-report their motivation with a seven-item standardized questionnaire "Intrinsic Motivation Inventory" (Deci & Ryan, 1982), rotate in pairs and receive a new task (yet being able to continue on the previous one because the tasks were continuous).

One group of subjects worked in pairs, and the subjects in another – the "control group" – worked alone. The partners of each pair were either designated "pilot" who controls the keyboard and codes, or "navigator" who conceptualizes the solution to the given task and looks for defects, with the subjects told to switch roles every 10 minutes.

The last session was without a control group. This effectively put each individual in three different conditions or "roles" (solo, PP-pilot, PP-navigator) for 6x10 minutes, yielding 6 motivation measures for each individual in each condition.

Each individual's "preferred role" is then related to his or her personality. The preferred role is defined as the condition with the highest average reported motivation level. The subject's personalities were measured with the Big Five personality test (Rammstedt et al., 2013) at the beginning of each session.

The subjects were instructed on how to pair-program during a 60-minute *pilot session* that preceded the three experimental sessions.

The subjects were working on predefined tasks in a static order. During the first session, the purpose of the tasks was to develop a contextual menu for the first semestral project, which is an adventure game in Java with a graphical user interface in JavaFX. In the second session, the tasks were about animating elements in the game. In the third session, participants developed the core of their team projects.

No external motivators were used, i.e., no credits were given for achieving correct solutions.

The task difficulty presented the main concern to the validity of the experimental design and that is why the first two sessions contained a "control group". In each session, the motivational differences between each 10-minute round were compared in both the control (solo programming) and the test group, and statistical tests confirmed that there was, in fact, no relation between the tasks and self-reported intrinsic motivation. This is consistent with the findings of Vanhanen and Lassenius (2005).

It is also worth noting that the subjects were of various backgrounds and abilities, and this would carry a greater possible distortion on the results if performance were measured (Arisholm et al., 2007) as $performance = ability \times motivation$ (Latham, 2012). We diminished those effects by measuring intrinsic motivation which depends on autonomy, competence, and relatedness.

The effect of “pair jelling”, i.e., relative improvement after the first task mentioned by Williams et al. (2006) was tested statistically and did not project itself into the motivation results.

Pairs were allocated randomly and irrespective of personality, similar to two existing experimental studies, one with 564 students and 90 % of pairs reporting compatibility (Katira et al., 2004) and another with 1350 students and 93 % reported compatibility (Williams et al., 2006).

3. Preliminary Results

The five preliminary hypotheses were confirmed and both preliminary RQ1 and RQ2 were answered positively, both by preliminary quantitative and qualitative results. The author would like to build on these results and extend the design. First, the quantitative results are presented, followed by the qualitative ones.

3.1 Quantitative results

Of 40 students, 2 were females and 38 were males. The students' software engineering experience ranged from a half to six years ($\bar{x} = 2.2$, $\sigma = 1.5$): 19 had up to one year of experience, 13 had more than one and up to three, and 8 had more than three and up to six years of software engineering experience.

Personality variables were mapped using clustering methods. The Dunn index metric (Bezdek & Pal, 1995) indicated the usage of a complete linkage method with the number of clusters set to three. The three centroids created by the hierarchical cluster analysis in RStudio (v4.0.5) are presented in Figure 1 with their respective means and standard deviations. The first cluster is characterized mainly by the predominant personality dimension “openness to experience” ($\mu = 8.29$, $\sigma = 1.21$). The second cluster is characterized by two dominant personality dimensions, “extraversion” ($\mu = 7.36$, $\sigma = 1.36$) and “agreeableness” ($\mu = 7.91$, $\sigma = 0.83$). The third cluster is characterized by the predominant personality dimension “neuroticism” ($\mu = 7.82$, $\sigma = 1.40$) and very low “extraversion” ($\mu = 3.55$, $\sigma = 1.04$).

		O	C	E	A	N
Cluster 1	η	8.29	6.53	6.35	6.12	5.94
	σ	1.21	1.66	1.58	1.54	1.43
Cluster 2	η	6.09	6.00	7.36	7.91	5.00
	σ	1.76	1.48	1.36	0.83	1.48
Cluster 3	η	6.73	5.82	3.55	5.91	7.82
	σ	1.68	1.72	1.04	1.14	1.40

Figure 1 - Cluster centroids characterized by their means and standard deviations

The relations between clusters and preferred roles were assigned by passing each cluster to the role by the maximum intercept and are displayed in Figure 2 with their respective counts. Cluster 1 prefers the Pilot (11 matches), cluster 2 the Navigator (6 matches), and cluster 3 the Solo (6 matches) role.

	Navigator	Pilot	Solo
Cluster 1	5	11	1
Cluster 2	6	3	2
Cluster 3	3	2	6

Figure 2 - Contingency table of clusters and their preferred roles

Various statistical tests – out of this paper’s scope – confirmed the statistical counterparts of H1-5.

3.2 Qualitative results

The qualitative results, produced by coding the responses of twelve interviewees (I#01-12) and evaluating them using thematic analysis, can be briefly summarized followingly:

From the qualitative data set, a total of 68 codes with 184 occurrences and 8 themes were generated.

First, the majority of occurring codes (31 out of 68), with a number of 106 occurrences, were positive. The four overarching themes generated from them included the perceived psychological, pedagogical, and therapeutic effects and also the topics of reduced cognitive load and increased performance:

*Activation of the “Hawthorne effect”,
Separating the concerns and essential complexities of software engineering,
Improved performance,
Teaching and knowledge sharing.*

Secondly, a large proportion of analyzed codes (21 out of 68) came under a neutral tone and captured statements about the relation between the agile development practice and personality factors and stressed the importance of rules:

*Psychometric testing and personality’s moderating effect,
Importance of rules.*

Lastly, there were two negative overarching themes produced from a minority of 16 codes, where the standard argument was that it is difficult to work with introverts or when the partner is on a superior level and does not listen:

*Attention deficit and focus deterioration,
Personality and competency differences.*

4. Extending the Design

While the results provided some valuable insight, the author believes there is still much to be explored. For instance, the current design does not dwell in the space of various “pairing constellations”. Also, some of the qualitative results are contrasting and could seem contradictory. On the contrary, they represent the opposite ends of the spectrum of programmers’ attitudes. Therefore, all opinions are equally valid for implementing pair programming and enhancing the motivation of a software engineering team and must be considered. The current design, however, does not differentiate between the views of persons coming from the “openness”, “extraversion and agreeableness”, and “neuroticism” clusters. It also does not provide insight on whether the members of each cluster work equally well (and in what roles) when paired with members of different clusters, i.e. on the “pairing constellations”.

Thus, the author would like to build on the initial design and extend it followingly:

- Introduce pairing constellations. Before the experiment rounds commence, the psychometrics would be assessed, and consequently, the participants will be assigned to 9 pair constellations: Pilot-Openness/Navigator-Openness, Pilot-Openness/Navigator-Agreeableness+Extraversion, Pilot-Openness/Navigator-Introversion+Neuroticism, etc., which would be executed in two sessions and then compared to one session of participants’ motivation reports in solo settings.
- For the qualitative part of the research, representatives of each cluster should be interviewed until the point of saturation is reached for the study of each cluster’s members’ motivations, values, and attitudes, leading to cluster-specific and general overarching themes.
- Pairs should be assigned to match competency levels, i.e. to prevent the re-occurrence of the last identified theme “Personality and competency differences”.

Several new hypotheses could be drawn, e.g.:

1. *Openness pilots paired with Extraversion-Agreeableness navigators exhibit the highest increase in intrinsic motivation when compared to their solo-work intrinsic motivation.*
2. *The positive effect of pairing on intrinsic motivation is lesser for homogeneous pairs, i.e., consisting of members from the same clusters, than for heterogeneous pairs.*
3. *Members of the Neuroticism cluster prefer to work with members of the Extraversion-Agreeableness cluster rather than with the Openness or Neuroticism cluster, in both roles.*

5. Discussion

The author presented a preliminary version with preliminary results and a concept of an extended version of the design for studying the effects of pair programming on the intrinsic motivation

of software professionals using controlled experiments and expects the discussion to continue at the PPIG workshop. While the preliminary design has proven to be viable and the preliminary results provided novel insight into the psychological aspects of programming, the untapped space remains vast and awaits scientific exploration. Advanced inquiry frameworks, such as those presented by Feldt et al. (2010) and Graziotin et al. (2021) allowed for measuring the links in our proposed nomological structure in a way that was not common before. This challenges one of the conclusions of the previous seminal paper by Hannay et al. (2009b) that the moderating effect of personality on pair programming is rather insignificant.

Personality was confirmed to be a valid predictor of intrinsic motivation in software engineering. Software engineering managers could try to infer from the psychometrics of their team and act based on the findings of future studies using the extended experimental pair programming study design.

The possible threats to the external validity of studies using the presented design are cultural aspects and competency level. In different countries, the personality clusters of software professionals can differ. Thus, as Latham & Pinder (2005) note, all motivational theories and frameworks should take the context of national culture into account. The studies should therefore be replicated before being acted upon in contexts of different regions. Another threat is the competency level. The results may not be applicable to experts with more than the maximum observed number of six years of software engineering experience in our subjects.

Acknowledgments

This work was supported by an internal grant funding scheme (F4/34/2021) administered by the Prague University of Economics and Business.

References

- Arisholm, E., Gallis, H., Dyba, T., & Sjöberg, D. I. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2), 65-86.
- Bezdek, J. C., & Pal, N. R. (1995, November). Cluster validation with generalized Dunn's indices. In *Proceedings 1995 second New Zealand international two-stream conference on artificial neural networks and expert systems* (pp. 190-190). IEEE Computer Society.
- Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2), 77-101.
- Deci, E. L., & Ryan, R. M. (1982). Intrinsic motivation inventory [measurement instrument].
- Feldt, R., Angelis, L., Torkar, R., & Samuelsson, M. (2010). Links between the personalities, views and attitudes of software engineers. *Information and Software Technology*, 52(6), 611-624.
- Goldberg, L. R. (1993). The structure of phenotypic personality traits. *American psychologist*, 48(1), 26.
- Graziotin, D., Lenberg, P., Feldt, R., & Wagner, S. (2021). Psychometrics in behavioral software engineering: A methodological introduction with guidelines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1), 1-36.
- Hannay, J. E., Dybå, T., Arisholm, E., & Sjøberg, D. I. (2009a). The effectiveness of pair programming: A meta-analysis. *Information and software technology*, 51(7), 1110-1122.
- Hannay, J. E., Arisholm, E., Engvik, H., & Sjøberg, D. I. (2009b). Effects of personality on pair programming. *IEEE Transactions on Software Engineering*, 36(1), 61-80.
- Katira, N., Williams, L., Wiebe, E., Miller, C., Balik, S., & Gehringer, E. (2004, March). On understanding compatibility of student pair programmers. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education* (pp. 7-11).
- Latham, G. P., & Pinder, C. C. (2005). Work motivation theory and research at the dawn of the twenty-first century. *Annual review of psychology*, 56(1), 485-516.

- Latham, G. P. (2012). *Work motivation: History, theory, research, and practice*. Sage.
- Ralph, P., Baltes, S., Adisaputri, G., Torkar, R., Kovalenko, V., Kalinowski, M., ... & Alkadhi, R. (2020). Pandemic programming. *Empirical Software Engineering*, 25(6), 4927-4961.
- Rammstedt, B., Kemper, C. J., Klein, M. C., Beierlein, C., & Kovaleva, A. (2013). A short scale for assessing the big five dimensions of personality: 10 item big five inventory (BFI-10). *methods, data, analyses*, 7(2), 17.
- Ryan, R. M., & Deci, E. L. (2017). *Self-determination theory: Basic psychological needs in motivation, development, and wellness*. Guilford Publications.
- Vaismoradi, M., Jones, J., Turunen, H., & Snelgrove, S. (2016). Theme development in qualitative content analysis and thematic analysis.
- Vanhanen, J., & Lassenius, C. (2005, November). Effects of pair programming at the development team level: an experiment. In *2005 International Symposium on Empirical Software Engineering, 2005*. (pp. 10-pp). IEEE.
- Williams, L., Layman, L., Osborne, J., & Katira, N. (2006, July). Examining the compatibility of student pair programmers. In *AGILE 2006 (AGILE'06)* (pp. 10-pp). IEEE.

Keynote

The Psychology of Programming and the Psychology of Mathematics

Henry Lieberman (work with Warren Robinett)
MIT Computer Science & AI Lab

Many fundamental results in mathematical logic and set theory, among other mathematical fields, were developed long before the advent of computers and modern computer science. Many of these proofs depend crucially on symbol-manipulation procedures, originally described in the proofs with prose or with equations that specify constraints on the procedures. We recast some of these procedures with modern computer science concepts, in modern programming languages, and discover (perhaps not surprisingly), that... some of them have bugs.

One reason that these bugs may have gone unnoticed for so long may be due to differences in how mathematicians and programmers think about procedures. Programmers think about control structures and temporal relationships, whereas mathematicians want to abstract away from time. The equation is ubiquitous in mathematical language, whereas programmers have discovered the "=" can have different roles: definition, equality testing, and assignment. Programmers think computationally, whereas mathematicians are comfortable with "there exists" descriptions, which may prove uncomputable. Finally, the two fields have different ideas about the notion of abstraction, which affect how their concepts are introduced and how they are used.

A Tour Through Code: Helping Developers Become Familiar with Unfamiliar Code

Grace Taylor

Microsoft

grace.taylor@microsoft.com

Steven Clarke

Microsoft

steven.clarke@microsoft.com

Abstract

Becoming familiar with an existing code base can be a challenging and time-consuming process. A developer who is new to a code base and needs to implement a new feature or fix a bug might browse and search the code base for locations providing good starting points for their task. From these starting points they may then navigate to other locations in the code base by traversing semantic and other relationships in the source code that may be relevant to their task. Although most developer tools provide ways to discover some of these different relationships, the relevance of specific relationships can often only be determined after traversing the relationship. This often necessitates some amount of backtracking as a developer navigates one relationship to a new destination in the source code, only to find that the destination is not relevant to their task. Furthermore, many such relationships between different locations in source code cannot easily be determined by developer tools alone. This paper presents the results of a study that investigated the use of CodeTour, a tool that enables the creation and presentation of annotated code tours through a code base. Such tours can indicate relationships between locations in a code base that are relevant to a task, but which could be difficult to identify using regular features found in developer tools. The paper describes the extent to which these code tours helped developers find the source of bugs and compares the experience with those who attempted the same task without the tool.

Introduction

Every time a software developer joins a new project or forks a new repository, they need to become familiar with the code in that project or repository. This onboarding process can involve a considerable amount of time, money, and effort (Dagenais et al., 2010, Begel & Simon, 2008). The result is that it can take some time before a new developer feels productive. One way to help developers onboard to a new codebase is to provide up-to-date documentation. However, even though up to 11% of the cost of a software development project can be attributed to documentation (An Overview of the Software Engineering Laboratory, 1994), the resulting documentation might not provide what developers really need (Aghajani et al., 2019).

Documentation content is often incomplete, outdated, or incorrect (Aghajani et al., 2019). In some cases, documentation may even lack any useful content at all. For example, an examination of documentation across two large frameworks indicated that a high proportion of the documentation contained no information and merely rehashed API names (Maalej & Robillard, 2013).

Resource constraints significantly impact a developer's ability to write documentation. (Stetting and Hiejstek, 2011) reported that developers they surveyed were able to spend on average 15 minutes per day creating documentation for their projects. Many implied that they felt this was insufficient time to spend on documentation, saying that they felt there was too little documentation available for the projects they work on.

The type of documentation that developers create and the type of documentation that they need can vary. Code comments are one of the most common types of documentation and are lightweight in nature, but often lack substance (Aghajani et al., 2020). On the other hand, tacit knowledge is considered crucial yet consistently missing from documentation. For example, knowledge related to the performance of the software or the environment in which it runs, differences between different versions etc is not often found in documentation, yet developers find this kind of information important for their work (Maalej & Robillard, 2013, Robillard & DeLine, 2010).

Documentation does not always have to come in the form of written text. In some cases, mentorship can be effective (de Janasz et al., 2003), but searching for the right mentor costs time. But even once the right mentor is found, they may not always be available to share their insights and expertise (Ko et al., 2007).

Understanding the ways that developers seek information about a new codebase can help identify reasons why existing documentation might fail. For example, (Sillito et al., 2008) lists the questions that developers ask themselves when learning a new codebase. In many cases, the answer to the question is found by piecing together multiple pieces of information that are found in multiple locations throughout the codebase (Ko et al., 2007). One code comment alone often does not suffice.

But stitching together multiple sources of information to answer one question can be difficult. Navigating between the disjointed external documentation and multiple files inside the codebase adds unnecessary complexity (Robillard & DeLine, 2010). Furthermore, there may be differences in the ways that different developers approach an information seeking task. Some strategies may be more likely to lead to success than others.

When working on certain programming problems, developers may be faster at finding information they need if they are more conscious of the different problem-solving strategies that they adopt (Loksa et al., 2016). Tools that prompt developers to ask and answer a different range of questions when learning code may help those developers learn about and onboard to the codebase independently.

Our insights about how developers onboard to a new codebase have also been informed through informal conversations we have had with developers. In those conversations we learned about what those developers have done to help others onboard to a new codebase. We spoke with eight developers who had used CodeTour, an extension for Visual Studio Code, which allows developers to create one or more 'tours' within a codebase, where a tour is a linked list of nodes, with each node containing a location in the source code (file name and line number) and text. We wanted to learn more from these developers about the kind of tours they created and the impact they had on the onboarding process.

The developers we spoke to suggested that code tours provided a better way to communicate tacit information such as little-known quirks around compiling and building the codebase. They also suggested that code tours help identify important locations in the codebase and the relationships between those locations, helping them become more familiar with the codebase.

This has motivated us to explore if code tours can address the learning and onboarding challenges identified by Robillard & Deline (2010), Ko et. al (2007) and Sillito et. al (2008) when developers collect information from multiple sources through the codebase.

In this paper we describe CodeTours, an extension for Visual Studio Code that allows developers to create tours in a code base. Then we describe the results of a usability study that aimed to validate the following hypotheses:

H1: Code tours provide a useful starting point for people orienting themselves to a new code base

H2: Code tours encourage developers orienting to a new code base to explore more of the code base than they would have done otherwise

CodeTour

CodeTour is a documentation tool that allows developers to create code tours within Visual Studio Code, a popular code editor. A code tour is a linked list of two or more nodes, where each node contains a location (file name and line number) in the source code and some descriptive text. CodeTour visualizes nodes in a tour by drawing an icon in the left margin of the editor adjacent to the specific location.

When that node is selected, the associated text for that location opens in a separate window within the code (see Figure 1).

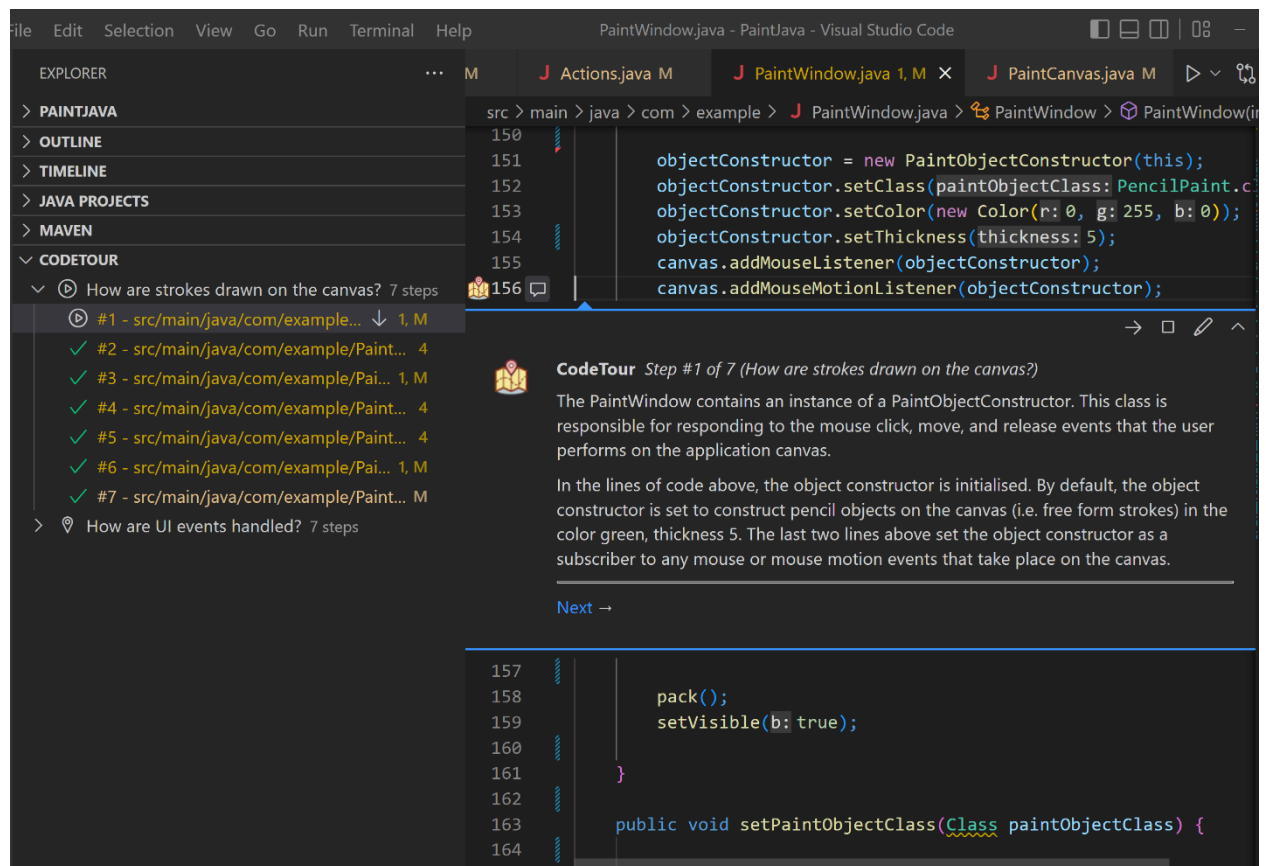


Figure 1 A screenshot from VS Code showing a step within a CodeTour opened in the editor on the right hand side and the full table of contents for the tour on the left hand side.

A table of contents view allows developers to view every node in the tour and to navigate to any of those nodes (see Figure 1).

Related Work

Multiple tools and prototypes have been developed that tackle different aspects of orienting to a codebase. Chat.code (Oney et al., 2018), for example, is a project that allows developers to text chat with each other inside the codebase, where messages are persisted to lines of code. Developers can reference current or previous versions of code in their chat messages and respond to each other over time. Documentation does not need to be kept up to date because developers can follow chat threads to learn about changes. Chat.code allows developers to reference previous versions of code in chat threads to tell a story about the evolution of the code. This helps with rich contextual information about code but does not provide threads throughout code.

Codepourri (Gordon & Guo, 2015) is a project that crowdsources annotations inside code by prompting users to compare and write annotations, and visualizing its execution state. The system asks all learners who are reading the tutorial to contribute annotations and vote on the best annotation to display. Instructors can create and share tutorial links with students. A given tutorial will remain up to date when users are incentivized to share and vote on annotations. The study authors write that users are more likely to contribute annotations when there are already existing annotations in the tutorial. Since the content is crowdsourced, the tutorial increases in quality when a higher volume of learners contribute to voting and writing. However, this tool does not address quality or trustworthiness issues for new tutorials who do not have many users.

Other projects, such as Adamite (Horvath et al., 2022), are developing tools that allow developers to leave annotations throughout a document without having to write code comments. Adamite allows

developers to categorize their own annotations, and when researchers coded the annotations from authors, they found that many of the annotations were redundant. The researchers also found that a lot of developers used the tool as a mechanism to leave notes to themselves. Adamite shows points of interest inside documentation dynamically, but relies on crowdsourced information to display accurate information. We found that some developers used CodeTour to leave notes to self, but this will not be the focus of this paper.

In the area of learning new technology from a vendor or external company, Codelets (Oney & Brandt, 2012) allows developers to copy a block of sample code containing interactive code context into an editor. It also proposes a “builder interface” and a “helper” in code which could write most of the code for the user, but requires them to set parameters. After the user copies the sample code into the editor, they can view a description of the snippet along with helper components, which aim to help the user understand and integrate the sample code into their own work. As a result, learners in the treatment group spent more time customizing their pasted sample code before running it, compared to the control group. Some developers copied and pasted sample code before understanding its explanation, which has the risk of introducing errors in one’s code. Codelets is most helpful as part of sample code documentation on a website where users assume the resource is trustworthy. However, Codelets does not support editing or sharing annotations with the sample code.

Some tools focus on providing different views of code. Code Maps (DeLine et al., 2010) creates diagrams that represent a codebase for different scenarios in hopes that new developers may better understand the codebase. The researchers generated these Code Maps by collecting examples of developers’ visual representations of different codebases, including sketches and diagrams, to find conventions for representing a codebase. It shows relationships between artifacts and pieces of a codebase, but it does not show paths or points of interest inside code. While CodeTour shows different views of the codebase through paths inside code and a table of contents, its focus is to share tacit knowledge through text, not create diagrams to describe relationships.

Other tools focus on showing threads between developer artifacts. For example, Codebook (Begel et al., 2010) describes a broad relationship analysis framework, which shows links between different artifacts such as work items, emails, and code files. The researchers implemented Codebook in two contexts, in an artifacts search, where users can search for artifacts along with the people who can be helpful, and in Deep IntelliSense, where users can trace that block of code’s change history and the people associated with that artifact. Codebook attempts to solve software team coordination problems by allowing developers to trace threads of interaction between team mates, but does not show paths throughout a codebase.

Another tool, Team Tracks (DeLine et al., 2005), is a recommender system that automatically identifies relationships throughout code by recording how frequently users navigate a certain path throughout a codebase, so that the system can guide new developers by recommending popular code paths. The Team Tracks recommendation system automatically generates a path without a description or context, whereas CodeTour’s paths are manually defined but offer detailed descriptions.

Method

In this study, our aim was to determine if CodeTour provided value to developers when working with an unfamiliar code base. We did not study how easily CodeTours could be created, or indeed how motivated developers would be to create CodeTours in the first place. We believe that without evidence that CodeTours would provide any value on top of existing annotation and code commenting practices, it would be difficult to motivate developers to create CodeTours.

To investigate the effect of using CodeTour to become familiar with an unfamiliar code base, we created two code tours for a Java code base that implements a simple Paint program (the same code base that was used in Amy Ko’s work on the Whyline debugger – Ko and Myers, 2008). The code base consists of 8 source files and a total of 598 lines of code (including white space). The application is implemented using the standard AWT and Swing libraries. The tours provide annotated paths through distinct locations in the code base. Each tour describes how the actions that a user takes results in strokes being

drawn on the main canvas and how UI events result in different commands being invoked in the application.

We recruited fifteen participants to take part in the study. Each participant had a minimum of two years professional experience with Java and spent at least 20 hours per week writing Java code. Each participant was given access to a computer that was set up with a code editor, the code base, and the Java runtime. They were asked to work on two tasks, thinking aloud, while we observed remotely:

- Task 1: The undo button does not undo the last stroke drawn on the canvas immediately. Find out the cause of this bug.
- Task 2: The tool to draw straight lines on the canvas has not been implemented. Write code that implements this tool.

One group of seven participants was provided with the two code tours, while the other group of eight participants was not. The code tours did not explicitly describe answers to either of the tasks but provided information about how UI events are handled in the application and how strokes are drawn on the canvas.

Participants were given one hour to work on these tasks. Participants worked on the tasks in sequence, not starting the second task until they completed the first task or until they had spent approximately 30 minutes on the first task. They were permitted to use any of the tools available to them in the code editor such as search, goto definition, find all references, and the debugger. As they worked, they were asked to talk aloud to describe what they were thinking as they worked.

Participants who were given access to CodeTour were presented with a two-minute demo of the functionality before they started work on the task. These participants were told to make use of the code tours while working on the tasks.

Measures

The audio and video (showing the code editing screen that participants worked on) of each session was recorded. Before performing each task, we asked participants how confident they were that they would be able to complete the task successfully. Participants responded using a five-point Likert scale where the lower value (1) represented not confident at all and the higher point (5) represented completely confident. At the end of each task, we also asked them how well they thought they understood the codebase. We used a similar scale with 1 representing no understanding and 5 representing complete understanding.

We recorded the time that each participant spent on each task, starting the timer when participants first started reading the task instructions and stopping when they told us they were either satisfied that they had completed the task or had reached a point where they were unable to continue. Due to differences in how participants carried out each task, in particular the extent to which they talked out loud while working, the time spent working on tasks does not serve as a uniquely reliable indicator of the impact of CodeTour, but combined with other measures it can provide some useful insight.

The files that participants opened while working on the tasks, the number of times these files were active in the editor, and the amount of time that each file was active in the editor were recorded.

Results

In this section, we will refer to participants in the CodeTour group as (*ctn*) and participants in the non-CodeTour group as (*noctn*).

Due to the small sample size, none of our results are statistically significant. However, we do believe that our observations raise some interesting questions worthy of discussion and invite further exploration.

Table 1 presents the success, confidence and understanding scores for all participants across both tasks. Seven participants were successful in task 1, and two participants were successful in task 2.

	Task 1			Task 2		
Participant	Success	Confidence	Codebase understanding	Success	Confidence	Codebase understanding
ct1	Yes	5	4	No	4	3
ct2	No	4	4	No	<i>No report</i>	3
ct3	No	3	4	No	3	3
ct4	Yes	5	<i>No report</i>	No	5	4
ct5	Yes	3	3	Yes	3	4
ct6	No	2	3	No	<i>No report</i>	3
ct7	Yes	5	4	No	3	5
noct1	No	2	2	No	1	2
noct2	No	2	3	No	2	2
noct3	No	2	<i>No report</i>	n/a	<i>n/a</i>	<i>n/a</i>
noct4	No	5	<i>No report</i>	No	3	<i>No report</i>
noct5	Yes	4	3	No	<i>No report</i>	3
noct6	Yes	4	3	No	3	4
noct7	No	3	2	No	2	3
noct8	Yes	3	<i>No report</i>	Yes	2	<i>No report</i>

Table 1 Success, confidence and understanding scores for both groups of participants across both tasks.

We did not see any major difference in success across both tasks. The ct participants were slightly more successful in task 1 but had the same success rate in task 2. However, we did see a difference in terms of participants' self-assessment of how well they understood the codebase after each task.

Given the small sample size it is not possible to point to any meaningful difference in the confidence or understanding scores between the ct and noct participants. From our own observations though, participants in the ct group performed slightly better when identifying areas of concern in code for explaining the undo bug.

Identifying cause of bug in task 1 – ct group

For the first task, the crucial code is the undo() method exposed by the PaintCanvas class, contained in the file PaintCanvas.java. This code is called by the PaintWindow.undo method which is in turn called by the undoAction.actionPerformed method defined in the Actions class. Figure 2 shows the sequence of calls that result in PaintCanvas.undo() being called.

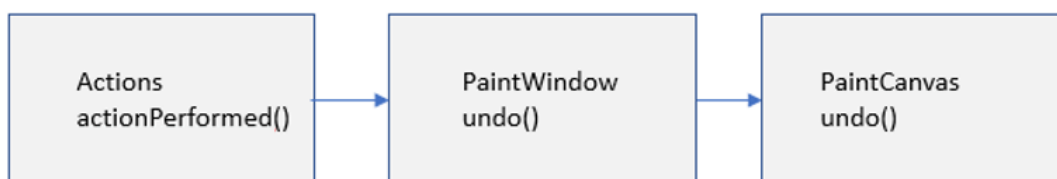


Figure 2 The sequence of method calls resulting in PaintCanvas.undo() being called.

During the first task, we observed the CodeTour participants using the code tour to follow paths around and within the undo() method. The code tours do not describe how the undo functionality works but they do describe how other similar functionality works.

As a result, we observed CodeTour participants focusing their efforts on the files that were contained in this path whereas the non CodeTour participants spent time looking in other files that were not related to the task. Table 2 shows the average number of times that a file was active in the editor (active means that focus was placed inside that editor) across all participants. For this task, the files Actions.java, PaintWindow.java and PaintCanvas.java were crucial to being successful since these contained the implementations of the actionsPerformed() and undo() methods, highlighted in Figure 2. The table shows that the files Actions.java and PaintCanvas.java were most active for those that used CodeTour and for those that did not. Both groups spent a fair amount of time in PaintCanvas.java. However, the noct group activated this file 10 times whereas the ct group activated this file 5 times, which implies that the noct group spent more time opening this file then focusing attention on another file, then returning.

In addition, the table shows that files such as EraserPaint.java and PencilPaint.java, which are not relevant to the task, were not active at all for CodeTour participants. Those that did not use CodeTour though spent some time browsing the code contained in these files.

Task 1: Average time and number of times active in each file per participant				
	Did not use code tour		Used code tour	
Time on task	26:31		23:35	
	Time in File	Active	Time in File	Active
Actions.java	04:23	6	03:40	6
EraserPaint.java	00:20	2	00:00	0
PaintCanvas.java	09:34	10	08:14	5
PaintObject.java	00:13	2	00:02	1
PaintObjectConstructor.java	01:31	2	02:10	2
PaintObjectConstructorListener.java	00:01	1	00:00	0
PaintWindow.java	07:05	10	06:13	10
PencilPaint.java	00:18	1	00:00	0

Table 2 Number of times that a file was active in the editor while participants worked on task 1.

One CodeTour participant, ct1, was able to correctly explain why the bug occurs and describe the logic for addressing the bug.

Figure 3 shows the code that contains the bug. When referring to this code, participant ct1 correctly identified the problem, saying “when it removes the last element from paint object, it should try to repaint it again”.

```
public void undo() {

    paintObjects = history.lastElement();
    history.removeElement(history.lastElement());
}
```

Figure 3 The code for the PaintCanvas.undo method.

The other CodeTour participants, ct2 and ct3, followed similar paths to and around the undo() method but never fully realized the cause of the issue. They spent time focusing on the history, lastElement, and

removeElement methods. They believed that the bug was within code that was already present in the function and did not consider that the bug might have occurred due to missing code. The code tour that they had followed had described how similar functionality works in the application and had highlighted calls to repaint when they occurred. However, even though they had browsed this tour and read the methods implementing similar functionality, they were not aware of the significance of the repaint() method.

Utilizing the CodeTour

We observed some differences in the way that participants used the CodeTour.

Participant ct1 was the only participant who was able to complete task 1 successfully. They spent the first 4.5 minutes of the task reading through the first CodeTour.

In contrast, participants ct2 and ct3 quickly skimmed through the CodeTour annotations at the start of the task. Participant ct2 never returned to either of the tours, but participant ct3 returned to either of both code tours four times throughout the course of their time working on the task. Each repeat visit to the tour was very brief though. Figure 4 shows a timeline view of ct3's activity over a roughly 20 minute period of time while working on the task.

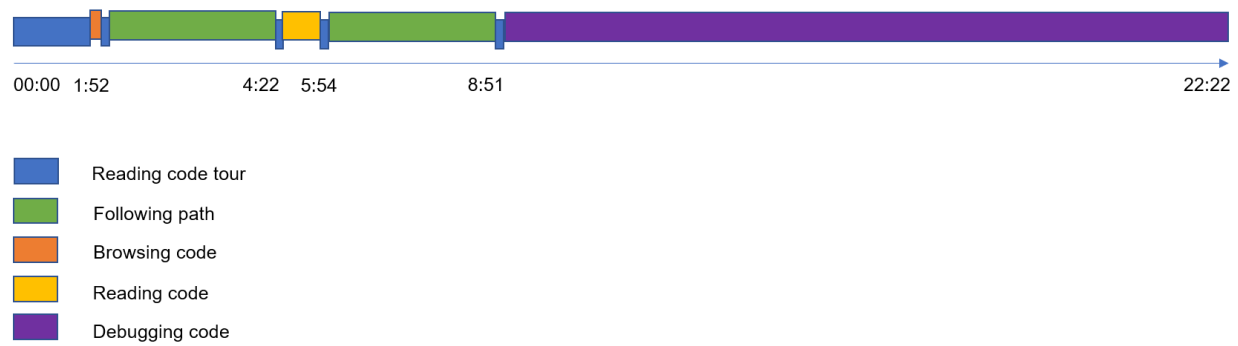


Figure 4 A timeline showing a sequence of activities that participant ct3 performed while working on Task 1.

In the timeline view, coloured blocks represent distinct activities that the participant performed. The length of the block indicates how long the participant performed that activity. Transitions between activities are indicated by a change in colour and a change in position of the activity block. The key underneath the timeline describes the activity that is represented by each coloured block. Most are self-explanatory with the exception of “Following path.” This activity represents any time a participant uses any of the code navigation tools inside the editor, such as “goto definition”, “find all references”, “goto implementation” two or more times in succession. Effectively the participant is linking together and traversing two or more locations in source code, following a path from one location to another.

The timeline shows that the participant spent approximately two minutes reading the code tour then returned to it briefly four times (the very short blue blocks that intersperse other longer periods of activity).

It would appear that this participant used the CodeTour to identify points of interest inside the codebase to navigate to. He rarely referred to the file structure of the codebase and instead selected a particular step in a code tour to navigate to that corresponding location in code.

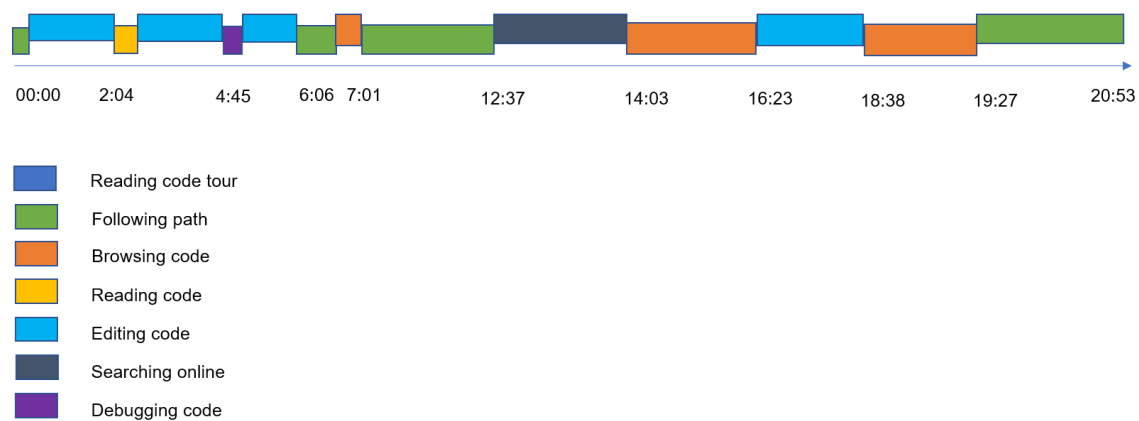


Figure 5 A timeline showing a sequence of activities that participant noct4 performed while working on Task 2.

In contrast, the timeline view shown in Figure 5 shows that participant noct4 spent a lot of time jumping between different activities while working on task 2. Noct4 had almost completed the task but was missing one key line of code that was preventing them from completing the task successfully.

After 14:03, the participant spent the remainder of the time browsing code to search for some clues that might help them complete the task. They ended up following multiple paths and browsing multiple files without finding anything that helped them.

Identifying cause of bug in task 1 – noct group

In the non-CodeTour group, participants browsed and searched through more files than the CodeTour group. None of the non CodeTour participants were able to identify the cause of the bug.

For example, participant noct2 was observed continuously switching between different files to try to identify the location of the bug. At first, he relied on the names of files to try to identify potential sources of the bug. He initially believed that the file `EraserPaint.java` must contain the code responsible for the bug and the undo functionality (“I am not sure how this works, the issue must be with `EraserPaint`”). Other participants behaved similarly, looking at files like `EraserPaint.java` and `PencilPaint.java`, in contrast to code tour participants who did not look at these files at all (see Table 2).

Implementing line tool in task 2 – ct group

For the second task, participants were required to implement a new class that would be responsible for drawing straight lines and then write code that would wire the instantiation of that class to the appropriate UI control. Participants could reference the implementation of the Pencil tool while implementing the line drawing tool. But to figure out how to wire that new class up to the UI control they would have to discover how paint objects are constructed. As in task 1, there were intermediate classes involved, with the `PaintObject` and `PaintObjectConstructor` classes being responsible for creating instances of the pencil and line drawing tools.

Table 3 shows the average amount of time each participant spent in each file and the number of times they opened that file while working on task 2. We don’t see a major difference between the two groups, but the file `PaintWindow.java` was crucial, and this was where many participants got lost.

Task 2: Average time and number of times active in each file per participant				
	Did not use code tour		Used code tour	
Time on task	21:25		23:19	
	Time in file	Active	Time in file	Active
Actions.java	03:13	4	04:37	5
EraserPaint.java	00:10	1	00:17	1
PaintCanvas.java	01:54	1	00:43	2
PaintObject.java	00:07	1	00:14	1
PaintObjectConstructor.java	00:54	1	01:59	2
PaintObjectConstructorListener.java	00:42	1	00:01	1
PaintWindow.java (Useful file)	01:53	3	05:28	5
PencilPaint.java (Useful file)	02:28	4	02:54	6

Table 3 Number of times that a file was active in the editor while participants worked on task 2.

Two participants, ct5 and noct8 were the only two participants able to complete both pieces of functionality for this task. Most participants were able to implement a class that would be responsible for drawing straight lines simply by referring to the existing implementation of the PencilPaint class.

Participant noct7 created a LinePaint file after inspecting the pencilAction.java file. However, after meandering between the different files, he gave up on the task (“This is as far as I’m going to get”).

Discussion

Our results suggest ways that tools like CodeTour may influence how developers become familiar with an unfamiliar code base.

We attempted to validate the following hypotheses:

H1: Code tours will provide a useful starting point for people orienting themselves to a new code base

H2: Code tours will encourage developers orienting to a new code base to explore more of the code base than they would have done otherwise

Our observations lead us to believe that CodeTour could provide a useful starting point is by making it clearer which files are relevant to a task and which are not. During the study, we observed that participants in the code tour group did not look at PencilPaint.java or EraserPaint.java during task 1 while those in the non code tour group did. Indeed, we observed non code tour participants using cues like file names as their way to forage for potentially relevant code. In such cases, ambiguity and lack of clarity in names can lead to time wasted browsing irrelevant code.

It is interesting that even in such a small codebase we observed these differences. With only eight files in the whole code base, it is difficult to spend a significant amount of time browsing irrelevant code. However, in a larger, more realistic code base with many tens or hundreds of files, it is conceivable that a developer could spend a large amount of time browsing irrelevant code as they forage for code that could help them complete their task. In these situations, CodeTour could potentially help by identifying files that are relevant.

Thus we believe that CodeTour does have some promise in helping developers orient to a new code base but this is still to be verified.

Similarly, another interesting observation was the way that some CodeTour participants used the CodeTour to navigate to interesting locations in code. We observed participants using the tours to navigate to locations in code that were associated with particular steps in a tour, rather than by file

name. We believe that these participants began to use the code tour as their primary way of referencing code, rather than referencing the code by the file that it was contained in.

This does suggest that a well-written code tour could provide a useful starting point for developers orienting to a new code base but it clearly depends on the availability and quality of the code tour.

However, both these findings do suggest that our second hypothesis would be invalidated. We saw that code tour participants activated fewer files than the non code tour participants, for example. It is possible that the code tour could actually discourage developers from exploring parts of the code that aren't covered by a code tour, perhaps thinking that since they are not covered by a code tour that they are not interesting or relevant. While reducing the amount of time spent browsing irrelevant code is worthwhile, reducing opportunities for serendipitous discoveries of useful code simply due to it not being contained in a code tour would be potentially an even worse outcome.

The scenario that our participants were placed in also had a bearing on our observations. We placed participants in the scenario of maintaining a legacy application. In such a situation it might be the case that developers aren't as motivated to spend time learning the code base. Perhaps all they care about is learning enough to be able to complete their task (fix a bug, add a new feature) and then move on. We believe that we observed some of this behaviour during our study as we saw that most CodeTour participants didn't initially care to learn the codebase. Two CodeTour participants were only motivated to fix the bug and move to the next task. They considered the cost of reading through a code tour to the benefits that they expected to gain. They weren't as motivated to learn about the code base since they thought that they just needed to do enough to fix the bug.

Another participant though suggested that they are always conscious about the risk of introducing a new bug while fixing an existing bug. They carefully read and understood the CodeTour before proceeding but it is clear that other developers might look at the cost-benefit analysis differently. If the developer is addressing what they consider to be a trivial bug in a legacy code base, the cost to learn the codebase could be higher than the effort involved to fix the bug.

One factor which would affect how developers determine if the cost of browsing a code tour is worth it is their reaction to a largely text based medium. One CodeTour participant said that the CodeTour was overwhelming at first: *"Coming into the first task I think I was confused with all the source code, you know the tool tips given etc. I got confused. Instead of debugging, I spent a lot of time in understanding the structure of the code."* They felt that they might have had a better time stepping through the code in the debugger to build up an understanding of how the code works, rather than reading the steps in each code tour (what they referred to as the tool tips).

On reflection, although there was not a clear difference in success rates between those participants that had access to the code tours and those that did not, participants that used the code tours suggested that they were useful in helping them get started and understanding the flows through the code. For example, after using the code tours, participants were asked for their thoughts:

- ct1: *"Code tours were really useful. Normally for this application there are many classes, so it guided me... to a particular spot on the code and it was like properly displayed in a great UI form, and I was able to figure out what this functionality is for...Because the application which I work on are performance dependent like even a small code change [if] it goes wrong it can be disastrous. So, like, I have to carefully see all the flows which are possible... the CodeTour was very useful because it redirected me to the particular function it was trying to describe and it was presented in a more UI friendly way, so it was very readable and I was able to understand everything."*
- ct2: *"Without the code tour that would have taken me more time to find where is the start point."*
- ct3: *"CodeTour was really helpful, description was really helpful. The navigation was really helpful. After the first task I came to realize CodeTour really helped."*
- ct5: *"It instills a level of confidence that somebody has thought this through."*

In conclusion, we believe that our study invites future exploration into how developers learn, navigate, and ask questions about code.

Limitations and threats to validity

Given the small number of participants in the study, our results are not statistically significant and so we cannot make any claims about the impact of CodeTour.

Furthermore, the scenario in which we observed participants was limited to that of spending a short amount of time addressing a bug and implementing a new feature in a legacy codebase. We did not observe participants in a scenario where they were working with a new codebase over an extended period of time.

Considering the codebase is small and the study length was one hour, on average all participants spent a roughly equal amount of time in the most relevant classes, but a typical legacy codebase would be longer and include code written by many different developers, and so developers would find it more difficult to find the most relevant classes.

Another limitation is that the study participants were all male. We know from the work of (Burnett et al, 2016) that there are differences between males and females in how they approach tasks like these. We have not been able to observe how female developers would use CodeTour to approach these same tasks.

Many developers are able to ask for help from mentors or colleagues when working on tasks like the ones we used in our study. However, to avoid leading or biasing participants we declined to answer any questions about the codebase.

Not all participants were familiar with the debugger in Visual Studio Code and in several instances, we coached participants to operate the debugging controls, which decreased the time available to complete the task.

CodeTour's usefulness is limited by the authors' ability to share helpful annotations, and for this study, we authored and refined two sets of accurate steps, which may not be realistic for every application of the tool.

References

- Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., & Shepherd, D. C. (2020). Software documentation. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. <https://doi.org/10.1145/3377811.3380405>
- Aghajani, E., Nagy, C., Vega-Marquez, O. L., Linares-Vasquez, M., Moreno, L., Bavota, G., & Lanza, M. (2019). Software documentation issues unveiled. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. <https://doi.org/10.1109/icse.2019.00122>
- An Overview of the Software Engineering Laboratory [PDF]. (1994). <https://ntrs.nasa.gov/api/citations/19950022293/downloads/19950022293.pdf>
- Begel, A., & Simon, B. (2008). Novice software developers, all over again. *Proceeding of the Fourth International Workshop on Computing Education Research - ICER '08*. <https://doi.org/10.1145/1404520.1404522>
- Begel, A., Khoo, Y. P., and Zimmermann, T. (2010). Codebook: discovering and exploiting relationships in software repositories. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. <https://doi.org/10.1145/1806799.1806821>
- Burnett, M., Peters, A., Hill, C., and Elarief, N. (2016). Finding Gender-Inclusiveness Software Issues with GenderMag: A Field Investigation. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, New York, NY, USA, 2586–2598. <https://doi.org/10.1145/2858036.2858274>

- Dagenais, B., Ossher, H., Bellamy, R. K. E., Robillard, M. P., and de Vries, J. P. (2010). Moving into a new software project landscape. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. <https://doi.org/10.1145/1806799.1806842>
- de Janasz, S. C., Sullivan, S. E., & Whiting, V. (2003). Mentor networks and Career Success: Lessons for Turbulent Times. *Academy of Management Perspectives*, 17(4), 78–91. <https://doi.org/10.5465/ame.2003.11851850>
- DeLine, R., Venolia, G., and Rowan, K. (2010). Software development with code maps. *Communications of the ACM*, 53 (8), 48–54. <https://doi.org/10.1145/1787234.1787250>
- DeLine, R., Czerwinski, M., and Robertson, G. (2005). Easing program comprehension by sharing navigation data. *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. <https://doi.org/10.1109/VLHCC.2005.32>
- Gordon, M., and Guo, P. J. (2015). Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing*. <https://doi.org/10.1109/VLHCC.2015.7357193>
- Horvath, A., Liu, M. X., Hendriksen, R., Shannon, C., Paterson, E., Jawad, K., Macvean, A., and Myers, B. A. (2022). Understanding How Programmers Can Use Annotations on Documentation. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. <https://doi.org/10.1145/3491102.3502095>
- Ko, A. J., DeLine, R., and Venolia, G. (2007). Information Needs in Collocated Software Development Teams. *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*. <https://doi.org/10.1109/ICSE.2007.45>
- Ko, A. J., and Myers, B. A. (2008). Source-Level Debugging with the Whyline. *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '08)*. Association for Computing Machinery, New York, NY, USA, 69–72. <https://doi.org/10.1145/1370114.1370132>
- Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J. and Burnett, M. M. (2016). Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, New York, NY, USA, 1449–1461. <https://doi.org/10.1145/2858036.2858252>
- Maalej, W., & Robillard, M. P. (2013). Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, 39(9), 1264–1282. <https://doi.org/10.1109/tse.2013.12>
- Oney, S., Brooks, C., and Resnick, P. (2018). Creating Guided Code Explanations with chat.codes. *Proceedings of the ACM Human-Computer Interaction*, 2(CSCW), 1–20. <https://doi.org/10.1145/3274400>
- Robillard, M. P., & DeLine, R. (2010). A field study of API learning obstacles. *Empirical Software Engineering*, 16(6), 703–732. <https://doi.org/10.1007/s10664-010-9150-8>
- Sillito, J., Murphy, G. C., & De Volder, K. (2008). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4), 434–451. <https://doi.org/10.1109/tse.2008.26>
- Stettina, C. J., and Heijstek, W. (2011). Necessary and neglected? an empirical study of internal documentation in agile software development teams. *Proceedings of the 29th ACM international conference on Design of communication (SIGDOC '11)*. <https://doi.org/10.1145/2038476.2038509>

What is it like to program with artificial intelligence?

Advait Sarkar
Microsoft Research
University of Cambridge
advait@microsoft.com

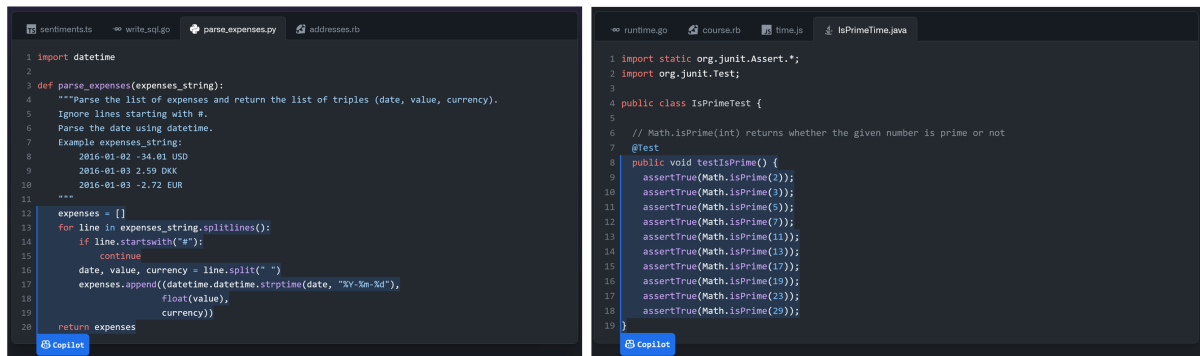
Andrew D. Gordon
Microsoft Research
University of Edinburgh
adg@microsoft.com

Carina Negreanu
Microsoft Research
cnegreanu@microsoft.com

Christian Poelitz
Microsoft Research
cpoelitz@microsoft.com

Sruti Srinivasa Ragavan
Microsoft Research
a-sruti@microsoft.com

Ben Zorn
Microsoft Research
ben.zorn@microsoft.com



```
1 import datetime
2
3 def parse_expenses(expenses_string):
4     """Parse the list of expenses and return the list of triples (date, value, currency).
5     Ignore lines starting with #.
6     Parse the date using datetime.
7     Example expenses_string:
8     2016-01-02 -34.61 USD
9     2016-01-03 2.59 DKK
10    2016-01-03 -2.72 EUR
11    """
12    expenses = []
13    for line in expenses_string.splitlines():
14        if line.startswith("#"):
15            continue
16        date, value, currency = line.split(" ")
17        expenses.append((datetime.datetime.strptime(date, "%Y-%m-%d"),
18                        float(value),
19                        currency))
20    return expenses
```

```
1 import static org.junit.Assert.*;
2 import org.junit.Test;
3
4 public class IsPrimeTest {
5
6     // Math.isPrime(int) returns whether the given number is prime or not
7     @Test
8     public void testIsPrime() {
9         assertTrue(Math.isPrime(2));
10        assertTrue(Math.isPrime(3));
11        assertTrue(Math.isPrime(5));
12        assertTrue(Math.isPrime(7));
13        assertTrue(Math.isPrime(11));
14        assertTrue(Math.isPrime(13));
15        assertTrue(Math.isPrime(17));
16        assertTrue(Math.isPrime(19));
17        assertTrue(Math.isPrime(23));
18        assertTrue(Math.isPrime(29));
19    }
20 }
```

Figure 1 – Code generation using the GitHub Copilot editor extension. The portion highlighted in blue has been generated by the model. Left: a function body, generated based on a textual description in a comment. Right: a set of generated test cases. Source: copilot.github.com

Abstract

Large language models, such as OpenAI’s codex and Deepmind’s AlphaCode, can generate code to solve a variety of problems expressed in natural language. This technology has already been commercialised in at least one widely-used programming editor extension: GitHub Copilot.

In this paper, we explore how programming with large language models (LLM-assisted programming) is similar to, and differs from, prior conceptualisations of programmer assistance. We draw upon publicly available experience reports of LLM-assisted programming, as well as prior usability and design studies. We find that while LLM-assisted programming shares some properties of compilation, pair programming, and programming via search and reuse, there are fundamental differences both in the technical possibilities as well as the practical experience. Thus, LLM-assisted programming ought to be viewed as a new way of programming with its own distinct properties and challenges.

Finally, we draw upon observations from a user study in which non-expert end user programmers use LLM-assisted tools for solving data tasks in spreadsheets. We discuss the issues that might arise, and open research challenges, in applying large language models to end-user programming, particularly with users who have little or no programming expertise.

1. Introduction

Inferential assistance for programmers has manifested in various forms, such as programming by demonstration, declarative programming languages, and program synthesis (Section 2). Large language models such as GPT mark a quantitative and qualitative step-change in the automatic generation of code and natural language text. This can be attributed to cumulative innovations of vector-space word embeddings, the transformer architecture, large text corpora, and pre-trained language models (Section 3).

These models have been commercialised in the form of APIs such as OpenAI Codex, or as programmer-facing tools such as GitHub Copilot and Tabnine. These tools function as a sort of advanced autocom-

plete, able to synthesize multiple lines of code based on a prompt within the code editor, which may be natural language (e.g., a comment), code (e.g., a function signature) or an ad-hoc mixture. The capabilities of such tools go well beyond traditional syntax-directed autocomplete, and include the ability to synthesize entire function bodies, write test cases, and complete repetitive patterns (Section 4). These tools have reliability, safety, and security implications (Section 5).

Prior lab-based and telemetric research on the usability of such tools finds that developers generally appreciate the capabilities of these tools and find them to be a positive asset to the development experience, despite no strong effects on task completion times or correctness. Core usability issues include the challenge of correctly framing prompts as well as the effort required to check and debug generated code (Section 6).

Longitudinal experience reports of developers support some of the lab-based findings, while contradicting others. The challenges of correctly framing prompts and the efforts of debugging also appear here. However, there are many reports that these tools do in fact strongly reduce task time (i.e., speed up the development process) (Section 7).

Programming with large language models invites comparison to related ways of programming, such as search, compilation, and pair programming. While there are indeed similarities with each of these, the empirical reports of the experience of such tools also show crucial differences. Search, compilation, and pair programming are thus found to be inadequate metaphors for the nature of LLM-assisted programming; it is a distinct way of programming with its own unique blend of properties (Section 8).

While LLM-assisted programming is currently geared towards expert programmers, arguably the greatest beneficiaries of their abilities will be non-expert end-user programmers. Nonetheless, there are issues with their direct application in end-user programming scenarios. Through a study of LLM-assisted end-user programming in spreadsheets, we uncover issues in intent specification, code correctness, comprehension, LLM tuning, and end-user behaviour, and motivate the need for further study in this area (Section 9).

2. Prior conceptualisations of intelligent assistance for programmers

What counts as ‘intelligent assistance’ can be the subject of some debate. Do we select only features that are driven by technologies that the artificial intelligence research community (itself undefined) would recognise as artificial intelligence? Do we include those that use expert-coded heuristics? Systems that make inferences a human might disagree with, or those with the potential for error? Mixed-initiative systems (Horvitz, 1999)? Or those that make the user feel intelligent, assisted, or empowered? While this debate is beyond the scope of this paper, we feel that to properly contextualise the qualitative difference made by large language models, a broad and inclusive approach to the term ‘intelligence’ is required.

End-user programming has long been home to inferential, or intelligent assistance. The strategy of direct manipulation (Shneiderman & Norwood, 1993) is highly successful for certain types of limited, albeit useful, computational tasks, where the interface being used (“what you see”, e.g., a text editor or an image editor) to develop an information artefact can represent closely the artefact being developed (“what you get”, e.g., a text document or an image). However, this strategy cannot be straightforwardly applied to programs. Programs notate multiple possible paths of execution simultaneously, and they define “behaviour to occur at some future time” (Blackwell, 2002b). Rendering multiple futures in the present is a core problem of live programming research (Tanimoto, 2013), which aims to externalise programs as they are edited (Basman et al., 2016).

The need to bridge the abstraction gap between direct manipulation and multiple paths of execution led to the invention of programming by demonstration (PBD) (Kurlander et al., 1993; Lieberman, 2001; Myers, 1992). A form of inferential assistance, PBD allows end-user programmers to make concrete demonstrations of desired behaviour that are generalised into executable programs. Despite their promise, PBD systems have not achieved widespread success as end-user programming tools, although their idea survives in vestigial form as various “macro recording” tools, and the approach is seeing a resurgence with

the growing commercialisation of “robotic process automation”.

Programming language design has long been concerned with shifting the burden of intelligence between programmer, program, compiler, and user. Programming language compilers, in translating between high-level languages and machine code, are a kind of intelligent assistance for programmers. The declarative language Prolog aspired to bring a kind of intelligence, where the programmer would only be responsible for specifying (“declaring”) *what* to compute, but not *how* to compute it; that responsibility was left to the interpreter. At the same time, the language was designed with intelligent applications in mind. Indeed, it found widespread use within artificial intelligence and computational linguistics research (Colmerauer & Roussel, 1996; Rouchy, 2006).

Formal verification tools use a specification language, such as Hoare triples (Hoare, 1969), and writing such specifications can be considered programming at a ‘higher’ level of abstraction. Program synthesis, in particular synthesis through refinement, aims at intelligently transforming these rules into executable and correct code. However, the term “program synthesis” is also used more broadly, and programs can be synthesised from other sources than higher-level specifications. Concretely, program synthesis by example, or simply programming by example (PBE), facilitates the generation of executable code from input-output examples. An example of successfully commercialised PBE is Excel’s Flash Fill (Gulwani, 2011), which synthesises string transformations in spreadsheets from a small number of examples.

The Cognitive Dimensions framework (T. R. Green, 1989; T. Green & Blackwell, 1998) identifies three categories of programming activity: authoring, transcription, and modification. Modern programmer assistance encompasses each of these. For example, program synthesis tools transform the direct authoring of code into the (arguably easier) authoring of examples. Intelligent code completions (Marasoiu et al., 2015) support the direct authoring of code. Intelligent support for reuse, such as smart code copy/paste (Allamanis & Brockschmidt, 2017) support transcription, and refactoring tools (Hermans et al., 2015) support modification. Researchers have investigated inferential support for navigating source code (Hendley & Fleming, 2014), debugging (J. Williams et al., 2020), and selectively undoing code changes (Yoon & Myers, 2015). Additionally, intelligent tools can also support learning (Cao et al., 2015).

Allamanis et al. (2018) review work at the intersection of machine learning, programming languages, and software engineering. They seek to adapt methods first developed for natural language, such as language models, to source code. The emergence of large bodies of open source code, sometimes called “big code”, enabled this research area. Language models are sensitive to lexical features like names, code formatting, and order of methods, while traditional tools like compilers or code verifiers are not. Through the “naturalness hypothesis”, which claims that “software is a form of human communication; software corpora have similar statistical properties to natural language corpora; the authors claim that these properties can be exploited to build better software engineering tools.” Some support for this hypothesis comes from research that used n -gram models to build a code completion engine for Java that outperformed Eclipse’s completion feature (Hindle et al., 2012, 2016). This approach can underpin recommender systems (such as code autocompletion), debuggers, code analysers (such as type checkers (Raychev et al., 2015)), and code synthesizers. We can expect the recent expansion in capability of language models, discussed next, to magnify the effectiveness of these applications.

3. A brief overview of large language models for code generation

3.1. The transformer architecture and big datasets enable large pre-trained models

In the 2010s, natural language processing has evolved in the development of language models (LMs,) tasks, and evaluation. Mikolov et al. (2013) introduced Word2Vec, where vectors are assigned to words such that similar words are grouped together. It relies on co-occurrences in text (like Wikipedia articles), though simple instantiations ignore the fact that words can have multiple meanings depending on context. Long short-term memory (LSTM) neural networks (Hochreiter & Schmidhuber, 1997; Sutskever et al., 2014) and later encoder-decoder networks, account for order in an input sequence. Self-attention (Vaswani et al., 2017) significantly simplified prior networks by replacing each element in the input by a weighted average of the rest of the input. Transformers combined the advantages of (multi-head) at-

tention and word embeddings, enriched with positional encodings (which add order information to the word embeddings) into one architecture. While there are many alternatives to transformers for language modelling, in this paper when we mention a language model we will usually imply a transformer-based language model.

There are large collections of unlabelled text for some widely-spoken natural languages. For example, the Common Crawl project¹ produces around 20 TB of text data (from web pages) monthly. Labelled task-specific data is less prevalent. This makes unsupervised training appealing. Pre-trained LMs (J. Li et al., 2021) are commonly trained to perform next-word prediction (e.g., GPT (Brown et al., 2020)) or filling a gap in a sequence (e.g., BERT (Devlin et al., 2019)).

Ideally, the “general knowledge” learnt by pre-trained LMs can then be transferred to downstream language tasks (where we have less labelled data) such as question answering, fiction generation, text summarisation, etc. Fine-tuning is the process of adapting a given pre-trained LM to different downstream tasks by introducing additional parameters and training them using task-specific objective functions. In certain cases the pre-training objective also gets adjusted to better suit the downstream task. Instead of (or on top of) fine-tuning, the downstream task can be reformulated to be similar to the original LLM training. In practice, this means expressing the task as a set of instructions to the LLM via a prompt. So the goal, rather than defining a learning objective for a given task, is to find a way to query the LLM to directly predict for the downstream task. This is sometimes referred to as Pre-train, Prompt, Predict.²

3.2. Language models tuned for source code generation

The downstream task of interest to us in this paper is *code generation*, where the input to the model is a mixture of natural language comments and code snippets, and the output is new code. Unlike other downstream tasks, a large corpus of data is available from public code repositories such as GitHub. Code generation can be divided into many sub-tasks, such as variable type generation, e.g. (J. Wei et al., 2020), comment generation, e.g. (Liu et al., 2021), duplicate detection, e.g (Mou et al., 2016), code migration from one language to another e.g. (Nguyen et al., 2015) etc. A recent benchmark that covers many tasks is CodeXGLUE (Lu et al., 2021).

LLM technology has brought us within reach of full-solution generation. Codex (Chen, Tworek, Jun, Yuan, Ponde, et al., 2021), a version of GPT-3 fine-tuned for code generation, can solve on average 47/164 problems in the HumanEval code generation benchmark, in one attempt. HumanEval is a set of 164 hand-written programming problems, which include a function signature, docstring, body, and several unit tests, with an average of 7.7 tests per problem. Smaller models have followed Codex, like GPT-J³ (fine-tuned on top of GPT-2), CodeParrot⁴ (also fine-tuned on top of GPT-2, targets Python generations), PolyCoder (Xu, Alon, et al., 2022)(GPT-2 style but trained directly on code).

LLMs comparable in size to Codex include AlphaCode (Y. Li et al., 2022a) and PaLM-Coder (Chowdhery et al., 2022). AlphaCode is trained directly on GitHub data and fine-tuned on coding competition problems. It introduces a method to reduce from a large number of potential solutions (up to millions) to a handful of candidates (competitions permit a maximum of 10). On a dataset of 10000 programming problems, Codex solves around 3% of the problems within 5 attempts, versus AlphaCode which solves 4-7%. In competitions for which it was fine-tuned (CodeContests) AlphaCode achieves a 34% success rate, on par with the average human competitor.

Despite promising results there are known shortcomings. Models can directly copy full solutions or key parts of the solutions from the training data, rather than generating new code. Though developers make efforts to clean and retain only high-quality code, there are no guarantees of correctness and errors can be directly propagated through generations.

¹<https://commoncrawl.org/>

²<http://pretrain.nlpedia.ai/>

³https://huggingface.co/docs/transformers/main/model_doc/gpt_j

⁴<https://huggingface.co/blog/codeparrot>

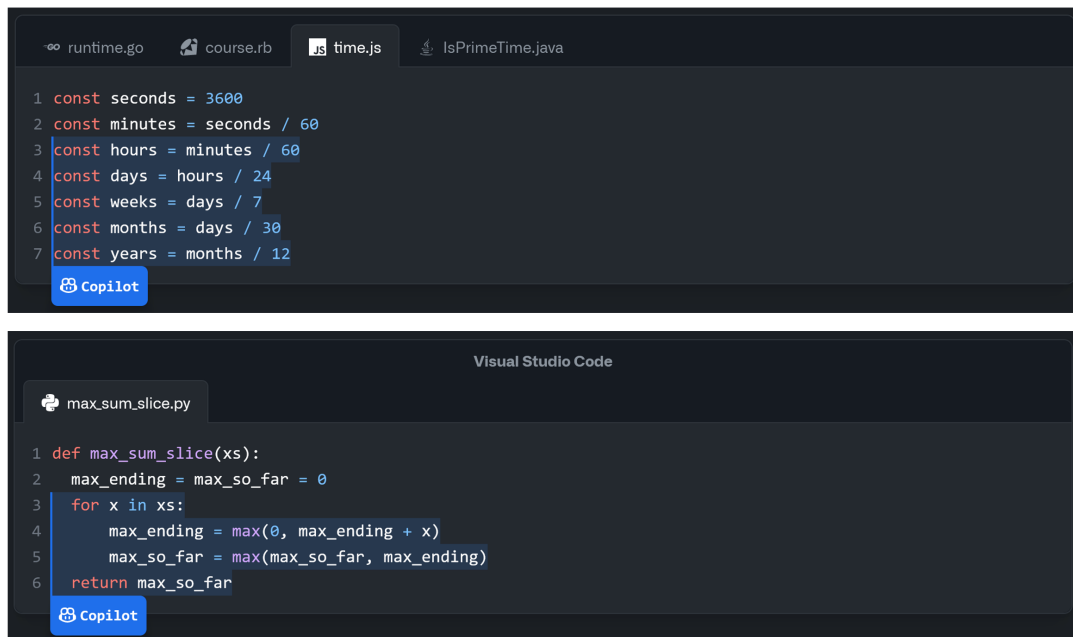


Figure 2 – Code generation with GitHub Copilot. The portion highlighted in blue has been generated by the model. Above: a pattern, extrapolated based on two examples. Below: a function body, generated from the signature and the first line. Source: copilot.github.com

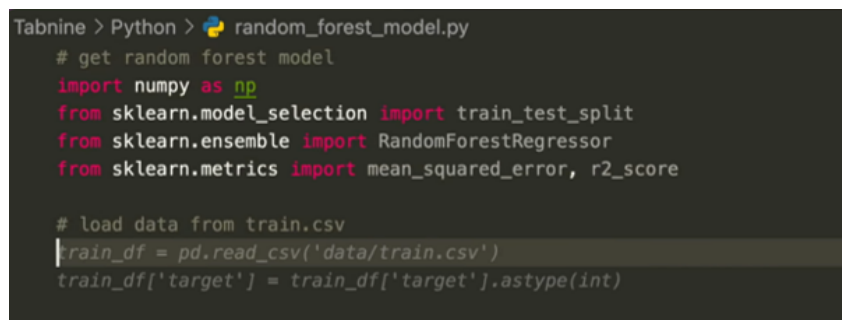


Figure 3 – Code generation using the Tabnine editor extension. The grey text after the cursor is being suggested by the model based on the comment on the preceding line. Source: tabnine.com

Codex can also produce syntactically incorrect or undefined code, and can invoke functions, variables, and attributes that are undefined or out of scope. Moreover, Codex struggles to parse through increasingly long and higher-level or system-level specifications which can lead to mistakes in binding operations to variables, especially when the number of operations and variables in the docstring is large. Various approaches have been explored to filter out bad generations or repair them, especially for syntax errors.

Consistency is another issue. There is a trade-off between non-determinism and generation diversity. Some parameter settings can control the diversity of generation (i.e., how diverse the different generations for a single prompt might be), but there is no guarantee that we will get the same generation if we run the system at different times under the same settings. To alleviate this issue in measurements, metrics such as `pass@k` (have a solution that passes the tests within k tries) have been modified to be probabilistic.

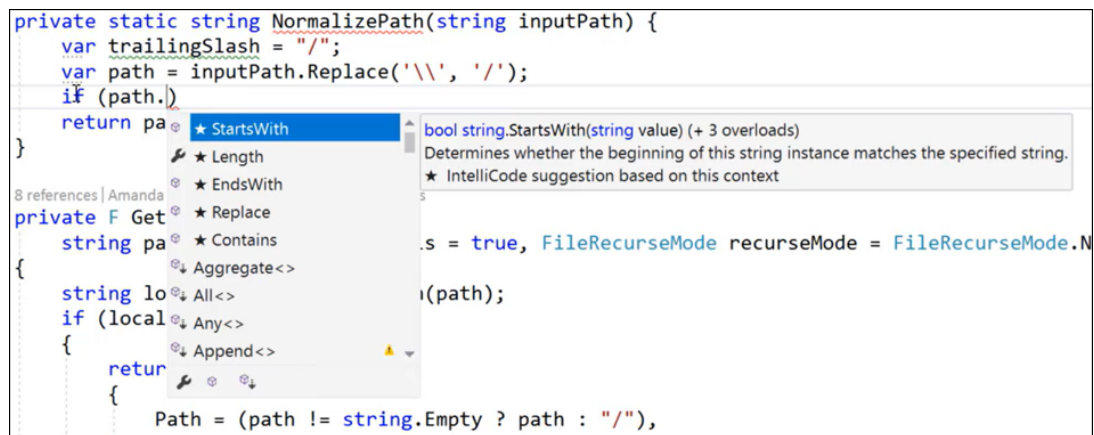


Figure 4 – API suggestion using the Visual Studio IntelliCode feature. Source: Silver (2018)

4. Commercial programming tools that use large language models

OpenAI Codex is a version of GPT that is fine-tuned on publicly available source code (Chen, Tworek, Jun, Yuan, de Oliveira Pinto, et al., 2021). While Codex itself is not a programmer-facing tool, OpenAI has commercialised it in the form of an API that can be built upon.

The principal commercial implementation of Codex thus far has been in Github Copilot.⁵ Copilot is an extension that can be installed into code editors such as Neovim, JetBrains, and Visual Studio Code. Copilot uses Codex, drawing upon the contents of the file being edited, related files in the project, and file paths or URLs of repositories. When triggered, it generates code at the cursor location, in much the same way as autocomplete.

To help expand developer expectations for the capabilities of Copilot beyond the previous standard uses of autocomplete, suggested usage idioms for Copilot include: writing a comment explaining what a function does, and the function signature, and allowing Copilot to complete the function body; completing boilerplate code; and defining test cases (Figures 1 and 5). Programmers can cycle between different generations from the model, and once a particular completion has been accepted it can be edited like any other code.

As of 23 June 2022, Amazon has announced a Copilot-like feature called CodeWhisperer,⁶ which also applies a large language model trained on a corpus of source code to generate autocompletions based on comments and code. The marketing material describes a set of safety features, such as: detecting when generated code is similar to code in the training set, detecting known security vulnerabilities in the generated code, and “removing code that may be considered biased and unfair” (although this latter claim induces skepticism). At present CodeWhisperer is not widely available and thus little is known of its use in practice.

Other commercial implementations of AI-assisted autocomplete features include Visual Studio Intellilcode (Silver, 2018) (Figure 4) and Tabnine (Figure 3)⁷. These are more limited in scope than Copilot and their user experience is commensurable to that of using ‘traditional’ autocomplete, i.e., autocomplete that is driven by static analysis, syntax, and heuristics.⁸ The structure of the machine learning model used by these implementations is not publicly disclosed; however, both rely on models that have been trained on large corpora of publicly available source code.

It is interesting to note, that despite the wide variety of types of intelligent programmer assistance we

⁵<https://copilot.github.com/>

⁶<https://aws.amazon.com/codewhisperer/features/>

⁷<https://www.tabnine.com/>

⁸As of 15 June 2022, Tabnine has announced a shift to language model-driven autocomplete that more closely resembles the abilities of Copilot (Weiss, 2022).

have discussed in Section 2 for several aspects of programming (authoring, transcription, modification, debugging, and learning), commercial implementations of assistance based on large language models thus far are aimed primarily at authoring. Authoring can be viewed as the first natural application of a generative language model, but the programming knowledge in these models can of course be used for assisting programmers in other activities, too.

5. Reliability, safety, and security implications of code-generating AI models

AI models that generate code present significant challenges to issues related to reliability, safety, and security. Since the output of the model can be a complex software artifact, determining if the output is “correct” needs a much more nuanced evaluation than simple classification tasks. Humans have trouble evaluating the quality of software, and practices such as code review, applying static and dynamic analysis techniques, etc., have proven necessary to ensure good quality of human-written code. Current methods for evaluating the quality of AI-generated code, as embodied in benchmarks such as HumanEval (Chen, Tworek, Jun, Yuan, de Oliveira Pinto, et al., 2021), MBPP (Austin et al., 2021), and CodeContests (Y. Li et al., 2022b), determine functional correctness of entire functions based on a set of unit tests. Such evaluation approaches fail to consider issues of code readability, completeness, or the presence of potential errors that software developers constantly struggle to overcome.

Previous work (Chen, Tworek, Jun, Yuan, de Oliveira Pinto, et al., 2021) explores numerous implications of AI models that generate code, including issues of over-reliance, misalignment (the mismatch between what the user prompt requests and what the user really wants), bias, economic impact, and security implications. While these topics each are extensive and important, due to space limitations we only briefly mention them here and point to additional related work when possible. Over-reliance occurs when individuals make optimistic assumptions about the correctness of the output of an AI model, leading to harm. For code generating models, users may assume the code is correct, has no security vulnerabilities, etc. and those assumptions may lead to lower quality or insecure code being written and deployed. Existing deployments of AI models for code, such as GitHub Copilot (Ziegler, 2021), have documentation that stresses the need to carefully review, test, and vet generated code just as a developer would vet code from any external source. It remains to be seen if over-reliance issues related to AI code generation will result in new software quality challenges.

Since AI that generates code is trained on large public repositories, there is potential for low-quality training data to influence models to suggest low-quality code or code that contains security vulnerabilities. One early study of GitHub Copilot (Pearce et al., 2021) examines whether code suggestions may contain known security vulnerabilities in a range of scenarios and finds cases where insecure code is generated. Beyond carefully screening new code using existing static and dynamic tools that detect security vulnerabilities in human-generated code, there are also possible mitigations that can reduce the likelihood that the model will make such suggestions. These include improving the overall quality of the training data by removing low-quality repositories, and fine-tuning the large-language model specifically to reduce the output of known insecure patterns.

6. Usability and design studies of AI-assisted programming

Vaithilingam et al. (2022) conducted a within-subjects comparative study (n=24) of Github Copilot, comparing its user experience to that of traditional autocomplete (specifically, the *Intellisense* plugin, not the same as the *Intellicode* feature mentioned previously). Participants failed to complete the tasks more often with Copilot than with Intellisense, and there was no significant effect on task completion time. Perhaps unsurprisingly, the authors find that assessing the correctness of generated code is difficult and an efficiency bottleneck, particularly when the code generated has a fundamental flaw or inefficiency that leads the programmer on an ultimately unsuccessful ‘wild goose chase’ of repair or debugging. However, the overwhelming majority (19 of 24) of participants reported a strong preference for Copilot in a post-task survey. While participants were less confident about the code generated by Copilot, they almost universally (23 of 24) perceived it as more helpful, because it had the potential for generating useful starting points and saving the programmer the effort of searching online for documented solutions

that could be the basis for reuse.

Ziegler et al. (2022) conducted a survey ($n=2,047$) of the perceived productivity of Copilot users in the USA. They matched these to telemetric usage measurements of the Copilot add-in, which included metrics such as how often an auto-completion was shown, how often it was accepted, how often it persisted unchanged in the document for a certain time period, how often it persisted with minor variations (e.g., measured by Levenshtein distance) and so on. They find that the acceptance rate (the ratio of accepted suggestions to shown suggestions) is the strongest predictor of users' perceived productivity due to Copilot. Fascinatingly, they find that the pattern of acceptance rates for all users in aggregate follows a daily and weekly "circadian" rhythm, such that users are more likely to accept Copilot completions out of working-hours and on weekends. However, for any given user, the acceptance rate depends on that user's normal working hours; suggestions outside of normal working hours are less likely to be accepted. Future work is needed to see whether this finding replicates, and if so to establish how and why acceptance rates are so significantly affected by working hours.

Xu, Vasilescu, & Neubig (2022) conducted a within-subjects study ($n=31$) comparing the programming experience with and without a code generation plugin. Their experimental plugin takes the form of a text field in which the user enters a natural language prompt, the system responds with a list of code snippets, and when clicked the desired snippet is inserted at the cursor. This workflow differs from Copilot's, where the 'prompt' is text within the source file, and can contain a mix of natural language comments and code. The plugin supported both code generation (using a tree-based neural network) and code snippet retrieval (searching the programming forum Stack Overflow). Results from both generation and retrieval are shown in the same list, but visually demarcated. The authors found no significant effect of the plugin on task completion time or program correctness. They found that simple queries were more likely to be answered correctly through generation, and more complex queries requiring multiple steps were more likely to be answered correctly through retrieval, and that it was possible to predict which approach would succeed based on the word content of the queries. Further, they found that most (60%) natural language queries that participants wrote in their experiment were not sufficiently well-specified for a human expert to write code implementing those intents. Retrieved snippets were edited more often than generated snippets, mostly to rename identifiers and choose different parameters. In a post-experiment survey, participants reported mostly feeling neutral or somewhat positive (30 of 31). These participants felt that the plugin was helpful for finding snippets they were aware of but cannot recall, and less disruptive than using a browser, but the interaction worked better when the developer had a pre-existing knowledge of the target APIs and frameworks, and it took experimentation to understand the "correct way" to formulate queries. There was no clear indication of preference between retrieval and generation.

Jiang et al. (2022) developed an LLM-based tool for converting natural language statements to code. As in Xu, Vasilescu, & Neubig (2022), prompts are entered in a pop-up dialog invoked at the cursor from within a code editor, rather than as comments. In a study ($n = 14$), participants were given a week to complete two website-building tasks with the tool, while recording the screen, and were interviewed afterwards. As in other studies, participants saw utility in the tool for facilitating quick API lookups and for writing boilerplate code. They found that novice programmers' queries were mainly natural language, whereas experts were more likely to mix code into their requests. While some queries were abstract, and expressed high-level goals, most had low granularity, being "roughly equivalent to a line of code". To cope with model failures, participants used a variety of strategies to reword their query, such as reducing the scope of the request or replacing words with alternatives, but no particular strategy was observed to be more effective than any other. Participants struggled with forming a mental model of what the model can understand and the "syntax" of the language it required – this is precisely the *fuzzy abstraction matching* problem we described earlier, which the authors call an "uncanny valley". The authors suggest possible solutions such as automated rewording of prompts, suggesting simpler tasks, suggesting task breakdowns, and better onboarding and tutorials.

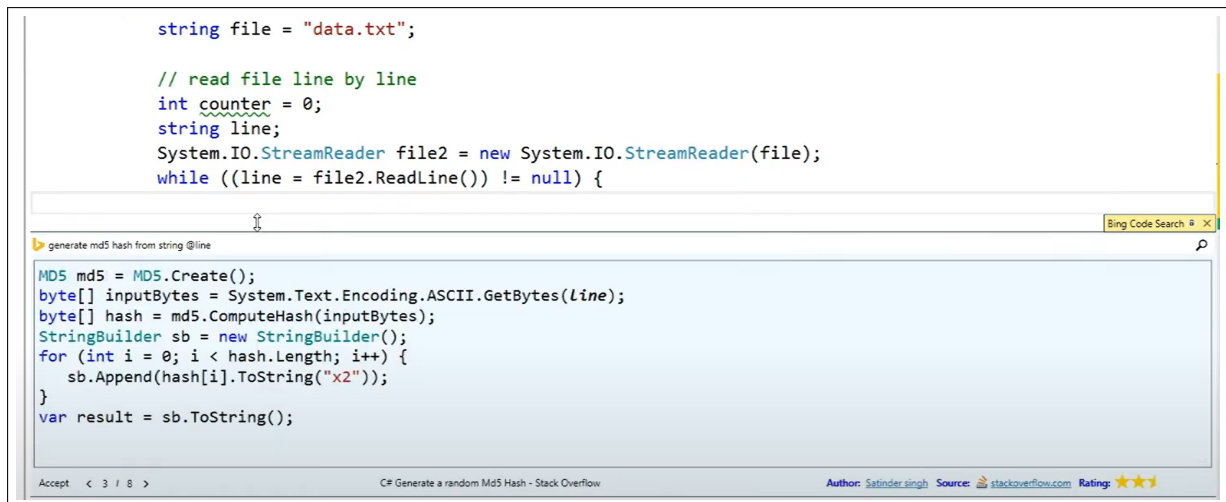


Figure 5 – Searching for code snippets using Bing Developer Assistant. A result for Stack Overflow is shown. Note how the query “generate md5 hash from string @line” contains a hint about the identifier line, which is used to rewrite the retrieved snippet. Source: <https://www.microsoft.com/en-us/research/publication/building-bing-developer-assistant/>

Barke et al. (2022) studied how programmers ($n = 20$) use GitHub Copilot to complete short programming tasks in Python, Rust, Haskell, and Java. Through analysis of screen recordings, the authors identified two primary modes of interaction with Copilot: *acceleration*, where the programmer has a well-formed intent and Copilot speeds up code authoring in “small logical units”, and *exploration*, where Copilot suggestions are used to assist the planning process, “help them get started, suggest potentially useful structure and API calls, or explore alternative solutions”. In acceleration, long code suggestions, which take time to read and evaluate, can break the programmer’s flow. Participants developed heuristics for quickly scanning suggestions, such as looking for the presence of certain keywords. In exploration, participants were more likely to prompt using purely natural language comments, rather than a mix of comments and code. Moreover, these prompt comments were often ‘cleaned’ subsequent to accepting a suggestion, which implies a form of ‘instruction language’ that is separate from ‘explanation language’.

Madi (2022) compared the readability of code generated by Copilot with that of code written by human programmers in a user study ($n = 21$). They found that model generated code is comparable in complexity and readability to human-authored code.

The Bing Developer Assistant (Y. Wei et al., 2015; Zhang et al., 2016) (also referred to as Bing Code Search) was an experimental extension for Visual Studio initially released in 2015. It enabled an in-IDE, identifier-aware search for code snippets from forums such as Stack Overflow. It had the ability to rewrite retrieved code to use identifiers from the programmer’s current file. A user study ($n=14$) comparing task time in performing 45 short programming tasks with the extension versus regular web search found on average 28% of time was saved with the extension. Moreover telemetry data gathered over three weeks (representing around 20,000 users and around 3,000 queries per day) showed that several programmers used the feature frequently. Some used it repeatedly for related problems in quick succession, showing its use in multi-step problems. Others issued the same query multiple times on separate days, suggesting that the speed of auto-completion was useful even if the programmer knew the solution.

7. Experience reports

At present, there is not a lot of research on the user experience of programming with large language models beyond the studies we have summarised in Section 6. However, as the availability of such tools increases, professional programmers will gain long-term experience in their use. Many such program-

mers write about their experiences on personal blogs, which are then discussed in online communities such as Hacker News. Inspired by the potential for these sources to provide rich qualitative data, as pointed out by Barik (Barik et al., 2015; Sarkar et al., 2022), we draw upon a few such experience reports. A full list of sources is provided Appendix A; below we summarise their key points.

7.1. Writing effective prompts is hard

As with several other applications of generative models, a key issue is the writing of prompts that increase the likelihood of successful code generation. The mapping that these models learn between natural language and code is very poorly understood. Through experimentation, some have developed heuristics for prompts that improve the quality of the code generated by the model. One developer, after building several applications and games with OpenAI’s `code-davinci` model (the second generation Codex model), advises to “*number your instructions*” and creating “*logic first*” before UI elements. Another, in using Copilot to build a classifier for natural language statements, suggests to provide “*more detail*” in response to a failure to generate correct code. For example, when asking Copilot to “*binarize*” an array fails, they re-write the prompt to “*turn it into an array where [the first value] is 1 and [the second value] is 0*” – effectively pseudocode – which generates a correct result.

Commenters on Hacker News are divided on the merits of efforts invested in developing techniques for prompting. While some see it as a new level of abstraction for programming, others see it as indirectly approaching more fundamental issues that ought to be solved with better tooling, documentation, and language design:

“You’re not coding directly in the language, but now you’re coding in an implicit language provided by Copilot. [...] all it really points out is that code documentation and discovery is terrible. But I’m not for sure writing implicit code in comments is really a better approach than seeking ways to make discovery of language and library features more discoverable.”

“[...] the comments used to generate the code via GitHub Copilot are just another very inefficient programming language.”

“[Responding to above] There is nonetheless something extremely valuable about being able to write at different levels of abstraction when developing code. Copilot lets you do that in a way that is way beyond what a normal programming language would let you do, which of course has its own, very rigid, abstractions. For some parts of the code you’ll want to dive in and write every single line in painstaking detail. For others [...] [Copilot] is maybe enough for your purposes. And being able to have that ability, even if you think of it as just another programming language in itself, is huge.”

Being indiscriminately trained on a corpus containing code of varying ages and (subjective) quality has drawbacks; developers encounter generated code which is technically correct, but contains practices considered poor such as unrolled loops and hardcoded constants. One Copilot user found that:

“Copilot [...] has made my code more verbose. Lines of code can be liabilities. Longer files to parse, and more instances to refactor. Before, where I might have tried to consolidate an API surface, I find myself maintaining [multiple instances].”

Another Copilot user reflected on their experience of trying to generate code that uses the `fastai` API, which frequently changes:

“[...] since the latest version of fastai was only released in August 2020, GitHub Copilot was not able to provide any relevant suggestions and instead provided code for using older versions of fastai. [...] To me, this is a major concern [...] If we are using cutting edge tools [...] Copilot has no knowledge of this and cannot provide useful suggestions.”

On the other hand, developers can also be exposed to *better* practices and APIs through these models. The developer that found Copilot to make their code more verbose also observed that:

“Copilot gives structure to Go errors . [...] A common idiom is to wrap your errors with a context string [which can be written in an inconsistent, ad-hoc style] [...] Since using Copilot, I haven’t written a single one of these error handling lines manually. On top of that, the suggestions follow a reasonable structure where I didn’t know structure had existed before. Copilot showed me how

to add structure in my code in unlikely places. For writing SQL, it helped me write those annoying foreign key names in a consistent format [...]

[Additionally,] One of the more surprising features has been [that] [...] I find myself discovering new API methods, either higher-level ones or ones that are better for my use case."

In order to discover new APIs, of course, the APIs themselves need to be well-designed. Indeed, in some cases the spectacular utility of large language models can be largely attributed to the fact that API designers have already done the hard work of creating an abstraction that is a good fit for real use cases (Myers & Stylos, 2016; Piccioni et al., 2013; Macvean et al., 2016). As a developer who used Copilot to develop a sentiment classifier for Twitter posts matching certain keywords remarks, *"These kinds of things are possible not just because of co pilot [sic] but also because we have awesome libraries which have abstracted a lot of tough stuff."* This suggests that API design, not just for human developers but also as a target for large language models, will be important in the near and mid-term future.

Moreover, breaking down a prompt at the 'correct' level of detail is also emerging as an important developer skill. This requires at least some familiarity, or a good intuition, for the APIs available. Breaking down prompts into steps so detailed that the programmer is effectively writing pseudocode, can be viewed as an anti-pattern, and can give rise to the objections cited earlier that programming via large language models is simply a *"very inefficient programming language"*. We term this the problem of *fuzzy abstraction matching*. The problem of figuring out what the system can and can't do, and matching one's intent and instructions with the capabilities of the system, is not new – it has been well-documented in natural language interaction (Mu & Sarkar, 2019; Luger & Sellen, 2016). It is also observed in programming notation design as the 'match-mismatch' hypothesis (T. R. Green & Petre, 1992; Chalhoub & Sarkar, 2022). In the broadest sense, these can be seen as special cases of Norman's "gulf of execution" (Hutchins et al., 1985), perhaps the central disciplinary problem of first and second-wave (Bødker, 2015) human-computer interaction research: 'how do I get the computer to do what I want it to do?'.

What distinguishes fuzzy abstraction matching from previous incarnations of this problem is the resilience to, and accommodation of, various levels of abstraction afforded by large language models. In previous natural language interfaces, or programming languages, the user needed to form an extremely specific mental model before they could express their ideas in machine terms. In contrast, large language models can generate plausible and correct results for statements at an extremely wide range of abstraction. In the context of programming assistance, this can range from asking the model to write programs based on vague and underspecified statements, requiring domain knowledge to solve, through to extremely specific and detailed instructions that are effectively pseudocode. This flexibility is ultimately a double-edged sword: it has a lower floor for users to start getting usable results, but a higher ceiling for getting users to maximum productivity.

In the context of programming activities, *exploratory programming*, where the goal is unknown or ill-defined (Kery & Myers, 2017; Sarkar, 2016), does not fit the framing of fuzzy abstraction matching (or indeed any of the variations of the gulf of execution problem). When the very notion of a crystallised user *intent* is questioned, or when the design objective is for the system to influence the intent of the user (as with much designerly and third-wave HCI work), the fundamental interaction questions change. One obvious role the system can play in these scenarios is to help users refine their own concepts (Kulesza et al., 2014) and decide what avenues to explore. Beyond noting that such activities exist, and fall outside the framework we have proposed here, we will not explore them in greater detail in this paper.

7.2. The activity of programming shifts towards checking and unfamiliar debugging

When code can be generated quickly, as observed with the studies in Section 6, checking the correctness of generating code becomes a major bottleneck. This shift, or tradeoff, of faster authoring at the expense of greater time spent checking code, is not without criticism. For some it is the wrong balance of priorities between system and programmer.

Correspondingly, some users have developed heuristics for when the cost of evaluating the correctness

of the code is greater than the time or effort saved by code generation, such as to focus on very short (e.g., single line) completions and ignore longer completions.

Furthermore, some users have found that rather than having suggestions show all the time, which can be distracting and time consuming, more intentional use can be made of Copilot by switching off auto-suggestion and only triggering code completion manually using a keyboard shortcut. However, this requires users to form a mental model of when Copilot is likely to help them in their workflow. This mental model takes time and intentionality to build, and may be incorrect. Moreover, it introduces a new cognitive burden of constantly evaluating whether the current situation would benefit from LLM assistance. Commenters on Hacker News raise these issues:

“I find I spend my time reviewing Copilot suggestions (which are mostly wrong) rather than thinking about code and actually doing the work.”

“[...] It’s much quicker to read code than to write it. In addition, 95% of Copilots suggestions are a single line and they’re almost always right (and also totally optional).[...] I admit that I’m paranoid every time it suggests more than 2 lines so I usually avoid it. [...] I’ve run into Copilot induced headaches twice. Once was in the first week or so of using it. I swore off [sic] of using it for anything more than a line then. Eventually I started to ease up since it was accurate so often and then I learned my second lesson with another mistake. [...]”

“[...] writing code is not the bottleneck in need of optimization. Conceiving the solution is. Any time “saved” through Copilot and it’s ilk is immediately nullified by having to check it’s correctness. [...]”

“What I want is a copilot that finds errors [...] Invert the relationship. I don’t need some boilerplate generator; I need a nitpicker that’s smarter than a linter. I’m the smart thinker with a biological brain that is inattentive at times. Why is the computer trying to code and leaving mistake catching to me? It’s backwards.”

“I turned off auto-suggest and that made a huge difference. Now I’ll use it when I know I’m doing something repetitive that it’ll get easily, or if I’m not 100% sure what I want to do and I’m curious what it suggests. This way I get the help without having it interrupt my thoughts with its suggestions.”

Another frequent experience is that language models can introduce subtle, difficult to detect bugs, which are not the kind that would be introduced by a human programmer writing code manually. Thus, existing developer intuitions around the sources of errors in programs can be less useful, or even misleading, when checking the correctness of generated code.

One developer reported their experience of having an incorrect, but plausible-sounding field name suggested by Copilot (`accessTokenSecret` instead of `accessSecret`) and the consequent wild goose chase of debugging before discovering the problem. As sources of error, these tools are new, and developers need to learn new craft practices for debugging. *“There are zero places that can teach you those things. You must experience them and unlock that kind of knowledge.”*, the developer concludes, *“Don’t let code completion AI tools rule your work. [...] I don’t blame [Copilot] for this. I blame myself. But whatever. At least I got some experience.”*. Commenters on Hacker News report similar experiences:

“[...] The biggest problem I’ve had is not that it doesn’t write correctly, it’s that it think it knows how and then produce good looking code at a glance but with wrong logic. [...]”

“[...] it has proved to be very good at producing superficially appealing output that can stand up not only to a quick scan, but to a moderately deep reading, but still falls apart on a more careful reading. [...] it’s an uncanny valley type effect. [...] it’s almost the most dangerous possible iteration of it, where it’s good enough to fool a human functioning at anything other than the highest level of attentiveness but not good enough to be correct all the time. See also, the dangers of almost self-driving cars; either be self-driving or don’t but don’t expect halfway in between to work well.”

“[...] The code it generates looks right but is usually wrong in really difficult to spot ways but things you’d never write yourself.”

Many developers reported concerns around such tools repeating private information, or repeating copyrighted code verbatim, which might have implications for the licenses in their own projects. Notions of the dangers of such “stochastic parrots” (Bender et al., 2021) are not new and have been well-explored, and are not as directly connected to the user experience of programming assistance as some of the other concerns we have listed here. As such, we will not enter that discussion in depth here, except to mention that these concerns were present in several blog articles and online discussions.

Thus, in practice, programmers describe the challenges of writing effective prompts, misinterpreted intent, code that includes subtle bugs or poor programming practices, the burden of inspecting and checking that generated code is correct, and worries about private information, plagiarism and copyright.

7.3. These tools are useful for boilerplate and code reuse

Despite the challenges we have described so far in this section, the utility of these tools in certain contexts is undeniable, and some programmers report having developed workflows, in certain contexts, that are heavily dependent on AI assistance. Particularly for simple tasks that require a lot of “boilerplate” code, or common tasks for which there are likely to be snippets of code online which prior to these AI assistants would have required a web search to retrieve. Hacker News commenters write:

“These days not having Copilot is a pretty big productivity hit to me. The other day Copilot somehow stopped offering completions for maybe an hour, and I was pretty shocked to realize how much I’ve grown to rely on just hitting tab to complete the whole line. (I was writing Go at the time which is on the boilerplatey side among the mainstream languages, so Copilot is particularly effective [...])”

“I use GTP-3 codex [sic] daily when working. It saves me time, helps me explore unfamiliar languages and APIs and generates approaches to solve problems. It can be shockingly good at coding in narrow contexts. It would be a mistake to miss the developments happening in this area”

“[...] for a lot of quick programming questions, I’m finding I don’t even need a search engine. I just use Github Copilot. For example, if I wanted to remember how to throw an exception I’d just write that as a comment and let Copilot fill in the syntax. Between that and official docs, don’t need a ton else.”

“[...] It’s changing the way I write code in a way that I can already tell is allowing me to be much lazier than I’ve previously been about learning various details of languages and libraries. [...])”

“[...] Github Copilot [...] pretty much replaced almost my entire usage of Stack Overflow.[...])”

“[...] GitHub Copilot really shines in rote work: when it can correctly infer what you are about to do, it can and will assist you correctly. It’s not able to make big decisions, but in a pinch, it might be able to give hints. [...] If used right, Copilot can give developers a significant velocity boost, especially in greenfield projects where there is lots and lots of boilerplate to write. [...])”

8. The inadequacy of existing metaphors for AI-assisted programming

8.1. AI assistance as search

In research studies, as well as in reports of developer experiences, comparisons have been drawn between the nature of AI programming assistance and programming by searching and reusing code from the Internet (or from institutional repositories, or from the same project, or from a developer’s previous projects).

The comparison between AI programming assistance and search is a natural one, and there are many similarities. Superficially, both have a similar starting point: a *prompt* or query that is predominantly natural language (but which may also contain code snippets). From the user perspective, both have an *information asymmetry*: the user does not know precisely what form the result will take. With both search and AI assistance, for any given query, there will be *several results*, and the user will need to invest time evaluating and comparing them. In both cases, the user may only get an *inexact solution*, or indeed nothing like what they want, and the user may need to invest time adapting and repairing what they get.

However, there are differences. When searching the web, programmers encounter not just code, but a variety of types of results intermingled and enmeshed. These include code snippets interspersed with

human commentary, perhaps discussions on forums such as Stack Overflow, videos, and images. A search may return new APIs or libraries related to the query, thus showing results at different levels of abstraction. Search has signals of provenance: it is often (though not always) possible to determine the source of a code snippet on the web. There is a lot of information scent priming to assist with the information foraging task (Srinivasa Ragavan et al., 2016). In this way, programming with search is a *mixed media* experience.

In contrast, programming with large language models can be said to be a *fixed media* experience. The only output is tokens (code, comments, and data) that can be represented within the context of the code editor. This has some advantages: the increased speed of code insertion (which is the immediate aim) often came up in experience reports. However, the learning, exploration, and discovery, and access to a wide variety of sources and media types that occurs in web search is lost. Provenance, too is lost: it is difficult to determine whether the generation is original to the model, or a stochastic parroting (Bender et al., 2021; Ziegler, 2021). Moreover, due to privacy, security, and intellectual property concerns, the provenance of code generated by large language models may be withheld or even destroyed (Sarkar, 2022). This suggests that in future assistance experiences, mixed-media search might be integrated into programmer assistance tools, or the models themselves might be made capable of generating more types of results than the simple code autocomplete paradigm of current tools.

8.2. AI assistance as compilation

An alternative perspective is that AI assistance is more like a compiler. In this view, programming through natural language prompts and queries is a form of higher-level specification, that is ‘compiled’ via the model to the source code in the target language, which is lower level.

Let us (crudely) assume that as programming notations travel along the abstraction continuum from ‘lower’ to ‘higher’ levels, the programmer becomes, firstly, less concerned with the mechanistic details of program execution, and secondly, more and more declarative, specifying *what* computation is required rather than *how* to compute it. In general, these are desirable properties of programming notations, but they do not always make the activity of programming easier or more accessible. As people who write code in declarative languages or formal verification tools will tell you, it’s often much more difficult to specify the *what* than the *how*. The much more broadly adopted practice of test-driven development is adjacent; while tests are not necessarily written in a higher-level language than the code, they aim to capture a higher-level notion of correctness, the *what* of the problem being solved. Learning to be a test engineer takes time and experience, and the entire distinct career path of “software engineer in test” attests to the specialised requirements of programming at higher levels of abstraction.

Some would draw a distinction between programming in a specification language and a compiled programming language. Tony Hoare himself considers these different, on the grounds that while a compiler only aims to map a program from the source language into a finite set of valid programs in the target language, a specification might be satisfied by an infinite number of valid programs (*pers comm.*, first author, ca. 2014). Thus the technical and interaction design problems of programming through specification refinement encompasses, but is much broader than, the technical and interaction design problems of compilers. While we acknowledge this distinction, there is insufficient empirical evidence from the experience reports summarised in Section 7 that working programmers themselves consistently make a meaningful distinction between these concepts.

Programming with large language models, like in a higher-level notation, also allows the programmer to be less concerned with details of the target language. For example, developers in our experience reports relied on AI assistance to fill in the correct syntax, or to discover and correctly use the appropriate API call, thus allowing them to focus on higher-level aspects of the problem being solved. However, there are fundamental differences between this experience and the experience of using a compiler. First, the abstraction is not complete, i.e., a programmer cannot *completely* be unaware of the target language, they must still be able to understand and evaluate the generated code in order to use such tools effectively. With compilers, although knowledge of the target language can help experienced developers in certain

circumstances, it is far from a prerequisite for effective usage. Moreover, compilers can be relied on almost universally to generate a correct and complete translation from source to target language, whereas programming with AI assistance involves the active checking and adaptation of translated code. Next, compilers are (comparatively) deterministic, in that they consistently produce the same output for the same input, but this is not the case for current AI programming tools (although this is not a fundamental limitation, and consistency can be enforced). Finally, though they are often criticised for being cryptic and unhelpful (Barik et al., 2018), compilers do offer levels of interaction and feedback through warnings and error messages, which help the programmer improve the code in the source language; there is currently no such facility with AI programming tools and this strikes us as an area with potential for innovation.

Perhaps more profoundly, while natural language can be used to express concepts at a higher abstraction level, the *range* of abstraction expressible in natural language is much wider than with other forms of programming notation. Traditional programming notations with ad-hoc abstraction capabilities (subroutines, classes, etc.) allow programmers to manually raise the level of abstraction of their own code and APIs. But with code generated by language models, as we have seen from the reports in Section 7, a prompt can span the gamut from describing an entire application in a few sentences, to painstakingly describing an algorithm in step-by-step pseudocode. Thus it would be a mistake to view programming with AI assistance as another rung on the abstraction ladder. Rather, it can be viewed as a device that can teleport the programmer to arbitrary rungs of the ladder as desired.

We close the discussion on AI assistance as a compiler with a few miscellaneous notes. The idea of using natural language as a programming notation has a long history (e.g., (Miller, 1981; Lieberman & Liu, 2006)), which we will not cover here. However, it is notable that there are many ways that natural language has been integrated with programming, such as debugging (Ko & Myers, 2004). With large language models, there are better capabilities for inference of intent and translation to code, but therefore also the potential to open up new strategies for inspecting and explaining code. There are also new failure modes for this paradigm of programming.

8.3. AI assistance as pair programming

The third common perspective is that AI-assisted programming is like pair programming. GitHub Copilot’s commercial tagline describes it as “your AI pair programmer”. As opposed to search and compilation, which are both relatively impersonal tools, the analogy with pair programming is evocative of a more bespoke experience; assistance from a partner that understands more about your specific context and what you’re trying to achieve. AI-assisted programming does have the potential to be more personalised, to the extent that it can take into consideration your specific source code and project files. As Hacker News commenters write:

“[...] at one point it wrote an ENTIRE function by itself and it was correct. [...] it wasn’t some dumb boilerplate initialization either; it was actual logic with some loops. The context awareness with it is off the charts sometimes.[...]”

“[...] It’s like having the stereotypical “intern” as an associate built-in to your editor. [...] It’s also ridiculously flexible. When I start writing graphs in ASCII (cause I’m just quickly writing something down in a scratch file) it’ll actually understand what I’m doing and start autocompleting textual nodes in that ASCII graph.”

Besides personalisation, the analogy also recalls the conventional role-division of pair programming between “driver” and “navigator”. When programming, one needs to form mental models of the program at many layers: from the specific statement being worked on, to its context in a subroutine, to the role that subroutine plays in a module, to the module within the program. However, code must be written at the statement level, which forces developers to keep this lowest level constantly at the forefront of their working memory. Experienced developers spend more time mapping out their code so that they can spend less time writing it. Research into code display and navigation has explored how different ways of presenting lines of code can help programmers better keep these different layers of mental models in mind (Henley & Fleming, 2014). Pair programming, the argument goes, allows two partners to share the

burden of the mental model. The driver codes at the statement and subroutine level while the navigator maps out the approach at the module and program level.

By analogy to pair programming, the AI assistant taking the role of the driver, a solo programmer can now take the place of the navigator. But as we have seen, the experience of programming with AI assistance does not consistently absolve the human programmer of the responsibility for understanding the code at the statement and subroutine level. The programmer may be able to become “*lazier [...] about learning various details of syntax and libraries*”, but the experience still involves much greater statement-level checking.

While a pair programming session requires a conscious, negotiated decision to swap roles, a solo programmer with an AI assistant might find themselves fluidly traversing the spectrum from driving to navigation, from one moment to the next. This may partially explain why, in a preliminary experiment (n=21) comparing the experience of “pair programming” with GitHub Copilot to programming in a human pair either as driver or navigator, Imai (2022) finds that programmers write more lines of code with Copilot than in a human pair, but these lines are of lower quality (more are subsequently deleted).

Moreover, meta-analyses of pair programming have shown mixed efficacy of human pair programming on task time, code quality and correctness (Salge & Berente, 2016; Hannay et al., 2009), suggesting that emulating the pair programming experience is not necessarily a good target to aim for. Multiple studies have concluded that the apparent successes of pair programming can be attributed, not to the role division into driver and navigator, but rather the high degree of *verbalisation* that occurs when pair programmers are forced to rationalise their decisions to each other (Hannay et al., 2009). Others have found that programming in pairs induces greater focus out of a respect for shared time; pair programmers are less likely to read emails, surf the web, or take long phone calls (L. A. Williams & Kessler, 2000). These particular benefits of pair programming are not captured at all by AI assistance tools.

The comparison to pair programming is thus relatively superficial, and today’s experience of AI-assisted programming is not comparable with pair programming to the same extent as it is with search or compilation.

8.4. A distinct way of programming

LLM-assisted programming assistance bears similarities to search: both begin with a prompt, both have an information asymmetry, there are several results, with inexact solutions. But there are differences: search is mixed-media, whereas LLM assistance is fixed. Search (often) has provenance, and language models do not.

It also bears similarities to compilation and programming by specification. Both enable programming at a ‘higher’ level of abstraction (for some definition of higher). Yet unlike with compilers, a programmer using AI assistance must still have a working knowledge of the target language, they must actively check the output for correctness, and they get very little feedback for improving their ‘source’ code.

It also bears a superficial similarity to pair programming, in that it promises to let the programmer take the role of ‘navigator’, forming high-level mental models of the program while delegating the role of ‘driver’ to the language model. But unlike with pair programming, the human navigator must often hop into the driver’s seat. And unlike with pair programming, LLM-assisted programming does not require verbalisation, nor does it coerce greater focus out of a respect for shared time.

Thus existing metaphors do not completely capture the experience of LLM-assisted programming. It is emerging as a distinct way of programming. It does not quite strike us as a distinct *practice* of programming, as that term has been applied to communities of programmers united by similar ethos and aims, such as enterprise software engineers, bricoleurs, live coders, and code benders; but as Bergström & Blackwell (2016) note, there are no clear criteria by which we can define the boundaries of a practice. Nor does it strike us as being a new *activity* of programming as per the cognitive dimensions framework, since AI assistance is clearly orthogonal to authoring, transcription, and modification, being applicable to each of these activities and others besides. Yet as a way of programming it seems to affect

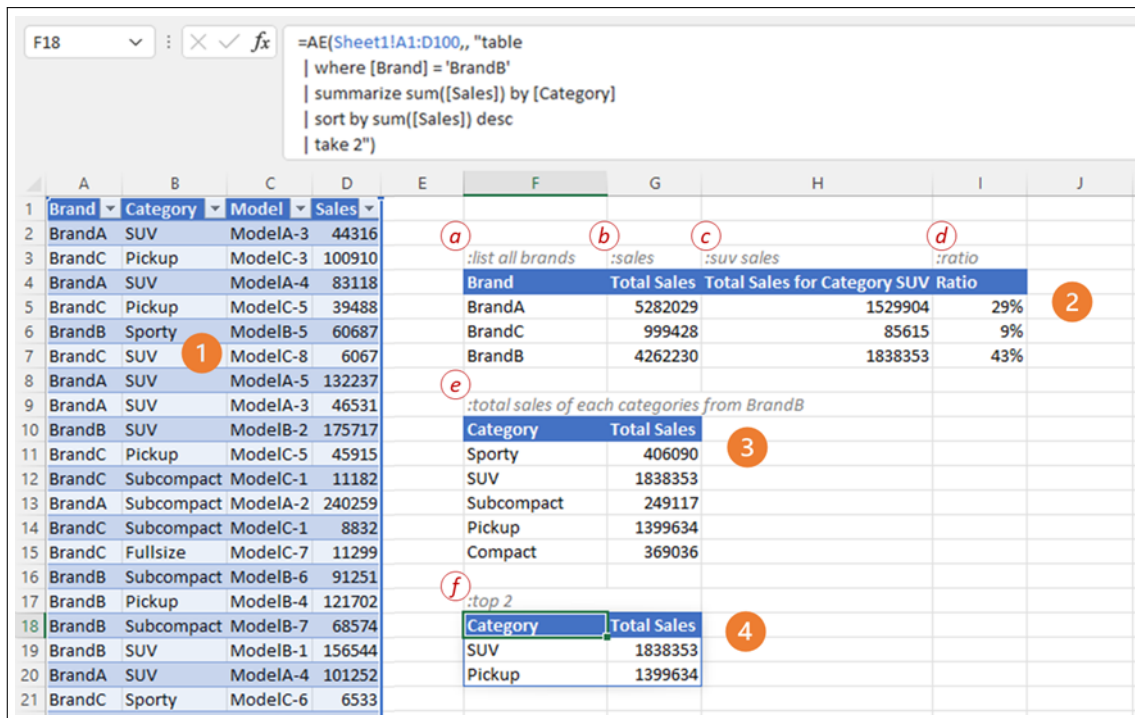


Figure 6 – GridBook interface showing natural language formula in the spreadsheet grid.

programmer's experience more profoundly than a feature such as autocomplete, having far-reaching impact on their attitudes and practices of authoring, information foraging, debugging, refactoring, testing, documentation, code maintenance, learning, and more.

9. Issues with application to end-user programming

The benefits and challenges of programming with LLMs discussed so far concern the professional programmer, or a novice programmer in training. They have formal training in programming and, often, some understanding of the imperfect nature of AI-generated code. But the majority of people who program do not fall into this category. Instead, they are ordinary end users of computers who program to an end. Such end-user programmers often lack knowledge of programming, or the workings of AI. They also lack the inclination to acquire those skills.

It is reasonable to say that such end-user programmers (e.g., accountants, journalists, scientists, business owners) stand to benefit the most from AI assistance, such as LLMs. In one ideal world, an end-user wanting to accomplish a task could do so by simply specifying their intent in familiar natural language without prior knowledge of the underlying programming model, or its syntax and semantics. The code will get generated and even automatically run to produce the desired output.

However, as we have seen so far, the world is not ideal and even trained programmers face various challenges when programming with AI. These challenges are only exacerbated for end-user programmers, as a study by Srinivasa Ragavan et al. (2022) observes.

Participants in the study were data analysts (n=20) conducting exploratory data analysis in GridBook, a natural-language augmented spreadsheet system. In GridBook (Figure 6, adopted from Srinivasa Ragavan et al. (2022)) users can write spreadsheet formulas using the natural language (Figure 6: a-f); a formal formula is then synthesized from the natural language utterance. GridBook also infers the context of an utterance; for example, in Figure 6, the query in label 4 is a follow-up from label 3. Both the natural language utterance and the synthesized formula are persisted for users to edit and manipulate.

9.1. Issue 1: Intent specification, problem decomposition and computational thinking

When attempting to accomplish data analysis tasks using natural language, participants had to refine their specification of intent in the natural language several times, before they arrived at the desired result (if they did). The NL utterances were often underspecified, ambiguous, too complex, or contained domain phrases not specified in the context (e.g., in the data being analyzed). Thus, the first issue is to communicate the capabilities of the system, and make it interpretable so users can see how their prompt is being interpreted.

End-user programmers often lack computational thinking skills (Wing, 2011), such as the ability to decompose problems into subproblems, reformulate problems in ways that can be computed by a system, etc. However, effective use of LLMs such as Codex requires such skills. For example, if these models are most accurate when solutions to a problem are single line, then the user should be able to break their problem into smaller sub-problems each of which can be solved in one or two lines. Moreover, they might also lack the ability to frame a problem as generic computational problems, rather than domain-specific problems. For example, a realtor is more likely to ask “which is the largest house” (declaratively), instead of “which is the house with maximum constructed area” (procedurally).

Therefore, end-user computing environments powered by AI should help end-user programmers think “computationally”: they must aid users in breaking down their problems to smaller steps, or guiding users towards alternative strategies to specify or solve a problem (e.g., providing examples, offering alternatives) or even seek procedural prompts where needed (e.g., for disambiguation).

9.2. Issue 2: Code correctness, quality and (over)confidence

The second challenge is in verifying whether the code generated by the model is correct. In GridBook, users were able to see the natural language utterance, synthesized formula and the result of the formula. Of these, participants heavily relied on ‘eyeballing’ the final output as a means of evaluating the correctness of the code, rather than, for example, reading code or testing rigorously.

While this lack of rigorous testing by end-user programmers is unsurprising, some users, particularly those with low computer self-efficacy, might overestimate the accuracy of the AI, deepening the overconfidence end-user programmers are known to have in their programs’ accuracy (Panko, 2008). Moreover, end-user programmers might not be able to discern the quality of non-functional aspects of the generated code, such as security, robustness or performance issues.

9.3. Issue 3: Code comprehension and maintenance

A third challenge with AI-driven programming is the issue of code comprehension. During GridBook’s user evaluation, participants mentioned that the generated formulas are hard to understand, even when users were familiar with the target language. This has potentially severe consequences: from evaluating the accuracy of the program by verifying logic, to the ability to customize code, to future debugging and reuse. As we discussed earlier, this problem also exists for trained developers.

One approach to address this issue is for the AI system to include some notion of code readability or comprehensibility as a factor in code synthesis, such as during the learning phase, or when ranking suggestions, or even take it as input to the model (similar to the ‘temperature’ parameter in Codex). This approach is useful more broadly to synthesize high quality code, such as optimizing for performance or robustness. A second solution to tackle the comprehension problem is to explain the generated code to their users in a manner that is less ‘programmerese’ and more centered around the user’s current task and context. Initial evidence suggests that participants were open to these ideas; thus, these areas are ripe for future exploration.

9.4. Issue 4: Consequences of automation in end-user programming

In any AI system, we need to consider the consequences of automation. End-user programmers are known to turn to local experts or gardeners (end-user programmers with interest and expertise in programming who serve as gurus in the end-user programming environment) when they are unable to solve a part of the problem (Nardi, 1993; Sarkar & Gordon, 2018). Task-orientation tendencies combined with

challenges of completing their tasks easily also leaves end-user programmers with limited attention for testing, or carefully learning what is going on with their programs. Assuming that LLMs and associated user experiences will improve in the coming years, making end-user programming faster with LLMs than without, it is tempting to wonder whether the programmer can be persuaded to invest the saved time and attention to aspects such as learning or testing their programs; if so, what would it take to influence behaviour changes?

Another question is in the role of such experts. We conjecture that LLMs or similar AI capabilities will soon be able to answer a sizeable fraction of questions that end-user programmers will go to local experts for. An open question therefore is how the ecosystem of end-user programmers in organizations will change in their roles, importance and specialities. For example, will gardeners take on the role of educating users on better taking advantage of AI? If so, how can we communicate the working of such AI systems to technophile users and early adopters, so they can enable others in the organization?

9.5. Issue 5: No code, and the dilemma of the direct answer

Finally, it is not a foregone conclusion that users are even interested in code. As Blackwell’s model of attention investment notes, in many cases the user may be content to perform an action manually, rather than invest in creating a reusable automation (Blackwell, 2002a; J. Williams et al., 2020). Spreadsheet users, in particular, are often not sensitive to the level of automation or automatability of a given workflow, using a mix of manual, automated, and semi-automated techniques to achieve the goal at hand (Pandita et al., 2018).

Spreadsheet users often need ad-hoc transformations of their data that they will, in all likelihood, never need again. It may be that we can express this transformation as a program, but if the user is interested in the output and not the program, is it important, or even necessary, to communicate this fact to the user? One can argue that increasing the user’s awareness of the flexibility and fallibility of the process of delivering an inferred result (i.e., enabling them to *critically evaluate* the output (Sarkar et al., 2015)) can build agency, confidence, trust, and resilience. This issue is related to information retrieval’s “dilemma of the direct answer” (Potthast et al., 2021), raised in response to the increased phenomenon of search engines directly answering queries in addition to simply listing retrieved results.

However, if the programming language used is not related to the languages familiar to the end-user, or the user is a complete novice, it is exceedingly difficult for them to make any sense of it, as was shown by Lau et al. (2021) in their study of Excel users encountering Python code. Yet, there are socio-technical motivations for using an unfamiliar target language: long-term testing of LLM assistance shows that it shines when paired with high-level APIs that capture use cases well (Section 7). One advantage of the Python ecosystem is that it has an unparalleled set of libraries and APIs for data wrangling. An LLM-assisted tool that emits Excel formulas is therefore less likely to solve user problems than Python statements. In the longer term, this might be mitigated by developing a rich set of data manipulation libraries in the Excel formula language.

10. Conclusion

Large language models have initiated a significant change in the scope and quality of program code that can be automatically generated, compared to previous approaches. Experience with commercially available tools built on these models suggests that they represent a new way of programming. LLM assistance transforms almost every aspect of the experience of programming, including planning, authoring, reuse, modification, comprehension, and debugging.

In some aspects, LLM assistance resembles a highly intelligent and flexible compiler, or a partner in pair programming, or a seamless search-and-reuse feature. Yet in other aspects, LLM-assisted programming has a flavour all of its own, which presents new challenges and opportunities for human-centric programming research. Moreover, there are even greater challenges in helping non-expert end users benefit from such tools.

A. Experience report sources

This appendix contains a list of sources we draw upon for the quotes and analysis in Section 7. While all sources were included in our analysis, we did not draw direct quotes from every source in this list.

A.1. Blog posts and corresponding Hacker News discussions

1. Andrew Mayne, March 17 2022, “Building games and apps entirely through natural language using OpenAI’s code-davinci model”. URL: <https://andrewmayneblog.wordpress.com/2022/03/17/building-games-and-apps-entirely-through-natural-language-using-openais-davinci-code-model/>. Hacker News discussion: <https://news.ycombinator.com/item?id=30717773>
2. Andrew Mouboussin, March 24 2022, “Building a No-Code Machine Learning Model by Chatting with GitHub Copilot”. URL: <https://www.surgehq.ai/blog/building-a-no-code-toxicity-classifier-by-talking-to-copilot>. Hacker News discussion: <https://news.ycombinator.com/item?id=30797381>
3. Matt Rickard, August 17 2021, “One Month of Using GitHub Copilot”. URL: <https://matt-rickard.com/github-copilot-a-month-in/>.
4. Nutanc, November 15 2021, “Using Github copilot to get the tweets for a keyword and find the sentiment of each tweet in 2 mins”. URL: <https://nutanc.medium.com/using-github-copilot-to-get-the-tweets-for-a-keyword-and-find-the-sentiment-of-each-tweet-in-2-mins-9a531abedc84>.
5. Tanishq Abraham, July 14 2021, “Coding with GitHub Copilot”. URL: https://tmabraham.github.io/blog/github_copilot.
6. Aleksej Komnenovic, January 17 2022, “Don’t fully trust AI in dev work! /yet”. URL: <https://akom.me/dont-fully-trust-ai-in-dev-work-yet>.

A.2. Miscellaneous Hacker News discussions

1. <https://news.ycombinator.com/item?id=30747211>
2. <https://news.ycombinator.com/item?id=31390371>
3. <https://news.ycombinator.com/item?id=31020229&p=2>
4. <https://news.ycombinator.com/item?id=29760171>
5. <https://news.ycombinator.com/item?id=31325154>
6. <https://news.ycombinator.com/item?id=31734110>
7. <https://news.ycombinator.com/item?id=31652939>
8. <https://news.ycombinator.com/item?id=30682841>
9. <https://news.ycombinator.com/item?id=31515938>
10. <https://news.ycombinator.com/item?id=31825742>

References

- Allamanis, M., Barr, E. T., Devanbu, P. T., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), 81:1–81:37. Retrieved from <https://doi.org/10.1145/3212695> doi: 10.1145/3212695
- Allamanis, M., & Brockschmidt, M. (2017). Smartpaste: Learning to adapt source code. *arXiv preprint arXiv:1705.07867*.
- Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... Sutton, C. (2021). *Program synthesis with large language models*. arXiv. Retrieved from <https://arxiv.org/abs/2108.07732> doi: 10.48550/ARXIV.2108.07732
- Barik, T., Ford, D., Murphy-Hill, E., & Parnin, C. (2018). How should compilers explain problems to developers? In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 633–643).
- Barik, T., Johnson, B., & Murphy-Hill, E. (2015). I heart hacker news: expanding qualitative research findings by analyzing social news websites. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 882–885).
- Barke, S., James, M. B., & Polikarpova, N. (2022). *Grounded copilot: How programmers interact with code-generating models*. arXiv. Retrieved from <https://arxiv.org/abs/2206.15000> doi: 10.48550/ARXIV.2206.15000
- Basman, A., Church, L., Klokmose, C. N., & Clark, C. B. (2016). Software and how it lives on-embedding live programs in the world around them. In *Ppig* (p. 19).
- Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? In M. C. Elish, W. Isaac, & R. S. Zemel (Eds.), *Facct '21: 2021 ACM conference on fairness, accountability, and transparency, virtual event / toronto, canada, march 3-10, 2021* (pp. 610–623). ACM. Retrieved from <https://doi.org/10.1145/3442188.3445922> doi: 10.1145/3442188.3445922
- Bergström, I., & Blackwell, A. F. (2016). The practices of programming. In *2016 ieee symposium on visual languages and human-centric computing (vl/hcc)* (pp. 190–198).
- Blackwell, A. F. (2002a). First steps in programming: A rationale for attention investment models. In *Proceedings ieee 2002 symposia on human centric computing languages and environments* (pp. 2–10).
- Blackwell, A. F. (2002b). What is programming? In *Ppig* (p. 20).
- Bødker, S. (2015). Third-wave hci, 10 years later - participation and sharing. *Interactions*, 22(5), 24–31. Retrieved from <https://doi.org/10.1145/2804405> doi: 10.1145/2804405
- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... Amodei, D. (2020). Language models are few-shot learners.
- Cao, J., Fleming, S. D., Burnett, M., & Scaffidi, C. (2015). Idea garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*, 27(6), 640–660.
- Chalhoub, G., & Sarkar, A. (2022). “It’s Freedom to Put Things Where My Mind Wants”: Understanding and Improving the User Experience of Structuring Data in Spreadsheets. In *CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3491102.3501833> doi: 10.1145/3491102.3501833

- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., ... Zaremba, W. (2021). Evaluating large language models trained on code. *CoRR*, *abs/2107.03374*. Retrieved from <https://arxiv.org/abs/2107.03374>
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., ... Zaremba, W. (2021). Evaluating large language models trained on code. *ArXiv*, *abs/2107.03374*.
- Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., ... Fiedel, N. (2022). Palm: Scaling language modeling with pathways. *ArXiv*, *abs/2204.02311*.
- Colmerauer, A., & Roussel, P. (1996). The birth of prolog. In *History of programming languages—ii* (pp. 331–367).
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019, June). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the north American chapter of the association for computational linguistics: Human language technologies, volume 1 (long and short papers)* (pp. 4171–4186). Minneapolis, Minnesota: Association for Computational Linguistics. Retrieved from <https://aclanthology.org/N19-1423> doi: 10.18653/v1/N19-1423
- Green, T., & Blackwell, A. (1998). Cognitive dimensions of information artefacts: a tutorial. In *Bcs hci conference* (Vol. 98, pp. 1–75).
- Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, 443–460.
- Green, T. R., & Petre, M. (1992). When visual programs are harder to read than textual programs. In *Human-computer interaction: Tasks and organisation, proceedings of ecce-6 (6th european conference on cognitive ergonomics)*. *gc van der veer, mj tauber, s. bagnarola and m. antavolits. rome, cud* (pp. 167–180).
- Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In T. Ball & M. Sagiv (Eds.), *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2011, austin, tx, usa, january 26-28, 2011* (pp. 317–330). ACM. Retrieved from <https://doi.org/10.1145/1926385.1926423> doi: 10.1145/1926385.1926423
- Hannay, J. E., Dybå, T., Arisholm, E., & Sjøberg, D. I. (2009). The effectiveness of pair programming: A meta-analysis. *Information and software technology*, *51*(7), 1110–1122.
- Henley, A. Z., & Fleming, S. D. (2014). The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 2511–2520).
- Hermans, F., Pinzger, M., & van Deursen, A. (2015). Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, *20*(2), 549–575.
- Hindle, A., Barr, E. T., Gabel, M., Su, Z., & Devanbu, P. T. (2016). On the naturalness of software. *Commun. ACM*, *59*(5), 122–131. Retrieved from <https://doi.org/10.1145/2902362> doi: 10.1145/2902362
- Hindle, A., Barr, E. T., Su, Z., Gabel, M., & Devanbu, P. T. (2012). On the naturalness of software. In M. Glinz, G. C. Murphy, & M. Pezzè (Eds.), *34th international conference on software engineering, ICSE 2012, june 2-9, 2012, zurich, switzerland* (pp. 837–847). IEEE Computer Society. Retrieved from <https://doi.org/10.1109/ICSE.2012.6227135> doi: 10.1109/ICSE.2012.6227135

- Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 576–580. Retrieved from <https://doi.org/10.1145/363235.363259> doi: 10.1145/363235.363259
- Hochreiter, S., & Schmidhuber, J. (1997, nov). Long short-term memory. *Neural Comput.*, 9(8), 1735–1780. Retrieved from <https://doi.org/10.1162/neco.1997.9.8.1735> doi: 10.1162/neco.1997.9.8.1735
- Horvitz, E. (1999). Principles of mixed-initiative user interfaces. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 159–166).
- Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1985). Direct manipulation interfaces. *Hum. Comput. Interact.*, 1(4), 311–338. Retrieved from https://doi.org/10.1207/s15327051hci0104_2 doi: 10.1207/s15327051hci0104_2
- Imai, S. (2022). Is github copilot a substitute for human pair-programming? an empirical study. In *2022 ieee/acm 44th international conference on software engineering: Companion proceedings (icse-companion)* (pp. 319–321).
- Jiang, E., Toh, E., Molina, A., Olson, K., Kayacik, C., Donsbach, A., ... Terry, M. (2022). Discovering the syntax and strategies of natural language programming with generative language models. In *Chi conference on human factors in computing systems* (pp. 1–19).
- Kery, M. B., & Myers, B. A. (2017). Exploring exploratory programming. In *2017 ieee symposium on visual languages and human-centric computing (vl/hcc)* (pp. 25–29).
- Ko, A. J., & Myers, B. A. (2004). Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 151–158).
- Kulesza, T., Amershi, S., Caruana, R., Fisher, D., & Charles, D. (2014). Structured labeling for facilitating concept evolution in machine learning. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 3075–3084).
- Kurlander, D., Cypher, A., & Halbert, D. C. (1993). *Watch what i do: programming by demonstration*. MIT press.
- Lau, S., Srinivasa Ragavan, S. S., Milne, K., Barik, T., & Sarkar, A. (2021). Tweakit: Supporting end-user programmers who transmogrify code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (pp. 1–12).
- Li, J., Tang, T., Zhao, W. X., & Wen, J.-R. (2021, 8). Pretrained language model for text generation: A survey. In Z.-H. Zhou (Ed.), *Proceedings of the thirtieth international joint conference on artificial intelligence, IJCAI-21* (pp. 4492–4499). International Joint Conferences on Artificial Intelligence Organization. Retrieved from <https://doi.org/10.24963/ijcai.2021/612> (Survey Track) doi: 10.24963/ijcai.2021/612
- Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... Vinyals, O. (2022b). *Competition-level code generation with alphacode*. arXiv. Retrieved from <https://arxiv.org/abs/2203.07814> doi: 10.48550/ARXIV.2203.07814
- Li, Y., Choi, D. H., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... Vinyals, O. (2022a). Competition-level code generation with alphacode. *ArXiv, abs/2203.07814*.
- Lieberman, H. (2001). *Your wish is my command: Programming by example*. Morgan Kaufmann.

- Lieberman, H., & Liu, H. (2006). Feasibility studies for programming in natural language. In *End user development* (pp. 459–473). Springer.
- Liu, S., Chen, Y., Xie, X., Siow, J. K., & Liu, Y. (2021). Retrieval-augmented generation for code summarization via hybrid GNN. In *International conference on learning representations*. Retrieved from <https://openreview.net/forum?id=zv-typlgPxA>
- Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., ... Liu, S. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv, abs/2102.04664*.
- Luger, E., & Sellen, A. (2016). "like having a really bad pa" the gulf between user expectation and experience of conversational agents. In *Proceedings of the 2016 chi conference on human factors in computing systems* (pp. 5286–5297).
- Macvean, A., Church, L., Daughtry, J., & Citro, C. (2016). Api usability at scale. In *Ppig* (p. 26).
- Madi, N. A. (2022). How readable is model-generated code? examining readability and visual inspection of github copilot. *arXiv preprint arXiv:2208.14613*.
- Marasoiu, M., Church, L., & Blackwell, A. (2015). An empirical investigation of code completion usage by professional software developers. In *PPIG*.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, & K. Weinberger (Eds.), *Advances in neural information processing systems* (Vol. 26). Curran Associates, Inc. Retrieved from <https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf>
- Miller, L. A. (1981). Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2), 184–215.
- Mou, L., Li, G., Zhang, L., Wang, T., & Jin, Z. (2016). Convolutional neural networks over tree structures for programming language processing. In *Aaai*.
- Mu, J., & Sarkar, A. (2019). Do we need natural language? Exploring restricted language interfaces for complex domains. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems* (pp. 1–6).
- Myers, B. A. (1992). Demonstrational interfaces: A step beyond direct manipulation. *Computer*, 25(8), 61–73.
- Myers, B. A., & Stylos, J. (2016). Improving api usability. *Communications of the ACM*, 59(6), 62–69.
- Nardi, B. A. (1993). *A small matter of programming: perspectives on end user computing*. MIT press.
- Nguyen, A. T., Nguyen, T. T., & Nguyen, T. N. (2015). Divide-and-conquer approach for multi-phase statistical migration for source code (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 585–596.
- Pandita, R., Parnin, C., Hermans, F., & Murphy-Hill, E. (2018). No half-measures: A study of manual and tool-assisted end-user programming tasks in excel. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 95–103).
- Panko, R. R. (2008). Reducing overconfidence in spreadsheet development. *arXiv preprint arXiv:0804.0941*.

- Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2021). *Asleep at the keyboard? assessing the security of github copilot's code contributions*. arXiv. Retrieved from <https://arxiv.org/abs/2108.09293> doi: 10.48550/ARXIV.2108.09293
- Piccioni, M., Furia, C. A., & Meyer, B. (2013). An empirical study of api usability. In *2013 acm/ieee international symposium on empirical software engineering and measurement* (pp. 5–14).
- Potthast, M., Hagen, M., & Stein, B. (2021). The dilemma of the direct answer. In *Acm sigir forum* (Vol. 54, pp. 1–12).
- Raychev, V., Vechev, M. T., & Krause, A. (2015). Predicting program properties from "big code". In S. K. Rajamani & D. Walker (Eds.), *Proceedings of the 42nd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2015, mumbai, india, january 15-17, 2015* (pp. 111–124). ACM. Retrieved from <https://doi.org/10.1145/2676726.2677009> doi: 10.1145/2676726.2677009
- Rouchy, P. (2006). Aspects of prolog history: Logic programming and professional dynamics. *Blekinge Institute of Technology, Sweden*.(English). *TeamEthno-Online*(2), 85–100.
- Salge, C. A. D. L., & Berente, N. (2016). Pair programming vs. solo programming: What do we know after 15 years of research? In *2016 49th hawaii international conference on system sciences (hicc)* (pp. 5398–5406).
- Sarkar, A. (2016). *Interactive analytical modelling* (Tech. Rep. No. UCAM-CL-TR-920). University of Cambridge, Computer Laboratory. Retrieved from <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-920.pdf> doi: 10.48456/tr-920
- Sarkar, A. (2022, March). Is explainable AI a race against model complexity? In *Workshop on Transparency and Explanations in Smart Systems (TeXSS), in conjunction with ACM Intelligent User Interfaces (IUI 2022)* (pp. 192–199). Retrieved from <http://ceur-ws.org/Vol-3124/paper22.pdf>
- Sarkar, A., & Gordon, A. D. (2018, September). How do people learn to use spreadsheets? (work in progress). In *Proceedings of the 29th Annual Conference of the Psychology of Programming Interest Group (PPIG 2018)* (pp. 28–35).
- Sarkar, A., Jamnik, M., Blackwell, A. F., & Spott, M. (2015). Interactive visual machine learning in spreadsheets. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 159–163).
- Sarkar, A., Srinivasa Ragavan, S., Williams, J., & Gordon, A. D. (2022). End-user encounters with lambda abstraction in spreadsheets: Apollo's bow or Achilles' heel? In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.
- Shneiderman, B., & Norwood, N. (1993). 1.1 direct manipulation: a step beyond programming. *Sparks of innovation in human-computer interaction*, 17.
- Silver, A. (2018, May). *Introducing visual studio intellicode*. Microsoft. Retrieved from <https://devblogs.microsoft.com/visualstudio/introducing-visual-studio-intellicode/>
- Srinivasa Ragavan, S., Hou, Z., Wang, Y., Gordon, A. D., Zhang, H., & Zhang, D. (2022). Gridbook: Natural language formulas for the spreadsheet grid. In *27th international conference on intelligent user interfaces* (p. 345–368). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3490099.3511161> doi: 10.1145/3490099.3511161

- Srinivasa Ragavan, S., Kuttal, S. K., Hill, C., Sarma, A., Piorkowski, D., & Burnett, M. (2016). Foraging among an overabundance of similar variants. In *Proceedings of the 2016 chi conference on human factors in computing systems* (pp. 3509–3521).
- Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Proceedings of the 27th international conference on neural information processing systems - volume 2* (p. 3104–3112). Cambridge, MA, USA: MIT Press.
- Tanimoto, S. L. (2013). A perspective on the evolution of live programming. In *2013 1st international workshop on live programming (live)* (pp. 31–34).
- Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts* (pp. 1–7).
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. In *Proceedings of the 31st international conference on neural information processing systems* (p. 6000–6010). Red Hook, NY, USA: Curran Associates Inc.
- Wei, J., Goyal, M., Durrett, G., & Dillig, I. (2020). Lambdanet: Probabilistic type inference using graph neural networks. *ArXiv, abs/2005.02161*.
- Wei, Y., Chandrasekaran, N., Gulwani, S., & Hamadi, Y. (2015, May). *Building bing developer assistant* (Tech. Rep. No. MSR-TR-2015-36). Retrieved from <https://www.microsoft.com/en-us/research/publication/building-bing-developer-assistant/>
- Weiss, D. (2022, Jun). *Blog / tabnine announcements / announcing our next-generation ai models*. Tabnine. Retrieved from <https://www.tabnine.com/blog/announcing-tabnine-next-generation/>
- Williams, J., Negreanu, C., Gordon, A. D., & Sarkar, A. (2020). Understanding and inferring units in spreadsheets. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 1–9).
- Williams, L. A., & Kessler, R. R. (2000). All i really need to know about pair programming i learned in kindergarten. *Communications of the ACM*, 43(5), 108–114.
- Wing, J. (2011). Research notebook: Computational thinking—what and why. *The link magazine*, 6, 20–23.
- Xu, F. F., Alon, U., Neubig, G., & Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*.
- Xu, F. F., Vasilescu, B., & Neubig, G. (2022). In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2), 1–47.
- Yoon, Y., & Myers, B. A. (2015). Supporting selective undo in a code editor. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 1, pp. 223–233).
- Zhang, H., Jain, A., Khandelwal, G., Kaushik, C., Ge, S., & Hu, W. (2016). Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (pp. 956–961).
- Ziegler, A. (2021, Jun). *Github copilot research recitation*. Microsoft. Retrieved from <https://github.blog/2021-06-30-github-copilot-research-recitation/>

Ziegler, A., Kalliamvakou, E., Simister, S., Sittampalam, G., Li, A., Rice, A., . . . Aftandilian, E. (2022). Productivity assessment of neural code completion. *arXiv preprint arXiv:2205.06537*.

POGIL-like learning and student's impressions of software engineering topics: A qualitative study

Bhuvana Gopal
School of Computing
University of Nebraska-Lincoln
bhuvana.gopal@unl.edu

Ryan Bockmon
School of Computing
University of Nebraska-Lincoln
ryan.bockmon@huskers.unl.edu

Stephen Cooper
School of Computing
University of Nebraska-Lincoln
stephen.cooper@unl.edu

Abstract

In this study, we analyze students' impressions and perceptions of professional software engineering topics and practices. We explore student thoughts on the interconnections between various industry relevant software engineering topics such as requirements analysis, UI/UX, work patterns, agile communication, documentation, and business value. We studied student voices in a semester long undergraduate software engineering course, after they underwent instruction using POGIL-like, a guided inquiry based pedagogy. At the end of the course, we collected student responses to open ended questions regarding their perceptions of professional, end user software engineering topics. We combined student responses to these questions with researcher memos as well as reflective researcher journals. We analyzed these written responses using content analysis and identified the themes that the data yielded. The main themes that emerged from our qualitative analysis were: 1) Software is more than just a tool to solve business problems. 2) The need for documentation varies based on the process model adopted. 3) Timely and frequent communication between team members and stakeholders is essential. 4) Downtime during work is best used to further the sprint goals of the team or improve one's technical acumen. We attempt to understand if POGIL-like helped students develop a professionally sound understanding of basic software engineering topics and how they work to get a software product developed, from requirements to release.

1. Introduction

Student perceptions of a discipline are closely tied to their motivation to do well in it. When students feel comfortable with the environment they are in and the abilities they need to study the discipline, they tend to remain in the program more often than not. Retention is closely tied to motivation (Tinto, 1997; Bean, Eaton, et al., 2000). A good understanding of student perceptions of the social and academic experiences in software engineering can help improve student motivation in the discipline.

In an attempt to understand how CS students perceived various topics in software engineering (SE) and how they saw themselves in the discipline, we conducted a qualitative study with 24 sophomore/junior students with varying computing majors. We used POGIL-like (our implementation of Process Oriented Guided Inquiry based Learning) as the pedagogical approach. Our goal was to understand if students were able synthesize concepts from SE topics beyond just a basic level understanding of each topic, as well as how they perceived themselves in the discipline of software engineering.

The rest of the paper is organized as follows. We present prior work in software testing, POGIL-like, and student reflections in Section 2. We present our research question in Section 3. Our research methods are explained in Section 4. Themes from our data analysis are presented in Section 5, with a detailed discussion in Section 6. We detail the threats to the validity of our study in Section 7, and conclude in Section 8.

2. Prior Work

There are several studies on students' motivation and perceptions across various fields, and specifically in the STEM disciplines including CS. Some studies involve pedagogical interventions that have shown to affect student motivation as well.

Ames and Archer (Ames & Archer, 1988) showed that students' perceptions of classroom climate were related to specific motivational variables that have implications for the development of self-regulated learning as well as a long term involvement and interest in learning. Biggers et al. (Biggers, Brauer, & Yilmaz, 2008) discuss ways to keep student interest in CS, and strengthen their experiences. One of their key takeaways that an asocial, community-absent environment with limited human interaction among peers was a deterrent in students staying motivated and interested in CS. They also found that several students found CS work to be time consuming, boring and tedious, with a pronounced lack of social interaction. Students also had little to no idea of what a CS career looks like. This led to students dropping out within a relatively short period. Agosto (Agosto, Gasson, & Atwood, 2008) measured students' perceptions, attitudes, self-efficacy, and identity with respect to their intentions to further pursue computer science. In their study, self-perception regarding one's own ability to use CS techniques for problem solving, and identity (whether students thought they were computer scientists) emerged as the primary driver for differences in intention.

Peters (Peters & Pears, 2013) constructed a theory based framework to study students' CS identities, how students perceive learning CS and IT as meaningful, and how they are supported or hindered by their education. They found many students are influenced by their prior experiences with computing as far as their CS identity. Social interactions and group experience participation play an important role in shaping the CS identity of students. Dempsey et al. (Dempsey, Snodgrass, Kishi, & Titcomb, 2015) studied how to recruit and retain women in CS. They found that an increased CS self perception, specifically in terms of students' perceived self ability to be a computer scientist was a driving force in who stayed in CS.

Souza (Souza, Moreira, & Figueiredo, 2019) investigated students' perception on the use of problem-based learning (PBL) in an introductory SE course and found that there is a positive perception of the contribution of the project assignment on learning specific SE topics (such as software requirements, software Design, and agile methods), and this perception was even more positive for the students in the PBL course. Melnik (Melnik & Maurer, 2005) explored student perceptions of agile methods. Their experiences introducing agile methods in the CS courses indicate that students are enthusiastic about core agile practices and that there are no significant differences in the perceptions of students of various levels of educational programs. Almulla (Almulla, 2020) studied PBL in relation to students' motivation regarding their development of feelings of autonomy, competence, and relatedness (Almulla, 2020). They found a significant correlation between PBL and student engagement.

As seen from the studies above, there are research questions regarding student perceptions and motivation in CS and SE that have been answered. This is still a fledgling field, with many aspects yet unexplored. There are not many studies on how students perceive SE topics especially in the light of collaborative and active learning pedagogies, and through our study, we hope to fill the gap.

3. Research Question

In this paper we study students' perceptions of agile SE topics, within the context of a POGIL-like software engineering course. Our research question for this study was:

RQ: Were undergraduate students able to learn from and build upon multiple relevant concepts to display a connected understanding of software engineering topics when instructed using a POGIL-like pedagogy?

4. Method

4.1. POGIL-like: The approach

We chose to implement POGIL in the software engineering classroom since it is a process heavy approach, spurring students to think deeper and co-construct (Ben-Ari, 1998) concepts in a small group setting, through inquiry and application (Kusssmaul, Mayfield, & Hu, 2017). We call our approach POGIL-like since POGIL is a copyrighted term to be used only by the POGIL project (*CS-POGIL | DCV (Directed, Convergent, Divergent) Questions*, n.d.).

POGIL-like is a pedagogy where students are organized into small teams. Students collaborate and actively learn to work together (Kusssmaul, 2011). Students start each class session with little or no prior knowledge of the topic, so they can benefit from the co-construction of knowledge through POGIL-like activities without added misconceptions. The instructor serves as an active facilitator, walking around the classroom and helping students as needed during the session. Each team consists of 4-6 students, and each student has a specific role to play. There are 4 roles: Manager, Recorder, Presenter and Reflector. Students engage with "models" to learn the content and complete "activities" to exercise their understanding of the content. An overview of the POGIL-like pedagogy and its salient features can be found in our previous work (Gopal & Cooper, 2022).

More information on the types of questions, the POGIL-like "model" and activities can be found in our paper on POGIL-like (Gopal & Cooper, 2022). POGIL-like a combination of one or more models and a set of one or more activities in each session. Models contain content presented with tables, figures, action verbs, and some text. Each model is followed by one or more activities. A model-activity set combination is called a POGIL-like cycle. This cycle encourages students to explore (E), invent (I) and apply (A) the content concepts. To create the activities we used three types of questions -Directed (D), Convergent (C) and Divergent (V) questions (Gopal & Cooper, 2022). Several such POGIL-like learning cycles (E-I-A) consisting of models and corresponding D/C/V questions were used in our classroom implementation.

4.2. Study Context

Our research project was reviewed and classified as exempt by our University's Institutional Review Board. Students who chose to participate in the study did so by granting the researchers their informed consent. Data for this study were collected from 24 participants of a cohort of sophomore/junior students taking a software engineering class in the Fall of 2021. All students were taught using POGIL-like in the classroom in person.

4.3. Data Collection

Students were requested to answer the same online questionnaire at the beginning and end of the semester. We created this questionnaire specifically to target concept correlations between SE topics such as requirements gathering, Agile ceremonies and artifacts, and software testing. This open ended questionnaire asked students to describe what they thought of various topics and workflow in the real world. We used prompts such as "What is the role of requirements elicitation in SE?", "In agile software engineering, how does communication take place between team members and stakeholders?", and "When you get stuck during development, what would you typically do?" We combined student responses for these questionnaires with our real time field notes and reflective journal on students and their individual performances in various roles during the POGIL-like sessions (Emerson, Fretz, & Shaw, 2011). During our analysis, we corroborated students' written responses to the open ended questionnaire along with these field-notes and journal entries.

4.4. Data Analysis

Our data analysis consisted of content analysis (Hsieh & Shannon, 2005). We identified themes based on a mutimethod data collection (including participant/non-participant observations and field notes). Our primary goals during content analysis were to identify patterns in ideas as expressed by our students in their written responses. The first and second author of this paper coded our participants' written responses individually, generating and assigning over 230 codes. We employed frequency mapping of

codes iteratively both manually and using the HyperResearch software (*Qualitative Data Analysis tool - HyperResearch*, n.d.). We arrived at a code map (Anfara, Brown, & Mangione, 2002) which we used to help the process of code grouping and categorization into themes. We converged the code groups into the notes from our field notes and journals to come up with our themes by two separate coders independently and in parallel (Strauss & Corbin, 2015). In doing so, we were able to identify, verify and collate the themes we both noted.

We present the following data sections where we aim to forefront the opinions and perceptions of our participants in their own voices. We have followed existing research guidelines on how to position participant participation, and our method for meaning making involves understanding recurring expressions of participant sentiments and ideas by taking into account the context in which they were written and submitted (Creswell & Poth, 2018; Ketelhut & Schifter, 2011; Foley, 2002). In the following section, we highlight participants' voices by thematically presenting their written questionnaire responses. We use participants' own words as subheadings in each theme presentation. We used pseudonyms for our participants' actual names. Our primary focus in this study was to understand students' perceptions of SE topics. When instructed using the POGIL-like pedagogical approach, were students able to make connections between various SE concepts? How did their perceptions of SE concepts and topics relate to their identity in CS? These were the main questions we sought to answer.

4.5. Reliability

Intercoder agreement or inter-rater reliability is a measure of reliability commonly used in qualitative studies (Creswell & Poth, 2018). We focused on intercoder agreement to ensure the reliability of our coded data based on multiple coders interpreting the data through the process of coding, code mapping, categorization and theming. The first two authors of this paper, both trained in qualitative research methods, served as coders for our data. We calculated Cohen's Kappa (Hsu & Field, 2003) to measure intercoder reliability and we report an intercoder reliability (Creswell & Poth, 2018) of 0.9 (90%) among all themes that emerged from our 230 codes.

5. Results: Themes

In the following subsections we position the voices of our participants and start with how they broadly viewed the purpose of software. We explore their thoughts on requirements gathering, and the role of documentation. How do students think about communication in an agile SE environment, and how do they handle being stuck during everyday development?

5.1. Theme 1: Software is more than just a tool to solve business problems

We found that students viewed software as being important to end users, beyond just the business needs. While software did solve business problems, our students found it to be a source of potentially informing, educating, and ultimately, uniting end users.

"Software is often created with input from domain experts. They usually provide insights into how a problem would be solved without computers, and then developers use that solution to help them solve general cases." - Pete.

"Software is just more than to solve domain problems, more after this pandemic software use is increased exponentially. Software has the potential to unite people." - Kim.

"In my opinion, software can do more than just solving domain problems. It can also be used to inform, educate and help raise awareness on social issues. I think that software can be, and is, used everywhere in today's world. I think that's also obvious to see when every company hires a software engineer." - Tim.

"Software is used mostly for solving a company's specific problems. Software cannot be used to solve every single problem that a company may have, but if a company has a specific problem, software can be a great solution to solve it." - John.

"Software is used to solve problems. But it is also an experience for the user to enjoy. Sometimes (de-

pending on what type of software you are talking about) the purpose could be solely for enjoyment. Although the idea for software development is to solve a problem/addition that can be used by consumers." - Alia.

Software is mostly used to improve and solve modern problems efficiently, quickly, and accurately. Students disagreed with the thought that software is primarily used as a tool for solving problems specific to a business and its needs - they opined that it could do more than that. They expressed that while software could reduce the time it takes to solve or completely solve problems for a business need, it could also be used to enhance an end users experience of the solution. It seems that students struggled to articulate what exactly the additional value of software was besides solving business/domain problems.

5.2. Theme 2: The need for documentation varies based on the process model adopted

The role of documentation in SE takes center stage in this section. Our students went beyond the surface level understanding that documentation is required and helps maintain software; they understood and expressed that the amount and granularity of documentation required in a software development project depends on many factors, as we detail below.

"Documentation truly depends on what kind of approach you are taking to developing software. Although all approaches should have some documentation, the amount of documentation and when the documentation is created depends on the approach. For example, if a business is in need of high security software that does a pre determined set of functions and will not change as it is vital software, a business will probably take a waterfall approach...documentation is followed and developed depends on the needs of the customer and the kind of software that is being developed." - Matt.

"Documentation will depend largely on the business's needs and what type of software project it is. If the business knows exactly what they need up front, and give those needs to the development team, documentation is more important. If a project is made agilely, the documentation may be more minimal because the software is everchanging." - Paul.

"There are times when documentation is not needed. One scenario would be when you are making software for yourself, and there really is not any reason for documentation to slow down the process." - Linda.

"Depends on the type of business needs driving the software, because I don't think it's so clean cut. For some businesses that just need simple software like the addition of a search bar to their online store doesn't need too much, it can be minimal. However a place like a hospital dealing with things like patient records should be pivotal because of all the care that needs to be taken with a database and formatting of records along with other requirements that need to be in place." - Lupe.

"It is absolutely critical that people other than you can understand your code. Even if you step away from your code, it makes it easier for you to also understand your code. In a waterfall method, there isn't a ton of revolving documentations, whereas in agile, the documentation is necessary." - Hannah.

Our students displayed a strong understanding of the idea that documentation is important, but depends on the business needs. They expressed an understanding of the different documentation approaches warranted by different software process models. They understood that how important documentation is can depend on a variety of factors, and that for some businesses it can be critical, but for some others it can be relatively minimal or unnecessary.

5.3. Theme 3: Timely and frequent communication between team members and stakeholders is essential

We can classify our participants' voices on agile communication under three topics: frequency of communication, parties involved in communication and the importance of communication. Our students displayed a good understanding of the evolving nature of requirements (when developing software using an agile process) and how the frequency of communication within the team and with stakeholders could affect the quality of the end product.

"Requirements change in every sprint - sometimes they could change drastically. It is very important that we keep communicating with the stakeholders and the other devs so that we capture and code to the correct requirements for each sprint." - Matt.

"Scrum masters, project managers, developers and product owners should all be involved in regular communication. Daily scrums for the development team, and at least weekly meetings with the stakeholders are crucial." - Sarah.

"The quality of the software being developed is highly dependent on how well the team communicated with itself and with the clients. Without getting requirements right, code won't be done correctly, and tests will be testing bad code that doesn't reflect what the customer wants." - Rachel.

"Without constant communication, the product will fail - how will developers know what the clients want as requirements change? How will they develop and test?" - Alia.

Based on the above statements, we can see that our students understood and advocated frequent communication between all parties involved - daily for team members and weekly for other stakeholders. Students also clearly understood the evolving nature of requirements on an agile project, and the importance of timely communication with stakeholders so that the correct requirements were captured for each sprint. The importance of meaningful and effective communication cannot be overstated in an agile environment, and our students exhibited an understanding of the ramifications of ineffective communication.

5.4. Theme 4: Handling downtime and time management in software engineering

We asked students how they would proceed when they were unable to proceed during development due to limiting circumstances. They had to put themselves in the context of a professional software engineering job, and had to think through the lens of a junior software engineer. The following comments illuminate their thought process on this topic. This is important to know because how a student proceeds when they are stuck, could indicate how they perceive themselves in the context of software engineering.

"Take a break, even though we are just sitting down coding drains a lot of energy, generally a lot of programmer feel mental stress, so relaxation is really important. I would take a time out and relax bring my way back to work with stable mental health." - Tim.

"During downtime I would want to try and relax to clear my mind so I could think of possible projects to work on that would be beneficial to my role at the company. I would also want to mingle with other developers to learn more about different types of projects." - Asher.

"I would probably work on resolving tasks in the sprint backlog. If there isn't anything in the sprint backlog, I guess I would create some more unit tests just to make sure the methods work. If there are already plenty of unit tests, I suppose I would work on some documentation or double checking the code for code smells." - Mike.

"Continue working on other items, bugs, or features that needed to be done. Interact with co-workers and gain insight with what they are working on or take a longer break. There is always a cycle to it and eventually you need to take advantage of it." - Wayne.

"When I have downtime during a workday as an industry professional I would try to pick up more responsibilities from the team and try to learn new skills." - Cody.

There are several threads that emerge from the above quotes: students want to individually help move the sprint/product backlog forward, focusing on immediate development needs; students wish to help their fellow teammates, fostering a sense of solidarity and teamwork; students want to improve their technical skills - learning new technologies and methodology elements; and finally, students recognize the mental stress that comes with agile software development with its constant deadlines, and wish to use downtime to focus on improving their mental health.

6. Discussion

6.1. Key takeaways

The most important takeaway from our analysis is that our students connected elements of several software engineering topics together. Their understanding of many topics was practical, well informed and deeper than a simple surface level understanding of said topic. For example, they were able to relate requirements to testing, communication to requirements, and the Agile process to end user focused software design. This is noteworthy because this was the first SE course that most of our students had encountered in their academic preparation. Students also expressed a clear sense of belonging in the way they spoke about SE topics, and their opinions on various issues relating to them.

Several interesting themes emerge from our analysis. First, our students said that software is more than a tool to simply solve domain problems. They perceived software development in general as leading to a product for the end user to enjoy. Our students also saw software as a living and breathing product. Changes could be constantly made and updates could be pushed out every other sprint. Software can therefore adapt to changes in the real world, in terms of business needs and end user human needs. It is interesting to note how students captured the intangible idea of software being more than the sum of its parts, with an ability to potentially inform, educate, and/or uplift human beings. In summary, according to our participants, software is designed to solve domain problems, but can be much more. The "much more" primarily related to other people using the software, and their experience and level of enjoyment with it. However, this idea of UX being separate from the actual software is indicative of an incomplete understanding of some aspects of SE.

Second, our students exhibited a mature understanding of the role of documentation in software development, recognizing that documentation, in most cases, will depend largely on the business's needs and what type of software project it is being developed. We see again that there is a sense of community and a recognition of their role as a software engineer being tied to the larger group of engineers in a company or enterprise.

Third, our students saw that requirements change in an agile environment, and requirements engineering is a tool for effective communication to help clear doubts and misunderstandings among developer and stakeholders. Students expressed that the role of requirements elicitation was to come to an agreement between the consumer (stakeholder) and development team of what the exact parameters of the software will be. Again, we see a recognition of software being engineered in the context of several people-stakeholders, end users, and development teams.

Next, our students understood the benefits of effective communication between all parties involved - development team, product manager, project manager, stakeholders, end users, and management. They also understood that when effective and continued communication is broken, the product suffers. Students also recognized that software quality depended heavily on reliable and steady communication between parties.

Finally, when asked how students would handle downtime, they almost unanimously expressed a wish to help other teammates, or take on more work from the existing sprint/product backlog. Some students also mentioned the importance of taking time to care for their mental health, recognizing the stress and mental strain that real world software development can entail.

Our analysis reveals that our students prioritized business needs, were curious, eager to take on more work as needed, with a strong work ethic, and strove hard to maintain good communication and transparency with their stakeholders and development team. These are in agreement with our previous study (Gopal, Cooper, & Bockmon, 2021) where we heard from industry partners on the advice they would give SE students to succeed in the industry, based on their interactions with and observations of students in a peer instruction (Mazur, 1997) based SE course. By varying the pedagogical approach to POGIL-like but keeping the content and instructor unchanged, and with similar prior academic preparation, our participants seemed to have gained some other things: Possibly due to the sustained collaboration and concept invention aspects of POGIL-like, our students developed a strong sense of where they belonged

in a software development team, in relation to stakeholders, and end users. They also got to apply their newly invented concepts in each POGIL-like session (E-I-A cycle) and had a more realistic grasp on the exhilarating and potentially demanding nature of individual and collective software development.

6.2. POGIL-like: its influence on student perceptions and student motivation

Literature shows that autonomy and self-driven inquiry has been showed to increase student motivation (Buchanan, Harlan, Bruce, & Edwards, 2016; Biggers et al., 2008). Student motivation is linked to the student perceived value or meaning in the academic work at hand. Student interest increases cognitive and affective outcomes, specifically student motivation (Ainley, Hidi, & Berndorff, 2002). "Student control of the learning process," not only influences academic achievement, but greatly increases student motivation (Mega, Ronconi, & De Beni, 2014).

Perhaps the largest difference in instructing students using traditional lecture vs POGIL-like is the autonomy afforded to students. To enable problem solving, critical thinking and reasoning skills, the D/C/V question pattern in POGIL-like creates fertile ground for inquiry based discovery: initial direction (D), convergence of knowledge from the directed discoveries (C) leading to co-construction and invention of concepts, and finally, divergent application of the newly constructed concepts in unfamiliar scenarios. In a topic like DevOps, for example, with a specific concept like continuous integration, there could be several ways of approaching the topic. With traditional lecture, the topic maybe taught with slides that all students receive. With POGIL-like, with the overarching guidance of the activities (with several D/C/V questions in E-I-A cycles) and models, each individual student explores the topic on their own. Each student invents the concept together with their team mates, co-constructing the ideas that thread together the complete concept, with the help of the model and activities. There is a rationale for every step in these activities, and students have to think about the "Why" in addition to the "How". Students have to then individually apply the newly invented concept - and this can be different for each student as well. We see that there is a high level of autonomy in each step of the E-I-A cycle, and a high level of engagement is required to complete each activity. The high level of autonomy in the POGIL-like paradigm allows students ways to think and function independently, as well as contribute freely within a group.

7. Threats to validity

Lincoln and Guba (Lincoln & Guba, 1985) indicate that validity in qualitative studies is expressed as respondent/participant bias, reactivity, and researcher bias. As for participant bias, there is always a possibility that our students responded to the questions based on what they thought was the right or acceptable answer instead of what they really felt. We took care to explain to participants that they received participation credit, not correctness credit. We also ensured that students knew that there were no right or wrong answers to the survey questions. The primary author of this paper acknowledges that her vast experience and background in the software engineering industry is likely to have influenced her interpretation of the the data with a marked bent towards industry relevant information. We attempted to lessen the effects of these aspects (Robson, 2002), we triangulated student response data with our own reflections and journals and audits.

Data saturation was achieved based on the guidelines by Creswell and Poth (Creswell & Poth, 2018). There is always a possibility that in spite of our systematic thematic analysis, other researchers may infer and construct different themes from our raw data. We are confident in our findings to be valuable and relevant within the context of our study because we triangulated our data from 24 student responses from a single cohort with the same instructor and instructional pedagogy with the same topics of instruction.

8. Conclusion and future work

In answering our research question, "Were undergraduate students able to learn from and build upon multiple relevant concepts to display a connected understanding of software engineering topics when instructed using a POGIL-like pedagogy?", we conclude that our students did indeed connect various software engineering topics in constructing their understanding, and displayed synthesis skills, nuance

and depth in explaining their impressions, when instructed using our POGIL-like pedagogy. Students showed analysis and synthesis skills beyond just basic knowledge where they could tie together topics like requirements engineering and testing, or communication with all aspects of software engineering. This leads us to believe that students were able to approach the higher layers of Bloom's taxonomy during their learning (Bloom, 1956). We think that this deeper understanding was fostered by the structured, process oriented approach in POGIL-like, specifically with the collaborative learning cycles utilizing the various types of Directed, Convergent and Divergent (D/C/V) questions the Explore-Invent-Apply (E-I-A cycle).

Through the structured process oriented POGIL-like approach, which places a heavy emphasis on collaboration and co-construction of knowledge, we find that students displayed a strong and connected understanding of the agile software development process and were able to place themselves in it. Their sense of identity within SE was revealed by their perceptions of SE topics, and we find that the inquiry based, constructivist learning approach through POGIL-like helped students relate themselves within the individual software developer context as well as the larger software engineering context involving the rest of the development team, product managers, project managers, stakeholders and end users. Previous literature shows that a positive self perception of one's ability was a key trait that helped retain students (Almulla, 2020).

POGIL-like assigns students into four distinct roles- manager, presenter, recorder and reflector. Our future work directions include understanding the impact of these specific POGIL-like roles in building students' confidence and motivation in SE, and whether other constructivist or collaborative pedagogies have similar effects on student affect. We also intend to compare our findings with a traditional lecture class with the same content and instructor, and delve deeper into how students' impressions varied between both approaches.

9. Acknowledgements

We would like to acknowledge and thank Dr. Justin Olmanson, Associate Professor, College of Education and Human Sciences at the University of Nebraska-Lincoln for his generous guidance on qualitative analysis techniques.

10. References

- Agosto, D. E., Gasson, S., & Atwood, M. (2008). Changing mental models of the it professions: A theoretical framework. *Journal of Information Technology Education: Research*, 7(1), 205–221.
- Ainley, M., Hidi, S., & Berndorff, D. (2002). Interest, learning, and the psychological processes that mediate their relationship. *Journal of Educational Psychology*, 94(3), 545.
- Almulla, M. A. (2020). The effectiveness of the project-based learning (PBL) approach as a way to engage students in learning. *Sage Open*, 10(3), 2158244020938702.
- Ames, C., & Archer, J. (1988). Achievement goals in the classroom: Students' learning strategies and motivation processes. *Journal of Educational Psychology*, 80(3), 260.
- Anfara, V., Brown, K., & Mangione, T. (2002). Qualitative analysis on stage: Making the research process more public. *Educational Researcher*, 31(7), 28–38.
- Bean, J. P., Eaton, S. B., et al. (2000). A psychological model of college student retention. *Reworking the student departure puzzle*, 1, 48–61.
- Ben-Ari, M. (1998). Constructivism in computer science education. In (Vol. 30, pp. 257–261). ACM New York, NY, USA.
- Biggers, M., Brauer, A., & Yilmaz, T. (2008). Student perceptions of computer science: a retention study comparing graduating seniors with cs leavers. *ACM SIGCSE Bulletin*, 40(1), 402–406.
- Bloom, B. (1956). *Taxonomy of educational objectives, handbook 1: Cognitive domain* (2nd edition Edition edition ed.). New York, NY, USA: Addison-Wesley Longman Ltd.
- Buchanan, S. M. C., Harlan, M. A., Bruce, C., & Edwards, S. (2016). Inquiry based learning models, information literacy, and student engagement: A literature review. *School Libraries Worldwide*, 22(2), 23–39.

- Creswell, J., & Poth, C. (2018). *Qualitative inquiry and research design: Choosing among five approaches* (4th ed.). Sage Publications, CA.
- CS-POGIL | DCV (Directed, Convergent, Divergent) Questions. (n.d.). Retrieved 2021-12-30, from [https://csPOGIL.org/DCV+\(Directed,+Convergent,+Divergent\)+Questions](https://csPOGIL.org/DCV+(Directed,+Convergent,+Divergent)+Questions)
- Dempsey, J., Snodgrass, R. T., Kishi, I., & Titcomb, A. (2015). The emerging role of self-perception in student intentions. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 108–113).
- Emerson, R. M., Fretz, R. I., & Shaw, L. L. (2011). *Writing ethnographic fieldnotes*. University of Chicago Press.
- Foley, D. E. (2002). Critical ethnography: The reflexive turn. *International Journal of Qualitative Studies in Education*, 15(4), 469–490.
- Gopal, B., & Cooper, S. (2022). POGIL-like Learning in Undergraduate Software Testing and DevOps - A Pilot Study. In *Proceedings of the 27th Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)* (p. Accepted).
- Gopal, B., Cooper, S., & Bockmon, R. (2021). Industry partners' reflections on undergraduate software engineering students: An exploratory pilot qualitative study. In *Proceedings of the 32nd Annual Psychology of Programming Interest Group Workshop(PPIG)*.
- Hsieh, H.-F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative Health Research*, 15(9), 1277–1288.
- Hsu, L. M., & Field, R. (2003). Interrater agreement measures: Comments on Kappan, Cohen's Kappa, Scott's π , and Aickin's α . *Understanding Statistics*, 2(3), 205–219.
- Ketelhut, D. J., & Schifter, C. C. (2011). Teachers and game-based learning: Improving understanding of how to increase efficacy of adoption. *Computers & Education*, 56(2), 539–546.
- Kussmaul, C. (2011). Process oriented guided inquiry learning for soft computing. In *International Conference on Advances in Computing and Communications* (pp. 533–542).
- Kussmaul, C., Mayfield, C., & Hu, H. (2017). Process oriented guided inquiry learning in computer science: The cs-pogil & introcs-pogil projects. In *ASEE Annual Conference and Exposition, Conference Proceedings* (pp. 1–7).
- Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic inquiry*. Sage Publications, CA.
- Mazur, E. (1997). *Peer instruction a user's manual*. Prentice Hall.
- Mega, C., Ronconi, L., & De Beni, R. (2014). What makes a good student? how emotions, self-regulated learning, and motivation contribute to academic achievement. *Journal of Educational Psychology*, 106(1), 121.
- Melnik, G., & Maurer, F. (2005, 06). A cross-program investigation of students' perceptions of agile methods. *Proceedings - 27th International Conference on Software Engineering, ICSE05*, 481–488.
- Peters, A.-K., & Pears, A. (2013). Engagement in computer science and it—what! a matter of identity? In *2013 Learning and Teaching in Computing and Engineering* (pp. 114–121).
- Qualitative Data Analysis tool - HyperResearch. (n.d.). Retrieved from <http://www.researchware.com/products/hyperresearch.html> (Last Accessed March 2021.)
- Robson, C. (2002). *Real world research: A resource for social scientists and practitioner-researchers*. Wiley-Blackwell.
- Souza, M., Moreira, R., & Figueiredo, E. (2019). Students perception on the use of project-based learning in software engineering education. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering* (pp. 537–546).
- Strauss, A., & Corbin, J. (2015). *Basics of qualitative research: techniques and procedures for developing*. Sage Publications, CA.
- Tinto, V. (1997). Classrooms as communities: Exploring the educational character of student persistence. *The Journal of Higher Education*, 68(6), 599–623.

On writing workshops for programming

Michael Nagle
mpnagle@gmail.com

Abstract

Over the past year, inspired by personal loss from COVID-19, I took a series of writing workshops in creative nonfiction, spanning nine months of time. In this paper, I am going to use those writing workshops as a jumping off point for a workshop of similar format focused on creative programming. The goal of this paper is to take insight from the form of writing workshops and use them in running a “writers workshop for programmers.”

1. Introduction

At the beginning of the COVID-19 pandemic in April 2020 I lost my dad to COVID. Because of mandates early in the pandemic forbidding me to be in the hospital with him, I talked my dad through his death over Skype instead of by his side.

His death caused me to quit my job leading product research at Coda, move out of the Bay Area, and travel for the next eighteen months, living primarily in Portland, OR.

While there, I signed up on a whim in Fall 2021 for a weekend writing workshop on “Grief and the Lyric Essay.” In the course I discovered Claudia Rankine’s “Citizen,” a lyric essay which explores race in modern America, from many angles: personal experiences, in the lives of sports celebrities, in the work of YouTubers, in intimate relationship.

It made me think: if Rankine can talk about her experience of race this vividly, then maybe I can talk about my own experience of grief.

2. The Writer’s Workshop

2.1 Workshop Format

I proceeded to sign up for a 10 week writing workshop, on Personal Essay and Memoir, again based in Portland. This was my first time in writing workshop.

The format was simple: every week we’d read 2-3 pieces, each about 10-15 pages, and 2 or 3 participants would submit a piece of up to 5,000 words. This workshop had 12 participants and 1 facilitator.

The idea was to take inspiration from the weekly readings - tailored to the class and what people were submitting - and use those to learn about the craft of writing.

Getting peer feedback and facilitator feedback (live in group discussion and then outside of class in the form of a feedback letter from each participant) quickly gave a creative heatmap - what worked, what didn’t, what needed more information, etc.

I found I began to be able to tell the specific grief story, and many more questions came up along the way (like: what made my father special to me, beyond the unusual circumstance of his death?)

I proceeded to sign up for a second 10-week workshop, and when that concluded, a third. These workshops were smaller: 9 participants and 1 facilitator, and instead of 30-40 pages of reading a

week, it was much lighter on reading (2-10 pages a week,) and featured a weekly writing prompt of 2 pages. The core format of 2-3 chances to workshop a long (up to 5,000 words) piece stayed the same.

2.2 Workshop Impact

Finding great writing in the first class felt like finding a hidden store of intellect that had been waiting for me my whole life, out there in the world, a resource hiding in plain sight. I was palpably inspired by the great writing I hadn't known existed prior to the class.

Writing for an audience got my own work moving. I found it much easier to write into an audience, to know feedback was coming quickly.

Over the nine months, what seemed a huge task - writing 5,000 words and sharing them with people basically new to me - became a more familiar, or routine one. The challenge didn't fade, but the comfort grew, and I found myself taking more risks over those nine months.

The workshops also fit the rest of my life - they were a commitment of about 8-12h a week, unlike a fulltime MFA writing program.

2.3 Writer's workshops for programming

I have since found myself yearning for a similar environment for creative uses of programming, either as participant or facilitator. What appeals to me is it is not industry focused (like a bootcamp or working in the tech industry) - which gives room for creative expression - and it isn't a fulltime model (like grad school or environments like the Recurse Center.)

I am deeply interested in the soul of programming, how computing plays a role in the accelerating global challenges of our time. How computing can be bent and warped to more novel and creative uses, and how those uses can get discovered in the first place (as opposed to continuing to implement computing as is currently practiced, i.e. a handful of monolithic companies and codebases dictating the vast majority of online interaction.)

2.4 A Case Study (Optometry Vision Task)

Before the pandemic, I made friends with my optometrist. I shared with my interest in human learning and development, and she in turn told me about some of her graduate work, thirty years prior. She's been part of a foundational study in vision and neuroplasticity, and had stories to tell of giving subjects vision tasks to see if their vision could improve. (Many subjects had substantial emotional responses to the tasks she was asked to administer, and she remembered that thirty years later.)

Intrigued, I tracked down the original study, and tried to replicate it with a friend. Working in a friend's artist studio, his neighbor saw what we were doing, and asked why we didn't just recreate the experiment in software instead of the original hardware, whose optics we were struggling with? A lightbulb went off, and we wrote a first draft in Processing.

On showing it to the optometrist, she became interested in trying the tool out with a vision rehabilitation client, I realized it was going to be a lot to ask someone who wasn't a programmer to install Programming and run a Processing sketch. I asked a second friend, who kindly implemented the code in the browser.

The pandemic came, my dad passed, and I forgot about the work. In remembering it a few months ago, I tried to make some changes to the code, only to discover it was all minified. When I asked if my friend if he had the original code, he apologetically told me he didn't.

I'd like to run a workshop where I could re-build a simple application like this over the course of a 10 week workshop.

2.5 Active Design Questions:

2.5.1 Genre:

Writing workshops organize themselves by genre. Fiction and non-fiction tend to be separate, and in non-fiction, lyric and braided essays can be distinguished from linear essay and memoir.

What genre of programming is right for a 10-week workshop? Right now I'm considering: tools for learning and human feedback, but I wonder if that's too narrow.

2.5.2 Scaffolding:

Programming takes a level of technical knowledge that writing really doesn't. I can't get stuck on my essay not compiling.

Part of what makes a writing workshop powerful is the easy transfer from accomplished works as well as peer works into one's own. It's easy to assimilate new words, ideas, or techniques once they're made conscious. Assimilating pieces of other software though, requires a barrier of engineering. Recurse Center, for example, bills itself as not for new programmers. Is there a necessary floor of experience people need to make a creative programming workshop successful? Is there scaffolding provided for specific tasks or techniques (for example working with audio or video, working with a 2d or 3d game engine) needed to create an environment where assimilating inspiration from professional and peer works happens easily?

2.5.3 Workshop Constraints

One useful constraint pattern in writing workshops is length of work. If writing was assigned weekly, it was 2 pages a week. On a 4-6 week rhythm, it was 15 pages per workshop session. The project of writing a longer form piece, like a book, naturally got broken up into 10-15 pages a month.

Are there similar constraints for creative workshop material? For example, does a single page web application enforce a similar constraint to "no more than 15 pages of writing"? A literal analog would be establishing a constraint in # of lines of code, though I am more drawn to constraints about the length and shape of the experience than it's implementation (i.e., what if a webcam API takes 5k lines of code?)

The constraint of "no user logins or databases" seems to speak against a class of program like the optometrist's example: where you want track someone's progress or lack thereof over a period of time. A softer constraint may be "no data storage more complicated than a simple table that could be expressed in a .csv."

Writing workshops providing an ambitious but achievable goal in 4-6 week timespans, and I wonder what that the analog is for a creative programming project.

2.5.4. Workshop Inspiration

Inspiration seems hugely important in motivating any creative medium.

Writing is naturally open source.

I expect good pieces of creative programming to be open source (or easily understandable from an inspector), and an experience that can be understood, at least at a surface level, in an evening's worth of time.

I think one piece of work to generating a good creative programming workshop structure will be to come up with the equivalent of a reading list to pull from, week over week.

Independent games and pieces of computational art (like Jason Rohrer's *Passage*), educational experiments (like Chaim Gingold's *Earth Primer*) easily provoke new ideas.

I think collecting a list of emotionally compelling demos and web experiences which will be an important piece of creating a workshop.

2.6 An Experiment

I want a creative programming workshop to exist like the workshops I took in creative non-fiction.

There are questions specific to programming: for example, how much technical scaffolding to provide participants (how to deploy an app, what if any UI framework to use, for example), how to invite an audience with good group cohesion (does “apps and games for feedback and learning” create too narrow a focus or too broad of one?) but in general, I am writing this up to try to generalize what I learned in nine months of writing workshop, and to start to see how I can create a similar experience for myself and others in the domain of computing.

Any ideas: great pieces of computing to take as inspiration, other workshop experiments to learn from, and any other feedback is welcomed. I believe there is something deep in the simplicity of the writing workshop format, and I’d like to try applying it to the medium of computing.

Mastery Learning and Productive Failure: Examining Constructivist Approaches to teach CS1

Cruz Izu, Daniel Ng, Amali Weerasinghe

School of Computer Science, The University of Adelaide
(cruz.izu,amali.weerasinghe)@adelaide.edu.au

Abstract. The struggles of novices taking introductory computer science courses to master basic constructs and develop an understanding of the notional machine continues to drive computer science education in the search of new pedagogical approaches. This work examines in depth two recent proposals: mastery learning and productive failure. Both approaches are grounded by constructivism, which should reduce the challenges that CS1 students face when learning to code. By exploring the concepts that drive these pedagogical approaches, this study aims to make constructivism more accessible to CS0/CS1 teachers. The two approaches illustrate and highlight key concepts that support constructive learning.

The main outcomes from this work are the concept maps generated for each pedagogical approach, along with descriptive tables of their concepts. Both approaches support constructive learning by utilising (1) adaptive instruction that aligns with the current constructed knowledge of students, and (2) the use of student's failure as key to identify knowledge gaps and improve learning.

Keywords: constructivism, active learning, failure, prior knowledge

1 Introduction

Core pedagogical concepts, such as scaffolding and learning trajectories, are used by tertiary teachers to structure their instruction (Anderson & Gegg-Harrison, 2013; Rich, Strickland, Binkowski, Moran, & Franklin, 2017; Wass & Golding, 2014). These approaches are not sufficient to teach computer science effectively, as shown by lower pass rates (mean pass rate of 67.7%) identified in the systematic review of 161 CS1 courses taught in 15 different countries Watson and Li (2014).

More importantly, many students that passed a CS1 course still struggle to "see the forest from the trees" (Lister, Simon, Thompson, Whalley, & Prasad, 2006) or even understand their own code (Lehtinen, Lukkarinen, & Haaranen, 2021). These documented struggles may explain the high attrition rates which in the past have been attributed to the idea that computing ability is innate (Hoda & Andreae, 2014; Robins, 2010), but may also be caused by poor pedagogy. Hence, further steps are needed to improve computer science education (CSE) at introductory levels. (Robins, 2010; Watson & Li, 2014).

A classic theory of learning, such as objectivism, might suggest that students passively gain knowledge from textbooks or lectures (Jonassen, 1991). Constructivism, a more modern theory of learning, suggests that this is not the case. Instead, it implies, that students learn by actively constructing and connecting knowledge recursively (Brooks & Brooks, 1993; Waite-Stupiansky, 1995).

Ben-Ari (2001) presented a theoretical analysis of computer science education (CSE) from a constructivist paradigm, focusing mostly on novice programmers (Ben-Ari, 2001). He suggests that some struggles students face in CS, such as building mental models and connecting concepts coherently, can be explained by constructivism. Robins (2010) describes in a similar vein the tightly integrated nature of the concepts comprising a programming language that creates an inherent structural bias in CS1 which drives students towards success, if they managed to quickly grasp early concepts, or failure if they don't.

Ben-Ari lists 3 pedagogical principles that support constructive learning (Ben-Ari, 2001):

1. A preference for **active learning** to enable the student to construct mental models.
2. Recognition of the importance of **pre-existing knowledge**.
3. The employment of the inevitable errors and misconceptions as a pedagogical device rather than as a symptom of **failure**.

Thus we want to explore in more depth recent approaches that are aligned with the three above principles so that teachers can support students in building correct mental models of programming plans and the notional machine. This work aims to examine and explore some pedagogical approaches that seemingly support constructive learning. After an initial review of the literature, we choose depth versus breadth and focused only on two approaches: Mastery learning (ML) have already been applied to CS1 courses and have demonstrated positive

results (McCane, Ott, Meek, & Robins, 2017; Gorp & Grissom, 2001); Productive failure (PF) is another interesting approach which has shown positive results in high school maths (Kapur, 2008) and seems to be a good fit in terms of constructive principles.

This paper revises these two pedagogical approaches, highlighting the core underlying concepts of each and utilizing concept maps to clearly convey the concepts and how they relate to each other. Whilst there are already a number of papers covering PF and ML, there is value in collating and revising their contributions, so we can draw out the core concepts of each approach. This can help teachers to better understand the underlying mechanics of these pedagogical approaches, making it easier for adoption in their classrooms. The analysis could also help to drive further research by suggesting different pedagogical mechanics for investigation.

The rest of the paper is organised as follows: after a literature review in section 2, a methodology section follows; section 4 presents the concept maps derived from the analysis and its associated tables. Section 5 discusses how each pedagogy supports constructive learning and section 6 provides the conclusions of this study.

2 Literature review

This section starts with a brief review of Constructivism before moving to the two selected approaches, ML and PF that apply constructivist principles. Then we will provide a brief description of each approach, summarising some of its applications, results recorded and why they are of interest to CS educators.

2.1 Constructivism

As the educational resources are ubiquitous and freely available online, the challenge for the current generation of students is to receive guidance to develop coherent mental structures that facilitates their application. Thus, active learning has been changed from being passive to active with many approaches being tried and evaluated in recent years, including problem-based learning (Greening, Kay, Kingston, & Crawford, 1996), team-based learning (Michaelsen, 2014), and peer instruction (Zingaro & Porter, 2014).

Constructivism provides explicit support to build coherent mental models by focusing on prior knowledge and also utilizing failure as a learning opportunity. However, there is no *explicit* mechanism in the active learning approaches cited above to build on pre-existing knowledge and using failure as an opportunity to learn. A large body of work in CS education has focused on identifying issues in prior knowledge by detecting and reporting misconceptions (Caceffo, Wolfman, Booth, & Azevedo, 2016; Kaczmarczyk, Petrick, East, & Herman, 2010). However, it is left to CS1 instructors the task of developing activities that correct or cross-examine those misconceptions.

As successful learning appears to require that the student reach an impasse (VanLehn, Siler, Murray, Yamauchi, & Baggett, 2003), failure provides that impasse that increases learning. Educators need to help students to cultivate a growth mindset (O'Rourke, Haimovitz, Ballweber, Dweck, & Popović, 2014), enabling students to understand that not being able to complete tasks or failing at tasks is a part of learning. Once students have identified the reason to fail, usually a knowledge gap, they are more receptive to instruction to reduce or eliminate that gap as it is more relevant to them. However, as failure has the potential to discourage some students, we should be aiming for small failures that provide a sense of progress or "being nearly there".

Some works support the idea of learning from errors and used a constructivist approach to teach a particular difficult topic. For example, Moreno, Sutinen, and Joy (2014) used conflictive animations of code to help students reflect and learn about operators, expressions and statements. Ginat and Shmalo (2013) used tasks that contains errors to explore OOP concepts. Although inspiring, such approaches are not transferable to teach other topics, limiting their use in the classrooms. However, both productive failure and mastery learning appear more suitable for wider use, as explained in the rest of this study.

2.2 Mastery Learning (ML)

The underlying premise of ML is simple, course content is broken down into sequential parts, where each part builds on from the prior part (Mercer, 1986). Students are required to "master" a part (e.g. achieve 90% on a diagnostic task) before moving on to the next (Mercer, 1986). Figure 1 illustrate this process: students must master *core 1* before starting to work on *core 2*. ML allows students to work at their own pace, helping to ensure that they have adequately learned each part before moving on to the next (Mercer, 1986). Extension work is provided for top students that master the topic first so that the whole cohort is able to move at the same time to the next topic.

The sequential structure of ML seemingly supports students in connecting concepts together and building effective mental models. This makes ML a particularly compelling candidate to use in CS1 as each program statement works in combination and relies on the knowledge of other statements. by providing either remedial work or extension work, all students benefit regardless of their performance.

ML has been applied to a variety of contexts, including high school mathematics (Zimmerman & K. Dibenedetto, 2008) and introductory programming at university (McCane et al., 2017). Both ML approaches were shown to

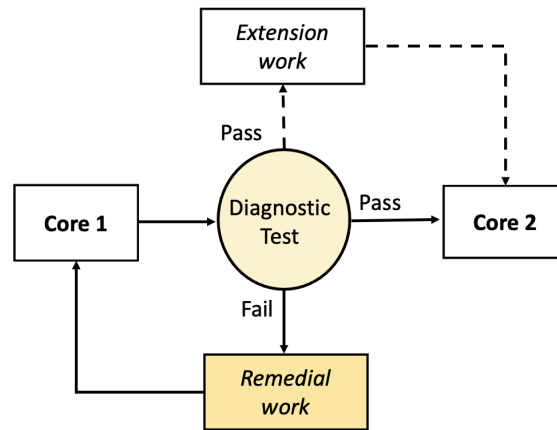


Fig. 1: Mastery learning visualised process from(Mercer, 1986)

increase student performance (McCane et al., 2017; Zimmerman & K. Dibeneditto, 2008). Research has also been shown mastery learning leads to stronger engagement from students and a greater desire to learn (McCane et al., 2017; Mercer, 1986; Zimmerman & K. Dibeneditto, 2008).

Mastery learning requires rethinking the whole course, thus it does not allow for small trials. However, ML has shown promising results and in particular it seems to be more effective in poorer performing students, when compared to high-achieving students (McCane et al., 2017). These traits make ML strong candidate for helping to improve CS1 education.

2.3 Productive Failure (PF)

Traditional instruction start with describing and demonstrating the application of a concept to students and then providing problems for students to work on.

PF **flips** the order by getting students to start with the problems first, before receiving instruction regarding the concept from the teacher. This is described in a PF alternative PS+I (problem solving plus instructions) as delayed instruction (Loibl, Roll, & Rummel, 2016). The essence of PF is that students will fail the complete the task given, as they haven't learnt the concept required to do so yet. That failure, caused by delayed instruction (Loibl et al., 2016), together with a reflective teaching of the concept afterwards is key to improved the learning of that new concept by students, as well as developing their cognitive skills (Kapur, 2008; Kapur & Bielaczyc, 2012)

PF has been applied to a range of educational levels, but primarily in the domain of high school mathematics and it demonstrated to improve the performance of students (Kapur, 2017; Kapur & Bielaczyc, 2012; Loibl et al., 2016; Westermann & Rummel, 2012). Figure 2 illustrate one of the main reasons behind PF's effectiveness: the attempt to problem-solve helps students to be aware of their knowledge gaps as described in Figure 2, as well as reflecting on the task at hand and its key features. Besides, PF helps students to construct their conceptual knowledge (Loibl et al., 2016) by providing just-in-time instruction after the gaps are identified, making that instruction more relevant to students. This increases student's engagement and awareness of the relationship between that new concept and the knowledge they have used in their failed attempt.

Similarly to ML, PF has been reported to be more effective in poorer performing students (Kapur & Bielaczyc, 2012), hence it appear to be a good candidate for teaching in a computer science context. Note PF is more flexible than ML in that we can choose a topic or a workshop to trial it out and gradually build up its application and our expertise using it. We are only aware of one work that reports the use of PF in an IT-related topic.

3 Methodology

The previous section has presented a brief literature review of ML and PF. Our task now is to examine them from the constructivist perspective. For each pedagogical approach explored in this work, a concept map and table of concepts is created. Each concept map redraws the figure from the literature (figs 1 and 2) in order to reflect the links of each approach to constructivism. The associated table aid in communicating how each concept applies or is shaped in each context.

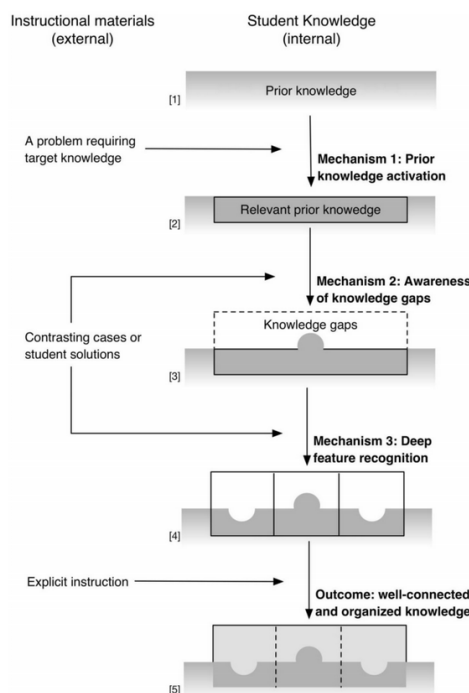


Fig. 2: Productive failure visualised process from(Westermann & Rummel, 2012)

3.1 Concept maps

Concept maps are great tools for expressing the key components of a topic and how they relate to each other. They comprise of concepts, regularities seen in the topic, and propositions, relationships between concepts (Novak & Cañas, 2006). Though a seemingly simple tool, the theoretical foundations of concept maps are rooted in psychology and they have proven to be effective (Novak & Cañas, 2006).

This work uses concept maps to explore the pedagogical approaches and help communicate their underlying concepts. Concepts maps are not only a good graphical representation of information, but the process of creating them is an effective research tool (Novak & Cañas, 2006). The use of concept maps helps to explore the underlying mechanics of the pedagogical approaches and relay the findings effectively. A concept map was made for each pedagogical approach, exploring the concepts of its application and how they are interrelated. The mapping process was broken down into three main steps:

Literature review We decided to prioritise depth versus breadth by focusing only on core or seminal papers.

Three to four papers for each pedagogical approach were selected and annotated. Key concepts were extracted by observing the applications between papers and noting the concepts/practices which were consistent between them. Once concepts were listed, they were reexamined to consider which would be better represented as a relationship between concepts.

Individual concept maps Once concepts and relationships were identified, preliminary concept maps were sketched. How concepts fitted into the map, how concepts related to each other and even the concepts themselves were reviewed in an iterative process by the first two authors until they captured the essence of ML(PF).

Concept map revision: Finally, the two maps were compared to find commonalities: looking at the concepts across both maps and using the same terminologies and structures where possible. At this stage, it was also recognized that the cross-links between concepts, rather than the hierarchical links, were more important to demonstrate the mechanics of each pedagogical approach. Because of this, the final maps were redesigned to be non-hierarchical.

3.2 Tables of concepts

In the last step, we identified the need to provide detailed definitions to complement the concept maps. Thus, the definitions were collated in tables to accompany each concept map. Tables are an efficient way of indexing information and relaying it to readers. They could help readers to quickly understand the contexts where each pedagogical approach can be applied and the concepts included in its implementation.

To keep them consistent, during the research a generalised structure for the tables was created. Each table has a header depicting the context for which the pedagogical approach can be applied and concepts relating to the overall structure of the approach. In our context, the readers would be educators that may consider integrating an approach into their classroom. Thus, we aim for a practical description that groups concepts by their practical roles such as teacher input or student tasks.

The concepts in the concept maps can be considered to be in one of four areas:

Knowledge Concepts related to the knowledge students have, require or lack.

Teacher input Concepts that describe teacher actions to aid students to complete or learn a task.

Student tasks Concepts that describe hands-on tasks given to students.

Response to student results Concepts that pertain to the results of students.

4 Results

4.1 Mastery Learning

Mastery learning considers that students need to connect existing knowledge with the new knowledge being taught in order to adequately learn a concept. To support this model, content is broken into subsequent components. The sequence of components makes it so that the new knowledge acquired in one component becomes part of the existing knowledge required to complete the next component.

Instruction for a component teaches new content that builds on from the last component. As the overarching structure for content is sequential, the instruction implicitly implies the need for students to utilize the existing knowledge gained from the last component.

The prime goal of the task/diagnostic in mastery learning, is to verify that students have adequately learned the component. Test failure serves as an opportunity for students to see their knowledge gaps and take additional adaptive instruction. The instruction is specifically catered to the areas where the student struggled in the diagnostic test. Students are able to re-attempt the diagnostic indefinitely until they pass. Once they succeed in the diagnostic, they are deemed to have "mastered" the component and are able to move to the next component.

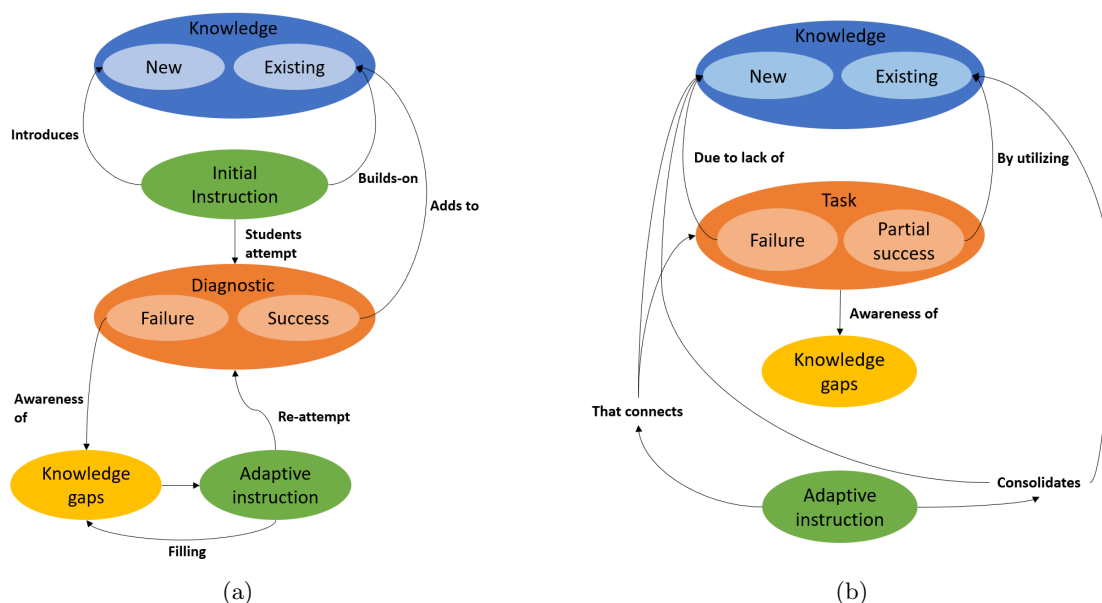


Fig. 3: Concept maps of (a) mastery learning and (b) productive failure

4.2 Productive Failure

Productive failure suggests that students failing, by attempting a task before receiving instruction, can improve student learning. To complete that first task, students require using existing knowledge in tandem with new knowledge. When they attempt the task, the absence of new knowledge encourages students to exhaust their

Table 1: Concepts - Mastery Learning (ML)

Structure: Course content needs to be broken down into subsequent components. Each component taught requires students to have an adequate understanding of the component taught before it.		
Area	Concept	Description
Knowledge	New knowledge	The new content that is being taught to students.
	Existing knowledge	Knowledge that students are assumed to already possess. This is knowledge acquired in previous components.
	Knowledge gaps	Knowledge linked to observed failure.
Teacher input	Initial instruction	Instruction phase focuses on teaching new content and links it (implicitly or explicitly) with the content taught in the previous component.
	Adaptive instruction	Students are individually given further work that focuses on the areas they struggled with in the diagnostic. This is self-administered, but materials may be selected by the teacher.
Student tasks	Diagnostics	Students are required to pass a diagnostic or assessment task for a component before being able to move onto the next component. If students fail the diagnostic, they are able to reattempt the diagnostic an indefinite amount of times until they pass.
Student results	Success	Completion of the diagnostic task is considered to reflect that a student has “mastered” that component. Mastery means that students have adequately understood the concepts taught in a component and they have the required knowledge to move to the next component.
	Failure	Failing a diagnostic stop progression but this is not seen as a negative outcome, but rather an opportunity for students to better learn the component being taught. Students are able to become aware of their knowledge gaps (reflected by the errors made in the diagnostic), and the teachers can adapt their instruction to fill those gaps.

Table 2: Concepts - Productive Failure

Structure: productive failure applies to a single session or lesson linking two concepts. A key aspect of productive failure is that the task phase comes <i>before</i> the instruction phase.		
Area	Concept	Description
Knowledge	New knowledge	The new content that is being taught to students and they expect to acquire.
	Existing knowledge	Knowledge students are assumed to already possess, from previous instruction or experience.
	Knowledge gaps	Conceptual knowledge linked to observed failure. It may indicate missing knowledge (new or existing) or misconceptions
Teacher input	Adaptive instruction	Teachers should connect the partial successes of students to the canonical solution being taught. Thus, instruction is related to the students’ results. This is also an opportunity for teachers to fill in any other knowledge gaps identified during the task phase.
Student tasks	Task before instruction	The task given to students covers in some degree the new content being taught. Students attempt the task before receiving instruction so they are expected to make some progress or at least explore some ideas.
Student results	Failure	Most students will fail the task given, due to not possessing the new knowledge required. Failure helps students to become aware of this gap in their knowledge and prepares them to learn the content that maps to it.
	Partial success	Students are expected to only partly succeed in completing the task. A partial success reflects that students have been able to utilize some of their existing knowledge in order to get part of the solution. How close students get to a full solution helps teachers to see what gaps they have in their existing knowledge.

existing knowledge as much as they can to produce partial solutions. Ultimately, they are expected to fail. The failure reveals, to students, gaps in their overall knowledge (the lack of new knowledge) and the partial success reflects that they have utilized their existing knowledge. The level of partial success reveals to the teacher any knowledge gaps students might have in their existing knowledge.

Instruction follows the task and is adaptive to the results of students. A key mechanic, within productive failure, is for the teacher to connect the new knowledge being taught, with the partial solutions students have generated. This connection is critical for two reasons: (1) it is the opportunity for the teacher to go over what existing knowledge students should be using, addressing any gaps students might have and (2) to help students connect their existing knowledge with the new knowledge, consolidating the two. Having students be self-aware of their gaps helps this consolidation.

5 Discussion

The discussion of this report is guided by three the pedagogical concepts listed by Ben-Ari (Ben-Ari, 2001), that support constructive learning. Each section discusses one of the pedagogical concepts and how the concepts of each pedagogical approach exhibit it.

5.1 Active learning to facilitate mental model construction

Both pedagogical approaches are amenable to active learning as we are to explore next.

ML leaves the instructor more room to choose the way the initial instruction is given to students, so it could be passive or active to start with.

After the diagnostic task is completed, adaptive instruction (in the way of teaching materials such as online content, podcast, quizzes etc) is individually selected for each student based on the knowledge gaps identified in that diagnostic. Whilst teachers may select the learning resources, students are expected to go over them by themselves, so it can be considered as a self-learning process. Mastery learning acknowledges that every student progress at their own pace, hence it make sense to be self paced. Teachers support students in that process, but they do not provide traditional active learning activities to the class.

Productive failure promotes active learning by beginning with a task phase that immediately engages students with the learning process, rather than starting with a passive instruction phase. Student engagement continues into the instruction phase. The adaptive instruction given by the teacher in the instruction phase is collaborative, connecting the canonical answer with the solutions of students. In other words, instead of showing them the correct solution or fixing their errors, the teacher guides students through a discussion of the part required to make each students solution complete. The collaborative approach allows to support the diversity of student solutions by exploring them in the class and building from them. This is more scalable compared to one-to-one tuition.

5.2 Utilizing pre-existing knowledge

Most instruction given by teachers builds on previous knowledge that students are *implicitly* assumed to possess. For example, when we teach to manipulate arrays we expect students to have learnt about iteration, because we have covered iteration the previous week. Approaches that support constructive learning, *explicitly* recognize how concepts build on top of previous knowledge and support students to connect the two.

Pre-existing knowledge is embedded into the structure of mastery learning. Fundamentally, mastery learning recognises that students need to learn some concepts before being able to adequately learn others. Moving on to the next related concept without a correct grasp of the current concept can be detrimental to their learning.

Hence, the sequential structure of ML ensures that concepts are learnt in an order that satisfy their required knowledge. ML also utilizes diagnostic tasks to ensure that students have properly learnt a given concept before being able to move on to the next. The sequential structure encourages students to apply what they've just learnt in previous components to the new component. There is also some guarantee they have mastered the required knowledge, although they may have forgotten some.

Note ML does not explicitly link the particular prior knowledge with the concepts currently being taught. This explicit connection may be indicated by the teacher under direct instruction.

In productive failure, the task phase is designed for students to both identify and apply their required prior knowledge to the task. Student should be encouraged to apply prior knowledge in order to come up with naive answers and partial solutions. As students have not been taught yet the additional concept or insight required to complete the task in full, they may identify the knowledge gaps. This approach would only work when students have adopted a growth mindset. More importantly, applying their prior knowledge to the task before receiving instruction helps students to discover how their existing knowledge fits with the new concepts being taught.

This connection between prior and new knowledge is reinforced when the teacher connect the partial solutions of students to the canonical answer.

The collaborative phase allows peers to share the ways they applied prior knowledge. As teachers connect the canonical answers to the partial solutions of students, they can highlight examples of students correctly applying their existing knowledge. Some students may have a sound knowledge of a prior concept but they were not aware it was relevant to the task. This is what Perkins called *inert* knowledge (Perkins, Martin, & Educational Technology Center, 1985). By looking at peers' examples of using that knowledge, it stops being inert as it gets grounded into the canonical answer.

This process helps to ensure that students correctly construct knowledge by connecting the two, as opposed to constructing an incorrect knowledge model. By highlighting what was correct about student solutions, or alternatively demonstrating how students should have applied their prior knowledge, the teacher can address any problems students have. In other words, this phase may identify novices' misconceptions that can be cleared to facilitate learning.

5.3 Utilizing failure

Failure is many times seems as a negative outcome because it reduces motivation. Thus, it is common to provide scaffold activities which minimize the chance of failure. However, Bjork remind us that "learning is different to performance" (Bjork & Bjork, 2011), and that in reducing failure we may not increase learning.

Failure is not always negative provided we can recover and learn from it. In constructivism, failure is seeing as an opportunity. In both ML and PF failure is utilized to reveal knowledge gaps in students, which are then used to guide adaptive instruction or are utilized to help students construct new knowledge. Regardless of whether a student succeeds or fails a task, traditional pedagogical approaches simply moves all students on to the next topic. Mastery learning doesn't expect that all students will immediately "master" each concept, and plans for it accordingly. Students failure is provides the teacher with data that indicates any misconceptions or knowledge gaps that students have. Once identified, the appropriate adaptive instruction can be provided to students to strengthen their learning.

Utilizing the failure of students is central to productive failure. Students are intentionally given a task which they are expected to fail (or at least to only make partial progress). The failure helps students to self-identify gaps in their overall knowledge, these gaps reflecting the new content that they need to learn. Teachers might already aware of these gaps, but enabling students to become aware helps them to construct conceptual knowledge. Awareness of knowledge gaps helps students to realise a need to learn new concepts, where the new concepts connect with their existing knowledge and prepares them to learn the new concepts. The extent to which students fail the initial task, and how thorough their partial solutions are, can reveal to teachers what gaps *this* group of students have in their existing knowledge. Teachers should supplement those gaps in the subsequent instruction. Teachers are further able to reinforce the connections by explicitly linking the partial solutions of students (reflecting that previous knowledge) to the canonical answer. Thus students can see students how their existing knowledge connects to the new concepts being taught.

In short, productive failure front-loads failure to happen early on in the class room, so that it can be utilized to focus the instruction and make it relevant. Experienced teachers will try to cover knowledge gaps they have seen in previous cohorts. Front-loading failure will aid teachers to better manage the gaps students have in their prior knowledge.

Other approaches are also using errors for students to learn. For example, Griffin has provided code with carefully designed bugs as a learning task (Griffin, 2019); this recent study shows that student can learn as much from debugging that code as from writing similar code from scratch. Ginat used task that contained key conceptual errors to teach OOP concepts such as the role of constructs and the scope of variables (Ginat & Shmalo, 2013). He followed constructivist principles and implement a collaborative approach in the classroom to discuss each error. In contrast to those strategies, in either ML or PF, the errors are produced by the students instead of inserted by the teacher. Thus, they support better the constructivist principles by forcing students to examine and correct their own faulty mental models. Additionally, the collaborative phase of PF allows students to share their mistakes and for the teacher to explain how to recover from a range of mistakes. This can be useful for students to be aware of potential pitfalls in similar problems.

6 Conclusion

We are looking for new ways for students to learn, particularly in an era of information overload, in which online content is ubiquitous and in the view of naive students any problem is only one google query away from being resolved. Undergraduate learning has moved in the last 10 years from being passive to becoming active with many new approaches tried and evaluated. In this context, the role of the teacher is to help students not to memorise or locate that content but to develop a coherent mental structure that facilitates its application.

From the constructivist perspective, active learning is enhanced by both building on prior knowledge and using failure as an opportunity to learn. In this study we have reviewed two approaches that support this view. Both pedagogical approaches relied on prior knowledge and use failure as an opportunity to learn. Mastery learning supports constructive learning by sequencing concepts, forcing mastery before moving to the next one, and

providing additional learning resources to help reaching that mastery level at their own pace. As sequential and self-paced progress is a core concept of mastery learning, the approach can't be applied to a single session/topic. Instead, mastery learning is suited for structuring a course, or teaching a series of connected topics.

Productive failure utilizes failure to help students connect their existing knowledge with the concepts being taught. The approach can be applied to an individual module and is especially applicable to teaching concepts which are interrelated as is the case with programming constructs. Productive failure key strength is to help students connect their existing knowledge with the new concepts being taught. Specifically, productive failure is best suited towards single sessions that introduce concepts which build on from previous instruction. In contrast, the approach would be less effective when teaching a new concept which has weak links to previous concepts. For students that have a fixed mindset, PF may be challenging to start with, and using this approach multiple times will help them to understand the benefits of desirable difficulties (Bjork & Bjork, 2011) and increase their resilience. Note that we could embed a PF session inside a ML module, for example to replace the initial instruction, so the two approaches could be used in combination.

The analysis and creation of concept maps and associated tables for each approach has highlighted the concepts that drive mastery learning and productive failure. Being aware of the key concepts of each approach, and the contexts that they can be used in, can help CS instructors apply constructivism to improve learning at CS1 level. Both ML and PF provide strategies to scale constructive learning from one-to-one or pairs to small and medium class sizes. It is a further challenge to make these approaches scale to large classes. Further real-life experience and analysis is required for both ML and PF to explore mechanisms that could facilitate their usage under different teaching environments.

There are two open lines of research from this study: (1) extend the use of productive failure to our CS0/CS1 with emphasis on implementing and providing guidelines for its collaborative approach, (2) use the concepts highlighted in this work to understand in depth how each pedagogy helps students construct knowledge.

References

- Anderson, N., & Gegg-Harrison, T. (2013). Learning computer science in the "comfort zone of proximal development". In *Proceeding of the 44th acm technical symposium on computer science education* (pp. 495–500). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2445196.2445344> doi: 10.1145/2445196.2445344
- Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1), 45–73. Retrieved from <https://www.learntechlib.org/p/8505>
- Bjork, E., & Bjork, R. (2011, 01). Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the Real World: Essays Illustrating Fundamental Contributions to Society*, 2, 56–64.
- Brooks, J. G., & Brooks, M. G. (1993). *In search of understanding: The case for constructivist classrooms*. Alexandria: Association for Supervision and Curriculum Development.
- Caceffo, R., Wolfman, S., Booth, K. S., & Azevedo, R. (2016). Developing a computer science concept inventory for introductory programming. In *Proceedings of the 47th acm technical symposium on computing science education* (pp. 364–369). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2839509.2844559> doi: 10.1145/2839509.2844559
- Ginat, D., & Shmalo, R. (2013). Constructive use of errors in teaching cs1. In *Proceeding of the 44th acm technical symposium on computer science education* (pp. 353–358). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2445196.2445300> doi: 10.1145/2445196.2445300
- Gorp, M. J. V., & Grissom, S. (2001). An empirical evaluation of using constructive classroom activities to teach introductory programming. *Computer Science Education*, 11(3), 247–260. Retrieved from <https://doi.org/10.1076/csed.11.3.247.3837> doi: 10.1076/csed.11.3.247.3837
- Greening, T., Kay, J., Kingston, J. H., & Crawford, K. (1996). Problem-based learning of first year computer science. In *Proceedings of the 1st australasian conference on computer science education* (pp. 13–18). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/369585.369588> doi: 10.1145/369585.369588

- Griffin, J. M. (2019). Designing intentional bugs for learning. In *Proceedings of the 1st uk & ireland computing education research conference* (pp. 5:1–5:7). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/3351287.3351289> doi: 10.1145/3351287.3351289
- Hoda, R., & Andreae, P. (2014). It's not them, it's us! why computer science fails to impress many first years. In *Proceedings of the sixteenth australasian computing education conference - volume 148* (pp. 159–162). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from <http://dl.acm.org/citation.cfm?id=2667490.2667509>
- Jonassen, D. H. (1991, Sep 01). Objectivism versus constructivism: Do we need a new philosophical paradigm? *Educational Technology Research and Development*, 39(3), 5–14. Retrieved from <https://doi.org/10.1007/BF02296434> doi: 10.1007/BF02296434
- Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st acm technical symposium on computer science education* (pp. 107–111). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1734263.1734299> doi: 10.1145/1734263.1734299
- Kapur, M. (2008). Productive failure. *Cognition and Instruction*, 26(3), 379–424. Retrieved from <https://doi.org/10.1080/07370000802212669> doi: 10.1080/07370000802212669
- Kapur, M. (2017, 11). Examining the preparatory effects of problem generation and solution generation on learning from instruction. *Instructional Science*, 46. doi: 10.1007/s11251-017-9435-z
- Kapur, M., & Bielaczyc, K. (2012). Designing for productive failure. *Journal of the Learning Sciences*, 21(1), 45–83. Retrieved from <https://doi.org/10.1080/10508406.2011.591717> doi: 10.1080/10508406.2011.591717
- Lehtinen, T., Lukkarinen, A., & Haaranen, L. (2021). Students struggle to explain their own program code. In *Proceedings of the 26th acm conference on innovation and technology in computer science education v. 1* (p. 206–212). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3430665.3456322> doi: 10.1145/3430665.3456322
- Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the solo taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 118–122). New York, USA: ACM. doi: 10.1145/1140124.1140157
- Loibl, K., Roll, I., & Rummel, N. (2016, 07). Towards a theory of when and how problem solving followed by instruction supports learning. *Educational Psychology Review*, 29(4), 693–715. doi: 10.1007/s10648-016-9379-x
- McCane, B., Ott, C., Meek, N., & Robins, A. (2017). Mastery learning in introductory programming. In *Proceedings of the nineteenth australasian computing education conference* (pp. 1–10). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/3013499.3013501> doi: 10.1145/3013499.3013501
- Mercer, D. (1986). Mastery learning. *British Journal of In-Service Education*, 12(2), 115–118. Retrieved from <https://doi.org/10.1080/0305763860120212> doi: 10.1080/0305763860120212
- Michaelsen, N. M. C. H., Larry K.; Davidson. (2014). Team-based learning practices and principles in comparison with cooperative learning and problem-based learning. *Journal on Excellence in College Teaching*, 25(3&4), 57–84.
- Moreno, A., Sutinen, E., & Joy, M. (2014). Defining and evaluating conflictive animations for programming education: The case of jeliot conan. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 629–634). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2538862.2538888> doi: 10.1145/2538862.2538888
- Novak, J. D., & Cañas, A. J. (2006). *The theory underlying concept maps and how to con-*

- struct and use them* (research report No. 2006-01 Rev 2008-01). Florida Institute for Human and Machine Cognition. Retrieved from <http://cmap.ihmc.us/Publications/ResearchPapers/TheoryCmaps/TheoryUnderlyingConceptMaps.htm>
- O'Rourke, E., Haimovitz, K., Ballweber, C., Dweck, C., & Popović, Z. (2014). Brain points: A growth mindset incentive structure boosts persistence in an educational game. In *Proceedings of the 32nd annual acm conference on human factors in computing systems* (pp. 3339–3348). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2556288.2557157> doi: 10.1145/2556288.2557157
- Perkins, D., Martin, F., & Educational Technology Center, M., Cambridge. (1985). *Fragile knowledge and neglected strategies in novice programmers. ir85-22 [microform] / david perkins and fay martin* [Book, Microform, Online]. Distributed by ERIC Clearinghouse [Washington, D.C.]. Retrieved from <http://www.eric.ed.gov/contentdelivery/servlet/ERICServlet?accno=ED295618>
- Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., & Franklin, D. (2017). K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In *Proceedings of the 2017 acm conference on international computing education research* (pp. 182–190). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/3105726.3106166> doi: 10.1145/3105726.3106166
- Robins, A. (2010, 04). Learning edge momentum: A new account of outcomes in cs1. *Computer Science Education*, 20, 37-71. doi: 10.1080/08993401003612167
- VanLehn, K., Siler, S., Murray, C., Yamauchi, T., & Baggett, W. B. (2003). Why do only some events cause learning during human tutoring? *Cognition and Instruction*, 21(3), 209-249. Retrieved from https://doi.org/10.1207/S1532690XCI2103_01 doi: 10.1207/S1532690XCI2103_01
- Waite-Stupiansky, S. (1995). The road to constructivist classrooms. *The Educational Forum*, 59(1), 98-100. Retrieved from <https://doi.org/10.1080/00131729409336369> doi: 10.1080/00131729409336369
- Wass, R., & Golding, C. (2014). Sharpening a tool for teaching: the zone of proximal development. *Teaching in Higher Education*, 19(6), 671-684. Retrieved from <https://doi.org/10.1080/13562517.2014.901958> doi: 10.1080/13562517.2014.901958
- Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on innovation & technology in computer science education* (pp. 39–44). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2591708.2591749> doi: 10.1145/2591708.2591749
- Westermann, K., & Rummel, N. (2012, 07). Delaying instruction: Evidence from a study in a university relearning setting. *Instructional Science*, 40. doi: 10.1007/s11251-012-9207-8
- Zimmerman, B., & K. Dibenedetto, M. (2008, 03). Mastery learning and assessment: Implications for students and teachers in an era of high-stakes testing. *Psychology in the Schools*, 45, 206 - 216. doi: 10.1002/pits.20291
- Zingaro, D., & Porter, L. (2014). Peer instruction in computing: The value of instructor intervention. *Computers & Education*, 71, 87 - 96. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0360131513002777> doi: <https://doi.org/10.1016/j.compedu.2013.09.015>

The construction of knowledge about programs

Federico Gómez

Instituto de Computación -
Facultad de Ingeniería
Universidad de la República
fgfrois@fing.edu.uy

Sylvia da Rosa

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
darosa@fing.edu.uy

Abstract

This paper presents an empirical study that uses the linear search problem to investigate the process of construction of knowledge about programs.

The study was carried out in an introductory programming course of a Computer Engineering Undergraduate Program, using two imperative programming languages (*Pascal* and *C++*) in two different groups of students.

Building on the theory of Genetic Epistemology of Jean Piaget and his understanding of the construction of scientific knowledge, the study applies Piaget's principles of a triad of *intra*, *inter* and *trans* stages, and the general law of cognition, in order to analyse the passage from the *intra* stage into the *inter* and *trans* stages. The passage involves the actions students undertake as they design an algorithm to solve the proposed problem, and the reflections needed for them to implement and execute a program on a computer.

The study describes several steps in which students carried out different activities. In order to encourage them to conceptualise and formalise their solutions, individual interviews were conducted with thirteen students, of approximately an hour each. The paper includes selected excerpts of students' responses, our analysis of the results, and some conclusions and future work.

1. Introduction

The study presented in this paper applies principles of Jean Piaget's theory of Genetic Epistemology and his understanding of the construction of scientific knowledge, briefly described in Section 2. The study starts from the resolution of a concrete instance of the linear search problem in which the participants are asked to look for a specific number in a row of numbered cards (*instrumental* knowledge). Subsequently, clinical interviews (similar to those carried out by Piaget) are conducted to help students to explain, in natural language, the method used and the reasons for success in solving the problem (*conceptual* knowledge), turning back to actions if necessary. The study ends with the writing, compilation and execution of a program that searches for a given value in an array of integers (linear search general problem), having to also explain how and why it works when executed on a computer (*formal* knowledge), both at the level of the program's source code (*the textual part* of the program) and its execution on a machine (*the executable part* of the program). According to the theory, in this dialectic interaction between the stages, the students construct conceptual and formal knowledge about the program from their instrumental knowledge of the problem instance (see Section 2).

Multiple investigations were carried out over many years of research by the Computer Science Education Research group of the InCo (Computing Institute, Faculty of Engineering, University of the Republic, Uruguay). The study presented here takes results from previous investigations in which the construction of knowledge about algorithms and data structures was analysed (da Rosa, 2010; da Rosa & Chmiel, 2012; da Rosa, 2015). The contribution of colleagues from Philosophy of Computer Science in the definition of the dual nature of programs as texts and as executable objects was of great importance in the research about the construction of knowledge about programs (da Rosa, 2016; da Rosa, S. & Chmiel, A. & Gómez, F., 2016). In line with this notion, a thorough understanding of the concept of program implies the construction of knowledge on two aspects: the symbolic part of the program (*textual* part) and the physical part (*executable* part) and of the relationship between them (da Rosa & Aguirre, 2018; da Rosa,

2018). The main goal of this study is to investigate the construction of knowledge about programs as dual objects.

The rest of this paper is organised as follows: in Section 2, Piaget's theory is introduced as a theoretical framework. Section 3 presents the design of the empirical study on linear search. Section 4 describes its implementation and includes excerpts from interviews with its participants. Finally, in Section 5, the conclusions of the work are presented and some lines of future work are proposed.

2. Theoretical framework

Piaget's theory offers a model for explaining the construction of knowledge that can be used in all domains and at all levels of development. The central points of Piaget's theory - Genetic Epistemology - have been to study the construction of knowledge as a process and to explain how the transition is made from a lower level of knowledge to a level that is judged to be higher (Piaget, 1977).

The supporting information comes mainly from two sources: first, from empirical studies of the construction of knowledge by subjects from birth to adolescence (giving rise to Piaget's genetic psychology), (Gréco, Matalon, Inhelder, & Piaget, 1963; Piaget, 1978, 1975, 1964, 1974), and second, from a critical analysis of the history of sciences, elaborated by Piaget and Garcia to investigate the origin and development of scientific ideas, concepts and theories. In (Piaget & Garcia, 1980) the authors present a synthesis of Piaget's epistemological theory and a new perspective on his explanations about knowledge construction.

The main idea of their synthesis consists in establishing certain parallels between general mechanisms leading from one form of knowledge to another - both in psycho-genesis and in the historical evolution of ideas and theories - where the most important notion of these mechanisms is the triad of stages, called by the authors the *intra*, *inter* and *trans* stages. The triad explains the process of knowledge construction by means of the passage from a first stage focused on isolated objects or elements (*intra* stage), to another that takes into account the relationships between objects and their transformations (*inter* stage), leading to the construction of a "système d'ensemble", that is, general structures involving both generalised elements and their transformations (*trans* stage), integrating the constructions of the previous stages as particular cases.

2.1. The general law of cognition

In Piaget's theory human knowledge is considered essentially active, that is, knowing means acting on objects and reality, and constructing a system of transformations that can be carried out on or with them (Piaget, 1977). The more general problem of the whole epistemic development lies in determining the role of experience and operational structures of the individual in the development of knowledge, and in examining the instruments by which knowledge has been constructed before their formalisation. This problem was deeply studied by Piaget in his experiments about genetic psychology. From these, he formulated a general law of cognition (Piaget, 1964, 1974) governing the relationship between know-how and conceptualisation, generated in the interaction between the subject and the objects that she/he has to deal with to solve problems or perform tasks. It is a dialectic relationship, in which sometimes the action guides the thought, and sometimes the thought guides the actions.

Piaget represented the general law of cognition by the following diagram:

$$C \leftarrow P \rightarrow C'$$

where P represents the periphery, that is to say, the more immediate and exterior reaction of the subject confronting the objects to solve a problem or perform a task. This reaction is associated to pursuing a goal and achieving results, without awareness neither of the actions nor of the reasons for success or failure. The arrows represent the internal mechanism of the thinking process, by which the subject becomes aware of the coordination of her/his actions (C in the diagram), the changes that these impose to objects, as well as of their intrinsic properties (C' in the diagram). The process of the grasp of consciousness described by this law constitutes a first step towards the construction of concepts.

In previous investigations (as cited above) the participants are asked to perform tasks such as games, ordering objects, searches, etc. in which they subconsciously apply instances of algorithmic methods (*instrumental* knowledge). They are later induced to reflect about the method employed to solve the problem and the reasons for their success (or failure), and to write down their descriptions in natural language. This constitutes evidence of construction of *conceptual* knowledge of algorithms and data structures, according to the general law of cognition. However, in the case that the object on which knowledge is to be constructed is a *program*, some challenges appear, which are inherent to the relevance of the *machine* that executes it, not the *person* who solves the problem. We found the necessity of developing an extension of Piaget's general law of cognition as we identified the need to describe cases where the subject must instruct actions to a computer (da Rosa & Aguirre, 2018). The thought processes and methods involved in such cases differ from those in which the subject performs the actions her/himself. The main objective of the work presented here is to deepen the study of these processes.

2.2. The extended law of cognition

Seymour Papert states in (Papert, 1980) page 28, referring to the programming of a turtle automata, "*Programming the turtle starts by making one reflect on how one does oneself what one would like the turtle to do*". In other words: "programming an automata that solves a problem, starts by making the student reflect on how she/he does themselves what she/he would like the automata to do".

In order to program an automata to solve a problem, the learners have to establish a causal relationship between the algorithm (she/he acting on objects), and the execution of the program (the computer acting on states). Not only do they have to be able to write the algorithm (the *text*), but also they have to be able to understand the conditions that make the *computer* run the program. The generalisation of Papert's words above can be described as: programming an automata starts by making one reflect on

how one does oneself
what one would like the automata to do

The causal relationship between both rows is the key of the knowledge of *a machine executing a program*. By way of analogy with Piaget's law we describe this relationship in the following diagram

$$\begin{array}{c} \underbrace{C \leftarrow P \rightarrow C'} \\ newC \leftarrow newP \rightarrow newC' \end{array}$$

The conceptual knowledge about the algorithm constitutes what is called *newP* (new periphery) in the diagram. The construction of knowledge about the program is explained by the transitions to new centers (*newC* and *newC'*) which represent awareness of what happens inside the computer. Respectively, the conceptualisation of how the computer executes the instructions of the program (*newC*) and the changes that they impose on the data structures used in it (*newC'*). The diagram describes an extension of the law of cognition to encompass not only algorithmic thinking (first row) but also computational thinking (second row). The relationship between them is indicated with the brace between both lines.

Piaget identified that the construction of knowledge of methods (algorithms) and objects (data structures) occurs in the interaction between C, P and C'. Likewise, we claim that the construction of knowledge of the execution of a program takes place in the internal mechanisms of the thinking process; marked by the arrows between *newC*, *newP* and *newC'*. In other words, the general law of cognition remains applicable to the thinking process represented by the arrows, in both lines of the diagram.

3. Designing the empirical study

The study was conducted with thirteen students from two groups of an initial programming course. Those in the first group (seven students) worked with C++ and those in the second (six students) worked with *Pascal*. The topics covered so far in the course were the same in both groups, except only for the programming language. The study is divided into two parts and all the activities were carried out by students working individually. The first part analyses the process of constructing *conceptual* knowledge (*inter* stage) from *instrumental* knowledge (*intra* stage) and is based on Piaget's general law of cognition.

The second part investigates the *formal* knowledge construction process (*trans* stage) from conceptual knowledge, and is based on the extension to the aforementioned law. The design of each part is described below relating the activities to the supporting theoretical principles.

3.1. Design of the first part (*intra* → *inter*)

For this part, the student is asked to carry out an activity designed to solve a concrete instance of the linear search problem and, after achieving it, to answer questions aimed at obtaining a precise description of how she/he solves it and why the method is successful. The student is presented with a row of numbered cards on a table and is told that they represent door numbers of houses on a street. The task is to search for a specific person within the row of houses.

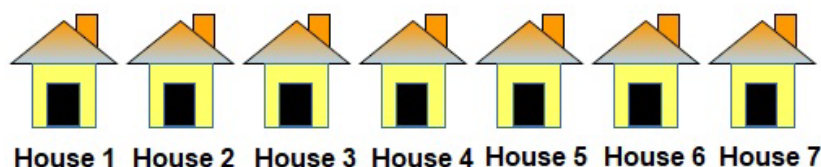


Figure 1 – Simulation with houses

Under each card there is a second card with another number, which represents the id number of the person inside the corresponding house. It is assumed that only one person lives in each house and that every id number is unique. The door numbers are ordered and visible to the student, while the id numbers are not ordered and hidden from the student's view (pretending to be the people inside their houses).



Figure 2 – Simulation with cards

The row of cards simulating door numbers resembles an array of integers in a programming language. The door numbers represent the (ordered) indices of the array, while the id numbers represent the (un-ordered) values stored in their cells. The array is a *computational* representation of the row of houses.

The student is asked to look for a certain id number in the row of cards, in a similar way to being physically positioned at the beginning of the street, in front of the first door. She/he is prompted to go through this process twice; one to search for a number that is in the row and another to search for a number that is not. Because of its nature, each student is expected to successfully solve the task by her/himself, both when the searched number is found inside a door (ending the search at that point) and when it is not (ending after having visited all of the doors).

Once the problem has been solved, each student is asked questions to help them accurately describe the actions performed and explain why her/his method is successful in both cases. It may be necessary to repeat the task so that the student becomes aware of what she/he says in relation to what she/he did, in some cases it could be necessary to pose new questions according to the student's answers. When the description finally matches her/his actions, she/he is asked to put it in writing. This constitutes a first expression in natural language of the linear search algorithm and will be used as a starting point for the second part of the study. In line with the general law of cognition, this process induces the student to become aware of the coordination of the actions carried out (transition from P towards center C) and the changes that said actions impose on the manipulated objects (transition towards center C').

3.2. Design of the second part (*inter* → *trans*)

The passage to the *trans* stage encompasses two aspects. The first is that, being *formal* knowledge, it involves the use of a *formal* language (a programming language). The second is that the execution of

the resulting program is not carried out by the student, but by an *external agent*: the computer. Just as the general law of cognition explains the construction process when it is the subject who performs the actions and builds conceptual knowledge about the algorithm, its extension explains it when the actions are instructions executed by the computer. For this part, three activities were designed to guide the student in the passage from the description in natural language to the writing, compilation and execution of a program that solves the problem on a computer.

In the first activity, each student is asked to write a version of the algorithm using *pseudocode*, based on her/his description in natural language. Pseudocode is an intermediate formalism used to facilitate the beginning of the passage to the *trans* stage. It has less rigorous syntax and semantic rules than a formal language, being easier to understand and allowing a first expression of the algorithm, in order to instruct an external agent (different from the student her/himself) to execute it. In this case, the external agent is an imaginary robot (played by the interviewer), which allows the student to visualise the behaviour of the algorithm, leaving for later aspects related to machine execution.

Each student is asked to write several progressive versions of the algorithm, as a way of gradually consolidate the concepts and correcting errors between each version and the next, until finally arriving at a correct one. According to the extension to the general law of cognition, the purpose is to make them aware that actions are now executed not by the student her/himself, but by the external agent (start of transition from *newP* towards *newC*), which imposes changes on objects to successfully reach the solution (start of transition from *newP* towards *newC'*) (for detailed application of the extended law, see for instance (da Rosa & Aguirre, 2018)).

Between each version and the next, *automation* is used. This mechanism involves the interviewer acting as an external agent, working as an imaginary robot that executes the steps while the student observes its behaviour. This is done to induce the student to reflect on errors that may be detected and correct them, posing questions to the student similar to those of the first part. This process is repeated until the robot executes the pseudocode and successfully solves the problem in any case.

In the second activity, each student is asked to write some code snippets to become familiar with the syntax and semantic rules for working with *arrays* in the programming language used by their group (C++ or *Pascal*) and review other elements of the programming language worked on in class before the study, which she/he will need later to write the linear search program. In addition, this activity helps the student to detach her/himself from the *concrete* instance of the problem (searching for a document in a row of 7 doors) and brings them closer to the more *general* problem of searching for a value in an array of N cells, which they will have to program in the next activity.

As in the first activity, each student is asked to write several versions for each snippet, resorting again to automation until she/he manages to do it correctly and prompting reflection by means of questions in case of errors. The interaction is now with formal object representations of the programming language (an array drawn on paper) rather than with physical objects (numbered cards in a row) that represent objects from the original problem (houses on a street). At this point, automation is used again and the interviewer executes the student's instructions, pretending to be the computer, working on the array.

Automation contributes not only to error detection, but also to the construction of a *general* solution. Although in (Gréco et al., 1963) the jump from specific instances to the general case is investigated by B.Matalon at any stage of knowledge construction, in this study it is especially analysed in the passage to the *trans* stage. Matalon states that in order to construct the concept of *generic* element, it is necessary to carry out a specific action that, by successive repetition, allows the construction of said concept. In this study, the results of Matalon are interpreted by urging the student to work with arrays of different sizes in the snippets, so she/he can build the notion of generic number of elements, using the constant N in the text of the program for the size of the array instead of a specific value (such as 7).

Finally, in the third activity each student is asked to write, compile and run a program that searches for a value in an array of integers. She/he is asked to write it based on the pseudocode version of the

first activity, using the syntax and semantic rules reviewed in the second activity. All variables are assumed already declared and initialised. The student must translate the pseudocode steps into formal language instructions. This translation implies a reflexive process, in which she/he must establish a correspondence between the steps carried out on the row of cards and the instructions executed on the data structure that represents it (the array). Like the two previous activities, this one also implies the construction of knowledge on two aspects: use of a formalism (now a programming language instead of pseudocode) and execution by an external agent (now the computer instead of an imaginary robot). The construction encompasses not only the *textual* part of the program but also its *executable* part, and the relationship between the two (see Section 1).

The student writes a first version of the program on paper, and automation is used again on an array drawn on paper to detect and correct errors. Questions are asked aimed at situating their thinking on what the *computer* must do to solve the problem instead of the student her/himself, focusing on issues typical of machine execution, such as, for example, an *out-of-range* index error or a iteration that does not end (*infinite loop*). The whole process is repeated until the student builds a final version that solves the problem correctly and explains how and why. This enables the student's thinking to consolidate the transitions from *newP* towards *newC* and *newC'* (see Section 2).

The activity ends by asking the student to transcribe the text of the program to the computer, compile it and run it at least twice: once to search for a value that is in the array and another to search for a value that is not. Machine execution constitutes the validation of the knowledge built on the *executable* part, since it is the interaction between the subject and the true external agent (the *computer*). The instructions for initialising the array and requesting the value to search are already provided in a source file. The student must complete the missing portion with the linear search code and display a message indicating the result. If a compilation error occurs, she/he is asked to read it and reflect on how to correct it. If an execution error occurs, she/he is asked to go back to the array drawn on paper and automation is repeated, in order to detect the problem, fix the code and recompile and execute the program again.

4. Implementing the empirical study

In this section we include the implementation of the designed activities and a brief analysis of students' work. All students were successful in writing, compiling and running a program that solves the linear search problem in the programming language used by their group (C++ or Pascal).

4.1. Implementation of the first part (*intra* → *inter*)

For solving this problem, three fundamental actions must be carried out: the comparison of the searched id number with that of the person within each door, the advance to the next door and their repetition at each visited door. Repetition generates a progressive reduction in the number of doors, which eventually leads to the student seeing one of two possible changes due to her/his actions: a change in the relation resulting from the comparison of id numbers (from *different* to *equal*) or a change in the number of doors that remain to be visited (from being *greater than zero* to being *equal to zero*). In terms of the algorithm, these changes represent its two possible conditions for stopping: when finding a number equal to the searched one or when no more doors are left to visit. As an example, the description in natural language given by one of the students is shown below.

Assuming that you want to find the person with id x, you go through the doors in order, opening them and asking the person behind them for their id number. If it's the one we were looking for, the search ends, otherwise, we go to the next door. If all of the doors are visited and the person with id number x is not found, it is deduced that he/she does not reside behind any of the doors.

The student expresses all three actions, comparison (*asking the person behind them for their id number. If it's the one we were looking for*), advance (*we go to the next door*) and repetition (he describes actions referring to the *doors*, in plural). Regarding the changes, he stops both when he finds the id number (*If it's the one we were looking for, the search ends*) and when there are no more doors to visit (*it is deduced that he/she does not reside behind any of the doors*).

All students wrote similar descriptions, describing both the actions (which constitutes evidence of the transition of thought from P to C) and the changes imposed on objects (evidence of transition from P to C'). As expected at this stage, each description was tied to the *concrete* instance of the problem (searching for a person within a row of houses). From this instance, the more *general* problem of searching for a given value in an array of N values was worked on in the second part.

4.2. Implementation of the second part (*inter* → *trans*)

As explained in section 3.2, the second part comprises three activities. In the first one, each student wrote several versions of the algorithm using a pseudocode, based on her/his own description in natural language. An example is shown below.

```
while (not find id number)
  ask for id number in door
  if it's the one I look for
    stop
  else
    end search
  end
end
```

In this version, the student expresses two of the three actions: comparison (*if it's the one I look for*) and repetition (*while*), but he does not express advancing to the next door (he writes *end search* after *else*). As for the *while* condition, he intends the external agent to stop when finding the searched number: *while (not find id number)*, but not when there are no more doors left. In terms of the extension to the general law of cognition (see section 2.2), he is focused on the desired result of making the external agent find the number (*newP*). He has conceptualised the algorithm after applying it by himself but he must conceptualise the action to be executed when the number is not found in the current door (his thought needs to advance towards *newC*) and the second condition to stop the iteration (needs to advance towards *newC'*). After automation, he notices the first problem and writes a second version, by replacing *end search* with *go to the next door*, but the second problem still remains.

```
while (not find id number)
  ask for id number in door
  if it's the one I look for
    stop
  else
    go to the next door
  end
end
```

The missing condition of the *while* instruction is conceptualised after restoring to automation once again, when trying to find a number that does not exist within the row. He intends to continue searching after the last door has been visited, which prompts him to write a third version.

```
while (not find id number) or (there are no more doors)
  ask for id number in door
  if it's the one I look for
    stop
  else
    go to the next door
  end
end
```

Although his thought has advanced towards *newC'* by including both conditions¹ he needs to reflect

¹We recall that *newC'* represents the modifications on data structures, on the one hand, the cards numbers become equals,

about the relation between the semantics of boolean operators and the *while* structure, in order to properly express how to stop. Once again, automation is helpful. When searching again for a number that does not exist within the row, he notices that the condition remains true after having visited the last door, due to the usage of the *or* operator, so he replaces it with *and* and removes the negation in the second condition, resulting finally in a correct solution, as shown below.

```

while (not find id number) and (there are more doors)
  ask for id number in door
  if it's the one I look for
    stop
  else
    go to the next door
  end
end
end

```

In the second activity, each student exercised the syntax and semantics rules of the programming language used by their group (C++ or *Pascal*) to work with *arrays* and their integration with other elements of the language necessary for the third activity. Students wrote short code snippets (not included here for space reasons) in which they not only exercised these rules, but also worked with arrays of different sizes defined by a constant N, in order to conceptualise the generic array of N elements from concrete instances, according to Matalon's research (see section 3.2).

In the third activity, each student wrote, compiled and executed a program that searches for a value within an array of N integers, based on the pseudocode version of the first activity. The focus is now on the correspondence between the steps of the pseudocode and the programming language instructions (knowledge about the *textual* part of the program) and on aspects specific to machine execution, not present in the pseudocode, for example those related to the errors mentioned below (knowledge about the *executable* part). As an example, a sequence of versions of the student's work in C++ is shown below, starting from the pseudocode above (the array indices belong to the range 0..N-1).

```

boolean found = FALSE;
int i = 0;
while ((found = FALSE) && (i < N-1)) {
  if (arre[i] == number) {
    found = TRUE;
  } else {
    i = i+1;
  }
}

```

The student establishes a suitable correspondence between the steps in pseudocode and the instructions in the program. He maintains the *while* structure, the conditions are in the same order, and he combines them with *and* (&& in C++), just like in his pseudocode version above. He unifies in one single statement *if (arre[i] == number)* two separate steps of his pseudocode (*ask for id number in door* and *if it's the one I look for*). His program remains faithful to the behaviour expressed in his pseudocode, which evidences student's construction of knowledge on the *textual* part of the program.

As for the *executable* part, two errors are present. The first is that he uses the *assignment* operator (=) instead of the *comparison* operator (==) in the first condition. The second is a border error in the second condition (he uses < instead of <=, leaving the last array cell unchecked). These type of problems arise specifically at this stage of the process because they involve knowledge construction about *execution on a computer*. Automation is used again, working on an array drawn on paper, allowing the student to detect and correct both errors, resulting in a second (and correct) version shown below, and being finally

and on the other hand, the row of cards becomes empty.

able to explain how and why it works correctly (not included here for space reasons). This constitutes evidence that his thought has continued to advance towards *newC* and *newC'*.

```
boolean found = FALSE;
int i = 0;
while ((found == FALSE) && (i <= N-1)) {
    if (arre[i] == number) {
        found = TRUE;
    } else {
        i = i+1;
    }
}
```

Finally, the student transcribed his code to the computer, compiled it, and successfully executed it. Just like this student, all the others managed to do it successfully as well. For all of them, the passage from pseudocode into a program required fewer attempts (up to three versions) than the passage from the natural language description into pseudocode (up to eight versions). This shows that the start of the process of transforming *conceptual* knowledge into *formal* knowledge was more difficult for all students. According to the extension to the general law of cognition, the beginning of transitions from *newP* to *newC* and *newC'* is a key milestone within the entire process, since at this point each student's thought needs to restructure in order to shift from knowing how to apply the algorithm her/himself towards instructing an external agent to perform the task. The difference in the programming language (*C++* or *Pascal*) made no difference in this regard at all. Below is the program written by a student from the second group, using *Pascal* instead of *C++* (in this program, the array indices belong to the range 1..N).

```
noidnumber := False;
count := 1;
while (noidnumber = False) and not (count = N+1) do
begin
    if (arre[count] <> number) then
        count := count + 1
    else
        noidnumber := True;
end
```

5. Conclusions and further work

This paper presents an empirical study that uses the linear search problem to investigate the process of construction of knowledge about programs, based on a theoretical framework we have constructed from Piaget's principles of a triad of *intra*, *inter* and *trans* stages (see Section 2). It constitutes a detailed case study that shows how our theoretical framework can be used to explain the complete process (from *instrumental* to *formal*) of knowledge construction on programming concepts.

The study analyses the role of students' construction of knowledge previous to any formalisation in the passage from the *intra* stage into the *inter* stage, regulated by Piaget's general law of cognition (see Section 3.1). The passage from the *inter* stage into the *trans* stage, is regulated by our extension of Piaget's law. The use of an *intermediate formalism* (pseudocode) and the mechanism of *automation* are introduced to help students construct knowledge about the program both as a *textual* and an *executable* object (see Section 3.2).

Several lines of future research arise, some regarding construction of knowledge on other algorithmic problems (such as *counting*, *filtering*, *ordering*, etc.) and data structures (such as *linked lists*, *binary trees*, etc.). Other lines of future work involve analysing knowledge construction when other programming paradigms are used for the formalisation at the *trans* stage, for instance, functional programming,

where recursive algorithms play a central role. It is expected that the results from such lines of future research will enable the development of pedagogical guidelines to introduce programming concepts.

6. References

- da Rosa, S. & Chmiel, A. & Gómez, F. (2016). Philosophy of Computer Science and its Effect on Education - Towards the Construction of an Interdisciplinary Group. *Special edition of the CLEI Electronic Journal* (see <http://www.clei.cl/cleiej/>), Volume 19 : Number 1 : Paper 5.
- da Rosa, S. (2010). The Construction of the Concept of Binary Search Algorithm. *Proceedings of the 22th Annual Psychology of Programming Interest Group Workshop, Madrid, Spain*, 100–111.
- da Rosa, S. (2015). The construction of knowledge of basic algorithms and data structures by novice learners. *Proceedings of the 26th Annual Psychology of Programming Interest Group Workshop, Bournemouth, UK*.
- da Rosa, S. (2016). Preconceptions of novice learners about program execution. *Proceedings of the 27th Annual Psychology of Programming Interest Group Workshop, Cambridge, UK*.
- da Rosa, S. (2018). Piaget and Computational Thinking. *CSERC '18: Proceedings of the 7th Computer Science Education Research Conference*, 44–50. <https://doi.org/10.1145/3289406.3289412>.
- da Rosa, S., & Aguirre, A. (2018). Students teach a computer how to play a game. *LNCS of The 11th International Conference on Informatics in Schools ISSEP 2018*.
- da Rosa, S., & Chmiel, A. (2012). A Study about Students' Knowledge of Inductive Structures. *Proceedings of the 24th Annual Psychology of Programming Interest Group Workshop, London, UK*.
- Gréco, P., Matalon, B., Inhelder, B., & Piaget, J. (1963). *La Formation des Raisonnements Recurrentiels*. Les Etudes Philosophiques, 18(4). Presses Universitaires de France.
- Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.
- Piaget, J. (1964). *La prise de conscience*. Presses Universitaires de France.
- Piaget, J. (1974). *Success and Understanding*. Harvard University Press.
- Piaget, J. (1975). *L'équilibration des Structures Cognitives, Problème Central du Développement*. Presses Universitaires de France.
- Piaget, J. (1977). Genetic Epistemology, a series of lectures delivered by Piaget at Columbia University, translated by Eleanor Duckworth. *Columbia University Press*.
- Piaget, J. (1978). *Recherches sur la Généralisation*. Presses Universitaires de France.
- Piaget, J., & Garcia, R. (1980). *Psychogenesis and the History of Sciences*. Columbia University Press, New York.

Evaluating and improving the Educational CPU Visual Simulator: a sustainable Open Pedagogy approach

Renato Cortinovis
Freelance researcher
Bergamo, Italy
rmcortinovis@gmail.com

Ranjidha Rajan
Department of Computer Science
MSU Denver
rranjidh@msudenver.edu

Abstract

This paper describes the user-evaluation of a recently significantly redesigned old but effective educational CPU visual simulator, and its subsequent further improvement in a second main iteration of an Open Pedagogy / OER-enabled Pedagogy project. The goal of the simulator is to support novices in understanding the key components of a CPU by means of detailed animations of the execution of its instructions, in understanding the mapping from high-level control structures to low-level (assembly and machine) code, and in coding meaningful programs with a simple but representative assembly language. The simulator developed in the previous iteration, published with an open licence, was evaluated in three different educational settings: technical high school and adult-education specialised computer science courses in Italy, and a university in the U.S.A. This evaluation, mainly based on the thematic content analysis of the feedback from about 50 students, has been very positive, and provided constructive feedback for further improvements and extensions. Grounded on the results of the previous evaluation, the simulator is being re-developed as a brand-new Web application, further extended with features such as the addition of synchronised audio description of what is happening while the simulator animates the inner working of the CPU, the introduction of support for array processing, and spin-off projects such as a supporting open e-book, and a natural language conversational agent to answer students' queries.

The above extensions are being carried out by students as an OER-enabled pedagogy project, integrated in their more conventional educational activities. This strategy aims to reduce the deplorable waste of resources associated with “disposable” traditional assignments, at the same time challenging students to address real-world professional problems. The open material resulting from this latest iteration will be adopted, further evaluated, and hopefully enriched once again next year on a wider scale, demonstrating the feasibility of a self-sustainable process where students fully engage in iteratively improving and extending open resources, developing their professionalism while benefiting the commons.

1. Introduction

Cortinovis (2021) describes in detail the recent redesign and extension of an old but effective educational CPU visual simulator (Decker and Hirshfield, 1998), which was quite popular in USA universities, but which used technologies that became obsolete. The development was carried out by students attending computer science continuing-education evening-classes of a technical high school in Bergamo, Italy (ITIS P. Paleocapa Serale). The extensions were substantial, including, among many others, the addition of CPU flags and related conditional jump instructions to better illustrate the control flow in low-level languages; the possibility to define and use labels for numerical addresses, in order to clarify the concept of a variable as well as the mapping of high-level programming constructs to assembly language; enhanced colour-coded animations to better understand the sequential nature of the control unit and the role of the control bus in addition to the address and data buses. Figure 1 shows a screenshot of the redesigned Educational CPU Visual Simulator, with a sample program computing the sum of all numbers from 1 to MAX (which has value 5 in this case), loaded in RAM.

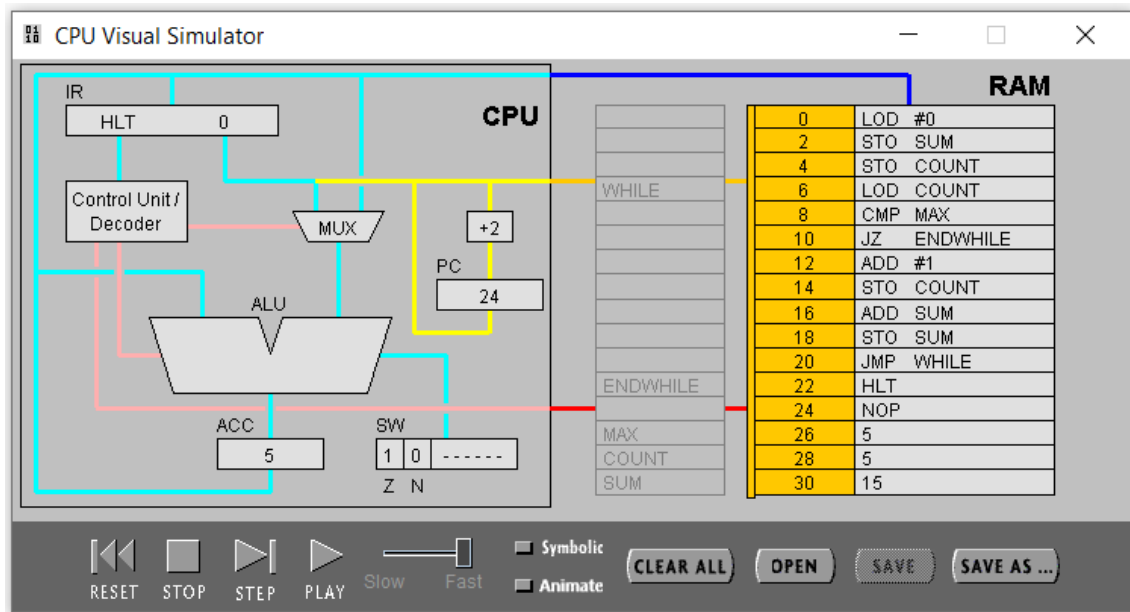


Figure 1 – A screenshot of the new CPU Visual Simulator

The literature reports that often students, despite studying both programming with high-level languages as well as the basics of computer architecture, do not fully grasp how code written in high-level languages is actually executed on the hardware of a computer (Evangelidis et al., 2001; Miura et al., 2004). Hence, the main goal of the simulator is to support novices in understanding the key components of a CPU by means of detailed animations of the execution of its instructions, in understanding the mapping from high-level control structures to low-level (assembly and machine) code, and in coding meaningful high-level programs with a simple but representative assembly language. As an example, Figure 2 reports the high-level pseudocode from where the assembler code in Figure 1 was derived.

```
// sum all the integer numbers from 0 to MAX
// assertion: MAX >= 0

SUM = 0
COUNT = 0
WHILE COUNT != MAX DO
    COUNT = COUNT + 1
    SUM = SUM + COUNT
ENDWHILE
```

Figure 2 – Pseudocode corresponding to the assembler code in Figure 1

The redesigned simulator was developed following an Open Pedagogy / OER-enabled pedagogy approach (Wiley and Hilton, 2018), where students modified the old simulator, carrying out all the necessary design, refactoring, and coding, as part of their educational activities in an otherwise traditional programming course. The result of their efforts was made openly available for use and for further improvements by other students and teachers. This approach challenged students to address a real-world professional problem, an experience which was deeply appreciated. One of them, for example, reported:

*“I was impressed to experience how solving real problems such as introducing a new feature in a broader complex application, had such a positive impact on my motivation and my **passion** for software development”.*

This strategy also aimed to reduce the deplorable waste of resources associated with “disposable” (write and forget) traditional assignments, while attempting to provide a contribution to the sustainable development goals (Lane, 2017). All the students involved in this open pedagogy project, indeed, were thrilled by the opportunity to personally contribute to the common good. One of them, for example, commented:

“Working for a purpose and contributing to a wider goal is definitely more rewarding than getting a good score on a typical classroom assignment for its own sake”

Besides motivating and benefiting the students who worked on enhancing it, the simulator is also benefiting the students who use it to learn computer architecture concepts; in this paper, indeed, we describe the international evaluation of the redesigned simulator from the perspective of these last students (Section 2). We also describe the resulting identification of further requirements and opportunities for extensions of the simulator (Section 3), and its development as a brand-new Web application (Section 4). The resulting product will be published again with an open licence, hence enacting a sustainable OER-enabled pedagogy process.

2. Evaluation of the simulator

The open simulator described above, was evaluated in three different educational settings: technical high school and continuing-education computer science courses in Italy, and Metropolitan State University Denver (MSU Denver) in the U.S.A.

2.1 Experimentation in a technical high school and in continuing-education evening courses

The simulator was first used by one of the authors in a continuing-education introductory class of 23 students, for a total of 30 hours. First, many students experienced difficulties installing the simulator, which proved to be a real hassle. Indeed, we discovered, different operating systems, Java runtimes, and computer settings, required minor but annoying ad-hoc interventions. Second, there was a fairly comprehensive help already embedded in the simulator, but many students expressed the desire for additional supporting material. This request led to the fast prototyping of a supporting e-book with comprehensive explanations of the hardware structure, the instruction set, patterns for the translation of high-level programming constructs, exercises with step-by-step solutions, and additional proposed exercises. The e-book was quickly prototyped by the students of an advanced class of the same school, and extensively used by both the students and the instructor in the previous foundational class. This e-book was developed in a cloud-based Content Management System in order to allow the developers to quickly make the necessary modifications, reacting in real time to the emerging needs.

The visual simulator plus its associated e-book were then used in another technical high school class with about 20 students. The feedback from the experienced teacher was definitely positive, and he contributed a few additional exercises that were further elaborated and integrated into the existing e-book. Yet the teacher observed that the simulator could be even more useful if it supported functionalities to handle arrays too.

2.2 Experimentation in CS organisation courses at university level

The simulator and related e-book were then experimentally adopted at MSU Denver by one of the authors, in a “2000 level architecture course” class of 30 students, and subsequently in a second “1000 level architecture” class with 20 students.

2.2.1 First MSU Class: “2000 level architecture course”

Despite the simulator being intended as a resource for an introductory course, it could only be used, in the first university class, after the students had already completed their ARM (Arm Limited, 2021) programming class, because the decision to use the material was taken after the policies and the syllabi for the university courses were already established. The material was introduced by the lecturer in a two-hour session, and was distributed to the students via the internal LMS. Students were required to carry out the exercises proposed in the documentation with the simulator, and to complete, within three weeks, a short survey to collect their feedback. The following questions were included:

- What was most interesting?

- What should be improved so you can get a better learning experience?
- What is your experience when learning a new instruction-set with the basic knowledge of ARM instructions and its datapath?

The 30 students of this first class provided more than 2000 words of meaningful feedback, that was analysed following a thematic content analysis approach (Cho and Lee, 2014; Stemler, 2001). Most higher-level codes (“themes”) were identified top-down from the questions, while the lower-level codes were identified inductively bottom-up. The most relevant codes were:

- Positive aspects
 - Detailed animations
 - Flow-control instructions
 - Simulator execution control
 - Switching between symbolic and binary
 - Embedded help
- Suggested improvements
 - Additional explanations
 - Real-time explanation of animations
 - Explanation of CPU sub-components
 - Usability
 - Installation
 - More detailed user control on the animations
 - Full screen
 - Use of colours
- Acquisition of previously missed concepts

Here a few excerpts are reported, to illustrate some of the codes.

Positive aspects / Detailed animations

The detailed animations of the simulator were, by far, the most appreciated feature. For example:

The most interesting part to me was visually being able to indicate what was happening when each instruction was executed. [Student 15]

I found it really helpful to see what was happening as the program moved and did the steps in the program. [Student 3]

The most interesting part of the program was the CPU’s circuit and how we can see how it’s getting looped through. [Student 11]

Suggested improvements / Additional explanations / Real-time explanation of animations

Various students expressed the wish to have additional real-time explanations of the animations. For example:

I think written details of what is occurring, an explanation of the animation would make a better learning experience. [Student 4]

It would be cool to have the option to have a guided tutorial/dynamic text instruction of what is happening while each instruction executes. [Student 17]

Acquisition of previously missed concepts

As previously discussed, the simulator was intended as introductory material, while students of this first MSU class were exposed to an ARM processor before the visual simulator. This unplanned situation proved very useful to demonstrate that the simulator helped the students to grasp aspects that they missed by working first in the ARM environment. This is suggested, for example, by the following excerpts:

*The most interesting part was watching how the PC directs the CU to the register [memory location] where the command is found and then storing the instructions in the IR. I always imagined the process as one; **instead**, the PC points to the register [memory location], the CU takes the command and stores it in the IR, decodes the instructions, and then carries out the instructions. [Student 5]*

*The flow from the program counter and the control unit, the way things were executed was **different** than I had really expected. [Student 14]*

*The most interesting part about the data path simulation to me was to see the instructions getting passed around. I was **surprised** to see that the control unit is pretty much always in use whether it is grabbing instructions or passing them along. [Student 19]*

This was likely induced by the visualization and animation of the CPU behaviour, which helped students to enrich their understanding from a partially confused abstract level to a more firm and concrete grasping.

2.2.2 Second MSU Class: “1000 level architecture course”

The second university class of 20 students fit precisely the target population for which the simulator was intended. Following the same process as in the previous class, with a change in the questions to align with the foundational course, the following questions were used:

- Which examples did you try for understanding the simulator (submit the screenshot of those)?
- Reflect on your understanding of how high-level code runs on hardware using Von Neumann architecture.
- What improvements can be made in the simulator for a better understanding?

More than 1000 words of feedback were collected and analysed. Despite the different background of the students, the feedback was consistent among the two classes, even if with a slightly different emphasis, and the codes obtained from the analysis of the previous class could be used in this second case too. For example:

Positive aspects / Detailed animations

The visualizer helps a lot in showing how assembly operates. [Student 17]

Suggested improvements / Additional explanations / Explanation of CPU sub-components

I think there are different parts that I wish I could hover my mouse over to get a detailed description of what it actually is. [Student 5]

Acquisition of previously missed concepts

Unlike the students of the previous class, the students of this class had not completed a previous course on computer architectures. Yet, they did have some prior knowledge coming from previous lessons, and the feedback from the following student shows again that the simulator was instrumental in integrating previously acquired knowledge:

I think the Von Neumann architecture model provides a great visual for putting together everything we have learned so far. [Student 2]

Finally, the following feedback provides some indication that one of the main objectives of the simulator was achieved:

I previously didn't know how higher-level languages got translated into something the computer can understand and seeing this is quite fascinating. [Student 17]

The simulator helps us to understand that high level code runs on Von Neumann architecture by first being run through a compiler, which converts it into assembly, which is then assembled into machine code. [Student 3]

3. New requirements identified

The analysis of the feedback as previously discussed, allowed us to identify requirements for the improvement and extension of the visual simulator in a new OER-enabled pedagogy iteration. The main requirements could be summarised as follows:

- Re-develop the simulator as a new Web application;
- Provide support for array processing;
- Provide synchronised audio (and textual) description of the animations in the simulation (with a multilanguage architecture);
- Improve user control over the animation speed and level of detail; in particular, provide the possibility to pause the animation and resume it without restarting the instruction;
- Port the existing supporting e-book to HTML/JavaScript and enrich it with additional interactive exercises;
- Provide appropriate mechanisms (possibly including a natural language conversational agent) to answer students' queries.

4. Development of the new enhanced version

Following the Open Pedagogy / OER-enabled Pedagogy approach, and according to the requirements identified, students of the continuing-education evening courses are now developing a new version of the educational material. That is, they are producing improved/extended open educational material, based on the evaluation of previously existing open educational material.

First, a brand-new Web-based version of the simulator is being finalized, to let users enjoy the material directly online through a familiar browser, avoiding the hassle of installing an application and the Java runtime. The new simulator provides improved functionalities, notably a synchronised audio or textual description of what is happening, such as: "The CPU is now going to fetch an instruction: the content of the PC is placed on the address bus... a read operation is signalled on the control bus... the memory retrieves the content stored at the address indicated and transmits it on the data bus...". The audio description has been selected because the visual channel of the user is already fully engaged in following the animations of the simulator. Yet the possibility to use a textual description is made available to avoid excessive noise when multiple students are using the simulator in places like a lab without headsets.

To address the issue raised by the teachers of the technical high school class concerning the desirable support for arrays, the possibility of self-modifying code (code that alters its own instructions at runtime) has been introduced. This solution has been selected, instead of alternatives such as adding a new register with indirect addressing, for the following reasons: (1) it does not increase the complexity of the visualization by requiring another implicit or explicit register; (2) it does not increase the complexity of the existing instruction set with a new addressing mode and possible register addressing instructions; (3) it naturally exploits the Von Neumann architecture of the simulator; (4) it opens the door to address interesting and contemporary issues such as security aspects and genetic programming.

Other features include the possibility to pause and resume the visual simulation without restarting the whole instruction, and a better control of the execution speed.

The associated e-book is being further extended, based on the requests for additional documentation and further examples. The current version is being ported from a proprietary CMS to pure HTML/JavaScript technology, further enriched with additional interactive exercises, and improved providing additional support that proved to be frequently needed by students and instructors. As an example, users got frequently stuck trying to use a label in an instruction as placeholder of a physical address, when the label was not previously defined.

4.1 Tentative spin-off projects

Finally, we are also working on the development of an experimental natural language conversational agent to answer students' queries. This might well sound over-ambitious: it would require a considerable effort in itself, while not even being essential to the project. Yet, this is just one of a number of tentative spin-off projects, aiming at providing opportunities to experiment with technologies some students are curious about.

The overall project provides context and meaning: a grand vision, a broad shared goal, a sense of community both at local and global levels. In such a context, every achievement, even if modest, will be a success, because it will provide anyway some useful contribution to the wider goal. At the very least, thanks to the "open" philosophy, it will advance the starting point for other students who will forge ahead an extra step. For example, the students working on this spin-off project have already collected a number of questions and prepared suitable answers (by actually providing support to other students with lower competences): at the bare minimum, these questions could be easily integrated as useful FAQ in the documentation of the simulator. Yet, the vision is to avoid setting upper limits on what students can accomplish, but rather to carefully shape and suggest opportunities that fit the students' natural interests, boost enthusiasm, sustain and occasionally steer efforts, and finally just make the most of whatever achievement.

5. Conclusions and future activities

This paper has described the qualitative user-evaluation – in the USA and in Italy – of a recently significantly redesigned old but effective educational CPU visual simulator, as well as the resulting plans for its development as a brand-new Web application. The development of the new simulator and associated complementary material, is being carried out by students attending continuing-education evening-classes of a technical high school in Italy. These developments are organized as OER-enabled pedagogy projects integrated in their conventional educational activities, exploiting open material that was previously developed with the same approach, in the same school, mostly by students of previous generations. This strategy aims to reduce the resources wasted in "disposable" traditional assignments, challenging students to address real-world professional problems, to produce open material useful to a wider community. The resulting material will be adopted the next academic year on a wider scale – in particular, it will be integrated in the "1000 level architecture" courses at MSU Denver to explain how high-level languages end up being executed in the CPU – which is one of the main objectives of the simulator. This experience will permit a further evaluation of the simulator and associated documentation, to be hopefully further enriched in a new future OER-enabled pedagogy iteration. These activities suggest that the OER-enabled pedagogy, where students engage in iteratively improving and extending open resources, developing their professionalism while benefiting the commons, is inherently self-sustainable.

6. Acknowledgements

We would like to thank the students who engaged in these OER-enabled pedagogy activities, in particular Jonathan Cancelli for his generous contribution to the development of the simulator, as well as the students who keenly provided their constructive feedback. We would also like to thank Cengage for granting the permission to modify, reuse, and republish the obsolete original applet (non-commercial, educational purposes only).

7. References

- Arm Limited. (2021) *Armv7-M. Architecture Reference Manual* (E.e ed.).
- Cho, J. Y. and Lee, E. H. (2014) Reducing confusion about grounded theory and qualitative content analysis: Similarities and differences. *The Qualitative Report*, 19(32), pp. 1-20.
- Cortinovis, R. (2021) *An Educational CPU Visual Simulator*. Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG), York, United Kingdom.
- Decker, R., & Hirshfield, S. (1998) *The Analytical Engine: An Introduction to Computer Science Using the Internet*. PWS Publishing, Boston.

Evangelidis, G., Dagdilelis, V., Satratzemi, M., & Efopoulos, V. (2001) *X-compiler: Yet another integrated novice programming environment* [Paper presentation]. Proceedings IEEE International Conference on Advanced Learning Technologies, pp. 166-169. IEEE.

Lane, A. (2017). Open Education and the Sustainable Development Goals: Making Change Happen, *Journal of Learning for Development - JL4D*, 4(3), pp. 275-286.

Miura, Y., Kaneko, K., & Nakagawa, M. (2004) *Development of an educational computer system simulator equipped with a compilation browser* [Paper presentation]. Proc. Int'l Conf. Computers in Education, pp. 1067-1071.

Stemler, S. (2001) An overview of content analysis. *Practical assessment, research & evaluation*, 7(17), pp.137-146.

Wiley, D., & Hilton, J. (2018) Defining OER-enabled Pedagogy. *International Review of Research in Open and Distance Learning*, 19(4).

A Grounded Theory of Cognitive Load Drivers in Novice Agile Software Development Teams

Daniel Helgesson
Dept. of Computer Science
Lund University
daniel.helgesson@cs.lth.se

Daniel Appelquist
Softhouse AB
daniel.appelquist@softhouse.se

Per Runeson
Dept. of Computer Science
Lund University
per.runeson@cs.lth.se

Abstract

Objective: The purpose of this paper is to identify the largest cognitive challenges faced by novices, developing software in teams, using distributed cognition as an observational filter.

Paradigm: Design science

Epistemology: Pragmatist

Methodology: Case study

Method: Using **grounded theory**, **ethnography** and **multi method data collection**, we conducted an observational study for two months following four 10-person novice agile teams, consisting of computer science students, tasked with developing software systems.

Result: This paper identifies version control and merge operations as the largest challenge faced by the novices, and provides a substantive theory generated from our empirical data explaining the observed phenomena. The literature studies reveal that little research appears to have been carried out in the area of version control from a user perspective.

Limitations: A qualitative study on students is not applicable in all contexts, but the result is credible and grounded in data and substantiated by extant literature.

Conclusion: We conclude that our findings motivate further research on cognitive perspectives to guide improvement of software engineering and its tools.

1. Introduction

By now it is well known that software development is a sociotechnical phenomenon, rather than a purely technical-technological one (Bertelsen, 1997). It is further well known that most, if not all software development activities are cognitively intensive (Sedano, Ralph, & Péraire, 2017) and rely on software development tools. The human cognitive ability to process and harbour information is limited (Miller, 1956), so for seemingly obvious reasons it would make sense to lessen the cognitive load on the individual user.

But inspite of more than 50 years of investigation in regards to psychological and cognitive dimensions and phenomena (Blackwell, Petre, & Church, 2019) softer, non-technical phenomena remain underinvestigated (Lenberg, Feldt, & Wallgren, 2015). Some twenty years ago Walenstein (2002) highlighted the need of cognitive design theory as means to inform software development tool design and suggested distributed cognition as a suitable foundation for such theory (Abend, 2008) building and theorization processes.

In an attempt to more broadly understand the cognitive load induced from software development tools and processes, we studied cognitive load drivers in large scale software development (Helgesson, Engström, Runeson, & Bjarnason, 2019) and found three clusters of drivers, namely tools, information, and work & process. We also noticed that the temporal perspective of software development, particularly revision control created specific problems. We have further described a set of 'perspectives' (Helgesson & Runeson, 2021) from which cognitive load in software development can be observed and analysed.

While Walenstein (2002) described software development from a distributed cognitive perspective, distributed, agile, software development software development has changed considerably over the past

twenty years, so in order to further advance the understanding of cognitive load in software engineering/development and agile software development as a distributed cognitive set of phenomena, we set out to ethnographically (Sharp, Dittrich, & de Souza, 2016) study cognitive load drivers in agile software development projects, using grounded theory (Charmaz, 2014) in conjunction with distributed cognition (Hollan, Hutchins, & Kirsh, 2000).

As we hypothesise that some of the cognitive loads are compensated and mitigated through increased experience and workarounds learned over the years, we choose to study novice software engineers (Höst, Wohlin, & Thelin, 2005) in order to capture the novice *point of view* (Sharp et al., 2016). Our study context is quite advanced for novices, an agile software engineering course, running for 14 weeks, in which students work in 10-person teams in a simulated work environment, adhering to XP principles. We observe four teams out of a total of twelve teams participating in the course.

Our research goals are to identify the most dominant *cognitive load drivers* and to observe differences and similarities between groups with different characteristics. We combine the teacher role of on-site customer with the ethnographer role, taking field notes of the observations. Further, we collect weekly questionnaires, short reflection notes from the students, and arrange a focus group discussion with each team. We use grounded theory practices in coding all the material, from which our theory emerges.

We conclude that version control, branching and merge operations are the dominant load factors in the projects observed, and subsequently explore these phenomenons in detail. The remainder of the paper is arranged as follows: background, method, analysis, literature review review, ethical consideration, validity and discussion.

2. Background

2.1. Distributed Cognition

Not only is software engineering, as previously mentioned, a sociotechnical (Bertelsen, 1997) phenomenon, it is also 'cognitively intensive' (Sedano et al. 2017). If we allow ourselves to theorize (Abend, 2008) in regards agile software development in teams, it is quite easy to envision the phenomenon as a distributed network of cognitive agents solving cognitively loaded tasks using computers and software development tools¹.

Distributed cognition is a sub-discipline of studies of cognition in which the one of the traditional cornerstones of cognition – “that cognitive processes such as memory, decision making and reasoning, are limited to the internal mental states of an individual” (Hansen & Lyytinen, 2009) – is questioned and rejected. Instead it argues that the social context of individuals as well as artefacts forms a cognitive system transcending the cognition of each individual involved (Flor & Hutchins, 1991), i.e., a cognitive system extending beyond the mind of one single individual (Mangalaraj, Nerur, Mahapatra, & Price, 2014). The concept was pioneered by Hutchins who studied the cognitive activities on the navigation bridge of US naval vessels (Hutchins, 1995).

Hollan, Hutchins and Kirsh extended distributed cognition into the realm of human-computer interaction as well as to some extent into software engineering, stating that a distributed cognitive process (or system) is “delimited by the functional relations among the elements that are part of it, rather by the spatial colocation of the elements”, and that as a consequence “at least three interesting kinds of distribution of cognitive processes become apparent: [a)] cognitive processes may be distributed across members of a social group[:] [b)] cognitive processes may involve coordination between internal and external (material or environmental) structure [and, c)] processes may be distributed through time in such a way that the products of earlier events can transform the nature of later events.” (reformatted but verbatim). (Hollan et al., 2000)

Despite the fact that the theory of distributed cognition was suggested as a fruitful approach for investigating and explicating phenomenon related to software engineering several decades ago – Flor

¹The dissertation (Walenstein, 2002) makes for excellent indepth reading on the matter

and Hutchins empirically studied pair-programming from a distributed cognition perspective as early as 1991 (Flor & Hutchins, 1991) – few examples exist of actual software engineering studies using distributed cognition as theoretical underpinning. Mangalaraj et al. (2014) highlighted Sharp and Robinson (2006), Hansen and Lyytinen (2009), and Ramasubbu, Kemerer, and Hong (2012) as “the few notable exceptions” of extant software engineering research utilising Distributed Cognition. To this list we would like to add Walenstein (2002), a recent study by Buchan, Zowghi, and Bano (2020) as well as Sharp and Robinson (2004), Sharp, Robinson, and Petre (2009), Sharp, Robinson, Segal, and Furniss (2006), Sharp, Giuffrida, and Melnik (2012), Sharp and Robinson (2008) and Zaina, Sharp, and Barroca (2021).

A recent “exploratory literature review” on “Cognition and Distributed Cognition” is presented by Begum (2021), where the authors reached a similar conclusion to ours – that despite its’ intrinsic promises distributed cognition remains largely unused in software engineering.

3. Method

3.1. Grounded theory

Grounded theory (GT) is a systematic and rigorous methodological approach for inductively generating theory from data (Glaser & Strauss, 1967) (Charmaz, 2014) (Stol, Ralph, & Fitzgerald, 2016). Stemming from social sciences, GT was developed by sociologists Glaser and Strauss, as a qualitative inductive reaction to the quantitative hypothetico-deductive research paradigms dominant in the 1960’s. The main difference, apart from being qualitative rather than quantitative, is that the purpose of GT aims at *generating theory*, rather than to be used as an instrument for validation, or testing, of theory (Stol et al., 2016). It is iterative and explorative (Charmaz, 2014) in nature, and thus suitable for answering open ended questions such as *what’s going on here?* (Stol et al., 2016).

We primarily opted for Charmaz GT handbook (Charmaz, 2014) as guidelines, using Bryant (2017) as a complementary perspective (in addition we also consulted earlier works by Glaser (1978),(1992)), specifically using grounded theory in conjunction with ethnography (Charmaz & Mitchell, 2001) – an approach that gives “*priority to the studied phenomenon or process – rather than the setting itself*” (Charmaz, 2014). The ethnographic approach allows for exploring not only *what* practitioners do, but also *why* they do it (Sharp et al., 2016). Core elements in the ethnographic approach is the empathic approach *to describe another culture from the members point of view* and the intrinsic *analytical stance* (Sharp et al., 2016). As with grounded theory, modern ethnography also stems from social sciences (Sharp et al., 2016). Not extensively used in Software Engineering (Sharp et al., 2016), it has however been used to study agile teams (Sharp & Robinson, 2006)(Sharp & Robinson, 2004).

3.2. Research goals

Central to ‘original’ Glaserian GT and Charmaz Constructivist GT is that the actual/final research questions are not defined up front. Glaser suggests that the researcher should start with an *area of interest* (Glaser, 1992) (Stol et al., 2016), while Charmaz suggestion is that the researcher should start with *initial research questions* that *evolve* through the study (Charmaz, 2014) (Stol et al., 2016). We decided to pursue two open ended research goals:

- A) To identify the most dominant *cognitive load drivers* from the *novice point of view*, and
- B) To chart what *differences* or *similarities* that can be observed between the different *group compositions*.

3.3. Case description

The course that we used as study object is a mandatory course for sophomore computer science² students aiming at teaching practical software development in teams using agile methodology, presented in detail by Hedin, Bendix, and Magnusson (2005). The course runs for two terms (14 weeks) and consists of

²Translations of educations are difficult, in international terminology ‘Computer Science’ is as close as we can translate it. It is a five year master (engineering) program mostly aimed at software rather than hardware. The program resides at the Faculty of Engineering, and the program responsables reside at the department of Computer Science.

one study block (seven weeks) consisting of lectures and practical lab work, and one study block (seven weeks) in which the students work together as 10-person teams, largely adhering to XP principles (Sharp & Robinson, 2004) developing a software product. All teams develop a software system based on the same basic stories, but the stories are somewhat open ended, leaving room for differentiation. The teams are coached by two senior students undertaking a course in practical software coaching, that runs in parallel for the same duration. PhD students serve as *customers*, for 3-4 teams each.

The teams develop their system for a term (seven weeks) in 6 full day sprints, each preceded by a two hour planning session in which the *cost/effort* for the user stories are estimated by the students and prioritised by the *customer*. The students make 3-4 incremental releases during the project, roughly with a cadence of one release every two sprints.

3.4. Design considerations

We opted for a flexible case study design (Runeson, Höst, Rainer, & Regnell, 2012), to allow for improvisation based on observations and forces outside of our control (which once you take research into the wild are plentiful). Once in the field, flexibility becomes utterly important (Sharp et al., 2016) as the researcher must be ready to adapt to changing situations quickly.

We had a strict time box for our field study, since the course executed over the duration seven weeks with one day sprints on Mondays, following a two hour planning session on the Wednesday before. Apart from the fixed schedule for observations we also had to take into account the work load of the students when injecting experiments and eliciting interviews. We had the ambition to cause as little disturbance as possible. In order to achieve triangulation we opted to collect as many data sources as possible.

We also decided to use *distributed cognition* (Hollan et al., 2000) as initial lens, or filter, for our observations. Distributed cognition, further described in Section 5, is a branch of cognition studying cognitive processes distributed in groups rather than cognition from the individual perspective. While the use of an initial lens could be thought of by some readers as contradictory to the central tenet in GT, we hold this (potential) critique as moot. We were targeting observations of cognitive load drivers in interconnected network of people and digital tools, so we needed some starting point for our observations.

3.5. Student selection

Firstly we anonymously picked 14 student candidates, based on a high grade (grade average in excess of 4.5 on five grade scale, where *pass* is denoted as 3) in the first two programming courses, and a lower grade (i.e. *pass* or *incomplete/fail*) grade in multidimensional calculus. Secondly we anonymously picked 14 student candidates based on a high grade (grade higher than *pass* on five grade scale, where *pass* is denoted as 3) in multidimensional calculus, regardless of their grade in programming courses.

The two anonymous candidate lists were then sent to the course responsible who then created one experimental group each out of the two candidate lists and two randomly selected groups. After this process we had four groups in total. It should be noted that the authors at no point in time were informed of what group consisted of what selection.

3.6. Consent

Together the course responsible and the first author ultimately reached the conclusion that the optimal solution (in regards to time constraints and complexity) was to inform the students in the four groups at the start of the course that we would be carrying out research throughout the course, describe the overall purpose/general research goal of the study, that we were looking at the groups and not the individual members and offer any student not willing to participate to change groups prior to the first sprint. No student asked to exchange groups.

In every interaction that was recorded or photographed, we actively asked every student participating for permission, while pointing out that everything expressed in the exchange would be anonymous and confidential, and that no recordings would be distributed outside of the three researchers participating in the study. For further ethical considerations, see Section 6.

3.7. Data collection

The first and the second author followed all planning sessions in parallel. As we had to monitor sessions in parallel we opted to alternate between observing in pairs and by ourselves. All in all we covered 24 planning sessions where the first author actively participated in the meetings acting as *customer on site* providing students with clarifications of stories, priorities etc, while the second author passively observed. After each session we spent, roughly, 15 minutes discussing what we had observed. Field notes were written by hand, and after the termination of the field work compressed in *memo* form. The first author actively participated in all full day sprints while acting as *customer on site*. The four teams were situated in two computer labs, allowing for observation of two teams simultaneously. Field notes were written by hand, and after the termination of the field work compressed in *memo* form. We specifically opted to not be part of breaks, lunch hour etc. for respect of the students integrity. Since our research focus is the phenomenon of cognitive load from a team perspective, rather than team work in general, we do not see this as a threat to our observations.

In addition, we added a weekly questionnaire to be filled out by each student after every sprint (all in all $4 \times 10 \times 6 = 240$ questionnaires) in order to follow up on what we had observed so far throughout the project. The first two weeks the questionnaire targeted sources of information and information tools used by the students. In the third and fourth questionnaire we introduced check boxes and free text space, allowing the students to express what they perceive as the major problems they had been challenged by throughout the project. In the fifth questionnaires questions were added to capture the outcome of one of the experiments, see Subsection 3.8.2. The final questionnaire was extended with questions regarding team spirit and over all satisfaction. The aggregated response rate for all 24 sets of questionnaires (6 for each team) were 93% (out of the 240 questionnaires we handed out we got 223 in return, and no single set had a lower response rate than 8/10).

Further, as a requirement of the course all students wrote short individual reflections after each sprint, as a retrospect exercise. After the course we aggregated these pages, anonymised all content and created one .csv file per team with the content broken down in line-by-line format for *open coding*.

After the final sprint we held one hour long focus group discussion with each team. The discussions took place in two by two parallel sessions, Two instances were held by the first author, one by the second and third author collectively and one by the second author. In order to keep the different sessions coherent and comparable we followed a semistructured manuscript containing four themes we had selected as emerging concepts from our observations. We used pair-wise post-it discussions, followed by group discussions where each pair reflected on what they had come up with. The post-it stickers were collected, numbered and digitized. Each session was also recorded using video and sound.

3.8. Field experiments

Inspired the reasoning on *ethnographically natural* experiments by Hollan et al. (2000), we decided to extend our data set with the result and observations from three minor field experiments. These were dressed as improvised exercises, a part of the overall course concepts, where *unplanned* customer changes could take place (Hedin et al., 2005). One of them had been used by first author in previous years, the others were new.

3.8.1. Field experiment – group constellation

Our first experiment consisted of creating four teams with different member compositions, with the purpose to see what differences, if any, we could observe during the observation study (and through the other data sources). See Subsection 3.5.

3.8.2. Field experiment – exploratory testing

The second experiment consisted of assigning the students with a surprise story in preparation for the fifth sprint. The story consisted of little more than the instructions to: *execute roughly 1 hour of exploratory user tests of the system under realistic race conditions using four team members documenting the issues encountered*, and further to reflect on the experience in their weekly reflections (that all students fill out after each sprint). The story was handed out during the planning session the week before

the full day sprint during which it was planned. We collected information of the activity from questionnaires (Q5/Q6) and from discussions with students and coaches during the following sprint and planning session.

3.8.3. Field experiment – merge-back

The third experiment consisted of the request to implement two sets of changes, in two separate files, and upon completion of the first task request a merge-back and recreation of the first release. Each team was handed a story card describing the two code blocks to be implemented *first thing in the morning* during the final planning session leading up to the final sprint. Each team was asked to notify their *customer* upon completion of the task. In order not to compromise that functionality/integrity of their respective systems the two code blocks were dummy snippets that were commented out. The experiment was documented using video and sound recording.

3.9. Analysis

Given that we had a limited time window for our observation, we did not have a lot of time for analysis during the field work. We exchanged notes and discussed our observations over lunch breaks. After the field work was completed the first and second author started a more formal analysis stage.

Initial coding (Charmaz, 2014) – the first and second author each performed open line-by-line coding of the student reflections and the post-it stickers. We then exchanged our reflections in short memo form. In parallel, the first author did an initial overview of the contents of the questionnaires.

Focused coding (Charmaz, 2014) – the first and the second author had a two day session in which the questionnaires, focus groups post-it stickers and student reflections were analysed from multiple perspectives and the parts that we found relevant was extracted and documented digitally. We also extracted relevant ‘soundbites’ from free text answers, and digitised them. The findings were condensed in a short memo.

Theoretical coding (Charmaz, 2014) – the theoretical coding was executed by the first author, using Glaser’s ‘6C’-coding family (Glaser, 1978) as a starting point. The work was done in memo form and visualized on an A1 sheet using postit stickers. After a few iterations of coding, sketching and memoing a theory was emerging. The first and third author had a one hour session in which the theory was discussed from various angles and a few of the constructs were redefined. After this the first author did a minor rewrite of the theoretical coding memo.

3.10. Theoretical saturation

Having iterated through *open coding* and *focused coding* of the data set, we saw the need of further *saturation* in order to provide some more insight from *the members’ point of view*. In order to do so, we went through the recordings of the focus groups in order to provide some additional insight. Finally we reached out to a handful of students whom previously agreed to do minor follow up interviews. We held three short (15–20 minutes) open interviews specifically aimed at understanding what the students perceived as tool interaction related issues. The interviews were conducted by the first author and were documented by additional field-notes. All quotes and findings were reread to the subjects at the end of these interviews.

3.11. Literature review

In its original form, research questions in GT studies should emerge from the research, not be defined apriori (Stol et al., 2016) and *extensive* literature should be avoided prior to the emerging of theory. That being said, Charmaz takes a more pragmatic stance on literature and research questions and emphasises the iterative nature GT, thus allowing for initial research questions that evolve through the research project as well as abductive reasoning on extant literature, recommending a *preliminary* literature review “without letting it stifle your creativity or strangle your theory” (Charmaz, 2014).

As a consequence we did an initial, rather limited, literature study of Distributed Cognition from a Software Engineering perspective. Following the coding cycles we did an additional, or final, literature

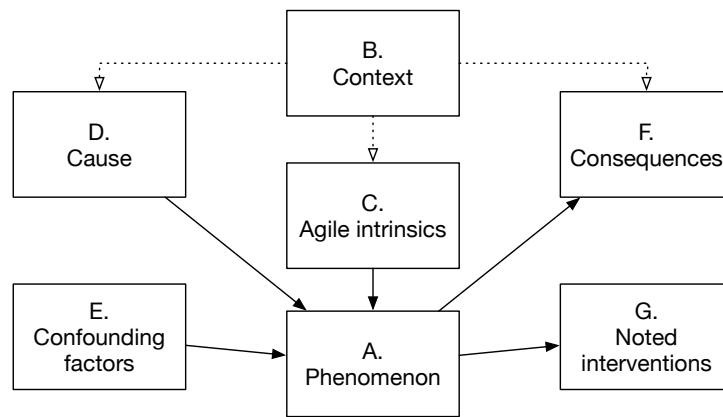


Figure 1 – Generated theory of the causal and consequential dimensions in regards to version control, branching and merge operations encountered in the projects.

review on the central phenomenon of the theory we generated, i.e. Git, version control and merge operations from a user perspective. See Section 5 for findings.

4. Analysis

This section presents the theory generated from the dataset. Based on the findings from open and focused coding of our data set, the emerging concept we focused on was *issues regarding version control, branching and merge operations*.

For the first attempt at formulating the theory, a theoretical conceptual explanation of what we observed, we based our theoretical coding on Glaser's 6 C-coding family (Glaser, 1978) (Stol et al., 2016), while observing Thornberg and Charmaz reflection that the researcher should avoid being *hypnotized* by Glaser's coding families (Thornberg & Charmaz, 2014). This is analogous to Glaser's argument that all codes should *earn* (Glaser, 1978) their way into the theory. Thus, we used *the 6 C's*³ makes for excellent indepth reading on the matter as a starting point, and allowed for modifications throughout the *theoretical coding* phase.

A conceptual rendering of our generated theory of the *issues regarding version control, branching and merge operations* encountered is illustrated in Figure 1. The center bottom rectangle describes the core phenomenon, *version control, branch & merge issues*, while the other codes are represented by surrounding rectangles. Cause, correlation and effect are represented by arrows. Context is represented using dotted arrows. For each code a corresponding subsection is found below. Along with the analysis, the theory is detailed in Figure 2.

Throughout the analysis section we provide examples of 'quotes' from the data set. 'S'/'I' denotes interview subject and researcher respectively. We have added *emphasis* for *clarity* and occasional further clarifications within [hard brackets].

4.1. Phenomenon

Throughout our observations (field notes) and our questionnaires we noted that version control, branching and merge operations caused a disproportionate amount of loss in productivity and time. The questionnaires for all teams systematically indicated *version control, branching and merge conflicts* as the most disruptive challenges encountered throughout the project, and as a consequence this is the phenomenon we chose to explore.

³First author note – this was my first real attempt at 'pure' grounded theory. Having made another couple of attempts, to a varying degree of success, I make the casual observation that a 'grounded theory' consisting of more than 6-7 elements/constructs is very hard to explain to an audience. This is a perfect analogy to/example of Miller (1956). It is very hard for the human mind to process more than 6-7 visual elements simultaneously. So, if there are more constructs than the *magical number seven* I humbly suggest breaking up the theory in subsets. If not, it will be very hard to convey the message visually – and it will in all likelihood be very hard to push through peer review...

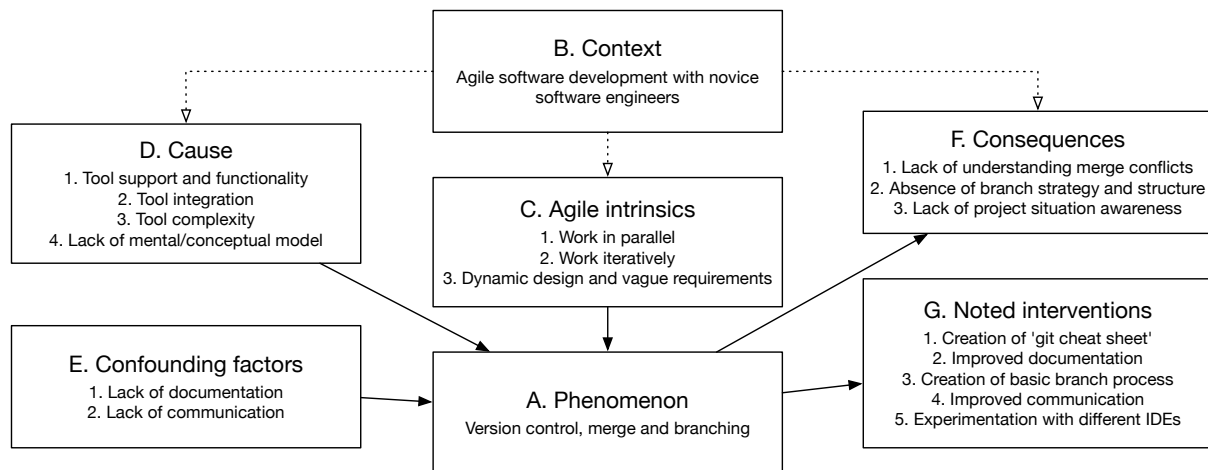


Figure 2 – Generated theory from Figure 1, further extended with the detailed codes from the analysis.

- E.g.: “Git/Merge – We are *unsure* of how to *use git properly*” (from student questionnaires – in response to what has been the biggest hurdle faced during the project).

We note that a *merge operation* in essence consists of a synthetical operation in which multiple sources/instances of software code is synthesised into a new instance, in an abductive process similar to what Walenstein (2002) describes as “the gulf of synthesis”.

4.2. Context

We define the context from which our observations are extracted, and in which they are valid, as that of agile software development teams, consisting of novices, using Git. Admittedly, this could result in a rather narrow validity window in terms of generalization. However, in our experience (both the first and second author has 15+ years experience of professional tool driven software development in large/distributed software projects) this observation, practitioners struggling with Git is commonplace in industry. Further, using novices as study objects would rather reveal cognitive challenges, as these challenges are not mitigated by *trained behavior*, *learning effect* or *status quo bias*.

We created our data set from observing and interacting with four different teams of *novice software developers* in parallel. All teams were using XP, and developed their system using the same basic stories/requirements (see Subsection 3.3 for details). While tool chain set up and development environment (IDE) differed somewhat between the teams, all teams used Git hosted by Bitbucket for version control (albeit with different branch strategies).

In light of the observed lacunae in extant software engineering literature, we note that version control from a user perspective is an area not thoroughly studied in the research community. Those few studies we found systematically indicate that our observations are valid in a wider context.

4.3. Agile intrinsics – root cause & driver

The iterative and parallel aspects of the nature of agile software development, *Agile intrinsics*, are from our observations, the identified underlying *Root cause* of the observed merge conflicts. In order to achieve some granularity we further break this construct up into three different subcodes: *(Work) in parallel*, *(Work) iteratively* and *Dynamic design and vague requirements*, since they are related in terms of root cause but have quite different consequences. As indicated by the *intrinsics* in the main category, these traits are inherent (largely by design) in the nature of agile software development. While these root causes could be compressed into one code, we feel that they are not interchangeable and each deserve a closer description. For further clarity we added an additional subcode, *Observed driver*, as means to further clarifying the underlying nature of these codes.

4.3.1. (Work) in parallel

Observed root cause: When starting up the project, the code base is very small, and different programming pairs are developing, and modifying the same code/classes/files, creating dependencies and diverging implementations ultimately leading to merge conflicts. While this to some extent was mitigated by adopting rudimentary branch strategies, the problem persisted throughout the projects.

We note that it appeared hard for the developers to find out *who did what, when and why?*, ultimately leading to a lack of understanding of implementation details, or a micro perspective, thus making subsequent merge conflicts harder to resolve. We also noted that this caused the developers to implement their own variations of similar methods (e.g. utility methods). E.g.:

- “Trying to merge code that someone else has written.”

(from student questionnaires and focus groups in response to what they found being difficult during their projects).

Observed driver: Diverging implementations leads to conflicting implementation details, further resulting in merge conflicts.

4.3.2. (Work) iteratively

Observed root cause: The iterative nature of the development results in constantly shifting implementation details and this subsequently drives merge work. The constant change in code leads to a lack of understanding from a micro perspective, reimplementing and duplication of code as different development pairs reimplement existing functionality. E.g.:

- “Parts of code *unknown*, having to interact with code that *someone else has written*, better after refactoring.”

(from student questionnaires and focus groups in response to what they found being difficult during their projects).

Observed driver: Refactoring implementation leads to changing implementation details, further resulting in merge conflicts.

4.3.3. Dynamic design and vague requirements

Observed root cause: Since there is no set architectural design/framework, nor a complete set of requirements or user stories, in the beginning of the projects, there was no cohesive collective goal for the developers. Further, architectural changes drives extensive refactoring and results in subsequent merge conflicts. Despite the fact that this is an inherent feature of XP – “XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements” (Beck, 1999) – it is nonetheless something we noted as a systematic cause of refactoring and merge conflicts. E.g.:

- “*Hard* to change data structures. This causes merge conflicts and bugs. Improve communication [within the team]?”

(from student questionnaires and focus groups in response to what they found being difficult during their projects).

Observed driver: No set design at the beginning of projects leads to refactoring of structure and changing of architecture, resulting in merge conflicts

4.4. Observed cause

This part of the analysis provides a reasoning on our observations on the observed causes of merge incidents.

4.4.1. The impact of tool support and tool functionality

Throughout the study we noted that the students were quite opinionated about functionality and support of the different tools and how well they were integrated. All teams started their respective projects using

Eclipse and Git hosted on Bitbucket. Out of the four teams, two ultimately migrated from Eclipse to another IDE; IntelliJ in one case and VSCode in one case.

When discussing tools during focus groups the importance of the user support the developers experienced became obvious. We noted that ease of use, intuitive interaction and visual support and offloading was something the students noted as very important in terms of reducing cognitive load. This is illustrated below in an excerpt from a focus group dialogue between three students (SI–SIII) and the interviewee (I):

SI: – “Many had problems *seeing* what changes that were being made, that is when you fetch; it might be related to Eclipse [integration with Git], it became better with VSCode, with the *colours* [indicating visual offloading], the *visual*, to be able to understand what has happened.”

I: – “But what *experience* have you had in regards to the *tool support* you have had in order to *solve merge conflicts*?”

SI: – “Eclipse was really *messy*.”

SII: – “...it was really hard to see *what* changes were coming from *where* – [extended pause, thinking] – and I think the *colours* in VSCode are really good [indicating visual support]. You really *see*, visually, *what is what*.”

SIII: – “...when we were using Eclipse we *switched to using the terminal* [use of Git through command line interface (CLI) instead of IDE integration] instead, it just feels a lot *easier*.”

(Focus Group, excerpt from video recording T@12.43).

The importance of visual support became even more obvious during saturation:

S: – “The merge support in VSCode is *graphical* and easy to understand – it is *intuitive*.”

S: – “The *merge support* in VSCode is very *clear*, it provides help on resolving the conflict, it *shows* <source code 1> and <source code 2> in the GUI and it is *simple* to choose by clicking a button.”

S: – “IntelliJ is actually, in my opinion, better than VSCode. It gives even better and *more visual* merge support.”

(Field notes – saturation)

4.4.2. Tool integration

We decided to further break down the analysis of the tool support further, in order to be able differentiate different angles of the experience of the students. We noted that the actual integration of Git in the IDEs was considered quite important, and a contributing factor when it came to changing IDE.

S: – “The *graphical integration of Git* in Eclipse is *difficult to understand*.”

S: – “Eclipse is *complicated* in terms of Git integration, and it is *easier to use* git through a *terminal* than through Eclipse.”

S: – “The *integration* between Git and VSCode is *superior* to that of Eclipse.”

(Field notes – saturation).

4.4.3. Tool complexity

The actual importance of tool complexity came to some surprise to the first author. We observed several reflections on the intricacies and complexities of Git in the dataset. We found compelling evidence that the complexity of Git was indeed a main cause of concern and cognitive load for novices, but the intricacies of Git was not the only cause of concern – the complexity of the IDE was also a definite issue and cause of confusion.

S: – “Version Control – Git is very *difficult*.”

S: – “What would make Eclipse better? Better *merge support* and better overview, making it *easier to find functionality*.”

S: – “VSCode feels *simpler*, with less functionality but it is a lot *less overwhelming*. It has a lot better learning curve.”

S: – “Eclipse is *complicated* and it is *difficult to understand* the structure.”

(Field notes – saturation).

Somewhat counter intuitively we also observed the following reflections on Git the command line interface:

- S: – “Git/CLI [in terminal] is good because it looks the same in every environment.”
- S: – “Git/CLI [terminal] is good because all Git online resources describe Git through CLI, so it is a lot easier to copy a line of commands and paste it into the terminal than to try do do the same thing through a GUI.”

(Field notes – saturation).

4.4.4. Lack of mental/conceptual model of version control and branch structure

Based on the outcome of the merge experiment (Subsection 3.8.3), which we considered a trivial Git/branch operation, we noted that the students’ understanding of reasonably straight forward branch operations in Git was somewhat limited. Out of the three groups that did the experiment (one team dropped out because of time constraints in their project), no one came up with a viable solution (albeit they came up with interesting and manually labour intensive ways to approach the task). At the end of the time-slot given for coming up with a solution, the first author provided a hint of the form “Well, maybe you should google *git squash* and *git cherry pick*?”. Subsequently all three teams adequately solved the exercise in a matter of minutes.

4.5. Confounding factors

4.5.1. Lack of documentation

We noted that a systematic lack of documentation (i.e. *code comments*, *commit messages*, *design documentation*) plagued the groups throughout their respective projects. This added to the lack of understanding the merge conflicts. We also noted that the students became aware of these aspects and, to a varying degree of success, tried to adress these issues at the later stages of their projects, see Subsection 4.7. Because of space limitations and the secondary nature of this code we have omitted any actual quotes, but the issues were systematic and affected all teams.

4.5.2. Lack of communication

We noted that a systematic lack of communication within the team (e.g. absence of *standup meetings* and use of *story boards*) plagued the groups throughout their respective projects. This added to the lack of understanding the merge conflicts as well as a lack of understanding the current project status. Further it added to *waste* and *loss of team productivity* when different pairs were working on the same task in parallel without knowing this. We also noted that the students became aware of these aspects and, to a varying degree of success, tried to address these issues at the later stages of their projects, see Subsection 4.7. Similar to the above, we omitted actual quotes, but the issues were systematic and affected all teams. One group started using Trello instead of a physical story wall, while the others continued using story walls.

4.6. Consequence

This part of the analysis provides a reasoning on our observations of consequences of the phenomenon under study.

4.6.1. Lack of understanding merge conflicts

The systematic lack of understanding of merge conflicts surprised us, and it became the focus of the analysis. These merge conflicts obviously lead to a loss of productivity, but it is not only limited to that. When going through the focus group material and the student reflections, we saw multiple examples of negatively loaded wording, indicating *fear*, *insecurity* and *stress*. We find this to be clear indicators that issues with merge conflicts not only cause a loss of productivity in terms of linear time, but also that the absence of the needed tool support causes considerable cognitive load and stress on the developers.

- S: – “It is *frightening* with a Wall of Text – merge conflict/difference [indicating a very complicated merge] when in reality there is only a minor difference in a character or so [e.g. trailing space etc.]. In VS code you see both versions and you can simply choose what code [snippet] you want.”

- S: – “You don’t know how to revert changes in Git you don’t know if you will *accidentally* [loss of control] replace/delete something [important]... you need to *dare* to use Git...”
- S: – “*uncertainty* results in many [of us] finding it *stressful* with merge conflicts... when there is a “merge message” that just appears you don’t really know what it means - will it result in overwrite - this makes it feel difficult, perhaps more so than it actually is...”

(from Field notes – saturation, questionnaires and focus group interaction).

4.6.2. Absence of branch strategy and structure

In addition to the systematic lack of understanding merge conflicts we also noted that branching itself was quite difficult for the teams. They had a hard time coming to grips with when to use separate branches (e.g. for bug fixes, tasks, stories and releases), when to close superfluous branches and branch naming conventions.

- S: – “It would have been better if we had used *story specific branches*.”
- S: – “We did not have a *strategy for branching* from the beginning [of the project].”
- S: – “We should have *closed branches* that were no longer in use.”

(from Field notes – saturation, questionnaires and focus group interaction).

4.6.3. Lack of project situation awareness

Further we noted that there were issues in regards to understanding the current project situation/status. This included multiple pair working on the same tasks, different pair implementing similar utility functions, a lack of understanding of components in the projects, and ultimately not knowing whom to ask about implementation details.

- S: – “Lack of communication – many of the problems we are facing would be solved if we would communicate better.”
- S: – ‘Architecture – attempts to communicate architecture changes during iteration without documentation resulted in a loss of micro perspective Only after an architecture spike was the issue finally resolved and understanding was shared.’
- S: – “People working on the same issue – sometimes people work with solving the same problems without knowing it/each other.”
- S: – “Lack of communication – this lead to several interesting issues during sprint III where we went in different directions regarding architecture.”

(from Field notes – saturation, questionnaires and focus group interaction).

4.7. Noted interventions

We here describe the interventions implemented by the different teams as means to circumvent the issues they encountered in their projects. On account of space limitations we omit the qualitative excerpts and keep the description short.

4.7.1. Creation of “Git cheat sheet”

We noted that the teams, after the first few sprints, realized that they needed a common manual for (and understanding of) basic Git operations. This was in most cases implemented as a *spike* by a pair of team members in between sprints. Further, we saw an interesting example of knowledge transfer within the team.

4.7.2. Improved documentation

We noted that the all teams throughout the project started realising the importance of documentation. The observed interventions included a systematic way of describing commits (i.e. pointing out what story or what task had been worked on, rather than the initial, rather void, messages like ‘*bugfix*’, ‘*gui implementation*’ etc.). We also noted that the teams started documenting the design of their architectures (using UML) and user interfaces (sketching on A3 paper). In addition we also noted that, while struggling with it in practice, all teams realised the importance of code documentation and made considerable attempts at documenting their code properly.

4.7.3. Creation of basic process for branch/cm/releases

We noted that all teams, after a few sprints started to develop a basic branch and configuration management process. This consisted of a more rigorous – less ad hoc – naming convention of branches, systematisation of main branch integration, and use of separate branches for stories, amongst other things. We do not consider the actual details as important as the observation that the teams, themselves, organically came to the conclusion that they needed a more systematic approach in regards to branching and configuration management. In addition we also noted that all teams, having experienced the value of explorative testing in the experiment presented in Subsection 3.8.2, started doing so well in advance of their releases.

4.7.4. Improved communication

We noted that all groups became aware of the need of improved communication. One team started using Trello as means of establishing a sound project overview. All teams further noticed the importance of standup meetings, and systematically started running more frequently.

4.7.5. Experimentation with different IDEs

As previously described we noted that two of the teams started exploring other IDEs in order to circumvent their perceived issues with Eclipse.

5. Literature review

5.1. Git & Merge

Our literature findings in regards to user experience of Git were surprisingly limited. What we could find was three relevant papers: Church, Soderberg, and Elango (2014), Perez De Rosso and Jackson (2013), and De Rosso and Jackson (2016).

We note that these papers, to some extent, validate our findings that Git is a very complex tool to use, and our conclusion is that there is considerable lacunae in literature in this regard. Future research should include a more thorough literature study in regards to Git and merge tools.

5.2. Eclipse and tool complexity

The issues related to tool complexity among novices are largely substantiated by extant literature – Moody (2009) discussed the different levels of support needed by novices and experts when it comes to visual languages based on Cognitive Fit Theory (Vessey, 1991) (Vessey & Galletta, 1991) (Shaft & Vessey, 2006). We can also see the same patterns in research on expertise by Chi et al. (1981) (2014). Further, Storey et al. (2003) as well as Rigby and Thompson (2005) have specifically described issues of novices in regards to Eclipse.

6. Ethical considerations

With the study focus on groups rather than individual students, there was no legal need for formal ethical hearing under the jurisdiction under which this research was conducted, however we did submit and register a description of the study to the local ethics board. As the course is graded *Pass* and *Fail* only, and the only way students to fail is by considerable absence, we felt that there was no major issue with conflicting roles of researcher/teacher for the first author. In addition, our presence during sprints and planning sessions allowed the student groups more teacher time than what they would have experienced otherwise. Further, we stress again that all students were systematically offered to retire themselves from the groups being observed.

Liebel and Chakraborty (2021), present an updated mapping study on ethical issues in empirical software engineering studies using students, and highlight that study conditions and power relations between students and instructors are special areas of concern. We would like to stress that the consent to enter the study was informed, and we feel that we presented all students with the systematic ability to retire from the research, that we have been transparent with the conditions and that the power relations were, in reality, unaffected by the study condition since the role of the researchers were to act as customers in the actual projects.

The experiments we exposed the students to had been used as improvised *project disturbances* embedded in the course design by first author in previous years and appeared to make a sound addition to the learning outcome of the students. Based on the fact that the learning outcome of the students was not compromised, that all data was collected with consent and anonymously, that the findings will benefit the students of the next instantiation of the course and the very high course evaluation grades the students awarded the course post completion, we do not feel that we have any ethical qualms in regards to the study.

7. Threats to validity

The use of students as basis for research can be controversial (Höst, Regnell, & Wohlin, 2000) (Svahnberg, Aurum, & Wohlin, 2008) (Henrich, Heine, & Norenzayan, 2010) from a generalisation perspective as well as from student integrity and learning perspectives. In terms of generalisation, Höst et al. highlight that students working under life-like circumstances serve can function as a reasonable proxy for real life settings/practitioners (Höst et al., 2005). In this study we selected students to capture a *novice point of view*, thus providing us with a different perspective of causes of cognitive load drivers. Further, by acting as customers on site we were able to take a participatory observation position allowing us to some extent ‘blend’, while retaining an analytical ethnographic stance (Sharp et al., 2016).

In addition to discussing ethical dimensions of software engineering carried out on student populations, Liebel and Chakraborty (2021) also discuss the scientific value of such studies. Highlighting that research conducted using qualitative methodologies such as *case studies*, *observational studies* and *ethnography* the actual *case context* is a “deciding factor” and therefore cannot generally be separated from the “studied phenomenon”. That being said, Stol and Fitzgerald (2018) highlight the value of knowledge seeking research approaches such as ethnography using the work by Sharp and Robinson (2004) as an example of such research.

Somewhat tounge-in-cheek (and not drilling into the taxonomy of elephants, which is extensive and contextually important), we respond to the elephant/jungle metaphors provided by Stol and Fitzgerald (2018) by stating that if you want to study juvenile elephants ridding themselves of bug infestation, and the methodologies deployed in such an activity you probably want to do it in the jungle⁴. If you, on the other hand, want to observe software development teams consisting of juniors solving software development issues, a computer hall at a university is, probably, about as ideal (and actually natural) as research environments come.

The procedures for the planning, data collection and analysis are reported in detail in Section 3.

GT studies are commonly evaluated based on the following criteria (Charmaz, 2014) (Stol et al., 2016):

Credibility: *Is there enough data to merit claims of the study?* – This study relies on the data set from one case study. The data set includes interviews, focus groups, observations and written reflections. The data set is quite extensive.

Originality: *Does the results offer new insight?* – While cognitive load is not an unknown phenomenon in software engineering, we note that merge operations seem disproportionately troublesome/difficult. We note a *research gap* when it comes research on version control and merge operations.

Usefulness: *Is the theory generated relevant for practitioners?* – This study generates a theory that offers one explanation of how merge operations and branch work becomes difficult in projects. This can be used for reasoning on cognitive load in software engineering. The main contribution, in our opinion, is the observation of merge phenomenon and version control issues and the corresponding *research gap*.

Resonance: *Does the theory generated resonate among participants/informants?* – While the final rounds of data collection was shut down prematurely on account on the pandemic situation, we were

⁴Technically, we would, for (obvious) visibility and (equally obvious) safety reasons, prefer open plains rather than the jungle for observational studies. We will however not push the elephant metaphor further beyond the casual observation that while novice software developers might charge, they are far less likely to kill you....

fortunate to gain access to four of students (roughly 10 percent of the population) whom participated in the study. These accepted to join a small follow up session in order to allow us to gauge the resonance. This session was held in a focus group format, and the students had prior to this: a) participated a second time in the course this time acting as 'coaches', and b) read a previous version of this paper including the theory. They found the paper and the theory to be a sound description of what had transpired, and noted that the measures taken by the teachers in order to change the course was very beneficial for the students.

We take a pragmatist (Bryant, 2017) epistemological position in this paper. Our aim is to provide a grounded theory for reasoning on cognitive load in software engineering, using abductive reasoning on literature and data, and our ambition is to provide knowledge for software engineering research community and practitioners. We use grounded theory as a method, not an epistemological position. We acknowledge that all qualitative knowledge is inherently constructed, but we are studying the sometimes fairly gritty surface between limits of the human mind and software development tooling through means of qualitative inquiry.

With that said, the phenomena we study do arguably exist, albeit in an artificial context largely unbound by natural laws. If the phenomena did not exist, there would be little point in studying them, nor their consequences on the human mind. So, just as with Bryant (2017) we want to close the door on relativisation.

8. Discussion and future work

The findings in relation to the first research goal, *to identify the most common cognitive load driver from the novice point of view*, was somewhat surprising. While we build our work on previous identification of the temporal perspective (Helgesson et al., 2019), the *who, did what, when & why*, we were quite surprised to see how large the impact of version control and merge operations were on the students. We also find it interesting to see the importance of tool support and functionality, tool integration and tool complexity in agile software development. To us the most interesting observation is the importance of visual merge support. We also noted that absence of communication and documentation was a contributing and confounding factor. We also note the absence of research on version control as an indicator for further research.

In addition to the codes described in our theory, we also noted other indications of cognitive load drivers in the material. The environment, in terms of ventilation and loud ambience was lamented on, the work situation was described as *draining*. Further we also noted disruptions and task switching as a cause of concern – described as a *disruption of flow*.

We noted that distributed cognition, from our perspective, is indeed a sound perspective for observing and analysing software development in agile teams, and it is further interesting to note the reflections of *history enriched objects* and the corresponding *temporal* cognitive dimension made by Hollan et al. (2000) alongside our findings on version control and merge operations. Future theory building and theorization based will include constructs describing distributed software development as a 'distributed cognitive production flow' and further explore the observed synthetical nature of merge operations.

In regards to our second research goal, to chart what *differences* or *similarities* that can be observed between the different group compositions, we noted that there were indeed observable, yet subtle differences between the different groups. With that said, during the field work we realised that the *differences* we could observe, to us, were significantly less interesting than the *similarities* we could observe. As a consequence we choose to use these similarities to strengthen the internal validity of our findings.

Following the 2020 iteration of the course during which these observations were made, the teachers working in the course had a few discussions on suitable interventions that could be extracted from the course. We added a more thorough introduction to Git and some harder actual hands on exercises as preparations for the 2021 course iteration. We further introduced more tool support to the students.

While the Covid-19 situation has forced us to teach the course via Zoom and arguably made the whole course (that depends on teamwork) considerably more difficult we systematically noted that the students were suffering less from version control issues and were actually appearing to be more productive than previous years. For obvious reasons the pandemic situation prevented us from doing a more thorough follow up in the field.

Acknowledgement

The authors wish to thank all the participating students for their invaluable contributions, as well as course responsables for providing the opportunity to conduct the study. We further thank Softhouse⁵ for providing time for the second author and the PPIG reviewers, and audience, for valuable feedback.. The work described in this paper was conducted in the ELLIIT⁶ strategic research environment.

9. References

- Abend, G. (2008, June). The Meaning of ‘Theory’. *Sociological Theory*, 26(2), 173–199. Retrieved 2021-05-19, from <https://doi.org/10.1111/j.1467-9558.2008.00324.x> (Publisher: SAGE Publications Inc) doi: 10.1111/j.1467-9558.2008.00324.x
- Beck, K. (1999). *Extreme programming explained: embrace change*. USA: Addison-Wesley Longman Publishing Co., Inc.
- Begum, M. (2021). Cognition and Distributed Cognition in Software Engineering Research[WIP]. In (p. 8). Psychology in Programming Interest Group.
- Bertelsen, O. (1997, November). Toward A Unified Field Of SE Research And Practice. *IEEE Software*, 14(6), 87–88. doi: 10.1109/MS.1997.636682
- Blackwell, A. F., Petre, M., & Church, L. (2019, November). Fifty years of the psychology of programming. *International Journal of Human-Computer Studies*, 131, 52–63. Retrieved 2019-12-02, from <http://www.sciencedirect.com/science/article/pii/S1071581919300795> doi: 10.1016/j.ijhcs.2019.06.009
- Bryant, A. (2017). *Grounded Theory and Grounded Theorizing – Pragmatism in Research Practice*. Oxford, UK: Oxford University Press.
- Buchan, J., Zowghi, D., & Bano, M. (2020). Applying Distributed Cognition Theory to Agile Requirements Engineering. In N. Madhavji, L. Pasquale, A. Ferrari, & S. Gnesi (Eds.), *Requirements Engineering: Foundation for Software Quality* (pp. 186–202). Cham: Springer International Publishing. doi: 10.1007/978-3-030-44429-7_14
- Charmaz, K. (2014). *Constructing Grounded Theory* (2nd ed.). London, UK: SAGE Publications.
- Charmaz, K., & Mitchell, R. (2001). Grounded Theory in Ethnography. In *Handbook of Ethnography*. London, UK: SAGE Publications.
- Chi, M. T. H., Glaser, R., & Farr, M. J. (2014). *The Nature of Expertise*. Psychology Press. doi: 10.4324/9781315799681
- Chi, M. T. H., Glaser, R., & Rees, E. (1981, May). *Expertise in Problem Solving*. (Tech. Rep. No. TR-5). Pittsburg Univ PA Learning Research and Development Center.
- Church, L., Soderberg, E., & Elango, E. (2014, June). A case of computational thinking: The subtle effect of hidden dependencies on the user experience of version control. In B. du Boulay & J. Good (Eds.), *Proceedings of psychology of programming interest group annual conference* (p. 123-128). Brighton, United Kingdom.
- De Rosso, S. P., & Jackson, D. (2016). Purposes, Concepts, Misfits, and a Redesign of Git. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (pp. 292–310). New York, NY, USA: ACM. doi: 10.1145/2983990.2984018
- Flor, N. V., & Hutchins, E. L. (1991). Analyzing distributed cognition in software teams: a case study of team programming during perfective maintenance. In J. Koenemann-Belliveau, T. G. Moher,

⁵<https://www.softhouse.se>

⁶<https://liu.se/elliit>

- & S. P. Robertson (Eds.), *Proceedings of Empirical Studies of Programmers* (p. 36-64). Norwood, NJ, USA: Ablex Publishing Corporation.
- Glaser, B. G. (1978). *Theoretical Sensitivity*. CA, USA: Sociology Press.
- Glaser, B. G. (1992). *Emergence vs Forcing - Basics of Grounded Theory Analysis*. CA, USA: Sociology Press.
- Glaser, B. G., & Strauss, A. L. (1967). *The Discovery of Grounded Theory*. New Jersey, USA: Aldine-Transaction.
- Hansen, S., & Lyytinen, K. (2009, August). Distributed Cognition in the Management of Design Requirements distributed cognition in the management of design requirements. In R. C. Nickerson & R. Sharda (Eds.), *Proceedings of the 15th Americas conference on information systems* (p. 266). San Francisco, California, USA.
- Hedin, G., Bendix, L., & Magnusson, B. (2005, January). Teaching extreme programming to large groups of students. *Journal of Systems and Software*, 74(2), 133–146. doi: 10.1016/j.jss.2003.09.026
- Helgesson, D., Engström, E., Runeson, P., & Bjarnason, E. (2019). Cognitive Load Drivers in Large Scale Software Development. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering* (pp. 91–94). Piscataway, NJ, USA: IEEE Press. doi: 10.1109/CHASE.2019.00030
- Helgesson, D., & Runeson, P. (2021). Towards grounded theory perspectives of cognitive load in software engineering. In *Ppig 2021*.
- Henrich, J., Heine, S. J., & Norenzayan, A. (2010, June). The weirdest people in the world? *Behavioral and Brain Sciences*, 33(2-3), 61–83. doi: 10.1017/S0140525X0999152X
- Hollan, J., Hutchins, E., & Kirsh, D. (2000, June). Distributed Cognition: Toward a New Foundation for Human-computer Interaction Research. *ACM Trans. Comput.-Hum. Interact.*, 7(2), 174–196. doi: 10.1145/353485.353487
- Höst, M., Regnell, B., & Wohlin, C. (2000, November). Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, 5(3), 201–214. doi: 10.1023/A:1026586415054
- Höst, M., Wohlin, C., & Thelin, T. (2005, May). Experimental context classification: incentives and experience of subjects. In *Proceedings of the 27th international conference on Software engineering* (pp. 470–478). St. Louis, MO, USA: Association for Computing Machinery. doi: 10.1145/1062455.1062539
- Hutchins, E. (1995). *Cognition in the Wild*. MIT Press.
- Lenberg, P., Feldt, R., & Wallgren, L. G. (2015, September). Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107, 15–37. doi: 10.1016/j.jss.2015.04.084
- Liebel, G., & Chakraborty, S. (2021, March). Ethical issues in empirical studies using student subjects: Re-visiting practices and perceptions. *Empirical Software Engineering*, 26(3), 40. Retrieved 2021-05-10, from <https://doi.org/10.1007/s10664-021-09958-4> doi: 10.1007/s10664-021-09958-4
- Mangalaraj, G., Nerur, S., Mahapatra, R., & Price, K. H. (2014, March). Distributed Cognition in Software Design: An Experimental Investigation of the Role of Design Patterns and Collaboration. *MIS Quarterly*, 38(1), 249–A5.
- Miller, G. A. (1956). The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2), 81–97. doi: 10.1037/h0043158
- Moody, D. (2009, November). The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6), 756–779. doi: 10.1109/TSE.2009.67
- Perez De Rosso, S., & Jackson, D. (2013). What’s Wrong with Git?: A Conceptual Design Analysis. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (pp. 37–52). New York, NY, USA: ACM. doi: 10.1145/

- Ramasubbu, N., Kemerer, C. F., & Hong, J. (2012, September). Structural Complexity and Programmer Team Strategy: An Experimental Test. *IEEE Transactions on Software Engineering*, 38(5), 1054–1068. doi: 10.1109/TSE.2011.88
- Rigby, P. C., & Thompson, S. (2005, October). Study of novice programmers using Eclipse and Gild. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* (pp. 105–109). San Diego, California: Association for Computing Machinery. doi: 10.1145/1117696.1117718
- Runeson, P., Höst, M., Rainer, A., & Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons.
- Sedano, T., Ralph, P., & Péraire, C. (2017, May). Software Development Waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (pp. 130–140). doi: 10.1109/ICSE.2017.20
- Shaft, T. M., & Vessey, I. (2006). The Role of Cognitive Fit in the Relationship between Software Comprehension and Modification. *MIS Quarterly*, 30(1), 29–55. doi: 10.2307/25148716
- Sharp, H., Dittrich, Y., & de Souza, C. R. B. (2016, August). The Role of Ethnographic Studies in Empirical Software Engineering. *IEEE Transactions on Software Engineering*, 42(8), 786–804. doi: 10.1109/TSE.2016.2519887
- Sharp, H., Giuffrida, R., & Melnik, G. (2012, May). Information Flow within a Dispersed Agile Team: A Distributed Cognition Perspective. In *Agile Processes in Software Engineering and Extreme Programming* (pp. 62–76). Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-30350-0_5
- Sharp, H., & Robinson, H. (2004, December). An Ethnographic Study of XP Practice. *Empirical Software Engineering*, 9(4), 353–375. doi: 10.1023/B:EMSE.0000039884.79385.54
- Sharp, H., & Robinson, H. (2006, June). A Distributed Cognition Account of Mature XP Teams. In *Extreme Programming and Agile Processes in Software Engineering* (pp. 1–10). Springer, Berlin, Heidelberg. doi: 10.1007/11774129_1
- Sharp, H., & Robinson, H. (2008, July). Collaboration and co-ordination in mature eXtreme programming teams. *International Journal of Human-Computer Studies*, 66(7), 506–518. doi: 10.1016/j.ijhcs.2007.10.004
- Sharp, H., Robinson, H., & Petre, M. (2009, January). The role of physical artefacts in agile software development: Two complementary perspectives. *Interacting with Computers*, 21(1-2), 108–116. doi: 10.1016/j.intcom.2008.10.006
- Sharp, H., Robinson, H., Segal, J., & Furniss, D. (2006, July). The role of story cards and the wall in XP teams: a distributed cognition perspective. In *AGILE 2006 (AGILE'06)* (pp. 11 pp.–75). doi: 10.1109/AGILE.2006.56
- Stol, K.-J., & Fitzgerald, B. (2018, September). The ABC of Software Engineering Research. *ACM Trans. Softw. Eng. Methodol.*, 27(3), 11:1–11:51. doi: 10.1145/3241743
- Stol, K.-J., Ralph, P., & Fitzgerald, B. (2016, May). Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 120–131). doi: 10.1145/2884781.2884833
- Storey, M.-A., Damian, D., Michaud, J., Myers, D., Mindel, M., German, D., . . . Hargreaves, E. (2003, October). Improving the usability of Eclipse for novice programmers. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange* (pp. 35–39). Anaheim, California: Association for Computing Machinery. doi: 10.1145/965660.965668
- Svahnberg, M., Aurum, A., & Wohlin, C. (2008). Using Students As Subjects - an Empirical Evaluation. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 288–290). New York, NY, USA: ACM. doi: 10.1145/1414004.1414055
- Thornberg, R., & Charmaz, K. (2014). Grounded theory and theoretical coding. In *The SAGE Handbook of qualitative data analysis*. London, UK: SAGE Publications.
- Vessey, I. (1991). Cognitive Fit: A Theory-Based Analysis of the Graphs Versus Tables Literature*. *Decision Sciences*, 22(2), 219–240. doi: 10.1111/j.1540-5915.1991.tb00344.x

- Vessey, I., & Galletta, D. (1991, March). Cognitive Fit: An Empirical Study of Information Acquisition. *Information Systems Research*, 2(1), 63–84. doi: 10.1287/isre.2.1.63
- Walenstein, A. (2002). *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework* (Unpublished doctoral dissertation). School of Computing Science, Simon Fraser University.
- Zaina, L. A. M., Sharp, H., & Barroca, L. (2021, March). UX information in the daily work of an agile team: A distributed cognition analysis. *International Journal of Human-Computer Studies*, 147, 102574. Retrieved 2021-03-04, from <https://www.sciencedirect.com/science/article/pii/S1071581920301762> doi: 10.1016/j.ijhcs.2020.102574

Livecode me: Live coding practice and multimodal experience

Georgios Diapoulis

Interaction Design

Chalmers University of Technology,

University of Gothenburg

georgios.diapoulis@chalmers.se

Abstract

I present a practice-led research design to explore relations between listening and non-listening conditions during a month-long live coding practice. A documentation of the live coding sessions and textual data from my daily diaries are presented in a git repository. The study offers a set of observations related to musical and programming practices, an ongoing work on a visual helper and outlines issues related to solo live coding practice.

1. Introduction

A central component of musicianship is diligent practice. Musical practice is a daily activity that musicians do, and the same applies to live coding. Musical live coding uses on-the-fly computer programs to generate sound patterns (Collins, McLean, Rohrhuber, & Ward, 2003). It can also be seen as an approach to experience composition while composing musical outcomes (Sorensen & Brown, 2007), and it is an improvisational practice. A significant difference between learning how to live code and learning how to master composition or instrumental music performance is that there is no school to learn how to live code, except a wiki page on TOPLAP website ¹. Learning by example and trial-and-error problem-solving technique is at the heart of every live coding practice. Click Nilson, a persona of Nick Collins, contributed the first study on live coding practice and proposed a set of exercises for improving coding and musical expertise (Nilson, 2007). He addressed these topics by consulting developmental and educational psychology literature studies and carried out a month-long daily practice with Fredrik Olofsson.

My perspective was to carry out daily practice sessions, having in my mind how a novice user would explore live coding. The primary motivation was to improve my musical live coding skills and explore the role of music listening during live coding practice. To explore the role of listening in the context of multimodal perception, I conducted daily sessions with listening and without listening to the generated sounds. The latter was done by muting the soundcard's audio output to the speakers. Listening to the sound while coding is an indispensable part of live coding practice (A. F. Blackwell & Collins, 2005). I posed the question, what if the live coder does not listen to the sound? Thus, I experienced whether listening to the generated sound patterns may help me to understand the written code and benefit my live coding practice.

For the present study, I carried out daily solo live coding sessions. At times it felt like a conversation with myself (Gamboa, 2022), but also a familiar thing to do as I have been practising musical instruments for many years. For the live coding sessions I used SuperCollider ², a programming environment for sound synthesis and algorithmic composition, and documented temporally accurate representations of the coding sessions. Every day I conducted both listening and non-listening sessions, and I wrote short diary entries at the end of the day, sometimes listening the sessions afterwards by replaying the code recording.

Multimodal experience in live coding studies has been approached from an audience perspective (Burland & McLean, 2016). In this article, I focus on the subjective experience of the live coder. Contrary to Burland and McLean (2016), who discuss how audio-visual information facilitates audience

¹https://toplap.org/wiki/Live_Coding_Practice

²supercollider.github.io/

aesthetic experience, my focus was how audio-visual information could be useful for the performer. Thus, instead of focusing on audience appreciation and enjoyment, I focus on the relation between the auditory and visual percepts of the live coder. How can I get informed when I do not listen to the musical outcome? For that purpose, I explored how visual information from the audio spectrum analyzer, a visual representation of the magnitude of a signal as a function of frequency, can be helpful for the performer.

This study is focused on reproducing the methodology of the seminal practice sessions by Collins and Olofsson and is also influenced by the methodological approach presented by Blackwell and Aaron (2015). It is a practice-led research approach, which can be seen as an extended concept of research through design. By focusing on the four elements by Blackwell and Aaron (i.e., identifying design exemplars, critical orientation, exploratory implementation, and reflective assessment), I contribute a code repository of the coding sessions and reflections related to musical and programming practices, multimodal perception and general issues related to the activity of practising live coding alone.

2. The practice of musical live coding

The computer can be seen as the 'natural tool' of electronic music (Nilson, 2007). During a live coding session, the code is translated to musical outcomes, and the musicians constantly listen to the generated sounds (A. F. Blackwell & Collins, 2005). Collins presented a month-long live coding practice session along with Fredrik Olofsson. The practice sessions were conducted in SuperCollider, and the documentation of the self-administered daily sessions of unaccompanied solo practice is available online (<https://swiki.hfbk-hamburg.de/MusicTechnology/819>). During this month-long practice, the two live coders had different approaches. Collins had a daily plan and exercised specific algorithmic problems, like the $3x + 1$ problem, whereas Olofsson did not have a specific plan per day. The sessions were 'blank slate', meaning there was no pre-written code to be executed. The goal of both coders was to practice one hour per day. That is, one hour-long practice sessions simulating a performance setting. Collins sums up his contributions on three main aspects: (i) isolation exercises, (ii) connectivity exercises, and (iii) repertoire implications. Isolation exercises are activities that a live coder can carry out alone and do not necessarily relate to musical practice. Examples include practising fast typing or solving mathematical problems. Connectivity exercises are music-related and address issues the live coder has to confront during a live session. These connectivity exercises may include controlling musical tension, mixing audio signals, and so on. The third aspect of, what I call, repertoire implications addresses issues such as code sharing in laptop ensembles, among others.

Sorensen and Brown (2007) report five computational techniques used during live coding practice. These techniques are generic and can be extended to different aspects of electronic music, like using probability functions, periodic functions and modulo arithmetics, among others. They elaborate on the programming practices during live coding, such as code expansion, function abstractions and keyboard shortcuts. Collaboration and communication are two live coding practices that are also essential. Collaborative live coding has also been a risk management technique (Roberts & Wakefield, 2018). In my month-long practice, I did not do any collaborative sessions. On the communication aspect, I communicated my daily practices with various people, from academics to practitioners and non-musicians. However, I feel I should have been more systematic in this.

Magnusson (2015) explored notational aspects of live coding practice. Magnusson identifies a difference between prescriptive and descriptive aspects of visual notations and offers a solution to the problem of making a connection between code representations and temporal representations of musical events. Another approach to visualisation of musical forms have been recently presented (Dal Rì & Masu, 2022), either as a linear temporal evolution or as clusters showing event density per family of sounds. The authors suggest that these two visualization techniques can be complementary to each other, and reflect on issues related to attention span between coding and visual representations of musical form.

2.1. Psychological aspects of live coding practice

The present study offers a perspective in which the user's task is unlike the 'normal model' of live coding practices (A. F. Blackwell & Collins, 2005). The normal model of musical live coding involves listening to the sound while simulating a performance setting. Still, I conducted at least one session daily without listening to any sounds. Consequently, it contributes to the literature with a use case where the user's needs are unlike the needs of a live coder. In that manner, I present an unusual case of live coding practice that is unlike previous research in the field. The outcome of this study is compiled by examining the reflective diaries, and I discuss how the non-listening condition can benefit or hinder a live coding session.

3. Design of the study

The design of the study is simulating a performance setting, similar to Collins and Olofsson. By performance setting, I refer to a continuous live set with a predetermined minimum duration and a continuous evolution of sound patterns. There was no systematic preparation before each session, and in most cases, no preparation. All practice sessions were 'blank slate' live coding, that is, with no use of pre-written code. That was my main design exemplar, and I conducted a pilot and an experiment proper (A. F. Blackwell & Aaron, 2015). The study was conducted in two parts, a pilot study took place in August 2022 and a proper one in October 2022. A series of 12 daily live coding sessions were carried out during August, and a month-long daily practice was carried out in October. The experimental designs were different as an outcome of the post-proceedings policy of the PPIG conference.

Part of the critical orientation (A. F. Blackwell & Aaron, 2015) is reflected in the experimental design which was influenced by the seminal study of Davidson (1993) on the perception of expressive performance. This critical orientation led me to question how music listening is useful to the performer. Davidson (1993) designed a study in which she assigned expressive manners to violinists and recorded both video and audio recordings. Later, for validating the expressive manners Davidson conducted a perceptual experimental showing stimuli of full-body movement from the violin performances. Part of the methodological novelty of the study was that the stimuli were based either on visual information only, audio information only, or showing both audio-visual information from the violin performances. In that manner, Davidson controlled the perceptual channels demonstrating that the visual channel can bias our perception of expressivity in music performance. Here, I applied a similar orientation and designed a listening and a non-listening condition, but without assigning any expressive manners or specific tasks during my practice.

During the pilot practice sessions in August, I did three daily sessions of 500 seconds each (36 independent sessions in total). This is approximately 8 minutes, considered a short duration for a live coding performance. The three conditions of the August daily sessions are coded in the file names of the practice sessions as 'No-Audio Level meter' (NAL), 'No-Audio Spectrum' (NAS), and 'Audio-Visual' (AV) conditions. Both NAL and NAS sessions did not include any audio, as I muted the audio from the sound card to the loudspeakers. The NAL sessions were conducted without audio from the loudspeakers, and the only informative cues about the generated sound patterns came from a stereo sound level meter (see Figure 1). The NAS sessions were also conducted without any auditory cues, and the visual cues included both a sound level meter and a spectrum (see Figure 2). During the AV sessions, I could listen to the generated sound patterns and see the visual cues coming out of the sound level meter and the spectrum (see Figure 2).

During the pilot study in August, I focused on a controlled experimental design, and I also aimed to control the listening and acoustic conditions. By controlled conditions, I refer to designing a daily plan about the order of the listening conditions (NAL, NAS, AV), but also to control the sound levels from the computer to the listener. In that manner, I used a MacBook Pro laptop, reproducing the generated sounds from the built-in loudspeakers on maximum levels. I realised that this experimental setup and a controlled experimental setup could be challenging to provide fruitful results for my study. After the PPIG conference and my presentation, I received valuable feedback, which made me redesign the ex-

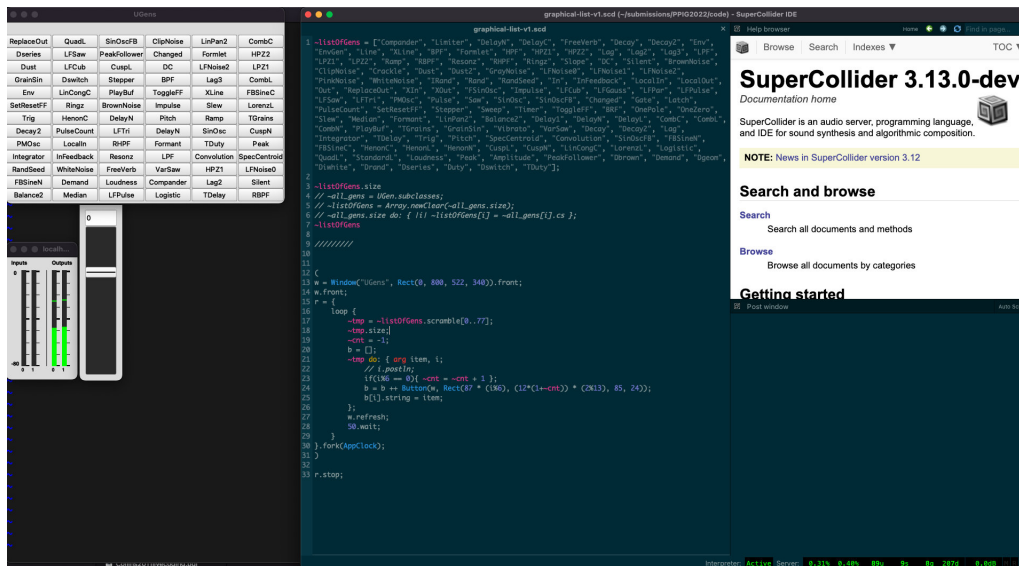


Figure 1 – Pilot experimental setup in SuperCollider for the NAL condition.

perimental setup. The main change was to discard the NAL sessions, as I found them uninformative and maybe annoying. That phase can be seen as the exploratory implementation (A. F. Blackwell & Aaron, 2015). During the experiment proper in October, I did not attempt to control the acoustic conditions. Thus, when I refer to listening to the sound output, I may interchangeably refer to listening from loudspeakers or headphones.

During the month-long experiment in October, I conducted two sessions (AV and NAS) per day of at least 1500 seconds each, by swapping the order of the first and second session every day. This is equivalent to two 25 minutes sessions per day, which is considered an adequate duration for a live coding performance. I manually monitored the duration of every session, and I did not set a hard limit on finishing the sessions. Contrary, during the August sessions, I had programmatically set a hard limit to finalise the sessions after the 500 seconds time limit. For the pilot study in August, every session was catalogued with an audio recording and a timestamped text file showing every code execution. For the experiment proper in October, the audio recordings were not conducted, mainly due to large allocations of memory storage.

The code was captured using the `History` class of SuperCollider. This class can provide reproducible live coding sessions in terms of code executions and audio output. The collected data are timestamped scripts, in the format of plain text files (`*.scd` files), for SuperCollider.

Short-length reflections were written about the daily sessions. The length varies but is no more than 200-300 words per day. Some daily diaries are as short as 1-2 sentences, especially during the October sessions. It is difficult to distinguish whether they can be considered reflective diary entries or note-taking prompts. Complete documentation of the code and the diaries is provided online (<https://gitlab.com/diapoulis/livecodeme/>).

3.1. Visual helper to aid creativity

Part of the experimental design includes a graphical interface (GUI) as a visual helper, as shown in the top-left of Figure 1 and Figure 2. It is a simple helper device that emulates using Post-it notes on the screen (A. Blackwell & Green, 2003). My motivation was to have some visual aid that shows various unit generators (UGens), and get inspiration when building sound synthesis engines. A UGen is the basic building block for sound synthesis engines in many programming languages. SuperCollider has a plethora of UGens, either in the main library or as third-party developments, which makes difficult to recall each one of them. My idea was to begin from a visual helper which could potentially be developed into a software agent. As shown in Figure 1, the initial implementation was done by simply

randomising a list of UGens and printing them on a GUI. Initially, I used a dense matrix format, with dimensions 13x6, refreshing the GUI every 50 seconds. During the pilot study, I selected a constrained set of UGens to be used across the study. The initial list had 128 entries, and I manually selected basic signal generators (like sine and noise oscillators), routing UGens, filters, triggers and envelopes. I found this family of UGens to be limiting my coding practice.

During the study, the GUI did not change significantly. I kept its main passive functionality, and I reduced drastically the amount of UGens shown at a time. During the experiment proper the GUI was showing 8 UGens at a time, and it was updated every 30 seconds (see Figure 2). Some preliminary research on how this can be developed into a software agent was done and is discussed across the diaries. Some of the early conclusions came out of the pilot study, which was to transform the GUI to a 'disruptive' software agent (Attanayake, Swift, Gardner, & Sorensen, 2020), that induces or simply replaces code segments with UGens of similar functionality. This part of the study is still a work in progress.

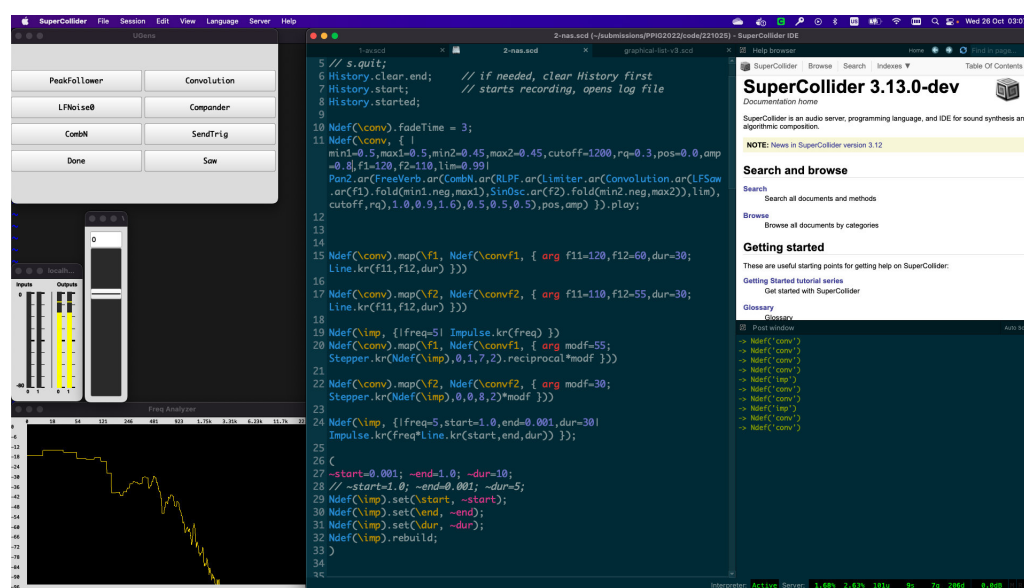


Figure 2 – Experiment proper setup in SuperCollider for the NAS and AV conditions.

3.2. Overall comments on the experimental setup

After my initial attempt to do a controlled study, I feel that I shifted to the other end of having no constraints set in place. Initially, I was inspired from the approach by Fredrik Olofsson while doing the month-long practice sessions with Nick Collins, and did not have a specific coding daily plan. Later I realised that Fredrik Olofsson had also imposed a set of constraints to another practice session that conducted with Marcus Fjellström³. In these sessions Olofsson used a single synth definition across all sessions. That is radically different from my experimental design, where I experimented with coding synth representations for sequencing, sound generation, control structures, machine listening among others.

4. Looking into the sound

The design of the study affords an alternative view to live coding. While playing music without listening to it sounds like a no-go, this is the main contribution of this study. Thus, I here present a reflective assessment compiled from the daily diary entries (A. F. Blackwell & Aaron, 2015).

4.1. Reflections from experience

It goes without saying that musical live coding without listening to the sound can hardly be a rewarding activity. Musicians use sound as the primary informative cue during a performance, and I have never

³<https://fredrikolofsson.com/f0blog/pact-april/>

attended a concert where the performers intentionally choose not to listen to the sound. During my non-listening practice sessions, I was sometimes curious to hear what the musical outcome sounded like, and I did sometimes unmute momentarily during the pilot study. During the experiment proper, I did not unmute the non-listening condition, instead, I listened to some of the code recordings after the end of the sessions. It is out of the question that when you live code and do not listen to the sound there are limited possibilities of what can be perceived from the musical outcome. For instance, if I program a sine oscillator and assign a specific frequency, I need perfect pitch to be able to 'hear' how the outcome may sound like. All in all, the only informative cue when non-listening to the sound can be reconstructed by imaging the musical outcome. Musical imagery is a valuable musical ability but it may be a less engaging activity in comparison to music listening. That is, we can be really engaged when listening to the music, and this may be expressed with overt bodily movements, such as dancing, whereas it is less likely to start dancing when imagining a melody of a song.

While live coding does not include any direct involvement between the sound energy and the performed actions, when listening to the sound we may be able to simulate sound actions (Jensenius, 2007, p. 19). Even if there is no direct link to any profound sound actions from daily experience or if the sounds are completely synthetic, we are still able to dance to the beat and produce synchronised bodily movements. The same cannot apply to visual percepts, as we generally perform better in audio-motor synchronisation tasks in comparison to visuo-motor tasks (Hove, Fairhurst, Kotz, & Keller, 2013). Thus, there is an embodied understanding that makes the sound and vibration an indispensable part of music making.

4.2. Reflections from the diaries

In the diaries, I found mixed feelings about the non-listening condition. It looks like I both appreciated some surprisingly appealing musical outcomes but was also annoyed when I had no idea what this may sound like. Several musical tasks are hard to do when non-listening to the sound. I found out that I often forget to apply any panning in non-listening conditions. In principle, many connectivity exercises (Nilson, 2007) can be almost impossible to control when non-listening to the sound. Certain aspects like having a percept about the tempo and the beat are impossible to extract by the visual information on the spectrum, during the non-listening condition. Furthermore, it is quite impossible to track for how long a specific musical pattern is active and produces repetitions, and difficult to understand the length of the musical patterns. I noticed that typically in the non-listening condition, the generated sound pattern had a short duration compared to the listening condition.

When listening to the sound, I sometimes experienced a feeling of performance anxiety. I feel that this can hardly be the case when non-listening to the sound. Furthermore, it felt like I spent more time and was more careful when listening to the sound. It seems logical as any programming mistakes can have significant differences in the produced sound levels, which can cause hearing damage. Thus, it looks like that the listening condition is linked to more careful actions, whereas the non-listening condition produces greater extent of experimentation and trial-and-error practices.

4.3. Psychology of programming when non-listening

Whereas I will not present a detailed analysis on the cognitive dimensions of notation between the listening and non-listening conditions, I will present certain aspects that may improve the quality of discussion (A. Blackwell & Green, 2003). I realised that *progressive evaluations* could be complex when non-listening to the musical outcome. When executing a new code chunk, it can be hard to spot differences from the spectrum. For instance, perceiving differences on the spectrum can be difficult when the musical modifications have slight frequency variations. Consequently this can cause *hard mental operations*, as the console may not show any programming errors. Furthermore, I have noticed that when I listen to the sound, I am more careful in progressive evaluations. That may indicate that the listening condition increases *viscosity*, that is resistance to change, but I think that the non-listening is more likely to cause such imbalance. This is because, *error-proneness* is also increased while non-listening to the sound which may cause *premature commitments*. For instance, I may execute a series of *progressive evaluations* and later realise I did no changes on the running program. Because of my inability to identify any differences on the spectrum, I may think that the progressive evaluations impacted the

generated sound patterns. All this description may only apply to me and amplified to some extent by my obsessive commitment on using only the `Ndef` class for sound synthesis in SuperCollider. Further, my obscure coding practice with long one-liners can also cause severe problems to progressive evaluations, error-proneness and hard-mental operations.

5. Discussion

One of my goals is to motivate other live coders to collaborate often and practice in a daily fashion. Because of the very nature of live coding, as a practice which differs substantially from traditional music performance, I have the feeling that live coders do not practice in a daily fashion (at least, this is the case for me). Carrying out a month-long practice made me realise different things, ranging from my programming skills with my favourite programming language, SuperCollider, to how musical agents can be used in live coding and what programming habits I have when live coding.

The present study began with a view more akin to a controlled experiment and developed into a first-person study based on research through design and reflective diaries. It is important to notice that while I keep an interchangeable order between listening and non-listening conditions (AV and NAS) in October I cannot say whether this had any impact on the study. I did learn several things about my live coding practice, and first and foremost that I do not practice as much as I should be practicing. The month-long practice exercise imposed to me to realise my characteristic incompetence (Dahlstedt, 2012) in a live setting, but also to provide me with hope as I did make some progress. I certainly feel more confident after a month of daily practice, and I also believe I have developed a more on-the-fly feeling about the generated sound patterns. A live coder can quickly get into the labyrinth of 'coding for the sake of coding' and sometimes prioritising the code instead of the generated music could be the case. Because of the nature of typing and textual languages, live coding is far from traditional music performance. Maybe the only sensorimotor relation between live coding and instrumental performance is that both are carried out using serial skilled actions (Palmer, 1997).

The non-listening condition demonstrated how cumbersome and sometimes annoying it can be to live code for the sake of coding. When there is no anticipation of the musical outcome, it can be hard to get motivated to continue a live coding session. Sound enhances our anticipatory reward system, especially when combined with visual correspondences. On the other hand, this lack or reduced levels of anticipation because of absence of sound, may lead to unexpected musical outcomes, which can appeal to the coder. It can give a sense of being outside of one's self. Thus, a non-listening condition can be used to generate novel and creative sound patterns. For instance, I can have an educated guess of how a music pattern will sound by simply writing the code expression. An essential skill set of sound synthesis techniques can enable the coder to surprise herself positively. Thus especially in the case of online live streaming, a no-listening condition can be used as a creative technique for music-making. Given that the audience cannot understand the acoustic environment of the performer, the audience can also be unable to perceive whether the coder is listening or not to the generated sounds. Can such audience-performer dynamics bring about any novel interactions? Or do they raise any artistic concerns? Both are hard to answer and go beyond the scope of this study.

The visual helper I used during the sessions showed a limited number of UGens and was not developed to a musical agent. Its impact on my practice was minimal because it did not require attentional resources. On the other hand, I found such a simple program to be an excellent approach to familiarize myself with the large collection of UGens available in SuperCollider. I did a preliminary investigation on the potential of such visual aid, and I can see the potential of transforming this to either an on-demand agent or to a disruptive agent (Attanayake et al., 2020). Such a system would require a natural language processing component to be trained on the code and generate novel code chunks, coupled to a machine listening and learning component capable to extract similarity measures between the running musical outcome and the simulated code evaluations. The git repository can provide the coding database as it contains 98 individual live coding sessions.

It is important to notice that machine listening and learning algorithms have become more accessible.

In the seminal practice session by Collins and Olofsson only a pitch follower UGen is used in a couple of live coding sessions. More studies will follow on that aspect, given the rapid advances in the field of machine listening and machine learning. Indicatively the FluCoMa environment (Tremblay, Roma, & Green, 2021) offers a rich library for applying such computational techniques and is compatible with a variety of programming environments, like MAX/MSP, PureData and SuperCollider. In my sessions I did use some machine listening UGens, but I did not use any machine learning algorithms. This is likely the case because of the blank slate approach, which would require much effort in order to adjust such algorithms to my sessions.

While this is an ongoing research in terms of live coding practice as a disciplined endeavour, several things become apparent. First and foremost when there are no informative visual cues during the non-listening condition it is almost impossible to live code. I found myself losing the thread of coding and had no idea how the musical outcome may sound like. Further on, from the spectrum alone it can be sometimes difficult to perceive whether command executions have actually any impact on the generated musical outcome.

A month-long practice is an excellent approach to get better on live coding. I found myself improving on several aspects, from creating naming conventions that can be informative, to how to separate triggers and control signals from sound generators. Such programming practices do have an impact on the musical outcome, as I was able to control musical structures more fluidly. Finally, I did carry the practice sessions alone and at times I was feeling demotivated and I was repeating my practices over and over. This feeling of demotivation amplifies the importance of collaboration. Whether the collaboration is as simple as turn-taking sessions or collaborative group performance, it is important to interact with more live coders and exchange ideas and knowledge. I recommend live coders that aim to commit to month-long practice sessions to find a human collaborator.

6. Conclusions

In this study, I challenged myself to do a month-long daily practice in musical live coding. I aimed to examine the relations between auditory and visual percepts and improve my live coding skills. I discussed how listening to the sound can help the coder in progressive evaluations and reduce error-proneness, while non-listening to the sound may be used as a creative coding practice technique. Further, when no informative visual cues and no sound are available to the coder, practicing musical live coding looks like an impossible task as I usually lose the thread of programming. I also experimented with a simple GUI visual helper, which can be developed into a software agent, and the documentation of the study can be used as a dataset for such developments. Practicing live coding in a daily and disciplined fashion is not easy, and several times, I repeated myself in both my coding practices and reflective diaries. The practice of musical live coding can be easier when it is a group activity, and I recommend to live coders who are determined to commit to month-long daily sessions to find at least one more coder.

7. Acknowledgements

I warmly thank the PPIG community for the valuable feedback during the conference, and special thanks to Alan Blackwell for his detailed review of the draft version of the present study. Both were catalytic to the improvement of this study.

8. References

- Attanayake, U., Swift, B., Gardner, H., & Sorensen, A. (2020). Disruption and creativity in live coding. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 1–5).
- Blackwell, A., & Green, T. (2003). Notational systems—the cognitive dimensions of notations framework. *HCI models, theories, and frameworks: toward an interdisciplinary science*. Morgan Kaufmann, 234.
- Blackwell, A. F., & Aaron, S. (2015). Craft practices of live coding language design. In *Proc. first international conference on live coding*.
- Blackwell, A. F., & Collins, N. (2005). The programming language as a musical instrument. In *Ppig*

(p. 11).

- Burland, K., & McLean, A. (2016). Understanding live coding events. *International Journal of Performance Arts and Digital Media*, 12(2), 139–151.
- Collins, N., McLean, A., Rohrerhuber, J., & Ward, A. (2003). Live coding in laptop performance. *Organised sound*, 8(3), 321–330.
- Dahlstedt, P. (2012). Between material and ideas: A process-based spatial model of artistic creativity. In *Computers and creativity* (pp. 205–233). Springer.
- Dal Rì, F. A., & Masu, R. (2022). Exploring musical form: Digital scores to support live coding practice. In *Nime 2022*.
- Davidson, J. W. (1993). Visual perception of performance manner in the movements of solo musicians. *Psychology of music*, 21(2), 103–113.
- Gamboa, M. (2022). Conversations with myself: Sketching workshop experiences in design epistemology. In *Creativity and cognition* (pp. 71–82).
- Hove, M. J., Fairhurst, M. T., Kotz, S. A., & Keller, P. E. (2013). Synchronizing with auditory and visual rhythms: an fmri assessment of modality differences and modality appropriateness. *Neuroimage*, 67, 313–321.
- Jensenius, A. R. (2007). Action-sound: Developing methods and tools to study music-related body movement.
- Magnusson, T. (2015). Code scores in live coding practice. In *Proceedings of the international conference for technologies for music notation and representation, paris* (Vol. 5).
- Nilson, C. (2007). Live coding practice. In *Proceedings of the 7th international conference on new interfaces for musical expression* (pp. 112–117).
- Palmer, C. (1997). Music performance. *Annual review of psychology*, 48(1), 115–138.
- Roberts, C., & Wakefield, G. (2018). *Tensions and techniques in live coding performance*.
- Sorensen, A. C., & Brown, A. R. (2007). aa-cell in practice: An approach to musical live coding. In *International computer music conference* (pp. 292–299).
- Tremblay, P. A., Roma, G., & Green, O. (2021). Enabling programmatic data mining as musicking: The fluid corpus manipulation toolkit. *Computer Music Journal*, 45(2), 9–23.

Keynote

Making program analysis useful

Emma Soderberg

Senior Lecturer and Reader in Computer Science at Lund University, Sweden

Program analysis is meant to aid software developers in creating software, but often fails at this task. Why is this and what can we do about it?

Mental Models of Recursion: A Secondary Analysis of Novice Learners' Steps and Errors in Java Exercises

Natalie Kiesler

DIPF Leibniz Institute for Research and Information in Education
Frankfurt, Germany
kiesler@dipf.de

Abstract

Teaching and learning recursive programming has been the subject of numerous research projects and studies. However, few research publications focus on learners' steps while solving recursive tasks and the corresponding identification of mental models. It is the goal of this secondary analysis to identify the challenges novice learners of Java encounter in recursive problem solving and to map them to the mental models and conceptions from the literature. The investigated dataset was collected via thinking aloud experiments with eleven first-year-students of computer science in a professional usability laboratory. Students had to recursively compute the factorial of n and the Fibonacci sequence in a learning environment. By using deductive categories from the literature, the students' performance was evaluated in terms of their programming steps, their challenges/errors, and thus their ability to generate a recursive function. The results show that mental models can partially be identified via the analysis of students' problem solving steps and errors. Moreover, recursive tasks with more than one recursive call are more challenging for novice learners. The passive flow of control along with the end of the recursion chain also seem to be counterintuitive for learners. The lack of viable, complete mental models implies the need for further educational research on instructional methods addressing these challenges of first-year students. Learning to program may be easy, but novices require fine-grained, step-by-step scaffolding and instruction, as well as time to understand and apply the more abstract concepts, which include recursion.

1. Introduction

Novice learners of programming experience a variety of challenges in the first year of their studies. Among them are the new circumstances due to the start of a new stage in life, organizational structures, high and unrealistic expectations during the study entry phase (Heublein et al., 2017; Große-Bölting, Schneider, & Mühling, 2019; M. McCracken et al., 2001; Luxton-Reilly, 2016). In the context of computing, students are further confronted with high performance requirements and possibly deficits. A great body of research is dedicated to the investigation of common obstacles for students in basic programming education (Spohrer & Soloway, 1986; Winslow, 1996; M. McCracken et al., 2001; Robins, Rountree, & Rountree, 2003; Luxton-Reilly et al., 2018). Among them is the lack of prior knowledge and expertise (Heublein, Richter, & Schmelzer, 2020), the cognitive gap between understanding and applying a concept (Ala-Mutka, 2004), a high level of abstraction (Kay & Wong, 2018), and the complexity of structuring and constructing algorithms, programs, and mental models (Kahney, 1983; Spohrer & Soloway, 1986; Winslow, 1996; Xinogalos, 2014). Programming primarily comprises cognitively complex tasks (Kiesler, 2020b, 2020a). Recursion is considered as one of them, which is due to its abstract nature, and the lack of everyday analogies; but also its introduction after loops, and the minor pedagogical emphasis towards it (Kurland & Pea, 1985; Leron, 1988; Levy & Lapidot, 2000; Pirolli & Anderson, 1985; Troy & Early, 1992; Wiedenbeck, 1988).

Despite this seemingly consensus on the difficulty of programming and the time and effort it takes to become an expert (Winslow, 1996), Luxton-Reilly summarizes that "learning to program is easy" (Luxton-Reilly, 2016); even children can do it. His arguments are compelling in the face of overwhelming, and repeated evidence of students' challenges in introductory programming (Tew, McCracken, & Guzdial, 2005; Whalley & Lister, 2009; Luxton-Reilly et al., 2018). Despite the great body of research, learners' challenges seem to continue. In the case of recursion, for example, learners still seem to experience more

difficulties related to the passive flow of execution control, especially if two recursive calls are involved. Moreover, students can hardly predict when a recursive program terminates (Scholtz & Sanders, 2010).

In cognitive psychology, the concept of mental models refers to the individual, cognitive representations of a students' knowledge (Johnson-Laird, 1989; Norman, 2014; Schwamb, 1990), whereas the models can be characterized by their viability. In the context of computing, several models have been developed to categorize students' representation of knowledge on recursion (Kahney, 1983; Close & Dicheva, 1997; Bhuiyan, Greer, & McCalla, 1994). Helping students to develop adequate mental models of recursion is a crucial step towards improving their understanding and application of this cognitively complex concept. Educators therefore need a better understanding of students thinking, their problems and above all, their mental models, in order to help support them via conscious pedagogical decisions and instructions. However, educators need indicators to detect non viable mental models early.

In order to eventually help foster students' construction of mental models of recursion, this paper analyzes first-year Computer Science (CS) students' steps and challenges while recursively solving two programming tasks in Java. In addition, the aim is to investigate the extent of which students' problem solving steps can be aligned with the known mental models (Götschi, Sanders, & Galpin, 2003; Scholtz & Sanders, 2010) and reflect the associated viability. The research questions of the present work are: *(1) How do first-year CS students write recursive functions in Java? (2) Which challenges do first-year CS students encounter while recursively solving programming tasks in Java? and (3) To what extent are mental models from the literature reflected in students' problem-solving steps and errors?*

The contribution of this research is a secondary, qualitative analysis of novice programmers steps and the identification of their challenges related to recursive problem solving in Java. It will contribute to the greater generalization of students' representations of knowledge due to the analysis of their problem solving processes in an actual programming language. The results further provide implications on the recognition of non viable models, as well as the development of instructional methods and interventions to foster students' construction of viable mental models of recursion.

The structure of the paper is as follows. Section 2 summarizes related research on mental models, and mental models of recursion. In section 3, the methodology of this secondary analysis will be outlined by introducing the utilized dataset, and the method for data analysis. Next, the results with regard to students' steps (RQ1), and challenges (RQ2) in the recursive problem solving process are presented. These are then mapped with the mental models of recursion and discussed to answer RQ3. An overview of the limitations follows in section 5. Conclusions and future work wrap up this secondary research.

2. Related Research on Learners' Mental Models of Recursion

Mental models are a concept that is rooted in cognitive psychology (Johnson-Laird, 1989; Norman, 2014). They aim at the description of individual, cognitive representations of knowledge (Schwamb, 1990; Wu, Dale, & Bethel, 1998) through a constructivist lens. However, there is not a single constructivist theory. It is rather the sum of ideas on learning via the personal construction of meaning and corresponding instruction (Stone & Goodyear, 1995): "There are many ways to structure the world and there are many meanings or perspectives for any event or concept. Thus, there is not a correct meaning that we are striving for (Duffy & Jonassen, 1991)." There is thus not an ultimate reality everyone agrees upon (Duffy & Jonassen, 1991). Accordingly, learning is defined as the active, subjective process that highly depends on individual interpretation, experiences, and perspective (Stone & Goodyear, 1995). Therefore, knowledge is described as viable, and not as "true" or "correct". For this reason, this work does refer to the challenges novice learners experience in recursive problem solving, and not to the so-called "misconceptions" mentioned in other related work (Close & Dicheva, 1997; Dicheva & Close, 1993). The more radical constructivist theories even neglect the idea of any objective reality (Von Glasersfeld, 1996).

Mental models have also been investigated in the context of computing, and to describe student's individually constructed knowledge of recursion (Bhuiyan et al., 1994; Kahney, 1983; Close & Dicheva,

1997; Dicheva & Close, 1993, 1996; Scholtz & Sanders, 2010). It was found that novice learners' models deviate from the expert model of recursion, making them unable to predict how recursive programs behave (Kahney, 1983; Wu et al., 1998). While novices tend to have constructed the incorrect loop model, experts share the copies model (at least in SOLO, LISP and Logo) (Close & Dicheva, 1997). With more experience and practice, however, novices can change their construction and representation of knowledge, which is at the heart of constructivist learning (Duffy & Jonassen, 1991; Von Glasersfeld, 1996). Norman (2014) concludes that mental models may not be stable, i.e., they can be forgotten or confused with other systems to reduce mental complexity.

Kahney (1983) defines recursion as “a process that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones (Kahney, 1983).” Similarly, George (2000a) defines recursion via the “(active) ‘flow of control’ to a new invocation/copy of the subprogram called”, and the passive flow of control “back from terminated ones” (George, 2000a). According to Kahney (1983), and Kessler & Anderson (1986), novices do experience challenges in understanding tail recursion with functions and embedded recursion, which is related to their lack of understanding the control mechanisms, especially the passive flow of control. This in turn implies them having unfavorable mental models of recursion. Scholtz & Sanders (2010) conclude that students often develop inadequate mental models when two recursive calls are part of a procedure or function. They also observed that students can hardly predict when a recursive algorithm terminates, as they tend to associate recursion with the (previously constructed) loop model. The passive flow of control back from the terminated instantiations is thus a common problem among novice learners of programming. Velázquez-Iturbide (2000) adds that recursion seems to be more difficult in imperative programming languages, as educators tend to confuse recursion and imperative recursion, where other mechanisms (e.g., parameter passing, control stack, etc.) have to be mastered simultaneously.

The following mental models of recursion are summarized in related research investigating and classifying students' constructs (Kahney, 1983; Götschi et al., 2003; Sanders, Galpin, & Götschi, 2006):

- The *Copies Model* is the viable model that reveals the active flow of control, and the switch to the passive flow once the base case is reached. The passive flow of control is made explicit.
- The *Loop Model* views recursion as a kind of iteration that halts once the base case is reached. It ignores both the active and passive flow of control. The base case is considered as stopping condition of the loop.
- The *Active Model* only reflects the active flow of control, but the indication of a passive flow is absent. Students evaluate the solution at the base case. The model can be viable in some cases.
- The *Step Model* is nonviable, as the students lacks understanding of recursion. Either the recursive condition, or the recursive condition and the base case is executed once.
- The *Return Value Model* describes the view that values are generated by each instantiation, which are then stored and combined to calculate a solution.
- The “*Syntactic*”, “*Magic*” *Model* reveals that students have no idea of recursion and how it works. Nonetheless, they can match syntactic elements. The active flow, base case, and passive flow can be traced. Due to their errors, a lack of understanding is assumed, which requires further teaching/learning activities.
- The *Algebraic Model* describes students who treat the program as algebraic problem.
- The *Odd Model* encompasses different misunderstandings, which lead to the student not being able to predict the program's behavior.

In addition to developing viable mental models, learners need time and practice to construct mental models to understand and apply program structures, whereas even varying mental models can lead to identical and correct problem solutions (Kahney, 1983). To conclude, it is difficult for learners to develop adequate schemata. Likewise, educators face challenges in assessing individual mental models and constructs required for the successful planning of algorithms and programs. Due to these reasons, it is important to investigate mental models of novice learners of programming further by examining their

ability to recursively solve problems in a commonly used programming language, such as Java. The results and implications on students' mental models will help address learners' challenges by developing new forms of instructions and pedagogical concepts.

3. Methodology

For this secondary analysis of research data, a publicly available, qualitative dataset with students' problem solving steps was utilized to identify their challenges, and to what extent their mental models of recursion are reflected. This sections briefly introduces the context of the primary dataset, its structure, and tasks. Moreover, the analysis of the data is presented. The goal is to investigate students' steps and errors, and to test the mental models further, as implied by prior research (Close & Dicheva, 1997).

3.1. Data Collection

In educational technology research, identifying research data for secondary research is challenging due to several reasons (e.g., lack of recognition for researchers, lack of quality, data provenance etc. (Kiesler & Schiffner, 2022)). In the context of programming education, few datasets are available that allow for the analysis of novice learners' steps while recursively solving programming exercises in Java. One of them is provided by a German research data center (Kiesler, 2022a, 2022b). It was part of a recent doctoral dissertation in the context of introductory programming education (Kiesler, 2022d).

The data had originally been gathered to investigate the effects of informative feedback offered by (on-line) self-learning tools (Kiesler, 2022a), such as CodingBat (Parlante, 2022a) (see Figure 1). Two recursive tasks in Java were selected for the conduction of two thinking aloud experiments (test series A and B) (Heine, 2005; Knorr, 2013; Konrad, 2017) where students had to recursively solve problems in Java. In sum, eleven students of the Department of Applied Computer Science participated as test subjects in series A and B. All of them had successfully participated and completed the course "Programming 1" (i.e., the basic programming course) at Fulda University of Applied Sciences in Germany, which also addresses the concept of recursion as part of both the lecture and exercise session.

CodingBat code practice

Recursion-1 > factorial

Given n of 1 or more, return the factorial of n, which is $n * (n-1) * (n-2) \dots 1$. Compute the result recursively (without loops).

factorial(1) → 1
factorial(2) → 2
factorial(3) → 6

Go ...Save, Compile, Run (ctrl-enter) [Show Hint](#) [Show Solution](#)

```
public int factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

Expected	Run	
factorial(1) → 1	1	OK
factorial(2) → 2	2	OK
factorial(3) → 6	6	OK
factorial(4) → 24	24	OK
factorial(5) → 120	120	OK
factorial(6) → 720	720	OK
factorial(7) → 5040	5040	OK
factorial(8) → 40320	40320	OK
factorial(12) → 479001600	479001600	OK
other tests		OK

All Correct

Good job -- problem solved. You can see our solution as an alternative.

[See Our Solution](#)

Figure 1 – Condensed screenshot of a CodingBat exercise on recursion.

Students had to solve two tasks, which served as test instruments. Both test series A and B included the computation of the factorial of n, resulting in a sample size of 11 students for the first task (students A01 to A06, and B01 to B05). In addition, the Fibonacci sequence was computed by the five students of test series B (students B01 to B05). A total of 16 transcripts from 11 students are thus available. In the primary research, another tool/task (Kiesler, 2016a, 2016b) was tested in series A, resulting in a lower N in the Fibonacci task. Computing the factorial of n as a task in both series A and B requires students to write a base case and one recursive call. Computing the Fibonacci sequence as second task in series B demands two base cases, and two recursive calls. Both tasks require students to apply *tail recursion*, which refers to the position of the recursive call as last action of the algorithms.

1. N=11: Given n of 1 or more, return the factorial of n, which is $n * (n-1) * (n-2) \dots 1$. Compute the result recursively (without loops) (Parlante, 2022b).

2. N=5: The fibonacci sequence is a famous bit of mathematics, and it happens to have a recursive definition. The first two values in the sequence are 0 and 1 (essentially 2 base cases). Each subsequent value is the sum of the previous two values, so the whole sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21 and so on. Define a recursive fibonacci(n) method that returns the nth fibonacci number, with n=0 representing the start of the sequence (Parlante, 2022c).

The dataset comprises the manually transcribed students steps in the problem solving process in form of a table, so that students' input to the available CodingBat working space can be traced. Every change in the working space (where the programming code is written) has been transcribed, as depicted in Table 1. Students' steps were further qualitatively analyzed into categories implying the type of change (e.g., keyword added (if), recursive function call added, parameter of recursive function call added, etc.). The dataset also reveals timestamps, and the system feedback provided by CodingBat (Parlante, 2022a), so that every change/step made by students can be reconstructed and students' reactions to the system feedback can be tracked. A detailed description of the dataset and the applied methodology is available online (Kiesler, 2022b).

Table 1 – Excerpt Representing the Structure of the Dataset on Students' Steps.

time	programming code	(qualitative) code	action (button)	system feedback
25:05	<pre>public int factorial(int n) { }</pre>	problem solving starts		
25:14	<pre>public int factorial(int n) { }</pre>	no change	Go button	Compile problems: missing return statement line:3 see Example Code to help with compile problems
25:17	<pre>public int factorial(int n) { }</pre>	no change	Hint button	First, detect the "base case", a case so simple that the answer can be returned immediately (here when n==1). Otherwise make a recursive call of factorial(n-1) (towards the base case). Assume the recursive call returns a correct value, and fix that value up to make our result.
25:49	<pre>public int factorial(int n) { if () }</pre>	keyword added (if)		
26:06	<pre>public int factorial(int n) { if (n == 1) }</pre>	one condition added (base case)		
26:09	<pre>public int factorial(int n) { if (n == 1) return }</pre>	keyword added (return)		

3.2. Data Analysis

Due to this research's secondary nature, the primary dataset and its structure (see Table 1) predetermine the secondary analysis as a whole. In the case of the present dataset, students' sequence of problem solving steps/changes in the programming code can be analyzed to answer the research question (1) *How do first-year CS students write recursive functions in Java?* In order to answer RQ1, the lines of code students write are analyzed with regard to their steps, for example, whether they relate to the base case, the recursive function call, or the parameter of the recursive function call. In this context, the changes of the programming code will also be relevant. The number of changes are defined via

the steps needed to come to the correct solution after having made a mistake that was executed via the Go-Button provided by CodingBat (e.g., deleting, adding, changing return values, parameters, etc.). Changes in indentations are ignored. A step aligns with the primary research's analysis, where every step (displayed per row in Table 1) was assigned a qualitative code. The qualitative codes of all 16 records will be analyzed and clustered with regard to these crucial elements of the recursive solution to the tasks. Timestamps, however, will not be considered.

Next, students' errors will be analyzed to answer the second research question (2) *Which challenges do first-year CS students encounter while recursively solving programming tasks in Java?* In this regard, the main components of recursion will be used as categories: the active flow, the passive flow, and the base case(s). If applicable, these categories are supplemented by further subcategories, which are inductively built and used to describe accompanying challenges, or error causes. In addition, commonly known mistakes from the literature (Close & Dicheva, 1997) are considered for the qualitative coding.

As a last step of this analysis, students' steps and challenges during recursive problem solving will be mapped to the known mental models described in Section 2 to answer research question (3) *To what extent are mental models from the literature reflected in students' problem-solving steps and errors?* As a part of this mapping, the definition of models will be used to identify patterns in the problem solving process that may be aligned with the construction of the mental models' categories as defined by Kahney (1983); Götschi et al. (2003); Sanders, Galpin, & Götschi (2006). The corresponding identification of indicators will help operationalize mental models for educators, and eventually help foster learners' understanding of recursion.

4. Results

4.1. How Students Write Recursive Functions in Java (RQ1)

The problem solving process to some extent manifests in how students write a program as a solution to a task. In the present dataset, all 16 solutions were analyzed with regard to the main components of the expected recursive functions and their sequence, e.g., the if statement related to the base case(s), the recursive function call(s) and its/their parameter(s), as well as the correct use of operators. The two tasks were analyzed separately, and the following analysis distinguishes between the two tasks.

4.1.1. Factorial of n

First of all, it should be noted that all 11 students who worked on this task were able to solve it. Among the four students who successfully solved the task without a single error (A02, A03, A04, B03), three different sequences were observed. In all of them, first-year students write the if statement representing the base case first. The sequence of the remaining components (recursive function call, parameter of recursive function call, and multiplication with n) deviates. However, writing the recursive function call in all cases precedes writing the parameter of recursive function call, which seems to be a successful sequence.

In the remaining seven problem solving processes, similar pattern were recognized. Six of the seven remaining students also started writing the if statement representing the base case first. Only one student (B04) who merely forgot a semicolon and was otherwise error-free wrote the if statement last and began with the recursive function call. Yet again, all students write the recursive function before adding its parameter. The multiplication with n is a step that seems to deviate in its position. Six of the eleven students (A01, A03, A04, A06, B02, B05) immediately write it after finishing the base case. Two of the eleven write it after the recursive function call (A02, B04), and the remaining three write it as a last step (A05, B01, B03).

Students' changes to their code are analyzed next. Due to the four error-free samples, seven samples with changes remain. Table 2 summarizes the changes students made to their code. It should be noted that the table does not represent the sequence of their steps. Most of the changes were related to the if statement and thus the base case. Four student had to adapt their code. Similarly, four students (A01, A05, B01, B02) edited the parameter of their recursive function call. The recursive function call, however, required

Table 2 – Number of changes made by students during recursively solving the factorial of n task.

Student Steps:	A01	A05	A06	B01	B02	B04	B05
if statement (base case)	-	5	-	4	5	-	1
multiplication with n	-	-	-	4	4	-	-
recursive function call	-	-	-	2	-	-	-
parameter of recursive function call	1	3	-	1	5	-	-

only one student (B02) to make changes to their initial code, which means that 10 of the 11 students correctly wrote that component. The multiplication with n required two students (B01, B02) to adapt their code.

4.1.2. Fibonacci Sequence

The Fibonacci task was successfully solved by two students (B01, B04), while two other students (B02, B03) aborted the problem solving process altogether without solving the problem. With the small sample size for this task ($n=5$), few data remains. Nonetheless, the sequence of students' steps is briefly summarized. The two students with no errors and changes applied the following sequence of steps:

1. if statement (first base case)
2. if statement (second base case)
3. first recursive function call
4. parameter of first recursive function call
5. add operator
6. second recursive function call
7. parameter of first second recursive function call

Although student B05 tried to use addition instead of two recursive calls first, the learner in general applied the same sequence of steps. Students B02 and B03 who did not develop a viable solution, made changes related to the second recursive function call, forgot it or misplaced it along with its parameter as part of the second base case (see, for example, Listing 1).

```
public int fibonacci(int n) {
    if(n==0) return 0;
    if(n > 1) n + fibonacci(n-1);
    return n + fibonacci(n-1);
}
```

Listing 1 – Example of student B02's positioning of the second recursive function call.

Students' changes of their code are summarized in Table 3. Again, two students (B02, B05) were concerned with the correction of their if statements, which represent the base case(s). Moreover, the first recursive function caused errors and therefore required changes from two students (B02, B05). Student

Table 3 – Number of changes made by students during recursively solving the Fibonacci task ($n.a.$ = not available).

Student Steps:	B02	B03	B05
if statement (first base case)	-	-	1
if statement (second base case)	4	-	-
first recursive function call	1	-	2
parameter of first recursive function call	1	2	2
add operator	-	-	-
second recursive function call	-	n.a.	-
parameter of second recursive function call	-	17	-

B02, for example, somehow added the first recursive function to the base case (see Listing 1). B03 was especially busy with the edition of the parameter of the second recursive function call, as the second recursive function call itself was not written, and the error could not be corrected after 17 attempts. The parameter of the (first and second) recursive function call required changes from all three students B02, B03 and B05. It should be noted that CodingBat did not provide a hint or model solution for the task. Only the expected values were presented and compared with students' results (see right-hand side of Figure 1).

4.2. Students' Challenges During Recursive Problem Solving (RQ2)

In this section, the first-year CS students' challenges during recursive problem solving are summarized. For each task, a categorisation of student errors is presented and related to the main components of the expected recursive solution.

4.2.1. Factorial of n

Although the recursive computation of the factorial of n may seem simple, and all 11 students were capable of solving the problem, errors were made during the process. The left column of Table 4 summarizes the observations into categories of detected student errors. The number in brackets behind the bullet points indicates the frequency of the errors, and thus how many of the students made these errors.

The list reveals that not all errors were related to recursion. The first bullet point reveals a number of syntax and semantics problems resulting in compile errors or the CodingBat feedback "Bad Code". The second bullet point reveals that a student did not adhere to the starter code provided by CodingBat, which is always correct. The student thus tried to alter the function's signature and solve the problem by introducing an additional parameter and variable. The last three bullet points concern students' errors related to recursion, e.g., the condition or return values of the base case, the recursive function call, and its parameter. They thus comprise all three key components of the recursive solution to the problem. It should further be noted that 4 of the 11 students were able to solve the task without any error.

4.2.2. Fibonacci Sequence

The second task required students to write two base cases and use two recursive function calls with correct parameters. Two of the five students with available data were able to successfully solve the task without any error. Two students did not develop a viable solution and aborted the problem solving process. The right column of Table 4 summarizes the categories of detected errors.

The enumeration contains only logic errors, although the first bullet point is not directly related to recursion. Instead, the student tried to solve the problem with an additional help function. The three other error categories, however, are closely related to the main components of recursion. Similarly to the first task, learners experienced challenges when writing the base case, the first and second recursive function call, and the corresponding parameters. In addition, the selection of an operator for the calculation of the return value of the else-block seemed to be challenging. The lack of the second recursive function call (with parameter) and students' substitution of it via some other operation should be noted as main cause of errors.

4.3. What Students' Steps and Errors Reveal about their Mental Models of Recursion (RQ3)

The last research question aims at the identification of indicators of students' mental models of recursion. Unlike related work, this research did not analyze students tracing of the execution of recursive programs (Kahney, 1983; Götschi et al., 2003; Sanders et al., 2006). Thus, the implications of students' steps and errors on their models will be discussed based on the qualitative dataset. Nonetheless, the categories Götschi et al. (2003) used for the analysis of students' traces related to the active flow, the base case, and the passive flow were considered as starting point. However, due to their (Götschi et al., 2003) lack of a detailed definition and anchoring examples for the categories, the coding scheme could not entirely be replicated.

Table 4 – Overview of Students’ Challenges While Recursively Solving Two Exemplary Problems.

Factorial of n ($n!$)	Fibonacci Sequence
<ul style="list-style-type: none"> • Syntax and semantics errors <ul style="list-style-type: none"> – using incorrect operators as condition of the case case or parameter (2) – lack of return statement in else block (2) – lack of return value of else block – adding redundant return statement – mixing up <code>System.out.println</code> and return statements – lack of semicolon • Falsification of starter code <ul style="list-style-type: none"> – changing the functions’ given correct signature by adding a parameter – follow-up error: introducing additional, unnecessary variable • Logic Error: Errors related to the base case <ul style="list-style-type: none"> – incorrect return value (zero) of base case (2) – incorrect condition of base case • Logic Error: Errors related to recursive function call <ul style="list-style-type: none"> – lack of recursive function call (with parameter) – adding a second, unnecessary recursive function call (plus parameter) • Logic Error: Errors related to parameter of recursive function call <ul style="list-style-type: none"> – multiplication of n is within the parameter of the recursive function call (2) – other incorrect parameter of recursive function call 	<ul style="list-style-type: none"> • Logic Error: Declaration of additional function <ul style="list-style-type: none"> – addition of function signature with additional parameter • Logic Error: Errors related to base cases <ul style="list-style-type: none"> – incorrect return value of second base case – lack of second base case – incorrect condition of base case • Logic Error: Errors related to recursive function calls <ul style="list-style-type: none"> – addition with n (or other operation, e.g., $n - 1$, $2 * n$, $n - (2 * (n - 1))$, $n - 3$, $n - 2$, $n - (n/2)$, $n - (n - 2)$, etc.) instead of second recursive function call (with parameter) as return values of the else-block (3) – lack of second recursive function call (with parameter) in else-block (2) – placement of second recursive function call (with parameter) as return value of the second base case • Logic Error: Errors related to parameters of recursive function calls <ul style="list-style-type: none"> – incorrect parameter(s) of first recursive function call

4.3.1. Active Flow

Some of the categories by (Götschi et al., 2003) could be applied to students’ steps, changes and errors without requiring more data or details. Among them were code alterations related to the active flow of control, thus calling `factorial()`, until its argument becomes 1. In this context, the algebraic manipulation of the function call ($+n, +n-1$, etc.) instead of a second recursive call was observed (B03, B05). Thus the need for a second recursive function call was not clear to all first-year students. The “algebraic” category (Götschi et al., 2003) is an indicator of students having adopted the *step model* or *return value model*. Determining the parameter of the recursive function call was challenging for many students. In 7 of the 16 records and in both tasks, students changed the parameters after initial, unsuccessful attempts to execute their code. Nonetheless, all students wrote at least one recursive call with a parameter, and thus made “a new invocation with a new argument” (Götschi et al., 2003). Moreover, several students achieved a correct solution without requiring feedback from the system. It can be assumed that they share at least this component of the *copies model* of recursion.

4.3.2. Base Case

Two categories (Götschi et al., 2003) assigned to the base case were observed. The first one is the “check incorrect” category describing an incorrect test for a base case. This error occurred in both tasks (e.g., B01, B02; factorial of n , and B02; Fibonacci). The incorrect test definition may indicate the *odd model*, as students are not able to predict the program’s behavior. The second category “base omitted” was detected when subject B03 tried to compute the Fibonacci sequence. The second base case was omitted.

Other errors related to the base case (see Table 4) occurred in both tasks, and concerned incorrect return values. This error is evaluated as yet another indicator for the *return value model* or the *step model*. Yet again, the six cases of immediate correct student solutions (out of 16) may imply that students developed the viable *copies model*, where the switch from active flow to passive flow takes place once the base case is reached. In any case, all students were aware of the necessity of a base case.

4.3.3. Passive Flow

Students' steps, their changes to the code and errors somewhat hint towards students' mental models of recursion. Although judgements about students' conception of the passive flow of control are challenging via the analysis of their program code, the "return problem" and "changed operations" implied non viable models. For example, student B02 added a recursive function call to the second base case in the Fibonacci task (see Listing 1), implying misconceptions about parameter passing and return value evaluation. The error could imply the *odd model*. The change of operations (addition, subtraction, multiplication, division), and their order was also observed when students worked on the parameter of the recursive function call. According to (Götschi et al., 2003) and (Kahney, 1983) this is an indicator of the *magic model*, because students seem to be "sensitive to the position of the different program segment" (Kahney, 1983).

4.3.4. Discussion and Implications for Teaching

The idea to analyze students' steps, and errors to gain insight into their mental models of recursion revealed a number of challenges and findings. These will be discussed in the following, along with the implications for introductory programming education. First of all, the categories and models developed in related work (Kahney, 1983; Götschi et al., 2003; Sanders et al., 2006) proved to be incomplete with regard to their categorization, category definitions and their application to student solutions. In addition, an evaluation of categories for all three components (active flow, base case, and passive flow) along with a mapping to the resulting model is not available. A full replication is thus impossible. However, some indicators for the models presented in prior work were identified. Among them were categories related to the active flow (algebraic), base case (check incorrect, base omitted) and passive flow (return problem, operation changed). This leads to the assumption that students' steps and errors do provide indicators on their mental models of recursion, especially, if their problem solving steps reveal errors related to the recursive function call(s), their parameter, the condition and return value of the base case, and the if statement in general. Therefore, the assessment of students' steps and errors seems to constitute a promising approach for educators to help facilitate student learning and the development of viable mental models. However, whether a student developed a viable mental model could not be confirmed, despite some students achieving the correct solution in either the factorial, or the Fibonacci task. As none of the students from the B test series succeeded in both tasks, the conclusion is that students did not develop fully viable models yet.

A second important finding concerns students' challenges and errors. For example, the numerous adaptations of students' code indicate that student do not know how to write a viable condition (if statement) with a return statement, even when receiving several types of tutoring feedback by a tool like Coding-Bat (Kiesler, 2022c). Recursion does not seem to be the only obstacle. Students are not convinced of their selection of parameters, they change them often, and seemingly random. This may indicate that their knowledge representation of parameter passing is not yet complete/viable. In case of the Fibonacci sequence, the many changes related to the second recursive function call, its parameter and the calculation of the return value seems to be most challenging for learners. They also need to recognize that two base cases with a certain return value are required. While analyzing students' application of the concept of recursion, we need to ask whether we as educators can evaluate mental models of recursion in isolation, without the review of related concepts, learning objectives, and competencies (Raj et al., 2022).

Third, the findings reveal that many of the first-year student solutions show indicators of non viable mental models of recursion. According to Luxton-Reilly (Luxton-Reilly, 2016), learners do understand the concept of recursion, but it may take them longer than educators (the experts) expect. George (2000b)

suggests that the explicit simulation of a recursive algorithm's execution via diagrammatic traces may be the best method when teaching recursion to students. Wilcocks und Sanders (1994) suggest animations to illustrate the execution of recursive programs, the flows of control and the resulting "copies" (Kahney, 1983; Sanders et al., 2006). Scholtz and Sanders (2010) recommend to use numerous examples of recursive problems and algorithms. Close further outlines a number of approaches for teaching recursion to children, aged 10 to 14 (Close & Dicheva, 1997). It has further been argued that recursion should not be treated as an advanced topic, but rather be taught early in the curriculum (Astrachan, 1994; D. D. McCracken, 1987): "If recursion is presented as one powerful tool among others, through many examples and with opportunities to practice using it" (D. D. McCracken, 1987). Similarly, a gradual approach to recursion is recommended by Velazquez-Iturbide (2000). The present findings support these perspectives and implications for teaching recursion to novices.

5. Limitations

The limitations of this work are due to the small sample size and its origin from a single institution. Moreover, the limitations of the primary research apply with regard to the unnatural situation in the usability laboratory where the experimental setup had been realized. Students may have behaved differently in a natural setting. The transcription and qualitative analysis of the dataset, however, allows for the precise reproduction of students' steps in that setting, and an in-depth perspective into students' recursive problem solving in Java. More qualitative, pre-processed data on students' steps would nonetheless help generalize the understanding of how students solve recursive programming tasks in Java.

6. Conclusions and Future Work

The goal of this research was to identify the steps and challenges of novice learners in the context of recursive problem solving, and to explore what students' steps and errors reveal about their mental models of recursion. The present work utilized a publicly available, qualitative, and pre-coded dataset with students' problem solving steps during two Java exercises which had been gathered in a usability laboratory. The data analysis clustered students' sequences of steps, their changes to the code, as well as their errors in alignment with the main components of the expected recursive solutions. Although prior research (Götschi et al., 2003; Sanders et al., 2006) could not be replicated, indicators for students' mental models were identified. Among them are students' errors related to the base case, the recursive function calls, and their parameters. The second recursive call in the Fibonacci sequence seemed another challenge. Moreover, syntax and semantics errors (e.g., operators, return statements, declaration of functions, parameter passing, etc.) posed challenges to learners. The analysis whether, and to what extent non viable mental models, such as the *step model* or *return value model* and their indicators are reflected and observable in students' problem solving steps and errors will help educators identify non viable models, and foster students' understanding of recursion in terms of their mental representation of the concept. Learning environments may also adapt their feedback accordingly.

To conclude, recursion is still a challenging concept for novice learners, which is why the need for adequate, learner-centered instruction is a continuing one. The paper further results in implications for introductory programming education, such as more critical teaching practices, revisiting expected norms for introductory programming and becoming more realistic towards achievable learning outcomes of introductory courses. In future work, big data or learning analytics approaches may be used to analyze, or pre-process quantitative datasets on students' steps, which are available at repositories (Koedinger et al., 2010). Such research will help educators find more and early evidence of non viable mental models in novice learners' programs. Another currently pursued approach is the development of expert feedback for the two selected exercises and students' steps (see, e.g. Jeuring et al., 2022).

7. References

Ala-Mutka, K. (2004). *Problems in learning and teaching programming – A literature study for developing visualizations in the Codewitz-Minerva project*. Online. Retrieved from https://www.cs.tut.fi/~edge/literature_study.pdf

- Astrachan, O. (1994). Self-reference is an illustrative essential. In *Proceedings of the twenty-fifth sigcse symposium on computer science education* (pp. 238–242).
- Bhuiyan, S., Greer, J. E., & McCalla, G. I. (1994). Supporting the learning of recursive problem solving. *Interactive Learning Environments*, 4(2), 115–139.
- Close, J., & Dicheva, D. (1997). Misconceptions in recursion: diagnostic teaching. In *Proceedings of the sixth eurologo conference “learning and exploring with logo* (pp. 132–140).
- Dicheva, D., & Close, J. (1996). Mental models of recursion. *Journal of Educational Computing Research*, 14(1), 1–23.
- Dicheva, D., & Close, S. (1993). Misconceptions and Mental Models of Recursion. In *Proceedings of the fourth european logo conference* (pp. 12–20).
- Duffy, T. M., & Jonassen, D. H. (1991). Constructivism: New implications for instructional technology? *Educational technology*, 31(5), 7–12.
- George, C. E. (2000a). Erosi—visualising recursion and discovering new errors. *ACM SIGCSE Bulletin*, 32(1), 305–309.
- George, C. E. (2000b). Experiences with novices: The importance of graphical representations in supporting mental models. In *Ppig* (p. 3).
- Götschi, T., Sanders, I., & Galpin, V. (2003, jan). Mental models of recursion. *SIGCSE Bull.*, 35(1), 346–350. doi: 10.1145/792548.612004
- Große-Bölting, G., Schneider, Y., & Mühling, A. (2019). It’s like computers speak a different language: Beginning students’ conceptions of computer science. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. New York: ACM. doi: 10.1145/3364510.3364527
- Heine, L. (2005). Lautes Denken als Forschungsinstrument in der Fremdsprachenforschung. *Zeitschrift für Fremdsprachenforschung: ZFF*, 16(2), 163–186.
- Heublein, U., Ebert, J., Hutzsch, C., Isleib, S., König, R., Richter, J., & Woisch, A. (2017). Zwischen Studierenerwartungen und Studienwirklichkeit. In *Forum hochschule* (Vol. 1, pp. 134–136).
- Heublein, U., Richter, J., & Schmelzer, R. (2020). Die Entwicklung der Studienabbruchquoten in Deutschland. *DZHW Brief*(3). Retrieved from https://doi.org/10.34878/2020.03.dzhw_brief
- Jeurig, J., Keuning, H., Marwan, S., Bouvier, D., Izu, C., Kiesler, N., ... Sarsa, S. (2022). Steps learners take when solving programming tasks, and how learning environments (should) respond to them. In *Proceedings of the 27th acm conference on on innovation and technology in computer science education vol. 2* (p. 570–571). New York: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3502717.3532168>
- Johnson-Laird, P. N. (1989). Mental models. *Foundations of cognitive science*, 469–499.
- Kahney, H. (1983). What Do Novice Programmers Know about Recursion? In *Proceedings of the sigchi conference on human factors in computing systems* (p. 235–239). New York: Association for Computing Machinery. doi: 10.1145/800045.801618
- Kay, A., & Wong, S. H. S. (2018). Discovering Missing Stages in the Teaching of Algorithm Analysis: An APOS-Based Study. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. New York: Association for Computing Machinery. doi: 10.1145/3279720.3279738
- Kessler, C. M., & Anderson, J. R. (1986). Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, 2(2), 135–166.
- Kiesler, N. (2016a). Ein Bild sagt mehr als tausend Worte—interaktive Visualisierungen in webbasierten Programmieraufgaben. In U. Lucke, A. Schwill, & R. Zender (Eds.), *DeLFI 2016—Die 14. E-Learning Fachtagung Informatik, 11.–14. September 2016, Potsdam* (Vol. P-262, pp. 335–337). GI. Retrieved from <https://dl.gi.de/20.500.12116/566>
- Kiesler, N. (2016b, 4-6 July, 2016). Teaching Programming 201 with Visual Code Blocks instead of VI, Eclipse or Visual Studio—Experiences and Potential Use Cases for Higher Education. In *EDULEARN16 Proceedings* (p. 3171–3179). IATED. doi: 10.21125/edulearn.2016.0169

- Kiesler, N. (2020a). On Programming Competence and Its Classification. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. New York: Association for Computing Machinery. doi: 10.1145/3428029.3428030
- Kiesler, N. (2020b). Towards a Competence Model for the Novice Programmer Using Bloom's Revised Taxonomy – An Empirical Approach. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 459–465). New York: ACM. doi: 10.1145/3341525.3387419
- Kiesler, N. (2022a). *Dataset: Recursive problem solving in the online learning environment CodingBat by computer science students*. Online. (Datenerhebung: 2017. Version: 1.0.0. Datenpaketzugangsweg: Download-SUF. Hannover: FDZ-DZHW. Datenkuratierung: İkiz-Akinci, Dilek) doi: <https://doi.org/10.21249/DZHW:studentsteps:1.0.0>
- Kiesler, N. (2022b). *Daten- und Methodenbericht Rekursive Problemlösung in der Online Lernumgebung CodingBat durch Informatik-Studierende* (Tech. Rep.). ([https://metadata.fdz.dzhw.eu/public/files/data-packages/stu-studentsteps\\$/attachments/studentsteps_Data_Methods_Report_de.pdf](https://metadata.fdz.dzhw.eu/public/files/data-packages/stu-studentsteps$/attachments/studentsteps_Data_Methods_Report_de.pdf))
- Kiesler, N. (2022c, June). *An Exploratory Analysis of Feedback Types Used in Online Coding Exercises*. arXiv. Retrieved from <https://doi.org/10.48550/arXiv.2206.03077> doi: 10.48550/ARXIV.2206.03077
- Kiesler, N. (2022d). *Kompetenzförderung in der Programmierausbildung durch Modellierung von Kompetenzen und informativem Feedback* (Dissertation). Johann Wolfgang Goethe-Universität, Frankfurt am Main. (Fachbereich Informatik und Mathematik)
- Kiesler, N., & Schiffner, D. (2022). On the lack of recognition of software artifacts and its infrastructure in educational technology research. In P. A. Henning, M. Striewe, & M. Wölfel (Eds.), *20. Fachtagung Bildungstechnologien (DELFI)* (pp. 201–206). Bonn: Gesellschaft für Informatik e.V. doi: 10.18420/delfi2022-034
- Knorr, P. (2013). Zur Differenzierung retrospektiver verbaler Daten: Protokolle Lauten Erinnerns erheben, verstehen und analysieren. In K. Aguado, L. Heine, & K. Schramm (Eds.), *Introspektive Verfahren und qualitative Inhaltsanalyse in der Fremdsprachenforschung* (pp. 31–53). Frankfurt: Peter Lang.
- Koedinger, K., Baker, R., Cunningham, K., Skogsholm, A., Leber, B., & Stamper, J. (2010). A Data Repository for the EDM community: The PSLC DataShop. In C. Romero, S. Ventura, M. Pechenizkiy, & R. Baker (Eds.), *Handbook of educational data mining*. CRC Press: Boca Raton, FL.
- Konrad, K. (2017). Lautes Denken in psychologischer Forschung und Praxis. In G. Mey & K. Mruck (Eds.), *Handbuch qualitative Forschung in der Psychologie* (pp. 2–21). Wiesbaden: Springer Fachmedien.
- Kurland, D. M., & Pea, R. D. (1985). Children's mental models of recursive LOGO programs. *Journal of Educational Computing Research*, 1(2), 235–243.
- Leron, U. (1988). What makes recursion hard. In *Proceedings of the sixth international congress on mathematics education*.
- Levy, D., & Lapidot, T. (2000). Recursively speaking: analyzing students' discourse of recursive phenomena. In *Proceedings of the thirty-first sigcse technical symposium on computer science education* (pp. 315–319).
- Luxton-Reilly, A. (2016). Learning to program is easy. In *Proceedings of the 2016 acm conference on innovation and technology in computer science education* (pp. 284–289). New York: Association for Computing Machinery. doi: 10.1145/2899415.2899432
- Luxton-Reilly, A., Simon, Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., ... Szabo, C. (2018). Introductory programming: A systematic literature review. In *Proceedings companion of the 23rd annual acm conference on innovation and technology in computer science education* (pp. 55–106). New York: ACM.
- McCracken, D. D. (1987). Ruminations on computer science curricula. *Communications of the ACM*,

30(1), 3–6.

- McCracken, M., Almstrum, V., Diaz, D., Guzdia, M., Hagan, D., Kolikant, Y. B.-D., ... Wilusz, T. (2001). A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (p. 125–180). New York: ACM. doi: 10.1145/572133.572137
- Norman, D. A. (2014). Some observations on mental models. In *Mental models* (pp. 15–22). Psychology Press.
- Parlante, N. (2022a). *Codingbat*. Retrieved from <https://codingbat.com/about.html>
- Parlante, N. (2022b). *Codingbat recursion 1 factorial*. Retrieved from <https://codingbat.com/prob/p154669>
- Parlante, N. (2022c). *Codingbat recursion 1 fibonacci*. Retrieved from <https://codingbat.com/prob/p120015>
- Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue canadienne de psychologie*, 39(2), 240.
- Raj, R., Sabin, M., Impagliazzo, J., Bowers, D., Daniels, M., Hermans, F., ... Oudshoorn, M. (2022). Professional Competencies in Computing Education: Pedagogies and Assessment. In *Proceedings of the 2021 Working Group Reports on Innovation and Technology in Computer Science Education* (p. 133–161). New York: Association for Computing Machinery. doi: 10.1145/3502870.3506570
- Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137–172.
- Sanders, I., Galpin, V., & Götschi, T. (2006). Mental models of recursion revisited. In *Proceedings of the 11th annual sigcse conference on innovation and technology in computer science education* (p. 138–142). New York: ACM.
- Scholtz, T. L., & Sanders, I. (2010). Mental models of recursion: Investigating students' understanding of recursion. In *Proceedings of the fifteenth annual conference on innovation and technology in computer science education* (p. 103–107). New York: ACM.
- Schwamb, K. (1990). Mental models: A survey.
- Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7), 624–632.
- Stone, C., & Goodyear, P. (1995). Constructivism and instructional design: epistemology and the construction of meaning. *Substratum: Temas Fundamentales en Psicología y Educacion*, 2(6), 55–76.
- Tew, A. E., McCracken, W. M., & Guzdia, M. (2005). Impact of alternative introductory courses on programming concept understanding. In *Proceedings of the first international workshop on computing education research* (pp. 25–35).
- Troy, M. E., & Early, G. (1992). Unraveling recursion part ii. *Computing Teacher*, 19(7), 21–25.
- Velazquez-Iturbide, J. A. (2000). Recursion in gradual steps (is recursion really that difficult?). In *Proceedings of the thirty-first sigcse technical symposium on computer science education* (pp. 310–314).
- Von Glasersfeld, E. (1996). Radikaler Konstruktivismus. *Ideen, Ergebnisse, Probleme. Suhrkamp, Frankfurt/Main*, 375.
- Whalley, J. L., & Lister, R. (2009). The BRACElet 2009.1 (Wellington) Specification. In *Conferences in research and practice in information technology series*.
- Wiedenbeck, S. (1988). Learning recursion as a concept and as a programming technique. *ACM SIGCSE Bulletin*, 20(1), 275–278.
- Wilcocks, D., & Sanders, I. (1994). Animating recursion as an aid to instruction. *Computers and Education*, 23(3), 221–226.
- Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM Sigcse Bulletin*, 28(3), 17–22.
- Wu, C.-C., Dale, N. B., & Bethel, L. J. (1998). Conceptual models and cognitive learning styles in

- teaching recursion. In *Proceedings of the twenty-ninth sigcse technical symposium on computer science education* (pp. 292–296).
- Xinogalos, S. (2014). Designing and deploying programming courses: Strategies, tools, difficulties and pedagogy. *Education and Information Technologies*, 21(3), 559–588.

The impact of POGIL-like learning on student understanding of software testing and DevOps: A qualitative study

Bhuvana Gopal
School of Computing
University of Nebraska-Lincoln
bhuvana.gopal@unl.edu

Ryan Bockmon
School of Computing
University of Nebraska-Lincoln
ryan.bockmon@huskers.unl.edu

Stephen Cooper
School of Computing
University of Nebraska-Lincoln
stephen.cooper@unl.edu

Justin Olmanson
College of Education and Human Sciences
University of Nebraska-Lincoln
jolmanson2@unl.edu

Abstract

In this study, we analyze students' understanding of unit testing, integration testing and continuous integration in a semester long undergraduate software engineering course, after they underwent instruction using POGIL-like, a guided inquiry based pedagogy. At the end of the course, we collected student responses to open ended questions regarding their understanding of these topics, combining them with researcher memos as well as reflective researcher journals. We analyzed these written responses and identified the themes that we came up with regarding how students learned and potentially overcame difficulties with software testing and DevOps. Some of those themes that emerged from our qualitative analysis were: What makes writing and maintaining tests difficult? Where do you start unit testing a method? How do you know if you have written enough tests for the System Under Test (SUT)? How do you identify the most important functionality to test, in a SUT? Does your testing accomplish all functionality goals?

We investigate and discuss students' answers to these questions in detail. We attempt to understand if the POGIL-like approach helped students overcome some of the difficulties students expressed in our earlier work on the same topic in a previously published study.

1. Introduction

Teaching software testing is an important part of teaching software engineering. How software is tested heavily impacts how reliable the code is (Lemos, Ferrari, Silveira, & Garcia, 2015). There is often a disconnect in how testing practices are taught in undergraduate software engineering education (S. Edwards, 2004), often with little or no emphasis on real practical training (S. Edwards, 2004; Bijlsma, Passier, Pootjes, Stuurman, & Doorn, 2020). How well do students know software testing? This is often overlooked because students often find learning software testing challenging, and assessing the extent of their learning is difficult (Gopal, Cooper, Olmanson, & Bockmon, 2021).

In this paper, we attempt to discover what students learned in the topics of unit testing, integration testing and continuous integration (CI) in an undergraduate, predominantly sophomore/junior level software engineering course. Students were instructed using a collaborative pedagogy called POGIL-like (Gopal & Cooper, 2022) which is an implementation of Process Oriented Guided Inquiry Based Learning (POGIL) (Yadav, Kussmaul, Mayfield, & Hu, 2019). "POGIL" is a copyrighted term and we use "POGIL-like" to indicate our pedagogy throughout this paper. For this study, we asked students to express in their own words their understanding of the topics. We conducted a qualitative analysis of the rich set of data we obtained from student reflections of their learning, and uncovered four themes, on which we elaborate in this paper. In delineating these themes we expand on how students overcame various difficulties in learning the topics.

1.1. Overview of POGIL-like

POGIL-like is a pedagogy where students are organized into small teams. Students collaborate and actively learn to work together to co-construct knowledge on the topic being taught, using the idea of

concept invention (Kussmaul, 2011). Students start each class session with little or no prior knowledge of the topic, so they can benefit from the co-construction of knowledge through POGIL-like activities without added misconceptions. The instructor serves as an active facilitator, walking around the classroom and helping students as needed during the session. Each team consists of 4-6 students, and each student has a specific role to play. There are 4 roles: Manager, Recorder, Presenter and Reflector. Students engage with "models" and "activities". Models are a compilation of content knowledge that the students need to master. Models typically contain figures, tables, equations, and code snippets in addition to plain text. Activities contain critical thinking questions and hands-on exercises. An overview of the POGIL-like pedagogy and its salient features can be found in our previous work (Gopal & Cooper, 2022).

The student designated as the Manager keeps track of time and keeps everyone in the group focused throughout the session. The Recorder jots down everything that is being discussed during the activities. The Reflector helps the team conduct a mini retrospective after each activity and reflect on what worked well and what did not. After working through the models and activities, the student designated as the Presenter provides a summary of what the team learned during the session, and the instructor discusses the findings with the whole class. The way that students build their knowledge in POGIL-like is by exploring the models to "invent" important concepts and eventually apply what they learned (Hanson, 2005) through the co-operative, role-based interactions that they have with fellow students within their small groups. This knowledge exploring, concept inventing and application process can be utilized for both technical content and process-related content (e.g. problem solving, teamwork, and written/oral communication) (Hu, Kussmaul, Knaeble, Mayfield, & Yadav, 2016).

The rest of the paper is organized as follows. We present prior work in software testing, POGIL-like, and student reflections in Section 2. We present our research question in Section 3. Our research methods are explained in Section 4. Themes from our data analysis along with a discussion are presented in Section 5. We detail the threats to the validity of our study in Section 6, and conclude in Section 7.

2. Prior Work

Software testers in the industry often lack adequate academic preparation (Buffardi & Edwards, 2014; Wong et al., 2011; Garousi & Zhi, 2013; Chen, Zhang, & Luo, 2011; Ng, Murnane, Reed, Grant, & Chen, 2004). Several studies have been conducted in software testing education, with different research questions (Clark, 2004; Elbaum, Person, Dokulil, & Jorde, 2007; Clarke, Pava, Davis, Hernandez, & King, 2012; S. H. Edwards & Shams, 2014). Some studies have focused on challenges with software testing (Drake & Drake, 2003; Aniche, Hermans, & Deursen, 2019; Greising, Bartel, & Hagel, 2018) and explored ways to teach testing and DevOps. All these studies explored different ways to teach software testing. There are fewer studies on the impact of a specific teaching approach on how well students learned the testing topic.

Taipale and Smolander (Taipale & Smolander, 2006) found that communication efficiency mattered, and that early, risk based testing was important. They also emphasized that testing needed to be specific and tailored to business needs. Memar et al. (Memar, Krishna, McMeekin, & Tan, 2018) qualitatively studied students' evaluations of a gamified approach to teaching software testing, emphasizing the importance of feedback. Kennedy and Kraemer (Kennedy & Kraemer, 2019) studied what students' thoughts were when they were asked to "Please, think aloud" as they developed code. They analyzed video and audio recordings capturing students' thoughts in real time, and found that students showed uncertainty regardless of success at task completion.

The qualitative study by Florea and Raluca (Florea & Stray, 2020) asked experienced software testing professionals in industry what was important for software testers in terms of background, skills, learning preferences, and role profiles. They found most of their participants preferred exploratory testing. Curiosity was the most valued quality in a tester. An interesting conclusion they came to was that software testing skills needed were largely undefined, unclassified and unorganized, and increased with each new task. Most participants learned testing on the job, informally, and felt the need for better approaches to

teaching testing in post-secondary education. Their study strongly emphasized the need for educators to think outside the box of traditional methods when it comes to teaching students software testing.

Stray et al (Stray, Florea, & Paruch, 2021) qualitatively studied the human factors of Agile software testers. They found strong software testers had: the ability to see the whole picture, good communication skills, detail-orientation, structuredness, creativeness, curiosity, and adaptability. They proposed that these seven qualities be taken into consideration when organizations recruit testers for agile software.

In our previous work (Gopal et al., 2021) we conducted a qualitative study using semi-structured interviews and identified various difficulties that plagued our novice testers during the testing and DevOps process. Some of those difficulties include: communication within the team and other stakeholders, prioritization of features to be tested, entry and exit criteria for tests, difficulties with learning tools associated with testing, the time commitment involved in designing, writing and implementing meaningful tests, not knowing what kind of questions to ask and of whom, and how to look for test completeness beyond code coverage. In another study, we studied quantitatively how students answered questions on unit testing, integration testing and CI, in pre- and post-tests, after being taught using the POGIL-like approach, and found that there were statistically significant raises in student scores in the POGIL-like group compared to a pure lecture based group (Gopal & Cooper, 2022).

As we can see from the studies above, several interesting aspects of teaching and learning software testing have been explored. There is not much literature on how students express their understanding of software testing and DevOps topics, especially in the context of specific teaching and pedagogical approaches.

3. Research Question

In this paper we examine how students understood the broad topics of software testing and DevOps, within the context of a POGIL-like software engineering course. Our research question for this study was:

RQ: To what extent did undergraduate students learn and overcome known difficulties with software testing and DevOps when instructed using a POGIL-like pedagogy?

4. Methods

4.1. POGIL-like: Student and instructor roles

Our student teams consisted of 4 distinct roles (Hu & Shepherd, 2014): Manager, Recorder, Presenter and Reflector. These roles were assigned to each student to foster interdependence as well as individual responsibility and accountability to the success of the team (Kussmaul, 2012).

In our classroom the instructor was able to specifically assign groups and observe closely how students interacted with each other using the prescribed roles (Hu & Shepherd, 2013). The instructor focused on helping students develop process skills, specifically, problem solving, teamwork and critical thinking (Hu & Shepherd, 2013). The instructor offered additional guidance as students worked in teams (Hanson, 2005; Kussmaul, 2011).

4.2. POGIL-like Activity development: Models, E-I-A cycles and D/C/V questions

A model in a POGIL-like activity denotes the content that we would provide for students to know, during lectures or required readings (Maher, Latulipe, Lipford, & Rorrer, 2015), but presented with action verbs, figures, pictures and tables as needed, instead of copious quantities of plain flowing text. We used a combination of three types of questions -Directed (D), Convergent (C) and Divergent (V) questions (Gopal & Cooper, 2022). A POGIL-like learning cycle employs a series of Explore-Invent-Apply (E-I-A) activities comprised of D/C/V questions.

4.3. Study Context

This research project was determined to be exempt by our University's Institutional Review Board. Data for this study were collected from 22 participants of a cohort of 62 sophomore/junior/senior students taking a software engineering class in the Fall of 2021. All students were taught using POGIL-like on

unit testing, integration testing, and continuous integration. Students were assessed on their knowledge through quizzes on each topic (conducted a week after each topic was taught) and a final end-of-semester exam.

4.4. Data Collection

We presented students with an online questionnaire and combined them with researcher real-time memos, and reflective researcher journals. At the end of the semester, we surveyed the students an open ended questionnaire where we asked students to describe what they learned from the POGIL-like sessions in software testing and DevOps. We used simple, non-leading prompts such as "Describe what you understood about unit testing", "What were some specific things you learned about integration testing", and "What are the main features of continuous integration?".

We created and maintained reflective journals based on our real time field notes, observing students during the POGIL-like exercises (Emerson, Fretz, & Shaw, 2011). We analyzed students' written responses to the open ended questionnaire along with these field-notes. We corroborated our findings with our notes on student code patterns and quiz performance on the topics of unit and integration testing to lend support to our findings.

4.5. Content Analysis

In analyzing our data, we used a parallel approach with reflective collaborative check-ins and content analysis (Hsieh & Shannon, 2005) combined with theming, consistent with the grounded theory approach in qualitative analysis (Creswell & Poth, 2016). We began with an initial individual coding of transcripts. We generated and assigned over 500 codes. We grouped the codes into meaningful chunks and counted the frequency of each code/code group. We performed this analysis iteratively and mapped these codes into code maps, which are essentially visual layouts (Anfara Jr, Brown, & Mangione, 2002). We used the code maps to help the process of code grouping and chunking. Finally, we developed our theory based on integrating these code groups with our journals and field notes (Corbin & Strauss, 2014). Two independent coders worked on coming up with overarching themes guided by participants' own voices. This approach helped us to identify and connect the elements we both noted among the emerging themes.

4.6. Reliability

To enhance reliability, we focused on intercoder agreement based on the use of multiple coders to analyze transcript data. Two of the authors, both trained in qualitative research methods, analyzed the individual codes separately to come up with themes presented through students' responses. We utilized Cohen's Kappa (Hsu & Field, 2003) as a measure of intercoder reliability. We report an intercoder reliability (Creswell & Poth, 2016) of 1.0 (100%) among all themes.

4.7. Data Organization

In the following data sections, our aim is to highlight and bring to the forefront, the voices and experiences of our participants. We have followed existing research guidelines on how to position student participation, and our method for meaning making involves understanding recurring expressions of participant sentiments and ideas by taking into account the context in which they were written and submitted (Ketelhut & Schifter, 2011; Foley, 2002). We elucidate a coherent set of data presentations in the following section, highlighting and bringing to the forefront the voices of participants through their written submissions, and utilizing students' own words in the subheadings. We have anonymized the names of students and used pseudonyms instead. In this study, our focus was to see if students had gained any further understanding on the content topics through POGIL-like, and we used our earlier pilot study (Gopal et al., 2021) to inform us of the difficulties that students faced while learning testing and DevOps. Utilizing POGIL-like, did students learn to overcome the difficulties we uncovered earlier? This is the primary focus of our analysis.

5. Analysis and Discussion

In this section we write about several thoughts and impressions of software testing and DevOps, from our participants, employing their own words. We first elaborate what students felt, within the context of learning with POGIL-like, what made learning testing difficult. Next, we present how they determined entry points into the System Under Test (SUT), followed by how they would identify the most important functionality to test. We then present how they determined when to stop testing, and determine test completeness. We conclude with their understanding of whether testing relates to requirements or not.

5.1. Theme 1: What makes writing and maintaining tests and CI pipelines difficult?

"Simply put, change. Code changes, expected behavior requirements change, the framework you work on may have even changed. Keeping up with unit tests, integration tests and maintaining regression testing can cumbersome over time." - Alice.

The fundamental difficulty in testing was the volatility of the entire system and its environment. Alice honed in on this very important problem and displayed an understanding of testing needs changing as a result of changing requirements, and hence code. She very elegantly summarized all the subsystems that could be involved in a cascade of changes when requirements change.

"I think the hardest part about writing tests is to account for all possibilities of regular output and for possible edge cases that could take place.....Testing for edge cases will just happen as you test your application. It makes it hard to maintain tests because as new features are added you will have to adapt your tests based off that." - Ben.

Ben brought a different difficulty to the forefront - edge cases. Edge test cases describe possible but unknown scenarios - which could very well be present at any stage of development. Testers need to be on a constant lookout for edge cases. Boundary testing is a useful technique to find edge cases, specifically with extreme input values. Students often find testing for edge cases difficult since it is hard to know all edge cases that can happen. To know where the boundary conditions are, one needs to know where the non boundary conditions are- where the normal test cases lie. Ben's statements also indicate that the ever changing nature of the codebase made it difficult to keep up with edge test cases.

It is interesting to note that while we uncovered two distinct difficulties with learning testing in our analysis above, the other difficulties we discovered in our previous study (Gopal et al., 2021) were absent.

5.2. Theme 2: Entry and exit points, anatomy of a unit test, integration testing, test doubles, setting up CI, new bugs

"Start with the least complicated portions of the code first and then the most complicated ones." - Lisa.

Lisa's approach to starting where to test involved a recognition of what was complicated in the method. This shows that the tester needs to have an overall picture of what the method is trying to accomplish and what parts of it are complicated.

"When I am testing a method, I start by testing to make sure it is receiving the correct inputs. This means the parameters are passed in properly, and the method receives the necessary data." - Matt.

Matt's approach focused on the inputs and parameters required to call the method correctly. This is a good way to start, without necessarily knowing what the actual complexity of the method is.

"Then, I test the method by running it locally." - Matt.

Matt explains that the method is run "locally" - leading us to believe that he meant that the method is called inside the test method's test context, inside a unit test.

"To test a method by Unit testing, there are three steps: Arrange, Act, and Assert. First you arrange the test and set up everything necessary for it. Second you act, which are the steps taken needed to do the testing of the method. Lastly you assert, which is to look at the outcome and check if it is what's expected." - Gabe.

"I begin by creating a test class to contain my test cases. In each test case, I call the method directly. This includes arranging, acting, and asserting. I finish by running all my test cases through the Test tab in Visual Studio and making sure each one passes." - John.

"We use the process of Arrange, Act, Assert. First, we must arrange, or set up and initialize a method to be tested. Secondly, we act upon that method, which means just executing it. Lastly, we will assert, or return a pass or fail value to show if we have passed or have failed the test." - Molly.

Several students, including Gabe, John, and Molly utilized the Arrange-Act-Assert concept to start testing. This was an intended learning outcome for both unit and integration testing modules, and unlike prior instruction using lecture or peer instruction (Gopal et al., 2021), students in this study seemed to have a better understanding of how to start testing a method.

"All components that will be tested in integration test should be already have unit tests." - Tia.

"Then, I integrate it with the rest of the system and test it there. This is important because it has to function properly with the other methods in the system, otherwise changes have to be made to allow them to integrate properly." - Matt.

With the above statements, we see that Matt and Tia had an understanding of integration testing in combination with unit testing. Testing a method does not simply mean that it works correctly within the test context in an isolated fashion, but it also means that the method works with the other methods it interacts with, other subsystems it depends on, and integrates seamlessly with the entire SUT.

"The unit tests should cover the workflows that the method will handle. I would generally start with a valid case, followed by broken data, whether it is malformed data, no data, etc. I begin by creating a test class to contain my test cases. In each test case, I call the method directly. This includes arranging, acting, and asserting. I finish by running all my test cases through the Test tab in Visual Studio and making sure each one passes." - Pranav.

Pranav's explanation of how to start testing a method demonstrates an understanding of several key concepts in how to start testing: workflows, valid inputs, broken/malformed/lack of data, the arrange-act-assert paradigm, and the tool support needed to run tests in a .NET environment (which is the technology stack the students learned).

"When bugs arises, add a test case for that bug, and modify the method to resolve that bug. Start with unit tests. Make sure to mock out any inputs that you expect and any dependencies." - Sam.

"When testing a method, I would first see if there are any obvious bugs like missing syntax. Have variables properly labeled. Have variables initialized and check for null values. Then create a Unit Test using a unit test framework to test each method in isolation. The Unit Test should follow an arrange, act, and assert pattern. Create a test following the pattern with arrange is to set up the objects or the environment for the testing, the act is to call the function, and assert is to verify if the actual output is with the expected output using the assert collection. Then run the test and analyze any test that fails so that the problem can be rectified." - Macy.

Sam and Macy focused on how we unit test bugs. Macy echoed the arrange-act-assert pattern as being vital to implementing a unit test. Sam explained that mocking dependencies is important. These students displayed a deep understanding of how the testing process works, accounting for new bugs that arise in the code, and using test doubles when needed. This is in direct contrast to the lack of understanding of test doubles and new bugs they displayed in our earlier work (Gopal et al., 2021).

5.3. Theme 3: Prioritizing test functionality and what to continuously integrate

In the following subsections we focus on what our participants told us regarding how they identified the most important functionality to test, and how they determined when they had tested enough.

"First, understand what the system is supposed to accomplish. I believe the most important functionality to test would be to test the logic or computational functions. If the computational functions are not

correct then the system would produce erroneous output. The entire purpose of any program is to have well-defined business rules implemented through logic to produce the correct output." - Alex.

"I'd recommended to look over the project documentation . If we see which part of the project being influenced and used by a lot of methods , we will prioritize testing it." - Kara

Alex and Kara explained in detail how they identified the most important functionality to test. Particularly noteworthy is his understanding of business rules and requirements being closely related to testing. Referring to documentation is another technique, but sometimes this might not be feasible, especially for greenfield projects.

"Another way to identify the most important functionality is to see which methods get called often." - Alex.

Frequency mapping of method calls is one of the methods that is commonly used in the software industry to see which methods need to be tested first, and as a novice tester, Alex seemed to have obtained a grasp of this useful technique. This could also be due to a prior industry internship, and not just an effect of the classroom instruction he underwent using POGIL-like.

5.4. Theme 4: Test completeness beyond code coverage

"Testing has accomplished all functionality goals when we have detected all known bugs and can prevent them. When the clients and stakeholders are satisfied with the product. When sensitive data is protected and tested in transit and rest." - Daniel.

"For more complex, abstract code you may never catch every test case, so you may never know for 100% certainty if it will function as intended. Even though you can try to cover every possible way to cover you code, there may just be one small input, or parameter that will break it, that no one accounted a test case for." - Abe.

Daniel and Abe displayed a good understanding of how test completeness can be determined. Particularly noteworthy is that beyond having enough tests to cover all possible scenarios, Daniel mentioned stakeholder satisfaction and sensitive data testing was accomplished, acknowledging that sensitive data could be at rest or in transit and need to be tested in both scenarios. Abe's acknowledgement that it is virtually impossible to completely test a complex system displays a nuanced understanding of the testing process.

6. Implications for teaching and threats to validity

Based on our data analysis, we found that students did not focus on DevOps enough. They understood various concepts behind and application scenarios of unit and integration testing, but were notably silent on continuous integration, its importance or benefits.

In any qualitative study, validity is expressed in terms of researcher bias, reactivity and respondent bias (Lincoln & Guba, 2006). As for researcher reflexivity, the primary author of this paper acknowledges that her extensive software industry background positions her to interpret the data with a decidedly industry focused bent. To mitigate the effects of these aspects, we followed some of the suggestions by Robson (Robson, 2002). We engaged with our students throughout the semester, and did not just ask them for responses on the written questionnaire. Students whose responses we used had all given their informed consent to the study. Students' written answers were a reflection of our semester-long involvement with them. We triangulated the data we obtained from their written responses with their quiz performances and our audit trail, to minimize the chances of students simply copying what they think is the right answer. We kept the entire process of data collection, analysis and dissemination transparent to the students, and tried to eliminate any potential "people-pleasing" answers.

We also followed recommended guidelines for saturation on the topic studied (Creswell & Poth, 2016). However, even with our systematic analysis, it is possible that other researchers may distill different themes and ideas than ours from the same raw data. Since we triangulated our data from 22 student

responses from a single cohort with the same instructor and instructional pedagogy with the same topics of instruction, we deem our findings to be valuable and relevant within the context of our study.

In our previous study (Gopal et al., 2021) we found several difficulties that students faced while learning software testing and DevOps. The pedagogy of choice in that study was peer instruction (Mazur, 1997). With the same content, same instructor, and similar prior academic preparation, but with the instructional mode being POGIL-like, our participants seemed to have overcome many of the difficulties expressed previously.

Students in our study exhibited an understanding of how to start testing a method. They understood the anatomy of a unit test, and how test doubles could be used in both unit and integration testing. They realized that test completeness went beyond code coverage metrics, and recognized the importance of edge cases and boundary conditions. In our previous study, most of the communication issues that our participants expressed were caused by late stage testing - in this study, our students, started testing early, and tested often, which resulted in no significant mentions of communication issues.

7. Conclusion and future work

In a nutshell, understanding business priorities, starting to test early and testing often, knowing that test completeness depends on stakeholder agreement, prioritizing tests, understanding entry points, recognizing the importance of integration testing, and relying on existing documentation are the major themes that POGIL-like instruction, with its emphasis on process and activities seems to have achieved on our students.

In answering our research question, "To what extent did undergraduate students understand software testing and DevOps when instructed using a POGIL-like pedagogy?", we conclude that POGIL-like is an effective way to teach unit testing, integration testing and continuous integration. POGIL-like helped students gain understanding beyond the surface level, well into the higher layers of Bloom's taxonomy (Bloom, 1984). We surmise that the heavily process oriented nature of the POGIL-like pedagogy, with its E-I-A cycles and D/C/V questions, forced students to think deeper, and go beyond a surface understanding of the "How" to involve more of the "Why?".

In future work we intend to focus on student understanding of CI, and its role in enhancing transparency and visibility to the stakeholders. We wish to explore the efficacy of active and collaborative learning approaches including POGIL-like in a focused CI environment using the Agile methodology.

8. References

- Anfara Jr, V. A., Brown, K. M., & Mangione, T. L. (2002). Qualitative analysis on stage: Making the research process more public. In (Vol. 31, pp. 28–38). Sage Publications Sage CA: Thousand Oaks, CA.
- Aniche, M., Hermans, F., & Deursen, A. (2019). Pragmatic software testing education. In *Proceedings of the 50th acm technical symposium on computer science education (sigcse '19)*, acm (p. 414–420). New York, NY, USA.
- Bijlsma, L., Passier, H., Pootjes, H., Stuurman, S., & Doorn, N. (2020). *How do students test software units? part one: Their natural attitude diagnosed* (Technical Report. Open Universiteit,). Faculty of Science, Department of Computer Science.
- Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. In (Vol. 13, pp. 4–16). Sage Publications Sage CA: Thousand Oaks, CA.
- Buffardi, K., & Edwards, S. (2014). A formative study of influences on student testing behaviors. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 597–602).
- Chen, Z., Zhang, J., & Luo, B. (2011). Teaching software testing methods based on diversity principles. In *2011 24th ieee-cs conference on software engineering education and training (csee t* (p. 391–395). Honolulu, HI, USA.
- Clark, N. (2004). Peer testing in software engineering projects. ACM Digital Library.
- Clarke, P., Pava, J., Davis, D., Hernandez, F., & King, T. (2012). Using wrestt in se courses: An empirical study. In *Proceedings of the 43rd acm technical symposium on computer science education (sigcse '12)*, acm (p. 307–312). New York, NY, USA.
- Corbin, J., & Strauss, A. (2014). *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications.
- Creswell, J. W., & Poth, C. N. (2016). *Qualitative inquiry and research design: Choosing among five approaches*. Sage publications.
- Drake, J., & Drake, J. (2003). Teaching software testing: Lessons learned. Citeseer.
- Edwards, S. (2004). Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th sigcse technical symposium on computer science education* (p. 26–30).
- Edwards, S. H., & Shams, Z. (2014). Do student programmers all tend to write the same software tests? In *Proceedings of the 2014 conference on innovation & technology in computer science education* (pp. 171–176).
- Elbaum, S., Person, S., Dokulil, J., & Jorde, M. (2007). Bug hunt: Making early software testing lessons engaging and affordable. In *29th international conference on software engineering (icse'07)* (pp. 688–697).
- Emerson, R. M., Fretz, R. I., & Shaw, L. L. (2011). *Writing ethnographic fieldnotes*. University of Chicago Press.
- Florea, R., & Stray, V. (2020). A qualitative study of the background, skill acquisition, and learning preferences of software testers. In *Proceedings of the evaluation and assessment in software engineering* (pp. 299–305).
- Foley, D. E. (2002). Critical ethnography: The reflexive turn. In (Vol. 15, pp. 469–490). Taylor & Francis.
- Garousi, V., & Zhi, J. (2013). A survey of software testing practices in canada. In (Vol. 86, p. 1354–1376).
- Gopal, B., & Cooper, S. (2022). POGIL-like Learning in Undergraduate Software Testing and DevOps - A Pilot Study. In *Proceedings of the 27th annual acm conference on innovation and technology in computer science education (iticse)* (p. Accepted.).
- Gopal, B., Cooper, S., Olmanson, J., & Bockmon, R. (2021). Student difficulties in unit testing, integration testing and continuous integration: An exploratory pilot qualitative study..
- Greising, L., Bartel, A., & Hagel, G. (2018). Introducing a deployment pipeline for continuous delivery in a software architecture course. In *Proceedings of the 3rd european conference of software engineering education* (p. 102–107).

- Hanson, D. (2005). Designing process-oriented guided-inquiry activities. In (pp. 1–6).
- Hsieh, H.-F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative health research*, 15(9), 1277–1288.
- Hsu, L. M., & Field, R. (2003). Interrater agreement measures: Comments on kappan, cohen's kappa, scott's π , and aickin's α . In (Vol. 2, pp. 205–219). Taylor & Francis.
- Hu, H., Kussmaul, C., Knaeble, B., Mayfield, C., & Yadav, A. (2016). Results from a survey of faculty adoption of process oriented guided inquiry learning (pogil) in computer science. In *Proceedings of the 2016 acm conference on innovation and technology in computer science education* (pp. 186–191).
- Hu, H., & Shepherd, T. (2013). Using pogil to help students learn to program. In (Vol. 13, pp. 1–23). ACM New York, NY, USA.
- Hu, H., & Shepherd, T. (2014). Teaching cs 1 with pogil activities and roles. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 127–132).
- Kennedy, C., & Kraemer, E. T. (2019). Qualitative observations of student reasoning: Coding in the wild. In *Proceedings of the 2019 acm conference on innovation and technology in computer science education* (pp. 224–230).
- Ketelhut, D. J., & Schifter, C. C. (2011). Teachers and game-based learning: Improving understanding of how to increase efficacy of adoption. In (Vol. 56, pp. 539–546). Elsevier.
- Kussmaul, C. (2011). Process oriented guided inquiry learning for soft computing. In *International conference on advances in computing and communications* (pp. 533–542).
- Kussmaul, C. (2012). Process oriented guided inquiry learning (pogil) for computer science. In *Proceedings of the 43rd acm technical symposium on computer science education* (pp. 373–378).
- Lemos, O., Ferrari, F., Silveira, F., & Garcia, A. (2015). Experience report: Can software testing education lead to more reliable code?. In *2015 ieee 26th international symposium on software reliability engineering (issre)* (pp. 359–369).
- Lincoln, Y., & Guba, E. (2006). *Naturalistic inquiry*. Newbury Park: Sage Publications.
- Maher, M. L., Latulipe, C., Lipford, H., & Rorrer, A. (2015). Flipped classroom strategies for cs education. In *Proceedings of the 46th acm technical symposium on computer science education* (pp. 218–223).
- Mazur, E. (1997). *Peer instruction a user's manual*. Prentice Hall.
- Memar, N., Krishna, A., McMeekin, D. A., & Tan, T. (2018). Gamifying information system testing—qualitative validation through focus group discussion.
- Ng, S., Murnane, T., Reed, K., Grant, D., & Chen, T. (2004). A preliminary survey on software testing practices in australia. , 116–125.
- Robson, C. (2002). *Real world research: A resource for social scientists and practitioner-researchers*. Wiley-Blackwell.
- Stray, V., Florea, R., & Paruch, L. (2021). Exploring human factors of the agile software tester. *Software Quality Journal*, 1–27.
- Taipale, O., & Smolander, K. (2006). Improving software testing by observing practice. In *Proceedings of the 2006 acm/ieee international symposium on empirical software engineering* (pp. 262–271).
- Wong, W. E., Bertolino, A., Debroy, V., Mathur, A., Offutt, J., & Vouk, M. (2011). Teaching software testing: Experiences, lessons learned and the path forward. In *2011 24th ieee-cs conference on software engineering education and training (csee&t)* (pp. 530–534).
- Yadav, A., Kussmaul, C., Mayfield, C., & Hu, H. (2019). Pogil in computer science: Faculty motivation and challenges. In *Proceedings of the 50th acm technical symposium on computer science education* (pp. 280–285).

Do mathematical proof skills in continuous areas of Maths develop algorithmic thinking in CS students in HE?

Julia Crossley
Department of Computer Science
City University of London

Abstract

The aim of this proposal is to suggest a possible mechanism for developing abstract skills in Computer Science students during their undergraduate studies. Some research has gone into the development of conceptual skills in school aged students and into the meaning of abstraction. My aim is to develop methods using the idea of Mathematical proofs, to help students generalise, which will help them develop self-efficacy and confidence in using their intellectual curiosity. Students will be exposed to formal proofs, and most of all encouraged to think of some themselves.

Introduction

The aim of my project is to identify whether exposure to general systematic and iterative processes in Maths help non-mathematicians develop stronger algorithmic skills; whether linking discrete and continuous processes helps students produce more efficient code. In doing this, I hope to investigate whether this will help them:

- Generalise results in order to trace bugs and possibly avoid them.
- Identify exceptions or limiting cases.
- Find expressions that result in less time and space complexity. Some expressions may not be common mathematical results that are easy to find – the computer scientists may need to work out the expression themselves!
- Calculating the complexity of an algorithm may also require some mathematical skill – particularly if the computer scientist needs to approximate an expression in a function.

Mathematical Proofs

Mathematical proofs are covered in first year undergraduate degrees in Maths and rarely touched on earlier. This area of pure Maths is often thought to be the most novel and difficult (Almeida, 2010), perhaps due to the following reasons: complex mathematical language and deeper, systematic thinking that may at times seem counter intuitive. The skills gained contribute to a more insightful approach to the broader subject and better abstract reasoning (Schoenfeld, 2009). Students encounter mathematical subtleties, learn to distinguish between them and use them.

Computer Science students as a group rarely study formal mathematical proofs, instead taking a more practical, ‘methods based’ approach to Maths in the same way Engineers and Scientists do. If they do, these are in areas seen as directly relevant to fundamental CS, such as discrete Maths, and are ‘case’ proofs rather than ones of general results. Studies have been conducted into measurement of long-term skills in these areas of Maths (Qian & Lehman, 2017); a comparable measurement in continuous and other areas of maths could be of use to CS educational research, particularly if the impact of these gained skills can be seen at a fundamental and advanced level.

Difficulties in Computer Science and Maths

Qian & Lehman (2017) identify Maths as an area causing difficulty, in that adherence to knowledge gained at secondary school impacts understanding of concepts in computer science. I would like to

take this further: can this skills gap be tackled by more exposure to continuous and deep Mathematics language and processes? Many CS programs do not emphasize this, particularly at the early stages, and this is seen as detrimental by some even early on (Baldwin, Walker & Henderson, 2013). The latter manifests itself as using a trial-and-error approach to Maths and programming.

Cognitive overload can be a problem, particularly when facing numerous levels of abstraction and more ‘practical’ considerations such as memory, types, etc. Lack of clarity over the meaning of symbols and how they are processed in a programming language can also be an area of confusion for students.

The practice of Mathematical Proofs certainly provides deeper insight for Mathematics students into abstract processes, by guiding them in breaking down concepts, understanding them and identifying both linguistic and mathematical subtleties. Can it do something similar for Computer Science students, in a manner that can be contextualised in a computer program?

Some research has gone into this, focusing on the areas of discrete maths and logic, as they can be applied to Computer Science at a more fundamental level. Other areas of Maths can be of use but are more relevant to advanced and applied areas of Computer Science (Hartel, Van Es, Tromp, 1995). I’d like to investigate whether proofs in these other areas of Mathematics can also offer insight for students through the abstract skills required to understand and use them. Proofs that require an iterative process and use continuous mathematical variables have the potential to help students understand algorithmic processes on a more abstract level.

My project

The method of instruction would be pattern-oriented, as suggested by Muller and Haberman, in that fundamental concepts would be illustrated through different processes (Muller and Haberman, 2008). My reason for choosing these mathematical processes, is somewhat “soft” (Hazzan, 2008), in that many of them present important CS concepts such as nesting, iteration, recursion and perhaps even arithmetic overflow using objects and syntax that is already somewhat familiar to students. There are also some similarities between learning to code and learning to prove: students ‘trace’ proofs; develop an intuition for the appropriate tools to use for a given problem, such as a mathematical inequality; use this intuition to implement a proof.

Specific areas of Mathematics to present to students also need to be decided. My original idea was to use proofs illustrating steps from discrete Mathematics to continuous ones, for example by starting from patterns in sequences and series, then leading on to continuous functions. The reason for this is that many proofs in these two areas are similar, which makes the progression between these two types of sets more natural. It may however be that students benefit more from having a deeper look into the uses of Mathematics in areas they are exposed to at the earlier stages of their CS education, and this would likely be in Discrete Mathematics.

Methods

Some work still needs to go into identifying appropriate methods for the study. The goal of the study is to build on the metacognitive skills of a group of students who will benefit the most from the intervention. My tendency would be primarily to choose an experimental group of students with a broad range of experience in Mathematics, but nevertheless a willingness to learn a new approach and curiosity in a variety of areas including Maths. Students who embody this mindset, despite having lower confidence in their Maths skills, will also be strongly encouraged to take part. The success of the intervention will be determined both from their project process and output, as well as their reported increase in confidence in their coding skills and ability to use new tools. Students will then

have been equipped with a toolkit of mathematical expressions and arguments that will be useable to them in their career as Computer Scientists.

References:

- Almeida, D.(2010) A survey of mathematics undergraduates' interaction with proof: some implications for mathematics education. *International Journal of Mathematics education in Science and Technology*, vol 31, 2000 – issue 6, pages 869 – 890.
<https://doi.org/10.1080/00207390050203360>
- Baldwin, D., Walker, H., Henderson, P. (2013). The roles of mathematics in computer science. *ACM inroads*, Volume 4, Number 4 (2013), Pages 74-80.
- Cetin, I., Dubinsky, E. (2017). Reflective abstraction in computational thinking. *The Journal of Mathematical Behaviour*, 47, 70-80. <https://doi.org/10.1016/j.jmathb.2017.06.004>
- Hartel, P.H., van Es, B., Tromp, D. (1995). Basic proof skills of computer science students. In: Hartel, P.H., Plasmeijer, R. (eds) *Functional Programming Languages in Education. FPLE 1995. Lecture Notes in Computer Science*, vol 1022. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-60675-0_50
- Hazzan, O. (2008). Reflections on Teaching Abstraction and Other Soft Ideas. *SIGCSE Bull.*, 40(2), 40–43. <https://doi.org/10.1145/1383602.1383631>
- Ko, Y., Knuth, E. (2013) Validating proofs and counterexamples across content domains: Practices of importance for mathematics majors. *The Journal of Mathematical Behaviour*, Volume 32, Issue 1, 2013, Pages 20-35, ISSN 0732-3123. <https://doi.org/10.1016/j.jmathb.2012.09.003>
- Loksa, D., Ko A., Jernigan, W., Oleson, A., Mendez, C. & Burnett, M. (2016). Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. 1449-1461.
10.1145/2858036.2858252.
- Muller, O., Ginat, D. & Haberman, B. (2007) Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM, New York, 151–155. <http://dx.doi.org/10.1145/1268784.1268830>
- Schoenfeld, A. (2009) *Teaching and learning proof across the grades: A K-16 perspective*. Routledge, New York, NY (2009), pp. xii-xvi
- Qian, Y. & Lehman, J. (2017) Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (March 2018), 24 pages. <https://doi.org/10.1145/3077618>
- Zeng, H. & Jiang, K. (2010). Teaching mathematical proofs to CS major students in the class of discrete mathematics. *Journal of Computing Sciences in Colleges*. 25. 326-332.

Tutorials Embedded in an IDE: A Feasible Way for CS Students to Learn Debugging? – A Study Design

Olli Kiljunen
Aalto University
olli.kiljunen@aalto.fi

1. Introduction

Although the importance of teaching novice programmers to debug computer programs is well-noted (*e.g.* McCauley et al., 2008; Li, Chan, Denny, Luxton-Reilly, & Tempero, 2019), finding optimal ways to help students to learn debugging is still an ongoing quest. Nevertheless, a common understanding among both the computing education researchers and practitioners seems to be that debugging, possibly due to its complex and often complicated nature, is a difficult subject for students in their early phases of programming education.

The goal of my PhD thesis, is to gain new insight into how debugging can be learnt and develop novel ways to teach CS students to debug. As a part of that, in this doctoral consortium abstract, I present and outline my plans for an evaluative study that tries to answer the question whether a certain learning tool and method – designed and developed by me in collaboration with our research group – would carry potential for becoming a both efficient and practical way to teach debugging.

The learning method I have been developing is based on interactive software tutorials that students take using an educational tool embedded in a programming environment, IDE. In my planned study, I test this learning method with novice CS students to see how they interact with the tool, whether they feel satisfied when learning with it, which features of it they found either useful or useless, and what kind of debugging skills or knowledge can be taught with it. The results are hoped to provide better understanding on how teaching of debugging can be improved and help me in the further development of this method of learning.

2. Background and wider context

Previous computing education research literature has suggested various approaches to teach debugging. For example, Carter (2014) used interactive software tutorials in her study with promising results. Those tutorials, however, take place in an exercise environment that is specifically designed for them and not otherwise used by the students.

Social constructivism and sociocultural learning theories emphasise the situated nature of learning and the learner's interaction with their environment and tools. In order to better align the tutorial-based approach with the understanding of socioculturally mediated learning, the learning method I am suggesting places the tutorials within an authentic programming environment – that is, a fully-featured, professional-grade IDE.

The study I am planning to carry out this autumn is a part of the larger continuum of my PhD thesis. The series of studies – of which this one is the first – takes the form of design-based research where the suggested teaching method and tool are developed, put into use, and evaluated iteratively. Drawing conclusions of my observations on each iteration, I aim at not only coming up with new learning methods but also contributing to the theory of how debugging can be taught and learnt.

3. Description of the tool

The tool I have developed is a software system that integrates into an authentic programming environment and allows students to work with interactive tutorials and/or exercises. In the context of this study, the system guides a student through various debugging tutorials step-by-step. At each step, the tool visu-

ally highlights those parts of the programming environment that the student is expected to pay attention to and interact with. It also provides students with hints and suggestions about what they should try next to be able to proceed in their debugging process.

The tool is implemented as a plug-in for IntelliJ IDEA development environment. The concept and underlying design could, however, be ported to mostly any modern programming environment.

4. Planned observation and analysis

In my study design, I plan to recruit a few (4–6) student participants currently taking their first Computer Science course with no or very little preliminary experience in programming. The study will be carried out at the stage of the course when the students have been taught some basic programming language constructs and they have written a couple of simple programs as exercises. At that point, they have not, however, received explicit training on how to debug programs.

The student subjects are taken one by one into an observation room where there is a computer on a desk and a researcher acting as a supervisor of the study. The subjects' interaction with the computer as well as their discussion with the supervisor are recorded. Before starting the study, each subject is asked for their previous knowledge of programming and debugging as well as familiarity of the programming and debugging tools the study involves. In the beginning, the computer has IntelliJ IDEA open with some buggy program's source code. The subject is asked to debug the program – that is, to locate the error in the code and fix it or at least verbally suggest a fix. For this task, subjects are given a few minutes, after which – whether successful or not – they are interviewed about their performance and how they felt about it.

After that, the study proceeds to its second phase, where our debugging tutorial tool is introduced to the subject. A new buggy program is opened for them, and they are asked to debug it following the tutorial. The procedure is repeated so that each subject goes through three tutorials in total. After the tutorials, subjects are asked about how they felt using the tutorials, what they think they learnt, and what they liked and what they did not like in the tutorials and why.

In the last phase, subjects are again given a buggy program to debug but this time without a tutorial. The time limit is similar as in the first phase. Finally, subjects are interviewed of their overall experience. They are asked to reflect on what they learnt when using the tutorials and how they used that knowledge in the last phase.

The target of this study is not to answer the question whether or not our tutorial system is a superior way to learn debugging or – by any means – compare it to other learning methods. Instead, the observed data is scrutinised to understand whether students find this way of learning intuitive and meaningful and whether it can teach them at least some aspects of debugging. Moreover, I analyse the observed data to recognise such interactions with the tutorials where students failed to follow the intended course of actions or felt frustrated, confused, or irritated about the tutorials. From these results, I seek to draw conclusions on how the tutorial system should be developed further to best serve its purpose. Comparing this method of learning debugging to other learning methods is a potential question of future research, based on the results I get.

5. Expectations for the doctoral consortium

By taking part in the PPIG 2022 Workshop and, in particular, its doctoral consortium, I expect to be able to review and refine my study plan based on the discussions with the other participants and the feedback I get. Furthermore, I hope those discussions and working sessions will provide me with creative ideas and viewpoints that will open new interesting directions for my research in the areas of debugging and learning. Also, I would be more than delighted if participating in the doctoral consortium sparked some future collaboration with other researchers who have overlapping areas of academic interest.

6. References

- Carter, E. (2014). *An intelligent debugging tutor for novice computer science students* (Doctoral dissertation, Lehigh University, Bethlehem, PA, USA). Retrieved from <https://preserve.lib.lehigh.edu/islandora/object/preserve%3AAbp-7256300>
- Li, C., Chan, E., Denny, P., Luxton-Reilly, A., & Tempero, E. (2019). Towards a framework for teaching debugging. In *Proceedings of the twenty-first australasian computing education conference* (pp. 79–86).
- McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2), 67–92.

About me

My name is Olli Kiljunen, and I am a second-year doctoral student in the field of *Computing Education Research* (CER) at Aalto University, Finland. I am writing my thesis as a member of *Learning+Technology Research Group* and under the supervision of Professor Lauri Malmi. In this study, I collaborate with DSc Otto Seppälä and DSc Juha Sorva.

I live in a suburban area in Helsinki, Finland. In addition to programming and education research, my interests include various forms of drama and improvisational theatre, where I fluctuate between the roles of actor, director, playwright, composer, and spectator.

An Investigation of Student Learning in Computing Education Research

Jude Nzemeke
judenzemeke@yahoo.co.uk

Literature Review

The offering and delivery of computer science has evolved over the years in the United Kingdom and the response of different institutions has been mixed. When The Rt Hon Michael Gove who was Education secretary in 2013, announced that “harmful” ICT curriculum will be replaced with a more rigorous computer science in schools [Department of Education 2013], a lot of schools were not prepared for this change. They felt the government had not put careful consideration into the challenges of retraining staff to give them the confidence to deliver a new curriculum. Since most schools lacked the expertise to deliver the computer science curriculum, they stopped offering it to pupils as an option. There has recently been a resurgence in the take up of the subject in schools especially with its inclusion as part of English Baccalaureate however, most schools still face challenges with delivery due to lack of expertise [Brown et al 2014].

The impact of this resurgence has been felt in universities in the United Kingdom with 129,610 students opting to study computer science in the year 2021 – a 68.63% increase when compared with the total of 76,860 applicants in 2012. We have more women who have applied to attend university in 2021 (58.93% of applicants) than men (41.07% of applicants). However, further analysis of the data shows that fewer women opt to study computer science at university – 17.52% (22,710 applicants) when compared with men – 82.48% (106,900 applicants) [UCAS 2021].

How should computer science be taught?

A number of professionals and institutions have done research on different teaching pedagogies. Amongst these are the Gradual Release of Responsibility model by Pearson and Gallagher (1983) and the Fisher and Frey model (2013). These models allow teachers demonstrate new skills first and allows learning to gradually progress to the stage where students can independently complete tasks.

The Semantic Waves model, which is an adaptation of the Pedagogical Content Knowledge by Shulman in 1986, expands on the idea and describes processes that means that computer science teachers who are subject specialists should differ their approach when teaching pupils in lessons from when conversing with fellow subject specialists [Maton 2013]. These models allow teachers to use effective metacognitive processes in the delivery of their lessons however, there is a need for further research on the impact of these paradigms in the delivery of computer science in schools [Curzon et al 2020].

Research Focus

- Computational Thinking (CT): CT is a life skill that can be applied not just in the field of computing but also in everyday tasks. However, it is sometimes the case that teachers use programming tasks to conceptualize CT whereas, a holistic approach in using CT when completing tasks may be more effective. Although elements of CT including abstraction, decomposition, thinking ahead and visualisation are included in the curriculum as individual units, the challenge is how often students apply these methods when solving problems [Steinmayr et al. 2019].
- The role/purpose of visualisation learning in computer science: The British Council in an

article on visualisation described it as involving the creation of real or unreal images in the mind's eye. Teachers can use visualisation to model information in ways that allow students understand new concepts better. There is an opportunity in this research to explore how computer science teachers use visualisation and how effective it is in aiding pupils with memory recall [McDaniel and Einstein 1986].

- Relationships between students' wider characteristics (e.g. ability, motivation): it will be easy to think that students who are motivated and those with ability should excel academically. It is however vital that these hypothesis are tested and evidence based.
- Misconceptions, understandings and misunderstandings of core CS concepts from CS1 through to CS3: a number of factors lead to general misconceptions in CS – these include how challenging a task is, students problem solving skills and the competence of teachers [Qian and Lehman 2017].

Research Questions

With the key focus above in mind, I have been able to deduce the following research questions:

- To what extent do students use CT techniques to understand new concepts better?
- How well/often do students use computational methods when solving problems? For example, if faced with a complex programming task, are they able to decompose it first before they start solving it?
- How do computer science teachers use visualisation and how effective it is in aiding pupils with memory recall.
- Do teachers use visualisation to create a vibrant atmosphere in classrooms that encourage learning to take place?
- Do students do well in computing because of their ability to grasp new content or think logically?
- Does the teaching style of particular teachers influence how students perceive their understanding of content and their motivation?
- Does feedback matter, what role does it play in the learning journey of students?
- Do CS teachers and students identify misconception?
- To what extent does misconceptions impact upon learning – if left unclarified?
- What impact does it have on the wellbeing of staff and pupils?

Proposed Methodology

I intend to use both qualitative and quantitative methods in gathering data for the research. Other methodologies I would use include interviews with staff and students, lesson observations, expert opinions, working with focus groups and literature reviews.

I anticipate my research will require the use of “big data”. I intend to use data analysis software to interpret and clean up collated data. Software will also be used to incorporate visualisation in the presentation of information and to show relations that may exist between datasets.

I will explore the use of what I will call a “Three Box Model” when teaching students. The idea here is to get teachers to compartmentalize new concepts when teaching and allow pupils to do the same, when solving problems – see table 1 and table 2 below.

Box 1		Box 2		Box 3
Introduce concept		Demonstrate key skills [Complete task together with students]		Students complete task independently
Check understanding		Check understanding		Check understanding

Table 1: Proposed Three Box Model for Teachers when teaching

Box 1		Box 2		Box 3
Read the task		Break task into solvable chunks.		Solve each chunk independently
Ask questions if any part of task is unclear		Seek help if any chunk is too challenging		Bring each chunk together.

Table 2: Proposed Three Box Model for Students when completing task

Scope of research and outreach

I will like to widen the scope of my research to include students in KS5, CS1 to CS3 in England and possibly Northern Ireland. I hope to throw more light on different teaching pedagogies that students are exposed to at these stages and test the hypothesis of the Three Box Model. It would also mean that findings from my research would inform educators and learning institutions on best practices for delivery of computer science across the spectrum.

Why I wish to pursue this research

I am interested in this research as it embodies everything I have aimed to achieve in my teaching career. Having observed general computing misconceptions of KS2 pupils who join my school, in 2013, I sought to bridge the gap between the delivery of computer science in primary schools and secondary. I decided to visit five local primary schools in my free periods to run programming sessions with their staff and pupils for an hour every week – free of charge. Nine years later, over 4,700 students and 351 staff in my local community and across the country have benefited directly from my sessions. Due to my efforts, my school was awarded a “Lead School” status in the delivery of computer science by Computing at School.

I lead a very successful department and I am always seeking ways to improve upon my lesson delivery. My headteacher and other senior colleagues have consistently ranked my teaching as outstanding. Parents have written to my headteacher to appreciate the length I go to support pupils.

As a member of Digital School House and National Centre for Computing Education – institutions with established contact with computing Lead Schools, I meet regularly with colleagues in these groups to discuss teaching pedagogies, research and best practices.

The findings from my research will have a greater impact on how computer science teachers deliver their lessons and how pupils learn. It will be another way I can give back to the community.

Proposed Timeline for completing the Thesis

It is my intention to complete the thesis within four or five years – see figure 1 below.

	2022			2023			2024			2025			2026			2027		
	Mar - Apr	May - Aug	Sep - Dec	Jan - Apr	May - Aug	Sep - Dec	Jan - Apr	May - Aug	Sep - Dec	Jan - Apr	May - Aug	Sep - Dec	Jan - Apr	May - Aug	Sep - Dec	Jan - Apr	May - Aug	Sep - Dec
Research Proposal																		
Literature Review																		
Development of Tools																		
Data Collection																		
Analysis of Data																		
Thesis write-up																		
Submission of Thesis																		

Figure 1: Draft Research Timeline

References

- Brown, N. C. C., Sentance, S., Crick, T., and Humphreys, S. 2014. Restart: The resurgence of Computer Science in UK schools. *ACM Trans. Comput. Educ.* 14, 2, Article 9 (June 2014).
- UCAS, 2021 Cycle Applicant Figures – January Equal Consideration Deadline, <https://www.ucas.com/data-and-analysis/undergraduate-statistics-and-reports/ucas-undergraduate-releases/applicant-releases-2021/2021-cycle-applicant-figures-january-deadline>.
- D. Fisher & N. Frey, 2013, *Better Learning Through Structured Teaching: A Framework for the Gradual Release of Responsibility* (2nd ed.).
- Pearson, P.D., & Gallagher, M.C. (1983). The instruction of reading comprehension. *Contemporary Educational Psychology*, 8(3), 317 – 344.
- K. Maton. 2013. Making semantic waves: a key to cumulative knowledge-building. *Linguistics and Education*.
- Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (October 2017), 24 pages. <https://doi.org/10.1145/3077618>.
- Paul Curzon, Jane Waite, Karl Maton, and James Donohue. 2020. Using Semantic Waves to Analyse the Effectiveness of Unplugged Computing Activities. In *WiPSCE '20: The 15th Workshop in Primary and Secondary Computing Education*, October 28–30, 2010, Online. ACM, New York, NY, USA, 10 pages.
- Department of Education (Press Release), 2012. <https://www.gov.uk/government/news/harmful-ict-curriculum-set-to-be-dropped-to-make-way-for-rigorous-computer-science>.
- British Council, <https://www.teachingenglish.org.uk/article/introduction-using-visualisation>
- McDaniel, M. A., & Einstein, G. O. (1986). Bizarre imagery as an effective memory aid: The importance of distinctiveness. *Journal of Experimental Psychology: Learning, Memory, and Cognition*.
- R. Steinmayr, A. Weidinger, M. Schwinger & B. Spinath (2019). The Importance of Students' Motivation for Their Academic Achievement – Replicating and Extending Previous Findings.

Sound-based music style modelling, for a free improvisation musical agent

Andrea Bolzoni

School of Computing and Communications
The Open University
Milton Keynes
MK7 6AA, UK
andrea.bolzoni@open.ac.uk

Abstract

This paper presents the first stages of development of an improvising musical agent capable of interacting with a human musician in a free improvised music context. This research aims to explore the creative potential of a co-creative system that draws upon two approaches of music and sound generation: the well-established practice of modelling musical styles with Markov-based models and recent developments in neural-network-based audio synthesis. At this preliminary stage, the focus is on the definition of style in a sound-based music context, and the outline of a formal evaluation framework for style imitation systems.

1. Introduction

The field of sound-based music has been one of the most hindered in Computational Creativity. The reason why lies in the wide range of unusual sounds employed, and the challenges in classifying them and modeling musical structures based on them. In particular, the missing link is in the development of a formal evaluation framework for sound-based music style modelling. In fact, even though some sound-based music style modelling systems have been developed (Tatar & Pasquier, 2017; Bernardes, Guedes, & Pennycook, 2013), it is unlikely to find a study that objectively evaluates their capabilities to do so.

Sound-based music style modelling can draw on the recent developments in the field of neural-based sound synthesis. In fact, neural-based sound synthesis would allow the expansion of the sound palette available. This will save time in building a big audio database to retrieve the sound samples from, as well as space to store it. In addition, I claim, it will facilitate the emergence of novel pieces through the exploration of a given style.

At this stage, I have implemented and tested already existent approaches to automatic sound-based composition. The aim was to gain experience for developing a musical agent that will be able to compose in real-time – improvising – along with a human musician.

2. Background

In the following subsections I will outline the context for this research. It embraces the fields of sound-based music, concatenative sound synthesis, statistical style modelling, and evaluation.

2.1. Sound-based music

Landy (2007) defines sound-based music as “the art form in which the sound, that is, not the musical note, is its basic unit”. In sound-based music, the individual entities that constitute a piece of music are commonly referred to as sound objects. Ricard and Herrera (2004) define a sound object as “any [sonic] entity perceived as having its own internal properties and rules”. Roads (2002) defines a sound object as “a basic unit of musical structure, generalizing the traditional concept of note”.

2.2. Concatenative Sound Synthesis

Concatenative sound synthesis draws on two other synthesis techniques: granular sound synthesis, where sound synthesis is performed through the generation of very short synthetic sonic grains (Roads, 1988); and granulation, where an audio corpus is segmented in tiny grains that are reassembled through time-domain-based operations (Roads, 2002). In concatenative sound synthesis the audio corpus is segmented into units (Schwarz, 2006). Each unit is a short sound segment of variable length. Sonic features - such

as pitch, duration, or audio descriptors - are extracted from the units. The resynthesis is performed through an algorithm that looks into the audio corpus for the closest units, most of the time in terms of Euclidean distance, in relation to a feature target.

Thanks to its potential, this synthesis technique has been applied in various forms and to various systems. CataRT (Schwarz, Beller, Verbrugghe, & Britton, 2006) allows the exploration of a sound corpus through a user interface where the segmented corpus is shown in a 2D space. MATConcat (Sturm, 2006) offers an implementation of adaptative concatenative sound synthesis where the feature target is controlled by the audio descriptors extracted from an audio file. Similarly, AudioGuide (Hackbarth, Schnell, Esling, & Schwarz, 2013) aims to extract morphologies from an audio file to generate new sonic material through concatenative sound synthesis. A different approach is offered in earGram (Bernardes et al., 2013), where temporal modelling is used to generate new sonic outcomes with a similar style to the audio corpus, for example in relation to the harmonic or timbral content.

More recently, thanks to neural network based generative techniques, a different approach has been proposed. Training a variational autoencoder, it is possible to learn a probabilistic distribution of the units, called a latent space. This is a continuous invertible space. Therefore, it is possible to synthesise units that match the feature target even if they were not in the audio corpus (Bitton, Esling, & Harada, 2020).

2.3. Statistical Style Modelling

Music can be seen as “organised sound” (Varèse & Wen-chung, 1966). Therefore, in principle, a music corpus treated as a sequence of organised events can be represented through a model. There are two main approaches to define such models: explicitly code the stylistic rules or infer the rules through statistical analyses (Conklin & Witten, 1995). Belonging to the latter approach, Markov-based models are widespread in music style modelling for their ability to model music patterns (Pachet, 2003).

If we look at the two main shortcomings of Markov Models, we will see that: 1) if the order is low they can model patterns, but they can't properly model the structure of a piece (Pachet, 2003); 2) if the order is high they generally overfit the piece (Papadopoulos, Roy, & Pachet, 2014). Indeed, Pachet (2003) states that an interactive system could benefit from the ability of Markov models to learn patterns, while the definition of the structure can be left to the human musician interacting with the system. In this way, there are no drawbacks from their inability to learn long-term structures. Another way to compensate the lack of ability to model long-term structures is proposed in Improtek (Nika & Chemillier, 2012), a developed version of Omax (Assayag, Bloch, & Chemillier, 2006). Here, the interaction between the human musician and the system is based on a predefined dynamic score.

2.4. Evaluation

In the context of Musical Computational Creativity, evaluation is at the same time one of the most important aspects and one of the most overlooked. Evaluation is necessary to show the progress and contributions to the field (Jordanous, 2012), but only a small number of the systems presented in conferences offer a formal evaluation. Tatar and Pasquier (2019) provide a clear example in their typology and exposition of the state of the art of musical agents. Here, they show that, out of 78 presented systems, only 17 had undergone an evaluation process. An approach to evaluating free improvisation is proposed by Linson, Dobbyn, Lewis, and Laney (2015).

This lack of evaluation could be due to a highly fragmented field, where many systems have been developed for specific creativity needs of their developers. Hence, the difficulty of objectively evaluate them (Gifford et al., 2018). Nevertheless, trying to reduce this fragmentation might not be a solution: even though it might give more opportunities to develop more solid evaluation frameworks, the specificity of the tasks carried out by the systems can increase their success (Truax (1980) cited in Pasquier, Eigenfeldt, Bown, and Dubnov (2016)). As a consequence of this fragmentation, the tasks that the systems are asked to carry out have not a “yes or no answer”; and, the evaluation of their outcomes very often depends on the subjective preferences of the users or the audience (Pasquier et al., 2016).

Looking at the bigger picture, the lack of consensus on what creativity is - human and artificial - makes the evaluation of artificial agents' creativity a non-trivial task (Jordanous, 2012). Although a number of evaluation frameworks have been proposed in the last few years, the differences among systems might result in the necessity of tailoring evaluation strategies "to specific research goals in ways that are relevant and have integrity" (Pasquier et al., 2016).

3. Research perspectives

In Computational Creativity, statistical modelling has been widely implemented to model and generate note-based music (Assayag et al., 2006; Conklin, 2003; Pachet, 2003). Sound-based music has been studied to some extent, primarily employing concatenative sound synthesis along with Markov-based style modelling (Tatar & Pasquier, 2017; Bernardes et al., 2013).

Even though note-based musical agents can only generate notes, those notes can be played and interpreted in a variety of ways through synthesised sounds or human musicians. Sound-based musical agents rely on an audio corpus. Therefore, their output is limited to the sonic material present in the audio corpus. This material can be expanded through sound processing techniques, but the audio quality could, nevertheless, easily degrade (Schwarz, 2006).

The development of neural-based synthesis techniques in the last few years opened new possibilities for sound-based musical agents. These techniques can model an audio corpus as an invertible space. Therefore, they give the opportunity to synthesise sounds that were not present in the audio corpus (Bitton et al., 2020).

We will provide a system that will draw on mature work from the field of statistical modelling merged with the expressivity of neural-based sound synthesis. The aim is to contribute to the study of musical computational creativity through a system capable of provoking novel interactions in a free improvised context.

4. Discussion

Style imitation is defined by Pasquier et al. (2016) as: "Given a corpus $C = C_1, \dots, C_n$ representative of style S ", style imitation is the generation of "new instances that would be classified as belonging to S by an unbiased observer (typically a set of human subjects)". As a general definition, style means "a particular manner or technique by which something is done, created, or performed"¹. Among the meanings of music style offered by Dannenberg (2010), we find that use of musical texture could be an aspect of a given style. Nevertheless, musical texture is a difficult component to define. From a sound-based musical point of view, it can certainly be related to the spectromorphological approach proposed by Smalley (1997). From a computational perspective, the spectromorphological approach has been implemented to model an audio corpus - and its style (Tatar & Pasquier, 2017; Bernardes et al., 2013). But, from an evaluation perspective it is still an open question how to define style in sound-based music - and, therefore, how to define the parameters to be evaluated.

In note-based style modelling the use of MIDI notes let us use a symbolic representation of music that can be resynthesised for the purpose of evaluation. As an example, Collins, Laney, Willis, and Garthwaite (2016) use a synthesised piano to reproduce MIDI files in order to evaluate the stylistic success of computationally generated mazurkas. The basic component used to model the style - the note - is detached from the sonic rendering of the music.

In sound-based music, to some extent the sounds used in the audio corpus define themselves the style. And, as exposed in the Background section, sound-based musical agents usually generate their outputs retrieving sounds from the same audio corpus they modelled. Therefore, another open question is to what extent the stylistic success of the music generated by such models is based on the modelling technique implemented, or on the sounds that constitute the audio corpus.

¹<https://www.merriam-webster.com/dictionary/style>, accessed on 7 April 2022.

5. Acknowledgments

Thanks to my supervisors Dr Robin Laney, Dr Alistair Willis, Prof David Sharp, and Dr Adam Linson for their suggestions and support. This research is funded by The Open University.

6. References

- Assayag, G., Bloch, G., & Chemillier, M. (2006). OMax-Ofon. In *Sound and music computing (smc)*. Marseille, France.
- Bernardes, G., Guedes, C., & Pennycook, B. (2013). EarGram: An Application for Interactive Exploration of Concatenative Sound Synthesis in Pure Data. In *International symposium on computer music modeling and retrieval* (pp. 110–129). Retrieved from http://link.springer.com/10.1007/978-3-642-41248-6_7 doi: 10.1007/978-3-642-41248-6_7
- Bitton, A., Esling, P., & Harada, T. (2020). Neural Granular Sound Synthesis. *CoRR, abs/2008.0*, arXiv preprint arXiv:2008.01393. Retrieved from <https://arxiv.org/abs/2008.01393> doi: <https://doi.org/10.48550/arXiv.2008.01393>
- Collins, T., Laney, R., Willis, A., & Garthwaite, P. H. (2016, feb). Developing and evaluating computational models of musical style. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 30(1), 16–43. Retrieved from <https://www.cambridge.org/core> https://www.cambridge.org/core/product/identifier/S0890060414000687/type/journal_article doi: 10.1017/S0890060414000687
- Conklin, D. (2003). Music Generation from Statistical Models. In *Proceedings of the aisb 2003 symposium on artificial intelligence and creativity in the arts and sciences* (pp. 30–35).
- Conklin, D., & Witten, I. H. (1995, mar). Multiple viewpoint systems for music prediction. *Journal of New Music Research*, 24(1), 51–73. Retrieved from <http://www.tandfonline.com/doi/abs/10.1080/09298219508570672> doi: 10.1080/09298219508570672
- Dannenberg, R. B. (2010). Style in Music. In *The structure of style* (pp. 45–57). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://link.springer.com/10.1007/978-3-642-12337-5_3 doi: 10.1007/978-3-642-12337-5_3
- Gifford, T., Knotts, S., McCormack, J., Kalonaris, S., Yee-King, M., & D’Inverno, M. (2018, jan). Computational systems for music improvisation. *Digital Creativity*, 29(1), 19–36. Retrieved from <https://www.tandfonline.com/doi/full/10.1080/14626268.2018.1426613> doi: 10.1080/14626268.2018.1426613
- Hackbarth, B., Schnell, N., Esling, P., & Schwarz, D. (2013, feb). Composing Morphology: Concatenative Synthesis as an Intuitive Medium for Prescribing Sound in Time. *Contemporary Music Review*, 32(1), 49–59. Retrieved from <http://www.tandfonline.com/doi/abs/10.1080/07494467.2013.774513> doi: 10.1080/07494467.2013.774513
- Jordanous, A. (2012, sep). A Standardised Procedure for Evaluating Creative Systems: Computational Creativity Evaluation Based on What it is to be Creative. *Cognitive Computation*, 4(3), 246–279. Retrieved from <http://link.springer.com/10.1007/s12559-012-9156-1> doi: 10.1007/s12559-012-9156-1
- Landy, L. (2007). Introduction. In *Understanding the art of sound organization* (pp. 1–20). MIT Press.
- Linson, A., Dobbyn, C., Lewis, G. E., & Laney, R. (2015, dec). A Subsumption Agent for Collaborative Free Improvisation. *Computer Music Journal*, 39(4), 96–115. Retrieved from <http://dx.doi.org/doi:10.1162/COMJa00323> <https://direct.mit.edu/comj/article/39/4/96/106783/A-Subsumption-Agent-for-Collaborative-Free> doi: 10.1162/COMJ_a_00323
- Nika, J., & Chemillier, M. (2012). Improtek: integrating harmonic controls into improvisation in the filiation of OMax. In *International computer music conference (icmc)* (pp. 180–187). Ljubljana, Slovenia. Retrieved from <https://hal.archives-ouvertes.fr/hal-01059330>
- Pachet, F. (2003, sep). The Continuator: Musical Interaction With Style. *Journal of New Music Research*, 32(3), 333–341. Retrieved from <http://www.tandfonline.com/doi/abs/>

- 10.1076/jnmr.32.3.333.16861 doi: 10.1076/jnmr.32.3.333.16861
- Papadopoulos, A., Roy, P., & Pachet, F. (2014). Avoiding plagiarism in Markov sequence generation. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 28, pp. 2731–2737).
- Pasquier, P., Eigenfeldt, A., Bown, O., & Dubnov, S. (2016, dec). An Introduction to Musical Metacreation. *Computers in Entertainment*, 14(2), 1–14. Retrieved from <https://dl.acm.org/doi/10.1145/2930672> doi: 10.1145/2930672
- Ricard, J., & Herrera, P. (2004). Morphological sound description: computational model and usability evaluation. In *Audio engineering society 116th convention*.
- Roads, C. (1988). Introduction To Granular Synthesis. *Computer Music Journal*, 12(2), 11–13.
- Roads, C. (2002). *Microsound*. MIT Press.
- Schwarz, D. (2006). Concatenative sound synthesis: The early years. *Journal of New Music Research*, 35(1), 3–22.
- Schwarz, D., Beller, G., Verbrugghe, B., & Britton, S. (2006). Real-Time Corpus-Based Concatenative Synthesis with CataRT. In *9th international conference on digital audio effects (dafx)* (pp. 279–282).
- Smalley, D. (1997, aug). Spectromorphology: explaining sound-shapes. *Organised Sound*, 2(2), S1355771897009059. Retrieved from http://www.journals.cambridge.org/abstract_S1355771897009059 doi: 10.1017/S1355771897009059
- Sturm, B. L. (2006). Adaptive Concatenative Sound Synthesis and Its Application to Micromontage Composition. *Computer Music Journal*, 30(4), 46–66.
- Tatar, K., & Pasquier, P. (2017). MASOM: A Musical Agent Architecture based on Self-Organizing Maps, Affective Computing, and Variable Markov Models. In *Proceedings of the 5th international workshop on musical metacreation (mume 2017)*. Atlanta, Georgia, USA. Retrieved from <https://musicalmetacreation.org/mume2017/proceedings/Tatar.pdf>
- Tatar, K., & Pasquier, P. (2019, jan). Musical agents: A typology and state of the art towards Musical Metacreation. *Journal of New Music Research*, 48(1), 56–105. Retrieved from <https://www.tandfonline.com/doi/full/10.1080/09298215.2018.1511736> doi: 10.1080/09298215.2018.1511736
- Varèse, E., & Wen-chung, C. (1966). The Liberation of Sound. *Perspective of New Music*, 5(1), 11–19. Retrieved from <https://www.jstor.org/stable/832385>