Psychology of Programming Interest Group Annual Conference 2015

Bournemouth University
Hotel Miramar, East Overcliff Drive, Bournemouth
15th – 17th July 2015

Proceedings

Edited by:
Melanie Coles and Gail Ollis
MESSAGE FROM THE CHAIRS

Welcome to the Psychology of Programming Interest Group (PPIG) Conference 2015. This year it is returning to Bournemouth (PPIG's 13th Annual Workshop was held here in April 2001), and it is taking place at the Hotel Miramar, up on Bournemouth's East Cliff overlooking the sea.

The Psychology of Programming Interest Group (PPIG) was established in 1987 in order to bring together people from diverse communities in universities and industry to explore common interests in the psychological aspects of programming and in the computational aspects of psychology. The group, which at present numbers approximately 300 world-wide, includes cognitive scientists, psychologists, computer scientists, software engineers, software developers, HCI people et al.

This year our focus is the growing importance placed on programming as an essential skill. As reflected in the open availability of online courses, programming is increasingly engaging with a wider audience. Online materials engage with children (Scratch, Tyner.com, Code.org), through to university students, independent adult learners, second and subsequent language learners (CodeAcademy, W3Schools, Lynda.com; MOOCs, e.g. Coursera, edX). There is also a wealth of resources to support the online collaborative programmer (online forums, Stack Overflow, wikis). How does (or should) what we know about the approaches, learning, tools, technologies and the interaction of programmers impact upon the resources for programming online?

The invited speakers reflect this theme with Simon Peyton Jones' talk on The dream of a lifetime: an opportunity to shape how our children learn computing and Russel Winder's exploring Tales from the Workshops: A sequence of vignettes, episodes, observations, and reflections on many years of trying to teach people programming. The theme acknowledges the increased importance of software use in all our lives so we also have two invited talks from Raian Ali, from Bournemouth University, exploring addiction-aware software and software-based motivation.

The conference continues a tradition of hosting a Doctoral Consortium specifically to enable research students in the relevant disciplines to come together, give presentations and exchange ideas. We thank Thomas Green for chairing the consortium and Keith Phalp, from Bournemouth University, for being on the consortium panel.

We are also running some lightning talks this year, including a number from Bournemouth University on intriguing topics: Guinness World record App-a-thon; Audio and Video Feedback; Many Digital Devices – Harnessing Opportunities; The Eyes Have It; Programming of Psychology.

We are grateful to all who made this conference possible: the Programme Committee, the Doctoral Committee, all three of our invited speakers, our lightning talkers (those booked and those yet to volunteer) and all presenters. We would also like to thank our helpers for making the event run smoothly.

Melanie Coles and Gail Ollis
ORGANISATION

Programme Chairs and Local Organisers
Melanie Coles and Gail Ollis
Department of Computing and Informatics
Faculty of Science and Technology, Bournemouth University, UK

Doctoral Consortium
Thomas Green, PPIG, UK (Chair)
Keith Phalp, Bournemouth University
Other panel members T.B.C.

Organising Committee
Melanie Coles, Thomas Green, Maria Kutar and Gail Ollis

Programme Committee
Roman Bednarik, University of Eastern Finland, Finland
Alan Blackwell, University of Cambridge, UK
Richard Bornat, Middlesex University, UK
Jim Buckley, University of Limerick, Ireland
Sylvia Da Rosa, Universidad de la República, Uruguay
Christopher Douce, Open University, UK
Chris Exton, University of Limerick, Ireland
Judith Good, University of Sussex, UK
Thomas Green, PPIG, UK
Yanguo Jing, London Metropolitan University, UK
Babak Khazaei, Sheffield Hallam University, UK
Charles Knutson, Brigham Young University, USA
Maria Kutar, University of Salford, UK
Clayton Lewis, University of Colorado, USA
Louis Major, University of Cambridge, UK
Lindsay Marshall, Newcastle University, UK
Alex Mclean, University of Leeds, UK
Marian Petre, Open University, UK
Jorma Sajaniemi, University of Eastern Finland, Finland
Markku Tukiainen, University of Eastern Finland, Finland
Rebecca Yates, University of Limerick, Ireland
INVITED SPEAKERS

Simon Peyton Jones, Microsoft Research and Chair of Computing at School, UK
Russel Winder, Independent consultant, UK
Raian Ali, Bournemouth University, UK
## INDEX BY SESSION

### Session 1

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marina Isaac, Eckhard Pfügel, Gordon Hunter and James Denholm-Price</td>
<td>Intuitive NUIs for Speech Editing of Structured Content (Work in Progress)</td>
<td>1</td>
</tr>
<tr>
<td>Shahin Rostami, Alex Sheneld, Stephen Sigurnjak and Oluwatoyin Fakorede</td>
<td>Evaluation of Mental Workload and Familiarity in Human Computer Interaction with Integrated Development Environments using Single-Channel EEG</td>
<td>7</td>
</tr>
</tbody>
</table>

### Session 2

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advait Sarkar</td>
<td>Confidence, command, complexity: metamodels for structured interaction with machine intelligence</td>
<td>23</td>
</tr>
</tbody>
</table>

### Session 3

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sylvia da Rosa</td>
<td>The construction of knowledge of basic algorithms and data structures by novice learners</td>
<td>37</td>
</tr>
</tbody>
</table>

### Session 4

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advait Sarkar</td>
<td>The impact of syntax colouring on program comprehension</td>
<td>49</td>
</tr>
<tr>
<td>Giovanna Maria Dimitri</td>
<td>The impact of Syntax Highlighting in Sonic Pi</td>
<td>59</td>
</tr>
</tbody>
</table>

### Session 5

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mariana Mărășoiu, Luke Church and Alan Blackwell</td>
<td>An empirical investigation of code completion usage by professional software developers</td>
<td>71</td>
</tr>
<tr>
<td>Antranig Basman, Colin Clark and Clayton Lewis</td>
<td>Harmonious Authorship from Different Representations (Work in Progress)</td>
<td>83</td>
</tr>
</tbody>
</table>
## Doctoral Consortium

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thomas Green</td>
<td>Foreword</td>
<td>89</td>
</tr>
<tr>
<td>Martijn Stegeman</td>
<td>Understanding code quality for introductory courses</td>
<td>91</td>
</tr>
<tr>
<td>Ermira Daka</td>
<td>Improving Readability of Automatically Generated Unit Tests</td>
<td>97</td>
</tr>
<tr>
<td>Simon Butler</td>
<td>Analysing Java Identifier Names in the Wild</td>
<td>103</td>
</tr>
</tbody>
</table>
Intuitive NUIs for Speech Editing of Structured Content  
(Work in Progress)  

Marina Isaac¹, Eckhard Pfügel², Gordon Hunter³, and James Denholm-Price⁴

¹ Faculty of Science, Engineering and Computing, Kingston University  
  M.Isaac@kingston.ac.uk
² Faculty of Science, Engineering and Computing, Kingston University  
  E.Pfluegel@kingston.ac.uk
³ Faculty of Science, Engineering and Computing, Kingston University  
  G.Hunter@kingston.ac.uk
⁴ Faculty of Science, Engineering and Computing, Kingston University  
  J.Denholm-Price@kingston.ac.uk

Abstract. Improvements in automatic speech recognition, along with the growing popularity of speech driven “assistants” in consumer electronics, indicate that this input modality will become increasingly relevant. Although good functionality is offered for word processing applications, this is not the case for highly structured content such as mathematical text or computer program code. In this paper we combine the principles of natural user interfaces with the concept of intuitive use, and adapt them for speech as the input modality in the context of editing content displayed on a screen. The resulting principles are used to inform design of the user interface of a specialist language editor for spoken mathematics.

Keywords: natural user interface, NUI, spoken mathematics, speech control

1 Introduction

Using speech as an input modality has been available since the 1980s but is not yet a mainstream form of human-computer interaction. Due to recent improvements in the capabilities of automatic speech recognition (ASR), and introduction of speech driven “assistants” in consumer electronics, this type of interface is likely to become increasingly relevant.

ASR products such as Nuance’s Dragon¹ have existed for many years, and provide great functionality in the context of word processing, allowing the user to dictate content in a variety of widely spoken languages. Such developments have not been applied to specialised languages used to describe structured content such as mathematical text or computer program code. Because of their formatting and punctuation, these are not served well by standard document editing facilities, and so even experienced ASR users have great trouble working with such content.

The problem of enabling casual users (that is, those not expert in LaTeX or similar languages) to create properly formatted mathematics is well known, and we hope that spoken mathematics² will help in this area. The difficulty with the spoken approach is that the use of standard ASR software imposes a high enough cognitive load on the user (through having to recall the command language) to present its own challenge.

The concept of the natural user interface (NUI) has so far been applied mainly to input via touch and gestures (Wigdor & Wixon, 2011). In this paper we consolidate the general NUI guidelines with those pertaining to “intuitive use” (Naumann et al., 2007) of interfaces, to provide a list of intuitive NUI principles.

After considering what might be regarded as “feeling natural” in a speech-driven environment, our contribution is to adapt these principles specifically to a speech controlled environment, and consider how they may be used to enhance the speech based mathematical expression editor TalkMaths (Wigmore, Hunter, Pfügel, & Denholm-Price, 2009).

2 Natural User Interfaces

The concept of the natural user interface was first developed by Fjeld, Bichsel, and Rauterberg (1998, 1999) to describe a user environment with minimal discontinuity between the physical actions required to complete a task

¹ http://www.nuance.com/dragon/index.htm
² In this context, the term “spoken mathematics” refers to that which would be dictated by a native English speaker.
and the user’s internal problem solving process. The idea is based on the activity cycle\(^3\) in action regulation theory (Hacker (1994), as cited by Fjeld et al. (1999)), and the fact that for optimal task performance, users need to be able to perform epistemic as well as pragmatic actions (Fjeld et al., 1998)\(^4\). Epistemic actions are treated as fundamental by Fjeld et al. (1998), who put the ability to engage in these at the top of their original list of NUI design guidelines. To give users the confidence to behave in this exploratory way, the negative effect of making any mistakes needs to be minimised – the second guideline. The third guideline is to permit the user to employ as much of their body as possible as well as their voice in interactions (Fjeld et al., 1998). This last guideline requires system observation of the complete user environment, including interaction with artefacts such as visual projections (Rauterberg, 1999) and, as a somewhat ambitious requirement, this remains purely the subject of experimental UIs, no longer appearing in the list of design principles later presented by Fjeld et al. (1999).

Developments in touch screen and gesture technology have motivated researchers to investigate the resulting opportunities in user interface (UI) design (Wigdor, Fletcher, & Morrison, 2009). Wigdor and Wixon (2011) present a practical guide for designers, including ways in which many of the ideas of Fjeld et al. (1999) may be implemented.

Jetter, Reiterer, and Geyer (2014) also address the area of NUIs in their Blended Interaction framework. This uses conceptual blending as defined by Fauconnier and Turner (2008) along with the image schemas\(^5\) described by Hurtienne and Israel (2007) in an effort to predict which metaphors will be easy for users to understand (Jetter et al., 2014). It is worth noting that the blends suggested by Jetter et al. (2014) incorporate the metaphors involved in image schemas, which themselves reflect the language used to describe relations and actions (Hurtienne & Israel, 2007). Although both areas of work relate to direct manipulation (typically using hands), the fact that language lies at the core of the ideas may prove useful to designers of speech UIs.

The original descriptions of NUIs refer to the property of an interface being “intuitive” (Fjeld et al., 1998, 1999). Blackler and Hurtienne (2007) describe intuitive use as taking advantage of the user’s existing knowledge of comparable situations, to make aspects of an interface seem familiar to them; Naumann et al. (2007) add the condition that the user should almost be unconscious of the fact they are operating a UI. One of the benefits of this lack of awareness by users will be a lower cognitive load associated with using the interface (Naumann et al., 2007), specifically with carrying out the low level actions required to achieve a goal in the activity cycle.

The following general principles combine the ideas of Fjeld et al. (1999) with the other work on NUIs and the above definitions of intuitive use. They are grouped according to general objective of NUIs.

**Encourage epistemic actions and exploratory behaviour.** As proposed by Fjeld et al. (1998), doing so will help the user complete their task efficiently, exhibiting the exploratory behaviour that will help them to progress to expertise in the application (Wigdor & Wixon, 2011, p.55).

1. Users with proficiency levels ranging from novice to expert should feel comfortable using the software (Wigdor & Wixon, 2011, p. 13).
2. The interface should provide alternative ways of invoking functionality for different classes of user, as well as employing other types of redundancy such as using both text and icons to describe controls (Blackler & Hurtienne, 2007).
3. Interaction with the system should feel robust to the user, so that they have the confidence to attempt new operations.
   - For major changes or destructive actions, the system should require confirmation from the user and provide previews where appropriate (Wigdor & Wixon, 2011, p. 55).
   - The system should minimise the impact of user errors (Fjeld et al., 1998) by allowing the user to reverse them easily (Fjeld et al., 1999).

**The user should feel that their interaction with the system is intuitive.** There is considerable overlap between the concepts of intuitive use and naturalness. The following principles pertain to intuitive use.

4. Where conventions have been established in the application area or medium (Wigdor & Wixon, 2011, p. 13), adhere to these, otherwise use appropriate metaphorical devices to represent objects or actions (Blackler & Hurtienne, 2007).
5. Take advantage (where appropriate) of users’ existing skills, to make their experience feel more familiar (Wigdor & Wixon, 2011, p. 13).
6. Facilitate the planning aspect of the activity cycle by indicating the current state of the software and the available actions at all times (Wigdor & Wixon, 2011, p. 45) (Fjeld et al., 1999).
7. Show the results of all user actions (Fjeld et al., 1999), with feedback being immediate, appropriate (Blackler & Hurtienne, 2007) and informative. Non-trivial feedback (for example, system messages) should increase user understanding of the system and provide appropriate help where needed (Wigdor & Wixon, 2011, p. 56).

\(^3\) The repeated steps of goal setting, action planning, performance and evaluation, taken by an individual to accomplish a task.

\(^4\) **Pragmatic** actions are those that bring a task physically closer to completion, while **epistemic** actions are performed primarily to aid mental processing (Kirsh & Maglio, 1994).

\(^5\) Abstractions that reflect the way humans relate objects in the real world.
8. Clear affordance\(^6\) in the design of controls will help the user identify both their functions and modes of use (Fjeld et al., 1999; Blackler & Hurtienne, 2007) (Wigdor & Wixon, 2011, p. 55).
9. The interface should be compatible with the user’s mental model of the system (Blackler & Hurtienne, 2007).

**Context of use should be taken into account.** The design should reflect:
10. the nature of the user’s task rather than the technology of the application (Blackler & Hurtienne, 2007), and also
11. the physical environment and social context in which the system is used (Wigdor & Wixon, 2011, p. 19).

### 3 What Feels “Natural” in a Speech Interface?

While natural language may seem to be the ideal choice for casual use of or novice support for an interface, not only is the current technology too immature for serious application, but speaking “wordy” sentences to describe repetitive actions is not desirable for most users. Research shows that brief commands are preferred to natural language sentences for such tasks (Elepfandt & Grund, 2012; Stedmon, Patel, Sharples, & Wilson, 2011), suggesting a language that is superficially simple but still allows for more complex commands may go a long way towards providing a natural feeling experience.

The approach of Wigdor and Wixon (2011) reflects a natural progression from manipulating objects on screen using keyboard and mouse (or other pointing device) to a subjective feeling of direct manipulation using the same extremities. Although it may be tempting to develop a “typing assistant”, we believe this would be unsatisfactory because such a proxy could not give the user the feeling of manipulating the content themselves. If instead we ask how objects on screen may be manipulated using voice commands, the user should be less aware that they are having to go through an intermediary.

A major challenge for speech control is how to indicate which objects are to be manipulated, and how to transform them. One promising method for object selection is to use eye gaze as an adjunct to speech (Elepfandt & Grund, 2012). The findings of Kaur et al. (2003) and Maglio, Matlock, Campbell, Zhai, and Smith (2000) – that a user’s gaze is naturally directed towards the object they wish to manipulate just before they issue a command – suggest that the use of this modality may contribute to the overall efficiency of such an interface. As Sibert and Jacob (2000) acknowledge, when gaze is used in this way (rather than as the main method of interaction), the interface is in fact making use of natural human behaviour and so partially meets the third original guideline for NUIs (Fjeld et al., 1998). In addition to its “naturalness”, eye gaze has been shown to enable objects to be selected more quickly than by mouse (Sibert & Jacob, 2000), so might be expected to become a popular means of interaction when the technology matures. Until then, other means are needed to refer to objects on screen, for example the type of grids described by Wigmore (2011). Nuance’s mouse grids, and the context-sensitive mouse grids described by Begel (2005). Rather than use numbers to index all non-word content, we propose meaningful labels, where possible, on the grid, to make selection easier. This may also help in frequently repeated sequences of actions, where using a label name will be easier than locating a label number, and may be particularly pertinent for “semantic grids” (Wigmore, 2011), where the labels would reduce the user’s cognitive load by eliminating the need to recall terms such as “numerator”. Use of such labels for semantic grids may also facilitate learning of mathematical concepts, an area investigated by Attanayake, Hunter, Denholm-Price, and Pfuegel (2013).

### 4 Redesigning *TalkMaths* to Reflect NUI Principles for Speech

The principles for intuitive NUIs are adapted for speech editing environments, and illustrated by application to *TalkMaths*\(^7\) (Wigmore, Hunter, Pfügel, & Denholm-Price, 2009). This is a web-based tool whose purpose is to allow users to enter and edit mathematical expressions using speech or keyboard input, and was created as the result of work initially carried out by Wigmore, Hunter, Pfügel, Denholm-Price, and Binelli (2009) when investigating the use of speech input for creating and editing structured documents. In this system, spoken commands are used to dictate mathematical expressions, including common mathematical symbols and operators, as well as to edit content. The numbering of the suggested modifications below follows that used in Section 2.

1. **Users with varying proficiency should feel comfortable using the software.** A mechanism should be in place to remind novice users of the basic commands, alongside a means of making them aware that more complex versions are available. This should help them in getting the feeling of gaining mastery of the command language (rather than continually having to resort to a help system), thus forming part of the “scaffolding” described by Wigdor and Wixon (2011, p.53). A command history screen area could show completed commands, which may also help novice users learn the language. Experienced users should be able to give more than one command in a single utterance. As with the command language, it may be desirable to hide the

---

\(^6\) See item 8 in Section 4.

\(^7\) We assume use of an ASR product such as those offered by Nuance or Microsoft, for recognition of English speech, and that the user is not severely visually impaired.
full size and complexity of the content language from novice users. For example, casual users will not want to be overwhelmed with the large number of mathematical symbols and operators expected by researchers. This requirement could be addressed either by showing only the most popular words by default, or by using a visual device to make the popular ones more prominent.

2. **Allow functionality to be invoked in different ways.** Because our discussion concentrates on a single modality, there is some overlap between this guideline and the previous one. While experts may give one or more entire commands in one utterance, novice users may need to build them up interactively. It should be possible to customise commands, perhaps changing specific words to ones less likely to be misrecognised given the usage environment, or create commands that replace a frequently used phrase with a single word (Fateman, 2013). Because a speech interface has words at its heart, rather than provide icons with text that is hidden by default, the reverse approach may be taken, using icons only where appropriate to guide the user’s eye to the text reminder. Where command reminders are used, explanatory words may be included in these, that do not need to be spoken as part of the command, and which are ignored if included in an utterance.

3. **Users should not be afraid of making mistakes.** Bearing in mind users’ preference for briefer utterances, previews should be shown at the same time as requests for confirmation. As well as confirmations and previews, it should be possible to use the command history as a means of undoing changes made. Syntax errors in user commands should be handled in a way that minimises the amount of further user input required.

4. **Follow established conventions, and use appropriate metaphors.** The system should respond to popular conventions in ASR software, for example, “What can I say?” or, “Scratch that”. It may be possible to arrive at appropriate metaphors by considering the points made by Jetter et al. (2014) regarding conceptual blends (Fauconnier & Turner, 2008) and image schemas (Hurtienne & Israel, 2007).

5. **Allow users to exercise existing skills.** In addition to helping the user learn to use the software, this may boost their confidence by giving them a feeling of prior familiarity with the command language. Permitting vocabulary customisation will help by allowing the user to employ terms belonging to their discipline, or use “shorthand” from their working environment.

6. **Always indicate system state and available actions.** Areas on the screen may be used to indicate the system’s state, for example to distinguish between the task of providing missing information for a command issued but not executed, and that of editing a command recalled from history. Because all actions may be invoked using spoken commands, only those command reminders that are appropriate for the context should be displayed as being sayable (see principle 8).

7. **Give appropriate feedback for all user actions.** Because of the lack of haptic feedback provided with speech control, a mechanism should be used to indicate the fact that the user’s input has been detected (Wigdor & Wixon, 2011, p. 45), to allow for any delay in the processing of this input. (This is to avoid the spoken equivalent of the phenomenon of clicking a button on a web page multiple times – not through impatience, but because the user thinks a click event may not have been registered.) The delay may be particularly noticeable when using non-incremental speech recognition. In such cases it may be useful to show the progress of the input handling, perhaps to indicate completion of initial speech recognition, parsing, and processing of the command itself. This is in addition to the usual feedback one would expect.

8. **Clear affordances.** Here, there are two levels to the idea of affordance: (1) the traditional notion of an input control indicating its function, and (2) a control indicating how it may be used, for example by clicking or typing in text. The loss of “tooltip” text in touch-screen GUIs has caused a resurgence in controls whose functionalities are a mystery until activated. This is one case where the addition of text to an icon would not only inform users of its meaning, but provide a cue for what to say to activate it. Given that in WIMP interfaces, even text controls are selected by clicking, the equivalent of ‘clickable’ in the context of speech input would be ‘sayable’. Command buttons could be labelled with the text of the command (with perhaps an additional brief description), while other controls could be labelled with appropriate names, that may play a role similar to nouns in a command. This approach is already used by ASR software to enable the user to select, for example, fields within a form or follow a hyperlink, but a consistent style to indicate sayable may make it easier for the user to recognise sayable objects as such.

9. **Compatibility with the user’s mental model.** The user should be able to understand the structure of the objects they are manipulating (at the task level). For example, if a user is provided with a choice of alternative parses of a spoken instruction (rather than a list resulting from probabilistic predictions), this fact should be made clear.

10. **Reflect the nature of the task rather than the technology.** The software needs to be compatible with an environment where the user may want to combine various input modalities, and provide different input styles – one mode might allow the user to build up an expression as they think about a mathematical problem, while another would be optimised for fast input of hand-written notes.

---

8 The integration of two or more apparently unrelated or incompatible notions to form a new idea that draws parts of its meaning from both.

9 That which requires the entire utterance to be complete before it is interpreted.

10 Brief help text displayed when the pointer is moved over a control.

11 Windows, Icons, Menus and Pointers.
11. *Work within the environment of the user.* The software may be used in a noisy or public environment, in which case a user who is able to do so may want to use all modalities except speech input or output. Appearance and vocabulary of the interface may need to be adapted to the user’s social environment – for example, professional mathematicians may want a very different interface from students who occasionally need to include mathematical text in an electronic document.

4.1 Additional Requirements Specific to Speech-based Specialist Language Editors

There are a number of other issues that need to be considered when designing the interface of editors for content described by a specialist language, that we summarise very briefly here.

**Cursor replacement** An alternative means is needed to specify the insertion point for new content, or for selection of content for editing. The fact that we are working with what could be viewed as a “random access” modality gives us the opportunity to allow a richer specification for navigating through code, using its natural structure as well as exploiting eye gaze tracking technology when this becomes feasible.

**Handle incomplete commands** It would be useful to handle errors either in the commands or recognition of content by allowing incomplete content to be specified. This will also allow users to give just vague descriptions to some parts of a structure, as suits their way of working.

**Deal with ambiguous commands** The system should have a strategy in cases where alternative parses may be obtained for what the user has said.

**Concatenability** The interface should permit the user to include more than one command in a single utterance.

**Permit multiple utterance commands** The system should allow commands that are too long to be said in one breath to be broken into several utterances.

**Restriction of vocabulary** It should be possible to limit the vocabulary for the ASR to words that are appropriate within the context.

5 Conclusion and Further Work

We have compiled a list of general principles for natural user interfaces optimised for intuitive use, and adapted them for the modality of speech control in the context of developing an editor for content described by a formal language. These have enabled us to suggest a number of modifications to the interface of our example editor *TalkMaths.* This system uses an operator precedence grammar that includes mixfix\(^{12}\) operators, so that the command and content languages may be described in the same way (Attanayake, 2014). We hope that by describing their language using such a grammar, that many types of structured document could potentially be handled by future versions of the editor, including computer programs.

Our next step in the process is to implement the proposed design changes in the *TalkMaths* system and test it for usability by comparing user experiences of the original and new interfaces.

References


---

12 Operators involving more than one symbol or word, e.g. function notation.


Evaluation of Mental Workload and Familiarity in Human Computer Interaction with Integrated Development Environments using Single-Channel EEG

Shahin Rostami1, Alex Shenfield2, Stephen Sigurnjak3, and Oluwatoyin Fakorede4

1 Faculty of Science and Technology, Bournemouth University
s.rostami@bournemouth.ac.uk
2 Faculty of Arts, Computing, Engineering & Sciences, Sheffield Hallam University
a.shenfield@shu.ac.uk
3 School of Computing, Engineering & Physical Sciences, University of Central Lancashire
s.sigurnjak@uclan.ac.uk
4 Faculty of Science and Technology, Bournemouth University
o.fakorede@bournemouth.ac.uk

Abstract. With modern developments in sensing technology it has become possible to detect and classify brain activity into distinct states such as attention and relaxation using commercially available EEG devices. These devices provide a low-cost and minimally intrusive method to observe a subject’s cognitive load whilst interacting with a computer system, thus providing a basis for determining the overall effectiveness of the design of a computer interface. In this paper, a single-channel dry sensor EEG headset is used to record the mental effort and familiarity data of participants whilst they repeat a task eight times in either the Visual Studio or Eclipse Integrated Development Environments (IDEs). This data is used in conjunction with observed behaviour and perceived difficulties reported by the participants to suggest that human computer interaction with IDEs can be evaluated using mental effort and familiarity data retrieved by an affordable EEG headset.

Keywords: Electroencephalography (EEG), Human-Computer Interaction (HCI), Integrated Development Environment, Programming, Interface

1 Introduction

Advances and widespread adoption of technology in the workplace have resulted in more information being presented to people, this has ultimately resulted in a higher cognitive demand for the processing of the information presented to extract key data and information pertinent to the task being performed. This increased cognitive demand and large amount of information can cause confusion, inhibit comprehension, and lead to mistakes being made or have consequences to personal health such as mental stress. Although there are methods available to monitor performance such as completion times, error rates, and qualitative feedback form questionnaires etc., these only look at the observable metrics. Physiological measures can also be implemented to monitor mouse movement, eye motions and gaze, heart rates, and galvanic skin response, but again these neglect cognitive processing effort. Generally, the attempt to observe this cognitive effort has resulted in clever experimental design and questioning but does not directly relate to cognitive workloads or cognitive strategies used (Cutrell & Tan, 2008).

The interaction between humans and machines and the cognitive learning of this process has given rise to various methods of understanding the underlying mechanisms behind them. One such example would be to include GOMOS (Goals, operations, methods and selection rules) (Wolpert, Ghahramani, & Jordan, 1995). When humans are initially presented with a new tool to learn they generally do not have the skills to use it effectively. This is due to lack of knowledge of the operation and purpose of the tool, the situation(s) to use it, or the results obtained from the use of the tool itself. Studies have shown that the process of learning to use a new tool can be summarised as developing an internal model (Wolpert et al., 1995). This internal model provides a neural representation of how the body responds to a command at a
given velocity and position, and prior to the acquisition of this model, the user cannot use the tool skillfully as this internal model does not exist within the brain. However, after building this model the skill level increases (Kitamura, Yamaguchi, Hiroshi, Kishino, & Kawato, 2003). It is logical to propagate this internal model from the use of a new tool to using and interfacing with a computer program. This logical extension would also lead to performing familiar tasks using a differing software.

Most neuroscience research on interaction and learning has focussed on imaging the brain using MRI scans (Sakai et al., 1998; Imamizu et al., 2000; Kitamura et al., 2003), whilst this provides exact positioning of the areas of activation in the brain and their respective durations, it is also very expensive to conduct, as it requires specialist equipment and is difficult to achieve experimental results where the person is free to perform tasks as they would in the workplace or at home. This restricts the application of this technology to the analysis of specific tasks and not everyday activities. Electroencephalography (EEG) is another technology that allows the activity of the brain to be studied. EEG is less expensive, less restrictive, and widely used within research and clinical studies as it is a passive technology which is safe for extended use.

The brain is a complex organ which consists of a network of nerve cells connected via neurons, with each of these connections communicating messages via electrical signals. This electrical transmission can be detected and classified using a series of electrodes that are placed on the scalp with minimum intrusiveness. The very weak signals detected from the brain, usually within the $5 - 100 \, \mu V$ range (Lee & Tan, 2006) are amplified and compared to a reference voltage by the use of a differential amplifier, this then forms the basis for frequency analysis. The results of the EEG produce a frequency spectrum subdivided into delta ($1 - 4 Hz$), theta ($4 - 8 Hz$), alpha ($8 - 12 Hz$), beta ($15 - 30 Hz$), and gamma ($30 - > 40 Hz$), with each band providing levels of wakefulness or sleep and even “levels” of these states (Strijkstra, Beersma, Drayer, Halbesma, & Daan, 2003). In general terms, the lower frequencies (delta and theta) are not seen in the waking state and the majority of activity occurring when the subject is awake can be found within the alpha, beta, and gamma ranges (Miller, 2007). This makes it possible to determine the levels of wakefulness and the cognitive load (Antonenko, Paas, Grabner, & van Gog, 2010), which in conjunction with the ability to provide a minimally intrusive method of data collection makes the use of EEG a distinct appealing prospect for Human Computer Interaction (HCI) research.

The adoption of EEG has developed from expensive research and clinical equipment to a more consumer oriented technology. This reduction in the cost of EEG devices has lead to the use of low cost equipment to research HCI issues such as task classification (Lee & Tan, 2006), games (Nijholt, Tan, Allison, del R Milan, & Graimann, 2008), and in adaptive user interfaces (Cutrell & Tan, 2008). Specific examples of the use of low cost EEG devices can be found in Wang, Sourina, and Nguyen (2010), where EEG is used to create a game for medical applications, or (Chu & Wong, 2014) where a player’s attention is measured whilst playing a mobile based game. Another example of the use of low cost EEG devices can be found in Mak, Chan, and Wong (2013a) and Mak, Chan, and Wong (2013b), where EEG is used to measure the mental effort and familiarity of participants when tracing shapes using their non-dominant hand. One such low cost device is the Neurosky Mindwave headset. This is an ergonomic, minimally intrusive, lightweight, and single-channel dry sensor EEG device capable of distinguishing the delta, theta, alpha, beta, and gamma frequencies as well as “attention” and “relaxation” states with data communication via Bluetooth.

This paper proposes to use the Neurosky headset to gather EEG data from a sample of 12 subjects performing a set of tasks related to programming. The users will perform a series of simple tasks using differing Integrated Development Environments (IDEs) with the cognitive functions being recorded. The subject will then complete a questionnaire pertaining to the level of effort and difficulty to complete the tasks in conjunction with observations made by the
facilitator. This data is then correlated with the brain activity to determine the mental effort and the brain activity when performing the set tasks.

The rest of this paper is organised according to the following. Section 2 describes the method used for this study, including the configuration of the experiments, the methods used for data collection, and the techniques used for the processing of the data. Section 3 presents and discusses the results from the experiments. Section 4 concludes the paper with summary of the findings and suggestion for future research direction.

2 Methods

2.1 Experiments

A sample of twelve volunteers were recruited and considered in this study. The volunteers consisted of all males aged between 22 and 37 years old. All participants were reported to have normal or corrected-to-normal vision, and have no known history of neurological or psychological disorder. Participants have given their informed consent before taking part in the experimental procedure, which was approved by the ethics committee at the Bournemouth University in the United Kingdom. Participants were reported to have not previously used the Integrated Development Environments (IDEs) they were assigned for the experiment.

Each participant was asked to perform eight trials involving Human Computer Interaction (HCI) with one of two considered software IDEs. Each trial required the participant to follow clearly defined instructions to complete tasks in either of the considered IDEs. To complete the task successfully, participants were required to acquire a new set of interface associations, for example, creating a new project in either of the considered IDEs. The details of these trials have been presented in Figures 9 and 10 which illustrate the exercise sheets handed to the participants.

Each participant was asked to complete only one of the exercise sheets, resulting in six participants who completed the exercise sheet for the Visual Studio IDE, and six participants who completed the exercise sheet for the Eclipse IDE. There were no crossover of participants between IDEs.

The IDEs considered were Visual Studio by Microsoft, and Eclipse by the Eclipse Foundation, the specific details of these two IDEs are listed in Table 1.

Table 1. Configurations and specific information regarding the IDEs considered during the experiment.

<table>
<thead>
<tr>
<th></th>
<th>Visual Studio</th>
<th>Eclipse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Name</td>
<td>Visual Studio Community 2013</td>
<td>Eclipse IDE for Java Developers</td>
</tr>
<tr>
<td>Version</td>
<td>12.0.31101.00 Update 4</td>
<td>Luna Service Release (4.4.2)</td>
</tr>
<tr>
<td>Operating System</td>
<td>Microsoft Windows 7</td>
<td>Microsoft Windows 7</td>
</tr>
</tbody>
</table>

Two shell scripts were developed and executed after each task was completed by a participant in order to reset the configurations and environments of the IDEs before the next task.

Two computers were used during the experiment, each with the configuration listed in Table 2. Two computers were used to separate the EEG data collection software (which ran on Visual Studio) from the software used by participants, and to allow real-time analysis of the mental effort and familiarity data.

There were two facilitators; one was observing the performance of the participants and monitored the progress of the task while the other facilitator was monitoring the data acquisition of the user interface software and ensuring that the subjects headset maintains connection.
Table 2. Hardware and software configurations of the computers by participants to complete the tasks, and by the facilitators to collect the EEG data.

<table>
<thead>
<tr>
<th>Configuration name</th>
<th>Configuration value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Windows x64</td>
</tr>
<tr>
<td>RAM</td>
<td>16 GB</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel(R) Xeon(R) CPU E5-1620 @ 3.70GHz ×16</td>
</tr>
<tr>
<td>Total CPU Cores</td>
<td>8</td>
</tr>
<tr>
<td>MATLAB version</td>
<td>R2014b (8.4.0.150421) 64-bit (win64)</td>
</tr>
</tbody>
</table>

2.2 Data Collection

EEG data was acquired using a commercially available, single-channel mobile headset developed by NeuroSky Inc. This headset is shown in Figure 1 and consists of a single non-invasive electrode that presses against a subject’s forehead approximately an inch above the left eyebrow, and a ground/reference electrode that clips on to the ear lobe. This headset is capable of acquiring raw EEG signals at up to 512Hz and contains an advanced Application-Specific Integrated Circuit (ASIC) module that performs noise filtering of both EMG and 50/60Hz AC power interference.

Fig. 1. The NeuroSky MindWave Mobile headset

The EEG headset was connected via Bluetooth to a separate PC from the one used for the tasks described in Section 2.1 to avoid any interference with the use of the IDEs. Figure 2 shows a block diagram of the system architecture for the data acquisition system used in this paper.

Fig. 2. System architecture for the EEG data acquisition system
To acquire the EEG data relating to the mental effort required to navigate the graphical interfaces of the development environments considered in this paper, a simple data logging application was developed on top of the API provided by NeuroSky (Neurosky Developer Tools 2.5 PC/Mac, 2015). This API provides a simple interface for connecting to the EEG headset and acquiring data, as well as implementations of the mental effort and task familiarity algorithms that calculate task related spectral power variations (Mak et al., 2013a, 2013b). The API also helps process the raw EEG data and remove artefacts, such as that produced by eye movement or eye blinks, using wavelet analysis (Zikov, Bibian, Dumont, Huzmezian, & Ries, 2002).

The mental effort algorithm measures the workload exerted on a subject’s brain by the task that they are performing. Raw EEG signals from a single channel frontal EEG device are sampled at 512Hz and filtered to remove both eye movement artifacts and electromagnetic interference are acquired and the band powers (i.e. the power spectral densities of each EEG band) calculated. The band power of the upper alpha band EEG signals (11-14Hz) in particular is then used to calculate the mental effort of the subject performing the task. Mak et al. (2013a) have shown that these upper alpha band EEG band powers exhibit consistent, statistically significant increases in mental workload. Mak et al. (2013a) have shown that there is some correlation between theta band power and mental workload, but these are only statistically significant at the beginning of difficult tasks.

The task familiarity algorithm measures how well a subject is learning a specific task. As a subject learns how to perform common tasks in an IDE for example, they become more familiar with it and thus the task familiarity measurement increases. To calculate task familiarity, the raw EEG signals are processed in the same way as for Mental effort (i.e. they are sampled and filtered to remove common artifacts). Mak et al. (2013b) have shown that, as a subject becomes more familiar with a given task, EEG activities in all frequency bands decrease. However, decreases in the delta and gamma bands were particularly significant - though in the case of gamma waves this decrease was only statistically significant ($p < 0.05$) at the start of tasks.

A UML activity diagram for the data logging application used in this research is provided in Figure 3 showing the work-flow through the application. One of the key activities within this work-flow is the initial calibration of the mental effort and task familiarity metrics. This involves the subject relaxing with eyes open for 60 seconds before starting the task to allow for the calculation of initial baseline values for both mental effort and task familiarity. These baseline values can then be used as a reference for comparing later values. New values for mental effort and task familiarity are calculated continuously every 10 seconds. It should also be noted that the application also checks the signal quality every second to ensure that the headset is connected and positioned properly - if a poor signal quality indicator is detected then no data is collected and the trial facilitator is notified so that they can help readjust the headset.

Fig. 3. UML activity diagram for the data acquisition application
In order to aid the experiment facilitators in monitoring the progress of tasks and ensuring that the subject’s headset connection is correct, a simple user interface has been created which plots the last 300 values obtained for mental effort and task familiarity. This user interface is shown in Figure 4 and also allows the facilitator to temporarily suspend the trial. This was useful in the event that the head-set signal was lost.

![Fig. 4. Graphical user interface for the data acquisition application](image)

Observations made by a facilitator were recorded in conjunction with the collection of EEG data during each task for every participant. These observations included noting:

1. When a participant had made an error.
2. When a participant had stopped referring to the exercise sheet.
3. When a facilitator was required to intervene in the task.
4. Comments made by the participant throughout the experiment.

Also in conjunction with the collection of EEG data, the participant was asked to complete a questionnaire after completing each of the eight tasks. This questionnaire would simply ask:

1. Was this task difficult? Answers: Not difficult; Indifferent; Difficult.
2. Did you make any mistakes? Answers: Yes; No.
3. Did the facilitator intervene? Answers: Yes, No.

### 2.3 Data Processing

The EEG data recorded by the software was stored in comma separated log files per participant. EEG data for each task was identified using a boolean value which would indicate whether a participant was “Resting” or “Active” (completing a task). The sequence for each log file would alternate from “Resting” to “Active” eight times (for each task) and finish on a final “Resting” state. This data would be copied to a MATLAB data structure with clear separation of the “Active” states and “Resting” states.

All EEG data for mental effort and familiarity has been normalised (between 0 and 1) to account for variance in calibration and base mental effort and familiarity for each participant. The loss in the raw range and magnitude of data as a result of the normalisation is not considered an issue as this study is interested in the relative increase or decrease of mental effort and familiarity across eight tasks for all participants. 1-D data interpolation was achieved by cubic convolution in MATLAB R2014b (using the “v5cubic” method). This allowed the mental effort
and familiarity data for each task to be contrasted between participants who completed the tasks in different amounts of time.

The observations made by the facilitator have been translated into binary matrices indicating which tasks had mistakes that were made by the participants, and at which task the participants stopped referring to the exercise sheet. The perceived difficulty of the tasks by the participants have also been translated into binary matrices indicating which tasks were perceived as difficult. These matrices will be used when plotting the data to aid the explanations of the behaviour of the recorded mental effort and familiarity, as it is expected that if a mistake is made in a task it will result in anomalous readings.

3 Results

This mental effort and familiarity data has been plotted in Figures 5 and 6 for Visual Studio respectively, and Figures 7 and 8 for Eclipse respectively. The plots include the mean averages of mental effort and familiarity data over each task as well as a plot of the linear best fit and indication of the slope.

Overall, the mean average results for familiarity across both the Eclipse and Visual Studio IDEs show the trend of moving from familiarity readings, which are ascending in the initial tasks, to familiarity readings which are descending in the later tasks. This is in accordance with the prediction that over time, a participant will become more familiar with the task which they have been asked to repeat for a total of eight times. In the samples taken and illustrated in Figures 8 and 6, it can be observed that on average the participants did not begin to show signs of increased familiarity with the Visual Studio IDE until the third task, and in contrast it can be observed that on average the participants began to show signs of increased familiarity with the Eclipse IDE after the first task. This observation is supported further by the observed mistakes and perceived difficulty matrices illustrated in the Figures, where it can be seen that five of the six participants for the Visual Studio IDE made mistakes in task 2, whereas only one of the six for the Eclipse IDE made a mistake in the same task.

In addition, only two out of the six participants who were completing the Visual Studio IDE exercises reported occurrences of perceived difficulty for the first task. This sample group reached a total of 23 mistakes throughout the experiments. In contrast, four out of the six participants who were completing the Eclipse IDE exercises reported occurrences of perceived difficulty for the first task. This sample group reached a total of only 16 mistakes throughout the experiments.

Overall, the mean average results for mental effort across both the Eclipse and Visual Studio IDEs show the trend of moving from mental effort readings, which are ascending in the initial tasks, to mental effort readings which are descending in the later tasks. This is again in accordance with the prediction that over time, a participant will become more familiar with the task which they have been asked to repeat for a total of eight times, and therefore require less of a mental effort to complete the tasks.

The figures show that the majority of the Eclipse IDE participants stop referring to the exercise sheet at task 3, whereas the majority of Visual Studio participants stop referring to the exercise sheet at task 4. Once participants stopped looking at the exercise sheets, the majority of Visual Studio IDE participants began to make mistakes soon after. This resulted in the mental effort for Visual Studio IDE participants to increase in task 4, where four of the six participants made a mistake. This caused an anomaly in the descending mental effort trend as the tasks were repeated. A similar event occurred for the Eclipse IDE participants on task 6, where three of the participants made a mistake and as a result the average familiarity reading for that task showed a descending slope.

All participants using the Visual Studio IDE found it difficult to create a C# project; instead they made the mistake of creating a Visual Basic project. A facilitator intervened whenever there
Fig. 5. Plots of the post-processed Mental Effort data over time obtained from the EEG head-set for participants using the Visual Studio IDE. Red lines indicate an error observed by the facilitator. Dashed lines indicate a perceived error by the participant. Shaded backgrounds indicate the task from which a participant stopped referring to the exercise sheet. Final row indicates mean average of task data, linear best fit, and gradient of the slope.
Fig. 6. Plots of the post-processed Familiarity data over time obtained from the EEG head-set for participants using the Visual Studio IDE. Red lines indicate an error observed by the facilitator. Dashed lines indicate a perceived error by the participant. Shaded backgrounds indicate the task from which a participant stopped referring to the exercise sheet. Final row indicates mean average of task data, linear best fit, and gradient of the slope.
Fig. 7. Plots of the post-processed Mental Effort data over time obtained from the EEG head-set for participants using the Eclipse IDE. Red lines indicate an error observed by the facilitator. Dashed lines indicate a perceived error by the participant. Shaded backgrounds indicate the task from which a participant stopped referring to the exercise sheet. Final row indicates mean average of task data, linear best fit, and gradient of the slope.
Fig. 8. Plots of the post-processed Familiarity data over time obtained from the EEG head-set for participants using the Eclipse IDE. Red lines indicate an error observed by the facilitator. Dashed lines indicate a perceived error by the participant. Shaded backgrounds indicate the task from which a participant stopped referring to the exercise sheet. Final row indicates mean average of task data, linear best fit, and gradient of the slope.
was a clear departure from the exercise sheet. Most of the participants found it difficult to create a class.

The majority of participants using the Eclipse IDE were unable to close the Eclipse welcome tab, this prevented them from seeing the project they had just created. The Eclipse welcome tab appeared to cause the greatest confusion for participants, one participant was quoted to say “I’m not entirely sure where the project is located after creating it”.

4 Conclusion

It was observed that the Eclipse users finished quicker than Visual Studio users. Eclipse users also made fewer errors than Visual Studio users. Generally, the participants found the Eclipse user interface easier to use. The EEG data recorded for mental effort and familiarity also suggest that the Visual Studio Integrated Development Environment (IDE) participants took longer to become familiar with the interface and required a higher mental effort when repeating the same task eight times.

The mental effort and familiarity data recorded by the EEG headset shows trends which correlate with the observations made by both the facilitator and the perceived difficulties of the participant. The results suggest that this method of evaluating Human Computer Interaction (HCI) in terms of familiarity and mental effort with IDEs is feasible using a low-cost EEG headset solution.

A drawback to this approach is the unpredictable behaviour of participants who undertake the tasks. Participants will at some point stop referring to the exercise sheets (in particular when repeating tasks) which will cause unpredictable readings. This is also the case with the unpredictable readings produced when a participant makes a mistake or a facilitator has to intervene. Future research should focus on the consideration of these events when processing the data.

References


Appendix
Calibration
The method used for data acquisition requires that the participant is connected to the headset for an initial calibration. The facilitator will ask that you position yourself (seated) and look out the window towards the sky. This will last for 60 seconds whilst the calibration completes, during which time you will be required to:

1. Deliberately relax all muscles.
2. Clear mind of any specific thoughts.
3. Let mind wander and drift.
4. Relax your breathing.
5. Keep eyes open but relaxed.

Tasks
Please complete the following tasks. A facilitator will intervene after you complete each task to reset configurations for the next task. A facilitator may intervene to correct your actions in completing tasks.

Task 1
Please complete the following instructions:
1. Please open the Visual Studio IDE from the desktop, and follow the interface instructions until you are presented with the main IDE interface (no obstructing dialogue boxes).
2. Please create a new Visual C# Console Application Project. Please name the project “Task 1”
3. Please Add a new C# class file and name the class file “Task1”
4. Please copy the following code into the code editor within Program.cs, after “static void Main(string[] args){”:
   System.Diagnostics.Debug.WriteLine(“Hello Task1”);
5. Please Run the code and note the output generated by the above code.

Task 2
Repeat Task 1, using “Task 2” as your project name, and “Task2” as your class file name.

Task 3
Repeat Task 1, using “Task 3” as your project name, and “Task3” as your class file name.

Task 4
Repeat Task 1, using “Task 4” as your project name, and “Task4” as your class file name.

Task 5
Repeat Task 5, using “Task 5” as your project name, and “Task5” as your class file name.

Task 6
Repeat Task 6, using “Task 6” as your project name, and “Task6” as your class file name.

Task 7
Repeat Task 7, using “Task 7” as your project name, and “Task7” as your class file name.

Task 8
Repeat Task 8, using “Task 8” as your project name, and “Task8” as your class file name.

Fig. 9. The exercise sheet for completing the eight trials for the Visual Studio IDE. These were presented to the participants of the experiments.
**Calibration**

The method used for data acquisition requires that the participant is connected to the headset for an initial calibration. The facilitator will ask that you position yourself (seated) and look out the window towards the sky. This will last for 60 seconds whilst the calibration completes, during which time you will be required to:

1. Deliberately relax all muscles.
2. Clear mind of any specific thoughts.
3. Let mind wander and drift.
4. Relax your breathing.
5. Keep eyes open but relaxed.

**Tasks**

Please complete the following tasks. A facilitator will intervene after you complete each task to reset configurations for the next task. A facilitator may intervene to correct your actions in completing tasks.

**Task 1**

Please complete the following instructions:

1. Please open the Eclipse IDE from the desktop, and follow the interface instructions until you are presented with the main IDE interface (no obstructing dialogue boxes).
2. Please create a new Java Project. Please name the project **“Task 1”**
3. Please Add a new Java class file and name the class file “Task1”. Please check the “public static void main(String[] args)” box.
4. Please copy the following code into the code editor within Task1.java, after “public static void main(String[] args) {”:
   
   System.out.println(“Hello Task1”);

5. Please Run the code and note the output generated by the above code.

**Task 2**

Repeat Task 1, using **“Task 2”** as your project name, and **“Task2”** as your class file name.

**Task 3**

Repeat Task 1, using **“Task 3”** as your project name, and **“Task3”** as your class file name.

**Task 4**

Repeat Task 1, using **“Task 4”** as your project name, and **“Task4”** as your class file name.

**Task 5**

Repeat Task 5, using **“Task 5”** as your project name, and **“Task5”** as your class file name.

**Task 6**

Repeat Task 6, using **“Task 6”** as your project name, and **“Task6”** as your class file name.

**Task 7**

Repeat Task 7, using **“Task 7”** as your project name, and **“Task7”** as your class file name.

**Task 8**

Repeat Task 8, using **“Task 8”** as your project name, and **“Task8”** as your class file name.

---

*Fig. 10.* The exercise sheet for completing the eight trials for the Eclipse IDE. These were presented to the participants of the experiments.
Confidence, command, complexity: metamodels for structured interaction with machine intelligence

Advait Sarkar

Computer Laboratory, University of Cambridge
William Gates Building, 15 JJ Thomson Avenue, Cambridge, UK
advait.sarkar@cl.cam.ac.uk

Abstract. Programming is a form of dialogue with machines. In recent years, we have become increasingly involved in a dialogue that shapes our surroundings, as we come to inhabit a newly inferred world. It is unclear how this dialogue should be structured, especially as the notion of “correctness” for these programs is now unknown or ill-defined. I present a speculative discussion of a potential solution: metamodels of machine cognition.

Keywords: POP-II.B. Program Comprehension; POP-IV.B. User Interfaces; POP-I.C. Ill-Defined Problems

Fig. 1. Are you thinking what I’m thinking? The old paradigm; programmer and program communicate primarily through direct channels of inspection.

1 Introduction: a paradigm shift in programming

We increasingly inhabit an inferred world. The dominant mode of programming is changing. To explain more clearly, I shall first paint a simplified caricature of the traditional programming paradigm. Figure 1 shows a diagram, representing the traditional interaction between programmer and program. Here, the programmer has a goal mental model of the information structure to be built. Through a direct channel, such as inspection of the source code, its output, and execution traces, the programmer is able to build a mental model of the information structure as it currently is. Thus, the programmer is able to compare these two models against each other in order to decide whether the program matches the goal, whether it is incomplete, or whether it contains errors.

The old paradigm is characterised by the utility of the explicit data in the direct channel. The expected output is sufficiently well-defined that should the output depart from the programmer’s expectations (i.e., an error), an inspection of the workings of the program will suffice to resolve the situation (i.e., debugging). A great deal of study (§2) has been conducted on enriching the debugging experience with implicit information through the “indirect” channel, for example, through descriptions of the program, through inspections of its time and memory requirements, and through visualisations of its operation. Nonetheless, it is still possible, and in many cases sufficient, to conduct debugging through direct inspection of the program source code, output, and traces. Thus, the activities surrounding the traditional programming paradigm can be summarised as “are you thinking what I’m thinking?”

1.1 End-user machine learning is the new programming

Programs are different now, however. We increasingly inhabit an inferred world (Blackwell, 2015), and the outcome of computer algorithms is becoming predominantly probabilistic and data-dependent, rather than deterministic.
The training of machine learning models can be regarded as an act of programming. End-users of systems such as recommendation systems (e.g., Amazon’s product recommendations, Pandora’s music recommendations), intelligent personal assistants (e.g., Apple’s Siri, Microsoft’s Cortana, Google Now), and intelligent consumer tools (e.g., Excel’s Flash Fill) etc. increasingly find themselves implicitly or explicitly programming their environment. However, our interaction with these systems largely remains opaque to their decision making process, which often involves considerable uncertainty. When the output departs from our expectations, neither are our expectations well-defined, nor does inspecting the workings of the system resolve the situation – this is where a dialogue is necessary.

Figure 2 illustrates how the inferred world has shifted the predominant programming paradigm. Programming in the new paradigm is characterised by the following three properties:

1. The “programs” are stored as massive quantities of model parameters, and thus are largely human-unintelligible.
2. The programmer is likely to be an end-user programmer who is not necessarily skilled at computing.
3. The goal state of the program is unknown or ill-defined.

The combination of these makes the traditional direct channel inapplicable, and can be summarised as a “what are we thinking?” approach to programming, where mental models of neither goal nor program are well-structured. As a consequence, we must place greater care with the way we exploit the indirect channel, that is, we must shift from an emphasis on facilitating the user’s understanding of the program to their understanding about the program. Previous debugging literature has by no means ignored this channel, and neither has the interactive machine learning literature. However, treatment of this channel has typically been on an ad-hoc basis. By explicitly acknowledging the interaction as dialogue, we are able to take a structured approach, which is descriptive as well as prescriptive.

In this paper, I propose a fundamental addition to the indirect programming channel: metamodels of machine cognition. Machine learning consists of algorithms which model their output as functions of their input. But the output of a machine learning model alone does not suffice for a rich interactive dialogue. Is the model confident in its own output? Has the model had adequate exposure to the domain? If these were known, we might be able to critically appraise its predictions in a wider context. We might be able to direct the learning of the model, to expose it to parts of the domain it still does not know about, or to provide appropriate training data to help improve its confidence. How complex was the prediction to make? If this was known, we might be able to spot and rectify trivial simplifications of the target domain that the algorithm is exploiting in order to make predictions.

### 2 Related areas of inquiry

#### 2.1 Interactive machine learning

Our primary application domain of interest is the field of interactive machine learning. An early exploration of how a visual interface might enable end-users to effectively build classifiers is given by Ware et al. (2001), describing the graphical interface for the popular Weka machine learning toolkit. However, their application was still very much directed towards expert statisticians. Subsequently, the work in the area has become focused on end-users with less awareness of statistical and computing concepts.

Perhaps the archetype of the field is the eponymous paper demonstrating the Crayons application (Fails & Olsen, 2003), where users could train a classifier for image segmentation by directly sketching over parts of the image to indicate positive or negative examples. Fogarty et al. (2008), and more recently Kulesza et al.
(2014) tackle the problem of labelling concepts in images, where “concepts” are not always predefined classes, but rather can be evolved over the course of the labelling exercise. Fiebrink et al. (2011) demonstrate interactive model training for realtime music composition and performance. To some extent, one can also consider the following to be examples of interactive machine learning: Brown et al. (2012), who demonstrate a visual interface for specifying distance functions, and Hao et al. (2007), who show how data visualisations can be used as a querying interface.

Fails and Olsen motivate their work by emphasising the ease of generating a classifier in an interactive visual manner. Similarly, Fogarty’s, Brown’s and Hao’s systems are presented from primarily an ease-of-use view. My own work in end-user machine learning in spreadsheets (Sarkar et al., 2014) focuses on ease-of-use. These systems achieve ease of use by massively abstracting away the workings of the system, which is generally a useful strategy, as long as the behaviour of the program corresponds to user expectations. But what happens when the system gets it wrong, and not in a way that is easily apparent (Szegedy et al., 2013; Nguyen et al., 2015)? To better involve the user in the process, the repeated use of the word “explain” throughout the interactive machine learning literature (Herlocker et al., 2000; Tintarev & Masthoff, 2007; McSherry, 2005; Pu & Chen, 2006) does not appear to be coincidental; clearly the underlying aim is to give our interactions with programs much more of a dialogue-like quality.

A critical assessment of end-user interaction with machine learning has been made by Amershi et al. (2011). The authors identify a few questions for this dialogue: What examples should a person provide to effectively train the system?, How should the system illustrate its current understanding?, and How can a person evaluate the quality of the system’s current understanding in order to better guide it towards the desired behavior? Systematic metacognitive modelling provides a partial answer to all of these questions.

Lim & Dey (2009) have directly addressed the problem of what types of information about intelligent applications should be given to end-users. They call these “intelligibility types,” and some examples of these are as follows: Input & output: what information does the system use to make its decision, and what types of decision can the system produce? Why, why not, & how: why did the system produce the output that it did, why did it not produce a different output, and how did it do so? What if: what would the system produce under given inputs? Modek: how does the system work in general? Certainty: how certain is the system of this report? Control: how can I modify various aspects of this decision making process? Kulesza et al. (2013) show that these information types are critical for the formation of users’ mental models. Kulesza et al. (2011) also proposed a set of information types which would benefit end-users who were debugging a machine-learned program, including: Debugging strategy: which of many potential ways of improving the model should be picked? Model capabilities: what types of reasoning can the model do? User interface features: what is the specific function of a certain interface element? Model’s current logic: why did the model make certain decisions? User action history: how did the user’s actions cause the model to improve/worsen? This presents excellent motivation for systematic metacognitive modelling, without which such information cannot be generated.

2.2 End-user debugging

The producers of these machine learning models are also their users. As such, this is related to end-user software engineering (Ko et al., 2011), and in particular end-user debugging. Interestingly, end-user debugging has so far been quite explicit in framing the interaction as dialogue. Wilson et al. (2003) argue that programming assertions in spreadsheets is difficult and boring, and present a strategy to incentivise users to write more assertions. This strategy – surprise, explain, reward – is much like dialogue. The software generates what it thinks is a surprising assertion that nonetheless fits a cell’s formula. It changes the value of the cell to be valid under this assertion, and explains this decision and how to change the assertion through a tooltip. Finally, the user is “rewarded” by virtue of having a more correct spreadsheet.

Ko & Myers (2004) present the “WhyLine,” a debugging tool that is meant to operate literally as dialogue. By scanning the function call structure of a program execution, the tool can create hierarchical menus which allow the user to formulate grammatically correct “why” questions about the execution of a program. Interestingly, Kulesza et al. (2009, 2011) take this approach to facilitate end-user debugging of the underlying naïve Bayes model of an email spam classifier.

As with interactive machine learning, allusions to “explanations” also appear throughout the end-user debugging literature. However, there is an important distinction to be made between the type of dialogue one engages in when debugging, and the type of dialogue one has with a machine learning model. The activity of “debugging” principally occupies the direct interaction channel, as in the old paradigm. Treating the training of machine learning models as debugging can only be informative for interaction design up to a point. The debugging situation assumes that the user’s mental information structure is the correct version, which the computer’s internal information structure must aim to reproduce. That is, we assume the human knows the right answer. This is not to say that the programmer always knows how to concretely express the required information structure in a given programming language; perhaps the programmer receives assistance from the system, as in WYSIWYG (Rothermel et al., 1998). However, in the old paradigm, the final arbiter of what is, and is not a “bug,” is the programmer.
Conversely, in the new paradigm, the right answer is either unknown or ill-defined. It follows that under these circumstances, “debugging,” or even a “bug,” cannot definitively exist. In the class of situations we are dealing with today: product recommendations, automated diagnostics, weather forecasting, etc., neither the human nor the computer knows the right answer, but rather they are in a dialogue to try and resolve the issue together. Thus, both parties must be transparent to one another. I suspect that one of the reasons Teach and Try (Sarkar et al., 2014) was so successful at generating an understanding of statistical procedures in non-experts is the deliberate selection of the word “Try” as opposed to “Fill” or “Apply model”; it implies fallibility and evokes empathy.

2.3 Mixed-initiative interaction

Mixed-initiative interaction explicitly acknowledges that program behaviour could be usefully augmented by models that were not strictly about the problem domain. In this case, the models being made are of the user, and of user intent. Horvitz (1999) argues that mixed-initiative systems (i.e. automated services) must exhibit certain “critical factors”, or principles. The most pertinent of these to this paper is that decisions must be made under awareness of uncertainty about user goals, and the cost of distracting the user.

As a case study, Horvitz uses a calendaring service which automatically parses emails for event date/time information and suggests actions based thereupon. Importantly, Horvitz provides a decision-theoretic heuristic for taking an action based on an expected utility function. This function is calculated given beliefs about a user’s goals derived from observed evidence. Action is taken when the utility for action exceeds that of inaction. In order to implement this, an explicit utility model must be built and updated as the user interacts with the software. This idea can be adopted for our use in the new programming paradigm, not to model the user, but to model the program itself.

3 A proposition: models of machine cognition

From interactive machine learning, I take the pertinent and emergent domain of end-user programmers of machine learning models. From end-user debugging, I adopt the strategy of treating interaction as dialogue. From mixed-initiative interaction, I appropriate the strategy of developing explicit metamodels, to consider thinking about the program rather than of the program itself. Consequently, I propose that it is a useful, systematic strategy to augment machine learning models with metamodels. What should be the subject of these metamodels, and how many are required? Let us begin with the following:

1. **Confidence**: how sure is the program that a given output is correct?
2. **Command**: how well does the program know the domain?
3. **Complexity**: did the program do a simple or complex thing to arrive at the output?

I believe that these three are necessary for successful dialogue of the kind outlined in the introduction. They are not exhaustive, but have emerged to be clearly important from careful consideration of the engineering requirements for improving end-user programming of machine learning models in a variety of scenarios, which shall be elaborated in §4.

The metamodels are intimately related to the information types proposed by Lim & Dey (2009) and Kulesza et al. (2011). Those frameworks prescribe types of information which would be beneficial to an end-user programmer of machine learning models, but do not prescribe how such information might be generated. So while these metamodels are a conceptual solution at the same level as the intelligibility types, i.e., they prescribe things which should be shown to the user, they are also an engineering solution at a technical level, i.e., they prescribe how this information can be generated. With systematic metamodelling, it may not be necessary to recreate methods for providing intelligibility for each new interface and machine learning system on an ad-hoc basis.

In the following subsections, I shall illustrate and elaborate upon each of these three metamodels in turn.

3.1 Confidence

Confidence has been dealt with throughout the statistics and machine learning literature. Methods for estimating the error or confidence for any given output have been developed for many models. Linear regression, one of the simplest statistical models, is accompanied by a procedure for computing the 95% confidence intervals for its learnt parameters, which can be interpreted as confidence: the narrower the intervals, the more confident the prediction. However, being able to estimate this confidence is not necessarily incentivised in benchmarks of machine learning performance, which are primarily concerned with the correctness of the output.

Table 1 presents some suggestions for how confidence may be computed for popular machine learning techniques. Measures of confidence can be used to prioritise human supervision of machine output; when there are large quantities of output to evaluate, the user’s attention can be focused on low-confidence outputs, which may be problematic. González-Rubio et al. (2010) use this approach to improve interactive machine translation, and
Kulesza et al. (2015) use this approach to improve interactive email classification. Behrisch et al. (2014) show a vision of enriched dialogue, made possible through a confidence metamodel: in their software, the user interactively builds a decision tree by annotating examples as “relevant” or “irrelevant,” but is able to decide when the exploration has reached convergence due to a live visualisation of how much of the data passes a certain threshold for classification confidence.

Table 1. Practical confidence metamodel suggestions

<table>
<thead>
<tr>
<th>Model</th>
<th>Suggested calculations of confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-NN</td>
<td>For a given prediction, confidence can be measured as the mean distance of the output label from its k nearest neighbours as a fraction of the mean pairwise distance between all pairs of training examples. A similar metric is proposed in Smith et al. (1994).</td>
</tr>
<tr>
<td>Neural Network</td>
<td>For a multi-class classification, where each output note emits the probability of the input belonging to a certain class, confidence can be measured simply as the probability reported. More sophisticated confidence interval calculations can be obtained by considering the domain being modelled, as in Chryssolouris et al. (1996); Weintraub et al. (1997); Zhang &amp; Luh (2005).</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>The confidence of a decision tree in a given output can be measured as the cumulative information gain from the root to the outputted leaf node. Alternatively, Kalkanis (1993) provides a more traditional approach.</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>The confidence of a Naive Bayes classifier in a given prediction can be measured as the probability of the maximally probable class. More sophisticated treatment of the problem is given by Laird &amp; Louis (1987); Carlin &amp; Gelfand (1990).</td>
</tr>
<tr>
<td>Hidden Markov Model</td>
<td>The primary tasks associated with HMMs (filtering, prediction, smoothing, and sequence fitting) all involve maximising a probability; the confidence can simply be measured as the probability of the maximally probable output. More fine-grained confidences can be measured by marginalising over the relevant variables (Eddy, 2004).</td>
</tr>
</tbody>
</table>

Confidence alone, however, can be deceiving. Recent work (Szegedy et al., 2013; Nguyen et al., 2015) has demonstrated how some apparently straightforward images with carefully injected noise, as well as completely unrecognisable images, are still classified with high confidence by a state-of-the-art image classifier. Thus, confidence is not the end of the story when it comes to understanding a machine’s abilities – it may be necessary, but is not sufficient.

3.2 Command

Addition of a second metamodel, “command,” is a further step towards enriching the description of machine understanding. It has been expressed in various forms in the literature. The dream of a self-regulated, autonomous agent is long lived in GOFAI and modern machine learning, motivated by such issues as the “exploration-versus-exploitation” tradeoff; i.e., should the agent do something which has been known to provide a certain reward, or should the agent explore the wider world in search of potentially better rewards, at the risk of wasting resources on less-rewarding world states?

Systems developed towards this aim often exhibit primitive forms of metacognition. A most basic example of a famous problem which benefits from this form of metacognition is that of the multi-armed bandit (Gittins et al., 2011). A gambler at a row of slot machines has to decide which machines to play, how many times to play each machine, and in which order to play them, in order to maximise the cumulative reward earned. Each machine provides a random reward from a distribution specific to that machine. Thus, the tradeoff is between exploration, i.e., playing machines in order to learn about their reward distributions, and exploitation, i.e., playing machines in order to gain the reward. A solution to this problem must necessarily involve a model of command, i.e., how much is known about the reward distribution of each machine, in order to effectively navigate this tradeoff.

Similarly, the concept of reinforcement learning (Watkins, 1989) involves a “reward function”, which records the reward an intelligent agent might hope to receive upon transitioning to any given world state; the agent can then probabilistically transition to world states that will either fulfil its information need by updating the reward function, or alternatively will pay off by way of actually receiving the reward. A related concept is active learning (Cohn et al., 1996; Settles, 2010), where the algorithm is able to select examples it believes to be most useful for its learning, and presents these examples to a human oracle (or other information source) for labelling. The motivation behind active learning is similar to exploration-vs-exploitation: that the algorithm may achieve greater accuracy with fewer training examples should it choose the data from which it learns. Savitha et al. (2012) show a “metacognitive” neural network which can decide for itself what, when, and how to learn from each training datum it is given. Interestingly, common to these techniques is their reliance on an additional model, that of the input domain, so that the agent is able to distinguish between what is known and what remains to be known. In
reinforcement learning, this takes the form of the state space. Thus, any practical definition of “command” has to be constructed in relation to a definition of the domain being modelled.

I can suggest two simple methods of illustrating the command of a machine learning algorithm over a certain domain. The first is to look at the training examples the classifier has so far received, as positioned in the input domain. The second is to look at the classifier’s confidence at all points in the domain. These are illustrated in Figure 3. It is clear that these two images paint a very different picture of the algorithm’s “command” over a domain. If we view command as some integral of confidence, then an algorithm with high levels of confidence in the majority of the domain can be considered to have a good command of the domain. If we view command as some integral of the occurrences of training examples encountered, then an algorithm which has received a uniform spread of training examples may be considered to have a good command of the domain.

The command metamodel is intimately related to the problem, in interactive machine learning, of seeking relevant examples for the efficient training of a classifier. When Amershi et al. (2009) discuss how one might seek examples providing greatest information gain for the classifier, what they are really doing is building a partial command metamodel; a full metamodel would allow generative dialogue – their software would not only be able to identify examples from the existing corpus but also generate examples which satisfy perfectly the classifier’s information need (provided that the human or other oracle who will label these examples can actually do a good job (Baum & Lang, 1992)). Groce et al. (2014) approach this from the perspective of end-user classifier testing, and show various strategies for selecting a testbed of evaluation examples. A method of eliciting examples is technically isomorphic to a command metamodel, since any such method must be able to define and identify deficiencies in the machine’s training.

While algorithmic notions of “confidence” and “command” are not completely novel, they are usually not considered for the benefit of an advanced dialogue between human and model. The notion of “confidence”, which is most mature, simply quantifies to human minds the quality of the prediction, but does not always suggest a further course of action. The notion of “command”, in the case of reinforcement learning, is internal to the intelligent agent and embedded in its data structures, and not amenable to presentation or interpretation.

### 3.3 Complexity

The notion of complexity is the least discussed, and perhaps most interesting. How can exactly the same model produce more or less complex results? Consider the case of a neural network. It can be argued that when an input highly activates many of the nodes in a neural network, the decision making process is more complex than one which involves fewer nodes. This, despite the fact that the model structure is identical, with identical edge weights. It is analogous to the difference between mentally computing 199+101 and 364+487. One can follow an identical arithmetic “algorithm,” and be equally confident in both answers, but one of these instances appears to be more complex than the other. A lot of what we think of as “complex” behaviour arises not out of a complex algorithm, but rather complex inputs. Deep Blue may have astonished with it famous defeat of Kasparov in 1997, but it did so not because it was following a complex algorithm; far from it. It did so because the input space and the domain carried with it considerable complexity. This idea is encapsulated in the allegory of Simon’s ant.
The complexity of a given prediction can be measured as the variance of the distances for the metamodel refers to the current state of the algorithm's knowledge, not necessarily dependent on a single output. In contrast, the “command” to a given prediction. That is, whenever the model is used to make a prediction or classification, there is an associated value of confidence and complexity unique to that run of the algorithm. In contrast, the “command” metamodel refers to the current state of the algorithm’s knowledge, not necessarily dependent on a single output.

Table 2. Practical complexity metamodel suggestions

<table>
<thead>
<tr>
<th>Model</th>
<th>Suggested calculations of complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-NN</td>
<td>The complexity of a given prediction can be measured as the variance of the distances for the k nearest neighbours. A larger variance can be interpreted as a more complex decision.</td>
</tr>
<tr>
<td>Neural Network</td>
<td>Setting a threshold t above which we consider a neuron to be “activated” (e.g., for a sigmoid activation function we might set t = 0.9), we can define the notion of “t-complexity”, where the t-complexity of a neural network prediction is the fraction of the nodes in the network which are activated to level t or above.</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>The tree-depth of a prediction provides a simple measurement for the complexity of a decision. For a more complex alternative, we might set a threshold i at which we consider the “majority” of the information gain to have been achieved, and define the notion of “i-complexity”, where the i-complexity of a decision tree prediction is the tree-depth at which the cumulative information gain exceeds i en-route from the root to the outputted leaf node.</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>Each classification decision in a Naïve Bayes classifier involves summing log probabilities of individual features given a class. The complexity of a Naïve Bayes classification decision can be measured as the variance of the log probabilities; a greater variance can be interpreted as a more complex decision.</td>
</tr>
<tr>
<td>Hidden Markov Model</td>
<td>Each of the primary HMM tasks will have different models of complexity. Intuition would suggest that a “simple” decision would be robust to small perturbations of the priors, transition function, and length of sequence that the algorithm is given to operate upon. Thus, if we define a threshold p on any of these quantities, then an HMM decision can be said to be p-simple if its output is robust to perturbations of magnitude less than p in its priors/transition function/sequence length.</td>
</tr>
</tbody>
</table>

It is important to note that the confidence and complexity measures are both always computed with respect to a given prediction. That is, whenever the model is used to make a prediction or classification, there is an associated value of confidence and complexity unique to that run of the algorithm. In contrast, the “command” metamodel refers to the current state of the algorithm’s knowledge, not necessarily dependent on a single output.
3.4 Substituting metamodels as explanatory metaphors

We can take the technique of metamodelling for the facilitation of dialogue one step further, and achieve some interesting things, if we relax the accuracy constraint. That is, what if our metamodels don’t strictly model what they’re supposed to, but still provide plausible representations of that model’s confidence, command, and complexity? This would be extremely useful in a case where our machine learning model is impossible to metamodel; we can nonetheless perform metamodel substitution to provide dialogue. Perhaps the decisions of a deep neural network are impossible to easily and correctly explain to an end-user, but if we present explanations as though the system is performing case-based reasoning (previously shown to be an intuitive approach (Sarkar et al., 2014)), then that may suffice. Thus, one metamodel can be used as a metaphor for another.

3.5 Metadialogue and metainteraction

The models of confidence, command, and complexity are repeatedly referred to as meta-models. I use this in the sense of “about,” as in metadata (data about data) and metacognition (cognition about cognition), and so forth. At this level, the word “metamodel” simply refers to the fact that these are models about other (statistical and machine learning) models. However, there is room to discuss the treatment of the “meta-” prefix in the sense of “above,” denoting a higher layer of abstraction, as in metaphysics, or perhaps metatheory. The primary object of study here is the interaction, or dialogue, between user and program, and not the models themselves. While we deal with the three metamodels as descriptors of the machine cognition, they could equally be descriptors of the interaction. For instance, while “complexity” is described here as a property of a statistical model, it could also be a property of the interaction itself, and this complexity may well be more noteworthy. This appears to bear greater relation to the problem of cognitive dimensions (Green & Petre, 1996), since it relates to the user experience of information structures as borne out through its visual and notational externalisations. Thus, it is quite possible that the brand of interaction we are considering here is more suitably called meta-interaction, or metadialogue. While a thorough treatment of this terminology is beyond the scope of the current discussion, a more detailed investigation in this direction would be an interesting subject of future study.

4 Analysis and applications

In this section I discuss how some examples of interactive machine learning systems are already benefiting from metamodel implementations, and can be usefully augmented by considering additional metacognitive models, or by newly considering their existing implementations.

4.1 Image segmentation

The Crayons application due to Fails and Olsen (Fails & Olsen, 2003) is a classic example of interactive machine learning. By “painting” positive and negative examples onto an image, the user can build a classifier which is able to segment areas of an image into two classes, e.g., a classifier which can classify human skin from non-skin objects in an image. It provides direct visual feedback on the image itself, by respectively darkening or lightening the negatively and positively classified images.

If a confidence metamodel was implemented, then instead of a standard intensity of darkening or lightening, the image could be overlaid with a colour whose intensity corresponded to the confidence with which pixels were classified as belonging into one class or another, as in Figure 4. This would further help the user refine their classifier, as it would be possible to identify regions which, while correctly classified, only just cross the decision boundary and thus have low confidence.

4.2 Email classification

Kulesza, Stumpf, et al. have pursued a line of work which has investigated how one might assist users to debug rules learnt by a naïve Bayes classifier to categorise emails in various user-generated categories (Kulesza et al., 2009, 2011, 2015). They present EluciDebug, a visual tool for providing explanations of the naïve Bayes classifier’s classification decision with respect to a given email. In doing so, they build an explicit metamodel of confidence, which can be used to sort emails and focus the user’s attention on emails which may have been misclassified. They build an explicit metamodel of complexity, wherein the entire set of weights used to make the decision can be inspected through a series of bars, and thus it is apparent whether the classification was straightforward (dominated by a few clear high weights) or complex (wide distribution of potentially conflicting weights).

They also approach the development of a command metamodel; they use the sizes of different folders (which represent different classes) to explain the machine’s prior beliefs regarding the likelihood of an unknown message belonging to any given class. This approaches a command metamodel since it alludes to the distribution of training examples the machine has thus encountered. However, it does not situate these examples in the input
domain. It is possible to envision a visualisation of all training examples, along with their text, projected from the high-dimensional space in which they reside onto a 2D manifold, such that deficiencies in the algorithm’s experience can be identified. This is the precise approach taken by Amershi et al. (2009) for interactive image classification. A full command metamodel, defined with respect to the input space (finite-length finite-dictionary word vectors), would also enable the effective eliciting of appropriate training examples. This could take the form of either identifying emails which would greatly improve the overall confidence of the classifier if a label was obtained for them, or could extend to the artificial synthesis of an email whose label would satisfy the classifier’s information needs.

4.3 Concept evolution in images

The CueFlik application due to Fogarty et al. (2008) presented a visual, programming-by-example method for designing classification rules to sort images in a database into different categories. Kulesza et al. (2014) expand upon this by acknowledging that users may not initially, or ever, have well-defined mental concept models (a key characteristic of the new programming paradigm discussed in §1.1), and so provide an interactive experience whereby the user is walked through a sequence of images which can be selected as belonging to a suggested class, or not. Automatic summaries of categories are generated to help the user remember what was distinctive about a particular category. Similar images from the corpus are displayed to assist the user in deciding whether creating a new category is warranted. Thus, the user can simultaneously refine their understanding, as well as the machine’s understanding, of the categories they are creating.

Kulesza et al. provide suggestions for classes based on a recommendation system-like algorithm which compares the similarity of the image currently being classified to already-categorised images. Currently, the only feedback presented is in the form of a yellow star icon being placed next to the category the algorithm thinks is most appropriate. Using a confidence metamodel, one can envision an interface where different category labels are ranked, sized, or coloured according to the machine’s confidence. This would help users identify categories which are potentially only weakly described by the training data.

A metamodel for complexity, driven by simplified representations of the input space, would potentially alert users to trivial simplifications being exploited by the algorithm, as in the examples presented in §3.3. One example of such a simplification is as follows: while categorising images of dogs and cats, it is possible that since most pictures of dogs are taken outdoors on green lawns, and most pictures of cats are taken indoors, what one is actually training is a classifier which detects the colour green. A complexity metamodel would be able to highlight how many, or which of the input image features are being used to make a decision, enabling the user to decide when to enrich the dataset or when to prune the feature space to prevent oversimplifications of the domain.
4.4 Analytical data modelling in spreadsheets

In Teach & Try (Sarkar et al., 2014), the user follows a two-step process to perform interactive machine learning in spreadsheets. The user first selects rows in which they have high confidence, and marks them using the “Teach” button. Next, the user selects rows to which they wish to apply the model, either in the form of populating empty cells with the model’s predictions, or by evaluating the cells’ current contents against the model’s expectations. Pressing the “Try” button applies the model.

While fairly simplistic, we were able to show that the experience of interacting with the software led users to gain some appreciation of statistical procedures. During a post-experiment interview, participants were asked questions such as how might the computer be doing this?, and why might the computer make a mistake? It is important to note that none of our participants had any formal training in statistics or computing. Nonetheless, participants were able to informally articulate several potential algorithms (e.g., nearest-neighbours, case-based reasoning, and linear regression), as well as well-known issues with statistical modelling (e.g., insufficient data, insufficient dimensions, outliers, noise, etc.).

With a confidence metamodel, Teach and Try would enable users to critically evaluate its predictions. When a large number of predictions has been made, the confidence metamodel provides a heuristic with which the user can assess its performance; the user can choose to prioritise examining and correcting low-confidence predictions. With a command metamodel, it would be able to show users how the ‘taught’ rows are spread across the input domain, it would potentially be able to highlight areas of the data where receiving a user label would be beneficial, and potentially synthesise examples to be labelled.

Here, a complexity metamodel would again help users identify potential simplifications of the domain that the algorithm might be exploiting in order to perform its predictions, such as the pneumonia prediction and multiple sclerosis diagnostics examples given in §3.3. Another example might be as follows: a spreadsheet containing patient data, where each row represents a patient and each column represents various attributes of the patient (e.g., age, blood type, results of various diagnostic tests), may also have within it a ‘date’ field, representing the date that patient’s entry was recorded. Prior to a certain date, only patients with a certain severity of illness were recorded in this spreadsheet. When using this spreadsheet to help assess whether or not a new patient may have a severe illness, a visualisation of the complexity model (perhaps in the form of how much each column contributed) might reveal that a decision tree has decided that the ‘date’ field contains enough information to conclude whether or not a patient will have a severe illness, and thus predicts, incorrectly (but nonetheless confidently), that no new patients can possibly be severely ill. Spotting this simplification, the user can take corrective measures such as excluding the date field, or removing the old records.

4.5 Commercial applications

Recommender systems: a common problem with music recommender systems, such as the engines underlying Pandora or iTunes radio, is that for an accurate model of your preferences to be built, the system needs to observe many examples of your listening history. As a consequence, users of such systems typically abandon the service before an accurate model is built, leading some to seek fast-converging estimates for recommendation systems, with varying levels of success. For example, Herbrich et al. (2007) tackle the issue of effectively recommending opponents in multiplayer games. It is important that opponents are well-matched, otherwise the game is not fun to play for either party. It is also important that these recommendations converge quickly, and that it is not necessary for a player to play several mismatched games before the system is able to correctly estimate their skill.

None of these recommendation systems exposes the underlying uncertainty associated with each prediction; by showing how the confidence of the system improves over time, and how its command of the domain of your music preferences improves as it is exposed to new examples (i.e., visible indicators of progress and improvement), the user may be more sympathetic to the amount of time required to properly train such systems.

Intelligent home devices: devices in our homes are getting increasingly intelligent. For instance, the Nest thermostat\(^1\) learns your usage patterns throughout the day and begins to adjust itself. Similarly, certain refrigerators on the market will detect when you are running low on a particular item and place an online order on your behalf. These devices may ostensibly be programmed through purpose-built Internet-of-Things languages, such as IFTTT,\(^2\) however, the primary programming interfaces many of these devices will have is through direct interaction, so that these interfaces can learn over time. In these situations, it can be quite important for the system to be able to express parts of its cognition to the user.

Driverless vehicles: the prospect of an autonomous car navigating its passengers past a complex array of obstacles at great speed evokes a visceral fear and suspicion, despite the fact that a tireless, emotionless controller with nanosecond reaction times can be orders of magnitude safer than human driving. Part of the reason for this reaction is that their interfaces have thus far been presented as completely opaque; the AI is portrayed to be in

complete control and the passengers have no intervention in its decision making process. Through metamodels of confidence, a driverless car might be able to identify situations where it defers to the judgment of a human driver. Similarly, through metamodels of command, the car might be able to identify road and scenery types which it had not previously encountered, and alert the driver to this.

5 Conclusion

I have discussed how we are undergoing a paradigm shift in programming, where the dominant mode of programming has moved from one with well-definable mental models to one without. This is accompanied by a movement from a direct, explicit information channel (the program) to an indirect, implicit, meta-information channel (about the program). Previous work in explanatory debugging and interactive machine learning has shown several different items which may be present in these meta-information channels, elevating our interaction with programs to a status resembling dialogue. To this channel, I have proposed a fundamental addition: models of machine metacognition. Unlike previous frameworks, mine is grounded in the engineering requirements for providing such types of information for intelligent systems.

I have argued for the utility and primacy of three models of machine self-metacognition: confidence in a given output, command of the problem domain, and complexity of the decision making process in producing a given output. I have presented some concrete suggestions for how such metamodels might be computed for popular machine learning algorithms. I have suggested how metamodel substitution may allow us to explain complex algorithms using simpler ones as metaphors. I have postulated a link between metamodels and meta-interaction. Finally, using examples from the literature in interactive machine learning and end-user debugging, I have demonstrated how these metamodels can enrich man-machine dialogue.

Acknowledgements

Many thanks to Alan Blackwell for his guidance on writing this paper and his proofreading of drafts at various levels of completion. Thanks to Cecily Morrison for recommending some of the relevant literature. My PhD is funded through an industrial CASE studentship sponsored by BT Research and Technology, and also by a premium studentship from the University of Cambridge Computer Laboratory.
References


The construction of knowledge of basic algorithms and data structures by novice learners

Sylvia da Rosa

Instituto de Computación - Facultad de Ingeniería - Universidad de la República
darosa@fing.edu.uy

Abstract. 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, based on establishing certain parallels between general mechanisms leading from one form of knowledge to another, both in psychogenesis and in the historical evolution of ideas and theories. The most important notion of these mechanisms is the triad of stages, called by Piaget the intra, inter and trans stages. The main goal of our work is to build an instance of that model for research about the knowledge of basic algorithms and data structures constructed by novice students. This paper presents some aspects of our work, focusing on the passage from conceptual knowledge (intra-inter stages) to formalized knowledge (trans stage).

Keywords: genetic epistemology, constructing knowledge, conceptualization, formalization

1 Introduction

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)(Piaget, 1975, 1964; Piaget & coll., 1963; Piaget, 1978b), and second, from a critical analysis of the history of sciences, elaborated by Piaget and García to investigate the origin and development of scientific ideas, concepts and theories. In (Piaget & García, 1980) the authors present a synthesis of Piaget’s epistemological theory and a new perspective on his explanations about constructing knowledge. They investigate the possible analogy between the mechanisms of psycho-genetic development concerning the evolution of intelligence in children, and sociogenetic development concerning the evolution of the leading ideas and theories in some domains of science. Throughout the chapters the authors present striking examples of this analogy in relation to the history of geometry, algebra, mechanical and physical knowledge in general. The main idea of their synthesis consists of establishing certain parallels between general mechanisms leading from one form of knowledge to another - both in psychogenesis 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 generalized elements and their transformations (trans stage), integrating the constructions of the previous stages as particular cases.

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 main goal of our work is to build an instance of that model for research on the knowledge of basic algorithms and data structures constructed by novice students. Over the years we have investigated the intra-inter-trans stages in the construction of knowledge of algorithms and data structures and in previous papers (da Rosa, 2010, 2007, 2005, 2004; da Rosa & Chmiel, 2012; da Rosa, 2003) we have described our research about this passage:

– from an intra stage, in which the knowledge is instrumental, that is, in the plane of actions (the students pursue a result but are unaware of how they achieve it)
– to an inter stage giving rise to conceptual knowledge, that is in the plane of thought (the students give accurate descriptions of how they did it and why they succeeded, being aware of the coordination of their actions and the transformation of objects).

A summary of this research is included in Section 3.1. The goal of this paper is to describe our research about the passage from earlier stages above to
– a trans stage of formal knowledge (where the students are able to express algorithms in given formalisms and modify their knowledge to solve similar problems).

Regarding the methodology of our research, the passage from intra to inter stage is investigated by means of conducting individual interviews, in the sense of Piaget’s studies of genetic psychology (da Rosa, 2010, 2007, 2005, 2004; da Rosa & Chmiel, 2012; da Rosa, 2003). The passage to the trans stage is investigated by means of conducting instructional episodes where students work in groups and some formalism is introduced (mathematical language, pseudo code and/or programming languages). In this part, our methodology follows Piaget’s studies about the role of social relations and formal education in knowledge construction (Youniss & Damon, 1997; Ferreiro, 1996). The goal is to help students in establishing correspondences between the concepts that they have previously constructed and expressions of the formalisms in order to obtain formal descriptions of their solutions. The dialectic process of this construction is explained in Section 3.2 and constitutes the main goal of this paper, and includes brief descriptions included in previous work. Our investigations have been conducted with students entering university or enrolled in the final year pre-university. That means that they have no (or very little) experience with programming (in Uruguay Computer Science is not part of High School curriculum). All the research episodes were recorded and/or filmed and students wrote out some of their responses.

Finally, we offer some comments about the motivation and the questions behind our research. Several researchers consider programming as a powerful and essential subject not only in computer science studies but in other studies as well (Dowek, 2005, 2013; Wing, 2000; Peyton Jones, 2013a; Bradshaw & Woolliard, 2012; Peyton Jones, 2013b; R. Page & Gamboa, 2013). At the same time it is seen as a difficult topic both to teach and to learn, and studies in the didactics of informatics have become necessary (Holmboe, McIver, & E. George, 2001; Saeli, 2012; Hubwieser, 2013; Nickerson, 2013; Ambrosio, 2014).

In contrast to several proposals to help students in the learning of programming involving the use of some programming language or computer tool (Gomes Anabela, 2007; Mostrom, 2011; Linda Mannila, 2007; Budd, 2006; E. George, 2000; Tina Götzchi, 2003), our approach is based on observations of situations in day-to-day life in which people successfully use methods to solve problems or perform tasks such as games, ordering of objects, different kinds of searches, mathematics problems, etc. In such situations an action or a sequence of actions is repeated until a special state is reached, which can be solved easily by a straightforward action. People’s descriptions include phrases like "I do the same until ... " and "now I know how to do it", referring to cases where they use the same method and arrive at the easy-to-solve special state respectively. These descriptions are related to programming in the sense that repeating actions until a special case is reached, is formalized by recursive or iterative program instructions.

These observations lead us to formulate questions such as "does there exist any connection between the 'knowing how to' (instrumental knowledge) revealed by people solving problems and formal algorithms? If there is, what is the nature of this connection and what is the role of the instrumental knowledge in the learning process? How is this instrumental knowledge generated and how can it be transformed into conceptual knowledge? How can the algorithms that the students learn to use taken into account in the learning of programming? Will the answer to these questions help in improving the teaching and learning of programming and how should this be done?" The approach of our research arises from the above observations and questions and from studying the theory of Jean Piaget that explains the construction of knowledge and the evolution of cognitive instruments from the interaction of the subject (his/her methods) with the objects (data structures).

The following sections of this paper include: the main theoretical principles of our research (Section 2), how these are applied (Section 3), some related work (Section 4) and conclusions and further work (Section 5).

2 Main theoretical principles

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 acquired before their formalization. This problem was deeply studied by Piaget in his experiments about genetic psychology. From these he formulated a general law of cognition (Piaget, 1964, 1978b), governing the relationship between know-how and conceptualization, generated in the interaction between the subject and the objects that he/she 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

\[ C \leftarrow P \rightarrow C' \]

1 Piaget’s ideas about social construction are integral to his epistemological theory but less known than those about child’s construction of logical thought.
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 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 his/her actions ($C$ in the diagram), the modifications 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 the general law of cognition constitutes a first step towards the construction of concepts.

Piaget also describes the cognitive instrument enabling these processes, which he calls the **reflective abstraction** and **constructive generalization**. Reflective abstraction is described as a two-fold process (Piaget, 1964): in the first place, it is a projection (transposition) to the plane of thought of the relations established in the plane of actions. Second, it is a reconstruction of these relations in the plane of thought adding a new element: the understanding of conditions and motivations. The motor of this process is called by Piaget the search of reasons of success (or failure). On the other hand, facing new problems presenting variations and similarities with the old ones causes a desequilibrium of students’ cognitive structures which have to be transformed in order to attain a new equilibrium, making possible the construction of appropriate knowledge to solve the new situation. Once a particular method is understood, students’ reasoning attempts to generalize what has been successfully constructed to all the situations, by means of inductive generalization where inductions or predictions are extracted from observations of the new objects. A process of inferences and reflections about the subject’s actions or operations by means of constructive generalization gives raise to new methods (Piaget, 1978a, 1975; Jacques Montangero, 1997) and opens possibilities for constructing structures characteristic of the trans stage.

The table below summarizes the main points of the theory related to our methodology of research.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Goals</th>
<th>Cognitive tools</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>individual</td>
<td>actions → operations</td>
<td>reflective abstraction</td>
<td>intra-inter</td>
</tr>
<tr>
<td>interviews</td>
<td>search of reasons</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>detaching concrete cases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instructional</td>
<td>formal description</td>
<td>reflective abstraction</td>
<td>inter-trans</td>
</tr>
<tr>
<td>instances</td>
<td>operations → structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>work groups</td>
<td>inductive and constructive generalization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>similar problems</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3 Applying the theory

Our previous work focused on the first part of our studies, in which we conducted individual interviews applying the general law of cognition described above, in the manner of (Piaget, 1964), accounting for the passage from intra to inter stage. In Section 3.1 we include a summary of these investigations. The main goal of this paper is to describe the second part of our studies, where instructional instances were conducted and the students worked in groups. The main point is to illustrate, on the one hand, the way we introduced a formalism and encouraged the students to represent their descriptions of algorithms in it, and on the other hand, how students attempted to solve new problems presenting similarities and differences to those already solved, applying previously constructed concepts. This is presented in Section 3.2 using as an example, the study of the construction of knowledge about sorting algorithms.

### 3.1 Summary of investigations conducted through individual interviews

This section includes a summary of our previous work (da Rosa, 2010, 2007, 2005, 2002, 2004; da Rosa & Chmiel, 2012). The problems that students had to solve were instances of some of the problems studied in basic programming courses (sorting, searching, counting) and the objects were instances of data structures (a paper-dictionary, numbered cards, words). All students succeeded in solving the problem in the plane of actions and the questions were aimed to obtain accurate descriptions of what they did and why it worked, as a first step towards conceptualization. Further, the students were encouraged to derive a general solution for the problem (detached from concrete cases) by means of teaching to a robot (played by the teacher) to do the task.

In the example used here, a bag containing an undetermined amount of numbered cards was given out to the students. These numbers were not necessarily consecutive and were not repeated. Students were asked to take cards from the bag, one by one, and to order them in an upward sequence on the table. A set of questions was elaborated for the interviews which were posed once students have solved the problem in the plane of actions. Throughout the interview, it was possible to pose new questions depending on the answers provided by the students. The interviews pursued three goals:
to conduct a process in which the students reflected about how they solved the problem. By means of reflective abstraction their actions were transformed into actions-in-the-plane-of-thought (concepts). This process was the source of knowledge of the repeating part of the algorithm. For the case of sorting the cards the actions were: pick up a card, compare numbers, insert the card in the right place, repeat the actions, finish.

– to apply Piaget’s ideas about the role of “searching for the reasons of success” in the conceptualization: the constant motor driving the subject to complete or to replace the observables of facts, by deductive or operative inferences is the search of reasons for the obtained result (Piaget, 1964, 1978b). We applied this principle by making the students comprehend that the reasons of success lie both in their actions and in the modifications of the objects. For the case of sorting the cards, in each repetition, a card was inserted in a partially sorted row and the number of cards in the bag decreased (until the bag was empty). This process was the source of knowledge of the base case, the invariants of the algorithm and its relationship with data structures.

– to help students to go from particular cases towards a general algorithm. We found that introducing automatization is of great help because the students have to strive to give general descriptions to a robot (played by the teacher) which otherwise does not understand the instructions. A set of primitive operations was given and the students had to design a list of instructions to make the robot do the task.

In the following a summary of results from students’ interviews for the case of sorting the cards is presented.

Towards to know how

The first descriptions of the students about how they sorted the cards clearly demonstrate that their thought was in the periphery (P in the diagram of the general law of cognition in Section 2), in other words, they were concentrated on the result: asked about how they did, they answered what they did (“I sorted the cards in increasing order …”). The goal of the questions was that students explicitly mentioned the actions composing the method, as accurately as possible. For instance, almost all students said something like “I picked up from the bag with all the cards on the table …”, that actually corresponded to the case in which the picked card was greater than all the cards in the row. Otherwise, they compared just to find the right place for the picked card2. One of the goals of the questions was to help the students become better able to describe accurately how they managed to insert a card into a partially ordered row, realizing that they compared only until a certain result of the comparison occurred.

Towards to know why

Further questions were related to the search of reasons for success: on the one hand, the existence of a base case (or several) (in this example, the bag became empty) and on the other hand, the invariant (or several), (in this example, all partial rows were sorted). The questions posed to students were like the following "Is it always possible to construct a row in this way, in a finite time? In other words, do we always finish the task with a ordered row on the table? Why?"

According to the theory the answers can be classified as:

– the reasons lie with the objects: "... because they are numbers" or "... because of the order of numbers",
– the reasons lie with the actions: "... because of my systematic actions" or "... because I do always the same".

To make students realize that the reasons for success lie both in their actions and in their modifications of objects, these kinds of question were posed:

– for the first type of answer:
  "If you are asked to sort cards with letters in alphabetic order, should you change your method? What about sorting objects of different sizes?"
– for the second type of answer:
  "Imagine you are asked to take one card at a time from the bag, and to set them on the table, one after the other. Would you have a sorted row? Observe that what you are doing is also systematic."

Although students answered correctly, none of them gave a satisfactory explanation of the existence of a base case as the reason for success. In previous work (da Rosa, 2010, 2007, 2005) we found, as well as other authors (Haberman & Averbuch, 2002; Velazquez, 2000), that it is significantly difficult to comprehend the base case (or base cases). To help students with this difficulty in an effective way, they themselves have to experience the need for the existence of a base case (or more). To do that, we asked the students to use another method by which the bag was never emptied: "Take from the bag one card at a time, and write down the numbers of the cards you took - in upwards sequence - on a piece of paper. Toss the cards back into the bag. Do you think that you will be able to finish by using this method?"

This strategy was effective: after using this method, all the students immediately became aware of the sequence of states of the bag, each time getting smaller until it became an empty bag, as a reason for success.

2 This is the source of a common programming mistake in which students use a for loop to access an element of a structure in cases where a while loop is more adequate.
Observe that by means of reflective abstraction the students reconstructed what they do in the plane of actions in the plane of thought, where the relationships become enriched by the comprehension of conditions and motivations (how and why). To finish, each student was asked to write down both the problem and the algorithm, step by step. Taking those descriptions as a starting point, they were asked to teach a robot to do the task, as described in the following section.

**A general sorting algorithm: the role of automatization**

In (Piaget & coll., 1963) Benjamin Matalon published a chapter entitled *Recherches sur le nombre quelconque*, in which he analyzed the relationship between the generic element concept and the reasoning by induction, which requires a proof that $P(n) \rightarrow P(n+1)$ for a generic number $n$ and a given property $P$. Matalon worked with the structure of natural numbers, stating that it is necessary to abstract away all the particular properties of $n$, except the property of it being a number, that is, an element belonging to the series of natural numbers.

Matalon addressed the problem of making the leap from particular cases to general ones and introduced variables for their reference. For example, he explained that Fermat made his arithmetic demonstrations using a particular number, but treating it as a generic number, for example, the number 17. If none of the specific properties of the number 17 were involved in the demonstration, then the demonstration could be considered valid for all numbers. Matalon added that in geometry, when a property is to be proven and the statement is "given a generic triangle" a particular triangle is drawn, avoiding right triangles, equilateral triangles or isosceles triangles, and not involving any particular properties of the triangle in the demonstration of the property. Among other things, Matalon concluded that to construct the concept of the "generic" element, it would be necessary to perform a generic action, that is, the repeated action to build a generic element. We applied these results interpreting the generic action that Matalon mentioned as the automatic version of an algorithm, that is to say, a program.

In this section we describe the process of going from a correct description of an instance of the method, to a general algorithm as the first step of program construction. Here is an example of a starting description: "To order the numbers that are in the bag on the table, I did the following: First, I took a card and placed it on the table, then I took another card from the bag. If the second card I took is of a higher value than the card on the table, I placed it on the right side of the card already on the table, while if it is of a lower value, I placed it on the left side. Then I continued the process in the same way, I took more cards and ordered them with the cards that were already ordered on the table as reference. For example, if I have these numbers: -2, 0, 5 and 8, and I pick number 1, I place it between 0 and 5, since 1 is less than 5 and greater than 0." Starting from that we asked the students to give oral instructions to a robot, played by a teacher, who tried to construct the ordered row by following the instructions. Our goal was to confirm the role of automatization to help students to detach themselves from particular cases, according to our interpretation of Matalon’s results.

The students read the instructions and the robot acted, until the sentence *Then I continued the process* ... which the robot was not able to follow. Further questions were posed to encourage the students to give more precise descriptions, until they came to a description similar to "What we are going to do is compare the card we want to insert with each card on the row, starting from the first card. When we find a card of higher value, then we insert our card before it. We do that until there are no more cards in the bag". That instruction the robot was able to perform (the robot was assumed to know how to compare the numbers of the cards).

The type of knowledge briefly described above is involved in the construction of concepts before formal knowledge, that is, no formalism intervenes (except for natural language). The introduction of a formal language was done in a group class as described in the next section.

### 3.2 Working groups: thematized knowledge or formalization

The main goal of this section is to describe our investigation about the process of formalizing constructed knowledge. We interpret this as the means to put into correspondence mental constructions (concepts) with some universal system of symbols or formal language.

Often, formal definitions or descriptions are presented to the students without taking into consideration their non-formal knowledge, that is, with no connection with what students already know about the subject (da Rosa, 2004). By contrast, in our approach the formalism introduced by the teacher is considered as a new object that students need to interact with in a process governed by the general law of cognition. As pointed out by Piaget (Piaget & García, 1980), the process of transiting the stages of the triad (intra-inter-trans) is of a dialectic nature, that is, the construction of formalized knowledge traverses its own stages. That means that an interaction between the students and formal representation of the objects (in our example rows, bag, cards) and their methods (inserting, comparing, deciding) has to take place. Our starting point is what students have said (and written) to teach a robot: "What we are going to do is compare the card we want to insert with each card on the row, starting from the first card. When we find a card of higher value, then we insert our card before it. We do that until there are no more cards in the bag". Our goal is that the students succeed in transforming that description into an algorithm using given pseudo code and primitives.

We adopted a notation suitable for expressing rows as lists, similar to that used in functional programming (empty row as [ ], non-empty row as first:tail (where first is the first element of the list, tail is the rest (a list) and
is the constructor function for lists), or with elements between brackets separated by commas (S. Thompson, 1999)). From the interviews we have learnt that the description of the action to place a card in a given non-empty row is relevant. Consequently, we decided to work on this part of the algorithm first. The main point here is that students understand that if we call this action insert t in row then if the value of t is not lower than the value of the first card in the row, then the first card remains the same and with t and the tail it is necessary to carry out the same action, that is to say, the result is first : insert t in tail. In the case that the picked card is greater than all the cards in the row, it is placed at the end, which means that it is inserted in an empty row (the tail). The students are asked to fill some tables as in Table 2 below. Observe that in this way, understanding the repetition of actions and the new objects in each repetition is straightforward.

### Table 2. Inserting a card in a non-empty row

<table>
<thead>
<tr>
<th>row</th>
<th>picked card</th>
<th>comparison</th>
<th>first of current insertion</th>
<th>tail</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-2,0,1,4]</td>
<td>3</td>
<td>(-2 &lt; 3)</td>
<td>(-2)</td>
<td>([0,1,4])</td>
<td>(-2 : \text{(insert 3 in } [0,1,4]])</td>
</tr>
<tr>
<td>0 &lt; 3</td>
<td>0</td>
<td>(0 &lt; 3)</td>
<td>(0)</td>
<td>([1,4])</td>
<td>(-2 : 0 : \text{(insert 3 in } [1,4]})</td>
</tr>
<tr>
<td>1 &lt; 3</td>
<td>1</td>
<td>(1 &lt; 3)</td>
<td>(1)</td>
<td>([4])</td>
<td>(-2 : 0 : 1 : \text{(insert 3 in } [4]})</td>
</tr>
<tr>
<td>4 &gt; 3</td>
<td>3</td>
<td>(4 &gt; 3)</td>
<td>(3)</td>
<td>([4])</td>
<td>(-2 : 0 : 1 : 3 : 4 : [])</td>
</tr>
</tbody>
</table>

Students were given a list of primitive actions that the robot (played by one of the students of the group) understands. In this first step towards formalization, the repetition of actions is modeled by "go to" instructions:

- decision: if something do a else do b (a and b are actions)
- compare cards: \(t < \text{first}\), \(\text{first} < t\)
- insert card \(t\) in a row: \(t: [\text{ }]\) or \(t:\text{first}:\text{tail}\) or first: (insert \(t\) in \(\text{tail}\))
- update (a variable becomes a new object): \(\text{row} \leftarrow \text{tail}\)
- sequence of actions: \{action1, action2, ..., actionn\}
- go to an action of the sequence: go to \(n\) (\(n\) is a natural number)
- \(a = b\) (the robot is able to verify if an object \(a\) is equal to an object \(b\))

The students were allowed to go back to making the actions with the bag and the cards as many times as they needed. They attempted several lists of instructions, until the robot student succeeded, in approximately 15 minutes. The following primitives were added to the list and students are asked to modify and complete the instructions for the robot to be able to order the cards from the bag on the table:

- pick a card from the bag
- decide if the bag is empty
- finalize

In 40 minutes approximately, all groups produced a list similar to the following:

0 pick a card \(t\) from the bag
1 if \(\text{row} = \[]\) then \(\{t: [\text{}], \text{go to 2}\}\)
   else if \(t < \text{first}\) then \(\{\text{row} \leftarrow t:\text{first}:\text{tail}, \text{go to 2}\}\)
      else first : \(\{\text{row} \leftarrow \text{tail}, \text{go to 1}\}\)
2 if the bag is empty then finalize
   else go to 0

The final stage of the study included introducing an interactive sentence in the manner of Pascal. We handed out to each group of students, a collection of small pieces of paper with 'if then else' and 'while' sentences, with suggested indentation in order to guide them. We explained the semantics of the sentences and the references to objects in each iteration (the rows on the table are \(H\) and \(H1\), and the picked card and first are \(T\) and \(T1\) respectively).

The interesting point to observe here is that students had to be able to construct another description of the same algorithm (the "while" sentences instead of the "go to" description). This means that the students needed to find a solution to a situation that had some similarities and differences with the one that they had already solved, by means of the cognitive instrument of generalization, both phases of which Piaget described (Piaget, 1978a): inductive generalization and constructive generalization. In the first phase, the individual transfers to new objects what has been previously constructed, without taking into account the transformations of the knowledge required for the conditions of the new situation. In this case, students tried to use the relation between cards (\(t < \text{first}\)) and the termination condition (if the bag is empty) of the "go to" version in the "while" version. (Observe that in the "while" version, these are "\(T\) is greater than \(T1\)" (that is, \(t > \text{first}\)) and "the bag is not empty" respectively).
Further instances in which the robot-student tried to follow the while version of the algorithm, resulted in correct positions for each piece of paper, as shown in Figure 1 below (in Spanish, an English version is included in Appendix A). That means that, because of constructive generalization, students understood the new conditions giving rise to structures of the trans stage, thus opening the possibility of studying new elements, such as other sorting algorithms and more formal representations, as well. This constitutes the focus for further work.

![Fig. 1. A version of the insertion sort algorithm](image)

## 4 Related Work

Although the amount of research in Computer Science Education (CSE) has grown significantly, in many countries the didactics of informatics is not considered as part of Computer Science (CS), including Uruguay where our work develops. The training of teachers of CS, for example, emphasizes content issues (which must be treated seriously) and neglects issues of Pedagogical Content Knowledge (Saeli, 2012; Hubwieser, 2013) of the concepts of CS. Few teachers of CS can answer with solid theoretical foundations, questions as to what, for whom, why and how to teach CS. Mathematics education is in this sense a model not only for having introduced the notion of specific didactics, that is, education of a discipline is part of that discipline, but also for the importance which it gives to the theoretical foundation in research. We share the concern of some authors (Holmboe et al., 2001; Zendler & Spannagel, 2008; Winsløw, 2005) about the need to conduct a significant amount of educational research in order for CSE to accomplish the same level as mathematics education.

Related works can be analyzed from several points of views, some of which we have included in previous work (for instance: learning of algorithms by novices, relationship between algorithms and data structures, recursion and induction, discrete mathematics and programming). Here we include some related work that shares Piaget’s theory as a theoretical framework with us. On the one hand, the question of how students learn is deeply investigated in the didactics of mathematics by mathematic academics, giving rise to theories and methodologies based on Piaget’s work (Dubinsky, 1996; Sánchez-Matamoros, García, & Llinares, 2006; Clark, 1997; Sadovsky, 2000). Among these, the Theory of Situations (Sadovsky, 2000) is closely related to our work as explained in Section 4.1. On the other hand, Section 4.2 includes recently developed neo-Piagetian work, research that considers advances in neuroscience and influences constructivist theories. Finally, we add a few words about related work referring to the content of our research (basic algorithms and data structures).

### 4.1 Brousseau’s theory and our work

Brousseau’s theory of didactical situations is one of the ways of thinking about mathematics as a subject for teaching. The theoretical bases are described in the following quotation (taken from the English translation of Brousseau’s book “Theory of didactical situations in mathematics” (Brousseau, 1997) in (Winsløw, 2005)): 
"Mathematicians don’t communicate their results in the form in which they discover them; they reorganize them, they give them as general a form as possible. Mathematicians perform a "didactical practice" which consists of putting knowledge into a communicable, decontextualized, depersonalized, detemporalized form. The teacher first undertakes the opposite action; a recontextualization and a repersonalization of knowledge. She looks for situations which can give meaning to the knowledge to be taught. But when the student has responded to the proposed situation (...) she will, with the assistance of the teacher, have to redepersonalize and redecontextualize the knowledge which she has produced so that she can see that it has a universal character, and that it is re-usable cultural knowledge."

In the 60s, when Brousseau was studying mathematics, he studied cognitive psychology with one of Jean Piaget’s collaborators, Pierre Greco. Not surprisingly, Brousseau applies the central hypothesis of genetic epistemology by Jean Piaget as a model framework for the production of knowledge. He argues that mathematical knowledge is constituted essentially from recognizing, addressing and resolving problems, which are generated in turn by other problems.

The constructivist concept leads Brousseau to postulate that individuals learn as a result of their adaptation to an environment that is a factor of contradictions, difficulties and imbalances. He suggests a model to teach and learn based on two basic types of interactions:

a) the interaction of the student with a problem that offers resistance and feedback on the mathematical knowledge used, and, b) the interaction of the teacher with students, in terms of their interaction with mathematical problems.

From this model, he postulates that an "environment" is required, with a didactical purpose: he defines an addidactical situation as a situation in which the student is allowed to use some knowledge to solve a problem (the environment) "without appealing to didactical reasoning [and] in the absence of any intentional direction [from the teacher]" (Sadovsky, 2000). The concept of an environment includes an initial mathematical problem which the individual copes with, and a set of mathematical relations, which are modified in the course of the subject’s actions during his production of knowledge, thus transforming his cognitive reality. The interaction of the teacher with the student is related to the student’s interaction with the environment and it is called didactical situation by Brousseau.

Brousseau’s model has points in common with ours: on the one hand, addidactical situations correspond in our model to instances in which the teacher does not intervene and students interact with an instance of a problem (Brousseau’s environment) or discuss in groups. The didactical situations, on the other hand, correspond to the instructional instances, where the interventions of the teacher are based on the knowledge constructed by the students through their interaction with the environment (the problem).

A significant difference arises, however. While Brousseau’s model is used in classrooms to the learning of mathematics, we have not had enough institutional support to do the same. One of the most important consequences of that is the difficulty of training the teachers.

4.2 Neo-Piagetian theory and our work

Several teaching investigations regarding basic algorithms have been performed within the frameworks of constructivism and mental model studies (E.George, 2000; Tamar, Dalit, & Paz, 1999; Ben-Ari, 2001). The central matters of these studies are implicitly related to tenets of the epistemological theory of Jean Piaget. Recent studies following neo-Piagetian theory consider the advances in neuroscience and whether these can influence constructivist learning theories. Neo-Piagetian theory is deemed as a suitable theoretical framework for some authors (Lister, 2011; Murphy, 2012; Falkner, Falkner, & Vivian, 2013; Gluga, 2012). Neo-Piagetian theory of cognitive development is a branch of cognitive psychology. It is influenced by the information-processing paradigm, the idea of linguistic theorists regarding the specific domain of cognition as well as the advances in neuroscience. Although several authors focus on different considerations, encompassed under the term "neo-Piagetian theories", they all have a common starting point, which are Piaget’s ideas from his earliest studies regarding children’s progress in cognitive development by stages (genetic psychology). Neo-Piagetian theory extends and complements the studies referred to. Genetic and cognitive psychology have almost always been related. Close collaborators of Piaget have continued and extended research on genetic psychology, reinforcing its bonds with cognitive psychology (Inhelder & Cellérier, 1992).

On the other hand, sustained by historical investigations, Piaget and García improved some aspects of Piaget’s theory (García, 2000, 1997; Piaget & García, 1980). They synthesized results from psycho-genetic studies and science history. Our work follows these results, including the application of a general mechanism, from the analysis of objects (intra-stage or instrumental knowledge) to the analysis of relations or transformations between objects (inter-stage or conceptual knowledge). This leads to the analysis of the construction of structures (trans-stage or thematized knowledge), a characteristic of the development of scientific theories and school subjects. In (Sánchez-Matamoros et al., 2006) the authors present a similar study in mathematics education, for the concept of the derivative of a function.
Finally, some words in relation to the content of our work (basic algorithms and structures). In (Zendler, 2013; Zendler & Spannagel, 2008; Zendler, McClung, & Klaudt, 2012) the authors indicate as examples of content concepts central to CS, problem and algorithm among others, and as central process concepts, categorizing and classifying. In our approach we consider that the source of knowledge for these concepts are basic actions that the students are familiar with, such as defining correspondences, sorting, searching and comparing. We explain the process by which students are able to transform this knowledge into conceptual and formal (thematized) knowledge, as well as the means by which teachers support this task.

5 Summary and further work

Piaget’s epistemological theory explains the construction of knowledge giving detailed descriptions of the processes, mechanisms and cognitive instruments intervening. The theory is suitable for most of scientific domains and all levels of knowledge. In this paper we present an application of that theory to investigate the construction of Computer Science concepts by students entering University or in the last years of High School. We have built this application over several years of study of learning different basic algorithms and corresponding data structures. Our approach can be taken as a model for doing research in Computer Science Education and also for designing instructional guidelines.

The questions of our research are related to ways of helping students to use their cognitive resources to attain higher levels of knowledge. One of the most important theoretical ideas about knowledge construction is the categorization of different stages, called by Piaget and Garcia in "Psychogenesis and the History of Sciences" as the triad intra-inter-trans. Accordingly, our research on the construction of concepts focuses on how students transform their instrumental knowledge (intra stage) into conceptualized knowledge (inter stage) and how this becomes an academic formalization (trans stage). The latter involves on the one hand, the construction of correspondences between concepts and expressions of some formal language as a universal system of symbols and on the other hand, the application of knowledge by students to solve new problems with differences and similarities to those already solved. Describing this process has been the goal of this paper, while the construction of knowledge before any formalization (except the natural language) is only briefly presented here as it has been described in previous papers, including excerpts from interviews.

At all stages of our research, results described by Piaget and collaborators in their work are indicated. The main points on which our strategies were effective are:

- students succeed in conceptualizing their know-how, by means of reflecting about how they solve a problem and why the solution works, especially when they themselves experience the need for a base case (or several). The evidence is given by students’ correct descriptions in natural language of their algorithmic solutions and of the reasons for their success (Section 3.1)
- how to surmount one of the main difficulties, that is, that students’ thought detaches from particular cases and makes the leap to general solutions, by means of introducing automatization (Section 3.1).
- introducing a formal language as a new object about which the students have to construct knowledge using the same principles as any other object. That means starting the interaction in the plane of action, allowing students to work with particular cases of formal expressions to facilitate the transition to a general algorithm in the formal language (Section 3.2).
- to face students with problems that encourage them to apply their knowledge taking into account necessary transformations. This is a first step to the construction of structures of the trans stage and the beginning of the comprehension of algorithms as classes of methods (sorting, searching, counting, etc) (Section 3.2 and further work).

Three lines of research are relevant as further work: on the one hand, to elaborate pedagogical proposals and didactic guidelines based on our model, which can be useful to teachers in supporting their researches and practices, and a way of getting institutional support as well. On the other hand, to study the construction of elements of the trans stage, that is to say, of classes of algorithms and formal representations. This includes for instance, other sorting algorithms, that could be compared with each other, and recursive or iterative implementations. Finally to complement this research, we need to investigate the historical evolution of the concepts of iteration, induction and recursion, that are the higher forms of thematized (formalized) knowledge of ways of solving problems by repeating actions. According to the methodological analysis of Piaget’s theory, a critical historical analysis of those concepts teaches about the construction of knowledge and can therefore cast light on the learning process.

6 Acknowledgements

I thank Ben du Boulay for correcting the English. The comments of the referees are gratefully acknowledged.
References


Peyton Jones, S. (2013b). Computing at school in the UK: from guerrilla to gorilla. *Under review by CACM (v4).*


A English version of Figure 1

```plaintext
let B be the bag
while B is not empty do
  take a card T from B
  let H be the row on the table
  if H is empty then
    put T in H
  else
    let T1 be the first card of H
    H1 be the tail of H
    while T is greater than T1 do
      if row H1 is empty then
        place T in H1
      else
        let T1 be the first card of H1
        H1 be the tail of H1
        place T in front T1 in H
```
The impact of syntax colouring on program comprehension

Advait Sarkar
Computer Laboratory, University of Cambridge
Cambridge, UK
advait.sarkar@cl.cam.ac.uk

Abstract. We present an empirical study investigating the effect of syntax highlighting on program comprehension and its interaction with programming experience. Quantitative data was captured from 10 human subjects using an eye tracker during a controlled, randomised, within-subjects study. We observe that syntax highlighting significantly improves task completion time, and that this effect becomes weaker with an increase in programming experience.

Keywords: POP-II.B. Program Comprehension; POP-III.D. Editors; POP-V.B. Eye-tracking

1 Introduction

Syntax colouring, commonly known as syntax highlighting, is a feature of some text editors which colours lexical tokens in source code text according to a certain categorisation. An example of how source code may look with and without syntax highlighting can be found in Figure 1.

Fig. 1. Left: code without highlighting. Right: same code with syntax highlighting.

Syntax highlighting is included in text editors on the premise that it makes working with code easier. It is easier to spot certain kinds of syntactic bug with syntax colouring: for instance, if string literals are assigned a unique colour, then an unclosed string is easy to spot as all subsequent text until the next delimiter is coloured. It is important to note that syntax highlighting is purely secondary notation (T. R. Green, 1989) meant for humans; it does not affect the behaviour of the code.

Close attention has not yet been paid to whether syntax highlighting has any impact on the speed, precision or ease with which a programmer can internalise a program’s semantics, i.e comprehend the program. A plausible intuition is that the improved readability and clearer structure of syntax-highlighted code would make it “easier” to understand the behaviour of a program, but the nature and degree of any such improvement has not previously been investigated.

In this study we establish whether source code syntax colouring has an effect on program comprehension time, and furthermore investigate whether this effect varies by programming experience. It is conceivable that a difference in comprehension speed might be attributable to differences in eye fixation patterns on syntax-highlighted keywords, on the basis that visual attention (focus on a particular location) triggers mental processes to comprehend or solve a given task (Just & Carpenter, 1980). We conduct an eye-tracking study, explained in §2, to substantiate this. These research objectives are formalised as hypotheses in §3. Finally, we discuss our findings in §4.
1.1 Previous work

Most previous work on colour in text has focused on English prose and not source code. Evidence suggests that colouring tokens in text has no effect on the speed of visual search when the colour of the target is unknown, but when the colour of the target is known, search times are considerably smaller (B. F. Green & Anderson, 1956; Rubin, 1988). Colour has been found to be useful for emphasising format and categorisation (Van Nes, 1986; Rubin, 1988). By contrast to the corresponding research in print media, Hakala et al. found that syntax highlighting did not have a significant impact on the speed of visual search (2006) onscreen.

Baecker (1988) studied aspects of typography with the specific purpose of enhancing source code readability and comprehensibility. The author prepared short programs, varying several aspects of the layout and typography, such as punctuation marks, kerning, and incorporation of colour. The prepared program along with a questionnaire was presented to the participants of the study. The impact of the improved layout on readability, comprehensibility, and recall was measured as a function of the number of questions answered correctly and the time taken to answer the questions. It was found that the improved layout increased the mean number of correct answers by 11%, and that program readability was improved by 25%.

Bednarik and Tukiainen (2006) demonstrated the feasibility of using eye-tracking data to study program comprehension, later using their methods to study the evolution of debugging strategies (Bednarik & Tukiainen, 2008). Sharif and Maletic (2010) used eye tracking to study the impact of identifier naming conventions (e.g. “camelCase” and “under_score”) on visual search. Participants visually searched for a known identifier amongst a word cloud of similar identifiers. Fixation counts and durations were used to measure “visual effort.” The underscore style was found to require lower matching times and visual effort than the camel case style. A summary of other work on eye-tracking in programming is given by Busjahn et al. (2014).

Gilmore and Green (1988) conducted a study wherein bugs were introduced into program fragments containing simple loop and conditional structures. Some bugs were surface level, like misspellings, and some were at a deeper level, like putting an assignment in the wrong place. A highlighter was used in combination with indentation to highlight control structures, cognitive-plan structures, both, or neither (a ‘cognitive plan’ relates to the programmer’s mental model for the algorithmic structure of the program (Spohrer, Soloway, & Pope, 1985)). The participants were to find the bugs. Participants found surface bugs faster with control-structure cues, and deep bugs faster with cognitive-plan colouring. The authors conclude, importantly, that the added information is only effective if it relates to the task. Although the colouring was not syntax-based, this relates to our final conclusion that highlighting improves comprehension speed.

2 Experimental methodology

2.1 Comprehension task design

The comprehension task took the form of a mental execution, where the participants were given a function definition and requested to compute its output for a given set of arguments.

An example task is presented below:

```python
What is the output of da([1,2],[5,6],3)?
def da(list1, list2, x):
    list3 = []
    for j in list1:
        for k in list2:
            list3.append(j+k)
    for e in list3:
        if (e%x) == 0: print e
```
To facilitate a within-subject comparison, each participant performed the task with pairs of programs, where one of the pair had syntax highlighting, and the other did not. Tasks were carefully designed to have a difficulty comparable to their paired counterparts by choosing programs that were of identical structure but with certain specific variations that had a major impact on their semantics. These variations included reversing the direction of an inequality, replacing additions with subtractions, or interchanging the arguments. Thus, tasks in a pair had comparable difficulty but very little transferable knowledge, effectively minimising order effects.

The participants were given 3 pairs of comprehension tasks. From each pair, the task to be highlighted was randomly chosen to reduce the likelihood of highlighting and difficulty becoming artificially correlated. We ensured ample difference between each pair of tasks, and randomised the order of tasks for each participant to account for repeated testing and order effects.

Since we intended to investigate the impact of programming experience, we chose to present the tasks in a language in which the available participants, graduate computer science students, had varying levels of experience. Python was the appropriate choice for two reasons:

1. The graduate students had a broad range of undergraduate backgrounds, so much variation in Python experience could be expected: some participants had been taught extensive courses, whereas some had never encountered Python prior to the experiment.
2. It is similar to other popular languages such as Java, C++, and even pseudocode, so a specific lack of Python experience would not make the task impossible to complete.

Python differs in two ways from languages with which the participants are more likely to be familiar: its use of indentation to indicate block structure, and its lack of explicit variable types. Participants were made aware of both of these issues before they began the task. To avoid datatype-related confusion, a uniform variable naming scheme was adopted in the tasks. For example, integers were named \texttt{x}, \texttt{y}, etc. and lists were named \texttt{list1}, \texttt{list2}, etc.

A final consideration is the colouring scheme of the syntax highlighting. It was not within the scope of this study to investigate the impact of different colouring schemes, although it would make for interesting future work. We focus on token-level colouring, as opposed to block-level colouring such as that proposed by Cigas (1990), since token-level colouring is far more prevalent in practice. We used the Solarized\textsuperscript{1} palette, which has been designed for usability.

2.2 Eye-tracking apparatus

We used a Tobii\textsuperscript{2} X120 eye tracker, a video-based remote eye tracker that captures eye movements using twin infrared cameras. No head gear is necessary. The tracker is placed on the desk directly underneath the monitor. For optimal tracking, the tracker is configured to be aware of its location with respect to the screen and of the screen’s dimensions and resolution. Sampling occurs at a temporal resolution of 60Hz with a latency of 25-35ms and accuracy of the order of 0.5°. The eye tracker compensates for head movement to some extent and is able to recover if the subject temporarily looks away from the screen. The eye gaze data includes timestamps of fixations, onscreen fixation coordinates, fixation durations, eye positions, pupil size, etc.

2.3 Measuring programming experience

The participants were asked to complete a questionnaire recording the following:

1. How long the participant has been programming.
2. A self-reported score from 1-10, 10 being “highly experienced”.

\textsuperscript{1} Solarized - Ethan Schoonover, \url{http://ethanschoonover.com/solarized} (as of June 15, 2015)
\textsuperscript{2} Tobii Eye-Tracking Research, \url{http://www.tobii.com/en/eye-tracking-research/} (as of June 15, 2015)
3. The participant’s self-estimated peer group decile (e.g. top 10%, top 20%, etc.)
4. For each language the participant has used (up to a maximum of 4):
   – How long the participant has been using that language.
   – How often the participant uses it.
   – The largest or most complex program the participant has written in it.
   – A self-reported proficiency score from 1-10, 10 being highly proficient.

Since programming experience is complex and multifaceted, it is impossible to measure along an objective, interval scale. Thus, these records were used purely to establish an ordinal ranking of participants by experience. The answers to questions 1-3 were used to perform a pairwise comparison of participants’ experience, and the answers to the question 4 provided enough information to adjust any unrealistic answers provided for the first 3 questions.

2.4 Experimental procedure

The experimental procedure was explained to the participant, including a brief explanation of the block structure and type inference in Python. The participant was seated approximately 70cm in front of the screen and eye tracker. The participant read and signed an informed consent form. Next, the eye tracker was calibrated using a nine-point calibration lasting approximately 45 seconds, during which the participant focused their eyes on nine points that appear on an equally spaced 3×3 onscreen grid in random order.

To begin with, the participant completed two example tasks, one with highlighting and one without. No measurements from these tasks were included in subsequent analysis. This allowed the participant to become familiar with the nature of the task and also ask questions if necessary. Once the participant understood the task, the actual study commenced. The participant stepped through each of 3 task pairs, 6 tasks in all, in random order. The tasks were presented in a slideshow for instant transitions between tasks. Timing and eye-tracking for each task began when the slideshow switched to the task slide, and ended when the participant identified the correct answer by saying the result of the requested computation aloud. Participants were requested to complete the tasks mentally without the aid of pen and paper, but were allowed to vocalise their thinking process. The experiment had an average duration of 13 minutes, never exceeding 20. Finally, participants completed the programming experience self-assessment questionnaire.

We gathered data from 10 graduate computer science students at the University of Cambridge. For each participant we recorded an average of ∼1300 fixation events. The eye-tracker recorded particularly poor fixation data for 3 participants; this was attributable to the fact that these participants wore glasses which confused the eye tracker. Their data was excluded from analyses of visual effort.

3 Results

In this section we report the results of the study previously outlined. The primary variables being studied are presented in Table 1. In §3.2 we introduce two derived variables which are not included in this table. We use the Shapiro-Wilk test to establish normality. We use the Wilcoxon signed rank test (WSRT) for paired nonparametric comparisons.

---

3 We made the anecdotal observation that experienced programmers tended to underestimate their competency, whereas less experienced programmers tended to gauge their ability more accurately. Evidence, perhaps, that the Dunning-Kruger effect applies to programming experience (Kruger & Dunning, 1999).
Table 1. Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highlighting</td>
<td>Whether the task was highlighted or plain. An independent variable.</td>
</tr>
<tr>
<td>Experience</td>
<td>Each of our 10 participants was assigned a rank from 1-10 with 10 being most experienced. An independent, ordinal variable.</td>
</tr>
<tr>
<td>Task completion time</td>
<td>The time it takes a participant to complete a task (in seconds).</td>
</tr>
<tr>
<td>Fixation count</td>
<td>The number of times a participant fixates on the task image while completing it. A dependent, interval variable.</td>
</tr>
<tr>
<td>Fixation duration</td>
<td>How long an individual fixation lasts (in microseconds).</td>
</tr>
<tr>
<td>Prompt fixations</td>
<td>The task image was divided into two so-called “areas of interest”: the prompt, which was the single line containing the instruction for the task, and the content, which was the code for the task. This variable counts a participant’s fixations on the prompt during a task.</td>
</tr>
<tr>
<td>Context switches</td>
<td>The number of times a participant fixates on an area of interest that is different from the area that was the subject of the immediately previous fixation. For example, if the participant fixates on the prompt, then on the content, and then on the prompt again, the number of switches is 2.</td>
</tr>
</tbody>
</table>

3.1 Effects of highlighting

The task completion times for highlighted as well as plain tasks were not normally distributed. Task completion times for highlighted versions of the tasks were significantly lower (WSRT: \( T = 136, p = 0.047 \)). The difference in medians is 8.4s. This is illustrated in Fig. 2.

We investigated the effect of highlighting on some features of the eye-tracking data, namely fixation durations, fixation count, fixations on prompt and context switches. These can be collectively thought of as the “visual effort” required to perform the comprehension task. We did not find a significant effect of highlighting on fixation counts, fixation durations or prompt fixations.

Recall that a context switch occurs when a participant fixates on an area of interest that is different from the area that was the subject of the immediately previous fixation. Highlighted code incurs significantly fewer context switches than non-highlighted code (WSRT: \( T = 13.5, p = 0.045 \)). The difference in medians is 23 context switches. This is illustrated in Fig. 3.

3.2 Effect of programming experience

As the data was not normally distributed, a 2-way ANOVA could not be used to investigate the interaction of experience with highlighting on task times. We instead introduce a new derived variable, time advantage, which is simply the ratio of the task completion time in the non-highlighted task to the completion time for its highlighted counterpart. Thus, if a participant completed the plain version of a task in 60s, and the highlighted counterpart in 30s, the time advantage for that task instance is 60s/30s = 2. We consider this to embody the effect of highlighting on task completion time. We then investigated the correlation of experience with this variable. On the raw data we observe a significant Spearman correlation of -0.37 (\( p = 0.044 \)) but a weaker Pearson correlation (\( r = -0.29, p = 0.12 \)), suggesting a nonlinear relationship. When log-normalised, the time advantage variable has a significant Pearson correlation with experience (\( r = -0.39, p = 0.033 \)). We conducted a similar study with context switches for the derived variable switch advantage, but did not find a significant correlation.
Fig. 2. Bar graph comparing task completion times for a highlighted task with its plain counterpart. Each pair of bars is an instance of a participant completing a particular task pair.

Fig. 3. Bar graph comparing context switches for a highlighted task with its plain counterpart. Each pair of bars is an instance of a participant completing a particular task pair. Fewer instances were available for this comparison since data from participants wearing glasses had to be excluded from analyses of visual effort.
4 Discussion

Our results confirm the intuitive but heretofore unsubstantiated idea that syntax highlighting in source code improves program comprehension. The median difference in task completion time was 8.4s in favour of highlighting. It is possible that the magnitude of this effect increases with the length of the program; an interesting area for future investigation.

We were unable to detect differences in fixation count and durations due to highlighting. Since 3 of our 10 participants were excluded from analyses of visual effort as their glasses rendered the eye-tracking data unusable, it is possible that statistical significance was not achieved as a consequence of the reduced sample size. It is also plausible that syntax highlighting genuinely does not affect fixation durations or count. As shown by the observation that syntax highlighting has an effect on context switches, the fixation sequence might be altered in a far more sophisticated manner.

Our inference that the presence of highlighting significantly reduces the number of context switches is of particular interest. Context switches during the experimental tasks almost always took the same form: while reading the content, the participant glanced at the prompt before continuing with the content. This sequence of 2 switches always served a single purpose: to remind the participant of the argument values. Our results show that the need to be reminded of the input values was significantly greater when the code was not highlighted.

There is no immediately clear explanation for this, but we suggest a simple one here: it is plausible that the mental overhead required to process and understand plain code is greater than the mental overhead required to process highlighted code, since highlighted code contains additional semantic richness by virtue of the colours of the tokens. This additional overhead in plain code causes other items, such as the values of the input arguments, to be displaced from the working memory (Baddeley & Hitch, 1974) of the participant. This would account for the increase in context switches in non-highlighted code.

The theory that there is less overhead involved in processing code with additional visual cues (such as those provided by syntax colouring) is qualitatively supported by our results, as well as quantitatively by the aforementioned study by Gilmore and Green (1988). The data from some participants suggests that syntax highlighting allows the programmer to focus on a
smaller region of code, as illustrated in the fixation heat map in Figure 5. Furthermore, while the quality of the eye-tracking data was not generally good enough to designate single keywords as areas of interest, the data from certain participants with very good readings suggests that syntax highlighting allows the programmer to even ignore some keywords entirely, as illustrated in the gaze plot in Figure 6. This theory is also supported by our inability to detect a relationship between experience and the effect of highlighting on context switches. If highlighting does affect context switches by the mechanism hypothesised above, there is no reason to expect it to vary by programming experience. Future work may study this effect in greater detail, providing better quantitative evidence.

![Fig. 5. Left: fixation heat map for code without highlighting. Right: heat map for the highlighted counterpart.](image)

Finally, we found that programming experience was negatively correlated with time advantage. It appears that syntax highlighting improves program comprehension speed to a greater extent in novice programmers than in experienced programmers. However, this may be a consequence of the brevity of the tasks; repeating the study with longer programs may reveal that experienced programmers stand to gain just as much as less experienced programmers.

5 Conclusions

We investigated the effect of syntax highlighting on program comprehension and its interaction with programming experience using eye-tracking data captured from 10 participants. Each participant was requested to mentally compute the output of a Python function for a given set of arguments, repeating this several times with highlighted and non-highlighted code.

The presence of syntax highlighting significantly reduces task completion time, but the magnitude of this effect decreases as programming experience increases. We did not detect an effect of the presence of highlighting on the durations of individual fixations, nor did we detect an effect on the total number of fixations required to complete the task.
The presence of syntax highlighting significantly reduces context switches. We hypothesise that syntax highlighting improves the ability of the programmer to mentally retain the state of the execution, and that highlighted code incurs a lower mental comprehension overhead. In some cases, the eye-tracking data suggested that the participants were able to ignore highlighted keywords entirely, as though perceiving them peripherally was enough to incorporate their semantics into the computation. Future work may investigate this effect, as well as the effect on longer programs, with more quantitative rigour.

6 Acknowledgements

Many thanks to Graham Titmus for help with hardware; the Rainbow research group\(^4\) for providing the eye tracker and laboratory space; Alan Blackwell for various discussions on the topic; Thomas Green for his considerate, detailed and excellent feedback on the written drafts; and all the participants for their valuable time and effort.

References


Busjahn, T., Schulte, C., Sharif, B., Begel, A., Hansen, M., Bednarik, R., ... others (2014). Eye tracking in computing education. In *Proceedings of the tenth annual conference on international computing education research* (pp. 3–10).


Abstract. We present an empirical study investigating the role of syntax highlighting in Sonic Pi, considering both program writing and program debugging tasks. Data were collected from 10 participants, who were asked to execute writing and debugging tasks, while their screens were recorded. We observed how syntax highlighting significantly improves task completion time in the case of writing as well as of debugging tasks. In particular we investigated debugging of a common mistake in the Sonic Pi environment, which we called colon-space feature because of the underlying Ruby syntax. Moreover we observed a correlation between syntax mistakes with programming experience and musical experience. If the task is highlighted the task completion time decreases with the increasing of programming experience, while this decreasing trend is not so significant in the case of an increasing musical experience.

Keywords: POP-II B. Coding; POP-II B. Debugging; POP-III D. Sonic Pi; POP-III C. Syntax Highlighting; POP-III D. Editors;

1 Introduction

Syntax highlighting is a feature of some text editors, that colour lexical elements in the source code, according to some semantic categorisations. It is used in source code text editors due to its capability of making the coding action easier. Moreover it should be easy to find certain types of code mistakes with syntax colouring. It is important to point out how syntax highlighting represents just a secondary notation, and has no effect on the behaviour of the code itself (Sarkar, 2015). The purpose of my study is to evaluate the impact of Syntax Highlighting in coding within the Sonic Pi environment, a new open source software tool and platform for the Rasperry Pi computer (Aaron, Blackwell, & Burnard, in review) and now also freely available for Windows and Mac distributions, designed for encouraging the learning of computing and music within schools (Sonic Pi Website, 2015). An example of how the Sonic Pi source code looks differently, with and without syntax highlighting, can be found in Figure 2 and Figure 1.
The main goal of Sonic Pi project was to create a new DSL (Domain Specific Language), having a double goal in terms of providing an engaging musical experience for students, and teaching them the basic elements of programming. The project was developed for a target class of 12 years old students and for 5 one-hour lessons, starting from teaching them what is a computer, up to coding (Aaron & Blackwell, 2013).

Sonic Pi was implemented as an embedded Ruby DSL language (Aaron & Blackwell, 2013). To make an example of Sonic Pi code, to trigger a new sound and then create a pause of a certain duration, before the next sound, the commands are:

```
play 60
sleep 0.5
play 61
```

The command in the first line will play the MIDI note 60 on the synthesiser used at that particular time, then a pause (whose duration is defined by the numerical number associated to sleep, in this case 0.5, that means half second) and then again another note.

To allow children to be able to code with Sonic Pi music language, a specific IDE was designed. The IDE is characterized by a scheme consisting of only 5 elements (Control buttons, Workspace tabs, Editor pane, Information pane, Error pane) and represents a simplified interface, suitable for children and non-expert teachers (Aaron & Blackwell, 2013).

No previous studies focused on the role of syntax highlighting in terms of speed and readability of the code in the Sonic Pi environment. We measured the time needed to complete an assigned musical-writing task in Sonic Pi and the time needed to debug a code in Sonic Pi. Speed in live coding has in fact a double meaning, as explained in (Zmölnig & Eckel, 2007). On the one hand it is interesting to know how fast a programmer/composer can write codes, since it is a way to measure its productivity in terms of line of codes produced in a certain amount of time. On the other hand, often writing code fast is a matter of virtuosity, as it is playing fast for a music performer. Live coding, in fact, quite naturally refers to some notion of performance, and speed is a performance factor which should surely be taken into account in its analysis.

We describe the experimental setting in Section 2. The research objectives are written as hypothesis in Section 3 and our findings described in Section 4.

### 1.1 Background

In the psychology literature there are many works focusing on the role of colouring tokens in prose texts. Van Nes (Van Nes, 1986) described the importance of colours, layouts and typography on display screens in order to create texts with the highest possible legibility.

Hakala (Hakala, Nykyri, & Sajaniemi, 2006) presented an experiment using three different colouring schemes for intermediate programmers, asked to look for local patterns in Java programs. The differences among colouring schemes proposed were not statistically significant, and the results showed how the classical used control scheme (black text on white background colour) exhibited the same performance in searching as the other control schemes.

Baecker (Baecker, 1988) developed a number of new techniques and tools to improve program source texts and program documentation. In particular he proposed a new effective presentation of source text in C programming language, using high quality digital typography and a processor implementing the design.

The main empirical study on syntax highlighting, which can be considered as a starting point for our work, is (Sarkar, 2015).

Sarkar (Sarkar, 2015) analysed the impact of syntax highlighting on program comprehension and the relation with programming experience of the subject. His studies were validated using eye-tracking data coming from 10 participants. The task submitted to each participant was to compute a Python function given a set of arguments. The python function was given several times with both highlighted and non-highlighted code. The results of the research study showed how the code with syntax highlighting reduced the task completion time, even if this effect decreased when participants were experienced programmers. Moreover it was shown how syntax highlighting improved the ability of the programmer to mentally retain the state of the execution, suggesting how a highlighted code brings to a lower mental comprehension effort (Sarkar, 2015).

### 2 Experimental Methodology

As participants we recruited 10 students of the University of Cambridge. The target students we looked at had to have some interest in music and programming. The variance of music and programming experience among the participants was quite high, from students who used programming everyday to students who just used programming for some courses. The same for the musical experience, some students had high musical experience, while others were listening to music as a hobby. The participants were recruited through the Department of Computer Science and college network.

The experiment procedure was composed by the following phases described in the following sections.
2.1 Tutorial Phase

The participants were given the Sonic Pi tutorial and a worksheet with some Sonic Pi examples to read. The worksheet was chosen from the Sonic Pi website, since it is a well built summary of the main topics of the tutorial and presented examples that could be useful to the participants to solve the tasks proposed. The maximum amount of time they could spend in reading the tutorial was 45 minutes. They were allowed to ask questions in case they needed further explanations. We decided that the tutorial was the best way to teach participants about Sonic Pi, to avoid the instructor bias while they were learning Sonic Pi basic commands. The tutorial was given to them in paper format, and they were not allowed to experiment the commands on the Sonic Pi software. This decision was taken because otherwise they would have tried either on the highlighted or non-highlighted version, and outcomes could have been biased by this. The printed tutorial had the same highlighting format as the one in Sonic Pi. Yet, most participants noticed how the highlighting of the syntax in the tutorial does not correspond to how the text is highlighted in the Sonic Pi text editor. This may be something that could be useful to change in the Sonic Pi tutorial, to make the learning process easier by having corresponding colours in the tutorial and the text editor. The tutorial and the worksheet were available to them also during the execution of the tasks, so they didn’t need to memorize instructions.

2.2 Writing Task Phase

After the tutorial phase participants were asked to complete the writing task. To allow a within-subject comparison, each participant was asked to write two pieces of code having comparable difficulties, where one of the codes was highlighted and the other plain. The two programs were different and submitted to them sequentially. The same program could not be used for both cases since once a task is performed, it may be considerably easier to repeat the experiment with syntax highlighting (or non-highlighted text editor), just because the code is already known. Moreover the participants were divided in two groups (Group A and Group B). Group A was asked to write the first piece of code in the highlighted environment and the second in the non-highlighted one, while Group B was asked to complete the tasks in the reverse order. This scheme was adopted to prevent results being affected by order effects.

We designed the instructions for the task pairs carefully, describing step by step the piece of code requested, submitting two tasks of comparable difficulties, yet different. These differences included the use of different coding structures (blocks in one and threads in the other, for example) having as a consequence little transferable knowledge between one and the other, in so doing minimising the order effects. The two pieces of code asked to write were two famous children songs (Brother John and Happy Birthday). The participants were given the instructions, and at the end they were asked to run them and try to recognize which song they had composed. This element of fun was introduced to let the participant enjoy even further the tasks requested and the musical aspect of coding with Sonic Pi. See Appendix for the task instructions.

2.3 Debugging Task Phase

Sonic Pi music language was built on the Ruby syntax, and many Ruby syntax conventions have been inherited. In particular the aspect analysed in our experiments is the role that the highlighted syntax code plays in understanding where to introduce a space after the colon in the code. This kind of colon-space feature is typical of Ruby. This peculiarity of Ruby regarding colon and spaces seems to be an issue also among Ruby users, as some coding forums report (Coding Forum 1, n.d.), (Coding Forum 2, n.d.), (Coding Forum 3, n.d.).

Consider the example below:

```
play 48, amp: 0.5
sample :ambi_piano
```

First of all, as we can see from the sampled two lines, all the numerical values in the Sonic Pi highlighting are blue, while all the strings are pink, and also the colon sign is pink.

In the first line the colon has to be followed by a space, because it means that we are assigning a certain value (0.5) to the amp (amplitude of a key) parameter. On the other hand, in the second line we have that the space is located before the colon, because we are instantiating a particular type of sample, which is ambi_piano in this case, and not assigning a specific value to a certain parameter. There are various considerations to be made on the way the code is highlighted, and how it works, that I have noticed while using Sonic Pi:

1. If in the first line we don’t insert a space, between the variable name and the value we want to assign, the code still works but the syntax highlighting varies. In particular it becomes:

```
amp 0.5
```
2. If in the first line we insert a space, between the name of the variable and the value we want to assign, then the code doesn’t work and the syntax highlighting of the code becomes:

\[ \text{amp} = 0.5 \]

3. If in the second line we don’t insert the space before the colon, the code still works correctly even if the difference is that no auto-completion possibility is given to the user.

\[ \text{sample:ambi\_choir} \]

4. If in the second line we insert the space after the colon, then the code doesn’t work, and the syntax highlighting becomes the following:

\[ \text{sample: ambi\_choir} \]

To evaluate the role of syntax highlighting, in the colon-space scheme, participants were asked to debug a code. All the mistakes (6 in total) were related to the colon-space scheme sparse within the code, and presenting all the possible configurations of the mistakes so that they were not easily recognizable by the participants. Not all the colon-space schemes included in the code were wrong, because otherwise the task would become too easy and it wouldn’t reflect what happens in reality.

As previously described in the writing phase participants were asked to debug pairs of codes: one highlighted and the other non-highlighted. Moreover, as in the previous phase participants were divided in two groups and all the considerations regarding preventing order effects and the comparable difficulties of the two pieces of codes designed can be repeated. See the Appendix for the task instructions.

### 2.4 Questionnaire

At the end of the experiments all the participants were asked to complete a questionnaire to understand their programming skills as well as their music knowledge and experience. In particular following (Sarkar, 2015) the questionnaire included specific questions regarding the musical and programming experience.

1. Do you have any programming experience?
2. If yes which language are you mostly familiar with?
3. How long have you been using that particular language?
4. How often have you used this?
5. Which is the largest or most complex program you have written with this?
6. Self-report a score form 1-10 assessing your experience, with 10 being highly experienced

1. Do you have any kind of musical experience?
2. Do you play any instruments?
3. If yes which one?
4. Self-report a score from 1-10 assessing your musical experience with 10 being highly experienced in music.

These questions have been used in order to understand the impact of programming and musical experience in the performances of the various participants. In particular, they were helpful to rank participants by experience, since evaluating programming and music experience is extremely difficult to achieve, and the self-questionnaire is one of the better methods to build such ranking, even if also this can be subject to self-bias (over/under-estimation of individual programming-music capacities). That is why question 6 for the programming and question 4 for the musical experience was used to make a pairwise comparison between participants, while the other questions provided enough information to adjust unrealistic answers.

### 2.5 Procedure

We conducted the study in an isolated environment, with no distractions, so that all the participants were able to focus on the tasks proposed. To screen, and voice recording the experiments, we used Camtasia Studio (Camtasia Studio, n.d.) that allows to record the screen and present a studio software permitting to watch frame by frame the video recorded. This allowed exact evaluation of the timings needed to complete the tasks.

The experimental procedure was explained to the participants through an introductory speech. They were informed that all the data collected were going to be anonymised and they were asked to agree on the recording of the screen, and on the voice recording present in the room.

The tutorial phase timing never reached 45 minutes, and all the participants completed their reading within the time allowed (on average 20 minutes). The instructions of the tasks were submitted sequentially and 5 minutes were given to each participant before the beginning of each task, in order to read the instructions.

The timing data collected were all considered from the starting of the typing till when the participant typed the last command on the text editor. Participants were allowed to vocalize their mental processes, or to use paper and pen in case they needed to write some notes while doing the task.
2.6 Variables and Hypotheses

The variables studied are presented in Table 1. Then for evaluating the programming experience with respect to the task completion times two more derived variables were introduced, that are not included in this table. Based on our research questions, we formulated the 4 null hypothesis in Table 2. The first 2 relate to task completion times, while the last two relate to the relation with programming and musical experience. The hypotheses just assume that the distribution is not the same between the highlighted and non-highlighted cases. The significance level adopted throughout the analysis is \( \alpha = 0.05 \) and any result reported as significant have a \( p \) value below this. The test used for establishing normality is the Shapiro-Wilk test.

<table>
<thead>
<tr>
<th>Highlighting</th>
<th>Independent variable, binary and categorical, that is associated with the presence or not of highlighted text editor (HL, NHL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Completion Time</td>
<td>A dependent, continuous variable, indicating the amount of time needed to a participant to complete a task (writing or debugging). The time is expressed in the format min.seconds</td>
</tr>
<tr>
<td>Programming Experience</td>
<td>An independent ordinal variable. This variable assigns a value between 1 and 10 to the programming experience of the participant, with 10 being most experienced</td>
</tr>
<tr>
<td>Musical Experience</td>
<td>An independent ordinal variable. This variable assigns a value between 1 and 10 to the programming experience of the participant, with 10 being most experienced</td>
</tr>
</tbody>
</table>

The null hypotheses are presented in Table 2.

3 Results

3.1 Results of Highlighting on the writing task completion time

The task completion times for highlighted as well as non-highlighted writing task with Sonic Pi were normally distributed. Therefore we performed a paired t test. The resulted \( p \) value shows how the highlighted version of the tasks were significantly faster than the non-highlighted ones. Consequently we reject \( H_{10} \).

Fig. 3: Boxplots comparing Task 1 highlighted and non-highlighted instances
Table 2: Null Hypotheses

<table>
<thead>
<tr>
<th>Null Hypothesis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{10}$</td>
<td>There is no significance difference, in the writing task with Sonic Pi, of the completion time, between code with highlighting and code without.</td>
</tr>
<tr>
<td>$H_{20}$</td>
<td>There is no significance difference in the debugging task in Sonic Pi completion time, between code with highlighting and code without.</td>
</tr>
<tr>
<td>$H_{30}$</td>
<td>The effect of highlighting on task completion time of the writing task is not related to programming experience.</td>
</tr>
<tr>
<td>$H_{40}$</td>
<td>The effect of highlighting on task completion time of the writing task is not related to musical experience.</td>
</tr>
<tr>
<td>$H_{50}$</td>
<td>The effect of highlighting on task completion time of the debugging task is not related to programming experience.</td>
</tr>
<tr>
<td>$H_{60}$</td>
<td>The effect of highlighting on task completion time of the debugging task is not related to musical experience.</td>
</tr>
</tbody>
</table>

Fig. 4: Bar Graph comparing task completion times for a highlighted writing task with its non-highlighted counterpart

3.2 Results of Highlighting on the debugging task completion time

The task completion times for highlighted as well as non-highlighted debugging task with Sonic Pi were not normally distributed. Therefore the comparison was made using the Wilcoxon signed-rank test which shows that the highlighted version of the tasks were significantly faster than the non-highlighted version. Consequently we reject $H_{20}$. 
Fig. 5: Bar Graph comparing task completion times for a highlighted writing task with its non-highlighted counterpart

Fig. 6: Boxplots comparing debugging task highlighted and non-highlighted

3.3 Effect of Programming and Musical Experience

To evaluate the effect of programming and musical experience in task completion times performances we introduced a new variable: time advantage (Sarkar, 2015). This variable is the ratio of the task completion time in the non-highlighted and in the highlighted task. For example if a participant completed the non-highlighted version of the task in 60 sec and the highlighted in 30 sec then the time advantage variable is 60/30=2. We investigated the correlation of both programming and musical experience with this variable. For the programming experience the time advantage variable for the writing task is distributed normally. Therefore we performed a Pearson correlation test, obtaining a strong correlation value of -0.67 and a p value of 0.0308. Therefore we reject $H_{30}$. 
Fig. 7: Each dot is a task instance completed by a participant. The regression line has slope -0.06716.

Regarding the debugging task, the Pearson correlation test give a weak negative correlation of -0.2 and the p value shows no significance difference, therefore we cannot reject \( H_{50} \).

Regarding the musical experience the Pearson test with the writing task shows a positive correlation, but the p value shows no significant difference, therefore we cannot reject \( H_{40} \). The same happens for the debugging task therefore we cannot reject \( H_{60} \). It is worth noticing the following. The programming experience plays a fundamental role in decreasing the completion time of the non-highlighted writing task, while the music experience has no effect in the completion time of the non-highlighted task. Qualitatively we could say that musical experience does not affect the speed up performances when the task is non-highlighted, but the programming experience is important to speed up performances when the task is non-highlighted.

Fig. 8: Task 1 non-highlighted and Musical Experience
4 Discussion

Our results confirm the fact that syntax highlighting improves writing speed and debugging speed of Sonic Pi code. Possibly these effects might increase with the increasing length of the code, and this could be investigated in further research studies. Moreover it is interesting to notice the strong correlation between the programming experience and the impact on the highlighting. A further consideration might be interesting. All the participants were able to complete the task and to let the code run. However some mistakes were still present in the code, and this is due to the fact that also with some syntactic mistakes the code still works, even if the auto completion of the text editor is not available anymore. We suggest that these little bugs maybe should be fixed in a future release of Sonic Pi, also to improve the usability for children learning coding. Moreover another suggestion would be to introduce in the tutorial the same highlighting format that is in Sonic Pi text editor. Finally, it is interesting to add that we executed the experiment also with a 11th participant, a 13 years old child, having interests both in Computer Science and music. The performances reported are on average in line with the ones reported by the 10 University of Cambridge students, and since Sonic Pi is primarily thought for children, this finding seems to confirm the effectiveness of the Sonic Pi approach to coding and music.

5 Conclusion

We investigated the effect, in terms of speed, of syntax highlighting in Sonic Pi, for code writing and debugging. Data were collected from 10 participants. Each participant was requested to write two tasks, in the non-highlighted and in the highlighted version, and to debug a Sonic Pi code in the highlighted and non-highlighted version. The presence of Syntax highlighting significantly reduced the task completion times. Moreover we observed that the task completion times reduces as the experience in terms of programming of the participant increases, possibly showing how the programming experience has a major impact in enhancing the difference between a highlighted and non-highlighted code. Moreover, though much less significant, also music experience has an effect in terms of reduction of completion times in the two tasks of writing and debugging code in Sonic Pi. Future works could investigate these effects in longer programs, or using different types of highlighting with Sonic Pi. Moreover it could be interesting to increase the number of participants to the experiments to investigate further the results obtained. Finally we could focus the data analysis also on other aspects, such as the relationship between musical experience and programming experience.

6 Acknowledgement

Many thanks to Dr. Alan Blackwell, Mariana Mărăşeiu and Hadil Charafeddine for the interesting and important discussions on the topic. Many thanks to all the participants for their valuable time and effort.
References


A Appendix 1-Additional Paper Guideline

WRITING TASK 1

You will have 5 minutes to read the instructions before start writing the task. Feel free to look at the tutorial and the worksheet whenever you want, either while you are reading the instructions and while you are completing the task. Ready for the instructions? Here we go!

You will create the first four bars of a famous children song. In the end you will try to play it and see if you recognize it. If you are able to do it, it means that you followed the instructions in the right way and composed your song with Sonic Pi!

Follow the instructions step by step:
1) Use the synth called fm.
2) Use two blocks of 2.times loop which you have seen during the tutorial phase. Lets call this two blocks A and B
3) In block A you will have to play the notes 60,62,64,60 (in this order)
4) In block B you will have to play the notes 64,65,67 (in this order)
5) Define the parameters for each note:
6) In block A the notes 60,62,64,60 will have a release equal to 1
7) In Block B the notes 64 and 65 will have a release equal to 1 and 67 equal to 2
8) Choose the amplitude for each note in block A
9) Choose the amplitude for each note in block B
10) Insert a pause after each note (also after the last one in each block).
11) All the pauses will have the same length except for the last one in the second block that will have double duration with respect to the others.

You are done!

Now try to Run Sonic Pi.
Can you recognize to which famous children song this first four bars belong to?
Let me know when you have finished the task

SOLUTION TO THE TASK:
Brother John
Welcome to Sonic Pi v
use_synth fm
2.times do
  play 60, amp: 0.6, release: 1
  sleep 0.5
  play 62, amp: 0.8, release: 1
  sleep 0.5
  play 64, amp: 0.3, release: 1
  sleep 0.5
  play 60, amp: 0.9, release: 1
  sleep 0.5
end

2.times do
  play 64, amp: 0.2, release: 1
  sleep 0.5
  play 65, amp: 0.4, release: 1
  sleep 0.5
  play 67, amp: 0.3, release: 2
  sleep 1
end
DEBUGGING TASK 2

Debug this piece of code. There are some syntactic mistakes. Find them and correct them. You have a maximum of 10 minutes to complete the task.

If you finish before the time given, let me know!

CODE WITH MISTAKES (the mistakes in the code here are highlighted)

```plaintext
use_synth fm
2.times do
  play 60, release:0.5, amp: 4, pan: -1
  sleep 0.5
  play 67, release: 0.3, amp: 2, pan: 1
  sleep 0.5
end
use_synth :saw
play 38, release: 0.1, amp: 3
sleep 0.25
play 50, pan: -1
sleep 0.25
use_synth :prophet
play 57, amp: 4, release:0.3
sleep 0.25
```
Abstract

Code completion is a widely used feature of modern integrated development environments. This study examines the ways in which code completion is used by professional software developers, as well as their actions when code completion doesn’t offer the expected results. We observe that code completion is used with the intention of writing code faster and more correctly, that a large fraction of the code completions are not accepted by the users and that users often used code completion as a debugging tool: when the suggestions are not useful or not expected they are seen as a indicator that there is an error in the program.

Keywords: POP-II.A. Professional programmer, POP-II.B. Coding, POP-III.B. Dart, POP-III.D. Editors, POP-V.B. Protocol analysis

1. Introduction

Autocomplete for source code, also known as code completion, is an important part of modern integrated development environments (IDEs). Aimed at helping programmers write code faster, a code completion system offers a menu with a list of possible suggestions, usually alphabetically ordered. From the given list, the user can either choose one of the options, or they can ignore the list and resume typing. The completion list appears automatically when the user types a trigger character (such as ‘.’ or ‘(’ or ‘::’, depending on the programming language and IDE), or it can be invoked with a specific keyboard binding. Intellisense1 from Microsoft Visual Studio and Content assist2 from Eclipse are two code completion system examples found in their respective IDEs.

A study of the Eclipse IDE observed that programmers used code completion as often as Copy & Paste (Murphy et al. 2006). This finding supports the general perception that code completion is a feature heavily used by programmers. We suggest that studying how professional software developers use code completion in their coding practices is a good starting point for future research on improvements of code completion systems and interfaces.

There are two common strategies for populating the list of possible completions: lexical and ‘semantic’. Lexical models of completion look for other code starting with the same prefix as code the user is completing. Semantic models use a model of the grammar of the language to constrain the list of suggestions to those that are legal at the point of edit, e.g. by filtering out private methods. In this project we have studied a tool that provides semantic completions.

We studied the code completion usage patterns of 6 professional software developers, with each session organized in two parts. In the first part, the participants worked on a programming task for 30 minutes. The second part consisted of a retrospective think aloud session with each participant, where the recording of the first part was played back and the participant was asked to discuss what they were thinking or trying to do for each interaction with the code completion window. We report on the patterns of interaction we observed from analysing this data.

Our objective was to explore the following research questions:

1. What are the intentions of programmers when using code completion?
2. What are the behaviours that programmers engage in with respect to code completion?
3. What are the ways in which the interaction with the code completion popup can fail?
4. What recovery actions do programmers take when faced with such disruptions?

2. Related work

Murphy et al.’s (2006) study on how Java programmers are using the Eclipse IDE reports that all their participants used code completion and that its usage is at a similar level to Copy & Paste. However, they give no detail as to how and why code completion is used. In recognition of the importance of code completion within a programmer’s environment, there has been a fair amount of research into trying to make autocomplete systems faster and more “intelligent”. The informal motivation for such research is to reduce the cognitive load of the programmer who has to search for the appropriate completion of the method call or variable name. These systems focus on moving the most relevant suggestions at the top of the menu (i.e. terms that have the highest probability of being the ones that the programmer wants to use), thus reducing the need for actively searching or filtering the list.

Robbes and Lanza (2008) discuss ways in which the program history can help in providing the user with more relevant completions. Bruch et al. (2009) use data mining of multiple code repositories to provide more relevant method call suggestions. Similarly, Calcite (Mooty et al. 2010) offers suggestions for class instantiations extracted from other existing code, but it also allows for user edits of the completion list. Another type of enhancement is offered by Seahawk (Bacchelli et al. 2012) and Blueprint (Brandt et al. 2010), who address the problem of finding source code snippets that answer a user’s query within the development environment.

The systems above focused on improving code completion with the goal of enhancing programmer performance (i.e. increased speed when typing source code). However, the purpose of using code completion seems to be more complex than writing faster code. Church et al. (2010) observe that code completion can be seen as a source of real-time feedback, both when it offers suggestions and when it is failing to do so. They suggest that the programmers are interpreting the failures of the code completion system as a sign that there are errors in their code.

Another perspective is to consider code completion as a form of negotiated interruption. As defined by the taxonomy of interruptions (McFarlane 1997, McFarlane & Latorella 2002), a negotiated interruption allows the user to decide when to deal with the interruption. The case of code completion has some resemblance to negotiated interruption systems, because the user can choose whether or not to interact with it (simply continuing to type), and if they do, how and when (returning to the code completion list by retriggering it).

Other previous literature suggests that programmers use code completion for the exploration of APIs (Stylos & Clarke 2007) and for faster code writing (Bruch et al. 2009). Further, Ward et al. (2012) mention spelling and locating known items as two main themes in the use of autocomplete for a library website. We use these previous studies as the starting point for developing the categories for the intentions of programmers when interacting with the code completion window.

3. Experimental design

The criteria we used for selecting a programming environment were: an existing user population with experience in using the environment, support for code completion, and suitability of the environment for instrumentation. We selected Dart Editor (Dart Editor | Dart, n.d.), an IDE for the Dart programming language built on the Eclipse RCP platform. Dart Editor exposes an API that can be used to report events within the system relevant for UX experiments (Instrumentation | Dart, n.d.).
3.1 Participants

We studied 6 professional software developers. Two of them had no previous experience with Dart, whilst the other four write Dart as part of their work. Of the 4 professional software developers experienced with Dart, three of them had no previous experience with the libraries used in the task. We report findings across the group as a whole, as the sample size is too small for any comparison between the levels of experience with the environment.

3.2 Data collection

We recorded the computer screen used by the participants throughout the study, and also recorded spoken audio. During their day-to-day development many Google engineers writing Dart use a custom, internal-only version of Dart Editor that implements the same instrumentation API, recording the data for later analysis. Whilst the publicly available versions of the Dart Editor do not use this API, we built an additional plugin for this study that implemented it and logged to disk the events such as accepting or closing the completion popup.

This gives us the opportunity to compare the data that has been recorded in this controlled study to a larger corpus of instrumentation data collected in a typical professional context.

3.3 Procedure

There were two phases to the study, each of approximately 30 minutes duration.

In the first phase, the participant was asked to perform a programming task. The task was selected to require use of several libraries, including at least one library that had to be used in depth. We expected that this would result in a set of interaction behaviours that represent a typical aspect of the day-to-day development work done by professional software developers.

The task was chosen so that it requires the participants to only write Dart code, without any need for working with HTML and CSS. This has several advantages. First, it offers us consistency when studying the code completion popup, as some of its features may be missing or be different for HTML and CSS. Second, because the participants do not need to switch contexts between files having different syntax, they will be able to focus more consistently on the aspects of the code that are supported by the semantic completion capabilities of Dart. We also avoid having the participants spend too much time on debugging activities and web designing. Whilst this decision limits the study to a specific part of the development cycle, it maximises the amount of code completion data that can be gathered within the experiment duration.

The required task was to implement the “ls” command for listing directory entries, including several of the following command line flags typically supported by implementations of “ls”:

- “-a” to include entries starting with “.”
- “-l” to specify a long listing format
- “-R” to list subdirectories recursively
- “-S” to sort the results by file size
- “-h” to print file sizes in human readable format rather than block numbers

Participants were asked to work at their normal pace, and were told that they were not expected to finish the entire task in the given time. They were asked to work on the parts of the task in any order. The participants were not asked to think aloud, although all of them chose to verbalize some of their thoughts as they were coding. They were informed that the purpose of the study was to observe the interaction with the tooling, but not specifically told that it was concerned with the code completion system.

The second part of the study consisted of a retrospective think aloud (RTA) session. The experimenter played back the screen recording that had been made during the previous task. Each time that the code completion popup appeared in the recording, the experimenter asked the participant what they had been thinking and doing at that time.
The retrospective think aloud protocol was first described and used by Hewett and Scott (1987) as a post-search review and is presented under various names in the literature: retrospective testing (Nielsen, 1994), retrospective report (Capra, 2002), think after (Branch 2000), retrospective verbal protocol and retrospective verbalisation (Kuusela & Paul, 2000). In a typical application, the participants first perform the assigned tasks in silence, followed by a verbalization of their thoughts after the completion of the task, usually supported by a video recording (Nielsen, 1994). A number of studies (Bowers & Snyder 1990, Van Den Haak et al. 2003, Nielsen 2002) have compared the results of concurrent and retrospective think aloud protocols. The general consensus of these studies is that there are few differences between the two approaches in terms of quality and quantity of data produced, but that the nature of the data gathered is somewhat different. For example, Bowers and Snyder (1990) report that concurrent think aloud results in more procedural verbalizations, whereas participants doing retrospective think aloud give more explanations. Further, it has been shown that retrospective think aloud is a valid method for finding out what the participants are thinking when solving an experimental task (Guan et al. 2006).

We used retrospective think aloud in order to address several problems of concurrent think aloud. First, programming is known to be a cognitively intensive task, and we expected interaction with the code completion window to be particularly demanding, involving search, exploratory understanding, and strategy formation. Thus we decided against using a concurrent think aloud protocol as we wanted to avoid disrupting the participants if they fell silent during these critical moments of the tasks. Using retrospective think aloud we were able to gather information about each interaction we were interested in without risking this disruption. Second, we wanted to collect instrumentation data that was as similar as possible to how professional software developers would use code completion in the wild. Hence we only informed the participants of our interest in the use of code completion at the beginning of the retrospective think aloud phase, in order to avoid sensitizing them in a way that might change their behaviour whilst coding.

Nevertheless, one of the reported problems of retrospective think aloud is that the participants may not remember what they were thinking. It was in order to avoid this, we showed participants the video of the previous task and prompted them to talk for each individual code completion. Another observed problem is that the participants can give different explanations for what they were thinking or doing during the retrospective think aloud. In order to account for these, during the analysis, we consider both the concurrent and retrospective vocalizations of the participants, as well as their observed behaviour from tool instrumentation.

4. Analysis method

4.1 Protocol analysis

The 6 sessions resulted in 192 instances of interaction with the code completion window. The concurrent and retrospective verbalizations for each interaction were transcribed into text files. The user interface actions of the participants were also transcribed using the screen recording from the first part of the study, together with the instrumentation logs. These 3 transcripts, for each of the 192 interaction instances, provided the corpus of data for coding.

This data was coded according to 4 aspects, which created our coding frame, as can be seen in Table 1. Each aspect corresponds to one of the research questions described in Section 1. Since each coded aspect describes a different aspect of interaction with the code completion window, a single interaction could potentially be assigned codes relating to multiple different aspects. Further, we observed that the users sometimes reported multiple intentions motivating their use of code completion, in addition to multiple behaviours during their actual interaction. We therefore allowed the assignment of multiple codes from the same aspect to each coded interaction.

After preprocessing of the data and selecting the main categories of the coding frame, we used both deductive and inductive qualitative content analysis in order to develop the set of codes (Mayring, 2000). The former was used to identify User Intentions, as our seed codes were derived from reasons for autocomplete usage that had been mentioned in previous literature: Exploration (Stylos & Clarke...
2007), Speed (Bruch et al. 2009), Correctness and Search (Ward et al. 2012). Because there were further themes in the data beyond those covered by these seed codes, we also used inductive analysis to generate new codes. For the remaining three aspects we used inductive qualitative content analysis, with the codes emerging from the raw data.

Table 1. Number of codes in the frame and assigned to the data per coded aspect

<table>
<thead>
<tr>
<th>Coded aspect</th>
<th># codes in the frame</th>
<th># codes used in the data</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Intention</td>
<td>7</td>
<td>286</td>
</tr>
<tr>
<td>Interaction Behaviour</td>
<td>8</td>
<td>247</td>
</tr>
<tr>
<td>System Disruption</td>
<td>6</td>
<td>69</td>
</tr>
<tr>
<td>Recovery Action</td>
<td>7</td>
<td>63</td>
</tr>
</tbody>
</table>

To confirm the reliability of the resulting coding frame, an additional researcher was recruited in order to independently code the qualitative data (transcripts, video and screen recordings) based on the developed coding frame. The researcher coded 48 of the interaction instances, representing 25% of the data. The interrater reliability calculation returned an average of 95.14% for percent agreement and a value of 0.81 for Krippendorff’s alpha. On Krippendorff’s (1980) scale, agreement $\alpha \geq 0.8$ is considered as reliable.

4.2 Pattern discovery

In order to be able to easily recognize patterns and co-occurrences of the codes in the data, we transformed it into a visual form. This new representation allows for visualising the entirety of the data at once, where each interaction with the code completion window is visualised as a glyph, containing elements that correspond to all the codes assigned to that interaction. We organize our glyphs in a tabular form, similar to the representation of charts in small multiples (Tufte, 1990) in order to allow for comparisons. Glyph representation has been also suggested by Gehlenborg and Brazma (2009) as a useful method for visualising and comparing between hundreds of samples.

Figure 1 presents an example of a glyph for a single coded interaction, where each colored square within the glyph represents one of the codes assigned to the interaction. The horizontal position and color of the square represent the aspect that is being coded, whilst the vertical position represents the index of that code within the coding scheme (see Tables 2-5). Figure 2 presents a subset of the data represented using the model from Figure 1. Using this visualization we were able to identify patterns and co-occurrences of codes by observing repeated instances of glyphs that were visually similar. These patterns are discussed in Section 5.

Figure 1. A glyph summarising the codes assigned to a single instance of interaction with the code completion window. Each colored square encodes a code (row) of the given aspect (column).
In order to compute the association between codes that were part of the patterns we identified visually, we used the normalized pointwise mutual information (npmi) statistic (Bouma 2009). A score of -1 between two codes means that they never occur together, a score of 0 represents independence and +1 represents complete co-occurrence in the data. The npmi statistic is used to verify the validity of the patterns observed through analysing the visualisation.

5. Results

5.1 What are the intentions of programmers when using code completion?

Table 2 presents the complete description of the set of codes used for coding programmer intentions when interacting with the code completion window, i.e. the User Intention category. The descriptions for each code are based on what we observed in our data during the pre-coding phase.

We would like to note the distinction that we drew between exploration and search behaviours. The UI-Exploration code represents undirected exploration, i.e. when the user doesn’t have a specific goal in mind. We considered directed exploration as a form of search, because the user was looking for something that would help them achieve a given goal. The merging between search and directed exploration in one code, UI-Search, is due to directed exploration behaviour and search behaviour being indistinguishable from each other in the data we gathered.

However, Stylos and Clarke mention that their participants used “code-completion as the primary means of exploration” (2007), more specifically, to gain confidence that the instantiated object was the correct one and to find out which “methods or properties perform the needed functions”. Since their participants had a goal that they wanted to achieve, we consider that this description fits better the category of search or even directed exploration, rather than (undirected) exploration.

Interestingly, one of the main findings so far is the lack of API exploration in the participants’ intentions. From the total 192 instances of interaction with the code completion window, only 3 of these were coded as UI-Exploration. We suggest that this is due to the participants being focused on working on the task and having a goal in mind that they tried to achieve through the use of code completion. As such, their interactions with unfamiliar APIs were mostly about searching for a method or attribute that would perform a desired action. However, during the Retrospective Think Aloud conversations, one of the participants mentioned the use of code completion for API exploration in his work:

“I definitely use autocomplete. To explore APIs. Because it's so... not a lot of times you have a good idea of what you want, especially if you're like, what getters do I have, right? it's gotta be helpful.”

ID3, Experienced with Dart
For another participant, the distinction between search and exploration was less clear:

*Here, you know, we have a list and I was trying to figure out what I can do with the list. I was trying to get one item, I guess, I don't remember exactly, but I was trying to see what I can do with the list. And it was showing me. At first I get the first one, yeah, and I said, yeah, probably I shouldn't use the first one, and after that that I said, maybe I should do a for loop.*

**ID4, Inexperienced with Dart**

<table>
<thead>
<tr>
<th>User Intention</th>
<th># instances</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UI-Ignored</td>
<td>29</td>
<td>The user doesn’t interact with the code completion window, they dismiss it as soon as it appears, or they interact with it, but don’t have any intention of using it. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Idle interaction (the user interacts with the code completion window whilst thinking about what to do next).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The user does the sequence of keystrokes that they would have done in the code editor, but these were grabbed by the code completion window</td>
</tr>
<tr>
<td>UI-Correctness and/or Speed</td>
<td>90</td>
<td>The user wants to ensure that the name of the method is correct (the method exists, getting capitalisation right). The user has typed the first part of the attribute or method name and uses code completion to fill in the rest</td>
</tr>
<tr>
<td>UI-Exploration</td>
<td>3</td>
<td>The user isn’t looking for something specific and doesn’t have a goal in mind, they’re just getting to know the API</td>
</tr>
<tr>
<td>UI-Search</td>
<td>68</td>
<td>The user knows what they want to achieve, but doesn’t exactly know how to get there. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The user doesn’t know the exact name of the method, but knows possible alternatives</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The user is looking for a specific property on an object, or a way to get it if it’s not an attribute of an object</td>
</tr>
<tr>
<td>UI-Test</td>
<td>9</td>
<td>The user is triggering code completion to see whether the tooling is working properly</td>
</tr>
<tr>
<td>UI-Documentation</td>
<td>4</td>
<td>The user is triggering code completion to read documentation</td>
</tr>
</tbody>
</table>

During the first step of the coding phase, we observed that it was very difficult to differentiate between when participants were using code completion to achieve speed, correctness or both. As such, we decided to combine the two seed codes into a single one, UI-Correctness and/or Speed. Further, as can be seen in Table 2, correctness and speed was one of the main intentions when using the code completion window. This confirms findings in previous literature – programmers use code completion in order to write less error prone code, whilst, at the same time, making the writing process easier and faster.

“So like, if the IDE fills in a method name for me, then I know that I didn't make a typo for example. So I have this many bugs less in my code, I'm sure. Right, because it doesn't make mistakes. So what I usually want to type in every method with autocomplete. Or like, most of the times.”

**ID1, Experienced with Dart**

We also observed that users use code completion to test whether their code is working, whether an import is behaving, or, in the case of software developers new to the language and the editor,
to simply check that code completion exists. This helps them check the state of the environment and the tooling.

“I realized I need to go back to Directory, and then went 'dot' to see what's happening. Cause this feature [code completion], I love it, you know. It's something that I would like to see like, you know, in all the editors. And I was assuming that it should be here.”

ID4, Inexperienced with Dart

5.2 What are the behaviours that programmers engage in with respect to code completion?

The behaviours that we have identified are presented in Table 3, as well as how many of the 192 interactions with the code completion window from the participants were labelled with each code.

We observe that a fairly typical way of interacting with the code completion is to filter the list of suggestions and then accept one of them, either with Enter or with a mouse click. In fact, 28.1% of all the interactions with the code completion window consisted of filtering and accepting a suggestion. The strength of co-occurrence between the IB-Filter and IB-Accept codes, as reported by the npmj statistic, is 0.39, which suggests a correlation between them.

Table 3. The codes for the Interaction Behaviour category, the number of instances coded and a description for each code.

<table>
<thead>
<tr>
<th>Interaction behaviour</th>
<th># instances</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IB-Scroll</td>
<td>35</td>
<td>The user goes through the list of code completions using mouse/trackpad scroll or arrow down/up.</td>
</tr>
<tr>
<td>IB-Filter</td>
<td>81</td>
<td>The user filters the code completion list by typing</td>
</tr>
<tr>
<td>IB-Unfilter</td>
<td>8</td>
<td>The user deletes a character at the end of the method name whilst code completion window is displayed, thus expanding the list of completions shown</td>
</tr>
<tr>
<td>IB-Accept</td>
<td>78</td>
<td>The user accepts a suggestion from the completion list</td>
</tr>
<tr>
<td>IB-Dismiss</td>
<td>20</td>
<td>The user actively dismisses the code completion window (with ESC or clicking somewhere else in the code editor)</td>
</tr>
<tr>
<td>IB-ManualTrigger</td>
<td>21</td>
<td>The user triggers code completion manually (either by deleting an existing ‘.’ or by using the keyboard shortcut)</td>
</tr>
<tr>
<td>IB-ReadDocumentation</td>
<td>3</td>
<td>The user reads the documentation for an attribute from the documentation popup of the code completion window</td>
</tr>
<tr>
<td>IB-InspectParameters</td>
<td>4</td>
<td>The user looks at the parameters or return types of a method to see whether the method is useful or if the parameters need default-value overriding</td>
</tr>
</tbody>
</table>

Further, the filter-accept behaviour also has a good correlation with the intention of using code completion for speed and correction (npmj score of 0.50). For example, one participant mentioned in the second part of the study, as they were watching their filter-and-accept behaviour:

"startsWith. I know what I'm looking for here. It helps me get capitalization correct as well, is the other, another primary reason for dealing with it."

ID5, Experienced with Dart

Scrolling through the list of suggestions was another behaviour that the participants often engaged in, and it has a good association with the intention of searching for something (npmj score of 0.39).
5.3 What are the ways in which the interaction with the code completion popup can fail?

We have identified a list of disruptions that can occur whilst using code completion, which are described in Table 4.

From the preliminary results, only 40.1% of all triggered code completion windows resulted in accepting a suggestion. By observing what the user was doing when they decided not to accept a completion we see that is partly due to the users not being familiar with some of the APIs used, and thus they engage more in searching for an item which then they don’t find in the code completion window. This goes some way to explain why SD-MissingSuggestion code accounts for 19% of all the disruptions recorded.

However, in a sample of 10,000 completions collected from instrumented usage of Dart within Google, this fraction only rises to 44.2%. This suggests that finding items may remain a common challenge even for developers experienced with the APIs.

Table 4. The codes for the System Disruption category, the number of instances coded and a description for each code.

<table>
<thead>
<tr>
<th>System disruption</th>
<th># instances</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD-MissingSuggestion</td>
<td>14</td>
<td>The wanted suggestion is missing</td>
</tr>
<tr>
<td>SD-NotExpected</td>
<td>15</td>
<td>The list of suggestions is unexpected or surprising to the user</td>
</tr>
<tr>
<td>SD-NoAppear</td>
<td>15</td>
<td>Completion window fails to appear</td>
</tr>
<tr>
<td>SD-Disappeared</td>
<td>17</td>
<td>Completion window disappeared undesired</td>
</tr>
<tr>
<td>SD-AccidentalDismiss</td>
<td>2</td>
<td>The user accidentally dismisses the code completion window</td>
</tr>
<tr>
<td>SD-FailedAccept</td>
<td>5</td>
<td>The user wanted to accept a completion but accidentally dismissed the code completion window</td>
</tr>
</tbody>
</table>

An important part of system disruptions is due to the user not finding the expected suggestions in the list (SD-NotExpected) and to the failures of the code completion window (SD-NoAppear and SD-Disappeared). We mentioned previously that code completion can be viewed as a negotiated interruption in the programmer’s workflow, albeit a very useful and often desired interruption. We argue that, in situations where the code completion engine fails to provide relevant suggestions, this interruption aspect is more evident: the failure is a signal that the code may not be correct, and the user can choose whether to try to find out what the problem is, or to continue typing without the aid of code completion. It should also be noted that this aspect of code completion is often overlooked when designing systems or improvements for code completion systems.

“Oh, and ‘fse’<sup>3</sup> doesn’t work here, which is interesting. I think it should. But the fact that it doesn't makes me actively go back. I would rather go back and fix the inference problem than carry on. Even though I'm right, just to increase the confidence value.”

ID5, Experienced with Dart

5.4 What recovery actions do programmers take when faced with such disruptions?

The possible recovery actions that we have identified during our studies are presented in Table 5. The recovery actions that the programmers engage in correspond to the way in which the system disrupted their coding activity. For example, if the suggestion they were looking for is missing, then the programmer goes to read the online documentation (npmi score of 0.53) or tries to change the code that they’re writing (npmi score of 0.33), as the absence of the wanted suggestion signals them that the object they have is not useful.

---

<sup>3</sup> [n.a.] An instance of FileSystemEntity, a type of object for manipulating files.
“’flags’ doesn’t seem to exist, so what do I get? Command, hashCode, name, so.. I’m actually not sure. So I’m gonna look at addFlag.” [goes to the online documentation]

ID3, Experienced with Dart

Table 5. The codes for the Recovery Actions category, the number of instances coded and a description for each code.

<table>
<thead>
<tr>
<th>Recovery action</th>
<th># instances</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA-Documentation</td>
<td>7</td>
<td>User goes to look for documentation</td>
</tr>
<tr>
<td>RA-ChangeCode</td>
<td>26</td>
<td>User decides that the code they were trying to write wouldn’t work and tries to write different code</td>
</tr>
<tr>
<td>RA-WrongProgram</td>
<td>6</td>
<td>User decides that some other code in their program is wrong and tries to fix it</td>
</tr>
<tr>
<td>RA-Override</td>
<td>8</td>
<td>User decides that code completion is misleading and continues regardless</td>
</tr>
<tr>
<td>RA-ResummonForAccept</td>
<td>1</td>
<td>Resummon completion to accept it</td>
</tr>
<tr>
<td>RA-ResummonForObserve</td>
<td>13</td>
<td>Resummon completion to look at what needs to be different</td>
</tr>
<tr>
<td>RA-ManualComplete</td>
<td>3</td>
<td>The user continues typing the name of the attribute, after accidentally dismissing the code completion window</td>
</tr>
</tbody>
</table>

We have also observed that if the list of suggestions is surprising or unexpected to the user, they use it as a signal that either the code they’re writing is wrong (npmi score of 0.31) or that some other code in their program is wrong (npmi score of 0.73).

“Again, here, I use code completion and it tells me that nothing works. And that tells me.. When I look at it it’s because I’m missing a close brace so there it is helping. The fact that it doesn’t work is telling me something’s wrong with the rest of my code.”

ID5, Experienced with Dart

For example, due to Dart’s optional typing, programmers can choose whether to add or not to add type annotations to their code, with the program having the same semantics either way. An interesting behaviour that we observed was that the participants actively added types in order to get better suggestions. When the list of completions is surprising to the user (e.g. when it shows a restricted list of attributes), then the programmer realizes that type inference failed and then they modify their code by adding types to some of their variables in order to trigger more relevant suggestions.

“The problem with this was that I didn't put in the type. So I was like, that's odd.”

ID2, Experienced with Dart

These behaviours of using the absence of information in the completion dialog as a prompt to change their program are made possible by the semantic (as opposed to lexical) completion model.

As expected, we observed that when a user accidentally fails to accept a suggestion, they engage in two main possible types of behaviour: they continue typing without needing help from a code completion window (npmi score of 0.87), or they manually trigger again code completion in order to accept the desired option from the list (npmi score of 0.69).

Similarly, when the code completion window disappears as the programmer was filtering or interacting with it, they will either continue regardless of the state of code completion (npmi score of 0.53) or they will try to re-trigger it again in order to observe what went wrong and what needs to be different (npmi score of 0.69).
6. Conclusions

Whilst limited in scope and sample size, this study provides some empirical evidence of how professional software developers use code completion in their work. We coded and described four aspects of their interaction with the code completion: intentions, behaviours, breakdowns and recovery actions. We have identified common co-occurrences of codes.

During the study, participants used code completion extensively even though they weren’t prompted to do so. This is confirms the findings of Murphy et al. (2006), which suggest heavy usage of code completion in programming work. However, we didn’t observe code completion being used as an exploration tool as frequently as has been reported in previous work (e.g. Stylos and Clarke, 2007).

One of the principal patterns that we found was the correlation between participants using code completion with the intention of writing code faster and more correctly. Further, their typical behaviour when having this intention was to filter the suggestion list and accept one of the few remaining options.

We also observed the main disruptions that can occur when interacting with code completion, as well as a set of recovery actions that participants engage in after such breakdowns. For example, not finding the desired completion had good correlation with the participants reading online documentation afterwards. We observed that many of the completions offered were not accepted by the participants, and that this behaviour is consistent with professional usage. This suggests that there is considerable room for improvement in the use of code completion as a search tool.

We noted the use of code completion failures as a static debugging tool confirming previous work (Church et al., 2010). When code completion didn’t offer any helpful suggestions, the participants considered this an indicator of an error elsewhere in the program, and switched to fixing their code.

We suggest that our findings might be used as a starting point for improving code completion systems. Given the observed high prevalence of code completion use as a debugging aid, further work might investigate how code completion systems can better support this usage.

7. References


Abstract. We describe the Infusion system, which is a library, language system or integration domain implemented in JavaScript, as well as a mentality and model for thinking about the expression of end-user applications. We promise that this system will bring together the worlds of different kinds of users engaged in different tasks at different times, and allow them shared authorial access to the same artefacts which are presented to each in a notation appropriate for them.

Keywords: POP-II-A. individual differences; POP-II.C. cognitive dimensions, data flow, visual languages; POP-I.A. programming economy; POP-I.C. web

1 Introduction

Differing notations bring different affordances — and are suited for different audiences and different tasks (Blackwell & Green, 2003). For example, notations with low viscosity might be appropriate during initial development of a new system, whilst others with few hidden dependencies might be more appropriate during maintenance. Those with powerful abstractions might be suited for experts, whilst others with good visibility might be better suited to novices or end-users. Traditionally, the choice of notation for a particular task implies more than a skin-deep commitment to a particular style of representation and way of working. For example, the choice of a conventional programming language such as Java or Haskell, based on the core representation of a stream of textual characters forming its source code, strongly limits the kinds of alternative notation which can be provided for other tasks and audiences. Correspondingly, the choice of a visual programming idiom such as Scratch (Maloney, Resnick, & al., 2010), Blockly (Google Developers, 2015), or Max/MSP (Cycling 74, 2007), cuts off the potential for engaging with audiences familiar with the power of traditional text editors and IDEs.

Our work for some years on the Fluid Project has been to build a system, Infusion, which aims to produce not just a single “middle way” between such extremes of notation, but also schemes for navigating between different notations in which “the same artefact” might be expressed. This will naturally involve some compromise between the needs of different audiences since, as in our examples above, the gap between the notational worlds of the visual and non-visual is not simply a matter of notation. The differences between the structure and referential style of, say, a Java program and a Scratch program are too profound to allow one to be usefully transformed and represented in the style of another.

Infusion evolves through repeated cycles of experimentation, validation and rationalisation, guided by some core heuristics. We still can’t clearly see the forms of notation that can deliver on the aims we have just described — but we have made some crucial architectural decisions which put us at variance with existing popular notations, without which we believe that these aims cannot be achieved. These focus on the use of what we call aligned, publically addressable state, an idiom we will enlarge on in later sections.

A clear source of inspiration for Infusion is taken from the highly successful “evolved” solutions embodied in web technologies — we claim both the document object model (DOM) and representational state transfer (REST) idioms as embodiments of the aligned state idiom just referred to. As well as being inspired by the web, Infusion is built for the web — it is a library of standard JavaScript that can be included in any modern browser, and harmoniously coexists with applications written in standard markup and widgets. It is also suitable for standalone JavaScript runtimes such as Rode.js.

2 Short guide and example

We’ll work through a simple application encoded by an Infusion component tree. This will be considered first from the point of view of some users (authors) named Users A, A’ and A”, employing traditional text-based development tools, and then from the point of view of other participants, Users B, C and D. We’ll then consider the kinds of interactions they might share through the application, and the potential lifecycles of these interactions.
2.1 A small example involving relay

The model relay system is used to set up permanent, possibly transforming, relationships between different bodies of state. This kind of capabilities is also currently comprised under today’s descriptions of reactive systems, particularly seen in the so-called functional reactive programming (FRP). In Figure 1, we’ll set up a small system consisting of two pieces of state linked by a transforming relay, held in two different components, and then show how we can interact with it from JavaScript, which we’ll call the base language. The components and relay are expressed as configuration in JSON, referencing function and component definitions in JavaScript via strings representing their globally namespaced names.

```javascript
fluid.defaults("examples.simpleRelay", {
  gradeNames: "fluid.component",
  components: {
    celsiusHolder: {
      type: "fluid.modelComponent",
      options: {
        model: {
          celsius: 22
        }
      }
    },
    fahrenheitHolder: {
      type: "fluid.modelComponent",
      options: {
        modelRelay: {
          source: "{celsiusHolder}.model.celsius" // IoC reference to celsius model field in the other component
          target: "{that}.model.fahrenheit" // this reference could be shortened to just "fahrenheit"
          singleTransform: {
            type: "fluid.transforms.linearScale",
            factor: 9/5,
            offset: 32
          }
        }
      }
    }
  }
});
```

Fig. 1. Short example showing a transforming relay from view of User A – temperature conversion

To start with, it’s worth noting that our design so far involves no JavaScript code. A single function call, `fluid.defaults`, is necessary to register the configuration with the system, but in other styles of interaction, for example the Nexus described in section 4 even this can be dispensed with. We’ll need to execute some further base language code to create an instance of this system and experiment with it, but one can imagine that this also could be dispensed with in other visual/non-visual authoring environments which might feature, for example, a graphical “playground” in which instances can be set up and torn down by direct manipulation (see Figure 5).

We now imagine that another user, User D, decorates this definition with some further elements (some shown in Figure 2) that can turn it into a live HTML interface, allowing a further user, User E, to use the interface (shown in Figure 3). User E, cast in the traditional role of an “end user”, can type numeric values into either field and see the opposite field update with the corresponding value in the other scale.

```javascript
// User D designates a "decorated variety" of our simpleRelay type which produces a live HTML interface
fluid.defaults("examples.relayApp", {
  gradeNames: ["fluid.viewComponent", "examples.simpleRelay"],
  components: {
    celsiusField: {
      type: "fluid.uiInput",
      options: {
        model: {
          value: "{celsiusHolder}.celsius"
        }
      }
    },
    fahrenheitField: {
      type: "fluid.uiInput",
      options: {
        model: {
          value: "{fahrenheitHolder}.fahrenheit"
        }
      }
    }
  }
});
```

// Construct an instance of the application bound to the current HTML document’s body element
var app = examples.relayApp("body");

Fig. 2. Decorating the model skeleton from Figure 1 to bind to a simple HTML interface (markup not shown)

Fig. 3. Simple HTML GUI for end user (User E) of temperature conversion tree shown in Figure 1

2.2 Decorating the skeleton for console interaction

In this section, we will imagine two further users decorating the same application skeleton in Figure 1 to perform interactions from the browser console. User A’ will decorate the base the base system with some model listeners
which will react to changes in the model values and report on them. We can do this i) without further application code, and ii) without needing to modify the above definitions. After that, user A” will use the language-level API to trigger modifications to the values and hence the reports. These interactions are shown in Figure 4.

```javascript
// User A’ designates a “decorated variety” of our simpleRelay type which will log messages on model changes
fluid.defaults("examples.reportingRelay", {
    gradeName: "examples.simpleRelay",
    distributeOptions: [{ // options distributions route options to the subcomponents in the tree compatibly
        record: { // options distributions route options to the subcomponents in the tree compatibly
            funcName: "fluid.log",
            args: ["Celsius value has changed to", "+{change}.value"],
        },
        target: "({that celsiusHolder}).options.modelListeners.celsius"
    }, {
        record: { // options distributions route options to the subcomponents in the tree compatibly
            funcName: "fluid.log",
            args: ["Fahrenheit value has changed to", "+{change}.value"],
        },
        target: "({that fahrenheitHolder}).options.modelListeners.fahrenheit"
    }
}],
});
fluid.setLogging(true); // send any logging output to the console
// User A’ uses the grade of User A’ to construct an instance of our decorated tree type
var tree = examples.reportingRelay();
// This will immediately report:
// Celsius value has changed to 22
// Fahrenheit value has changed to 71.6
// User A” uses the grade of User A’ to construct an instance of our decorated tree type
// Celsius value has changed to 20
// Fahrenheit value has changed to 68
// Celsius value has changed to 232.7777777777778
```

Fig. 4. Example of operating a transforming relay by Users A’ and A” — output is shown in comments

This shows that the relay has set up a lens between the state held in the two components. The relay operates from the point of construction onwards — and ensures that the model constraint is satisfied by the initial system as well as with respect to modifications at either end of the relay. This relationship will persist until one or other of the related components is destroyed, which will also remove the instance from its parent, as required by the cellular model described in section 3.3.

2.3 The application from different points of view

The original authoring of the application was by User A who finds it convenient to use traditional software development tools based on text buffers and function calls. We’ll now explore how we envisage how the authoring of this short application snippet could be shared with users of other kinds — for example, User B, who prefers a visual “boxes and wires” environment (mocked up in Figure 5) allowing development using direct manipulation by mouse, and the closely related User C, who would prefer a topologically identical environment, but instead mediated by speech and keyboard, in the style of the “nonvisual visual programming” environment presented in (Lewis, 2014). These would lead to the same experience by the end user E in Figure 3.

Fig. 5. Mockup of User B’s visual environment for authoring temperature conversion tree shown in Fig. 1

The notation/interface shown in Figure 5 contains the same information as that in Figure 1 (as would user C’s hypothetical non-visual representation). Since this information has been expressed in the form of aligned state, we can directly correlate parts of these interfaces together, as well as user actions directed at them — we speak more about this kind of alignment in section 3.2. Because of this correlation, we plan for these interfaces to be usable simultaneously, to author one and the same underlying “application”. Another result from this correlation is for user E’s view, “the actual application itself” to be the target of authoring actions, in the style of Self’s “the thing on the screen is the thing itself” (Ungar & Smith, 2013) model. This could be enabled by a “progressive disclosure” UI exposed, perhaps, to only some users in some contexts, allowing access to a progressively rich set of editing primitives drawn from the worlds of users C, B and A. The underlying application would be “live”,
to the extent that, if any of the participating authors introduce, say, a fresh temperature field showing values in Kelvin, all views would update to show it (user E’s only if a matching UI component were provided for it) — as well as, through the same underlying state-directed idiom, the current temperature value that a user had entered into any live embodiment of the application (for example, user A” or E) would be preserved, and shown in the new temperature scale too.

2.4 The link to Inclusive Design

Infusion is based on Inclusive Design practices, where software should be freely adaptable to meet the requirements of users with different notational requirements, whether these are prompted by cognitive, physical, interactional or other differences. It’s crucial that this can be done in an aligned way, such that the complete community of users sharing a need can share a particular coordinate, relating their embodiment of the notation to that used by another community. Our simple application can be seen as a direct example of such notational accommodation, in that a value (attached to a source of state in the world) is rendered to one user in one scale (Celsius), and another user in another (Fahrenheit). New adaptations can be freely introduced and removed from individual systems, without disturbing the wider community of cooperating users. More substantial adaptations can be introduced, such as accommodating special devices or modes of input and output.

2.5 “One person’s excess intention is another person’s secondary notation”

A crucial requirement in order to meet our goal of harmonious authorship from different notations is the construction of notations that are as free as possible from the expression of excess intention. Excess intention results when the notation we have available unavoidably captures more than what we intend to express in our design. Traditional programming languages, especially procedural ones, are famously rich in excess intention — some of which are being recognised and combated by newer notations, others of which are not. Here are two examples we have characterised:

Sequential Intention — Imperative programming languages unnecessarily force the creator to commit to an exact sequence of executed instructions, which is usually far in excess of the real requirements underlying the goals they are interested in. This is a criticism that is broadly acknowledged, and some responses to it are becoming widespread — for example, as expressed in the model of dataflow programming, or in monadic styles of packaging control flow.

Artefact Boundary Intention — Object-oriented languages force the designer into a single, exhaustive decomposition of their domain of interest into a non-overlapping collection of objects with well-defined names, properties, relations and contracts. However, another view of the same domain by a creator with different aims, skills or preoccupations might very well decompose it into an entirely different set of entities — which may or may not bear a strict hierarchical relationship with those from the first view.

Other sources of intention excess raise similar issues. In transforming from one notation to another, one must somehow capture all that is “excess” from one viewpoint with respect to another, and store it somewhere as an annotation to the structure — in exactly the same way one would be required to capture a secondary notation that had been attached in a notation, to preserve it during a stage of processing that was blind to it.

Our system addresses intention excess issues in a few ways. Our choice of configuration primitives is guided by an autoethnographic process that attempts to explain our intentions when writing base language code. Configuration styles which fail to do this economically are discarded, and the process continues to iterate. In general, intention cast in the form of aligned, transformable state naturally involves less excess than that in terms of other primitives (such as function or object definitions) — since, for example, both sequential and artefact boundary intention is minimised. Finally, the coordinate system that the state is endowed with provides a natural set of “hooks” on which to hang secondary notations as the primary notation is transformed between representations.

3 Theoretical underpinning and links to existing paradigms

Our configuration is organised as a set of globally named elements which are known as grades, which fulfil a few of the traditional roles of types in other systems, but fail to qualify in other areas. The configuration part of the system, since it consists of pure state aligned with a natural coordinate system, is ripe for transforming, expressing, and authoring in a variety of forms.

3.1 The role of programming languages and computational power

It is arguable whether Infusion is best described as a “programming language”, a “framework” or some other thing. It shares clear characteristics of both. The best designation that we have found so far is that of an integration domain (Kell, 2009), which is an arena for the naming and scheduling of effects, computations and their units of organisation, rather than an system in which computation is expressed directly. This issue, we feel, has long misdirected the field — since every notation which has been put into the role of “programming language” has been put under immediate pressure to demonstrate that it can express any computation (“is Turing-complete”) in order to qualify for this role. On the contrary, an integration domain, as noted by Kell, can easily be endowed with lesser computational power, and we argue, should be so. It is crucial, for example, for the transparency and
responsiveness of authoring tools, that relationships between parts of a program can be determined by the exercise of limited computational power. The Self family of languages emphasise the importance of such responsiveness for the feeling of authors that “the thing on the screen is the thing itself” (Ungar & Smith, 2013). A system or language whose structure implies the potential for unbounded computations (for example, those of a complex type system such as ML or Haskell) directly fights this aim. Such type systems, if provided, should be an optional addition to the system just for the use of a particular audience. Recent work on “gradual typing” (Siek & Taha, 2006) has tended in this direction, but so far there is little work on systems promising multiple independent, completely optional type systems for the same artefacts.

3.2 The first-class role of state, and transparent access to it

We promote the use of transparent, publically addressable state. The Infusion system should maintain all state — not just that on behalf of its users, but also its own book-keeping — in public view, with each piece of state available through an utterable1 public address. This is at odds with both object-oriented and functional programming, which insist that the state which the application manages on behalf of users must be hidden from view, either through data hiding in the former paradigm, or prohibition of side-effects in the latter.

Publically addressable state is the touchstone of the prevalent REST (Fielding, 2000) style of conversation or API for web applications, and this analogy has guided our development since the start. REST stands for representational state transfer — describing a conversation where state is represented, rather than opaque tokens traded, which represent mere behaviour or methods as is common in procedural or object-oriented API contracts, and also where state is transferred — that is, that the representation is an exhaustive summary of the state that can be used to move it from place to place.

The manifest nature of public state is crucial for many of the most successful embodiments of end-user programming. For example, in the spreadsheet paradigm, the programming surface consists purely of values in a grid. Each grid element has a well-known and mostly stable public address which can be used to access its value. Unfortunately from here on, the spreadsheet idiom starts to fall down, since any programming directives which are issued must skulk in a “hidden world” behind each cell, unaddressable either as a whole or in part. (Burnett & al., 2001) address this deficiency within the spreadsheet paradigm in particular. The public addressability of all design elements is crucial for a notation to allow good visibility and a lack of hidden dependencies when required.

3.3 A system inspired by the web, and built for the web — IoCSS selectors

The web represents the most highly successful “evolved strategy” for dealing with the problem of distributed and shared authorship. Whilst it appears to fall short of what are claimed as its antecedent blueprints, for example, in Ted Nelson’s elaborate hypertext system Project Xanadu (Nelson, 1982), as well as being regularly claimed as a deficient abstraction by object-oriented and functional programmers alike, we feel that there is a great deal to study, admire, and learn from the solutions and strategies that it embodies.

Together with REST, discussed in section 3.2, another successful idiom that is essential for the day-to-day running of the web is CSS, the scheme familiar to designers and developers alike for describing the styling information applied to web pages. CSS fills a crucial role in brokering between distributed authors of “the same document” who live in different communities, with differing workflows and tools. The space of DOM elements in a web page is a shared authorial space that must be malleable in the face of demands of varying strength from different ends of the process (design and logic). The space of CSS selectors can be “negotiated” in that the requirement to identify a particular piece of the document could be met “opportunistically” by choosing a selector which matches it contingently and unstably, or by arranging/negotiating to alter the document structure to allow a selector to match it more stably and semantically.

Analogously, Infusion implements a selector system that can be used to flexibly refer to components within an application’s component tree. We refer to this system as IoCSS, named after the framework’s role as an “Inversion of Control” system. This implies that what has been previously thought of as “an application” has been endowed with a regular but free-form cellular structure. In the case of an Infusion application, the cellular unit is the component, rather than as it is with an HTML document the DOM node. The affordances of an Infusion component are unusual set against those of typical units of software designs, given that they may be freely embedded recursively, and that further subcomponents may be injected into existing parents without their “knowledge” or disturbing the design. In object terms, Infusion components offer the possibility for containment without dependency, which is not possible in an object-oriented system.

Once we have the cellular structure in place, we now need some landmarks. In the DOM, these are provided by CSS class names, tag names and other well-known DOM attributes. In an Infusion application, these are provided by the context names which can be derived from the grade names attached to a component and the member name used to embed it in its parent. Some roles for IoCSS selectors using these landmarks are summarised in Table 1.

1 All state in a system has some kind of address, but in practice not all such addresses are utterable by ordinary (that is, not specially privileged through forming part of the compiler, runtime or operating system) users or authors. For example, state held in a function closure is held at an address that cannot be named from programming language code outside the closure. This implies that intentions held by users about such state cannot be encoded and acted upon.
4 Current Work and Future Developments

Examples of real-life applications built using Infusion can be seen at http://fluidproject.org/ — in particular our User Interface Options tool, itself an instance of our Preferences Editing Framework embedded on our documentation site at http://docs.fluidproject.org/infusion. This tool can be dropped into any web page to allow the user to customise the page’s presentation — for example, by selecting a custom font size, line spacing, contrast colour scheme or other accessibility adaptations. For the EU project “Prosperity4All” (P4A - see http://docs.fluidproject.org/infusion), we will be developing a portable and self-contained embodiment of the framework’s facility as an integration domain named the Nexus. The decomposition of updates from a text buffer into constituent Nexus messages will also be useful in other environments. In working with the Flocking (Clark & Tindale, 2014) system for audio synthesis on the web, we plan to close up the gap between the nature of performance and score by treating both as harmoniously cooperating elements on a common footing in a sea of state.

References


Table 1. Guide to terms used in this paper and relation to common forms

<table>
<thead>
<tr>
<th>Term</th>
<th>Correlates</th>
<th>Distinction and Similarities</th>
<th>Intention and Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grade</td>
<td>Type/Class</td>
<td>Rather than establishing contracts or describing storage, a grade is a block of (JSON) configuration with a globally-qualified name which is merged in an aligned way with others to produce a description from which component instances can be built. Grade names can also be used as landmarks (context names) in order to bind segments of IoCSS selectors.</td>
<td>The use of grade-based descriptions reduces excess intention in descriptions of parts of implementations. The run-time structure of an instance is much more closely tied to the authoring-time structure, allowing for the “notation” of authors and users to be directly corresponded.</td>
</tr>
<tr>
<td>Model</td>
<td>Model (MVC Programming) / Model (Model-based development/MBD) / Behaviour (Functional Reactive Programming/FRP)</td>
<td>Infusion models encode mutable state in a JSON-equivalent form. Taken together with the associated model relay rules, these can constitute a model from the MBD point of view, since the space of model states can be deduced. Finally, the stream of values of a model over time can be compared to an FRP behaviour, transcended into other streams via transforming relay rules.</td>
<td>Similar to the use of grades, Infusion models minimise divergence between run-time and authoring structures. They also aid liveness and transportation of applications — it should be possible to effectively move an application between systems or users by transmitting just its models.</td>
</tr>
<tr>
<td>Options</td>
<td>Advice (AOP)/Diff (VCS)</td>
<td>An options distribution, like an aspect-oriented programming “advice”, allows an existing application (component tree) to be modified by an author from the outside — that is, they can derive a variant application without modifying the expression of the original author. Unlike an advice, distributions have the same structure and syntax as ordinary configuration.</td>
<td>Since options distributions form a closed system, it is clear how multiple authors can collaborate on the same system, and multiple modifications competing to target the same piece of the design can have their relative priorities negotiated. This also implies that the same authoring tools can be used to write and check distributions as well as ordinary configuration.</td>
</tr>
</tbody>
</table>

Table 1. Guide to terms used in this paper and relation to common forms
DOCTORAL CONSORTIUM

Interdisciplinary research is always difficult. The researcher needs to be well-informed and up-to-date in at least two disciplines – for PPIG, that might mean being on top of say software design or computer education and at the same time fully conversant with relevant aspects of cognitive psychology, or maybe ethnomethodology or sociolinguistics or .... Hard enough to be well-informed in one discipline, let alone two or more.

If interdisciplinary research is hard for established researchers, how much more so for doctoral students, who have so little time to master so much scholarship. In many cases we see PhD students finding excellent support in one discipline but much more limited support in other aspects, for the simple enough reason that they are members of a department that specialises in that one discipline but has no staff member also interested in the second discipline. So we see computer science doctoral students trying to get help with psychology or the like, or maybe – as it was in my own case – a psychologist trying not make howlers in computer science. For the student, the problem is how to find people who can help with the bits that their own department doesn’t deal with.

The PPIG doctoral consortium is one resource that we hope provides help. Here, students can describe their work in a context where generous support is offered and where specialists in other areas can make useful suggestions. Sometimes students will need to be guided to literature that they might have missed, sometimes their research methods will need refinement, sometimes their research questions will have overlooked valuable contributions from the past that could lead to sharper and more focused questions. PPIG doctoral consortiums have fulfilled all those roles. And we have a happy history of doing so in a friendly, constructive way.

That’s what we PPIGgers deem important. We don’t offer prizes for Best Student Paper, but we do offer as much support as we can muster, not only from experienced researchers but also from other students. Past generations say they have found this approach welcoming and helpful, and we hope the same will be true this year.

Don’t let this sound too pious and worthy. Believe me, it’s not a hardship to listen to students and offer help. They have such enthusiasm, such interesting and unexpected ideas and insights! I for one think I’m lucky to have the chance to take part in the doctoral consortium, and I’m looking forward to meeting this year’s students.

Thomas Green
Understanding code quality for introductory courses

Martijn Stegeman
Faculty of Science, University of Amsterdam
martijn@stgm.nl

Abstract. An exploration of the meaning of ‘code quality’ for introductory programming courses has led to a model that brings together teacher feedback and suggestions from professional handbooks. We translated this model into a rubric for giving feedback to students in their first courses. Using this rubric in teaching has encouraged us to reflect on the intent of programming language features and how these can be used appropriately. We now propose future work: how can ‘appropriate use’ be more systematically integrated into the model? Furthermore, we propose that using the derived rubric in teaching may very well confront students with unnoticed misconceptions and help them eliminate these. Is this a reasonable assumption? How can we test it?

Keywords: POP-VI-E. computer science education research, POP-V.B. phenomenology, POP-III-C. procedural/object oriented, POP-I-A. learning to program

Supervisors: Erik Barendsen and Sjaak Smetsers of Radboud University, Nijmegen, Netherlands

1 Theory: rubrics

In the past, teachers have used various tools to systematically assess student work. A very simple case would be a checklist, where a number of criteria are listed and can be checked off to indicate performance. Rubrics add to this by defining a number of levels of accomplishment, and optionally verbal descriptors to explain each combination of criterion and level (Sadler, 1985).

Checklist and rubrics have often been used as scoring tools, initially to gain reliability in the calculation of grades. In contrast, Andrade (2005) argues for the idea of an instructional rubric, primarily designed as a teaching tool instead of a scoring tool:

“A rubric that is cocreated with students; handed out; used to facilitate peer assessment, self-assessment, and teacher feedback; and only then used to assign grades is an instructional rubric. It is not just about evaluation anymore; it is about teaching. Teaching with rubrics is where it gets good.”

Defined as such, a rubric is composed primarily for use by students instead of teachers. By studying that rubric, their own work, and the work of others, students are encouraged to form a conception of expected (and current) quality.

2 Previous work: patterns of feedback in introductory classes

In previously published work, we have examined the feasibility of creating a rubric for introductory programming courses\textsuperscript{1}. Our focus was on deriving criteria and descriptors in a systematic fashion by studying the practice of teachers and professional software engineers. We first analyzed standards of code quality embedded in three popular software engineering handbooks and found 401 suggestions that we categorized into twenty topics. We also recorded three instructors who performed a think-aloud judgment of student-submitted programs, and we interviewed them on the topics from the books, leading to 178 statements about code quality.

\textsuperscript{1}This paragraph presents work that has been previously published; it is partially based on the abstract of Stegeman, Barendsen, and Smetsers (2014).
The statements from the instructor interviews allowed us to generate a set of topics relevant to their practice of giving feedback, which we used to select criteria for the model. We then used these instructor statements together with the book suggestions to distinguish three levels of achievement for each criterion. This resulted in a total of 9 criteria for code quality. The interviews with the instructors generated a view of code quality that is very comparable to what was found in the handbooks, while the handbooks provide detailed suggestions that make our results richer than previously published grading schemes.

This process has led us to a model of code quality criteria and accompanying levels that we have used to construct a preliminary rubric for introductory programming courses, as included at the end of this paper. This rubric seems to be much more complete and less arbitrary than previously published rubrics or grading schemes for introductory programming courses (Hamm, Henderson, Repsher, & Timmer, 1983; Howatt, 1994; Becker, 2003).

3 Next up: help students understand the role of language elements

When using the rubric with students, we gradually started motivating its contents by explaining that programming languages embody a certain intent. This is because programming languages are presumably created by programmers to solve certain problems that pop up when programming. Features in these languages are thus expected to be used in a certain way, and, as time passes, are used to solve other problems as well. In short, it seems that the question that the rubric wants to answer is this: “How do I appropriately use the tools that the language provides?” These ‘tools’, then, are indeed the language features that we find as the nine criteria in the first column of the rubric: comments, decomposition, etcetera. In this light, it becomes clear that the rubric tries to convey a common-sense approach to using such features. This immediately presents us with the question: can we validate or even select the criteria in an even more systematic fashion, by studying the theory and history of programming language design?

3.1 Reframing the model

In the current version of the rubric, all parts, including the language features, have emerged from a single bottom-up coding process. However, by studying programming language publications and history, we should be able to construct an up-front inventory of these features. To make the data analysis more systematic, and possibly to come to a more complete rubric, it would seem appropriate to make this part more theory-based. Doing this can then help us re-focus the data analysis on patterns in the use of programming language features; for example, in the first version of the rubric, we have already seen that for a whole family of formatting features (indentation, white space use), it is usually consistency that is asked for. Aren’t there many more of these patterns? Using those to select and formulate criteria may help us answer the “How do I appropriately...” question more clearly in the rubric.

3.2 Potential questions

1. Is there existing theory of programming language features?
2. ...that includes stylistic features such as use of whitespace?
3. How do these relate to existing models of code quality?
4. If needed, how can we create a complete overview of programming language features?
5. How is ‘appropriate use’ defined in literature and/or historical documents?
6. What separates ‘appropriate use’ from concepts like guidelines, patterns, etc.?
7. How useful is the rubric in follow-up courses that deal with more elaborate use of object-oriented features?
4 Later: confronting and eliminating misconceptions

Apart from teaching students to write ‘good code’, there is another possible motivation for giving systematic feedback on code quality. We think that bad code may well be a sign of remaining problems with learning to program.

4.1 Varying difficulties

Difficulties of learning to program have been studied extensively. Researchers have found, for example, that some students lack the skill to interpret and understand the problem statement (McCracken et al., 2001); some have misconceptions of the notional machine that a programming language represents (Utting et al., 2013); some produce semantically correct fragments but have difficulties combining those (Soloway, 1986); and some lack the skill of mentally simulating the runtime behavior of programs (Lister et al., 2004). In these studies, the authors discovered the difficulties as a result of studying bugs in the programs that students produce. However, Du Boulay (1986) described that even without bugs, there may still be difficulties:

“I have often been surprised at the bizarre theories about how the computer executes programs held even by students who have successfully ‘learned to program’.”

Berges, Mühlung, and Hubwieser (2012) studied this phenomenon. The authors asked students to create concept maps as proof of understanding, and compared those with programs the students had written while making use of those same concepts. In several categories, the students appeared to be able to apply concepts of which they showed no clear understanding. More recently, Teague and Lister (2014) found evidence that some students are able to trace programs without being able to explain them.

Something else that may go unnoticed without the presence of bugs is the difficulty with combining correct fragments of code into a working program. Dorn and Guzdial (2010) studied professional web designers. Most developers that needed new information in order to make progress used a process that they describe as trial-and-error: writing code, checking the results and finding appropriate information to make the code work. The authors compare this to the opportunistic programming described by Brandt, Guo, Lewenstein, Dontcheva, and Klemmer (2009), where developers construct functional programs while sometimes actively avoiding to learn the complicated syntax for future use.

4.2 Going forward

Students being at least able to construct a working program according to specifications is arguably a fine result of an introductory programming course. However, isn’t it possible to further the understanding of students by providing feedback that addresses the hidden difficulties they may still have? When assuming that low quality code may actually be the result of the hidden difficulties that we discussed, we can speculate that giving feedback on those aspects of code might indeed confront students with the underlying difficulties they are still having, and as such have the potential to encourage learning. One course of action for us would be to use a concept-test and combine this with systematic formative feedback on code quality, in order to find if positive effects can be seen.

4.3 Potential questions

1. Do students ask more conceptual questions when they are systematically given feedback on code quality?
2. Do students perform better on conceptual test when they are systematically given feedback on code quality?
3. Should we use a more qualitative study to gain insight into student understanding in this regard?
5 Conclusion

There is quite a bit of work embedded in the proposed refinement of the method of creating a code quality rubric, as well as in the proposed evaluation of its usefulness in the introductory programmer’s learning process. The above is a first try at defining future work and integrating the various potential strands of research. Suggestions on the proposed questions, on related theory and on the proposed methods are very welcome.

References


<table>
<thead>
<tr>
<th>Documentation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>names</td>
<td>names appear unreadable, meaningless or misleading</td>
<td>names accurately describe the intent of the code, but can be incomplete, fuzzy, lengthy, misspelled</td>
<td>names accurately describe the intent of the code, and are complete, distinctive, concise, correctly spelled</td>
<td>all names in the program use a consistent vocabulary</td>
</tr>
<tr>
<td>headers</td>
<td>headers are missing or descriptions are redundant or obsolete; or use mixed languages</td>
<td>headers summarize the goal of parts of the program and how to use those, but may be incomplete or misspelled</td>
<td>headers accurately summarize the role of parts of the program and how to use those, are spelled correctly, may be wordy</td>
<td>headers contain only essential explanations, information and references</td>
</tr>
<tr>
<td>comments</td>
<td>comments are generally missing, redundant or obsolete; or use mixed languages</td>
<td>comments highlight important decisions and potential problems, but may be wordy or misspelled</td>
<td>comments highlight important decisions and potential problems, are concise and spelled correctly</td>
<td>comments are only present where strictly needed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Presentation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>layout</td>
<td>old code is present</td>
<td>arrangement of code within source files is not optimized for readability</td>
<td>arrangement of code within source files is optimized for readability</td>
<td>arrangement of code is consistent between files</td>
</tr>
<tr>
<td>formatting</td>
<td>formatting is missing or misleading or lines are too long to read</td>
<td>indentation, line breaks, spacing and brackets highlight the intended structure but erratically</td>
<td>indentation, line breaks, spacing and brackets fully highlight the intended structure</td>
<td>formatting makes differences and similarities clearly visible</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow</td>
<td>there is deep nesting; code performs more than one task per line; control structures are customized in a misleading way</td>
<td>flow is complex or contains many exceptions; choice of control structures and libraries is inappropriate</td>
<td>flow is simple and contains few exceptions; choice of control structures and libraries is appropriate</td>
<td>flow prominently features the expected path</td>
</tr>
<tr>
<td>expressions</td>
<td>expressions are repeated or contain unnamed constants</td>
<td>expressions are complex; data types are inappropriate</td>
<td>expressions are simple; data types are appropriate</td>
<td>expressions are all essential for control flow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Structure</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>decomposition</td>
<td>most code is in one or a few big routines; variables are reused for different purposes</td>
<td>most routines are limited in length but mix tasks; routines share many variables; parts of code are repeated</td>
<td>routines perform a limited set of tasks divided into parts; shared variables are limited; code is unique</td>
<td>routines perform a very limited set of tasks and the number of parameters and shared variables is limited</td>
</tr>
<tr>
<td>modularization</td>
<td>modules are artificially separated</td>
<td>modules have vague subjects, contain many variables or contain many routines</td>
<td>modules have clearly defined subjects, contain few variables and a limited amount of routines</td>
<td>modules are defined such that communication between them is limited</td>
</tr>
</tbody>
</table>

- highlight features from all levels that are present in the code, starting at the lowest
- for each criterion, circle the level that is most representative of the features that are present
- no need to circle a level that is not relevant to the assignment
- level 2 implies that the features in level 1 are not present, level 4 implies that the features in level 3 are also present

Level 1: problematic features are present
Level 2: core quality goals not yet achieved
Level 3: core quality goals achieved
Level 4: achievement beyond core quality goals
Improving Readability of Automatically Generated Unit Tests

Ermira Daka
The University of Sheffield
ermira.daka@sheffield.ac.uk

Abstract. Unit testing is a commonly applied technique in object-oriented programming, where classes are tested using small, executable tests written as code. It is a laborious, time-consuming, and error-prone job, and even after tests are written, it requires developer to maintain them, and to understand code written by other developers. To support developers, unit test can be generated automatically using different testing techniques. However, since algorithms used for test generation are typically guided by structural criteria, generated unit tests are often long and confusing, and with possible negative effect in the test oracle problem and test maintenance. To overcome this problem we investigate the readability of unit test cases. We propose a domain-specific model of unit test readability based on human judgments, and use this model to guide automated unit test generation. The resulting approach can automatically generate test cases with improved readability with the overall objective of reducing the effort for developers to understand these test cases.

Keywords: Readability, unit testing, automated test generation

1 Introduction

Software testing is one of the key processes for quality assurance in software systems. Every little change in a software needs to be tested in order to verify and validate its correctness in the application. Due to the ever-growing size and complexity of software programs, testing is becoming more challenging and costly.

Unit testing is a well known technique in object oriented programming, where different frameworks have been created (e.g. JUnit, NUnit) in order to reduce human effort and increase probability of finding bugs. To further support developers on manually creating test inputs, automated techniques have been devised that based on program under test, generate test suites. This technique is found to be very supportive, however, any test failures require fixing either the software or the failing test, which is a manual activity that needs one to understand the behavior of the test.

How difficult is to understand a unit test depends on many factors. For example, unit tests consist of sequences of calls to instantiate objects and bring them to appropriate states. There are often several such objects within a single test case: at least one for the class under test, several as parameters of calls on this instance, and possibly further objects to permit checks of correctness (for example, when explicitly creating an object that is in the expected state, and then checking whether the state of the object under test equals to that expected object state). The difficulty of understanding such a test case is thus directly influenced by the particular choice of sequence of calls. For example, consider the two test cases in Figure 1. Both tests check the same method equals() in the class ComparatorChain, but they look different in presentation. In terms of readability features, the first test is longer, contains more lines, defines more variables, has more parameters, that makes it less readable, hence difficult to understand.

```java
ByteBuffer byteBuffer0 = ByteBuffer.allocate((int) (byte)0);
BitSet bitSet0 = BitSet.valueOf(byteBuffer0);
ComparatorChain comparatorChain0 = new ComparatorChain((List) null, bitSet0);
ComparatorChain comparatorChain1 = new ComparatorChain();
boolean boolean0 = comparatorChain0.equals((Object) comparatorChain1);
assertFalse(comparatorChain1.equals((Object)comparatorChain0));
assertEquals(false, boolean0);

ByteBuffer byteBuffer0 = ByteBuffer.allocate((int) (byte)0);
BitSet bitSet0 = BitSet.valueOf(byteBuffer0);
ComparatorChain comparatorChain0 = new ComparatorChain((List) null, bitSet0);
ComparatorChain comparatorChain1 = new ComparatorChain();
boolean boolean0 = comparatorChain0.equals((Object) comparatorChain1);
assertEquals(true, boolean0);
```

Fig. 1. Two versions of a test that exercise the same functionality but have a different appearance and readability.
The aim of this research is to improve the quality of generated test cases by increasing their readability. Readability of generated unit tests will be improved by creating additional optimization functions in the tool that will generate the test. By improving readability in software unit tests cases, time required for understanding the code can be saved, accuracy and productivity will also increase, thus the software can be tested faster and correctly. Consequently, this research focuses on creating a readability model for automatically generating more readable test cases. which further, will serve as an additional test adequacy criteria in EvoSuite (Fraser & Arcuri, 2011) automatic test suite generation tool, which will be able to generate readable test cases.

The main contribution of this study will be improving automatic unit test generation. The readability prediction techniques and concepts presented in this research, implicitly provide requirement for automatic unit test suites with readability optimized.

2 Research Questions

The main aim of this work is to optimize test cases in terms of readability such that developers quickly understand the behavior of the test. Therefore, we will investigate on these research questions:

1. How can we create a model that can quantify readability of unit tests?
2. How can we integrate our metric in a testing tool, such that we can improve the readability of automatically generated unit tests?
3. Does improved readability lead to developers understanding the tests better?

3 State of the Art in Test Readability

Although test code readability does not affect the functionality of the program at all, it has an important role on increasing the understandability of the code under test while fixing it. Considering that a test case may contain several statements, and all of them are important for bringing the object under a certain state, the probability of interpreting it in the wrong way is very high, which may bring up errors in the whole system.

In order to make test understandable Zhang in his work (Zhang, 2013), presents a technique for test code simplification in semantic level. The work aims to create simpler and shorter test cases that can be better understood. The process starts with an initial test case, replaced referred expressions with alternative ones and each time constructing a simpler test until the same code is still covered. Test minimization is investigated further from Leitner at al. (Leitner, Oriol, Zeller, Ciupa, & Meyer, 2007), who presented a combination of static slicing and delta debugging techniques that automatically minimizes the sequence of failure-inducing method calls in a test suite. The proposed minimization approach based on static slicing is highly efficient, practical, and easy to implement. One other work that include test case minimization, is implemented in Evosuite (test case generation tool) (Fraser & Arcuri, 2011), that provides small test suites (in terms of both test data and assertions) to the user that have to manually verify them. Although, proposed minimization techniques are successful on producing simpler/shorter test cases, they still can generate tests that are too difficult, which require a technique that makes them more readable.

Beside various approaches used to evaluate readability, there is another work done for improving the readability of string test inputs that came from Afshan at al. (Afshan, McMinn, & Stevenson, 2013). This work uses the natural language model approach into a search-based input data generation process in order to improve string inputs from perspective of human readability. Using an empirical study they evaluated their model which showed a great improvement in input readability. Developed technique improves readability of a feature (string inputs) without weakening the test criteria, however it concludes that given a certain time budget for testing, readable inputs reduce oracle checking time, but increase the number of test cases that may be considered.

Program comprehension and readability is an important activity during software evolution and maintenance. Buse and Weimer (Buse & Weimer, 2010) introduced a code readability metric based on human judgments. They collected human annotation data for code snippets and trained a classifier based on those scores. Posnett et al. (Posnett, Hindle, & Devanbu, 2011) used the same dataset to learn a simpler model of code readability, using fewer features based on size, Halstead metrics, and entropy. Although unit tests are also just code in principle, they use a much more restricted set of language features which require a specific model for readability.

Considering that automatically generated unit tests have a specific structure, since they are generated based on some objective, they also have to go through restructuring phase where developers need to understand the testing code. With this work we aim to add the readability as an additional feature in unit test case generation tool, which will help developers understand the tests.

4 A Readability Model to Optimize Unit Test Generation

Automatically generated unit tests are often unreadable because there is no immediately measurable metric for the readability of a unit test. Automated test generation techniques can typically only target simple, code-related
metrics, such as structural coverage criteria. How nice the resulting test cases look like is usually coincidence - from the point of view of an automated test generator. In order to investigate the readability of test cases that are written or generated, we performed a couple of experiments with human annotators that were able to judge on test code readability.

A high-level overview of our approach is provided in Figure 1. Our approach consists the following steps: 1.) Training data collection, 2.) Feature extraction for training set, 3.) Machine learning model, and 4.) Automatically generating more readable test cases.

To collect our readability data we performed an experiment on Amazon Mechanical Turk\(^1\), where human annotators were presented with a test case, and based on their perception of readability they had to rate it with a score from 1 to 5. Test methods were selected from open-source Java projects (Apache commons, poi, trove, jfreechart, joda, jdom, itext and guava), containing automatically generated test cases with EvoSuite (Fraser & Arcuri, 2011), and manually written test cases. Participants were required to pass a Java qualification test to ensure familiarity with the language. As result we collected 15,669 human judgments of readability (in a range of 1–5) on 450 unit test cases.

For our readability machine learning process, we defined a set of 116 initial structural, logical and density features listed in Table 1. In order to extract features from unit test cases, we implemented an application that results with a numerical vector of size 117 where each position represents a value for a feature, while the last position is reserved for class attribute (value to be predicted). We extracted features of 450 test cases and together with average scores collected from human annotators, we used them for model training in Weka (Holmes, Donkin, & Witten, 1994) data mining software. To learn the test readability model, we used a simple linear regression learner (Witten, Frank, & Hall, 2011), although in principle other regression learners are also applicable (e.g., multilayer perceptron (Witten et al., 2011)).

We investigated which features have the most predictive power by re-running our all-annotators analysis using different feature selection techniques (one feature at time, leave one feature out, reliefF, correlation, forward feature selection, backward feature selection). Although different techniques agreed with each other, with forward feature selection we learned a formal model based on 24 features, including line width, aspects of the identifiers, class ratio, single char occurrence, and byte entropy.

In order to test our model prediction performance, we conducted another experiment with unit test pairs. Each pair contains test cases with equal functionality but unequal in terms of readability features. Annotators were presented with pairs of test cases, and based on their readability perception they had to choose the most readable sample in the pair. With the trained model, for each of the tests in a pair we predicted the readability score, and then ranked the paired tests based on their score (represented with 0 or 1). Then, we measured the agreement between the user preference (i.e., 0 or 1, depending on whether more human annotators preferred the first or second test in the pair), and model prediction, using Pearsons correlation method. In the end, the overall best model with 24 features (shown in bold in Table1) achieves a correlation of 0.79 with a root relative squared error rate of 61.58%, and has a high user agreement (0.73).

The last part of this work, was to apply the test readability model in EvoSuite automated unit test generation, and produce readable test cases. EvoSuite generates unit tests, typically with the aim to maximize code coverage of a chosen test criterion. In order to keep this as primary objective, we applied readability as a post-processing step. The implemented algorithm takes as input a test case, minimizes it with respect to its coverage criteria.

---

\(^1\) www.mturk.com

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Total</th>
<th>Max</th>
<th>Avg</th>
<th>Feature Name</th>
<th>Total</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Length</td>
<td>✓</td>
<td></td>
<td></td>
<td>Line Length</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Identifier Length</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Identifiers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Numbers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Keywords</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Commas</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Periods</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Keywords</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Branches</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Assignments</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Assertions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Unique Methods</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Class Instances</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Characters</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>String Length</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Unique Classes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>String Content Score</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Methods</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Nulls</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Fields</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Single Characters</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Exceptional Comments</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Class Ratio</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Method Ratio</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Boolean</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>String Accesses</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Arrays</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Token Entropy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Byte Entropy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Halstead Effort</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Halstead Difficulty</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Single Identifier Occurrence</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1. Features used for Test Readability Model
and generates alternative test cases covering the same code. Thus, individuals of the same population having an equal fitness value, are ranked based on the readability score, where the more readable test case is generated.

5 Preliminary Results

We integrated our test readability model in test generation (EvoSuite), and achieved to produce our readability optimized test cases. The results are further evaluated using three different techniques, and they are already published in (Daka, Campos, Fraser, Dorn, & Weimer, 2015).

First, using an independent testing set we compared our model performance and predictive power, with existing code readability models. From the results that we collected, we saw that our test readability model performs better on test snippet datasets, achieving a higher agreement with human annotators than previous code readability models. Although the existing code readability models show a good performance on code, they could not achieve to have higher agreement on test code. Thus, the result from this evaluation suggests that our choice of features is well adapted to the specific details influencing test readability.

Second, using readability model together with test generation, we evaluated whether test optimization approach is successful. We integrated our metric into EvoSuite, and with a post-processing step we optimized test cases in terms of readability score. This process is applied on generation of 30 classes with 10 repetition, with tool defaults parameters and with readability optimization. Each pair of tests is generated for the same coverage objective, but differs in readability features. On all but three classes there is a significant improvement on the readability score, by an average of 1.9%.

Although test readability model can increase readability level in test cases, we conducted another experiment that gather information whether humans prefer optimized tests. Using the same readability optimization approach and tools default configuration, we generated test case pairs (differing on readability score) for 30 other classes and collected human choices. Given total responses, 69% of humans agree on what a readable test case is.

Our last evaluation experiment consisting of knowledge questions (whether a test case will pass/fail) was conducted to gather information about the effect of readability on test understanding. Users were presented with 10 different optimized/non-optimized test cases together with source codes needed for that specific test. Results collected showed that response time for readability optimized tests is 14% less, even that it did not directly influence participant accuracy. Moreover, for seven out of the ten classes, the time participants required to make a decision about the pass/fail status of a test was lower for the optimized test cases.
5.1 Open Issues and Future Directions

Our approach aims at generating test cases that are readable. However, from initial results we find that our technique is still quite limited in the scope of its changes to test appearance. For example, identifier names are generated according to a fixed strategy (i.e., class name in camel case, and a counter attached at the end). Our feature analysis suggests that identifiers have a very strong predictive power and influence on readability, and indeed the participants of our experiment mentioned the choice of variable names repeatedly in the post-experiment survey. This confirms previous research showing the effect of identifier names in source code (Caprile & Tonella, 1999), (Takang, Grubb, & Macredie, 1996), (Anquetil & Lethbridge, 1998), (Eshkevari et al., 2011), (Lawrie, Morrell, Feild, & Binkley, 2006), (Binkley et al., 2013), (Guerrouj, 2013), and suggests that future work will need to investigate this problem for unit test cases (e.g., (Allamanis, Barr, Bird, & Sutton, 2014), (Deißenböck & Pizka, 2006), (Caprile & Tonella, 1999), (Caprile & Tonella, 2000)).

Furthermore, we aim to extend our approach beyond readability, by taking other factors into account that may influence how difficult tests are to understand, for example the complexity of the control flow. Our last experiment about the effects of readability on test understanding has also demonstrated that there is a boundary between readability and understandability. The problem of understanding has been seen of high importance especially if a test fails, and investigated from Leitner at al. (Leitner et al., 2007) and Lei at al. (Lei & Andrews, 2005). They worked on minimised failing (randomly generated) tests using delta-debugging in order to simplify debugging the failure cause. Zhang also presented an approach to synthesis natural language documentation to explain the failure (Zhang, Zhang, & Ernst, 2011). However, to the best of our knowledge there is still not any approach that tries to optimize test cases in terms of understandability.

6 Acknowledgements

I thank Gordon Fraser for supervising me during my research, Jose Carlos for test generation and creating web interfaces for evaluation process, Jonathan Dorn for experiment conductions in Amazon Mechanical Turk, and Westley Weimer for his supervision during our Readability Model study.

References


Analysing Java Identifier Names in the Wild

Simon Butler
Department of Computing and Communications,
The Open University, Milton Keynes MK7 6AA, United Kingdom

Abstract. Identifier names represent the entities manipulated by a computer program and the actions performed on them and are crucial to program comprehension. This research aims to improve understanding of the forms of identifier name created by software developers and advances techniques for analysing names that can also be applied to improve software engineering tools.

1 Introduction

Source code is the written expression of a software design consisting of identifier names – strings consisting of abbreviations, acronyms and natural language words that represent the entities manipulated by the program and the actions performed upon them – embedded in a formal framework of keywords and operators provided by the programming language. Identifier names are crucial for program comprehension [25], a necessary activity in the development of software. For example names are used to support requirements tracing [2] and concept location for program maintenance [10][17][22], and to determine the consistency of concepts expressed in source code [24][12][23]. The development of techniques to analyse identifier name structure and content [9][21][20][17][7][5] improves understanding of the mechanisms used by developers to encode information in names and increases the accuracy and usefulness of analytical techniques.

Extraction of information from identifier names by software engineering tools is constrained by limitations to the understanding of how developers structure identifier names. Though a lot is known about the structure of method names in Java [20], class and reference (field formal argument and local variable) names — around 62% of unique names and 72% of name declarations in a corpus of 60 FLOSS Java projects [8] — are less well understood and the evidence supporting that understanding is incomplete.

This research attempts to improve understanding of the content and forms of class and reference identifier names created by developers. To understand the variety of names found in multiple Java projects we needed to develop automated tools, which were unavailable when the study started, to extract tokens from names, perform phrasal analysis and to specify and test naming conventions. In Section 2 we describe improvements to existing techniques to split identifier names into their component tokens we made to support our work, and Section 3 outlines techniques we developed to categorise tokens found in names and analyse the phrasal structure of names, where appropriate. Our work has many potential applications for program comprehension tasks, which we illustrate in Section 4 with a system for checking adherence to naming conventions and a proposal for a system for recommending corrections to names. In Section 5 we draw our conclusions and identify some future work.

2 Identifier Name Tokenisation

To access the content of identifier names requires splitting each name into its component tokens. We use the term token because names are not always composed of words. For the majority of names (around 80% in our corpus [6][8]) the task of identifying and separating tokens is trivial because developers follow typographical conventions that indicate the boundaries between tokens. In Java, the target programming language of our investigations, two conventions are used. The most common capitalises the initial letter of each token, and is commonly known
as camel case, e.g. `isEmpty`, which may be divided into tokens by splitting the name every time a transition from lower to upper case is encountered. The alternative convention, used for constants in Java, is to insert underscores between tokens in upper case, e.g. `BUFFER_SIZE`, which can be split by dividing the name each time an underscore is encountered.

The remaining names contain ambiguous or unclear token boundaries. Some developers, conventionally, capitalise acronyms (e.g. `HTMLEditorKit`), which implies rather than marks a token boundary and, sometimes, do not mark token boundaries (e.g. `commonkeys`). A human can easily divide both names into their tokens, but for software the task is more difficult. A further problem arises from the use of digits in names. Digits may be part of an acronym (e.g. `MP3`, `J2SE` or `3D`), used as a suffix (e.g. `index2`), or a homophone substitution for a preposition (e.g. `html2xml`). Existing techniques for tokenising identifier names developed by Feild et al. [13], Enslen et al. [11] and Guerrouj et al. [15] do not fully address these problems and, at best, treat digits as separate tokens.

We developed an approach to tokenise names containing consecutive upper case characters where the remaining typography is camel case such as `HTMLEditorKit` and `PBinitialize`. The use of two or more consecutive upper case characters indicates that there is a token boundary either before or after the last upper case character in the group. The name is split on both boundaries and the resulting tokens evaluated to determine whether they are recognised words, abbreviations or acronyms, i.e. `HTMLEditorKit` is split to {`html`, `editor`, `kit`} and {`htmle`, `ditor`, `kit`} with the former being chosen as the better solution [6].

Tokenising single case names — by which we mean names without typography that might indicate a token boundary — such as `outputfilename`, employs a similar approach. The name, or part name, must first be tested to determine whether it is a recognised token, or may be a simple neologism created by derivation, e.g. `throwable`, or possibly a spelling mistake. Feild et al. created the example of `thenewestone` to illustrate some of the problems to be solved when designing algorithms to tokenise a single case name [13]. We developed algorithms to identify component tokens that improved on existing methods in two ways. Firstly, we examined all feasible tokenisations of a string processing it from left to right and right to left, rather than the existing approach of recursively extracting the longest recognised token while processing the string from left to right [13]. The candidate tokenisations are then evaluated and preference given to tokenisations found by passes in both directions, and then to those containing the fewest whole words. Should a satisfactory tokenisation not be identified, we apply an enhanced version of the algorithm that recursively processes the string by removing initial characters before testing if the remainder of the string begins with a known word. For example a name such as `xxfilenameyy` is tokenised as {`xx`, `file`, `name`, `yy`} following two iterations of the left to right pass where the leading ‘x’s are removed.

We also developed a set of heuristics for tokenising names containing digits [6]. Initially a test is made for known digit containing acronyms and the name split accordingly. If the digits are not part of a recognised acronym they are separated from the surrounding name, and heuristics used to assign them to the token to the immediate left or right, or retain the digits as a separate token. All the approaches described are implemented in INTT (identifier name tokenisation tool) a Java library used in our research projects.

3 Analysis of Name Content and Phrasal Structure

Høst and Østvold undertook extensive analysis of Java method names [19][20] and most of the phrasal structures used by developers are understood. Understanding of Java class and reference names is limited. Singer and Kirkham observed that around 85% of Java class names are composed of one or more nouns preceded, optionally, by one or more adjectives [26]. Liblit et al. [21] identified a richer use of phrases in identifier names than predicted by naming conventions, such

---

1 INTT is available from [http://oro.open.ac.uk/28352/](http://oro.open.ac.uk/28352/)
as *The Java Language Specification* [14] and *The Elements of Java Style* [27], particularly for reference and method names, though they did not report detailed quantitative results. Høst and Østvold’s and Liblit *et al.*’s work have informed pragmatic approaches to the extraction of information from names by Hill [17] and Abebe and Tonella [1]. While both methods are able to extract information from source code, they rely on constrained models of name structure and are therefore not comprehensive solutions.

Naming conventions state the majority of class and reference names should be nouns or noun phrases, while Liblit *et al.*’s observations give rise to the expectation that some names — boolean references specifically — will be verb phrases such as ‘is empty’. Further complications arise for reference names from the content specified in naming conventions: some emphasise use of abbreviations and acronyms derived from type names [14], while others insist on the use of dictionary words as much as possible [27]. We have undertaken studies of Java class [7] and reference names [5] to try to understand their structure in more detail, both in terms of what components are used and, where appropriate, their phrasal structure.

### 3.1 Class Names

We investigate the lexical and syntactic composition of Java class identifier names in two ways. Firstly, we identify conventional patterns found in the use of parts of speech (PoS). Secondly, we identify the origin of words used in class identifier names within the name of any super class and implemented interfaces to identify patterns of class name construction related to inheritance [7].

Researchers have tried three approaches to PoS tagging identifier names. The first two use off the shelf PoS taggers either alone [17], or in combination with template sentences into which tokenised names are inserted before the sentence is tagged [1]. The third approach is the creation of a PoS tagger specifically for names. Høst and Østvold developed a tagger for Java method identifier names [19], and Gutpa *et al.* created a more generic tagger for names [16].

Available PoS taggers are designed to process sentences, and have been trained on corpora other than identifier names. The Stanford tagger, for example, is trained on a corpus of Wall Street Journal articles. Wall Street Journal articles typically lack the technical vocabulary often found in identifier names, and the sentences in the corpus provide considerably more context than an identifier name for the tagger to use to determine the correct parts of speech.

Our initial experiments with the Stanford tagger found the default model was unreliable and experiments with Abebe and Tonella’s idea of using template sentences [1] to nudge the tagger by providing additional information also seemed not to improve accuracy greatly. We trained a model for the Stanford PoS tagger using 9,000 hand-tagged Java class names. This approach proved more accurate than using the default model with class identifier names, with an accuracy of 87% for class names and 95% for individual tokens.

We analysed 120,000 class names from our corpus of 60 FLOSS Java projects. Analysis of the PoS tags confirmed Singer and Kirkham’s observation that 85% of class names are noun phrases. Furthermore we identified another two commonly used patterns of PoS that, together with the noun/noun phrase pattern, account for 90% of the observed class names.

We also undertook a case study of the 652 class names in FreeMind, a mind mapping application, to try to understand the significance of unusually structured class names. We found that 53 names did not match the most common forms of name. While some described actions in the GUI, and others followed project specific naming conventions for small groups of related classes, a few indicated a need for either the name to be refactored to one of the common patterns, or the class to be refactored.

Java classes may extend zero or one super class, and implement zero or more interfaces, or super types. As well as being interested in the phrasal structure of names, we are also interested in the source of name components reused in class names. Developers use a number of patterns including one where the name of the superclass or a supertype is repeated in the name of the
subclass or subtype and preceded by a noun or sometimes adjective denoting the specialisation embodied in the class. The pattern occurs widely, for example in the `java.util` package where the `Map` interface is implemented by the class `HashMap`, among many, which is further extended/specialised by the `java.util.concurrent.ConcurrentHashMap` class. We found a wide variety of forms of reuse of components of superclass/supertype name. Our study only examined a single generation of inheritance and we think that detailed study of naming patterns in inheritance trees will support a more complete understanding of the patterns observed.

3.2 Reference Names

Naming conventions specify a variety of content types for reference names. The Java Language Specification, for example, advocates the use of ciphers (well-known single letter abbreviations that represent a generic value of a specific type such as the familiar `i`), type acronyms (e.g. `StringBuilder sb`), abbreviations, acronyms, words and phrases [14]. To analyse the phrasal structure of reference names, we first need to identify those names consisting of the content types — acronyms and dictionary words — that might be combined to form phrases. We identified five types of token content specified in naming conventions — cipher, type acronym, abbreviation, acronym and word — a further type, the redundant prefix (a single letter prefix used to indicate type or species), that we have observed in some projects, and a unrecognised category that includes unrecognised abbreviations, neologisms and spelling mistakes. Abbreviations, acronyms and words are identified using `mdsc`2, a spell checker developed for identifier names, which incorporates the SCOWL wordlists [3] used by the GNU Aspell spell checker and lists of abbreviations and technical terms from our work and the AMAP project3 [18]. Categorising tokens allows us to identify four categories of name: two non-phrasal composed of ciphers and type acronyms, one phrasal composed entirely of words and acronyms, and a fourth category where one or more tokens require additional processing, such as abbreviation expansion, checking for spelling errors and neologisms, before the name can be determined to be phrasal.

We analysed 3.5 million reference name declarations and found the majority of projects in our corpus contain a mixture of all four categories of name, with a predominance of phrasal content. The outliers are an important consideration for those building software engineering tools because some projects investigated contain names almost entirely formed of dictionary words and acronyms, while in a few projects 50% or more of name declarations contained at least one unrecognised token.

To analyse the phrasal structure of names we trained another model for the Stanford PoS tagger because the model trained on class names was less accurate with reference names. Again we achieved a similar accuracy for tagging names at 90% and individual tokens at 95%. Using the Stanford parser to identify phrases, we found names consist largely of the phrases observed by Liblit et al. However, there are a small proportion of names with different phrasal forms: verb phrases to describe GUI events, prepositional phrases, and complex names consisting of multiple phrases (39 words in one case), most commonly used as identifiers for strings used in GUI messages translated to provide support for internationalisation. We also found a few names consisting of dictionary words with no recognisable phrasal structure, e.g. `isShowLines` [5].

4 Analysis of Adherence to Reference Naming Conventions

Identifier naming conventions are the basis of a uniformity of naming that supports the readability of source code. Naming conventions include typography, content and phrasal structure. Reference names form 52% of the unique names in our corpus and 69% of the declarations.

A significant question arising from the preceding study concerns whether developers use the range of reference name content and phrasal structures according to naming conventions.

2 `mdsc` is open source Java software and is available at https://github.com/sjbutler/mdsc
3 https://msuweb.montclair.edu/~hillem/AMAP.tar.gz
The preceding study provides a partial answer because we found unanticipated forms of name, particularly the extremely long names sometimes used for strings in resource bundles, and the use of type acronyms as field names.

We developed Nominal, a Java library and configuration language\(^4\) that allows users to specify naming conventions and test names for conformance with the conventions. We tested the 3.5 million reference name declarations in our corpus against the naming conventions defined in The Java Language Specification [14] and The Elements of Java Style [27], and a further set of conventions that reflect the phrase structures observed by Liblit et al. [21]. We found that developers do follow conventions, but that they are not always published conventions. Indeed in some projects local conventions on both typography and phrasal content are followed. For example, redundant prefixes are used extensively in a few projects, and typography is sometimes used to highlight details of the declaration, such as the use of the static Java modifier [4]. We also found occasional use of type acronyms as field names, which is not specified in naming conventions. Usage appears to be confined to fields of inner classes that provide generic services such as string processing [4].

Nominal provides us with an effective tool for the identification names that do not conform to naming conventions related to content and typography. We are working to extend Nominal so that the library can recommend corrections to names. The correction of typography is straightforward, where the typographical scheme is well defined. The correction of content, particularly phrasal content, is more challenging because changes to a name’s meaning may not reflect the developer’s intention.

5 Concluding Remarks and Future Work

Our research has investigated the content and structure of Java class and reference names. To undertake this work we have developed novel approaches to the tokenisation of single case names and names containing digits, and demonstrated the viability of training PoS tagger models for names. We have confirmed and reinforced the findings of Liblit et al., and extended knowledge of the phrasal structures used by developers in class and reference names. We also identified forms of name reuse in class names.

The knowledge acquired is applicable in a number of software engineering scenarios. Improved name splitting and understanding of name content supports better identification of semantic content that might be used for concept or feature location. Feature location may be further improved as a result of a better understanding of the roles words play in a name. Libraries such as Nominal and the knowledge of name content and phrase structure can be used to develop tools to support the work of developers and improve the internal quality of software.

There is scope to improve the techniques we developed. Our approach to PoS tagging achieves similar accuracy to approaches tried by other researchers. We believe that the development of a PoS tagger for identifier names could be more accurate, particularly if it is able to recognise type names and treats them as single semantic units. Developers can be creative with language, coining new words to fit their purpose. INTT contains a simple system to identify neologisms created by derivation. However, the development of better techniques for identifying neologisms, especially word blends, could improve the analysis of names.

Throughout this work we have been constantly surprised by the inventive ways in which developers use language in some names, and by the forms of name that they decide are appropriate for their needs. An in depth study of names in multiple projects and programming languages could be used to develop a comprehensive classification of name and token types to support the development of further techniques to process names.

\(^4\) Nominal is open source Java software and available from [https://github.com/sjbutler/nominal](https://github.com/sjbutler/nominal)
References