_Abstract

There is a diversity of models explaining organizational culture and how these complex aspects can be addressed in connection to organizational change efforts. This workshop paper claims that models already exist for dealing with the cultural change that an agile transition is in the software engineering context. Instead of realizing this again through agile success stories, and thus reinventing the wheel, it is argued that the research in the software engineering field should build on these models instead and investigate how/if they differ. Practitioners already work as the change agents described in other fields and they should get recognition through the presence and integration of these models in the software engineering process research.

1. Introduction

A few studies set out to investigate the social or cultural aspects of agile development. Whitworth and Biddle (2007), for example, verifies that agile teams need to look at social-psychological aspects to fully understand how they function. There are also a set of studies connecting agile methods to organizational culture. These connect the agile adoption process to culture to see if there are cultural factors that could jeopardize the agile implementation, which there are (Iivari & Iivari, 2011; Tolfo & Wazlawick, 2008). One study divides culture in different layers depending on their visibility according to Schein (2010). This article shows that an understanding of culture layers increase the understanding of how an agile culture could be established (Tolfo, Wazlawick, Ferreira, & Forcellini, 2011). However, the research within this sub-field lacks an analysis what happens if a hard (process) solution is used for soft (cultural) problems, something that needs to be realized in research on agile software development and through that give credibility to practitioners working on these aspects. Strategies already exist in other fields that could be directly implemented in the software engineering context and research can then focus on specific areas where the software engineering field is different and not reinvent the wheel.

1.1. Organizational Culture and the Iceberg

A well established model is to view organizational culture as an iceberg. Above the surface is the external (surface) culture and is only representing 10% of the organizational culture. These behaviors, traditions, customs, and structures are explicitly learned, conscious, easily changeable and constitute objective knowledge. However, 90% of the culture (which is often ignored to a large extent) is below the surface. Here we have habits, core values, beliefs, priorities, politics, attitudes, perceptions, and assumptions which are all implicitly learned, unconscious, difficult to change, and constitute subjective knowledge (French & Bell, 1999). Most of the organizational change initiatives fail due to lack of these under-the-surface aspects of an organization of people (Strebel, 2009).

1.2. Agile Methods Over and Under the Surface

The organizational (or cultural) iceberg metaphor is highly relevant for agile management just like any other human endeavor. To only focus on process, no matter if it is on waterfall methods or agile practices, is to only look at the peak of the iceberg. There are some research done in agile software development that point to this mistake (see e.g. Ranganath (2011)), but also a study showing the cultural anchoring needed when implementing such practices successfully (Sharp & Robinson, 2004). The agile community often separate agile principles from agile practices. The principles are the values and culture of being agile and is often directly cited. These have become a popular way of defining “agility” and are as follows (http://agilemanifesto.org/): (1) Our highest priority is to satisfy the customer through
early and continuous delivery of valuable software. (2) Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage. (3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. (4) Business people and developers must work together daily throughout the project. (5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. (6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. (7) Working software is the primary measure of progress. (8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. (9) Continuous attention to technical excellence and good design enhances agility. (10) Simplicity—the art of maximizing the amount of work not done—is essential. (11) The best architectures, requirements, and designs emerge from self-organizing teams. (12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

With the lens of the iceberg metaphor it is clear that many of these principles are a dive under the surface. This metaphor makes it clear what the differences are between agile principles (being agile) and agile practices (doing agile). This is also connected to the lack of need for agile maturity models for practitioners (Fontana, Reinehr, & Malucelli, 2014). Tolfo et al. (2011) also found it useful to sort organizational observations to further understand agile transformations in companies. Practitioners need tools for dealing with culture and not only structures for measuring it. The more formal methods focus on top-of-the-iceberg aspects whilst other models (like Sidky, Arthur, and Bohner (2007)) blend agile principles and practices in their assessment. Also, according to Williams (2012), 64.6% of 326 experienced agile practitioners stated that the reason why agile principles are valuable is: “because all agile teams choose among software development practices, but, if they want to be agile, they should choose practices that are in line with the principles”. So we need to start with a below-the-surface plan and then select practices to support those organizational changes. If we try to change the culture by changing visible processes (or in other words, using “hard” solutions for “soft and messy” problems) we will surely fail (Strebel, 2009). We will now describe what this means in more detail.

2. Discussion on Hard and Soft Solutions in Connection to Agile Development

There are mainly two different aspects of any organizational change, the content of change and the context where it happens. The management ideas are often believed to be generic and we are taught to see the similarities of all types of organizations, instead of their differences. We think of a world full of organization instead of unique operative units. In order to translate organizational ideas they have to be “decontextualized” and contextualized again in a new organization. A key to implement new management ideas is to have what Røvik (2011) calls translator skills. One must have knowledge about the context in which one tries to implement new methods. A great problem when using generic methods is that focus always is on mean value of success. This knowledge says very little about how well a method works in one specific case. This is also a reason why organizations often measure their agility in their own adapted way, i.e. the measurement models do not take the context into account enough (Jalali, Wohlin, & Angelis, 2014).

What is a hard solution to a soft problem? In order to understand the effect of this, we must define what the differences are. The spectrum and difference are well described by Paton and McCalman (2008) and can be seen in Table 1. In fact, a hard problem solution is based on systems engineering and older management ideas (Paton & McCalman, 2008) just like traditional waterfall methods. A software engineering process change is far from that clear to the organization. Also, the cultural changes described in the Agile Manifesto and research conducted by e.g. Ranganath (2011); Tolfo et al. (2011) shows that agility is undeniably a soft issue as well as hard.

Hard problems solution can be, for example, a simple change in an IT support system used by employees or acquisition of equipment or maintenance of supplies needed for the work place. A hard systems methodology of change usually has a description phase (with an analysis of the situation, identification of
Table 1 – The TROPICS test.

<table>
<thead>
<tr>
<th>TROPICS factor</th>
<th>“Hard” problem</th>
<th>“Soft” problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timescales</td>
<td>Clearly defined: short to medium term</td>
<td>Ill defined: medium to long term</td>
</tr>
<tr>
<td>Resources</td>
<td>Clearly defined and reasonably fixed</td>
<td>Unclear and variable</td>
</tr>
<tr>
<td>Objectives</td>
<td>Objective and quantifiable</td>
<td>Subjective and visionary</td>
</tr>
<tr>
<td>Perceptions</td>
<td>Shared by those affected</td>
<td>Creates conflict of interest</td>
</tr>
<tr>
<td>Interest</td>
<td>Limited and well defined</td>
<td>Widespread and ill defined</td>
</tr>
<tr>
<td>Control</td>
<td>Within the managing group</td>
<td>Shared out with the group</td>
</tr>
<tr>
<td>Source</td>
<td>Originates internally</td>
<td>Originates externally</td>
</tr>
</tbody>
</table>

objective and constraints, and how to measure a successful change are included), options phase (evaluate different option compared to the performance measure), and finally, an implementation phase (carry out and evaluate the changes with given measures).

The solution to soft problems can be called an OD (Organizational Development) approach. An almost too accurate description of OD to agile development processes was written by French and Bell (1999) (first edition out in 1973!). They describe the OD process as: “A long term effort, led and supported by top management, to improve an organization’s visioning, empowerment, learning and problem-solving processes, through an ongoing, collaborative management of organization culture – with special emphasis on the culture of intact work teams and other team configurations – using the consultant-facilitator role and the theory of technology of applied behavioral science.” All the agile development lessons learned from success stories the last decade are there. The long term cultural effort, the executive buy-in, the empowered team members, the collaborative team environment, the facilitator role, and the recent findings of the usefulness of applied behavioral science (Iivari & Iivari, 2011; Tolfo & Wazlawick, 2008).

The OD approach is considering the whole system as well as its parts. The first step is then to diagnose the current situation with regards to organizational purpose, goals, structure, culture, prevailing leadership approaches and styles, recruitment practices, career paths and opportunities, reward structures and practices, individuals’ motivation and commitment to their work and organization, employee training and development provision, intra- and inter-group relationships etc. The second step is to develop a vision for change. One does not convince people without meeting them or showing them why our change is important. The vision is the core values and the end goal of the change (the desired state). After this, there is a substantial work with gaining commitment to the vision and the need for change. This is where the hard solution approach fail with devastating consequences. Unless concerned – and to be involved in the process – are consulted and have been a part of the creation of the vision they will have little incentives for “buy-in”. At this state one can not communicate too much because the more information the group members get of what is going on the more will the back up the process. One common mistake is to focus on the people that are against you, or the ones that are not yet convinced. The key to a cultural change is to instead focus on the people that are with you and let them be bearers of the change culture. The forth step is to develop an action plan and have change agents that help the process. In the successful agile examples these change agents (that help with the change and let managers focus on day-to-day issues more) are often called “agile coaches”. To have a process change facilitator is key to success in OD. The fifth step is to implement the change, assess it, and reinforce it. The later means that the change needs to be institutionalized to be long-term.

If there is too much focus on process (no matter if it is on the new agile process or the old waterfall method) the initial problem remains. Replacing a waterfall model by a shallow agile set of practices will not work and the problem has then not been that one did not follow the agile process as strictly as one probably needed. We believe the idea about building flexibility into a process is an old idea from other fields and that agility is about implementing a responsive culture and not about the practices, which are only enablers for responsiveness and not the responsiveness itself.
A reason for why a focus on above the water surface of the iceberg could be that it would seem safer for engineers to work on these concrete areas of concern, which makes the work for the OD change agent more difficult in the IT field. They resemble more the simplified and deterministic systems they were taught in their education (Yu, 2014). One has to be braver when investigating motives, values, cultures, etc. of people in an organization, especially with little training in these topics. This is also why intelligent technical experts sooner or later write books on behavioral subjects (like, for example, coaching (Adkins, 2010)) that are well-known facts to people with a behavioral science background.

The software engineering field should realize that the idea of agile development processes’ responsiveness and flexibility and their complex implementation are not new aspects of human or organizational behavior and theorize around these concepts in the light of what is already know in other fields. That way, the field can focus on researching how/if they differ instead of making the same mistakes others did decades ago.

3. References


The role of Visualisation in the study of Computer Programming

Sarah Alhammad\textsuperscript{1,2} \quad Shirley Atkinson\textsuperscript{1} \quad Liz Stuart\textsuperscript{1}

\texttt{sarah.alhammad@plymouth.ac.uk} \quad \texttt{shirley.atkinson@plymouth.ac.uk} \quad \texttt{L.stuart@plymouth.ac.uk}

\textsuperscript{1}School of Computing, Electronics and Mathematics, Plymouth University
\textsuperscript{2}College of Computer and Information Sciences, Princess Nourah bint Abdulrahman University

\section*{Abstract}
This paper presents a study of how visualization tools and methods are used to support the study of computer programming. The purpose of this study was to establish whether visualization could be exploited more fully or more effectively to support this learning. Furthermore, if that were the case, then this study would aim to define the requirements of novice programmers. Greater understanding of the requirements of novice programmers is expected to steer this research towards finding better solutions to the challenges faced by students of programming.

The paper presents a study of three distinct visualization tools Jeliot (Moreno, Myller, Sutinen, & Ben-Ari, 2004), Online Python Tutor (Guo, 2013) and Visual Logic (Gudmundsen & Olivieri, 2011). The nature of how visualization is exploited in each of these tools is evaluated. This evaluation is based on how the tool visualises the execution of (i) loops, (ii) object-oriented programming and (iii) the parameter –passing by value/reference. These three problems are a sample of problems that students find complex when writing programs (Eckerdal et al., 2006; Boustedt et al., 2007; Sanders et al., 2008).

The results presented in this paper show the characteristics in the visualization tools and the students’ preferences in using them. The students evaluated the following eight characteristics: tool availability, error explanation, expression evaluation, the interface, programming languages the tool supports, animation, class hierarchy and save the execution history.

\section{1. Introduction}
Technology is central to the way we live our lives these days and programming is the “engine” behind this extensive use of technology. Many students are keen to study programming. However, one of the challenges when teaching programming is helping students to understand traditional static program code maps designed to serve as a dynamic representation of program memory (Ala-Mutka, 2004). Teachers often try to illustrate the effect of code on memory simply using paper and whiteboards. However, this can be time-consuming, and it often lacks interaction with and engagement of the student.

Visualisation tools are used in many academic institutions to support learning in many diverse areas. Programming is no exception. Indeed, it is common for teachers and students to use visual tools to trace the execution of a program as well as showing the outcome from each statement of code as it is executed. The use of visualisation tools is an alternative to the use of paper and whiteboards to demonstrate the execution of the program. However, visualisation tools deliver a more effective understanding of the execution as the students can interactively “see” exactly what changes are happening in the program memory as each line of code executes.

As the research domain of visualisation expands, more visualisation methods are emerging which present the teacher with an alternative to manual program tracing. These visualisation tools are aimed at supporting novice programmers as they learn to program (Baldwin & Kuljis, 2000,2001; Kasurinen, Purmonen, & Nikula, 2008; Salcedo & Idrobo, 2011). This research focuses on tools which visualise the program memory and enable the students to step-through the code, watching the changes in the memory as each line executes. Note that it is important to establish the effectiveness of existing visualisation tools to fully appreciate the user requirements.
This work is organized as follows: First, section 2 views studies that have been conducted to describe the challenges of learning programming. In section 3, we discuss the educational visualisation method for learning programming. Section 4 is concerned with the threshold concepts in programming. The sixth section discusses the experiment and its results. Finally, the last two sections are about limitations of the study and the conclusion.

2. The challenges of Learning Programming

Numerous studies have investigated the difficulties in teaching and learning computer programming. For instance, Milne and Rowe (2002) conducted an experiment that demonstrated the inability of a student to absorb what is happening in the memory of the program. Another study (Husain, Tarannum, & Patil, 2013) was conducted to enhance students’ problem-solving skills and improve the learning and teaching of programming. That study showed that even better students (one with higher grade point average -GPAs) struggled to apply the programming concepts they have learned in lectures. The two most challenging factors for those students were (i) in the design and (ii) in the implementation phases when they started writing programs.

There is high demand for innovations that support the teaching of programming and deal with these many inherent challenges faced by both teachers and students.

3. Visualisation

Research studies have demonstrated the advantages of using visualisation when learning to program. Naps et al. (2003) define the visualisation in education as engagement taxonomy which means demonstrate the lecture using visualisation as viewing engagement. Kelleher and Pausch (2005) describe how the visualisation environment contributes in lowering the barriers to programming such as Mechanical and Sociological barriers. Kasurinen et al. (2008) introduced the approach of using visualisation in teaching programming for the purpose of increasing motivation and decreasing students’ failing or dropout rates.

Many visualisation tools have emerged with the aim of supporting novice programmers as they learn to program. Examples include BlueJ (Hagan & Markham, 2000; Kölling, Quig, Patterson, & Rosenberg, 2003), DrJava (Allen, Cartwright, & Stoler, 2002), ProfessorJ (Gray & Flatt, 2003), Jeliot3 (Moreno, Myller, Sutinen, & Ben-Ari, 2004), VIP (Virtanen, Lahtinen, & Jarvinen, 2005), Web Tasks (Rößling, 2010), Alice (Salcedo & Idrobo, 2011) and Online Python Tutor (Guo, 2013).

Support tools often adopt visualisation to help students overcome the challenges of learning programming. Approaches which visualise code and/or use diagrams to represent the program memory have a vital contribution to the process of teaching and learning how to program.

3.1 Visualisation tools used in this study

The three sample visualisation tools used were Jeliot, Online Python Tutor and Visual Logic. These tools all use memory referencing to show the effect of execution of each line of the program memory. However, each tool has its own features. The selection of each tool was done specifically in order to ease the comparison of student feedback. However, the tools differ in many aspects such as the way in which they trace through the code, how they represent the output, how they create error explanations, the programming languages they support and whether they are available online or offline.

Jeliot is a visualisation tool for object-oriented programming. It provides dynamic visualisation for tracing the program execution (see figure 1). Jeliot builds a model of the program during execution. Therefore, students can understand the program construction. The Jeliot interface consists of four areas: the ‘method frame’ area, the ‘expression evaluation’ area, the ‘constant’ area and the ‘instance’ area. Each area is used to show the dynamic change of its components (method, expression, constant, instance). The error explanation in Jeliot highlights the line of the error and explains the reason for the error in the ‘error viewer’. Jeliot also has frames where inputs are requested, and outputs are printed (Moreno et al., 2004).
Online Python Tutor is a web-based programming tool which uses visualisation extensively. It is open-source software where the user embeds their code into the web page on the left. Subsequently, the code can be traced through using navigation buttons (see figure 2). The visualization of the code is shown on the right. This visualisation enables the user to watch the dynamic execution of the program. As the program executes it depicts changes to frames and objects. Additionally, it has a program output area. The tool provides explanations of errors, with indicators pointing to the line on which the error occurred (Guo, 2013).
Visual Logic uses the concept of iconic programming (icons and flowcharts) to visualise the program. Visual Logic has no code to be written (see figure 3). Instead, the user creates the flowchart which represents the code. Subsequently, the tool traces the flow of it. The tool demonstrates the outcomes of ‘executing’ each icon in the flowchart in popup windows. Visual Logic does not support object-oriented programming (Gudmundsen & Olivieri, 2011). However, the researchers have chosen the tool to introduce the use of flowchart in the tracing and observe whether this method will admire the students.

4. Threshold concepts
Meyer and Land (2003) developed the idea of the threshold concepts in 2003. Threshold concepts are theories that link subjects together and are essential for mastering how to think and practice in the discipline. Meyer and Land (2003) defined the characteristics of threshold concepts that can be used to evaluate any scientific concepts, whether they are threshold concepts or not.

The characteristics of threshold concepts according to Meyer and Land (2003) are the following:

- Transformative: bringing about an alternate way view of things to students in the discipline.
- Integrative: linking the concepts together and exposing the interrelatedness.
- Irreversible: shifting a student’s perspective and unlikely to be forgotten unless spending efforts.
- Troublesome: being hard for students to learn.
- Limited Boundary: helping students stay within the boundaries of the field and the concepts that belong to it.

The experiment, as reported in this paper, focuses on three threshold concepts used in programming. These are the use of loops, object-oriented programming and the parameters. Many research studies have investigated these concepts and determined that, among students and teachers, they are the most popular threshold concepts in computing: Eckerdal et al. (2006) found the most popular threshold concepts, among teachers, were the program’s abstraction, pointers, objects, classes, instances, and recursion. In contrast, Boustedt et al. (2007) argued after the study that they had evidence that two concepts in particular—object-oriented programming and pointers—satisfy the requirements for threshold concepts. Note that Boustedt et al. also agreed on five troublesome areas of programming. These are: memory model, objects, control statements (loops), parameters, and sequential thinking.
Moreover, the findings of Sanders et al. (2008) and Bühlmann (2011) highlighted the difficulty which students have with separating the concepts of ‘class’, ‘object’ and ‘instance’ and with understanding the inheritance concept.

5. The study

5.1 Research methodology
This research aims to evaluate current visualisation tools used to support novice users learning to program. In particular, it will focus on identifying the strengths and weaknesses of these tools. This research was conducted on computer science students to gather information about their experience of using visualisation tools, to understand their needs better and to gather their opinions about these tools.

This study involved the use of semi-structured interviews with 20 participants, over 22 years of age. The participants were either undergraduates (second year and above) or students who had recently graduated. The interviews have been conducted in Saudi Arabia and translated to English after recording them. During interviews, it was verified that each participant had studied programming concepts in depth including the core aspects of the use of a loop, object-oriented programming and the parameters. The aim of the interview was to understand the learners’ experience and to establish their background in using visualisation tools to support the process of learning to program.

Three tools were presented to the students as a sample of visualisation support tools which are available. These tools were Jeliot, Online Python Tutor and Visual Logic (as discussed in section 3.1). The students discussed the tools, their usability, and determined the strengths and weaknesses of each tool. The tools were compared on the basis of their usefulness in solving programming problems. The programming problems were selected based on the threshold concepts, including object-oriented programming, loops and parameters. Opinions were also obtained on their assessment of the best and worst tools.

5.2 Tools selection
The majority of the students were expected to have little or no experience using visualisation tools. The researcher used three samples of visualisation tools: Jeliot, Online Python Tutor, and Visual Logic. The tools all share the common method of using memory referencing to show the effect of execution on each line in the code. However, each tool has its own specific features. The tool selection was done carefully to make the comparison of students’ points of view clearer. The tools differ in many aspects, such as in the way they trace code, how they represent the output, how they create error explanations, programming languages they support and whether they are available online or offline. Moreover, the tool selection was done based on the students’ experiences. The tools are designed for novice programmers, which were suitable then for our participants, and require no training or special skills.

5.3 Example problems
The problem areas considered in this study are the three programming concepts described earlier as threshold concepts. Many studies (Boustedt et al., 2007; Bühlmann, 2011; Eckerdal et al., 2006; Sanders et al., 2008) have shown that loops, object-oriented programming and the parameter are threshold concepts that students find difficult to understand. Therefore, each of these core threshold concepts is embedded into the three problems investigated in this study.

5.3.1 Example problem 1 (the loop)
For this problem, Program1 was coded. Initially, this program populated an integer array data structure of five elements with predefined integer values. The overall aim of Program1 was to evaluate the average value of the integers stored in the array. To achieve this, Program1 established the number of elements in the array. Then, the program used a ‘FOR loop’ to calculate the sum total of all the values. Subsequently, the average was calculated.

5.3.2 Example problem 2 (the object-oriented program)
For this problem, Program2 was coded. The program used the inheritance from the superclass (polygon) to calculate the area of the subclass (square). The main program defines an instance of the
class square and passes the length of the side to the method ‘get_area()’, which is inherited from the superclass (polygon). The area was calculated by the inherited method “get_area()” and the result will be inherited to the subclass (square). Finally, the result printed in the main program as an element of the instance from (square) class.

5.3.3 Example problem 3 (the parameter – passing by value/reference)
For this problem, Program3 was coded. Problem 3 is used to highlight the difference between passing parameters by value and passing by reference. The main program defined variable (parameter) as a global variable and send its value to the procedure (change_value) as pass by value and not by reference. The value of (parameter) has been changed locally in the procedure but the change has no effect on the (parameter) in the main program. The variable was printed in the main program to show whether or not the value of the variable had been changed.

5.4 Data Analysis Strategy
The researcher’s intention was to collect feedback about the existing visualisation frameworks to determine their benefits to develop a new framework. The qualitative data support the research to focus on the basic features and aspects that novice students need on the educational tools.

This study analysed the coding process through initial and final coding using Cycle Pattern Coding Method. This method of coding was chosen to find the keywords and group the features. Text analysis was used to conduct the first cycle of coding to determine phrases that were common amongst interviewees. Phrases appeared as word clouds, which were analysed and encoded with suitable category labels. Next, a Second cycle of the Cycle Pattern Coding Method was used to recognize similarly coded data and further summarize it into subcategories or consolidate it. Finally, findings were narrated as they related to the implications of the study. For more details of the method see (Strauss & Corbin, 2008) and (Saldaña, J. 2015).

6. Results
During the semi-structured interviews, a substantial amount of data was recorded from the participants including what they liked and disliked about each tool. During the analysis of this data, it was realized that the majority of this feedback was concentrated around eight core characteristics of support tools. All of these characteristics were features of the presented tools. Therefore, the researcher adopted this list of eight characteristics as a means of categorising the analysis of the data.

The list of characteristics was

1. Controlling the execution of the code
2. Availability of the tool (online or offline)
3. Error explanation
4. Interface/usability of the tool
5. Programming languages supported
6. Expression evaluation
7. Representation of Class hierarchy
8. Maintaining an event history.

The following explains how the participants described the tool’s characteristics. The number of respondents (n) is indicated in the headings.

6.1 Controlling the execution of the code (n=18)
The analysis stage of this study has revealed that the manner in which the participant was able to control the execution of code was significant. When this study was designed, careful attention was given to the selection of which support tools to include in the study. The outcome of this selection was that the tools chosen represented a variety of mechanisms for the control of execution. Furthermore, the participants realised that there were different kinds of execution control and they subsequently used them to compare the tools.
During the interviews, the participants used a variety of phrases to describe the control of execution. These phrases included: the animation, the control buttons, the execution speed, tracing and visualisation.

Participant 5 complained about the inability to have full control of the execution, such as the ability to go back and forth with each statement. Participant 5 said: ‘The execution control is not enough, I prefer if I can control the execution more, for example, going back and forth for each statement.’ Conversely, Participant 19 appreciated the animation used to visualise the execution. They stated: ‘I like the animation used to visualise the execution’.

Regarding the speed of the execution and the ability to control it, Participant 7 suggested a slower speed: ‘I want to slow down the execution speed’. Furthermore, Participant 11 suggested an overall more flexible execution process which gave more control to the user. Participant 11 stated: ‘I like the control buttons that make the control of execution more flexible.’

The interviews demonstrated that the students clearly understood the importance of how the visualisation was being controlled. The majority of the students, 90% of them (18 out of 20), reported that they preferred to have precise controls for the line by line execution of the code and it’s visualisation.

However, these preferences did vary. Of these 18 participants, 7 participants appreciated the code animation which had few controls. Conversely, 5 participants preferred the user to have complete control using the control buttons. The remaining 6 participants (30% of the participants) suggested that a mixture of both methods would be the optimal solution. This was based on the fact that the type of control required was about the length of code involved and the user’s familiarity with the code.

6.2 Availability of the tool (n=16)
Secondly, the data analysis revealed that establishing whether the tool was available online and/or offline is a relevant aspect to participants. It was clear that this characteristic would influence their choice of tool. However, for some participants the online availability of the tool was a genuine concern, as shown in the feedback:

- ‘I do not like the online tool, because of any internet issues or problems’ [Participant 1].
- ‘I don’t like the online tools to avoid the connection problems’ [Participant 7].

Conversely, for others, online availability was assessed as a positive aspect of the tool: ‘I like that the tool is online tool’ [ Participant 17].

Thus, answers given by participants did vary on this characteristic. In total, 16 of the 20 participants highlighted this characteristic as important. Of these 16, 3 expressed a preference for the tool to be downloaded to their devices. They explained that their preference was due to concerns about their Internet speed and connection. Conversely, 9 participants preferred the online mode, explaining that they would have access to the tool at anytime and anywhere. The remaining 4 participants preferred the option of a mixed mode (combining both online and offline) where the user can choose the mode required.

6.3 Error explanation (n=13)
Thirdly, the data analysis revealed that the explanation of errors and support for debugging are important aspects to participants. The participants reported that they required (i) support for finding an error, (ii) a meaningful explanation of the error and (iii) a suggestion of a suitable means to correct the error. They highlighted that ambiguity in the explanation of the error may impede their understanding of the cause of the error in code as shown in the following feedback:

- ‘The error explanation was not good’ [Participant 10].
- ‘The correction suggestion for the error is not helpful enough’ [Participant 17].

Subsequently, this may have a negative effect on their learning. When the participants were asked about the reason for their opinions, they said that they could not understand the cause of the error and that the suggestions were ambiguous. Therefore, clarity in error explanation and removal are a high
priority. In total, 13 of the 20 participants highlighted this characteristic as important. All of these 13 participants expressed a preference for a support tool which explains the error in detail and provides a variety of ways to correct the error.

6.4 Interface/Usability of the tool (n=13)
The data analysis revealed that the style and appearance of the interface were another important aspect to participants. The participants commented on the font type, size and colour used in the text: ‘The code font is small’ [Participant 9].

Some of the feedback gathered from the participants was about how the windows and how the tool presented the final output: ‘I like the appearance of execution in Jeliot which is next to the code directly’ [Participant 1].

The participants also gave varied comments about the use of an indicator icon to point out the statement currently being executed in the Online Python Tutor. For instance, Participant 2 stated they liked ‘The use of green and red pointers (arrows) to indicate the statement execution whether it is under execution or will be the next statement to be executed’. This opinion was reinforced by Participant 6 who agreed ‘I like the use of red and green arrows as an indication of statement execution.’

Out of the total number of 20 participants, 13 participants (65%) reported on the interface and its usability from different perspectives.

Note that they used different terminologies to describe their opinion of the interface of each tool including font, colour, window, ease of use. In conclusion, it is clear that the colour, the font, and the windows’ appearance played a significant role in attracting the students to use the tool.

6.5 Programming languages supported (n=9)
Next, the data analysis revealed that both the number and range of programming languages supported by a tool is important to participants. Again, the participant’s feedback was mixed. Some participants preferred to use multi-programming language tools while other participants expressed a preference for a bespoke tool for each programming language.

- ‘I don’t like that Jeliot supports only one programming language’ [Participant 10]
- ‘I like the Online Python Tutor tool because it supports more than one language’ [Participant 18]

Out of the total number of 20 participants, 9 participants (45%) recommended a tool which could be used for multiple programming languages. The remaining 11 (55%) reported that the choice of programming language was not a huge issue to consider and that supporting only one language is enough when it comes to using the tool.

6.6 Expression evaluation (n=5)
Next, the data analysis revealed that details on how and when an expression is evaluated are important aspects to participants. The participants reported that they had not recognised when expressions were being evaluated, although there was a variety of feedback on this characteristic.

For example, in Program1 where the condition of the loop is evaluated, the participants wanted to ‘see’ the working out of evaluating the condition of the loop at every iteration.

- ‘The design needs to be developed, similar to the manual tracing, and the absence of statement expression’ [Participant 4].
- ‘One of the strengths of the tool is the existence of expression evaluation’ [Participant 18].

Participants reported that this evaluation of expressions was primarily used to understand ‘what’s going on.’ Clearly, this is a vital aspect for increasing the learners’ comprehension.
In total, 5 of the 20 participants highlighted this characteristic as important. The participants reported that they relied on the expression evaluation for evaluating the condition of control statement. All of the participants who reported the use of the expression evaluation agreed that they liked it because it is similar to what they had done on manual tracing. Therefore, there was a requirement to present the user with the automated version of what they would otherwise do manually.

6.7 Representation of Class Hierarchy (n=2)

The analysis stage of this study has revealed that the manner in which the tool represented the class hierarchy was critical to users. Example problem 2 presented the participants with code from an object-oriented program. That program is an example of the concept of inheritance as the program inherits variables and methods from a predefined superclass. The participants assessed how well each tool represented the classes and their hierarchy. They also reported how much the tool aided their comprehension of the classes in the code.

For example, Participant 1 made the following statement about Jeliot stating ‘in case of representing the classes and inheritance, it is not clear because the classes cascaded and not represented as hierarchy.’ Participant 3 was agreed ‘There was no weakness in the tool, but then went onto to highlight one exception ‘its way to represent the class hierarchy and the variables and methods which are inherited or private all of these was not satisfied.’

6.8 Maintaining an event history (n=2)

The final characteristic of support tools which was identified as important by study participants was the tools’ ability to record the events (a history) of everything that took place during execution. The participants used the phrase ‘save history’ to describe their requirement for the system to automatically generate a list of events which took place throughout the whole process of execution.

Participant 2 complained about the tools’ being unable to save the history. Participant 3 also supported this by indicating that they would also like a tool that keeps results for each execution and does not omit events. Based on this feedback, it is clear that users place a high value on having this history of events to trace through after execution.

6.9 Tool comparison

The final part of this study focused on a comparison of the three tools. This comparison was based on how each tool visualised the execution of the three problems (mentioned in section 5.3) and then gathering the participants’ opinions about which tools they preferred.

The students excluded the Visual Logic tool from the comparison since it received a lot of negative feedback on its usefulness for support students learning to program. It was a clear outlier that they were not interested in pursuing. The reason for this negative feedback was that Visual Logic does not include any program code and that it relies too heavily on tracing through the flowchart.

Regarding example problem 1, solving the FOR-LOOP problem, all of the 20 participants preferred the Jeliot tool. The reason for this unanimous decision was that Jeliot has ‘expression evaluation’. This ‘expression evaluation’ feature shows the evaluation of the condition of the loop for each iteration. Furthermore, it shows the loop counter and the loop body. Participant 3 stated: ‘I choose Jeliot, in case I am novice programmer because it has the feature which is expression evaluation.’ Furthermore, Participant 4 stated: ‘Jeliot was clearer than the other tool, I like its way on how it shows the expression evaluation to clarify the loop counter, loop condition, and the body of the loop.’ These opinions were reinforced by other participants:

- ‘Jeliot was clearer because it represents the dynamic change on the loop counter and condition’ [Participant 7]
- ‘Jeliot is better because the loop was clearer in how the counter change each time and how the loop condition checked every time’ [Participant 15]
Regarding example problem 2, based on object-oriented programming, 7 participants expressed a preference to use Jeliot. They attributed this choice to the fact that cascading the classes is more effective than representing them on a hierarchy. This is particularly the case when there are numerous classes where the screen space will be insufficient to demonstrate the class hierarchy. Participant 1 said: ‘Jeliot is better in case of long code and plenty of classes because the class cascaded which make space add classes.’ Participant 4 said: ‘I prefer Jeliot because the way on how representing the classes were clear specifically if the number of classes was large.’ Conversely, the majority of the participants (11 out of 20) preferred the Online Python Tutor tool as it represents the classes sequentially. Therefore, the participants reported that they ‘could see’ the variables and methods of each class. Participant 10 said: ‘Online Python Tutor because the classes were presented sequentially so it is clearer than representing the classes on Jeliot’. Participant 18 reported that: ‘Online Python tutor is better because I like how it is drawn boxes sequentially for objects and use arrows to represent the references.’ Only two of the participants were impartial. Those two participants stated that both class representations were clear and helped them to understand the classes inheritance equally.

Finally, regarding example problem 3, the researcher asked the participants to compare the tools when calling procedures and passing parameters. The vast majority of the participants, 18 out of 20, agreed that the Online Python Tutor tool was best. The reason they gave was that the process of parameter passing was significantly clearer than in the Jeliot tool.

There were different opinions based on the clarity provided by the Online Python Tutor tool. For instance, Participant 2 stated that ‘Online Python Tutor is clearer on how the transition done from the main to the procedure and how it affect the value of parameters.’ Participant 11 said: ‘Online Python Tutor was clearer when calling procedures and giving the returning value, it always writes what is the return value from any procedure even if its “void”.’ However, the remaining two participants said they understood the problem equally when using both Online Python Tutor and Jeliot.

7. The limitations of the study
The study has explored three threshold concepts while there are other threshold concepts which could be explored such as pointers and recursion. Building a visualisation tool to support learning programming should probably account for more than the three concepts.

Moreover, some of the characteristics that have been found in the study mentioned by very few participants for example expression evaluation, representation of class hierarchy and maintaining an event history. Whilst these characteristics do seem fairly important for a learning system. The claim in the study suggesting that they are critical to users should be weakened however we need more supporting data for this.

8. Conclusion and future work
The aim of this study was to assess how well current visualisation systems support students as they learn to program. Overall, the coverage of threshold topics is variable between systems. This study identified a set of 8 core characteristics used to assess support tools. In essence, the following list is essential requirements for tool support in this area. These requirements include a tool which:

- is highly flexible and enables the user to use it for 1 or more languages (as required)
- is available online and offline
- is intuitive based on good design of the interface which delivers a usable product
- has a clear evaluation of expressions in the code (as in a manual trace)
- has a clear, meaningful and helpful explanation of errors with support for error correction
- has an intuitive control of the execution of the program code,
- provides an clear intuitive representation of the class hierarchy which can be compacted when a number of classes is large
• maintains an event history for the user to trace through and search.

Future work in this research aims to develop a software tool based on the findings of this study. The aim is to learn from existing systems and to develop a more complete, flexible tool for supporting students who are learning to program.

Finally, there are some factors that could serve as implications for the study. Firstly, the data collection period was time limited, and due to the fact that the research method involved semi-structured interviews (as opposed to a case study), the interviewers had a limited period to gather students’ opinions. To perform a more in-depth evaluation, the students would need a longer time to practice using the tools and becoming familiar with them. Some of the participants in the study wanted to use the tools for longer periods in order to provide more accurate feedback. Secondly, the researcher aims to extend the study to cover a variety of international sites. Different kinds of cultures and different academic institutions, outside of the Saudi Kingdom, may enrich the study with valuable results and findings.

9. References


A fox not a hedgehog: What does PPIG know?

Luke Church  
Computer Laboratory  
University of Cambridge  
luke@church.name

Mariana Mărăşoiu  
Computer Laboratory  
University of Cambridge  
mariana.marasoiu@cl.cam.ac.uk

Abstract
We outline a thematic history of the Psychology of Programming Interest Group based on a coding of [~400] publications. We highlight the changing interests of the community, draw out trends and discuss missing topics. We compare and contrast with [~50] publications from the PLATEAU community. We find that, fox-like, PPIG has a very broad coverage of a wide range of topics. We characterise trends in these and go on to discuss missing topics and areas for future work.

1. Introduction
How people go about programming and the how to designing systems that support them is of growing interest once more to both the HCI and technical communities.

Victor (2012) helped restart the collective imagination of building live, usable, programming systems. Work on live programming has continued within the software engineering communities at LIVE at ICSE¹ and ECOOP², within the live coding scene via TOPLAP³, the Dagstuhl on Live Coding (Blackwell et al., 2014) and the new conference series ICLC⁴. There are also other meetings interested in specific programming technologies, such as the workshop on block based languages (Turbak et al., 2015).

Further, as programming has become a required element of the curriculum in the United Kingdom there has been substantial interest via the Computing At Schools project (Peyton Jones, 2015).

Simultaneously, there has been a growing interest in the experience that developers have via workshops such as LIXD⁵, and the new PX⁶ workshop at ECOOP. There was recently a well attended group at the Special Interest Group of CHI 2016 on the Usability of Programming Languages (Myers et al., 2016).

Matters that concern the PPIG and PLATEAU (Sunshine et al., 2009) communities are on the rise again. At the same time, what constitutes knowledge within the communities is increasingly contested. For example, a substantial theme of discussion within the SIG-CHI was related to the work of (Stefik et al., 2014), where they conclude that little of the work in the PPIG and PLATEAU groups is about programming language design and meets their selected standard for empirical research (1.1% and 14.3% of papers respectively, within the constraints explained in their paper).

This leads to the question: if these communities haven’t had much to say in the past about high quality randomized trials of programming language design, what do they talk about? In this work we explore this question.

¹ http://liveprogramming.github.io/2013/  
² http://2016.ecoop.org/track/LIVE-2016  
³ http://toplap.org  
⁴ http://icl.livecodenetwork.org  
2. Analytical Methodology

2.1 The Corpora

We analyse papers available in digital form published at the annual workshops of the Psychology of Programming Interest Group (PPIG) and at the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU). Table 1 shows the number of papers for each conference that were analysed, organized by year.

<table>
<thead>
<tr>
<th>Year</th>
<th>PPIG</th>
<th>PLATEAU</th>
</tr>
</thead>
<tbody>
<tr>
<td>'92</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>'93</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>'94</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>'95</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>'96</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>'97</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>'98</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td>'99</td>
<td>26</td>
<td>-</td>
</tr>
<tr>
<td>'00</td>
<td>21</td>
<td>17</td>
</tr>
<tr>
<td>'01</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>'02</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>'03</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>'04</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>'05</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>'06</td>
<td>17</td>
<td>-</td>
</tr>
<tr>
<td>'07</td>
<td>17</td>
<td>-</td>
</tr>
<tr>
<td>'08</td>
<td>22</td>
<td>-</td>
</tr>
<tr>
<td>'09</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>'10</td>
<td>26</td>
<td>11</td>
</tr>
<tr>
<td>'11</td>
<td>21</td>
<td>-</td>
</tr>
<tr>
<td>'12</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>'13</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>'14</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td>'15</td>
<td>24</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 1. Number of papers published and analysed from each conference, grouped by year.

There are 400 papers from PPIG and ?? from PLATEAU.

We make no distinction between ‘work in progress’, ‘graduate consortium submissions’ and ‘full papers’, but didn’t include the work in progress workshops of PPIG. The purpose of this analysis is to reflect on the communities interests, not quality control.

2.2 Data analysis

We used thematic analysis methods (Braun & Clarke, 2006) which we adapted to document analysis in order to generate topics. The analysis proceeded as follows. One author read through the titles of the papers in the corpora and extracted a set of research topics, which were discussed with the other author. These topics were then organized in larger themes by both authors. In the next step, the authors then independently read through the text of the papers with the dual purpose of assigning topics and generating new topics when necessary. Each paper was assigned multiple topics.

Since the coding scheme evolved in time, the corpora was searched again for some of the newly added codes as an extra check to ensure that the papers were labeled correctly. Papers found this way were reread and assigned potentially assigned the new code.

We only considered a code for the paper if it was the central theme, e.g. a language, for a programming language: where its features were discussed? Was it used in empirical or analytical studies described in the paper? We didn’t consider it as a topic if it was mentioned in passing, or if it was mentioned as the language that the system described was implemented in.

At this point an inter-rater reliability score for a 5% sample was computed giving a Kappa of 0.84, indicating broad agreement between the reviewers.

In order to improve the analysis, we then discussed the differences in the codes to reach consensus. Typically, we found this was resolved by adding codes to papers and occasionally by refining the coding scheme itself. Some of the outcomes of the discussions are included in a later section. Whilst this processes decreases the objectivity, it also decreases the likelihood of accidental oversights. In practice the authors found that accidental omission was by far the most common reason for the addition of new codes to a paper during the discussion phase.

2.3 Notes on the coding

We elected to engage in this coding activity rather than to use the keywords that the authors have tagged their work with due to many papers not having keywords and to allow a somewhat independent reflection on what the papers were about.

We initially intended to use Latent Dirichlet Allocation (LDA) (Blei et al., 2003) to extract topics automatically. This technique was inspired by the work of (Greenberg et al., 2015) in their analysis of
the programming languages literature. However, the application of LDA resulted in poor topics for the PPIG corpus. Consequently we elected to identify and code the topics manually.

Both authors have developed and used open coding schemes in several analyses in past, but were struck by the difficulty of coding the PPIG papers. Over the 23 years covered in the analysis, the PPIG community does not seem to have converged on any particular strategy of narrative structure, and the lexicon for describing topics is still somewhat undecided. Moreover, the range of technologies, aspects and domains covered within the conference is very broad.

This intellectual diversity acts as a core strength of the community but, as we shall see, raises some challenges in the development of theory. Let’s start by looking at the clearly identifiable themes and patterns.

3. Languages, People, Analyses

3.1 Languages

Looking first at which languages are studied, we see a fairly similar distribution between PPIG and PLATEAU. Java dominates both conferences, and the broader family of C-style languages even more so. There are a wider range of programming languages considered in PPIG than PLATEAU, but this be accounted for by their relative age.

Looking at how interest in the languages studied at PPIG evolved over time, we see a definite transition in [YEAR] when Java replaced a waning interest in Pascal and Prolog. Other languages such as C++ and Visual Basic show continued but lower intensity interest.
There isn’t an easily available baseline to compare these to. There are a number of different schemes for assessing language popularity\(^7\). The TIOBE index\(^8\), which lists languages by popularity, seems in general agreement with the Figure above, except with a considerably higher emphasis on C.

---

\(^7\) [https://en.wikipedia.org/wiki/Measuring_programming_language_popularity](https://en.wikipedia.org/wiki/Measuring_programming_language_popularity)

\(^8\) [http://www.tiobe.com/tiobe_index?page=programminglanguages_definition](http://www.tiobe.com/tiobe_index?page=programminglanguages_definition)
3.2 Who are the programmers?
Looking at the programmers who were being studied at PPIG, we can categorise the papers broadly as either studies of professional or expert programmers (those whose primary occupation is programming), of end-user programmers, (those who are programming to support some other primary occupation), and of novices and those learning to program.

Figure 4. Interest in different programmer types over time

Papers were labeled as any of the three categories only when it was explicit which population was being studied.

Figure 5. The distribution of programmer types being studied at PPIG (A) and PLATEAU (B)
Contrasting the two conferences, we can see that PPIG has been more interested in novice programming and PLATEAU more in professional programmers.

3.3 What techniques are used for discussing these programmers?
We broadly categorised the techniques used into analytical, empirical, qualitative and qualitative. These are non-exclusive labels, meaning that a study can use a combination of these techniques and receive a combination of labels. We can see that analyses within these communities is predominantly empirical (231 papers and 38 papers for PPIG and PLATEAU) as opposed to analytical (29 papers and
5 papers), with a balance of qualitative methods (108 papers and 18 papers) and quantitative methods (138 papers and 24 papers).

The communities start to vary in the techniques they are using for performing the analysis. Considering analytical techniques, there are insufficient PLATEAU papers in this category to make further separation useful. However, within the PPIG community, there are several analytical and theoretical frames that have been used repeatedly, as can be seen in the figure below.

![Figure 6. Theoretical frames and Analytical techniques at PPIG over time](image)

This shows that some analytical and theoretical approaches fluctuate over time (e.g. Cognitive Dimensions, Constructivism) whereas others such as the Mental Models perspective on programming remain relatively steady.

Similarly, we can look at the empirical techniques that are used at PPIG:
Figure 7. Empirical techniques at PPIG over time

We can see that new methods get added (e.g. Grounded theory, eye tracking or corpus analysis). These new techniques don’t displace existing ones but compliment them resulting in an increased range of techniques. PLATEAU has similar characteristics, but with slightly more restricted techniques. The community has focused on programming experiments, corpus analyses, surveys, usability studies and experience reports but, so far, there are no publications using eye tracking studies, grounded theory or ethnographic techniques.
3.4 Which aspects of computation have been studied?

Figure 8. Aspects of computation studied at PPIG over time

Considering computational artefacts the difference between PPIG and PLATEAU is most evident. In the 6 years of PLATEAU, more papers which have been primarily concerned with concurrency and parallelism (7) have been published than in the 22 years of PPIG proceedings. However, in the same time, PPIG has been considerably more concerned with the roles and naming of variables (16 papers) compared to PLATEAU where this has not been a substantial area of study.

As (Blackwell & Morrison, 2010) point out, professional programmers tend to concentrate on semantics and control flow, whereas variable names are a primary concern of end user programmers and those relatively new to programming. As such, the continued interest in variables at PPIG may correspond to a continued interest in end-user programmers and novice programmers as discussed earlier.

3.5 Which aspects of programming have been studied?

As we see, there has been an ongoing interest in program comprehension, debugging and problem solving at PPIG, with some testing and software maintenance publications as well.
3.6 Which types of technology have been studied?

Here we can see the resurgence of interest in visual languages alluded to in the introduction. We also see from this and the studies of aspects above that the PPIG community is interested in programming systems, combinations of notations and the tools that people use to interact with them.
3.7 Which social interactions have been studied?

![Diagram showing social concerns studied at PPIG over time]

*Figure 11. Social concerns studied at PPIG over time*

This shows how some themes within the PPIG are relatively transitory and associated with investigating interests from the broader software engineering community.

3.8 Which factors influencing behaviour have been considered?

![Diagram showing behavioural concerns studied at PPIG over time]

*Figure 12. Behavioural concerns studied at PPIG over time*
Relatively recently there has been a growing interest in non-cognitive psychological concerns at PPIG. These started with discussions of personality influences, but of late self-efficacy has become an increasing concern.

4. Discussion

4.1 PPIG and PLATEAU

Many of the differences between PPIG and PLATEAU stem from the audience being considered. PPIG has substantial End User Programming and educational interests, whilst PLATEAU is more concerned with professional and expert programmers. This leads to different artefacts being studied (e.g. IDEs vs. visual languages) and, to a certain degree, to different techniques. The most notable place where this has shaped the research agenda is PPIG’s interest in variables rather than control flow constructs.

This difference in the study focus of the two cultures is institutionally useful - it allows the conferences to complement each other in the content they cover, whilst the overlap allows for effective exchanges of knowledge between the communities.

This then leads to the question of what hasn’t been talked about much at PPIG. Note that we are not suggesting that these topics haven’t been discussed at all, but instead suggesting that they might warrant more attention than they have received so far.

4.2 Missing groups of users

PPIG as a community has learnt a lot from studying a wide variety of users, including End User Programmers, live coders and even security workers (Biddle, 2014). However these are not the only groups of people who engage in programming-like behaviours (Blackwell, 2002).

There are other notable communities to study. For example, the communities that design ‘high-integrity’ systems have presumably made considerable progress along the axis of quality - what might we learn from them about the psychology of ‘normal’ programming?

Likewise, computer games often include substantial aspects of end user programming, such as the players mentally simulating ‘what might be’, creating artifacts that are repeatedly reused, and managing long range dependencies between their actions and their emergent outcomes in the game - all things that we find programmers doing. They do this with a user experience that is so compelling that people interact with it for its own sake. Studying both the large, available, gaming population and their analytical discourse (e.g. (Salen & Zimmerman, 2003)) might yield interesting learnings.

4.3 Missing psychologies and economies

As we have seen in the analysis above, the psychological techniques that PPIG has employed have primarily been cognitive in nature. Given that programming is a heavily cognitive activity, this is unsurprising. However, there are many other aspects of psychology that are of relevance. There are some good indications in this direction. For example, the application of Prospect theory (Kahneman & Tversky, 1979), which at least partially derives from mathematical psychology, resulted in Attention Investment, one of the award winning theories within the field. Similarly, Conway’s Law, an adage that systems reflect the organisation that built them (Conway, 1968), can be viewed as a form of organisational psychology applied to programming.

Similarly, there may be more to learn from applying behavioural economics to understanding programmers. There is theoretical work missing on understanding why programming languages and
features get adopted when they do. Moreover, there is relatively little work in understanding how notations evolve over time and how their users respond to these changes.

Finally, as (Bowker & Star, 2000) argue, abstractions have political consequences. How these political consequences affect the day to day work of programmers is poorly understood but very important in understanding the relationships between computers and society.

4.4 Theory formation
As the above results indicate, there is much more empirical work within PPIG than there is theory application. In Kuhnian terms (Kuhn, 1996), PPIG is still pre-scientific in its methodologies: there are few stable theories that are widely used to ground the experimentation and the ones we have are relatively poorly validated.

The problem with the lack of stable theory is that it becomes difficult to construct artefacts within a discipline that is continuously reinventing its own basis. This leads to two questions: what are good theoretical foundations that can be used, and how can they be used to extract understanding out of the considerable body of empirical work that is already published at PPIG. This could take the form of either attempting to empirically validate the dominant theories within PPIG (e.g. The Cognitive Dimensions of Notations). Projects like Blackbox (Brown et al., 2014) should provide data that can serve as an empirical basis, if we can understand how to analyze it effectively to ask the right questions.

4.5 Modern technologies
The languages that PPIG has extensively studied are old. Many new languages have been released and grown in adoption since Java. There are no doubt lessons that can be learnt in programming from, for example, Dart (Bracha & Bak, 2011), F# (Syme et al., 2007), Hack (Facebook, 2014), Go (Pike, 2009), Rust (Matsakis & Klock, 2014), Scala (Odersky et al., 2004), Swift (Inc, Apple, 2014). Some of these bring new debates and technical hypotheses (e.g. gradual typing can decrease premature commitment).

Similarly there are new technical trends, such as the widespread adoption of machine learning, that raise new notational and interaction challenges. Debugging machine learning systems is an open problem and one that PPIG is ideally situated to study.

These should be aided by progress in technology. As we highlight in (Church et al., 2016), very large scale computing power can now be economically applied to assist developers, if we can work out how to use it effectively. This offers a very useful opportunity for PPIG, asking the question - how can we use thousands of CPUs to assist developers?

Distributed systems not only offer a technical potential to assist developers, but they also represent a new domain for PPIG to study. Site Reliability Engineers (Beyer et al., 2016) represent a new audience with new roles and challenges.

It is not only new technologies within the domain of programming that pose interesting questions that PPIG is well placed to address. New, or re-imagined, interaction technologies such as Dialog Systems, Augmented Reality and the Internet of Things all carry challenges that could both inform and be informed by the perspectives that PPIG brings.

4.6 Fundamental challenges
There have been hints at PPIG suggesting new directions for the philosophy of computation, from viewing naming as a primary operation rather than a syntactic necessity (Church et al., 2012), to concerns about divergence in the worlds of the program and representation. However, as we suggest
above, there is much more to be done at the intersection of politics, philosophy and the psychology of programming, and even more to be done by building and characterising actual programming languages designed from these insights. This is a fundamental challenge: turning the Psychology of Programming from a reflective community to an actively generative one.

5. Conclusion
In (Berlin, 1953), Berlin dichotomised thinkers and writers into two categories, those with one defining idea through which they see the world - the hedgehogs - and those that explain it through many different ideas - the foxes. Reflecting on the analysis above we must conclude that PPIG is a fox, not a hedgehog.

As the new excitement for the design of programming systems grows, we propose that PPIG should engage by expanding its horizons to study the new languages, domains, psychologies, philosophies and programmer populations - synthesizing this knowledge into new theoretical frames. In other words, PPIG is at its best as a fox and its scope for influence grows, the PPIG of the future needs to be foxier still.

6. Acknowledgements
We would like to thank the PPIG and PLATEAU communities, as well as the programming language research community at large for a fascinating set of material to study.

7. References


Bracha, G., & Bak, L. (2011, October). _Dart, a new programming language for structured web_


A Formal and a Cognitive Model of Anaphors in Java

Sebastian Lohmeier
sl@monochromata.de

Abstract
A formal and a cognitive model are in development to demonstrate the feasibility of programming with direct and indirect anaphors. The formal model is based on the Eclipse IDE. The model generates executable code, handles referential ambiguity, highlights anaphora relations, and permits programmers to switch between source code with anaphors and normal Java code. The cognitive model will forecast when a programmer will be able to understand specific indirect anaphors and when normal Java code should be presented instead. Both models lay the foundation for indirect anaphors that are resolved at edit-time and that shorten source code in cases when comprehensibility is expected to be maintained.

1. Introduction
While reference in statically typed object-oriented programming languages like Java typically uses local variables that declare an identifier, it would also be possible to refer using anaphors that do not require the declaration of an identifier but exploit readily-available textual information. An instance created by the expression new ServiceRegistrar() could e.g. be referred to using the direct anaphor serviceRegistrar that is based on the recurrence of the words service and registrar in the direct anaphor and its related expression new ServiceRegistrar(). It would also be possible to refer to parts of wholes. E.g. given a new RegistrarLocator() and the knowledge that the getter method RegistrarLocator.getRegistrar() returns a ServiceRegistrar instance, the indirect anaphor serviceRegistrar could be used to refer to the service registrar available from the previously mentioned new RegistrarLocator().

Lohmeier (2015) attempted to test experimentally when indirect anaphors in Java are understood by programmers and when they are not. It could be shown that indirect anaphors based on less familiar part-whole relations are understood less easily than indirect anaphors based on more familiar part-whole relations. It remained unclear whether indirect anaphors based on more familiar part-whole relations are understood as easily as normal Java code. It could not be shown reliably that indirect anaphors based on well-known part-whole relations improve the code comprehension of expert programmers. Effects of indirect anaphors on task durations were inconclusive. Responses to the post-test questions and statements that participants made during de-briefing indicated that the use of anaphors could benefit from a number of modifications. Programmers might benefit from (indirect) anaphors while authoring (instead of reading) source code. Anaphors should, in addition, come with the typical tool support of IDEs (e.g. marking other occurrences of a name at the current cursor position or going to the declaration of the current name) that was disabled during the experiment.

Because programming can be modelled formally as well as cognitively, a prototypical formal model of anaphors has been implemented besides the prototypical cognitive model implemented as part of Lohmeier (2015). While the formal model was implemented in the Eclipse IDE to test whether it is possible to switch between anaphors and normal Java code, the cognitive model was implemented to see whether jACT-R\(^1\), an Eclipse-based re-implementation of the cognitive architecture ACT-R (Anderson et al., 2004), can be used to compute activation levels of knowledge representations automatically derived from the abstract syntax tree (AST) of the Eclipse IDE. Both models are briefly described in the following.

2. JDT with anaphors
The Java development tools (JDT) of the Eclipse IDE\(^2\) have been used to create a prototypical editor for anaphors. The editor is based on the Java editor of the JDT. In the editor, anaphors are translated at

\(^1\)http://www.jact-r.org/

\(^2\)http://www.eclipse.org/jdt/
Figure 1 – The direct anaphor $b$ and the indirect anaphor $int_1$ in a Java editor that supports anaphors (left) and the code generated at edit-time in a normal Java editor (right). The editor on the left marks the occurrences of the referent of the indirect anaphor $int_1$ at the cursor position as well as its related expression `new B()`.

<table>
<thead>
<tr>
<th>Related expression</th>
<th>Anaphora resolution</th>
<th>Referentialisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIC</td>
<td>DA1Re</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td>The anaphor refers</td>
<td>The referent of the</td>
</tr>
<tr>
<td></td>
<td>to the referent of</td>
<td>anaphor has a name</td>
</tr>
<tr>
<td></td>
<td>the related</td>
<td>that is equal to</td>
</tr>
<tr>
<td></td>
<td>expression.</td>
<td>the simple name</td>
</tr>
<tr>
<td>LVD</td>
<td>IA2F</td>
<td>Rt</td>
</tr>
<tr>
<td></td>
<td>The anaphor refers</td>
<td>The referent of the</td>
</tr>
<tr>
<td></td>
<td>to the field of</td>
<td>anaphor has a type</td>
</tr>
<tr>
<td></td>
<td>the related</td>
<td>with a name that</td>
</tr>
<tr>
<td></td>
<td>expression that</td>
<td>is equal to the</td>
</tr>
<tr>
<td></td>
<td>the referentialisa-</td>
<td>simple name that</td>
</tr>
<tr>
<td></td>
<td>tion strategy can</td>
<td>acts as anaphor.</td>
</tr>
<tr>
<td></td>
<td>be applied to.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IA2Mg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The anaphor refers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>to the return value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>of the getter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>method of the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>related expression</td>
<td></td>
</tr>
<tr>
<td></td>
<td>that the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>referentialisation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>strategy can be</td>
<td></td>
</tr>
<tr>
<td></td>
<td>applied to.</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 – Strategies available for anaphora resolution in the formal model

edit-time, i.e. when entered, instead of at compile-time. The prototype currently supports combinations of the related expressions, anaphors resolution strategies and referentialisation strategies listed in Table 1. (The left part of Figure 1 contains the related expression `new B()` in line 6, followed by the indirect anaphor `int_1` in line 7 that combines the strategies CIC, IA2F and Rn from Table 1.) Referential ambiguity is eliminated in a small number of cases. When there is a local variable declaration `B b = new B()` that contains the class instance creation expression `new B()` as its initializer, both are potential related expressions but the local variable declaration is the preferred referent. When a getter method declaration `public Integer getValue() { return int_1; }` and a field declaration `public Integer int_1;` are found as potential referents of the indirect anaphor `int_1`, the getter method declaration is preferred over the field declaration.

Figure 1 shows two anaphors $b$ and $int_1$ (on the left) and the code generated by the refactoring that translates anaphors into normal Java code (on the right). The use of the JDT acknowledges the influence of the programming environment highlighted by Green (1989) not only by shifting the translation of anaphors from compile-time to edit-time but also by providing typical guidance to programmers like the `mark occurrences` feature. In the left part of Figure 1 this feature marks the related expression `new B()` (the whole) of the indirect anaphor `int_1` (the part). Guidance like `mark occurrences` is expected to ease comprehension of indirect anaphors based on lesser-known part-whole relations.
Anaphors are translated into normal Java code by applying a refactoring. The refactoring is applied automatically when the anaphors-enabled editor is used to enter anaphors. While the refactoring generates Java code, the generated statements and expressions are hidden by the anaphors-enabled editor – only the entered anaphor is shown in this editor. The editor therefore separates the user-interface and the knowledge representation functions of the programming language. When the normal Java editor is used instead, no anaphors are displayed but all Java code is shown. Java code is saved to disk without anaphors. This enables programmers to write anaphors regardless of whether other programmers might understand them.

When a programmer opens a file she added anaphors to previously, they could be restored automatically, e.g. from a separate meta-data store on disk. The prototype therefore shows that edit-time anaphors can be implemented in an editor for the Eclipse IDE. The editor can be used besides the normal Java editor of Eclipse. It would also be possible to generate anaphors for previously unread code. A cognitive model could be used to decide when to generate such anaphors.

3. A model of anaphor comprehension in jACT-R
Eye movement data obtained in Lohmeier (2015) has been input into a cognitive model of source code reading that re-generates fixation durations. The fixations of a programmer reading source code are therefore mapped to words displayed by the Java editor of Eclipse (see Figure 2). The words in the source code are mapped to nodes in the AST of Eclipse (see the left part of Figure 3) and lead to a sequence of fixation durations on AST nodes. The AST nodes are also used to generate an intermediate knowledge representation (see the right part of Figure 3). The intermediate knowledge representation is fed into a jACT-R model that creates chunks of declarative knowledge for which activation values are computed. The sequence of fixated AST nodes is fed into the jACT-R model and the model re-generates the durations of these fixations. The duration of a re-generated fixation is calculated based on the activation of the chunk in the declarative memory of the jACT-R model that represents the AST node underlying the fixation. The more often a chunk is retrieved, the more active it becomes, but activation decays with time. Fixations that involve a retrieval of a chunk are shorter the higher the activation
Figure 3 – Abstract syntax tree (left) and intermediate knowledge representation (right) for the code shown in Figure 2. The knowledge representation is input into a jACT-R model that creates chunks in declarative memory from the intermediate representation.

of the chunk at the time of the retrieval, because retrieval is faster for higher activation. The jACT-R model is assumed to model the comprehension of anaphors in source code, if it re-generates the fixation durations input to it. This is not the case so far, the model currently over-estimates fixation durations (Lohmeier & Russwinkel, 2015). If the generated fixation durations are reasonably close to the durations of experimentally-obtained fixations input into the model, the model may be assumed to match cognitive processing in humans reading anaphors in source code in situations resembling the experimental setup. At that point, the activation values of a chunk computed by the model might be used to forecast whether it will be easy or hard to understand an indirect anaphor whose comprehension requires the chunk to be retrieved from memory.

4. Conclusion

The two prototypes demonstrate that it is possible to implement edit-time anaphors in a Java-based editor and to model their comprehension in jACT-R. Both models could be integrated by using the cognitive model to forecast whether a file of source code opened in a new editor should be displayed with indirect anaphors or with normal Java code. That would permit to vary redundancy in source code in order to improve the comprehension of individual programmers: programmers would be presented with indirect anaphors that do not repeat the relations that they already know well, relations that are not well known would be presented as normal Java code that explicates these relations.

5. References


Programming with simulated neurons: a first design pattern

Carl Evans
Dept. of Computer Science
Middlesex University
C.Evans@mdx.ac.uk

Ian Mitchell
Dept. of Computer Science
Middlesex University
I.Mitchell@mdx.ac.uk

Chris Huyck
Dept. of Computer Science
Middlesex University
C.Huyck@mdx.ac.uk

Abstract
An investigation has been carried out with regard to programming a form of deterministic logic based entirely in terms of biologically plausible neurons. To this end, a prototype has been successfully developed that incorporates a neuron version of the classic state design pattern. This neuron version is based on a novel programming technique, which models logical states as persistently active cell assemblies. These are populations of intra-connected neurons that have been triggered to continually fire until programmatically suppressed, thus enabling a neural form of state-transition logic. These neural-state cell assemblies have been developed using a specialist neuron simulation software library that is commonly employed by neuroscientists and is the adopted software protocol for the hardware platforms currently being developed for the Human Brain Project. An underlying inspiration of the work is to look forward to the possibility of a programming paradigm based entirely on biologically plausible neurons. It is envisaged that such a neural programming paradigm would benefit from established techniques, and that the neural cell assembly state pattern that has been developed and described in this report is a next step in that direction. In addition, a new graphical notation has been formulated in order to visualise the prototype. Whilst not a primary focus of the research to date, this visualisation notation may prove beneficial to the computational neuroscience community who work with similar neuron simulation software as that employed for the prototype presented here.

1. Introduction
This report describes a first design pattern for programming with simulated neurons. It is, in part, an adaptation of a simulated neuron programming technique pioneered by Huyck (2009). It is a novel approach that intersects the typical perspectives of computer scientists specialising in artificial intelligence (AI) and computational neuroscientists for whom a primary concern is modelling the biochemical function of the brain. That is to say, in practical terms computer scientists working with standard network topologies tend to focus their interest on the design of artificial neural networks (ANNs) for learning (and classification) tasks, whereas neuroscientists tend to investigate and measure response to interconnected populations of neurons, via the use of specialist neuron simulators, based on a variety of mathematical models. Members of the AI community often develop ANNs using general purpose, symbolic, programming languages. One could suggest that a desirable goal of the connectionist approach to programming AI systems would be to have a programming language based entirely on neurons. Perhaps this would be a domain-specific language rather than a general purpose language, but central to its approach would be a much more biologically plausible mapping to biochemical neuron activity than can be achieved with the general purpose symbolic languages that currently exist. However, whilst a purely neuron-based programming language does not yet exist, there are hardware platforms that are based on a neuron architecture rather than a traditional Von Neumann structure, e.g., the SpiNNaker architecture (Furber et al. 2013). Furthermore, a number of such neuron hardware platforms support neuron simulator libraries such as PyNN (Davison et al. 2008). The research groups employing neuron simulation libraries generally belong to the neuroscience community. Huyck and Mitchell (2014), on the other hand, are computer scientists who have demonstrated the use of this combination of neuron-based hardware and neuron simulator software to develop a classification system based on the earlier simulated neuron programming technique developed by Huyck. That work has a broader goal of machine learning rather than promoting a novel programming paradigm per se. However, the focus of the work reported here is to adapt this programming model and to make some further steps towards a simulated neuron design pattern that aims to provide a blueprint for, at least, one aspect of programming deterministic logic in simulated neurons. In particular, this work has mapped the concept of programming with neuron populations to the classic state design pattern of Gamma et al (1995). The state design pattern
is specifically an object-oriented design pattern that has a distinct model of state object transitions. This pattern does lend itself to a neuron model, which represents state as a firing neuron populations, but such a design is less straightforward than the object-oriented version. Addressing this design problem is a key feature of the work described in this report. In addition, the authors have a desire to demonstrate integration of a simulated neuron programmed module within a broader software framework and demonstrate more general utility. Accordingly, a small prototype has been successfully developed, which incorporates the neuron state model as a component within a model-view-controller (MVC) architectural pattern, whereby another of the classical design patterns, the strategy pattern, is employed to interchange the underlying model component between an object implementation (using object-oriented Python) and a neuron implementation (using PyNN).

The remainder of this report is structured as follows. Section 2 provides some brief background to aspects of computational neuroscience and design patterns that underpin this research. Section 3 describes the neuron-based state pattern of programming which is the main focus of this work. In presenting this, the report describes a new graphical notation that has been developed in order to visualise the implementation of the prototype. Section 4 discusses the integration of the neuron model within the context of an MVC framework, and Section 5 provides conclusions and proposes future evolution of the research.

2. Background

The following background sections aim to convey some basic underpinnings to the work. They mainly relate to a few key neuroscience concepts, the fundamentals of design patterns, and refer to the software technologies that have most relevance.

2.1 Spiking neuron models

The concept of a ‘spiking’ neuron model is a fundamental feature. When one (pre-synaptic) neuron signals another (post-synaptic) neuron across their synapse, a change in the neuron’s electrical (membrane) potential occurs. If the potential is large enough, the charge on the post-synaptic neuron rapidly changes. This is known as an ‘action potential’, which reaches a peak and resets, thus forming a voltage pulse, or ‘spike’ in the membrane potential.

![Figure 1](image.png)

*Figure 1 – Characteristic plot of an integrate and fire spiking neuron*

Figure 1 illustrates the characteristic sharp rise and peak of potential, which then ‘leaks’ away and resets to a baseline voltage (typically achieving a negative charge prior to settling to the baseline). This represents a classic neuron model that is typically represented in computational neuroscience as a numerical integration, hence the term ‘integrate and fire’ neuron model (Brette and Gerstner, 2005). There are numerous mathematical neuron models based on this theme.
2.2 Neuron simulation and hardware

Several neural simulation tools have been developed with the purpose of allowing neuroscientists to simulate biologically plausible networks of neurons based on the types of spiking neuron model described above. These simulation systems allow computational neuroscientists to build neuronal networks at a high level of abstraction and their application programming interfaces (APIs) allow researchers to vary the parameters of the mathematical formulae on which the chosen neuron model is based. Furthermore, these simulation technologies enable large scale neuron networks, and developers often think in terms of a network of neuron ‘populations’ rather than a network of single neurons (although, of course, still possible and potentially of interest). The simulator software libraries provide support for creating sets of neurons that can be both intra-connected within a given population, as well as being inter-connected with other sets of neurons. They also provide numerous algorithms to define different types of cell population connection, along with various other features such as the generation of electrical inputs (such as spikes), and facilities to examine network activity. Examples of popular spiking neuron simulators are NEST (Plesser et al. 2015) and Brian (Goodman and Brette, 2013). When creating neural networks with these simulators, developers can take advantage of the Python programming language. PyNN (pronounced ‘pine’) is a Python package that provides a simulator-independent library for building neuronal network models (Davison et al. 2008). PyNN operates at a higher level of abstraction such that the (same) Python code used to create a network will run on several underlying simulator implementations that are supported, effectively providing a type of ‘write once, run anywhere’ meta-language. Currently NEST and Brian are both supported by PyNN. Like the supported simulators, PyNN provides a high-level API for neuron populations, synapse models, connectivity algorithms etc., particularly for large-scale networks, whilst still allowing a lower-level API that may be suited to smaller networks but allows more flexibility. An important aspect of the PyNN library is that it is the adopted software API for the European Human Brian Project1 (HBP). In particular, two (complementary) neuromorphic hardware projects are under development: BrainScaleS2 at Heidelberg, Germany, and SpiNNaker3 at Manchester, UK. These projects are developing semiconductor computer chips based on novel neural architectures rather than the traditional Von Neumann architecture. Both the BrainScaleS and SpiNNaker systems have an interface, designed for neuroscience researchers, based on Python scripts using the PyNN API. An HBP funded project being conducted by the AI group at Middlesex University, entitled ‘Neuromorphic Embodied Agents that Learn’ (NEAL) aims to develop an agent system that learns, specifically using a test environment of PyNN in combination with the HBP Neuromorphic Platforms (Huyck et al. 2015). The NEAL project provides a context for the work reported here, and this has been influential in the adoption of PyNN and Python technologies for this research.

2.3 Cell assemblies

Using a neural simulation middleware, such as PyNN, one can create connections between neuron populations using various synapse models. Synapses are either excitatory or inhibitory. Put very simply, the type of synapse model can determine whether or not the action potential of a pre-synaptic neuron stimulates the post-synaptic neuron (excitatory) or negates its existing activity to some degree (inhibitory). An important biological aspect is that the strength of a synapse can change over time, either in the short term or long term. With repeated modification, the strength of the synapse excitation (or inhibition) becomes increased (or decreased), and semi-permanent, an effect known (biologically) as long-term potentiation (LTP) or long-term depression (LTD). LTP is thought to be the basis of learning and memory, a process known as ‘plasticity’. Furthermore, it is understood that the proximity of connected neurons is very influential, such that, if cells (neurons) that are near to each other repeatedly fire together, their firing efficiency is increased, or reinforced, a phenomenon sometimes referred to as Hebbian learning, named after Donald Hebb, the father of neuropsychology (Hebb, 1949). Hebb was also particularly interested in how neurons acted together in groups, or ‘cell assemblies’, which has been a focus of work by Huyck and his co-researchers. In particular, Fan and

---

1 https://www.humanbrainproject.eu/
2 https://brainscales.kip.uni-heidelberg.de
3 http://apt.cs.manchester.ac.uk/projects/SpiNNaker
Huyck (2008) developed a variation using a ‘fatiguing leaking integrate and fire’ (FLIF) neuron model to form cell assemblies. This FLIF neuron model has been employed to demonstrate how a population of intra-connected neurons (i.e., a cell assembly) can be configured to fire in a persistent manner. That is to say, the neurons within a given assembly can be wired to maintain a continuous firing state, and this has been used to demonstrate that a network of cell assemblies can be arranged to represent a finite state automaton (FSA). Further discussion on this model is provided in Section 3. With regard to the use of cell assemblies, a priority of the work of Huyck et al. has been on learning, with NEAL being one of the more recent projects. The research described in this report is orthogonal to this theme. The focus is not on learning, as such, but rather to demonstrate how the cell assembly approach can be used to represent deterministic logic by way of a neuron-based state-transition model. In particular, a blueprint for programming with simulated neurons is proposed that adapts the cell assembly model to represent the concepts behind a classic object-oriented design pattern, namely the state design pattern.

2.4 Design patterns
The world of software design patterns has broadened considerably since the landmark text by Gamma et al. (1995). The ‘gang of four’ (GoF) authors, as they are frequently referred to, proposed specifically twenty-three object-oriented patterns, which have become regarded as classic software design patterns. It is generally accepted that some of these classic patterns are, perhaps, more pervasive than others. For example, the observer design pattern is fairly ubiquitous, whereas a pattern such as the flyweight pattern tends to be seen in quite specialised domains such as computer graphics. Buschmann et al. (1996) extended the patterns concept to an architectural level (commencing a series of texts describing software architectures, the so-called Pattern-Oriented Software Architecture, or POSA, patterns), and since then a large patterns community has evolved and continues to grow.4 A fundamental concept of design patterns is that they provide a solution template to a recognised design problem. Very importantly, design patterns are not invented specifically to solve a given design problem. Rather, they are recognised for what they really are, i.e., acquired wisdom with regard to an existing approach to solving a recurring design problem. It is the categorisation and documentation of that solution blueprint that constitutes the design pattern. Gamma et al. did not invent their patterns, rather they documented and formalised known techniques to solve known problems, and it is also particularly important to acknowledge that they aim only to represent guidance rather than absolute frameworks. As an example, consider the state design pattern. This is one of the original GoF patterns and has applicability when a component’s dynamics should vary when its internal state is modified (it is thus regarded as an example of a behavioural pattern). The configuration of an object-oriented pattern is typically reflected in a UML (Unified Modelling Language) class diagram, which presents the general template of the design solution. Figure 3 illustrates the standard UML class diagram for the state design pattern. There is no absolute requirement to follow this exact model in order to call one’s implementation a state pattern. But key to the class configuration is a hierarchy of state classes that extend from a common parent class, which in turn defines a consistent protocol for the client component (the ‘context’ class). The basic arrangement is that, at any point in time, the context would be linked with only one of the implemented state class objects, and the currently-referenced object would represent the current state of the system. Hence, the specific implementation of the operations of the current object define the dynamics of the current state. However, a key aspect that the general GoF state design pattern UML class diagram, as shown in Figure 3, does not explicitly illustrate is that the spirit of the pattern is for an individual state object to determine, or control, which state it transits to. That is to say, it is a particular feature of the object-oriented state design pattern that the implemented state sub-classes take responsibility for changing the class of the current state object (as referenced by the context object) whilst adhering to the abstract protocol specification.

Referring to Figure 3, if the context object (i.e., an object of ContextClass) is currently linked to an object of class ConcreteStateA, and a request to execute an operation on that object is made, then the implementation of that object’s operation may determine whether or not the reference from the context object to a state object should be changed, and if it is to be changed, which state object (of a

4 www.hillside.net
sibling sub-class of the abstract State class) it should be changed to (in which case it would typically ‘call-back’ to the context object to set its new link).

Figure 3: State design pattern (object oriented)

This state sub-class control is particular to the object-oriented design pattern, rather than the more general FSA concept. Furthermore, different operation implementations may determine different state change decisions, thus promoting a potentially complex object state transition model. For the described prototype, however, the relatively simple model of a traffic light simulation has been selected. This system has only four discrete states, each with a single operation to essentially move to the next state in the sequence (hence only one possible state transition per state). In terms of an object model, the typical implementation is for each concrete state object to set the reference of the context object (acting effectively as a state-machine object) to point to the next state object in the sequence, with the traffic signal system having a cyclic state transition model as illustrated by the UML state diagram in Figure 4.

Figure 4 (a) illustrates the structural (class) configuration of the general GoF state pattern applied to the traffic signal example. This only requires one abstract operation that each of the four concrete (light) states implement, and that is to change the reference of the context object. The dynamics of this are represented in the UML state diagram illustrated by Figure 4 (b).

Figure 4: Object-oriented state pattern representing a traffic signal

2.5 Summary

The thrust of the work described in the remainder of this report addresses the premise that simulated neurons can be used to program in a general purpose manner. Such a programming paradigm would naturally appeal to those working within the field of artificial intelligence and could be employed to complement typical machine learning and classification tasks (Huyck and Fan, 2007). The underlying theme of this position is to adopt the connectionist approach to AI and aim to develop ‘intelligent’ programs in a biologically plausible manner. Some connectionists might argue that this cannot
ultimately be achieved with symbolic programming alone. It is envisaged that this will take on more importance with the emergence of biologically inspired computer architectures that will, perhaps one day, support a programming language that is predominately neural. In addressing this challenge, much of the work of Huyck and his co-researchers has had a strong focus on the use of cell assemblies as a biologically plausible model. Whilst, in the main, the work of Huyck et al. has investigated numerous aspects of learning and memory, the idea of being able to program more generally, via the cell assembly model, underpins a number of their projects and provides the inspiration for this investigation. This branch of the research deviates from a specific focus on learning and memory, and attempts to validate the cell assembly based FSA design by adapting it to a blueprint for, at least, one aspect of deterministic programming in neurons. That is to say, a first design pattern that is specific to programming with simulated neurons.

3. A neuron-based state pattern

This section will describe the prototype software that has been developed in order to demonstrate the neuron-state model described in the report. The source code is available to download directly from the following url: http://www.cwa.mdx.ac.uk/NEAL/code/neural-state-simulation.zip.

3.1 Visualisation of PyNN code

Rather than presenting Python code, this report will convey the structure of the program using a visual representation of the network, and in particular, visualisations of specific PyNN programming structures that are critical to the model. These visualisation are, in essence, a novel feature in their own right. They have been designed to aid in comprehension (and documentation) of some earlier projects that employed similar cell assemblies, such as NEAL. There is clear potential to extend this visualisation aspect further, but this is not a primary focus of this report.

A PyNN program is a timed simulation. Following initial set up (mainly related to neural timing parameters), a simulation typically comprises the following aspects: a spike generator, one or more populations of neurons, and a set of projections that link populations of neurons. Any given population can be based on a single cell type, or a combination of cell types. The populations created for the prototype that has been developed comprise a single cell type, which is an integrate-and-fire model. The PyNN class representing a cell type provides numerous parameters to configure those cells according to the mathematical model upon which they are based. A projection makes synapses from the neurons of one population to another population, and does so using a specified connector. Connectors comprise a combination of a synapse type (which specifies excitatory and inhibitory weight values), and a connector algorithm. Various standard connector algorithms are provided via the PyNN API. For example, one can connect a single neuron from one population to all neurons in another population, make ‘all-to-all’ mappings, make random connections, or provide bespoke mappings.

Whilst PyNN provides a wealth of neuron component classes within its API, only a few visual artefacts are required to represent the neuronal state model that has been developed: a neuron population, an intra-population projection, and inter-population projection (which specifies connector or synapse type) and a spike-source component. Examples of these are illustrated in Figures 5a and 5b. PyNN provides a series of specialised cell types (as programming constructs) to represent a range of electrical inputs to a neuron (or a neuron population). Collectively, these are termed ‘spike sources’ and are typically used to stimulate (i.e., add excitatory or inhibitory weight to) neurons within a given simulation. The prototype employs a straightforward ‘spike source array’ cell type. This essentially allows the programmer to code an array of specific ‘spike times’ throughout the simulation (in milliseconds). The spike input is achieved by connecting the spike array to a population of neurons. Figure 5a (i) illustrates a spike source of one neuron. A key feature of the cell assembly approach is to create a population of neurons that are intra-connected. So, for each neuron in a given population, the programmer can ‘project’ a connection from that neuron to its neighbours within the same population. Furthermore, the programmer can specify to which neighbours to project the connection. Figure 5a (ii) illustrates a visualisation that is used to represent a population of neurons, with the number of neurons in the population indicated by the number enclosed within the triad of connected nodes. Figure 5a (ii) is intended to illustrate that each of the 10 neurons in the population is connected to
each of the other 9 neurons in the population, but not self-connected. Figure 5a (iii) illustrates the
same type of population with neurons self-connected as well as to all neighbouring neurons.

![Image](image1)

Figure 5a: Visualisation of neuron populations

With populations defined, one can use several variations of connector to establish the overall network. Some of these are illustrated in Figure 5b. For example, Figure 5b (i) illustrates a connector that is of type ‘all to all’ with a net excitatory weight (indicated by the plus symbol). The arrows indicate the direction of the connection.

![Image](image2)

Figure 5b: Visualisation of population connectors

Figure 5b (ii) indicates an ‘all to one’ connection algorithm with net inhibitory weight (indicated by the minus symbol), Figure 5b (iii) shows one neuron connected to all in a given population with net inhibitory weight, and Figure 5b (iv) illustrates a one-to-one neuron connection with positive weight (excitatory), which is employed in the prototype for connecting spike sources to a cell assembly.

3.2 The model

The characteristic operation of a PyNN script is that it is executed for a predetermined time specified in milliseconds. The selected spike mechanism provides network input, and the response of the network to those inputs is recorded. There are several options for determining how data relating to network activity is recorded and examined but it is typically plotted in the form of a graph. The goal of the described prototype, however, is to represent firing states in real-time, and so requires an ability to interpret population activity during the simulation run rather than inspecting recorded data after the program has completed. The PyNN API does not specifically provide for this requirement, so it was a small design challenge that needed to be overcome. This was manageable, however, because the PyNN simulation control does allow for repeated invocations of Python ‘callable objects’. This allows timed repetition of a section of code that can record the current activity of the network, inspect the recording during the same program iteration, and then reset the recording parameters ready for the next ‘call back’ from the simulation controller. Key to the implementation solution is the capability to project a population onto itself (i.e., an intra-projection) and that the cell assemblies of the defined states essentially all exist in a ‘primed’ state. In particular, Huyck and Fan (2008) demonstrated how two active cell assemblies could ignite a third, and effectively spread their activation to that third cell population. In addition, they were able to control how a cell assembly could suppress the activation of another population so that its neurons stop firing. A similar design is used with regard to the implementation of the neural-based traffic-light state model implementation that is illustrated in Figure 6. It should be noted that the population, connector and spike source network illustrated in Figure 6 is one configuration that achieves a solution, but there are alternative configurations that can also achieve the desired results. For example, it is possible to employ populations with fewer neurons and configure the connectors with alternative algorithms and replicate the overall network activity.
In other words, Figure 6 illustrates an overall structural pattern that can use different population sizes and connector algorithms. The network visualisation in Figure 6 includes labels (a) through (f) to aid explanation of how the model operates. With respect to the state populations (indicated as containing 10 neurons) illustrated in Figure 6, state transitions operate from left to right. A spike-source generates a timed series of positive pulses throughout the simulation to represent triggers for state changes. This is indicated by label (a) in the diagram. This source is connected to a single neuron in each of the four cell assemblies (i.e., intra-projected neuron populations). The synapse weight is sufficient to ‘prime’ the populations, but not to make them fire. The network is essentially started by a second (single) pulse injected to the first cell assembly, indicated by label (b). The combined excitatory weight is configured to reach a threshold such that the first cell assembly now fires and maintains an active state. This active state is connected to the next cell assembly in the chain via an all-to-all projection as illustrated by label (c), but the net excitatory weight is insufficient for the second cell assembly to fire and it remains primed. After a predetermined interval, the next spike is injected into all four cell assemblies at point (a). The combined excitatory synapse weights that feed into the second cell assembly at points (c) and (d) are sufficient to reach a threshold to make the second cell assembly fire and maintain an active state. A one-to-many projection goes from that cell assembly back to its predecessor, which has sufficient inhibitory weight to suppress its activity, effectively taking it back to the primed state. The second cell assembly is projected forward to the third cell assembly, denoted at position (f) in the network, but again, there is insufficient weight to fire the third cell assembly until the next timed spike from position (a), and the sequence continues such that the network switches between all four states with only one cell assembly firing at a time. Deterministic control of the state transitions between cell assemblies is effectively under the control of the spike (trigger) input at position (a). In the prototype, this is a timed series of pulses, but this could be event-driven. Each timed pulse is transmitted to all four cell assemblies in the system such that the spike input is not targeted at any one specific population. In effect, the network of cell assemblies itself, via its wired projections, determines the active neuron state transition. Thus the spirit of the state design pattern is captured in the model.

4. Prototype architecture
As noted above, PyNN developers are typically neuroscientists who visualise the operation of their neural models via several types of graphical plots at the end of a given simulation. Certainly, the activity of the connected neuron populations in Figure 6 can be presented in a plot. However, the work described here has a more specific goal of conveying the utility of this network model as a neural programming pattern, and as such there is a desire to demonstrate the pattern as an implementation model within the context of a broader software system. To this end, the prototype has been integrated within a model view controller (MVC) architecture. What is, perhaps, interesting about this approach is that it is the model component that is interchangeable. MVC is a fairly ubiquitous architectural pattern, but the vast majority of implementation descriptions that one finds
reference to focus on the interchangeability of the view component. For example, many web frameworks utilise MVC in this manner. However, the interchangeability of the model component was always a key aspect to the MVC pattern described by Gamma et al. (1995). Although MVC was not listed as one of their 23 classic patterns, Gamma et al. made specific reference to the importance of their strategy design pattern within MVC. The strategy design pattern facilitates the encapsulation and exchange of an implementation algorithm at runtime for a given controller, and the prototype described here has adopted this architectural approach, which is illustrated as a UML class diagram in Figure 7.

The view component of the prototype is a simple Python GUI from which the user can run a simulation of a timed sequence of transitions of a standard UK traffic signal (represented as simple coloured graphical widgets). In the prototype, this can be achieved with either a classic object-oriented state design pattern (object model) or via the new neuron-based state model. The GUI provides a selection widget to change the implementation model from the object-oriented version to the neuron-based PyNN implementation, and vice versa, and does so whilst the program is running (thus satisfying the strategy pattern requirement to be dynamically interchangeable). This is managed by the controller component, which employs the strategy pattern to assign the specific implementation model of the simulation and delegate to that state model sub-type accordingly. In fact, structurally the strategy pattern looks similar to the state pattern in that a context component (the controller in this case) delegates to one of a number of model implementation strategies that satisfy an abstract parent class protocol. There is, however, a distinct difference in dynamics between state and strategy patterns, with the intent of the strategy pattern being focussed only on encapsulation of the algorithms of a model. In this prototype MVC implementation, communication of state transitions from model to view, via controller, is facilitated via a publish-and-subscribe model using a queue data structure. In order to allow for both models to be interchanged during a single run of the prototype, this queue stores an abstracted enumeration of the four states of the traffic signal.

5. Conclusions
Programming with simulated neurons has two main benefits: neurocognitive modelling, and neuron inspired AI. Neurocognitive modelling of the brain using a simulated version of a relatively low-level and well understood primitive, i.e., neurons, will help advance understanding of neural and cognitive functioning. The benefits of neuron inspired AI are perhaps less obvious, but more immediately important. The waning of Moore’s Law can be partially compensated with parallelism, so concurrency is significant. Programming in simulated neurons gives a very fine grained parallelism, using billions of processors. The advent of neuromorphic hardware takes advantage of this, but the software community does not really know how to take full advantage of these systems. Currently, there is no neuron-specific programming language. At some point in the future, dedicated neuron-oriented programming languages may exist, and the developers of such languages might look to base their APIs on established neural programming patterns.
The work described in this report is still in its early stages but a working prototype has been achieved as a proof of concept demonstration. At this time, a neuron-based equivalent of the classic state design pattern is tentatively proposed. There is some further work to be achieved before, for example, announcing this to the patterns community. For one thing, the prototype is based on a very simple model with a single predetermined transition from one state to another single state. The next stage is to develop the existing prototype to operate with a more complex state model in which the currently firing cell assembly has a choice of more than one possible transition to alternative cell assemblies. The authors are confident in achieving this, but only at that point might one accurately suggest a full state design pattern equivalent.

A very important aspect of this work is that this neuron-based pattern of programming has not simply been invented. Rather, it has been adapted from a technique that was developed by a computer scientist and his co-workers specialising in machine learning with neural networks, and who have not been consciously operating within the world of patterns. Key to the idea of proposing a design pattern is in the recognition of a technically sound, and repeatable, programming method and its representation. Uniquely, in this case, its representation is not object-oriented but neural. The authors view this as the beginning of an interesting strand of investigation. Mapping the neural prototype described in this report to a classic design pattern is a first step. Possibly, the authors may investigate the mapping of other well-known patterns from symbolic programming paradigms in terms of their usefulness in neural programming. However, there may well be neuron-specific patterns to be discovered, and it is reasonable to suspect that concurrency will be at the heart of some of them.

6. References
Abstract

Computer science educators agree that, for many beginners, learning to write computer programs is very difficult. And, in spite of our best efforts at remedying the situation, many novice programmers still struggle. We report some preliminary findings from our study that seeks to understand what sort of preconceptions novices use when learning to program. Our early results show unsurprisingly that student programmers can be separated distinctly into the have and have-nots. This separation seems based on mastery of programming skills that are likely developed in an incremental fashion that suggests a hierarchy. But, in spite of these findings, we have also noted that predicting whether a student can correct or debug a program does not seem to depend on their success in these other areas.

1. Introduction

It has been fifteen years since the first McCracken group concluded that introductory programming students completing their first course do not “know how to program at the expected skill level.” (McCracken, 2001) No doubt many of us had experienced this anecdotally, but the report suggested that the situation was both widespread and measurable.


Experimenting with languages, courses, and teaching methods is useful but most of these efforts sidestepped a more fundamental issue stated over thirty years ago by Spohrer and Soloway,

A reasonable pedagogical philosophy is The more we know about what students know, the better we can teach them (their emphasis). Yet in the past, what educators knew about novice programmer was largely “folklore”: anecdotal evidence from their own and their colleagues’ experiences. (Spohrer, 1986)

In short, do novice programmers (introductory students) think about programming in the same manner as expert programmers (their teachers)? The research on novice programming over the past fifteen years is united on one point: novice programmers are different from expert programmers in how they think about and approach problem-solving using programming. (e.g., Hofer, 2010, Jimoyiannis, 2011, Lahtinen, 2007, Lui, 2006, Ramalingam, 1997, Weiser, 1983, Ye, 1996). Defining the nature of these differences is another matter.

Some of the more interesting and extended studies have been conducted by researchers who were originally associated with the BRACElet project. (e.g., Clear, 2008, Clear, 2009, Clear, 2011, Tan, 2010, Whalley, 2009) A significant outcome of their research is that writing programs is a higher-order skill that is built upon the acquisition of more basic cognitive skills (Lister, 2009, Lopez, 2008, Venables, 2009). Introductory students cannot be expected to write programs effectively until they master these other skills. Several hierarchical learning taxonomies have been applied to help explain this (e.g., Corney, 2012, Lister, 2000, Lister, 2006, Teague, 2012, Teague, 2015, Whalley, 2006).
2. Research Agenda

Our ultimate goal is to develop and test more effective interventions for teaching introductory programming. But, first, it is imperative to have a better grasp of the typical conceptual or mental models that novice programmers apply to these learning tasks. It would also be useful to have a more accurate assessment of how students’ conceptual/mental models develop as they progress through the programming sequence.

Furman University is a small, selective liberal arts college in the U.S. with an enrollment of 2,800 students. We offer three degrees in computing and currently have 60+ majors. Consequently, our subjects represent both a smaller and perhaps different sample compared to those of previous studies.

The major objectives of our study are these:

• to replicate (repeat) and extend previous research on novice vs. expert programmers (Lister, 2008b).

• to contribute to a more informed model of the novice programmer in order to investigate effective pedagogical interventions.

We hope to determine whether the performance of novices conforms to what traditional research tells us (e.g., Kuolori, 2015, Robins, 2003). In short, novices focus on syntactic details of code in terms of line-by-line execution. They lack viable mental models for problem-solving and programming. In contrast, more expert programmers understand programs based on abstracting parts of the code and have a variety of mental models for problem-solving.

Our focus will be on finding the dominant mental models that novices develop for programming solutions using variables and assignment statements, control structures such as selection and repetition, as well as code writing and debugging strategies.

3. Methods

Many of the previous studies on novice programmers collected evidence from testing novice programmers using standard instruments such as multiple-choice tests. While it is possible to assess the correctness of answers, it is not clear how or why the students selected their responses. Thus, the researcher can only hypothesize what sort of mental models students employed. If the research is accurate, then expert programmers think very differently than novice programmers when solving problems. As a result, instructors (as expert programmers) would find it difficult to imagine how their students (as novice programmers) think. Recently, though, think-aloud methodology has been used for more direct evidence about how students go about solving programming problems (Teague, 2012, Teague, 2014, Van Someren, 1994, Whalley, 2014).

Over the past year, we have employed both traditional testing for larger groups as well as recording think-aloud sessions with a smaller number of students.

4. Preliminary Results

To achieve a better understanding of the kinds of skills that qualify distinctions among students, we attempted to develop a typology of students based on item response patterns from a short written test. Most of the questions were derived from or based on those employed in previously reported research (Denny, 2012, Ramalingam, 1997, Teague, 2012, Teague, 2015). The intent was to compare our results with those from other studies. We also added questions in order to assess basic (logical) debugging skills.

We sought to identify subsets of students with a similar pattern of correct and incorrect answers to the computing tasks posed by the test questions. To this end, we used Latent Class Analysis (LCA). LCA is a person-centered method used to explain variability of responses as a function of membership to unobserved but assumed existing groups. Sample members are assigned to unique homogeneous latent classes based on similar arrays of correct or incorrect responses. Members of each latent class are more similar than dissimilar and clusters are formed in such a way that differences between clusters are augmented or maximized (Muthen, 2000). Using this statistical approach, we identified two clusters of students with unique response patterns. The two-latent class model was superior to a
one-class solution (Lo-Mendell-Rubin Adjusted LL = 181.15, \( p < .001 \)) as well as the best fitting model in comparison to all other models considered. The entropy statistic (a measure of classification accuracy) of the two-class model was .86. The number of cases were evenly distributed between classes with about half of the students identified as belonging to either LC1 (n=71, 49%) or LC2 (n = 74, 51%). Results are presented in Table 1.

The first column in Table 1 shows the proportion of students in the sample who were able to provide a correct response to each of the problems given. We see that nearly all students answered one of the tracing questions (i.e., T2) correctly whereas less than 50% of the sample succeeded on the problems involving tracing code (T1, 47%), explaining code (E2, 47%). The remaining three columns in Table 1 display the conditional probability of a correct response for a member belonging to a given class. As seen, the likelihood for a student belonging to LC1 of being correct on items assessing tracing, writing, and explaining code is quite high. By contrast, a student assigned to LC2 has lower chances of having a successful attempt on the same types of items. Overall, compared to LC2 students, LC1 students were much more likely to produce correct solutions to a variety of item types; at the same time, the success rate across items is notably lower for LC2 students. LC2 students appear to be particularly deficient with respect to writing code (W1 and W2) and explaining code (E2). The findings suggest also that the most reliable differences between the identified categories lie in the area of debugging. A LC1 student is four to five times as likely to answer the first (DB1: Odds Ratio = 4.79, \( p = .028 \)) and second debugging item (DB2: Odds Ratio = 4.37, \( p = .02 \)) correctly as is a LC2 student. Differences between latent classes are depicted in Figure 1.

<table>
<thead>
<tr>
<th>Test Items</th>
<th>Overall Proportion</th>
<th>Two-Class Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace [T1]</td>
<td>.469</td>
<td>.823</td>
</tr>
<tr>
<td>Write [W1]</td>
<td>.371</td>
<td>.619</td>
</tr>
<tr>
<td>Explain [E1]</td>
<td>.793</td>
<td>.861</td>
</tr>
<tr>
<td>Explain [E2]</td>
<td>.465</td>
<td>.838</td>
</tr>
<tr>
<td>Debug [D1]</td>
<td>.382</td>
<td>.552</td>
</tr>
<tr>
<td>Debug [D2]</td>
<td>.374</td>
<td>.534</td>
</tr>
<tr>
<td>Write [W2]</td>
<td>.391</td>
<td>.806</td>
</tr>
<tr>
<td>Trace [T2]</td>
<td>.928</td>
<td>.963</td>
</tr>
<tr>
<td>Trace [T3]</td>
<td>.816</td>
<td>.877</td>
</tr>
<tr>
<td>Trace [T4]</td>
<td>.618</td>
<td>.669</td>
</tr>
<tr>
<td>N</td>
<td>145</td>
<td>71</td>
</tr>
<tr>
<td>Percent</td>
<td>100%</td>
<td>49%</td>
</tr>
</tbody>
</table>

*Table 1: Results from Latent Class Analysis*

*Figure 1: Probability of correct solution by latent class*
Results from a chi-square test of independence estimating the relationship between type of course and latent class membership indicated that the students in the pre-programming course (CS0) were more likely to belong to LC2, $\chi^2_{145} = 58.25, p < .001$. Additional analyses – unrelated to focal questions of interest – revealed that (a) GPA was positively correlated with LC1 probability, $r = .31, p = .026$, suggesting that students who were stronger academically were more likely to be part of LC1; (b) gender did not have a measurable effect on the likelihood of belonging to either class, $t_{50} = .31, p > .05$; and (c) LC1 students tended to find the items easier in greater proportions as shown by a moderate positive correlation between number of items endorsed as being easy and latent class probabilities, $r = .57, p = .001$.

![Figure 2: Distribution of students in latent classes by course level.](image)

Our latent class analysis demonstrates that, regardless of classification, students tend to be only moderately successful on computing problems requiring debugging. While in regards to the category of LC2 students this finding does not seem to be surprising, the somewhat lower probabilities for success among the abler LC1 subgroup suggest that debugging is relatively independent of other skillsets. This observation is consistent with results from additional correlation analyses showing weak correlations between the two debugging items in the assessment and other skills. Performance on the tracing sub-questions (based on the Parson’s problem) was virtually unrelated to performance on the two debugging items (DB1 & DB2) as documented by small, non-significant correlations in the range from -.01 to .26. The correlations between the first tracing item (T1) and DB1 ($r_{143} = .20, p < .05$) and T1 and DB2, ($r_{143} = .267, p < .001$) were statistically significant, yet small. The strongest associations were between DB1 and E2 ($r_{143} = .37, p < .001$) and between DB1 and W2 ($r_{143} = .32, p < .001$). The corresponding coefficients between DB2, E2, and W2 were even smaller ($r_{143} = .27$ and $r_{143} = .31$, respectively). Overall, we failed to uncover evidence of co-variation between ability to correct error in code and other computing skills. Although somewhat related to other essential skills, ability to debug appears to be sufficiently distinct.

5. Future Work
During the summer (2016), we plan to code and analyze the 30+ hours of audio/video collected from the think-aloud sessions of our twelve subjects. Each subject completed four separate sessions. These included problems involving assignment statement, selection and iteration control structures. Subjects were asked to trace written code, predict outcomes, supply missing statements from code segments, read and explain written code segments, fix (mostly) logical errors in supplied code segments, and write a short program. Most of the work was done using paper and pencil, but the latter two activities were completed using a computer for feedback. The preliminary results suggest the subjects had varying levels of skills and abilities in these areas. In addition, the subjects’ approach to problem-solving often differs significantly when the work is done using a computer. We expect that these sessions will provide a rich source for qualitative analysis.

At any rate, these results will provide a basis for the next stage of our project. However, based on our preliminary results, exploring the interplay of writing and debugging programs appears to be a useful avenue of further investigation.
6. References
Ambrosio, Ana Paula, et al. (2011) “Identifying Cognitive Abilities to Improve CS1 Outcome,”

Barg, Michael, Alan Fekete, et al. (2000) “Problem-Based Learning for Foundation Computer Science Courses,”


Campbell, Jennifer, Diane Horton, et al. (2014) “Evaluating an Inverted CS1,”
*SIGSCE’14*, pp. 307–312.


*CPRIT’12*, pp. 77–86.

*ICER’08*, pp. 113–124.


*SIGSCE’13*, pp. 283–284.

Guo, Philip. (2014) “Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities,”
*BLOG@ACM*, http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities.

*ITiCSE’03*, pp. 104–108.

Guzdial, Mark. (2015) “What’s the Best Way to Teach Computer Science to Beginners?”


Hu, Helen, and Tricia D. Shepherd. (2014) “Teaching CS1 with POGIL Activities and Roles,”


Abstract
This paper proposes a way of thinking about physical making, from the perspective of end-user programming. It addresses current interest arising from the Maker/Hackerspace movement, which aims to democratise the means of creating new mechanical and electronic (mechatronic) devices. By applying insights from the domain of end-user programming, it proposes ways that mechatronic making can be made more accessible to a wider audience, just as programming has been made more accessible to a wider audience through end-user programming.

1. Introduction
There are many fashionable areas of Ubiquitous Computing – Physical computing, Mechatronics, and the Internet of Things – based on the presumption that digital infrastructure can be readily interfaced with the mechanical properties and behaviours of physical objects. However, the practicalities of those mechanical interfaces often require specialist tools and training – electronics, breadboards, soldering irons, workshop tools and so on. Should we assume that specialist expertise is unavoidable in this field, or can this kind of functionality be made accessible to a wider audience?

In the domain of software, we are familiar with the concepts of end-user programming, end-user customisation or end-user development, in which we provide tools usable by people without specialist technical training, that are nevertheless suitable for accomplishing real-world tasks with computers. The purpose of this paper is to explore whether similar principles might be extended to the electro-mechanical systems that are controlled by computers. The initial argument is by analogy between programming (or ‘software construction’) and electro-mechanical design (‘physical construction’).

In addition to helping people achieve real-world tasks (for example, as supported by spreadsheets), end-user programming (EUP) research is often associated with educational initiatives that aim to broaden access to computing, for example through initial learning environments (ILEs) (McKay & Kölling 2012) such as Scratch, SonicPi or GreenFoot. However, the relationship between end-user programming and educational computing is a complex one (Blackwell 2006a), and it is therefore useful to review a few recurrent issues that often get tangled together here.

1. Both EUP tools and ILEs are designed to be simple and approachable, emphasising a ‘gentle slope’ from initial experiences to greater computational power (Pane & Myers 2006).
2. The ultimate goal of an ILE is to communicate computational concepts rather than achieve a practical objective, where these might be either broad cognitive skills such as ‘computational thinking’ (Wing 2008), or more concrete curriculum elements (Computing at School 2012).
3. Nevertheless, many ILEs place a strong emphasis on motivating children to learn programming. This is done by providing EUP support for the creative leisure activities of children, such as story animations (Kelleher & Pausch 2007), video games (Resnick et al 2009), or electronic dance music (Aaron et al in press).
4. When powerful computing capabilities are simplified to a degree that makes them suitable for children, the increased usability is often appreciated by users of all ages – most famously in the outcome of Kay’s KiddiKomp/DynaBook project (1972) that led eventually to the development of Smalltalk and the Graphical User Interface.
5. As a result, the design of purely educational programming languages such as Pascal or Scratch, even where they are motivated by curriculum rather than application, does often influence the design of more accessible mainstream or general purpose programming languages for practical and end-user applications. For this reason, it is valuable to consider the needs of children, even when research (such as this paper) is aimed at end-users generally.
This paper anticipates that the same dynamics observed in the interaction between EUP tools and ILEs might also apply in the domain of mechatronic construction. To summarise the resulting research questions: Can we identify a ‘gentle slope’ for mechatronic construction? Can this be based on a more general cognitive understanding? Are there creative applications that would appeal to children? Might the resulting designs lead to more usable and utilitarian tools for everyone?

2. An agenda for end-user construction: Utility, creativity or education?

In developing an analogy between end-user software construction and physical construction, we should note that there are already some physical products that are constructed by end-users, for example self-assembly furniture. In this kind of DIY physical construction market, the tools, components and instructions necessary to accomplish useful practical tasks have been simplified and refined over many years, to a stage where large numbers of people are able to accomplish practical mechanical results without any specialised training.

An interesting consequence of end-user construction, in the case of furniture, is that the end-user places more value in the result, because of the effort that they have invested in it (Norton et al 2012). Although it might appear that there is little ‘creativity’ in assembling a piece of flat-packed furniture, it appears that the craft experience of working with one’s hands is itself a source of creative reward. The craft element of this ‘IKEA effect’ reinforces the potential value in paying attention to the creative play of children as a source of design inspiration. It also promises potential benefits arising from recent attention to craft aspects of programming (Woolford et al 2010, Blackwell & Aaron 2015).

However in extending the physical construction analogy to creative and educational play, we must consider construction toys such as Lego and Meccano. These toys are even simpler to use than the components of self-assembly furniture, but this is associated with a trade-off. In general, Lego constructions do not have the mechanical strength, motive power, or external ‘interfaces’ that would be necessary to achieve useful mechanical functions. Furthermore, they are closed systems – it is difficult to integrate a Lego construction with an arbitrary piece of woodwork or a cardboard model. In this respect, Lego resembles digital sandbox games such as Minecraft, or even ILEs such as Scratch - it allows you to build toys, and only toys. The reason that so many Lego Technic constructions are vehicles and robots is that this is the only thing it is good for – things that drive around by themselves, but never get attached, mounted or tethered to any part of the child’s real life.

The physical interface technologies used in these ‘sandboxed’ products are not solely an incidental consequence of their original purpose as toys, but also determine their potential future applications. Although it is fun to balance blocks, and snap pieces together with plastic studs or magnets, we do not build our houses, cars or furniture out of these things. Magnets and Lego studs are intentionally impermanent, encouraging reuse, fluidity and experimentation. In terms of cognitive dimensions and their tangible correlates (Edge and Blackwell 2006), they have low viscosity, low premature commitment, high shakiness, and low permanence. They are characteristic of a sketching medium, intended for exploratory design, rather than a construction medium. Furthermore, their uniformity and predictability, although encouraging ease of manipulation, results in a relatively impoverished materiality - their surfaces and forms are smooth and uniform rather than richly textured and resistant. Rather than offering a back-and-forth ‘conversation with materials’ (Schön 1983) they meekly submit to the user’s choice from among a restricted set of possibilities.

In the domain of adult ‘end-user’ material construction, it is well understood that there is a clear distinction between the physical techniques appropriate to playing with toys, and the physical techniques appropriate to the construction and maintenance of houses, cars, bicycles or shirts. Specialist shops sell the tools and materials required for such serious work, and school pupils are instructed in the special methods necessary for working with such ‘resistant materials’ (AQA 2012). There may be some basic level of manual dexterity that is transferred from toys such as Lego to screwdrivers and handsaws, but it seems more likely that any correlation is a matter of cultural expectation and self-efficacy, rather than a lowest-common denominator ‘mechanical thinking’ that could be explained by analogy to the computational thinking movement.
The cognitive and tangible dimensions trade-offs between the design attributes that are associated with playful simplicity, and those that support powerful general-purpose application, is perfectly familiar in the case of software. Neither EUP tools such as spreadsheets, nor ILE game authoring tools such as Scratch, have the versatility or computational power of general purpose programming languages. Much programming language design, whether for end-user developers or children, involves identifying a sweet-spot between power and usability, as well as a gentle slope for those users who become more familiar with the tools, more ambitious in their goals, and want to explore a wider range of functionality. Given the central role of social expectation and self-efficacy in both domains, it would appear valuable to address these issues together, spanning both mechanical and computational ways of thinking. Indeed, a previous PPIG paper exploring the development of new creative tools for the Internet of Things evaluated the self-efficacy of professional artists as potential end-users by drawing an analogy from their familiarity with mechanical construction tools to their confidence in first encountering programming (Blackwell, Aaron & Drury 2014).

3. Why a vernacular language?

The agenda set out in the previous section offers a motivation for a new kind of ‘end-user construction language’ for mechatronics, that is distinct from sandboxed toys like Lego, and could potentially be informed by past research in end-user programming language design. To be clear, this is taking a psychology of programming approach, but applying it to questions that are not normally related to programming. It is not usual to speak of physical objects or electronic circuits as if they were a ‘language,’ other than through the structural semiotic perspectives of cultural studies as developed by Roland Barthes and others. My goal here is more abstract, proposing an engineering language (rather than a cultural one), but one that can be used to describe physical objects and mechanisms from a functional perspective that is accessible to, and usable by, end-user audiences other than engineers.

I should note that although the emphasis of this paper is to investigate physical construction rather than software, it may well be the case that programming would also be involved in an actual end-user project enabled by such a language, for example if novel mechanical devices were being attached to microcontrollers or embedded devices such as Raspberry Pi, Arduino, ARM mBed, and many others. This is an obvious (and likely) extension of the discussion in this paper, but will not be addressed further here, in order to concentrate on the novel aspects of the proposal.

3.1. Functional description as API specification

The argument so far has taken a determinedly user-centric perspective, in setting out the case for an end-user construction language by analogy to end-user programming. Nevertheless, this analogy is also wholly plausible from a technical perspective. In systems design, it is well understood that information flows between software and the physical world via sensors and actuators, and that system functionality is embedded in and constrained by physical apparatus such as cases and mountings.

Although sensors, actuators, I/O peripherals, and the physical environment of the computer are usually considered to be outside the scope of programming language design, it is possible to analyse their functional descriptions as a more extensive and general kind of Application Programming Interface (API). Just as APIs can be designed to offer a more consistent set of user-centric abstractions with the assistance of Cognitive Dimensions analysis (Clarke 2003), a language for electro-mechanical abstraction might also be constructed from a user-centric perspective. We might even describe this as a Cognitive Dimensions of Physics, in which the physical world is treated as a kind of notation, through the functional language that we use to analyse and describe it.

3.2. Naturalness in API descriptions

If we apply this user-centric strategy to evaluate the I/O interfaces and mounting hardware that are conventionally used for mechatronic projects, this offers an interesting critical perspective, by analogy to the design of the most commonly used APIs: those of language utility libraries. General purpose programming languages such as Java (in contrast to end-user languages) often rely heavily on libraries that implement generic computational abstractions – lists, stacks, heaps and so on. Without prior education in the algorithms and data structures of computer science, those abstractions are rather impenetrable to the end-user.
Domain-specific libraries tend to be described in terms that are more directly related to the user’s actual tasks, but it is often the case that such libraries build on the intermediate abstractions of general purpose utilities, making them also difficult to use and understand for those unfamiliar with the basic abstractions. Designers of conventional programming languages are often unable to accept that this is a problem, or even to notice it. From their expert perspective, conventional library abstractions such as a stack have become ‘natural’ or even ‘intuitive,’ because of the way they are so tightly integrated with the design of mainstream programming languages. As a result, new libraries and APIs continue to be designed according to the same abstractions as those that are already established. If there is a problem, experts often perceive this as a problem of education, to be addressed by imparting the computational thinking principles that are apparently necessary to use conventional abstractions.

In contrast, some end-user programming systems extend the principles of domain-specific language design to deeper levels of abstraction, by ensuring that the semantics and syntax of the language are derived from abstractions and notations already familiar to the user. This is relatively routine in the field, as used for example in the “Natural Programming” project of Myers et al (2004), in which educational programming languages such as Pane’s HANDS (2002) were derived from the vocabulary and concepts that children already used to describe behaviour in videogames.

3.3. Naturalness and naïve physics

An interesting liaison between applied anthropology and artificial intelligence during the 1980’s (Gentner & Stevens 1983) encouraged the field of naïve physics – an attempt to build computer systems that could reason about physical phenomena following common-sense principles, rather than those of trained physicists (Hayes 1978). The goal of such research was in part to simulate human intelligence, but also to produce expert systems that might be able to explain the reasons for their decisions, in terms that were accessible to lay people. The legacy of naïve physics includes qualitative spatial reasoning systems (Forbus 1983), which support navigation or spatial queries expressed in natural language rather than geometric terms.

My purely-visual language Palimpsest (Blackwell 2014) consciously applied this perspective to programming language design. Having previously conducted research in the field of qualitative spatial reasoning (Blackwell 1989), it was apparent that Myers’ Natural Programming approach could be applied to non-verbal domains such as image processing, using qualitative terminology. Of particular interest was the development of a set of data types and geometric operations that would be adequate for a wide range of pictorial transformations and processes. Many of the Palimpsest functions were initially implemented in terms that would be familiar from school geometry, but subsequently re-implemented using concepts that offered more “natural” interaction with images. For example, the mathematical operation “translate” was replaced with "move it to here". Similarly, the most common application of “rotate” turned out to be "make it spin round" (as an animation), and “scale” was "stretch or squash" (not uniformly, but in various ways that drag handles can produce). Similarly, the basic data types of Palimpsest were refined to represent the most salient features of images: basic data types include ‘image’, ‘colour’, ‘shape’ and ‘location’ rather than conventional mathematical data types such as ‘integer’ and ‘real’.

As was the case with expert systems employing naïve physics, geometric processes defined in terms of naïve computation are unlikely to be as concise or generalizable as conventional mathematical terms and notation. However, this trade-off applies to many innovations in HCI (the GUI versus the command line, for example). To some extent, user-centred design always involves stepping away from an existing, possibly elegant, engineering description in order to accommodate alternative mental models. From this perspective, the goal of the 1980s naïve physics movement could have represented a user-orientation within the AI community, trying to create knowledge representations that were better aligned with common sense although somewhat inelegant. The anthropological origins of the “naïve” movement always emphasized the need to respect the customs and reasoning strategies of other communities, rather than simply disparaging them as undisciplined or murky thinking.
3.4. From natural to vernacular

Although the ambitions in applying Myers’ Natural Programming project to EUP (Pane & Myers 2006) are laudable, the potential for broadly differing interpretations of ‘naturalness’ means that it is not always clear how this ambition might be carried through into language design. What is natural to one person (a computer scientist) is not necessarily natural to another (a child), and indeed something that seems natural to a person at one age may not seem natural to the same person at another time. Furthermore, to return to the cautions given in the first section of this paper, the distinction between EUP environments and ILEs is not always clear. In some cases, ILEs are designed with the intention of communicating curriculum concepts, and thus changing the ‘natural’ (i.e. uneducated) conceptions that the users might previously have held (Rode et al 2003). In other cases, domain-specific languages emphasise the use of terminology and concepts that are already understood by users. When the users are children, the problem arises whether their ‘natural’ terminology should be regarded as arising from domain expertise, or from lack of understanding.

The perspective offered by the naïve physics agenda does sidestep this dichotomy, by clarifying that there may be valid and useful descriptions that do not correspond to textbook definitions. Of course, we should remember that the word ‘naïve’ is not very diplomatic in a user-centred design context, where it is important to use terminology that respects the user’s competence and judgment, rather than accusing them of naivety. The key insight in naïve physics is not ordinary people are naïve, but that they are generally able to act quite successfully in the physical world without recourse to specialist expertise, whether Newtonian mechanics, quantum physics, Ohm’s law, or computational thinking.

Rather than ‘natural’, the goals of this paper seem to be best served by the word ‘vernacular’ – which is to say, the kind of informal and everyday language that is used by ordinary people in their native context, as opposed to the kind of language that is used by professional specialists from other communities. This word makes it clear that naturalness of language or description is not a universal attribute, but rather one that is specific to particular cultures and groups of people. In particular, we should not assume that vernacular descriptions of software should be the same among computer scientists and end-users, and we should not assume that vernacular descriptions of mechatronics should be the same among engineers and children. (Remembering here, point 4 of my opening rubric, which is that designing for children has often served as a useful strategy in escaping previous design fixation – the intention is not simply to achieve a vernacular appropriate only to children, but to use the critical lens of creative childhood experience as a means of reconceptualising end-user tools).

4. Toward playful electro-mechanical abstractions

In order to evaluate the potential benefit of this strategy, this section considers the ways in which current mechatronic construction technologies demonstrate the properties discussed above, and then suggests some initial features of an alternative vernacular language. Throughout, I maintain a focus on the potential for supporting creative childhood experience, but as explained, with the intention that this is likely to be a valid strategy for broader future applicability.

4.1. The abstractions of mechatronics

A quick survey of online accessory catalogues for popular hobbyist platforms such as Raspberry Pi, Arduino, Phidgets and .NET Gadgeteer reveals an array of specialist terminology, including ‘potentiometers’, ‘quadrature encoders’, ‘relays’, ‘gyrosopes’, ‘solenoids’, ‘accelerometers’, ‘proximity sensors’, ‘differential air pressure’, ‘thermocouples’, ‘linear actuators’ and many more. As an engineer, many of these devices are familiar to me, and there must clearly be some demand for them among the customers of such catalogues, otherwise they would not be offered for sale. However, they do not appear at first sight to represent a vernacular language outside of this technical community, and certainly not for the children who I have proposed as a target audience.

If we compare these interface components to the standard APIs of programming language utility libraries, it is clear that there is a specialist conceptual language embedded in the names of the components, and also in the conceptual design of their functionality. The technical terms are precise, but the concepts that they represent are not easily described in a more vernacular manner, other than by mundane explanation.
If we look beyond these relatively well-encapsulated sensor and actuator devices, the situation deteriorates. A typical introductory presentation explaining how to ‘Blink an LED’ with a Raspberry Pi reads as follows:

An LED is a Light Emitting Diode. A diode is a circuit element that allows current to flow in one direction but not the other. Light emitting means … it emits light. Your typical LED needs current in the range of 10-30 mA and will drop about 2-3 volts. If you connect an LED directly to your Pi’s GPIO it will source much more than 30 mA and will probably fry your LED. To prevent this we have to put a resistor. If you want to do math you can calculate the appropriate resistance using the following equation … [and so on]

(Minardi 2013)

In discussing the relationship between domain-specific languages and general purpose ones, I introduced the problem of ‘intermediate abstractions’ that might be considered natural to a person already familiar with a wide range of programming languages, but would not have been encountered by someone who does not have the necessary background in computer science or computational thinking. The ‘intermediate abstractions’ of these standard electro-mechanical interface components and techniques are similarly foreign to end-users. From current-limiting resistors as described above, to op-amps, transistors and many other components, we find abstractions that are familiar to electronic engineers, but not to DIY handymen, or children (or even computer science undergrads). The mechatronic ‘library’ components of solenoids and thermocouples, even when packaged for end-user plug-and-play, still require a conceptual engineering language to describe and apply.

Furthermore, while every child can join Lego bricks, many are unable to strip the insulation from a wire, use a soldering iron, read the value code of a resistor and so on. Furthermore, even if we consider the cruder level at which an engineer or hobbyist might mount a Raspberry Pi or Arduino in a wooden box, or shape the coupling between a solenoid and a lever, many children (or adults) might struggle to hammer in a nail, or cut a straight line with a handsaw and file. Although those are routine skills and concepts for an engineer, and even for the typical Raspberry Pi ‘maker’ hobbyist, they present obstacles for children who might want to emulate those adult examples.

4.2. Physical craft as vernacular

Might it be possible, when identifying and designing user-oriented electro-mechanical components, to specify them, not in standard engineering terms (‘solenoid’, ‘stepper-motor’, ‘microswitch’), but in terms related to the practical experiences of an imaginative child (‘cardboard’, ‘string’ and so on)? Ideally, these should not be constrained to toy assembles and sandbox contexts, but potentially usable for practical tasks, offering a gentle slope from first experiences to useful applications. A reasonable benchmark would be the mechanical properties of Ikea furniture – requiring only a small range of physical construction skills, but sufficiently strong and powerful to support real-world functionality. So although toy makers such as Lego do provide sensor and actuator interfaces to introductory programming languages, the objective would be to avoid these in favour of a craft vernacular that is engaged with adult life.

By analogy to domain-specific programming languages, such components could have electro-mechanical functions that are understandable to children because they correspond to mechanical actions already made by children. Examples include:

- Pull a string
- Squirt a hose
- Swing a door
- Flick a switch

The advantage of household functions, rather than toy scenarios, is that they can be integrated with the existing commercial infrastructure of standardised electro-mechanical components that are familiar to children:

- Strings can be tied to nails or hooks
- Hoses can be connected using the Hozelock system
• Door hinges can be nailed or screwed to pieces of wood
• Switches can power small appliances via extension cords and plugs

Although it is attractive to provide a gentle slope from childhood play to practical application, parents of Lego-using children who read this might balk at the new potential for harm suggested by these scenarios – with power comes responsibility! Nevertheless, each of these interface technologies is also regulated by safety standards specifically intended for domestic use, and there is no reason in principle why the creativity of children should be completely isolated from the adult world.

Alternatively, there are vernacular mechanical actions that can be applied to physical objects, without involving sufficient force for damage, but offering social and communicative potential:

• Pluck
• Twist
• Knock
• Brush
• Tickle
• Scratch
• Feed (a pet)

Finally, while conventional sources of motive power intended for the engineering functions of a typical house are sufficiently powerful to be somewhat dangerous, many natural forms of energy are relatively safe. Relatively small amounts of surplus energy in these scenarios would even be sufficient to power an embedded processor offering end-user programmable functionality. (The last two are somewhat fanciful, but perhaps more appealing to children).

• Wind chime – powered by wind, makes sound
• Grandfather clock – powered by wind-up weight, pulls string
• Light show – powered by daytime sun, makes bright light at night.
• Dog whistle – powered by motion of pet, communicates with pet
• Stink bomb – releases smell, powered by motion of trousers

4.3. Systematising vernacular abstractions

Much of the enthusiasm driving the naïve physics movement may have been associated with the academic impulse to systematise the hitherto unsystematic. It is possible that the same impulse adds appeal to the research agenda of natural programming. It is certainly likely that the abstract functions listed above could be collected more systematically, for example as in the application of Reuleaux’s taxonomy of kinematic pairs to analyse the possible configurations of moving parts in a tangible programming language (Blackwell & Edge 2009).

However, it might be wiser to resist this impulse to systematise, especially at this early stage. As observed by Umberto Eco, the desire to systematise vernacular languages is a largely Quixotic enterprise (Eco 1995). There are several sources of systematicity already embedded in the illustrative suggestions made above, which arise in part from:

• The commercial dynamics of industrial standardisation
• The distinction between classes of domestic energy reticulation
• The refinement of building materials to fill economic niches

If new end-user tools were to be developed in response to the arguments presented here, then the necessary engineering design process would itself impose commonalities and distinctions, whether or not they were ‘naturally’ present in the vernacular contexts. If there are forms of tacit knowledge embedded in craft, that might be resistant or even opposed to that engineering logic, then it would be advisable to follow an explicit strategy of late-binding in future work that develops these ideas. Rather than looking for a mechatronic language that resembles the existing infrastructure of sensors and actuators, while simply substituting alternative names or power sources, there might be more opportunity for insight from paying closer attention to the physical vernacular of everyday life, as in the ‘unremarkable computing’ of Tolmie et al (2002).
One interesting opportunity is the potential to focus on materials that are softer or more malleable than typical engineering components. At the engineering end of the hacker/maker movement, as in student laboratories, robot competitions and science fair projects, end-user construction is facilitated by products and standards such as the OpenBeam construction system – a kind of larger-scale Meccano. However, as the basis of a vernacular language, systems such as OpenBeam are oriented toward a very characteristic building style – precise, but also hard- (and sharp)-edged. We should perhaps seek opportunities to support soft materials such as fabric, fur, feathers and faces, and vernacular ‘interfaces’ to clothing and other social signifiers such as zippers or outfits. As I have noted in the past, these types of material support abstraction, while resisting the gender normativity that is associated both with engineering construction and with DIY (Blackwell 2006b).

5. Related Work
The challenges in interfacing low-cost computers for practical mechanical purposes are extremely well known, and there are countless historical and contemporary products that have addressed this potential market. The two most common product classes at present can be grouped under the general headings of robotics kits, and IoT / smart home products. Robotics kits tend to be aimed at children or for classroom use, while IoT hub products are aimed at the hobbyist market. However, there is some degree of crossover between the two markets, with major products such as Arduino and Raspberry Pi catering to both, and a range of accessories and guidebooks oriented toward one or the other.

Most of this activity, and in particular the large number of Kickstarter initiatives that cater to one or other of these markets, is driven by technical enthusiasm and subject to the two primary flaws that I have discussed: the electro-mechanical robots tend to be autonomous in a way that restricts them to toy sandboxes, while the hobbyist applications are modelled on existing engineering descriptions and components. I will not review these in detail, other than to note their prevalence.

There have been a small number of exceptions, generally emerging from academic research contexts. MaKey MaKey (Silver et al 2012) is an interface kit based around the abstraction that any object can be treated as a ‘key’ (in the sense of a key on the computer keyboard), and can then control software applications in accordance with that abstraction. The digital ‘key’ abstraction is supported by auto-ranging and signal conditioning of analog inputs, such that electrical connections can be made to a wide range of objects with varying conductive and capacitive characteristics, and in particular, natural objects such as plants, fruit and vegetables.

A series of projects by Orth et al (e.g. 1998) has explored the opportunities for computers to be mounted in, and interfaced with, fabrics of various kinds. Circuits can be constructed using embroidered or woven conductive thread, although mounting rigid chips and controllers has often been problematic (Buechley & Eisenberg 2009). The majority of fabric computing applications have focused on toys, soft furnishings and wearables, with the last of these often featuring LEDs or colour-changing materials that respond to touch and temperature. To date, these have generally been decorative (although this is undeniably a functional feature of wearable technology) rather than delivering end-user capability for novel engineering functionality.

Among the many educational robotics systems, Schweikardt’s (2011) modular robotics system ‘Cubelets’ presented an alternative set of abstractions for sensing and actuation, based on a metaphor relating these peripherals to the capabilities of living organisms rather than engineering systems. The Cubelets kit interestingly includes magnet-to-Lego interface components, but the ingeniously fluid magnet mechanism is not compatible with mechatronic component standards. Furthermore, communication with the external world is achieved only via Bluetooth wireless, making Cubelets impractical for inclusion in larger assemblies.

Microsoft’s .NET Gadgeteer (Villar et al 2012) offers a relatively conventional set of hardware peripheral modules, similar to those of the Phidgets system, but offers a software-oriented abstract view through the Visual Studio development environment. Each module is associated with a C# class that is automatically loaded into the development environment when that module is connected to a tethered Gadgeteer master unit. As a result, the functionality of the module can be presented from a software perspective rather than focusing on hardware description.
6. Conclusions
This paper has proposed that the distinctive user-centric perspective characteristic of psychology of programming research could also be applied to the physical context of computing – the construction materials, sensors and actuators that are involved when a computer is embedded in a physical context. At present, educational computing systems tend not to be strongly attached to the physical world, while hobbyist technologies are defined in specialist engineering terms. As an alternative, I have suggested that a vernacular language of mechatronic construction could be used to shape components, standards, products and their capacities for playful, creative and useful end-user applications.

7. Acknowledgements
I am grateful to Robert Mullins, Saar Drimer, Jeff Osborne, Mark Gross, and Nic Villar for conversations that have contributed to the development of ideas in this paper.

References


Enhancing Programming Lectures Using Interactive Web-Based Lecture Slides

Paul Albinson
Faculty of Science and Technology
Bournemouth University
palbinson@bournemouth.ac.uk

Abstract
Programming is often seen as a difficult subject to teach and keep students engaged and motivated about. Also programming results are frequently found to be lower than for other subjects (Bennedsen & Caspersen, 2007; Jenkins, 2002; Robins, Rountree, & Rountree, 2003). Therefore, the challenge is to find a way of improving programming education to address these problems. This paper considers the use of innovative pedagogy approaches to do this due to their ability to enhance learning experiences. An innovative pedagogy case study is presented that was designed to test whether interactive web-based lecture slides can enhance programming lectures to make them more engaging and enjoyable and make programming easier to understand. The lecture was an introduction to the jQuery JavaScript library/framework for first year undergraduates. Results were overall positive and show value for approaches like this and that they can enhance lectures to make them more engaging and enjoyable and can be used to make programming easier to understand.

Keywords: Web Technologies, Interactive Web-Based Lecture Slides, Innovative Pedagogy, Programming, Tackling High Failure Rates, Technology Enhanced Learning, TEL, Reveal.js

1. Introduction
Diligent academics frequently look for ways to improve their teaching practices and potential to introduce curricula changes that can enhance teaching, student motivation and engagement, student outcomes, etc. (Albinson, 2016; Vieluf, Kaplan, Klieme, & Bayer, 2012). One way of doing this is by using innovative pedagogy approaches which, as many studies that have tried such approaches (such as Morley (2012), Vieluf et al. (2012), and Higher Education Academy (ca. 2015)) have found, can enhance learning experiences which can lead to better student outcomes due to more engaged and motivated students. Considering these findings, the author wished to see if an innovative pedagogy approach can help improve programming teaching to tackle difficulties with teaching the subject.

Having reviewed a number of contemporary pedagogic innovations, for example Technology Enhanced Learning (Gordon, 2014; Kirkwood & Price, 2014), Blended Learning (Blended Learning Toolkit, ca. 2015; Friesen, 2012), Flipped Classrooms (Brame, ca. 2015; Herreid & Schiller, 2013) and use of web technologies (Anderson, 2007; Gosper et al., 2008), the author chose to explore using web technologies. The rationale being it offers opportunities for: flexible learning (flexibility for where, when and how students wish to learn), the use of technology to enhance learning, and the ability to create a more active student-centred andragogical learning experience; also other researchers (Gosper et al., 2008; Morley, 2012) have found web technologies are useful for enhancing learning experiences and outcomes, student engagement, motivation etc. This andragogical approach is advised for adult learning and will appeal to the way adults prefer to learn, and research has found it can increase engagement and enhance learning and the student experience (Albinson, 2016; Knowles, Holton, & Swanson, 2011). It should also help enhance lectures and make the format more effective which should help tackle the difficulties with teaching programming.

Programming (both web programming and traditional software programming) is often seen as a difficult subject to teach and keep students engaged and motivated about (Bennedsen & Caspersen, 2007; Jenkins, 2002; Wray, 2007). Empirical evidence and research (Bennedsen & Caspersen, 2007; Robins et al., 2003) has found results for programming are frequently lower than for other subjects and engaging students in the subject is difficult.
One reason for the problems with teaching programming could be due to the way it is taught with the traditional lecture format perhaps being inappropriate for the subject. Traditionally lectures for programming have consisted of explanations of concepts and syntax with blocks of related sample code displayed to show how they are used. As this approach is purely theoretical and doesn’t show the output of code students may struggle to understand the concepts being explained as they don’t see what the code does. However, if code output could be shown on the screen then programming should become more understandable. Research by Lahtinen et al. (2005) suggests this would aid learning as they found via a student survey that learning by doing, where students tried code themselves thus seeing how it works, was seen as more useful than traditional lectures. Therefore, if seeing what code does aids understanding then enhancing lectures with examples of how code works should be beneficial.

However, the problem with traditional presentation software is it is designed to display text, images and videos from a central screen in a passive way (Matheson, 2008; Young, Robinson, & Alberts, 2009). There is no option to embed code to show how it works and no way for the audience to interact with the content. However, with modern web technologies we can create web-based presentations that can include more advanced features such as demonstrations of code and tests within the presentation which students can interact with to aid their understanding. This can make lectures more active (and less passive) student-centred andragogy focused experience which is beneficial for adult learning and aiding engagement (Albinson, 2016; Knowles et al., 2011). This is not only useful for teaching programming, it can assist understanding of any subject via active involvement in the learning process and interactive features (for example tests) to aid memory recall and help students construct their own understanding of a concept following the constructivist and cognitivist learning theories (Blondy, 2007; Petty, 2014).

This paper presents a case study designed to test whether interactive web-based lecture slides can enhance programming lectures to make them more engaging and enjoyable and make programming easier to understand. The lecture was an introduction to the jQuery JavaScript library/framework for first year undergraduates. The session was evaluated via an anonymous voluntary student feedback survey; unfortunately, attendance of the lecture was poor and only 36 students completed the survey. Therefore, results simply provide an initial small sample to evaluate the value of this innovation with a larger sample required to validate the findings. It did however present interesting positive results that show value for approaches like this and that they can enhance lectures to make them more engaging and enjoyable and can be used to make programming easier to understand.

2. Case Study - Interactive Web-Based Lecture Slides

In response to the problems discussed about difficulties teaching programming a case study of a potential solution was created using an innovative pedagogy strategy.

This case study introduces a web-based presentation solution using Reveal.js (El Hattab, ca. 2015) which is a web-based slideshow system similar to traditional presentation software such as Microsoft PowerPoint; it uses HTML5, CSS and JavaScript for the basic presentation and Node.js and Socket.IO for more advanced features. This web-based approach with the option of utilising server technology for adding advanced features allows for presentations which are more flexible and can include a wider variety of content and interactive elements enabling a more active learning experience rather than the passive experience of traditional presentations/lectures.

As web-based slides are created using web technologies each slide is a web page allowing programming to be demonstrated directly within the presentation; instead of saying this code does X requiring the audience to imagine the code in action (the traditional presentation approach) the presenter/lecturer can actually demonstrate what the code does. Not only could this demonstrate actual web programming code in action (for example when teaching HTML or JavaScript) it can also be used to demonstrate code examples from traditional software programming languages too. However, when the code isn’t from a native web programming language/technology (e.g. it is from a traditional software programming language like C#) you can’t run the code directly within the presentation but you can show the code and use other web technologies (e.g. JavaScript) to
demonstrate/simulate how it is meant to work; i.e. show C# code but use JavaScript to create the output of what the C# code does (you wouldn’t even see the JavaScript code that is being used).

Reveal.js (El Hattab, ca. 2015) also has the option of having master and client presentations (reveal.js refer to this as multiplexing) to allow the audience to see and interact with the presentation on their own devices. This means there are two versions of the presentation, the master presentation which the presenter uses and shows in the lecture theatre (usually via a projector), and a client presentation which the audience can access on their own devices (such as a tablet computer, laptop, or mobile phone). The master presentation controls the client presentation allowing the presenter to control what the audience sees on their own devices; whatever slide is shown on the master presentation is what appears on the client presentation. When the audience use the client presentation option on their own devices it offers a variety of advantages such as:

- A better viewing experience as the presentation is in front of them rather than on a screen at the side of the room which may not be easy for them to see. Reading text is also easier.
- Improved accessibility: should the audience member need the text size increasing or colours adjusting they can do this on their own device.
- The ability for the audience to interact with the slides, so for example if there is a demonstration of some code which is interactive (e.g. click a button to make something happen) they can try it on their own device.
- When tests are added to the presentation (made possible due to the use of web technologies) it allows the audience to answer questions using their own device for self-testing their understanding to highlight areas they need to study further or seek advice on (hopefully increasing questions asked in the session and audience participation). This is useful formative feedback and student-centred learning.

Therefore, not only does a web-based system allow for more features than a traditional presentation it can also allow the audience to access the presentation on their own device and interact with it. Additionally, as it is web-based it can be hosted online for students to access it anytime anywhere with an internet connection to refresh and self-test their knowledge of the topics covered, try code demonstrations etc. Also research (Gosper et al., 2008) has found online content helps students take control of their learning and become less reliant on the teacher. Reveal.js also has many other useful features such as exporting the presentation to PDF, speaker notes, and many plugins to add extra features.

To investigate the potential of using such a system a lecture was created that used interactive web-based lecture slides. The slides used multiplexing (master and client presentations) and included interactive content for the students to use on their own devices, tests for students to check their own knowledge, and demonstrations of code. It was an introduction to the jQuery JavaScript library/framework for first year undergraduates. It was loosely based on content from a similar lecture from previous years but instead of traditional slides using Microsoft PowerPoint it used interactive web-based lecture slides. You can see the presentation at http://presentations.paulalbinson.info/jquery-introduction; this is the version the audience would use (the client version) to see it on their own devices, interact with code demonstrations and tests, use it anytime afterwards, and so forth.

The session was evaluated via an anonymous voluntary student feedback survey. It asked for opinions on statements regarding learning and understanding, session organisation and clarity, teaching, interactive web-based lecture slides, general opinions, and views on the lecture compared to regular lectures; possible answers/responses were Strongly Disagree, Disagree, Agree and Strongly Agree. A middle neutral answer/opinion “Neither Agree or Disagree” was excluded for the majority

---

1 The lesson plan can be requested from the paper’s author via email.
2 The master version is accessed via a separate link but this link has not been included because using it would change the slides of anyone using the client version therefore it should only be used in a lecture.
3 The survey form can be requested from the paper’s author via email.
of questions to force students to think more carefully about their answers and to avoid indecision and the temptation of answering with the middle/neutral option. Research (Albinson, 2013; Garland, 1991; Johns, 2010) has found that when surveys have middle/neutral options people are tempted to choose them to avoid making decisions or to avoid questions they don’t like or don’t understand, or they feel this response would help or please the interviewer/researcher (e.g. they may not wish to give a harsh or negative response), causing undesirable results. It was however allowed for the questions about comparing the case study lecture to regular lectures as it is feasible for students to be indifferent over lecture styles; although of course it increases the chance of undesirable results. There was also a free text box for any comments students wished to make about the session such as things they liked or disliked, areas that could be improved etc. Results are discussed later in this paper.

3. Reflection on Related Literature and Learning Theories

3.1 Is it Blended Learning?
The case study’s hybrid approach of using technology and traditional class-based teaching is a type of Technology Enhanced Learning (TEL). It shares some characteristics of blended learning as it uses technology to improve the learning experience and for students to use technology to enhance their learning both inside and outside the classroom. However, while sharing the ideals of blended learning of using technology to assist in the delivery of course content it is not replacing any part of the learning with online delivery, it compliments rather than replaces lecture content delivery, and so in that respect it isn’t blended learning. It meets older definitions of blended learning as discussed by Oliver and Trigwell (2005) which described adding some aspect of online learning to the learning experience but it does not meet the more modern understanding and standard definition of blended learning. The modern definition also specifies that some aspect of the learning must be replaced by an online component like for example using the flipped classroom approach where some taught content is delivered online releasing the lesson time for other activities.

When the web-based lecture slides are presented in a lecture at an educational institution it probably isn’t blended learning but when the students use the slides outside the classroom to aid their learning, thus taught content is delivered online, it is blended learning. Also as the lecturer can control the slides the audience sees and everyone’s slides are in sync it would be possible to run the presentation remotely with the audience offsite (and perhaps even the lecturer too) if audio (and ideally video) of the lecturer is added; a method for communicating with the students would also be a useful addition. This delivery would be blended learning as long as it was only part of the course; if the whole course is online it would be online learning.

Therefore, whether a session like the case study is considered blended learning or simply just a use of Technology Enhanced Learning depends on your opinion of what defines blended learning and/or whether the lecture is delivered remotely.

3.2 Student-Centred Learning
Use of web-based lecture slides including additions of interactive elements helps to make lectures more student-centred by helping learners construct their own understanding of content and also makes lectures a more active rather than passive experience. As research has found (Albinson, 2016; Knowles et al., 2011) adults prefer an andragogical student-centred approach as it appeals to their learning preferences and desire for independence, self-direction and self-learning. While the case study is not fully student-centred as it is still a pedagogical teacher-led lecture to allow the teacher to impart the knowledge students need to know this is not necessarily a problem. While in general an andragogical student-centred approach to teaching tends to be preferred by students, especially as learners mature, it is sensible and necessary to use a pedagogical teacher-led approach to introduce new content due to students’ lack of prior knowledge or experience of the content (Albinson, 2016; Knowles et al., 2011). Also as the content is designed to be suitable for self-study and revision outside the classroom with aids to assist self-learning, such as code demonstrations and tests, it supports andragogy in this respect.
3.3 Learning Theories
The case study lecture, like traditional lectures, primarily takes a didactic passive pedagogical teacher-led approach which mostly follows the behaviourist learning theory as it treats learners as dependents with no free will and dictates how they will learn (Petty, 2014). In this case it is due to the learners’ inexperience and lack of knowledge of the content meaning students need to be introduced to the content before they can actively participate in lessons. However, with the web-based interactive lecture slides allowing students to be more actively involved in the lecture the teacher can reduce the behaviourist style and use other more andragogical student-centred learning theories. As discussed, this will appeal to the way adults prefer to learn and research has found it can increase student engagement (Albinson, 2016; Knowles et al., 2011).

By showing students how code works and providing demonstrations they can interact with along with the self-testing elements helps students construct their own understanding of the content. This aligns with constructivism which is based on the belief that learning is most effective when learners construct their own meanings for subjects based on existing knowledge and experiences. It also aligns with cognitivism’s approach of building understanding/cognition based on previous knowledge to assist learning and aid memory recall (Blondy, 2007; Petty, 2014). Cognitivism also advocates learning by doing/practical learning and providing challenging questions to analyse concepts and develop a deeper understanding (Petty, 2014) which this approach facilitates. Students’ knowledge at the beginning of the lecture may be minimal but as they see how code works and can test their knowledge via the tests they can build up knowledge by linking past knowledge, for example how something works in a previous basic example, to later more complex concepts and examples. In addition, it aids self-learning, especially if students make use of the lecture slides outside of taught sessions to aid their learning which links with humanism’s belief of allowing students freedom to learn independently and develop in any way they prefer (Petty, 2014). Social learning theory (Petty, 2014) isn’t used as it relies on students learning via social interaction and there isn’t time to allow such activities to occur in the limited time available in a lecture. However, students could make use of the self-study opportunities that this approach supports for social learning via, for example, group study and discussions amongst themselves.

4. Survey Results
The results were overall positive with the majority of responses being agree or strongly agree. Additional comments (omitted to save space) were also overall positive but some students cited minor concerns and made suggestions for improvements, these and full survey responses can be requested from the author via email. Unfortunately, attendance of the lecture was poor and only 36 students completed the survey making this just an initial small sample to evaluate the value of this innovation with a larger sample required to validate the findings.

4.1 Learning and Understanding
The results of questions relating to learning and understanding were overall positive. These questions related to the aims/learning outcomes of the session so the results give some indication as to whether the lecture can deliver the required learning for the aims to be met. All students felt they gained a better understanding of jQuery as a result of the lecture with 75% agreeing (27 students) and 25% strongly agreeing (9 students). As jQuery uses JavaScript (it is a JavaScript library) students’ opinions on if the session had improved their understanding of JavaScript was also questioned. Results were overall positive with 75% agreeing (27 students) and 8.33% strongly agreeing (3 students). The remaining results were 13.89% disagree (5 students) and 2.78% strongly disagree (1 student). These results are encouraging but there was some negativity which could perhaps be because students don’t see the relationship between jQuery and JavaScript and that code used for jQuery is actually JavaScript code; improving clarity of this point would be worthwhile for future uses of the lecture.

Students were also asked about whether they had a better understanding of JavaScript libraries and frameworks as a result of the lecture and if they see the usefulness of them for supporting project development and reducing development time; for example, such libraries simplify complex operations by having functions that deal with the complexity. Overall students felt they understood JavaScript libraries and frameworks better after attending the lecture with 80.56% agreeing (29 students) and
2.78% strongly agreeing (1 student) but 16.67% disagreed (6 students), note nobody strongly disagreed. Overall opinions on the usefulness of JavaScript libraries and frameworks for supporting development were positive with 61.11% agreeing (22 students) and 30.56% strongly agreeing (11 students) they are useful, while the rest disagreed (8.33% - 3 students), note nobody strongly disagreed. The presence of disagreement in the responses for both of these statements indicates some students don’t understand the point of JavaScript libraries and frameworks so perhaps more time should be spent explaining this if the lecture were repeated.

4.2 Session Organisation and Clarity

Students were asked for their opinions on session organisation and clarity of its content. The majority (94% - 34 students) felt the session was well organised, with 69.44% agreeing (25 students) and 25% strongly agreeing (9 students), with the only negativity being 5.56% disagreeing (2 students). Results are similar on slides being clear and informative with 94% positive responses (34 students) but with a larger proportion strongly agreeing (36.11% - 13 students). Opinions on information being concise were almost identical, with 55.56% agreeing (20 students) and 36.11% strongly agreeing (13 students), although with a slightly larger amount disagreeing (8.33% - 3 students). With such low levels of disagreement for all 3 questions these results are very encouraging for continuing lectures in this style, however investigating reasons for the negativity (albeit small) would be useful.

4.3 Teaching

Opinions were collected on how well the session was taught to allow the teacher to reflect on their teaching practice. Students overall were happy with the teaching with most responses being positive. The majority of students felt the lecture was clearly presented with 52.78% agreeing (19 students) and 33.33% strongly agreeing (12 students), however 13.89% disagreed (5 students). Almost identical to this were the opinions on whether the teacher made content easy to understand which is no surprise as they are similar (clarity aids understanding). The majority of responses were positive with 58.33% agreeing (21 students) and 30.56% strongly agreeing (11 students) with only 11.11% disagreeing (4 students) and nobody strongly disagreeing. Opinions on whether students liked the teacher’s lecturing/presenting style were a little more mixed but still overall positive, 47.22% agreed (17 students) and 30.56% strongly agreed (11 students) but 19.44% disagreed (7 students) and 2.78% strongly disagreed (1 student). The teacher was reasonably inexperienced at presenting lectures at the time so these opinions are understandable and surprisingly very positive for a novice.

4.4 Interactive Web-Based Lecture Slides

Opinions were gathered about interactive web-based lecture slides to assess whether students feel using them is worthwhile. Results were overall very positive and encouraging. Most students found the code displayed on the slides was clear and easy to read with 97% positive responses (35 students), 69.44% agreed (25 students) and 27.78% strongly agreed (10 students), with the only negativity being 2.78% disagreeing (1 student). All students felt the demonstrations of what code does was useful for aiding their understanding with 63.89% agreeing (23 students) and 36.11% strongly agreeing (13 students).

The interactive elements were also well received with only small amounts of negativity. 94% (34 students) felt having test questions is useful to identify areas they need to study further (61.11 agreed (22 students) and 33.33% strongly agreed (12 students)), with the only negativity being 2.78% disagreeing (1 student). Most students felt being able to interact with the presentation was useful with 89% positive responses, 50% agreed (18 students) and 38.89% strongly agreed (14 students), with the only negativity being 8.33% disagreeing (3 students). The majority found the interactive elements made the lecture more enjoyable and engaging with 94% positive responses (34 students), 58.33% agreed (21 students) and 36.11% strongly agreed (13 students), with the only negativity being 5.56% (2 students) disagreeing. Most students agreed that web-based lecture slides should be used in future

---

4 88.33% agreed (21 students), 36.11% strongly agreed (13 students) and 5.56% disagreed (2 students), note nobody strongly disagreed.

5 Note 1 student (2.78%) didn’t add an answer/opinion

6 Note 1 student (2.78%) didn’t add an answer/opinion
with 86% positive responses (31 students), 55.56% agreed (20 students) and 30.56 strongly agreed (11 students), with only 8.33% disagreeing (3 students) and 2.78% strongly disagreeing (1 student)

These responses are very supportive of using interactive web-based lecture slides, and the reasons they were used for (features and benefits) were positively received.

4.5 General Opinions
When asked more general questions about the lecture students’ opinions were diverse. When asked if they thought the lecture would make them better web developers or designers 92% of responses were positive (33 students), 77.78% agreeing (28 students) and 13.89% strongly agreeing (5 students), with the only negativity being 8.33% disagreeing (3 students). The results for whether students felt the lecture would enhance their future work and grades achieved were a little more mixed but still overall positive with 78% positive responses (28 students), 63.89% agreeing (23 students) and 13.89% strongly agreeing (5 students); there were however 11.11% of respondents who disagreed (4 students) and 8.33% who strongly disagreed (3 students). This negativity is confusing as it conflicts with opinions on other questions relating to the value of the lecture which have much lower negativity.

When asked whether they would recommend the session is repeated in future years there were 97% positive responses (35 students), 69.44% agreed (25 students) and 27.78% strongly agreed (10 students), with unfortunately the remaining 3% strongly disagreeing (1 student). When asked if the session was valuable and they were glad they attended it the results were a little less positive with 89% of responses being positive (32 students), 69.44% agreed (25 students) and 19.44% strongly agreed (7 students), with the rest being 8.33% disagree (3 students) and 2.78% strongly disagree (1 student). The additional negativity between these two questions is confusing because if students say the session is worth repeating but they saw no value in it then why are they making the recommendation? Also overall responses to all statements were positive thus saying the session is worthwhile further conflicting with these findings.

4.6 Comparison to Traditional Lectures
Finally, students were asked for their opinions on whether the lecture was better in comparison to traditional (PowerPoint based) lectures. These questions allowed a response of “Neither Agree or Disagree” as it is perfectly feasible to be indifferent over lecture styles. When asked if they felt they learned more in the lecture compared to regular lectures the majority of responses were positive (58% - 21 students), with 38.89% agreeing (14 students) and 19.44% strongly agreeing (7 students), and there were only 2.78% negative responses (1 student) which were disagree responses. 38.89% (14 students) however chose “Neither Agree or Disagree” perhaps meaning they felt they learned the same amount compared to regular lectures or it could mean they wanted to avoid answering the question or they didn’t understand the question. When asked about if they enjoyed the lecture more in comparison to regular lectures there was less neutrality with only 16.67% saying “Neither Agree or Disagree” (6 students). A much larger proportion of the responses were positive (81% - 29 students) with 58.33% agreeing (21 students) and 22.22% strongly agreeing (8 students); also, just like the previous question, 2.78% disagreed (1 student). Indecision aside this is very encouraging as those with an opinion were mostly positive about the style of lecture and there was only 2.78% negativity (1 student) on both questions.

---

7 Note 1 student (2.78%) didn’t add an answer/opinion
8 Note 1 student (2.78%) didn’t add an answer/opinion
5. Discussion

Feedback was overall very positive and showed the aims of the session have been met; however, investigating reasons for the negativity (albeit small) would be useful.

Overall students thought the lecture was useful for aiding their understanding of the lecture content and that the interactive elements helped enhance their learning. They saw value in the session and enjoyed it and recommend it is repeated in future. Responses also show interactive web-based lecture slides are useful for maintaining engagement and motivation and students thought they should be used more in future.

Students consider the use of clearly displayed code, interactive demonstrations of code outputs and self-testing opportunities useful for making programming easier to understand which satisfies one of the main aims of the case study.

Additionally the case study lecture showed how interactive web-based lecture slides can make lectures a more active andragogical student-centred learning experience helping address the passive nature of lectures which, as discussed, is beneficial for adult learning and aiding engagement (Albinson, 2016; Knowles et al., 2011).

Some students ignored some questions perhaps suggesting they didn’t understand the questions or simply didn’t want to answer them. Similarly, for questions where a neutral/middle answer/option was available many chose it which could be due to no preference or indecision, or given the other unanswered questions it may be due to misunderstanding of questions or unwillingness to answer. This overuse of the neutral/middle answer/option corresponds with findings from other research (Albinson, 2013; Garland, 1991; Johns, 2010) which found including such an option can result in undesirable results; it is for this reason that the author was reluctant to include such a response and only added it where it was absolutely necessary.

Unfortunately, attendance of the lecture was poor and only 36 students completed the survey making this just an initial small sample to evaluate the value of this innovation with a larger sample required to validate the findings. Also, assessing this approach over multiple sessions would be advantageous to produce more results to allow deeper analysis to take place. Additionally, with the experiment running over a longer period it would allow for more detailed assessment methods to be used giving more evidence of improved learning and one could also assess success rates compared against previous years where this approach wasn’t used.

6. Conclusion and Future Improvements

As discussed, programming can be a challenging subject to teach and keep students engaged and motivated about and programming results are typically lower than for other subjects (Bennedsen & Caspersen, 2007; Jenkins, 2002; Robins et al., 2003). Therefore, the challenge is to find a way of improving programming education to address these problems. This paper considered the use of innovative pedagogy approaches to do this due to their ability to enhance learning experiences. An innovative pedagogy case study was presented that was designed to test whether interactive web-based lecture slides can enhance programming lectures to make them more engaging and enjoyable and make programming easier to understand.

The way programming is taught was queried and whether the use of lectures could be to blame for the problems with teaching programming. Traditional programming lectures are a passive experience delivered in a didactic pedagogical way containing explanations of concepts and syntax with blocks of related sample code displayed to show how they are used. As this approach is purely theoretical and doesn’t show the output of code students may struggle to understand the concepts being explained as they don’t see what the code does. However, if code output could be shown on the screen then programming should become more understandable.
However, the problem with traditional presentation software is it is designed to display text, images and videos from a central screen in a passive way (Matheson, 2008; Young et al., 2009). There is no option to embed code to show how it works and no way for the audience to interact with the content. However, with modern web technologies we can create web-based presentations that can include more advanced features such as demonstrations of code and tests within the presentation which students can interact with to aid their understanding. This can make lectures a more active andragogical student-centred learning experience helping address the passive nature of lectures which, as discussed, is beneficial for adult learning, helping learners construct their own understanding of content, and encouraging engagement (Albinson, 2016; Knowles et al., 2011).

The interactive web-based lecture slides case study presented here used this approach and results were overall positive and show value in approaches like this and that they can enhance programming lectures to make them more engaging and enjoyable and make programming easier to understand. However, it was only one session with a small amount of students completing the survey, therefore to properly assess this approach a wider sample over multiple sessions would be advantageous to produce more results to allow deeper analysis to take place. Additionally, with the experiment running over a longer period it would allow for more detailed assessment methods to be used giving more evidence of improved learning and one could also assess success rates compared against previous years where this approach wasn’t used. Also wherever there is significant negativity, especially relating to content students don’t fully understand, the lecture will be revised accordingly to address the problems.

7. References


Helping programmers get what they want

Gail Ollis
Faculty of Science and Technology
Bournemouth University
gollis@bournemouth.ac.uk

Abstract

Notions of "good practice" exist for many aspects of a programmer's work. They are intended to bring benefits on a variety of dimensions from the most concrete and measurable, such as program speed, to more subjective characteristics such as readability. This research addresses programmer practices good or bad in any part of their professional work but considers them from the perspective of a single outcome: impact on the productivity of fellow programmers. The findings are now being applied in an experimental new framework for professional development.

1. Introduction

The inspiration for this research was the author's experience as a commercial programmer. It was evident that colleagues were more or less equally capable of creating a working program. People employed as programmers could, unsurprisingly, program; not necessarily at the same pace or with equal mistakes, but they could all get a computer to follow their instructions to produce the desired output. The differences between them were was not so much the success they had in producing machine behaviour but the degree of difficulty for a human reader in understanding the code that brought it about.

Programming work more often involves wrangling existing code than starting with a blank slate (Verhoef, 2000). Even a brand new project written from scratch - a pristine greenfield site upon which software can be built without historic constraints - soon becomes constrained as new decisions are made about tools, interfaces, standards, system partitioning and so on. Time is also a constraint; project managers are reluctant to permit rework when poor programming or infrastructure decisions become apparent. A system can easily become the sprawling, unstructured development that Foote and Yoder (1997) describe as a "big ball of mud". A comment from a participant in the study recounted in section 2 illustrates just how rapidly this can happen: "I saw him write a legacy system in three weeks".

In the author’s experience of working on existing code two patterns repeatedly emerged. In one pattern, making the necessary changes felt like following an obvious route made straightforward by the careful planning and signposting done by a predecessor. In the other it resembled struggling through a maze, encountering numerous dead ends and seeing the same features over and over again, sometimes only to discover that it wasn’t even the right maze. Given enough time at the same workplace programmer "fingerprints" became evident in both patterns, often to the point of correctly guessing "this looks like X’s code".

The research began with curiosity about why these differences exist in people with similar qualifications, experience and even theoretical grasp of "good practice". Many of those whose work had been most impenetrable to the author were vocal advocates of entirely respectable, mainstream teachings. They were not ignorant of programming axioms, but maladroit in applying them. Some advice, such as adding comments for things "that are not obvious" (Hunt & Thomas, 2000, p.249), is easy enough to cite as a principle but demands a deeper empathy and understanding to put it into practice effectively.

Formally researching the question demanded first a clear impression from a wide cross-section of programmers about what matters in peer behaviour. The study which elicited this information is described in section 2. Its findings invited a new focus for the research (section 3): applying the methods used and important topics discussed to facilitate professional awareness of peer impact issues.

73
2. Initial study: What programmers want

One programmer’s experiences are insufficient foundation for research. To identify issues on which there is a broader consensus, an exploratory study collected the opinions of other experienced professional programmers.

2.1. Data collection interviews

A total of 28 participants from 7 companies across a variety of application domains were interviewed one-to-one so that they could speak freely in the absence of colleagues. It was important for both the ethical standards of the study and the willingness of companies to take part in it that individual opinions remained anonymous and commercial confidentiality was respected.

The participants were sufficiently experienced that any problems they reported were not the transitory difficulties of a novice. Each had been employed in the industry for at least 5 years, a level of experience at which managers are willing to commit customer-critical and mentoring tasks to developers (Zhou & Mockus, 2010). The goal was to find which peer behaviours have most impact (positive or negative) on professional programmers at this level in their day-to-day work.

The scope was intentionally broad so as not to pre-judge the outcome by focusing solely on, for example, existing notions of good practice. These can be "good" for a variety of reasons but the sole criterion of interest was the effect a practice has on colleagues; this criterion had to be carefully explained because participants came from a population more accustomed to discussing technical measures of impact.

Faced with a question about how the actions of peers (whether predecessors or current team colleagues) affect the difficulty of a task, programmers easily recalled a few recent or particularly significant events such as memorably bizarre variable names; recent events are usually more accessible than older ones (Baddeley, Eysenck, & Anderson, 2015, pp.51) and autobiographical recollection is also better for emotional or unusual incidents (Bower & Forgas, 2000, p.92). Incidents which were more mundane or about auxiliary activities such as writing tests or bug reports rather than code were less freely recalled. Interviews therefore started with an open question but followed it up with a card sorting exercise, a form of cued recall (Baddeley et al., 2015, p.304) to prompt participants to consider all the elements of their work.

The cards covered categories such as testing, bug reporting and interpersonal interactions as well as coding. Each card described a design decision or style of behaviour drawn from texts on good practice (Hunt & Thomas, 2000; Henney, 2010), from personal experience, or from Henney and other textbook authors’ Twitter complaints about contemporary events, such as this:

*Suffering the tyranny of the "TODO" comment. People: "do" your "todo"s. (Goodliffe, 2012)*

A draft set of cards was piloted with 3 experienced programmers to check that the process and content made sense and to determine the amount of time needed for the exercise. As a result an initial set of 88 was reduced to 54 cards by consolidating onto a single card wherever very similar content had elicited the same response. This set can be sorted within 20 to 30 minutes.

Participants sorted the cards on a 5-point scale according to how much their own work is impacted by the peer action:

- **Bad. Noticeable impact.** Makes my job harder/slower.
- **Bad. Slight impact.** Makes my job a little harder/slower.
- **Neutral.** Does not much affect my own tasks.
- **Good. Slight impact.** Makes my job a little quicker/easier.
- **Good. Noticeable impact.** Makes my job quicker/easier.
This was not intended to make a quantitative measurement cataloguing the topics in order of impact but rather to help participants assess the relative impact of peer behaviour and cue them to consider the whole breadth of activities, not just coding, that their work entails. The discussion then continued, talking about any new examples that had come to mind during the exercise and exploring the reasons for rating behaviours as ones that have good or bad 'noticeable impact'.

2.2. Analysis
The purpose of the analysis was to identify common themes in the interview data, addressing the research question by looking for areas of consensus among programmers about what matters in peer behaviour. A template analysis (King, 2012) was used because this method flexibly accommodates both an open discussion in which participants can raise any topic, and topics anticipated prior to the study (a priori themes).

The a priori themes used in the initial coding of the data were the card topics. Such codes are provisional; as each of an initial sub-set of transcripts was examined, the coding was refined to reflect participants’ own take on the cards and new codes were added for relevant material that did not fit an existing theme. Once the number of distinctly different new themes tailed off (this occurred after 3 transcripts), the codes were organised hierarchically under a smaller number of higher order codes describing broader themes to create the initial template.

The template was then developed by applying it to remaining transcripts and modifying when necessary: occasionally new topics were added, but changes were mostly refinements to existing codes (e.g. refining the description of a priori codes to reflect the scope of discussion that the related cards inspired). This process continued until, after eight transcripts, no further new and relevant information was emerging (i.e. saturation was reached). The template at the end of this process became the final template that informs the interpretation below. The remainder of the recordings were reviewed in audio format to identify particularly illustrative comments and check that they did not throw up common topics not covered by the template.

2.3. Findings
2.3.1. Overview
There were three parts to each interview: the initial part in which participants spontaneously recalled their experiences; the card sorting exercise in which they rated the impact of peer behaviours; and finally, a discussion structured by the cards rated to be in the most significant impact categories.

Feedback from participants suggested that the cards achieved the goal of presenting quite a comprehensive picture of a programmer’s work. Thus many of the spontaneous contributions naturally pre-empted card topics, and none of the themes explored below were drawn solely from discussions of the cards. When spontaneous topics not anticipated by the cards came up they were typically specific to the participant’s role. Sometimes they reflected team leader concerns rather than a programmer perspective as sought by the research question - although still identifying the same kind of issues. Other topics such as product deployment and support were relevant but not common because they depended on the nature of the business and the participant’s role within it.

2.3.2. Fine-grained code detail issues: the importance of naming
Fine-grained details about usage of programming language features rarely excited strong enough feelings to arise as spontaneous topics. This characteristic was also reflected in the card sort. While detailed code topics represented 50% of the cards, only two were rated as examples of ‘noticeable impact’ by the majority of participants: “Uses code comments in ways that aid understanding” and “Chooses identifiers which are not succinct, meaningful and distinct”. Both of these feature among the topics which were raised spontaneously.

Although participants agreed that good comments have significant positive impact there was a distinct absence of consensus in their detailed accounts of this topic. Opinions ranged from a programmer who felt that any comment has potential use as a clue to the code’s purpose, even if not completely accurate, to
the programmer who sets syntax highlighting to render comments in the same colour as the background, and thus invisible. Comments were the only divisive topic. Differences were in the perceived scale of the impact or the contingent nature of some responses. For example the boy scout rule (Martin, 2010), represented by the card "Tries to leave a module a bit better than when they checked it out", emerged as a good ideal to have in theory but not always a desirable behaviour in practice, depending on the context.

The other significant topic at the fine-grained level of code detail, choice of identifiers, had no such equivocation. Most cards at the "lines of code" level were rated as having some impact, but for the majority of participants it was only a slight impact. Syntactically correct code, however ill-written, is unambiguous and it would seem that these experienced programmers are adept at reading it; they recognise its shortcomings but these are not a particularly significant hindrance. However, no amount of experience can make meaningless identifiers meaningful. This participant’s recollection of the frustration they cause was a typical example:

Choosing vegetable names for variables... years ago there was somebody at another company that thought it was humorous. (Programmer B)

Commenting and naming aside, axioms of coding practice were not the dominant factors. This revealed that the topics of most concern to the participants lay at a higher lever of granularity than understanding lines of code; the challenge is to find the appropriate lines to look at in the first place.

2.3.3. The importance of communication for navigable code

In discussing the problems of navigating their way to the right lines of code, participants spoke of the value of early and frequent communication between a programmer and their peers as an ongoing and informal review process:

You always need to be listening to and talking to other people in the same team to make sure that you’re all on the same track. (Programmer A)

Such communication allows other team members to give timely feedback on approaches which make structure hard for others to make sense of, such as inventing a new and therefore unfamiliar way of doing things, using something that is unsuitable or unnecessary (e.g. a new and interesting framework) or simply writing code that is not needed. The feedback is essential:

To my mind development of computer systems is always a team oriented activity. There’s very few people who will sit down by themselves in a dark room and produce a good, useful and maintainable system. (Programmer B)

The impact of software created in the daylight by an open-minded creator constrasts sharply with that of idiosyncratic software produced by someone ploughing their own furrow:

Immediately understand what they’re doing... without having to sort of wonder why you’ve come up with some other alternative. So that kind of thing, that saves time I think. (Programmer C)

The more you dig deeper, the more you get exasperated by it. Because you’re thinking "what are, what have you done here? what, what monster have you created?" (Programmer D)

Along with some technical skill to write code that is not too complex, social conduct is important. Whether a peer ‘works in isolation’ or ‘asks questions’ emerged as two sides of the same coin for the impact they have. It is not obvious that quietly getting on with the job alone can adversely affect anyone else nor that asking questions of colleagues is of particular benefit to any but the questioner, but they play a key role in creating a solution that works, fits in and makes sense.
Productive communication also calls for an open attitude and an understanding that code belongs to more than just an individual:

You say "I don’t understand this bit of code" and their response basically is "well that’s because you’re an idiot". And by the time six people have knocked on the door and said "I don’t understand this bit of code" you really think they should start getting the hint that it doesn’t matter how clever they are compared to the rest of humanity, they’ve gotta write for their audience. (Programmer G)

They build an attachment to what they themselves have produced and a particular way of doing things. (Programmer H)

Having an open attitude includes being able to participate in a mature discussion of a question without giving or taking offence:

There’s someone in the current role who is very stubborn and that’s not helpful if you’re trying to come to something together. So he has this ability to make himself come across as affronted by any questioning. If you can do it in a way where you can separate the person from what you’re talking about you can have a fantastic discussion and we actually promote that, that’s fantastic. (Programmer E)

The phrase "trying to come to something together" summarises the essence of participants’ talk about fruitful technical dialogue: it is not about ego or point scoring but moving forward with an appropriate technical solution and a better understanding. Programming is an exercise in problem solving.

3. An experimental framework for professional awareness of peer impact (work in progress)

Outline plans for the next step in the research after the exploratory study originally proposed setting code modification tasks; these would focus on decision making in aspects of coding practice that emerged as important. The plans were set aside because themes concerning coding syntax were not the ones that had most impact on participants.

What did emerge were issues with attitude and culture, as briefly summarised in section 2. It also became evident that the methods used in the interviews had possibilities as a tool to facilitate discussion of these unstated and potentially sensitive issues. Two things suggested this. Firstly, participants frequently seemed to enjoy the card sorting process and many said as much. Secondly, one participant asked to take a set of cards to discuss in one of his team’s regular lunchtime seminars. Afterwards he reported:

I suggested we lay out the cards, and each pick one or two that made us think ’oh yeah, that reminds me of...’ and tell the story. Which went pretty well, with one or two very gentle nudges during the course of it, away from 'I think this is good practice' towards 'this is how it affected me on this occasion’. I found that really helped allow everyone to contribute, as well, including the less experienced and quieter people- it’s less intimidating to tell a story of your experience (others can’t argue that you’re wrong if you’re only reporting your experience) rather than nailing your colours to a particular good/bad practice mast.

His approach, akin to a focus group used to elicit all members’ opinions, inspired a way to apply the findings. Work is now in progress to develop this approach as framework for professional development workshops. Once it has been piloted it will be tested first at some of the original participating companies and then at some new ones. The conjecture is that the opportunity to pick topics already written on the cards offers a defusing effect which enables them to be raised more easily.

The cards will be revised to contain only common themes from the analysis of the interviews; group members will know that the cards reflect the consensus of very experienced programmers. Being offered
for discussion as materials in an independently designed workshop structure, topics are less likely to seem so personal and cannot reasonably be dismissed as "just your opinion". The opinion of a team member should not be so lightly dismissed anyway, but the knowledge that it is widely shared may help to discourage some from doing so and help others to share their views.

As well as encouraging little-discussed topics to be raised in a gentle manner, the workshop structure also allows the discussion to have local relevance. Selecting their own topics from the range available should allow teams to derive maximum benefit from the process by focusing on specific issues that affect them rather than, say, receiving a general presentation of the issues identified by the research. If the process is to bring about change it helps for the participants to have ownership of the process (O’Driscoll, Pierce, & Coghlan, 2006), so one goal of developing and studying the workshop structure is to see whether the discussion can be conducted without an external facilitator.

To evaluate the workshop procedure two distinct enquiries will be made in a questionnaire immediately afterwards: what participants thought of the process itself, and what changes they expect or hope the experience will bring about. A survey of team behaviours will be administered approximately a week beforehand to provide a snapshot of the team’s current state since the conduct of the workshop and its outcome are a function not only of the workshop itself but the context in which it is applied. This survey will be repeated 4-6 weeks later to see if any change can be observed, although this would represent a rather impressive effect. To assess whether small changes have occurred, participants will also be asked specifically whether they have observed any effects that they attribute to the workshop.

4. Conclusion
The participants in the initial study spoke freely about colleague behaviour in a way that is not easily achieved in the presence of colleagues without risk of disagreement, offence or embarrassment. It may be possible to have such a conversation outside the workplace but the interviews offered an unusual opportunity to do so confidently with someone who understands the field well and is genuinely interested. They responded to the opportunity with honest reflections and insights. It was apparent from their parting words that they also found the process interesting and enlightening.

While it is impossible to create quite the same environment for a group, the workshops will allow others to benefit from the insights that the interviews helped to draw out. This research has captured a perspective that is often overlooked among more easily measured indicators of good practice. Applying it requires more than a definition of principles to follow; just as with programming axioms, effective application depends not on declarative knowledge but a deeper understanding. By facilitating communication about the impact of behaviour, the ongoing work aims to create just that.

5. Acknowledgements
Thank you to Sherry Jeary, Keith Phalp, Jacqui Taylor and Kevin Thomas for their sage advice throughout this research and for kindly reviewing this account of it. Thanks also to the PPIG reviewers, who not only improved this paper but also generously gave advice that will be invaluable in writing up the results of the research.

6. References


Abstract

Visual programming systems, including blocks languages like Scratch (Maloney et al., 2010) are widely used to introduce children and other learners to programming, but in their present form they cannot be used by blind people. Following up work on the nonvisual dataflow language Noodle (Lewis, 2014), and building on the Blockly language system (https://developers.google.com/blockly; Fraser, 2015) we are building a nonvisual interface to a blocks language. We present a pseudospatial interface for the language, similar to that of Noodle, and compare it with other approaches to accessibility for blocks languages.

1. Introduction

There is a surge of interest in introducing programming to children, through initiatives like the BBC micro:bit in the UK (https://www.microbit.co.uk/), and the international Hour of Code program (https://hourofcode.com/us). Some of the most popular platforms for this work are blocks languages, in which program elements are presented as blocks, somewhat like jigsaw puzzle pieces. Blocks have tabs and sockets that suggest how the elements can be assembled. The constraints in program grammar are conveyed as rules of fitting shapes together, rather than as syntax rules, so that the often-frustrating syntax errors in textual languages are eliminated. Examples of blocks languages include Scratch (https://scratch.mit.edu/), Snap (http://snap.berkeley.edu/), MIT App Inventor (http://appinventor.mit.edu/), Microsoft Block Editor for the BBC micro:bit (https://www.microbit.co.uk/create-code), and many more.

Unfortunately these languages can’t be used by learners who can’t see, an issue with visual languages generally. While one might think that using visual languages is an inherently visual activity (as the name suggests), in fact there are approaches to delivering the conceptual benefits of these languages, that is, the logical structure and meaning of programs, in nonvisual representations. Some of these approaches derive from the work of T.V. Raman, who has suggested that many, if not all, visual tasks can be substituted by nonvisual ones (Raman, 1996; Raman and Gries, 1997; see also Lewis, 2013.) Lewis presented work on a nonvisual dataflow language, Noodle, based on Raman’s ideas, at PPIG 2014 (Lewis, 2014).

2. Approaches to accessibility for blocks languages

2.1. Screen reader access

A common approach to accessibility for blind users is support for screen reader programs, software tools that render textual material displayed on a computer screen as synthesized speech, and support interaction with common controls and widgets. A team at Google is creating Accessible Blockly, (https://blockly-demo.appspot.com/static/demos/accessible/index.html), a system that presents blocks and blocks programs as HTML structures that can be read by a screen reader. This approach has the advantage that some blind users are skilled screen reader users, and are thus already familiar with all the controls and conventions needed to navigate and manipulate the HTML. Stephanie Ludi (2015) has proposed a related approach, also based on Blockly, that will also support visual access.

Schanzer’s Bootstrap system (http://www.bootstrapworld.org/) is being extended to support a blocks language made accessible by a screen reader in a different way (Emmanuel Schanzer, personal communication, July 15, 2016). The Bootstrap language is normally presented textually in a Web page, in a form that a screen reader can access. By using alternative CSS style rules these textual
programs can be presented as structures of blocks, and can be edited in that form (though the system does not use shape constraints to guide program creation.)

2.2. Beyond the screen reader

As Raman (1996) points out, the screen reader approach starts with a presentation of content intended for sighted people, and seeks to provide nonvisual access to that presentation. But it may be possible to provide a different presentation, nonvisually, that is superior to the visual presentation, in some respects. For example, Lewis (2014) suggested that a nonvisual dataflow language can make it easier to trace relationships between program elements than the lines used in visual presentations.

This argument supports the investigation of presentations that are not based on presentations for sighted users, but it does not rule out the use of screen readers. The representation presented by the screen reader can be structured differently from that shown to sighted users. That is, the screen reader can be used as a means of navigating and manipulating a presentation that need not be the same as what sighted users work with. This is true of Accessible Blockly, in that its HTML representation includes many controls that are not presented to sighted users.

2.3. Pseudospatiality

Screen reader navigation of HTML is serial and hierarchical. That is, one moves up and down a list of elements, some of which are sublists that can be entered and navigated as well; the sublists can contain subsublists, and so on. Commands are provided to read all material, or to read only elements at a given level, to skip to elements of a specified type, and more. While this system is quite flexible, some designers of applications for blind users have moved outside this structure to support navigation in a virtual two dimensional structure, with operations provided that move right to left or up and down. For example, the TWBlue Twitter client (http://twblue.es/) uses arrow keys to navigate an invisible two-dimensional array of content. Noodle (Lewis, 2014) uses a somewhat similar scheme. (It is sometimes suggested that spatial arrangement would be meaningless for blind users, but this is simply not true. Spatial relationships are fundamental in a vast range of nonvisual activities, including walking around, reaching for objects, and so on.)

In systems of this kind feedback for navigation and manipulation is provided by synthetic speech. For example, the system will speak a description of a content element when the user moves to it. The synthetic speech can be provided by the system itself (so-called self voicing applications) or, if the user normally uses a screen reader, the speech can be produced by the screen reader (by placing data in a “live region” that the screen reader monitors.)

A navigation scheme can be pseudospatial, rather than simply spatial, if the geometry of movement is distorted. In TWBlue, moving “left” from a content element might take one to the top of the column to the left, rather than to an element directly to the left. In Noodle, following an edge “to the right” might go to an element that could also be reached by moving “up”, if there is a loop in the dataflow.

We describe below Pseudospatial Blocks (PB), a blocks language presentation based on arrow key navigation. In PB a toolbox of blocks is to the “left” of a workspace (as is common); blocks in either area, and parts within blocks, are arranged “vertically”. The arrangement is pseudospatial in that (for example) moving one step “down” from a block, or moving one step “down” in a list of parts of the same block, would not reach the same location.

2.4. Tactile presentation

Richard Ladner (personal communication, June 1, 2016) has proposed using a touchscreen to permit learners to explore and operate on blocks by touch. Because many blind children are familiar with touchscreen devices and the screen reader programs used with them, this presentation should be easy for children to learn to use. More radically, it may be possible to create blocks languages with physical blocks; see e.g. http://cubescoding.com/ and https://www.primotoys.com/ for beginnings.

2.5. Presenting fit constraints

As mentioned earlier, a distinctive feature of blocks languages is the use of shape fitting to indicate how program elements can legally be put together. For example, a block that produces a value has a different shape from a block representing a statement; two statement blocks fit together to form a sequence, but a value block will not fit there. How can this information be presented nonvisually?
In Ladner’s tactile system shape information will still be presented, but will be detected by touch, as children explore the outlines of the blocks with their fingers, receiving auditory feedback when they touch the block. Ladner proposes to enlarge the blocks to make it easier to explore their outlines.

In Accessible Blockly and PB shape information is not presented as such. Rather, the systems enforce the placement rules that the shapes would convey: a block of a given kind can only be placed where it is legal (see more below on PB’s treatment of his issue.) In this respect these designs both follow Raman’s suggestion that presentations for blind users should not be aimed at conveying the information presented visually to sighted users, but should focus on the underlying content.

3. Pseudospatial Blocks (PB)

3.1 Design and implementation

We based the design of PB on an analysis of user tasks in Scratch (the most widely used blocks language). We aimed to support these tasks, without requiring a screen reader, using common keyboard commands: arrow keys to navigate, space to select, ENTER to open (or move inside) a complex entity, and period to operate. Responses to all commands are provided in synthetic speech.

Like Scratch, PB presents a toolbox of program elements, navigated by up and down arrows. Selecting a block causes a copy to be added at an insertion point in the workspace. Moving to the right takes one into the workspace, where up and down arrows allow one to move among the blocks in the program. Pressing the period key executes the program. For each key press the user receives audio feedback about the current location in the program, delivered as synthesized speech.

Some blocks, for example a block for repeating a sequence of operations, have statement blocks nested within them. To create such a structure the user navigates to the outer block and then presses ENTER; the up and down arrows then move among the elements of the block that can be edited, including the place where nested statements can be inserted. Pressing ENTER at this point establishes the insertion point there, so that selecting a statement block in the toolbox will add a nested statement.

When choosing a block to insert, the user is only presented with block choices that are legal for insertion at the current insertion point. For example, only statement blocks, and not expressions, can be inserted within a repeat block. This distinction is enforced in visual blocks languages by shape constraints. Similar operations support other forms of editing. For example, if a statement includes a dropdown menu of alternatives (such as musical notes), one ENTERs that statement, navigates to the dropdown, ENTERs that, navigates the alternatives, and selects the desired one.

The implementation of PB is based on the Blockly library (https://developers.google.com/blockly/; Fraser, 2015), an extremely flexible and widely used platform for creating blocks languages. Blockly supports an xml representation of blocks programs, and can generate code for these in a number of languages. This support makes it easy to create new interfaces like Accessible Blockly or PB. PB is implemented as a Web application that builds a JSON (JavaScript Object Notation) representation of programs, and then uses Blockly to generate JavaScript code.

3.2 Status of PB

PB is a running proof of concept program that supports a small number of block types, most of which produce sounds, adequate to create very simple melodies. We have focussed on programs for producing sounds for the obvious reason that the results are easy for blind learners to understand, whereas common visual activities, such as producing animations or turtle graphics, are themselves inaccessible, apart from the accessibility of the language used to specify them. Music may also have conceptual benefits as an application domain for learners, with rich structure; see e.g. Blackwell and Collins, 2005; Shapiro et al., 2016.

Currently, hand coding is required to support new blocks, even when these have been defined for other Blockly applications, but in future we hope to be able to generate the representations needed by PB automatically from Blockly’s representations. With that improvement in hand we plan to support a wider range of operations on sound. We will then undertake user testing and iterative redesign.
4. Discussion
Creating accessible blocks languages may have impact in a number of ways, beyond the obvious value of supporting blind children learning to code. As Jamal Mazrui (remarks at M-Enabling Summit, June 2012) and others have observed, making programming more accessible to people with disabilities will allow them to satisfy their technology needs themselves, rather than relaying on other people to understand and respond to their requirements. Nonvisual programming interfaces may be more broadly useful in allowing people to program on small screen devices, something difficult today.

In pursuing these aims we should cast a wide net, as represented by the three approaches described above. We are far from understanding the tradeoffs in this design space, so we should pursue, and compare, multiple lines of work. Here are just some of the important issues we need to understand:

How important is it that interfaces used by blind children can also be understood by their sighted classmates? Screen-reader based approaches, using representations very similar to those provided for sighted users, may have an edge there, as may Ludi’s design. How important is it to support people who are blind but do not use screen readers? This is common today among older people who become blind, but may less often be an issue for children. When and how can nonvisual interfaces be superior to visual ones? In particular, can the use of shapes to express constraints on the assembly of program elements be improved upon, by providing easy access to elements that fit a selected insertion point? Can non-visual interfaces support increasingly complex programs, as learners mature?

5. Acknowledgements
We thank Sina Bahram for telling us about TWBlue, and for suggestions for structuring pseudospatial interfaces; Richard Ladner for sharing his proposal for tactile support for blocks languages; Neil Fraser and Madeeha Ghorii of Google for their support for our use of Blockly, and sharing their work on Accessible Blockly; Ben Shapiro and Annie Kelly for advice on platforms; and Andy Stefik and Brian Harvey for convening a workshop on accessibility of Blockly at MIT in April, 2015, that provided the inspiration for our work. We thank the PPIG reviewers for helpful suggestions.

6. References

Fraser, N. (2015). Ten things we've learned from Blockly. In Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE (pp. 49-50). IEEE.


A neurofeedback system to promote learner engagement

James Lockwood  
Dept. of Computer Science  
Maynooth University  
James.lockwood@nuim.ie

Aidan Mooney  
Dept. of Computer Science  
Maynooth University  
aidan.mooney@nuim.ie

Susan Bergin  
Dept. of Computer Science  
Maynooth University  
susan.bergin@nuim.ie

Abstract

This paper describes a series of experiments that track novice programmer’s engagement during two attention based tasks. The tasks required participants to watch a tutorial video on introductory programming and to attend to a simple maze game whilst wearing an electroencephalogram (EEG) device called the Emotiv EPOC. The EPOC’s proprietary software includes a system which tracks emotional state (specifically: engagement, excitement, meditation, frustration, valence and long-term excitement). Using this data, a software application written in the Processing language was developed to track user’s engagement levels and implement a neurofeedback based intervention when engagement fell below an acceptable level. The aim of the intervention was to prompt learners who disengaged with the task to re-engage. The intervention used during the video tutorial was to pause the video if a participant disengaged significantly. However other interventions such as slowing the video down, playing a noise or darkening/brightening the screen could also be used. For the maze game, the caterpillar moving through the maze slowed in line with disengagement and moved more quickly once the learner re-engaged. The approach worked very well and successfully re-engaged participants, although a number of improvements could be made. A number of interesting findings on the comparative engagement levels of different groups e.g. by gender and by age etc. were identified and provide useful pointers for future research studies.

1. Introduction

Computer Science (CS) non-progression rates in Ireland are alarming with a large number of students failing to progress every year. Currently non progression rates are at 25% in CS, which is significantly higher than the national average of 16% (Mooney, 2010). It is well acknowledged that a main contributor is introductory programming modules, a staple in most first year CS courses. At our university numerous methods have been tried to improve performance and retention to varying degrees of success (Mooney, 2014; O’Kelly et al., 2004; O’Kelly et al., 2004, Nolan et al. 2015). One of our recent initiatives was the development of self-paced video tutorial on threshold programming concepts (Hegarty-Kelly, 2015). These videos were introduced to allow students to revisit core concepts in their own time, at their own pace. The goal of this paper is to measure how well students engage with learning videos using an EEG device to monitor engagement and where a student disengages prompt them to reengage using neurofeedback. To this end a number of experiments were implemented to monitor engagement during a threshold concept video and also during a mind-control game. The rationale and background for this work are provided in the following section.
2. Background

A review of the literature was carried out on (1) the relationship between engagement and learning, (2) the use of electroencephalogram (EEG) devices (especially the Emotiv EPOC used here) for monitoring neural activity during cognitive tasks, and (3) the use of videos as a learning tool.

An EEG is a non-invasive technique to measure activity in the brain. The EPOC is a 14 channel EEG device for research. It is a dry (uses a small amount of saline) high resolution consumer grade EEG system. A number of verification studies have been carried out comparing the EPOC to a clinical EEG but there has been little independent study as it is a relatively new technology. However, proprietary claims suggest its appropriate for research and, of the limited studies, all have found that it performs well and can be used in place of more expensive units. Taking this into consideration the results obtained can be considered accurate for example, Ekanayake (2010) concluded that the EPOC does capture real EEG data but has considerable noise. This noise can be minimised using different techniques such as averaging. It was also found that electrode placement is fairly fixed and thus this reduces the number of studies it can be used for. Andujar (2011) used the EPOC in a study that compared the engagement level of participants studying a paper handout with using a video game to learn about the Lewis and Clark Expedition (www.classbraingames.com). The game disseminated information about the expedition and the control group received the same information in the handout using only text. Twenty-six participants took part (13 in the experimental group and 13 in the control). Each group had 20 minutes to complete the learning task. Participants wore the EPOC during the task and engagement levels were measured. The results of the study suggested that educational video games might not be significantly engaging and found that learning from handouts could be better for retaining information. This is an interesting finding and the closest study we could find on the use of the EPOC to measure engagement. But this study uses a video game and not a video tutorial and it could be argued, that a tutorial video reduces any cognitive load associated with aspects of the games or story-line issues which might impact on learning.

Aside from EEG studies on engagement other empirical studies on engagement have highlighted its importance for learning and comprehension, some examples follow. Carini et al. (2006) found that student engagement is linked positively to desirable learning outcomes such as critical thinking and grades. In a study on the use of clickers in the classroom Blasco-Arcas (2013) found that increased engagement through the clickers improved student learning. Other studies have shown that increased engagement in reading leads to higher comprehension (Guthrie & Wigfield, 2000 and Miller & Meece, 1999).

With respect to using educational videos to improve student learning, Merkt et al. (2011) conducted two complementary studies, one in the laboratory and one in the field, comparing the usage patterns and effectiveness of interactive videos and illustrated textbooks when German secondary school students learned complex content. They used two separate videos of differing degrees of interactivity (the second of which they made themselves) and an illustrated textbook. Both studies showed that the effectiveness of interactive videos was at least comparable to that of print, in contrast to previous studies working with non-interactive videos. More recently, Willmot et al. (2012) show that there is strong evidence that creating an on-going video report during projects
can inspire and engage students when incorporated into student-centred learning activities through: increased student motivation, enhanced learning experience, higher marks, potential for deeper learning of the subject, enhanced team working and communication skills, and by providing a source of evidence relating to skills for interviews. These studies indicate that videos can be used as an enhancement to a lesson, or unit of study. Furthermore, Guo et al. (2014) provided a list of recommendations on what makes a video engaging and beneficial. To do this they used both data analysis of 6.9 million video watching sessions and interviews with video production staff. These recommendations and findings include: shorter videos are much more engaging, videos where instructors speak relatively fast and with high enthusiasm are more engaging and that videos should incorporate motion and continuous visual flow in tutorials, along with extemporaneous speaking. Pre- and post-production planning is essential as topics can be segmented into smaller sections. They also argue that filming in an informal setting is more engaging than a recording studio. The recommendations provided above influenced the development of the threshold video described in Hegarty-Kelly et al. (2015).

The goal of the work presented here is to monitor, track and respond to changes in engagement whilst cognitive tasks are being attended to by gathering neuro-physiological data from the student. A neurofeedback system like this, which can detect, track and respond to these changes in an online learning environment could be beneficial to learning, especially online learners in the absence of a teacher to monitor and gauge attention. This is especially true and timely with the large growth of MOOC’s (Massive Open Online Courses) in recent years.

This paper describes a neurofeedback system which monitors and responds to engagement levels in two cognitive tasks. First, whilst watching a tutorial video the feedback system would pause the video if a learner disengaged and only restart after re-engagement had occurred. This system could be of huge benefit in MOOC’s and long-distance learning courses which use videos as a major learning tool. The second task is a maze game in which a caterpillar moves around a maze, and the speed at which this happens is related to a participant’s engagement level. This application has the potential to be used in therapy settings for conditions such as Attention deficit hyperactivity disorder (ADHD). This engagement therapy taps into the brain’s ability to re-wire its own connections (neuroplasticity) and has been shown to be a superior method of treatment for ADHD than classical medicated interventions of the past (Fuchs et al. 2003).

3. Method
3.1 Hardware and software Description

The Emotiv EPOC was designed for practical research applications and comes with a software suite that measures several emotional states including engagement, as well as boredom, excitement, frustration and meditation level in real time. Two applications were developed. The first monitored a user watching a video whilst wearing the EPOC and allowed interventions to take place depending on a user’s engagement level. The second was a maze game, where an object would travel around a maze and the speed at which it moved was determined by the user’s engagement level. The idea for this game was based on similar neurofeedback games developed for children with ADHD such as that of Fuchs et al. (2003). Their version of pac-man was developed for children with ADHD where if they start to lose focus on the game pac-man starts to fade. The idea of the maze game developed here,
in the Processing language (Foundation, 2016), was a very simplistic idea in which a caterpillar-like object moves around a maze. The speed of the caterpillar is related to the participants level of engagement, the higher the engagement level the faster the caterpillar moved, the lower the the engagement, the slower it moved.

3.2 Participants, Experimental design and pilot study
Ethical approval was sought and granted for the study: all participants signed a consent form and an information sheet was provided prior to the session with questions. In total 21 undergraduate (final year) students along with 5 postgraduate students were recruited to participate in the study (16 of the undergraduates and 4 of the postgraduates students studied CS). A video of 8 minutes duration on if statements, was used. This is material that all 4th year CS students would be extremely familiar with and so would potentially increase the likelihood that the participants would disengage and show the capabilities of the system.

3.3 Instruments:
A post-experiment questionnaire was used to gather feedback on perceived engagement for both the maze and video tasks based on a survey by Wigfeld & Guthrie (1997) (further adapted by Hyun-Gyung Lee (2012)), which all participants filled out directly after doing both the video and maze tasks (see Lockwood & Bergin 2016). All data was anonymised and each participant was given a unique identifying code. This code was used by participants in another survey that gathered demographic and academic background information.

3.4 Experimental Design:
All participants completed the survey and took part in all of the tasks in the following order: Calibration, Video, Video Survey, Maze and Maze Survey. Although counter-balancing an experiment that involves discrete components can be important, the video was always shown first as we were most interested in it.

Before the main study a three-person pilot study was carried out to ensure that each of the experiments worked correctly and to test whether the selected baseline engagement value (set at 0.5, the mid-point value for engagement in the proprietary software) was reasonable. After two participants it was clear that this crude approach to measuring engagement was insufficient and an individualised calibration procedure was needed as engagement levels differ greatly from person to person. As such a calibration process was carried out where (1) the participant fixated on a black cross on a grey screen for sixty seconds, and (2) the participant closed their eyes for sixty second. During the latter, participants were instructed to try to relax and to not focus on anything in particular. Whilst their eyes were closed the program took a running average of their engagement level for that minute and this value was used as each participant's baseline value. In theory, when the participants eyes are closed they should have been disengaged and this was confirmed in their levels. This baseline was then used as the threshold for whether the video should pause and also as to how slow the caterpillar would move in the maze game. This means that the video would pause and would not restart and, similarly, the caterpillar would continue to go slowly, until the level was above the threshold value. The formula used for the caterpillar was as follows:

\[ \text{if the engagement level was above the threshold the caterpillar moved at } 10 \times \text{Engagement level} + 1 \text{, if it was below this level then it moved at } 1 \times \text{Engagement level} + 0.1. \]

After the application had been designed, a third and final
participant took part in the pilot study; all three experiments were conducted on this participant. This calibration task was found to work well and interacted correctly with both the maze and video applications, however it is a simplistic technique and future work should seek to validate or improve upon it.

3.5 Experiments

After building the experiments and running the pilot study, the next step was to carry out a main study using the developed applications. Nineteen CS students and six non-CS students participated in the study. This resulted in twenty usable data sets (16 CS, 4 non-CS). The experiment took ~30-40 minutes to complete. Below is a full demographic breakdown of the participants from who data was successfully collected.

![Demographic breakdown](image)

Figure 1 – Breakdown of some of the demographics of the participants.

The experiments were run on a Lenovo IdeaPad U330p laptop (8GB RAM, Intel i5 processor running Windows 10) which participants had on a table in front of them. The first author remained in the room to monitor the headset’s signal as occasionally it would drop out, that is, one or more of the contacts would move out of ideal placement. After reading the information sheet the participants were instructed, and helped as appropriate, on the correct placement of the EPOC on their heads. Once a good signal was achieved (i.e. the 14 channels were detecting a good signal) participants took part in the calibration procedure. After an individual baseline had successfully been obtained through the calibration system, they then viewed the video, being advised to watch the video and that if it paused, to try to refocus on the task. After completing the video, the participants completed an engagement questionnaire. If needed, extra saline solution was applied to the sensors of the EPOC. The participants then completed the maze task and associated engagement questionnaire.

Twenty full data sets were successfully obtained over a period of four weeks. Six more participants took part but their data was either corrupt, too noisy or not obtainable. Four of these were female participants and the cause was most likely the amount and thickness of their hair prevented the EPOC’s sensors establishing a good connection. Another contributing factor may have been head size, that is, they had slightly smaller heads than other participants and so sensor contact may have been sub-optimal, this is down to the restricted movement of the sensors and flexibility available of the EPOC compared to medical EEG devices.

4. Analysis

Statistical analysis including frequency tests, t-tests and P-values were performed on the data collected. Due to the small sample size, and even smaller breakdown, anything with a p-value of less than 0.15 (i.e. Alpha = 0.85) was treated as potentially interesting. Although the sample size is small (n=20), some of the findings are
significant enough to warrant further investigation. It should be taken into account that this was an initial experiment and project and so the findings and methods can be built upon and improved in future studies.

A point worth noting is that the uninteresting nature of the video content and the production quality were almost unanimously confirmed in post-video response forms (19/20 participants responded to the question “What did you find that was boring or too easy?” (participants were instructed to answer this question and all others in regards to the task completed, either the video or maze) with comments that the video itself or the content was boring or easy). At the beginning of each experiment participants were told what would occur if they disengaged, but were not informed to try harder than normal to engage, just to watch the video/maze and try to re-engage if the video paused or the caterpillar slowed.

4.1 Calibration

Finding 1 – During the calibration task, participants who wear prescription glasses had lower average engagement levels during the fixation cross section

Participants who wear prescription glasses (n=6) had a statistically lower average engagement level during the minute in which they focused on a fixation cross than those who don’t (p value = 0.148). These same participants also had a greater range in their average values (eyes open/shut) than those who don’t wear glasses (p-value = 0.131). This tendency towards an effect may suggest something interesting such as, for example, that glass-wearers have learnt to concentrate harder when instructed than non-glass wearers but ultimately did not concentrate as well as non-glass wearers. Such a hypothesis would have to be tested for merit however.

Finding 2 – Participants over the age of 23 had a lower average baseline during the fixation cross section

Mature participants (over 23) had a significantly lower average baseline engagement level when staring at the fixation cross than participants < 23 years old (p-value = 0.115). This could be an interesting finding as mature participants seem to disengage with both the video and maze much less than non-mature participants.

Finding 3 – Those with previous programming experience had significantly higher ranges than those who had no previous experience

Those who have previous programming experience had a larger range (eyes closed compared to eyes open) on average than those with none (p-value = 0.057) during the calibration task. They also had a higher average level with their eyes shut, however this was not as statistically significant (p-value = 0.275).

4.2 Video Data

With regards to the video, related data initial analysis was focused on the number of times the video paused during the duration of the video. This indicated a drop below the baseline level recorded during the calibration task, referred to as a “drop” or “dropping” in the next few sections. Of the 20 participants, 7 watched the whole video without it pausing once. Of these seven participants, a brief summary of their breakdown is follows: 1 female; 6 males, 2 mature, 5 non-mature, 2 postgraduate students, 5 undergraduates, 1 non CS student, 6 CS students.
Across the whole population the mean number of drops was 12.55, the median was 9 and the mode was 0, the maximum value was 46 (see Figure 2).

![Pauses in the Video](image)

**Figure 2-** The number of pauses (drops) during video playback.

Some points of interest from this data include the following:

**Finding 1 – Participants with lower baseline engagement levels paused the video significantly less than those with higher baseline levels**

On both tasks those who had lower engagement levels with their eyes shut (the value taken as a baseline) caused the video to pause less and had similar results with the maze game i.e. caused it to move slowly for less of the time. To compare the two groups, the average of the whole populations’ baselines was determined ($\bar{x} = 0.501407$) and the group was then split using this value. The baseline values were the unique values that were calculated for each participant during the calibration process. It should be noted that the sample size was small and the calibration task was crude thus more work is required to determine the validity of these findings.

All participants with a baseline below the population average ($n = 9$, referred to here as the Low Group (LG)) were compared with those above it ($n=11$ referred to as High Group (HG)). The LG dropped significantly less than the HG, with the LG pausing on average 2.333 times whereas the HG paused on average 20.9 times (p-value = 0.002). This is interesting as it could suggest those who are able to disengage before completing the tasks are more likely to succeed at the cognitive task. This suggests that learning meditative techniques and applying these before a cognitive task (test, studying etc.) can improve performance (Zeidan et al., 2010).

**Finding 2 – Participants who rated their Java level as lower paused significantly more than those who rated their level as high**

Participants who rated their Java level as 1-3 ($n=10$) (with 1 being “Very poor” and 5 being “Very good”) paused significantly more than those who rated it 4-5 ($n=10$). On average those who rated their Java level as lower paused 18.1 times with those rating it higher pausing on average 7 times (p-value = 0.0852). The video is designed for beginner programmers, and so in theory those with greater knowledge and experience of the language “should” find it less engaging. However, most of the participants responded that the video was boring. This finding also holds true if you remove those who have never programmed before; for four of the participants this was true and they rated their level as 1 (no-one rated their level as 2). If you take those participants ($n=6$) who rated their Java...
level as 3 and compare it to those rated 4-5, those rated 4-5 paused significantly less, on average 19.33 times for level 3 participants compared to 7 for 4-5 (p-value = 0.125). It would be interesting to take this experiment and run it on those learning to program (possibly in two groups, one who want to continue CS and one just making up credits) to see if this holds for the target population.

**Finding 3 – Mature participants (over 23) caused the video to pause less**

Mature participants (n=4) caused the video to pause less on average than non-mature. Although not statistically different it was numerically quite different, with matures averaging 6.5 drops compared to 14.06 with non-mature. Further investigation in this area is warranted.

**Finding 4 – No difference in people whose self-perceived engagement level was low (1-2) or high (3-5)**

On another note, no statistical difference was found between those who responded to how engaged they were during this task with a 1-2 rating or a 3-5 rating (with 1 being “Not at all” and 5 being “Completely/Always”). Those who rated how engaged they were as 1-2 dropped on average 11.4 times compared to 13.7 times with those who rated engagement as 4-5 (p-value = 0.732). This is interesting as it could mean that people felt they were more engaged than they actually were, or conversely that they were less engaged than they felt. From anecdotal evidence and observation, many participants felt they were focusing and engaging in the video but still caused it to pause, however all agreed the video was boring and unengaging. This could mean that even though they felt they were engaging, the EPOC and associated baseline engagement level did not indicate they were. It could also mean that people are poor self-reporters, however due to the limited number of studies into the accuracy of the EPOC it is hard to know which of these is more likely.

4.3 Maze Data

During the maze game if participants’ engagement levels dropped below their baseline level, the speed of the caterpillar dropped dramatically. The speed was constantly changing due to the level but dropped drastically if this occurred. This is again referred to as a drop.

Of the 20 participants, 7 played out the duration of the game without it dropping once. The profiles of the seven participants were: 7 males, 3 mature, 4 non-mature, 3 postgraduate students, 4 undergraduates, 7 CS students. Five of the 7 who didn’t drop during the maze task also didn’t drop during the video task.

The mean number of drops for all was 3.15, the median was 3, the maximum value was 10 (see Figure 3).

![Figure 3 – The number of times the caterpillar slowed (drops) during the maze game](image-url)
Finding 1 – Those with higher baseline engagement levels dropped (slowed the caterpillar) less than those with lower baselines.

As previously noted, a significant difference was found in the number of between the LG and HG. Those in the LG dropped (or slowed) on average 1.11 times compared to 4.81 times for those in the HG (p-value = 0.021).

Finding 2 – Those who rated their Java level as low (1-3) dropped more than those who rated their level as high (4-5)

Similar to the video data, those who rated their Java level as low (1-3) caused the caterpillar to slow more times than those who rated it high (4-5). On average this was 4.3 times compared to 2 times (p-value = 0.126). It is worth noting that those who rated their Java level as 1 (n=4, all the non-CS participants) had the highest average drops in the maze with an average of 5, compared to 3.83 with level 3 (6/20), 2.5 with level 4 (8/20) and 0 with level 5 (2/20). This is interesting and warrants further investigation to identify why this is observed.

Finding 3 – Mature participants (over 23) dropped less than those under the age of 23

Another similarity with the video data, with greater statistical significance, is that mature participants caused the caterpillar to slow less than non-mature. Matures caused it to slow on average 1 time compared to an average of 3.69 for non-mature (p-value = 0.149).

Finding 4 – No statistical difference in those who found the maze more engaging, interesting or challenging than those who did not

In line with the data collected from the video task there was no statistical difference between participants who found the task more or less interesting, more or less challenging or more or less engaging.

4.4 Maze vs Video

In terms of survey feedback for both the maze and video, participants were asked to rate three survey items from 1-5 (see Lockwood & Bergin 2016 for details) as follows: (1) I was challenged by this task, (2) This task was interesting to me, (3) This task engaged me. Interestingly and insightful for future work, all participants found the maze more challenging, engaging and interesting than the video (see Figure 4).

![Figure 4 - Number of people who rated the maze/video more or less in each question.](image-url)
5. Conclusions

This project has shown that a consumer grade EEG device can be used to track and respond to a learner’s engagement level. This background work could be expanded into a more robust system which could help improve both a learner’s experience and their engagement.

The findings from the experiments provide interesting insight into student engagement whilst watching tutorial videos. From the analysis the most significant findings that warrant further investigation are:

- Those with lower baseline engagement levels caused both the video and the maze game to drop less than those with higher baseline levels. This could support evidence that practising meditative methods increases cognitive performance.
- Those who rated their Java level as low caused both the video and the maze to drop more than those who rated their level as high. This is interesting especially given the nature of the video (a beginner Java tutorial) and it would be useful to investigate this with similar tutorial videos.
- Mature participants (those over 23) caused the video and maze to drop less than those under the age of 23. This is another interesting finding as potentially it could show, with further and more intensive study that those over the age of 23 may find it easier to engage with cognitive tasks than those younger.

As stated, given the small sample size, the findings, although interesting, require further investigation.
6. References:


Programmers’ experiences with working in the restricted-view mode as indications of parafoveal processing differences.

Pavel A. Orlov  
1) School of Computing  
University of Eastern Finland  
2) Engineering Graphics and Design Dept.  
Peter the Great Saint-Petersburg Polytechnic University  
paul.a.orlov@gmail.com

Roman Bednarik  
School of Computing  
University of Eastern Finland  
roman.bednarik@uef.fi

Liudmila Orlova  
Emperor Alexander I St. Petersburg State Transport University  
mila.e.orlova@gmail.com

Abstract
Understanding of programmers’ attention provides benefits for developing comprehension models, bug-prediction models, for increasing software productivity, and facilitating programming education activities. Here we conduct a gaze-contingent study involving a real-time restriction of the viewing area and compare professionals’ and novices’ verbal feedback after the parafoveal view was restricted during source-code comprehension. Such information provides clues about the differences in parafoveal processing during programming.

We recorded the participants’ verbal feedback and divided their answers into seven topics and types. Then we analysed the differences between the answers given by the experts and the novices. We compared the proportion of utterances used to comment upon a certain topic by each group. This allowed to identify the relative importance of a particular topic. Some topics turned out equally important for both the experts and the novices: 1. analysis of the working process, 2. personal evaluation of the source code, and 3. the use of life-hacks. The experts and the novices used a different proportion of utterances to comment upon the experimental conditions and visualizing, which was unexpected. The restriction of the extrafoveal area evoked a more emotional response from the expert programmers. At the same time, the novices perceived the restriction of semantic information in the extrafoveal area in a less emotional way. We suggest that the explanation is that the experts have increased expectations for the information to be obtained from the attentional objects located in the extrafoveal area. The masking of parafoveal objects makes them less easily available, which runs counter to the experts’ expectations and, therefore, produces a stronger emotional feedback. This illustrates that it is a common thing for expert programmers to use the parafoveal information during source code comprehension.

1. Introduction
1.1. Background
The program-code comprehension is a high-level cognitive task. At the same time, it involves a number of low-level processes – e.g., perception, attention and memory (Hoc, Green, Samurcay, & Gilmore, 1990). One of the areas of the research of programmers behavior is advancing the understanding of the links between the low-level and high-level processing, such as the links between eye-movements and program comprehension.

The role of attention has been greatly investigated in programming. Programmers attend to various elements of the source code in different order than in reading, forming strategies of task solving (Crosby & Stelovsky, 1990). The understanding of programmers’ behaviour as an attentional strategy provides benefits in a number of ways: developing bug-prediction models, enhancing teaching and learning, and increasing software productivity (Fritz, Begel, Müller, Yigit-elliott, & Züger, 2014; Busjahn et al., 2014; von Mayrhauser & Vans, 1997). Hence, the importance of the research into the process of attention allocation during source-code comprehension arises.

The common way to identify programmers’ attention is to link it with visual attention that can be measured by gaze-fixation patterns (Bednarik, 2007). Human eyes are permanently moving. Visual in-
formation is mostly processed during fixation phase, when eye-movements are very slow (from 0 to 40 grad./sec.) (Yarbus, 1965). During fixations, vision samples the maximum clarity of information from the foveal area, which makes up 2 degrees of the visual field (Bridgeman, Van der Heijden, & Velichkovsky, 1994). Then fixations are terminated by the ballistic eye-movements that are called saccades (Yarbus, 1965). After a saccade, a new image stabilizes on the retina for the next fixation. The eye-tracking studies of source-code comprehension rest on an assumption that the foveal area plays a pivotal role in localizing attention, thus linking the direction of gaze and the focus of mind (Reichle, 2006; Bednarik, 2007).

Evidence from the domains of reading and visual studies indeed suggests that the foveal area is among the most important factors of visual attention, but not the only one (Rayner, White, Kambe, Miller, & Liversedge, 2003; Engbert & Kliegl, 2003). Visual information is obtained from an operational area that exceeds the foveal area (Gippenreyter, 1978). For instance, the perceptual span in the reading of natural-language left-to-right texts achieves up to 14–15 letters to the right of the fixation point, which extends beyond the 2 degrees area of fovea (Rayner et al., 2003; Angele et al., 2015). The operational area is utilized to plan the next fixation during a process called saccadic planning. Therefore, human attention shifts from the foveal object during fixation (Rayner, Castelhano, & Yang, 2009).

The area of source-code that can be captured by a single fixation correlates with the increase of the useful operational area. Neurophysiological studies show that visual attention can be focused on an area that is different from the orientation of gaze position (Engbert & Kliegl, 2003). There is also an evidence that attention switching could be indicated by micro-saccades (Engbert & Kliegl, 2003; Engbert, 2006; Martinez-Conde, Macknik, Troncoso, & Hubel, 2009). The micro-saccades are small involuntary saccades that take place during fixation (Engbert & Kliegl, 2003). There are still debates how this knowledge can be implemented in such a complex process as source-code comprehension (P. A. Orlov, 2015). In any case, the question about the role of extrafoveal information processing during source-code comprehension remains open.

If we limit the size of the useful perceptual area to fovea only and quantify how the programmer’s behaviour changes compared to unrestricted vision, we may be able to understand the role of the parafoveal information. The studies of visual attention in a variety of other professional domains such as car driving, sport, and chess playing, show that experts have a wider perception span; see meta-analysis by Gegenfurtner and colleges (Gegenfurtner, Lehtinen, & Säljö, 2011) for more details. In the domain of computer programming, it was previously shown that partial restriction of the extrafoveal area influences programmers’ performance (Bednarik & Tukiainen, 2005).

The emotional response of the expert programmers to working in the restricted-view mode was negative, while the novices perceived it in a neutral way (Bednarik & Tukiainen, 2007). Authors conducted informal interviews with subjects about the restricted viewing conditions after a debugging session. They provide only the summary of programmers feedbacks and one extreme situation where expert programmer declined to participate. Authors reported how the expert explained his decision: "I do not want to work with that" (Bednarik & Tukiainen, 2007). In this present study, we analyze the working process as well, however, our work differs from the previous in a number of ways: we conduct a detailed analysis of the post-comprehension self-reports on the working process, and we employ a real gaze-contingent environment, as opposed to the mouse-driven precomputed tool employed in the previous research.

1.2. Goals and aims

The motivation of our study is to uncover the underlying processing related to the utilization of the extrafoveal area during source-code comprehension. We focus on the personal experience accounts of the participants, in terms of information interpretation and affect, as proxies to understand programmers’ behaviour.

In this work we evaluate how programmers experience the restriction of the extrafoveal area, how they feel in the restricted conditions, and what topics they provide feedback about. If experts’ and novices’
experiences under the restricted visual conditions differ, we hypothesize, they utilize the extrafoveal area differently.

In sum, the goal of the study is to report and analyze how experts and novices reflect upon their working process when the extrafoveal area is restricted. To that end, we defined the following research questions:

- **RQ1.** Experts and novices reported on their source-code comprehension experiences in the restricted-view mode. What topics can be singled out in the experts’ and the novices’ reports?
- **RQ2.** What are the differences between the experts’ and the novices’ reports on working in the restricted-view mode?

2. **Method**

2.1. **Design and Procedure**

We interviewed the programmers about the nature of the experiment and their feelings about working in the given experimental conditions (Hoc et al., 1990). We employed methods from the qualitative research to obtain insights into the behaviour of expert and novice programmers (Biggerstaff, 2012). Textual analysis was based on self-reports collected from the interviews that followed the experiment. Self-report measurements are the most common methods used to ascertain subjects’ affective state (Brave & Nass, 2003; Robinson & Clore, 2002; Salah, Hung, Aran, & Gunes, 2013). Discourse analysis was used to understand how the participants organise their thinking about their experience of source-code comprehension in the restricted-view mode (Starks & Trinidad, 2007).

2.2. **Materials**

We prepared five short programs in Java. Each program was compilable and runnable. The result of each program was a value shown in the output console. The first program consisted of two class definitions and definitions of two methods. A method was calling from a class object and the result of the formula contained was printed to the output. A similar logic, but with a for-loop was used in the second and third programs: objects were created in a loop and were collected into an array. Then in the forth and fifth programs we used a loop nested within a loop. An example listing is provided in Figure 1.

The graph-theoretic (McCabe) complexity was used to prepare programs with different complexity of understanding. The complexity of the programs corresponded to different McCabe complexity levels: 1, 1.33, 1.5, 1.66, and 2 (McCabe, 1976). The calculation was done using the Metric framework plugin for Eclipse (http://metrics.sourceforge.net/). The linguistic aspects, like names of variables, were also individual for each source code. The stimuli were presented randomly in the restricted-view mode.

```java
class MyPointSM{
    public int cx, cy; 
     MyPointSM(int x, int y){ 
        cx = x; 
        cy = y; 
    }
    public int getDistance(int x, int y){ 
        int dist = 2 * (cx + y) + cx * x; 
        return dist; 
    }
}
public class Test{
    public static void main(String[] args){
        MyPointSM m0) = new MyPointSM(4, 3); 
        System.out.println(m0.getDistance(2, 2)); 
    }
}
```

*Figure 1 – An example of Java program used in the experiment.*

The subjects, as per instructions, were to calculate the result of the program. The instructions were:

‘Please, read the program and answer: What will be displayed in the console when the program is
finished? You can skip the current source code (press Skip button), but, please, use this option only as a last resort’. The original instruction was in the Russian language: here we present the translation. The subjects were to remain silent when solving the task. When the answers were ready, the subjects pressed the ‘Answer’ button to supply the answer in a new screen. At the beginning, each subject underwent training with five tasks to become familiar with the gaze-contingent environment and the experimental conditions. There were no time limits for either the warm-up trials or the test itself.

2.3. Participants and Apparatus

The experiment involved 13 male participants: 6 experts (aged 19-25) and 7 novices (aged 18-21). All subjects had normal or corrected-to-normal vision, according to their own report. All six experts had at least two years of professional experience as programmers in a software development company as a Senior Developer. All were graduates from different universities with a technical background. They were familiar with at least three programming languages. The main programming languages for them were object oriented languages such as C++, Java or PHP. Subjects’ company projects corresponded with game development for Android platforms and back-end side web development.

The novices were students of the local polytechnic university. The novices did not learn Java at a professional level. They passed a university-level exam on Processing programming knowledge and obtained an average grade of 3.5 (M: 3.5, SD: 0.46, the maximum of the scale is 5). Participation in the experiment was voluntary. Their participation was not compensated and was based only on the subjects enthusiasm.

ScreenMasker, a gaze-contingent tool, was used to allow real-time restriction of the extrafoveal area (P. A. Orlov & Bednarik, 2015). We built a patterned texture picture to mask the extra-foveal area as semi-transparent (see Figure 2 A). The transparent window had a round shape and was 200px in size. The window’s size corresponds to the 5 deg. viewing angle. Figure 2 B shows the transparency of the stencil: the area 0-2 deg. is fully transparent; the next 2-5 deg. are transparent by gradient; the area exceeding 5 deg. is fully restricted (Calvo & Lang, 2005). The area that was in clear view changed dynamically and it was contingent on the direction of gaze in real time. Thereby, we implemented the window-moving paradigm – that is, the subject could clearly see only through the stencil placed on the screen in the point of gaze-fixation position, like through the window (McConkie & Rayner, 1975).

The size of the window was determined from the physiological limit of the 2 degrees angle of the foveal vision. The viewing window size was smaller than the window presenting the code on a screen. To solve the task subjects should comprehend the whole program. Programmers had to point attention onto different source code elements and understand the coordination of methods calling and variable definitions.

After the tasks, we asked the subjects to describe their feelings and experience during the experiment in a free form, verbally. All self-reports were collected by a researcher in writing.

Eye movements were registered with an SMI RED250 eye-tracker. The stimuli were presented used a desktop PC with Intel(R) Core(TM) 2 Duo CPU E8400 @ 3.00G Hz, 3.25GB memory, NVIDIA GeForce 8800GT, running the Microsoft Windows 7 operating system. The graphics system contained High-performance monitor BENQ XL2411 with 24” screen size, 53cm x 30cm,1920 x 1080px with 144Hz refresh rate. Direct latency measurement of ScreenMasker environment was done with a Go-Pro HERO 3+ Black Edition camera (240fps with 848x480px screen size in WVGA mode), and an established measurement procedure (P. Orlov & Bednarik, 2014). The mean latency of the system was approx. 25-28 ms, well under the recommended limit of 80ms (Loschky & McConkie, 2005).

2.4. Analysis of Self-reports

Analysis of program summaries is often used for the insights into subject’s mental models of computer programs (Good, 1999; Good & Brna, 2003; Hughes, Buckley, Exton, & O’Carroll, 2005). One open question for this type of approach is related to the issue of replicability (Byckling, Kuittinen, Nevalainen, & Sajaniemi, 2004). To solve this problem Good suggested an analysis scheme to probe the comprehen-
The Good’s scheme is based on two types of classifications of summaries: information types classifier and object description categories. Nevertheless, both classifications focus on the utterances that subject used and they miss how the utterances were used, and, the previous work does not analyze the emotional state of subjects. That is why we generate topics of discussion from the content of the self-reports and we did not use a-priori categories of analysis.

After the experiment sessions, the third author sampled the programmers’ self-reports by the topics that they reported about. We analysed the self-reports to break them down by categories of topics: what the subjects were talking about, what was interesting for them, and what seemed important to them. For the analysis, we took the ratio of utterances in each category to the total number of utterances by subject.

The utterances were placed into Analysis of the working process category if a subject reflected on his strategy of task solving and patterns he used, for example: ‘I tried to summarize constant values in advance to use one instead of several when calculating the formula’.

The category Personal evaluation of the program includes utterances used by a subject to express an opinion about the program, for instance: ‘Who produced this inefficient source code here?’

Evaluation of the experimental conditions category: here the subjects reported on the gaze-contingent mode of working. For example, one subject said: ‘It was cool to control the spot by gaze’.

Viewing conditions category: here the subjects reported on their feelings about the view being restricted, the system’s lag, or the psychological pressure created by the environment. For example, one expert reported: ‘I had psychological pressure due to the restricted-view mode. It was distracting me’.

For the Emotional state category we collected all the utterances corresponding to the latent emotional condition of a programmer. For example, one expert reported: ‘I was tired at the end of the experiment’.

The Life-hacks category contained the utterances that described any additional methods of solving the task. Life-hacks could include, for example, repetition of values in a low voice in order not to forget them; using fingers for calculations and for ‘additional memory’.

We determined six topics and furthermore, we singled out the category of emotionally charged lexicon – e.g., abusive language. As a result, we obtained seven categories from the content of the self-reports, which are shown in Table 1.
Table 1 – Topics of utterance with explanations

<table>
<thead>
<tr>
<th>Topic of utterance</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of the working process</td>
<td>Subject’s reflection on his strategy of task solving and patterns he used</td>
</tr>
<tr>
<td>Viewing conditions</td>
<td>Subject’s feelings about the view being restricted, the system’s lag, or the psychological pressure created by the environment</td>
</tr>
<tr>
<td>Personal evaluation of the program</td>
<td>Subject’s expressions and opinions about the program</td>
</tr>
<tr>
<td>Emotional state</td>
<td>All the utterances corresponding to the latent emotional condition of a programmer</td>
</tr>
<tr>
<td>Evaluation of the experimental conditions</td>
<td>Gaze-contingent mode of working</td>
</tr>
<tr>
<td>Life-hacks</td>
<td>Additional methods of solving the task</td>
</tr>
<tr>
<td>Emotionally-charged language</td>
<td>Abusive language used by the subjects in their reports</td>
</tr>
</tbody>
</table>

Then third and first authors independently coded self-report’s utterances into these categories from every subject. As the reliability results indicated good inter-observer agreement (Kappa = .781).

3. Results

Novice developers skipped 8% trials, while expert participants did not skip any task. As we expected, experts performed better than novices. On average, experts solved 4 from 5 tasks correctly (M: 4, SD: 0.71) when novices solved 1.33 (M: 1.33, SD: 2.23).

The mean completion time was about 150 seconds per task. We calculated the total number of words; the experts were more vocal than the novices. On average, their self-reports contained 150 words (SD: 92.04), while the novices’ ones contained only 67 words (SD: 17.15). A two sample t-test shows that the means differ (t = 2.361, df = 6.483, p = .053).

3.1. Analysis of the working process.

A two sample t-test shows that there is no differences between the means of utterances percentage between Novices and Experts (t = .921, p = 0.383). On average, the programmers spent 50% of the utterances in their feedback to discuss the working process (Experts: M: 56.37, SD: 17.03. Novices: M: 45.18, SD: 25.25).

3.2. Personal evaluation of the program

There were no differences in means of utterance percentage (t = .298, p = 0.771). On average, the programmers spent 9% of their utterances to give their personal evaluation of the program (Experts: M: 9.997, SD: 13.744. Novices: M: 7.81, SD: 12.709).

3.3. Evaluation of the experimental conditions

The experts spent about 5% of their utterances to discuss this topic (M: 5.42, SD: 10.24). Novices reported nothing here: not a single novice commented upon the experimental conditions in his feedback.

3.4. Viewing conditions

A two sample t-test shows that there is no differences in means of utterances percentage (t = -1.755, p = .123). 12% of utterances used by the experts (SD = 8.21) and 25% of those used by the novices (SD = 17.07) can be attributed to this category. We did not separate utterances on positive or negative here.

3.5. Emotional state

A two sample t-test shows (t = -0.993, p = .358) that there is no differences in means of utterances percentage (Experts: M: 4.982, SD: 9.121. Novices: M: 15.205, SD: 23.759).

3.6. Emotionally-charged language

We found that on average 1.5% of the utterances used by the experts fall into this category. Typically, these are charged with negative emotions directed at the gaze restricted environment. The novices did not use abusive language at all.
3.7. Life-hacks
We found that both groups of subjects talk about their life-hacks in a similar way. Two Sample t-test shows \( t = .305, p = .766 \) that there is no differences in means of utterances percentage (Experts: \( M: 4.787, SD: 6.198 \). Novices: \( M: 3.824, SD: 5.168 \)).

Figure 3 shows a summary of the utterances’ percentage used by reported topics. Subjects from both groups reported more about their working process. We can see that professional developers reported less about they emotional state directly and about viewing conditions.

![Figure 3 – A summary of utterances percentage for topics by experts and novices.](image)

4. Discussion
The goal of the current study was to discover how professionals and novices report on working in the restricted-view mode. The experimental conditions did not allow programmers to process restricted extrafoveal objects even when they paid attention to them. Even though restricting parafoveal information is not practical for everyday programming, it is one of the methods to gain important understanding of the role of extrafoveal processing. The outcome of these investigations is to obtain implications on the development of future IDEs. For example, if we learned that programmers employed the extrafoveal area intensively during a certain stage of program development, this areas should be distinctively presented in the user interface of development environment to respond with its heightened role. Design of programming languages can be informed by the way programmers interact with the various structures of the source code. An additional motivation is educational: new knowledge about the expert programmers’ attention can be used for developing novel teaching methods for students. The attentional process is latent from the direct measurement systems. More over, programmers can direct their visual attention at the extrafoveal objects that are at different locations that the direction of overt visual attention focus.

The analysis of self-reports allowed to single out six categories of topics and one emotional charged state. The professional programmers were found to produce more voluble reports compared to the novices. This was expected and accords with the professionals’ familiarity with professional conversations about source code at their workplaces (RQ1).

One of the interesting findings is that both groups of programmers spent about 50% of utterances in their reports to comment upon the working process – in particular, to describe the steps of task solving.
We expected a different outcome, with the strategies description yielding a larger proportion of the professionals’ words, which would be a typical scenario (Koenemann & Robertson, 1991; Aschwanden & Crosby, 2006; Bednarik, 2012). However, the professionals decided to describe the experimental conditions instead. They reported on how they dealt with source code in a different way. Conversely, the novices reported nothing on the experimental conditions (RQ2).

The programs was evaluated by both groups using a similar number of utterances. The professionals discussed possible optimisation, while the novices were speaking about syntax recollection. For example, professionals suggest some optimisations like keeping all functionality in a one place and not dividing it into several methods or loops. We can defined it as a higher level of understanding. Novices focus more at the low level of understanding: they talk about general syntax and variables naming.

While discussing the visual aspects, both groups reported the restricted-view mode as disturbing for them. Bednarik and Tukiainen (2007) showed that novices were not disturbed much by the restriction of view. This differs from the findings presented here. The inconsistency may be due to the different experimental conditions: the previous study used a mouse-controlled window-moving tool that should be more invasive, while the present work applied a real-time gaze-contingent tool.

The description of emotional state and life-hacks took the same share of utterances in both groups. But the professionals used more emotionally loaded utterances, including: ‘It was awesome; It was tough; It distracted me quite a deal; I am psychologically crushed’. No one expert expressed affinity with working under restricted-viewing condition. Moreover the negative emotionally-charged language was found only in some reports by the experts (RQ2). These results are in agreement with the previous findings that showed the professionals to universally dislike the restriction of viewing conditions (Bednarik & Tukiainen, 2007).

We propose that the increased emotions indicate that expert programmers in fact heavily and routinely rely on the use of parafoveal information. Visual information is obtained from the extrafoveal area especially when human attention shifts from the foveal object during a fixation (Rayner et al., 2009). Humans emotionally react when they expects to attend some object and do not manage to access it easily (Calvo & Lang, 2005). Because experts are generally better aware of the relevance of the information attended, their expectations of the attentional objects are higher then for novices. They are also better at prediction of the next object of attention before switching the gaze position on it. When that object is masked and the information about it not easily accessible, the increased emotional feedback is both likely because of the mismatch with expectations and also an evidence of the importance of extrafoveal information.

The results have implications in educational purposes and on the development of future IDEs, for example if programmers use the extrafoveal area intensively this area should be better presented and more easily accessible in the user interfaces of the development environments.

As a limitation of the experimental setup, we are aware that our results may be due to a different explanation: novices felt more restricted to express themselves because they were students and were affiliated with the university. We minimized such kind of risk in our experiment conditions: in our experiment subjects were free to express their self as the like without any restriction; subjects were instructed that their results of task solving do not influence their study process or diploma defence in future.

Green (1977) showed that upstream calculation is more beneficial for the experimental conditions then downstream evaluation of the program that was used here. Upstream evaluation is ecologically valid, being a common task in program development and debugging (T. R. G. Green, 1977; T. R. Green, Petre, & Bellamy, 1991). A further study with more focus on experiment design is therefore suggested.

5. Conclusion
In our study we check the hypothesis that if professionals use extrafoveal area of vision they also self-reports about programming or the comprehension in different way. We show that experts not only talk more, but that the quality of their self reports was different.
This paper argues that professionals express themselves in a more negative emotional and nervous way. We conclude that it is the unavailable extrafoveal objects of the source code that evoked negative emotional feedback from the professionals. This study found that generally both groups report on the working process, life-hacks, and source code features in the same proportion. This study raised important questions about the nature of programmers’ reflection about the restricted-view mode. If the debate is to be moved forward, a better understanding of the role of extrafoveal information processing needs to be developed by means of the quantitative methods of eye-movement analysis.

6. References


End user programming with personally meaningful objects

Andrew Cyrus Smith
CSIR Meraka Institute and University of South Africa
acsmith@csir.co.za

Helene Gelderblom
University of Pretoria
helene.gelderblom@up.ac.za

Abstract
This project investigated what a tangible programming environment could look like in which the program is an arrangement of personally meaningful objects. We identified Gestalt principles and Semiotic theory to be the theoretic foundations of our project. The Gestalt principles of good continuation and grouping by proximity are particularly relevant to our research. Following the Design Science Research methodology, four iterations each focussed on a different design aspect based on the outcome of the previous iterations. The fifth and final iteration combined learning from the previous designs and introduced the Gestalt principle of grouping by proximity to the programming environment. We concluded the project by deriving a model that reflects the programming environment constructs and the relationships between these.

1. Introduction
Current dominant text and graphic based programming environments such as Java (Eckel, 2006) and Scratch (Resnick et al., 2009) apply the principle of “form follows function”. Krippendorff (1989) proposed an alternative approach that he called “form follows meaning” and our research considers what a programming environment could look like in which this is true.

Programming languages are designed by language architects and implemented by software developers. The architect determines what algorithms to include in the language and the developer decides how the algorithms are represented. The user then constructs a program using these representations. In almost all cases, the architect, developer, and user are three distinct persons. The result is that the user is burdened with making sense of the representations chosen by the developer. Our research contemplated what a programming environment could look like in which the user chooses how to represent the language architect’s algorithms. In searching for an answer, we encountered Semiotic theory and Gestalt principles and found that these helped solve our problem.

We developed five instantiations of a tangible programming environment with each addressing either a problem identified in the preceding iteration or a new concept that would get us closer to a solution. Our research concluded with a model of a tangible programming environment that explicitly incorporates Gestalt principles and Semiotic theory.

The rest of the paper is structured as follows: Section 2 provides the theoretical background that underpins this research with these being Gestalt principles and Semiotic theory. The methodology we followed is described in Section 3 and Section 4 describes related work. In Section 5, we describe the five iterations and the model we developed. Examples demonstrating the usefulness of the model is also given in this section. Section 6 concludes.

2. Theoretical foundations

2.1. Gestalt principles
As light passes through the eye, it forms an image on the retina at the back of the eye (Young, Freedman, & Ford, 2007). We see the image, but what we perceive is something else (Shepard & Levitin, 2002). Instead of perceiving individual and distinct objects in the world, we perceive objects that are logically connected to each other (Kimchi, Behrmann, & Olson, 2003). The Gestalt school of thought puts it that our perception of the world is influenced by the way we group and segregate the stimuli (Kimchi et al., 2003). Perceptual organisation is a neuro-cognitive process that dictates how we perceive objects in physical space and how we interpret some objects as being distinct from others and yet other objects as part of a whole (Helm, 2014). The Gestalt School of thought has identified multiple principles of perception, including grouping by proximity and the principle of good
continuation. The strength of grouping is inversely proportional to the distance between the elements (Bergman, 2009); therefore, the closer the elements are to each other the stronger our perception is that these belong to the same group. Perceptual grouping by good continuation puts it that we tend to follow a gentle curve and not deviate from this path when the curve is interrupted by sudden changes (Chandler, 2007).

2.2. Semiotic theory
Krippendorff (1989) developed a model (Figure 1) that illustrates the relationship that exists between an artefact, the designer, and the user. This model separates the designer who creates the artefact from the user who makes sense of the artefact when observed in context. Artefacts used in tangible programming environments are created this way. For the user to find meaning in the artefact requires an initial mental exercise that must be repeated after some time has passed and the meaning associated with the form has faded from memory.

Our approach differs; instead of separating the designer from the user, we propose that the two roles be incorporated into a single person called the designer-cum-user. Using this approach, the user creates a personally meaningful artefact that represents the target product. Although not confirmed, we anticipate that this approach reduces the initial cognitive load on the user when he uses the artefact as a program element. Figure 2 is Krippendorff’s model adapted to reflect our approach. This model does not include the reason the artefact is created. In our model (Figure 4), we make explicit the reason the artefact is created and indicate it as the instantiation of the system architect’s algorithm (labelled B9). Algorithms include concepts such as turning off a light or to draw a line on the computer display.

![Figure 1 - Krippendorff’s model of the user and the designer’s view of an artefact.](image1)

![Figure 2 - Krippendorff’s adapted model reflects the case where an individual is both the designer and the user.](image2)

2.3. Combining Semiotic theory and Gestalt principles in a programming environment
Our T-Logo (Andrew Cyrus Smith, 2014) programming environment explicitly incorporates the Gestalt principle of grouping by proximity to associate objects with each other and result in a program element. The arrangement of objects within a group is inconsequential. This is similar to how two English language sentences convey the same meaning: Consider the meaning of the sentences “the red car” and “the car is red”. Both contain a noun and an adjective that describes the noun. Our approach is that when the words “car” and “red” are in close proximity to each other, then they are considered to be related to each other. Figure 3 illustrates this concept. Written in textual form and using an object oriented language (Java is an example), this can be expressed as `car.colour = red` where “car” is a programmatic object.

In terms of the Saussurian (Saussure, 2011) model, the two objects on the left in Figure 3 are the signifiers and the item on the right is the signified. In terms of Peirce’s (Peirce, 1935) semiotic model, the two objects on the left are representamen and the imaginary object on the right is the interpretant. It is the process of semiosis that binds the representamen to the interpretant and it is the
representative character of the toy car and the presentative character of the red cloth that, when combined, result in the red car on the right. In terms of Peirce’s model, the cloth is a qualisign and the car is an icon.

![Perceptual grouping by proximity](image)

Figure 3 - An adjective describes the noun.

Semiotics research considers the meaning that objects hold for individuals and recognises the fact that the meaning one person attaches to an object may be different to the meaning another person attached to the same object. Our model of a tangible programming environment also makes provision for this difference in perception by using two representations of the same concept: One representation is in the form of a user-chosen tangible object and the system software developer chooses the second representation (being an identification number in the form of an optical marker). The two representations are reconciled when the user attaches the marker to the selected object.

3. Methodology
We embarked on our research without knowing in advance what theory would support our work. Our departure point was the domain of computer programming and as we discovered later, the general research domain of psychology would ultimately provide the theoretical underpinnings for our work. These are the theory of Semiotics and the principles of Gestalt. We therefore had a starting point but did not know what was missing in our body of knowledge.

Subsequently, we identified the Design Science Research Methodology (Hevner, March, Park, & Ram, 2004) as suitable to guide our research. This methodology recognises that at times a researcher has to rely on experience, intuition, and trail-and-error (Hevner et al., 2004) (and so did we). Vaishnavi and Kuechler (2015) put it that no single knowledge base is complete and this resonates well with our experience in this project when we had to access knowledge beyond the research domain of Computer Science by accessing Psychology domain knowledge. We applied Vaishnavi and Kuechler’s (2008) process model since it makes explicit that the knowledge base is incomplete when research commences.

4. Related work
Although a number of tangible environments apply the Gestalt principle of proximity, they do so without making this theoretical foundation explicit. None of these programming environments support the user in using personally meaningful objects as program element representations. Our research identified a group of environments that implicitly use Gestalt principles: The principle of grouping by proximity is evident in ReacTable (Jorda, Kaltenbrunner, Geiger, & Bencina, 2005) while the Aggregate Cube in Blackwell and Hague’s (2001) Media Cubes relies on grouping by common region.

The Gestalt principle of good continuation can be seen in programming environments such as Tern (Horn & Jacob, 2007) and GameBlocks (Andrew Cyrus Smith, 2007). Perlman’s (1974, 1976) TORTIS slot machine is the first documented tangible programming environment and it enforces the Gestalt principle of good continuity. Tern and GameBlocks both infer good continuity along straight
lines; however, Gallardo et al.’s (2008) Turtan takes this a step further by including curved trajectories in the program layout.

4.1. Navigation Blocks
Camarata (2002) developed a database query system based on tangible cubes that each represent a thing, time, place, and a person in a database. Each of the six sides represents a unique place or person. By placing a cube next to a second cube, a database query is constructed that can be written as a logical AND condition. Camarata suggested, but did not implement, OR and NOT logical conditions because the author was concerned that the additional expressions would result in an interface that is less understandable. The awkward cube combination suggested to represent logical OR is easily avoided by applying the Gestalt principle of grouping by proximity: Assuming items grouped closely together implies that everything should hold true in that grouping, then the grouping constitutes a logical AND. Conversely, if objects are in separate groups, then each group is considered independently of the others and are therefore a logical OR representation.

4.2. Media Cubes
Media Cubes inspired our approach to use grouping by proximity in our solution. Blackwell and Hague’s (2001) ontological programming paradigm includes sensing of events and actuation to change something in the environment. Events in a domestic setting include the ringing of the doorbell and the sounding of an alarm clock. Actuation includes changing a television channel or initiating a video recording. A Media Cube represents an “abstraction” and includes the change of state of a device. Media Cubes are combined to form small programs and rely on close proximity to other cubes to form a program. Although not implemented, Blackwell (2001) suggests that an object that represents time might actually look like a clock. This fits well with our overall aim and our model makes provision for the user to select his own time representamen. Association is achieved by placing the cube next to the appliance being controlled (A.F. Blackwell & Hague, 2001).

5. T-Logo

5.1. The initial iterations
The first two iterations explored placing cubes in a linear sequence. The tangible programming environments that resulted from these iterations are respectively called GameBlocks I and II. The tangible elements in GameBlocks I (Andrew C. Smith, 2006) are constructed using acrylic sheets whereas those in GameBlocks II (Andrew Cyrus Smith, 2008) make use of soft closed-cell foam. The third iteration considered hand-made programming elements carved from natural stone as an attempt to get the user closely involved in the design of the program artefacts. The resultant implementation is called RockBlocks. The fourth design explored repurposing everyday materials as program objects and resulted in a tangible programming system called Dialando (Andrew Cyrus Smith, 2010). Learning that emerged from the first iteration prompted the application of Gestalt principle of good continuation in the three iterations that followed. The Gestalt principle of grouping by proximity was introduced in the fifth and final iteration.

5.2. The final iteration
The final iteration incorporates both Gestalt principles of grouping by proximity and the principle of good continuation. It also includes Semiotic theory by making the user responsible for choosing the objects that represent program elements. Program elements are tangible objects that play a specific role in the program and elements include parameters, actions, and states. An object becomes an element when the user associates the object with a role. Association is done using a numbered optical marker (also known as a fiducial) attached to the object. An element has two representations: The first representation is in the physical world and the second is in the digital domain. The user interacts with the element using the physical representation and the software interacts with the digital representation. This final instantiation is called T-Logo (Andrew Cyrus Smith, 2014).
5.3. The model
From our learning based on the five iterations, we designed a model that captures the constructs involved in tangible programming. The model explicitly includes elements of semiotics and Gestalt principles. Figure 4 illustrates our model of a tangible programming environment in which the user chooses personally meaningful objects as program elements.

The model includes constructs and shows the relationships between the constructs. Constructs include persons such as the user who wants to solve a problem (B1), a system architect who specifies which algorithms (B10) to be include in the programming environment, and a software developer who interprets the algorithms and implements them as actions, conditions, and parameters (B12).

Figure 4 - A model of the T-Logo programming environment.
Additional constructs are a tangible program (B2) that addresses the user’s problem, the computing system (B3), materials (B14), tangible objects (B13), Gestalt principles (B8), Semiotic theory (B16), meaning that objects hold to the user (B15), and the user’s cognitive associations (B11) between tangible objects and software routines. The computing system in turn consists of executable code (B4) that address the user’s problem, markers (B5), a marker sensor (B6), an interpreter (B7), software routines (B12), and digital associations between markers and software routines (B9).

A developer implements software routines based on algorithms defined by the system architect. Making a drawing on paper is an example of an algorithm. An algorithm is realised using software routines that consist of actions, conditions, and parameters. Conditions reflect the world state while actions change the world state. Condition examples include facts such as “it is windy”, and starting a drawing and ending the drawing are examples of actions. Parameters refine the conditions of interest, and detail how actions are applied to the world. Parameter examples include ink colour and wind speed. To the user, the routines hold meaning.

Based on Gestalt principles and Semiotic theory, the user either creates a tangible object using available materials or chooses an existing object to instantiate a personally meaningful representation of actions, conditions, or parameters. This establishes an association in the user’s mind between the object and an action, condition, or parameter.

The user attaches a marker to the object and instructs the computing system to associate the marker with a particular routine. Because the marker and the object are now cognitively and digitally combined, the user and the computing system associate the combination with the same routine.

The user addresses his problem by constructing a program that incorporates personally meaningful objects representing routines. Individually, objects hold meaning to the user, while an arrangement of objects carry another meaning. Whereas the user views a program as an arrangement of one or more objects, the system interprets the program as a collection of one or more markers. Using information from the sensor, the interpreter identifies the markers and how these are arranged. It segments the arrangement according to the Gestalt principles of 1) good continuation and 2) grouping by proximity. It then produces executable code according to the user-created digital associations. The outcome is executable code that addresses the user’s problem.

5.4. The model applied
In practice, a user is presented with concepts and he can craft or repurpose existing objects to signify them. For example, a hypothetical user has chosen to represent the program termination instruction using a soft toy of a sleeping puppy. In addition, three posable mannequins represent the commands FORWARD, RIGHT, and LEFT respectively. A dog modelled from clay represents GROWL (a growling sound) and a bobblehead toy giraffe is the user’s interpretation of SHAKE (a shaking motion). Finally, a handheld torch represents the IR-BEAM test condition (is the beam active?) while two toy cars do the same for BUMP (has a bump been detected?). Table 1 summarises the user’s selection of tangible objects for these commands and conditions. Another user may have chosen different representations.

Table 1 – User’s mapping between program concepts and tangible objects, and Tern sign equivalents.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Object of user’s choice</th>
<th>Tern sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Command)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORWARD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To test our approach to having the user select tangible program objects, we chose three previously published Tern programs and show how they can be implemented using our hypothetical user’s objects. The first (Figure 5) is a sequence of six program statements: FORWARD, GROWL, RIGHT, FORWARD, SHAKE, and LEFT. Since the T-Logo programming environment continually executes the user’s code, we added a program termination object (the soft toy in this example) at the end of the
T-Logo sequence in Figure 5b. The dashed line highlights the principle of good continuation. Six slanted lines show the correlation between the Tern (Figure 5a) and the T-Logo implementations.

![Figure 5 - Example of a linear program.](image)

The second example (Figure 6) demonstrates a conditional IF statement and the use of a parameter. This is a typical code segment for a toy robot and determines the robot movement when a bumper (being the parameter) is activated. Figure 6a is a program using the Tern language while Figure 6b is the same program logic implemented using T-Logo.

![Figure 6 - Example of a conditional statement in a loop.](image)

The third example illustrates a logical AND expression. In this example, the compound AND expression in Figure 7a has been rewritten as four expressions and implemented using T-Logo. The original expression is:

\[
\text{IF (IR-BEAM = TRUE) AND (BUMP = TRUE) THEN do RIGHT ELSE do FORWARD.}
\]

Figure 7b shows the Tern implementation. Equivalent logic can be composed and the result is expressed in Figure 7c using the T-Logo language:

\[
\begin{align*}
\text{IF (NOT-IR-BEAM = TRUE) AND (NOT-BUMP = TRUE) THEN do FORWARD} \\
\text{IF (IR-BEAM = TRUE) AND (NOT-BUMP = TRUE) THEN do FORWARD} \\
\text{IF (NOT-IR-BEAM = TRUE) AND (BUMP = TRUE) THEN do FORWARD}
\end{align*}
\]
6. Conclusion
We have set out to develop a tangible programming environment in which the user can choose his own objects to represent program elements but discovered that we had insufficient knowledge on how to proceed. Our search for an appropriate research methodology identified Design Science Research as appropriate. We subsequently found theory and principles in the Psychology domain to underpin our research, specifically Semiotic theory and Gestalt principles. An analysis of existing tangible programming environments revealed that some Gestalt principles were implicitly present.

We presented a model for a tangible programming environment in which the user can use personally meaningful objects and apply Gestalt principles when constructing a program. Finally, to test the effectiveness of our T-Logo programming environment we constructed three hypothetical programs that are equivalent to previously published Tern code segments.

7. References


TALES: An E-Learning Application to teach programming concepts to the Early Years Foundation Stage

Melenie Schatynski  
Department of Informatics  
University of Sussex  
ms660@sussex.ac.uk

Sharon Wood  
Department of Informatics  
University of Sussex  
s.wood@sussex.ac.uk

Abstract
As a pupil concludes Key Stage 1 they will have been taught to understand basic algorithms, typically in a recipe style, create and debug simple programs and understand why their programs behave the way they do, in addition, gaining knowledge of common technologies. Introducing these concepts to children in their pre-school education will prepare them for the new changes to the curriculum. This study describes the development of an educational system called TALES which is designed to teach programming concepts through a series of mini games. It is aimed at children at the Early Years Foundation Stage (EYFS), specifically children with a cognitive ability between three and six years of age, and therefore does not assume the user to be a fluent reader or writer. The application is evaluated through a small user evaluation study. Early results indicate that learners engage enthusiastically with the games, taking more time to follow instructions and greater care as they become increasingly motivated to succeed in advancing through progressively more challenging levels of the game.

1. Introduction
As technology evolves, the demand for developers and engineers has increased. However, in 2014 only 4% of graduates studied Computer Science at degree level (Higher Education Statistic Agency, 2014). Since then the Department of Education has introduced Computing as a compulsory subject (Department for Education, 2013) in the national curriculum to prepare children for their future in the Digital Revolution. In keeping with these recent curriculum changes, this paper discusses the development of an application designed to educate and interest younger users in a STEM-based subject.

The new curriculum focuses on programming and aims to ensure that children leave education as computational thinkers (Department for Education, 2013). The skills learnt in a computing lesson are strongly linked to other compulsory subjects such as maths, science and design and, although not every school leaver will become a software engineer, the skills acquired will benefit them in other careers and in their personal progression (Resnick, 2012). Programming and debugging code enables pupils to solve problems by logical thinking, whilst constructing their own programs enables them to express their creativity and improve their design skills (Computing at Schools, 2015a). It gives children from all backgrounds a chance to express themselves (Resnick, 2012), share their ideas and have access to technologies they may not have access to at home.

Organisations such as Computing at School (CAS) (Computing At School, 2015b) and BCS (BCS, The Chartered Institute for IT, 2015) have provided teachers with the confidence to teach computing by supplying them with a range of resources (Computing at School, 2015c). QuickStart, established by CAS and Microsoft (Microsoft, 2015), is an online resource aimed at primary school teachers and provides a collection of games and visual programming applications such as Scratch and Lightbot to assist in lesson planning (figure 1). These applications are aimed at different ages and teach children the skills required according to the Department of Education (Computing At School, 2015b).

---

1 Education institutes in England and Wales refer to pupils in the first and second year of their education as Key Stage 1, during this stage, pupils are aged between 5 and 7.

2 A visual programming languages allows a user to create a program through a graphical interface instead of the traditional characters and tokens. In this example the user is able to build their program by arranging “blocks” on the screen.
Although computing is now a compulsory subject in the national curriculum, Academies are not required to follow the same curriculum as state schools (Department for Education, 2015) and an increasing number, 55% in 2015, deviate from the national curriculum (Gee, Worth, & Sims, 2015) which therefore affects the number of children being taught computing in the classroom. Similar to other educational programming languages aimed at the early years (such as Scratch Jr (Lifelong Kindergarten Group, 2016)), TALES is an iPad application that teaches its users about programming concepts. Scratch Jr allows free play, whereas TALES uses a series of mini-games to introduce concepts gradually, using colourful, friendly and entertaining visual representation and graphics and aims, to grasp a child’s interest regardless of the subjects they are taught at school.

2. Background

2.1. Education and Development

Educators in the early learning stages use a common framework which encourages children to learn through playing, exploring, active learning and creative thinking (Foundation Years Organisation, 2015). During the EYFS, a child will develop quickly, therefore between 30 and 60 months a child will be assessed against two overlapping development stages in accordance with the UK Government and Foundation Years Organisation framework (Department for Education, 2015).

Children between these ages should be able to understand and respond to simple instructions to place objects (including those that involve prepositions) and will begin to understand questions in the form of interrogatives (‘why’ and ‘how’). TALES communicates with the child in the same way by asking questions regarding the characters on the screen, e.g. ‘How many steps will Jack need to take to be next to the table’. In addition, children between 30-50 months will begin to expand their vocabulary by using conjunctions to form more complex sentences to express their ideas. This enables the application to explain the logic surrounding an action, e.g. ‘Jack moves 10 steps because…’. It uses very simple language that enables the user to understand the connection between their choices and what is appearing on the screen.

To explain these ideas, the application provides the user with simple sentences in the form of text on the screen. According to the Foundation Years Organisation children older than 40 months will begin to sound out words and read simple sentences, whereas children with a cognitive age of 30 and 40 months will still be developing these skills, therefore, the application relies on also providing the child with audio feedback and visual explanations to ensure the entire target audience understands.

The decision to create an education application for iPad was due to the rise in assistive technology in classrooms and the many benefits that have been discovered. Recent case studies have shown an
improvement in a child’s development after introducing iPads as an educational tool in various K-12 schools across the United States. Educational institutions have identified a correlation between iPad assisted learning and their pupils’ overall engagement and motivation to learn (Apple Education, 2012). Similarly, in the UK, tablets are now being used in 68% of primary schools providing assistance in everyday lessons (Coughlan, 2014). An additional group to see the benefits of using tablets in the classroom is children with special educational needs. They are being used as an assistive technology to help pupils of special education institutions overcome the additional struggles they are faced with in communication and learning (McKnight, 2013). The addition of tablets has been proven to enable disabled users to become more independent and focused when learning (Queensland Government, 2012).

2.2. Learning with Multiple Representations
The application uses multiple representations to explain challenging programming concepts through animations, text, and audio. The use of multiple representations, in this case, will be to construct a deeper understanding of an idea. Multiple representations have many advantages when presented to the user in a way which is most appropriate to them. In order to grasp the idea successfully, the user must have prior knowledge of at least one of the representations especially when used to assist the interpretation of another (Van der Meij & de Jong, 2003). By keeping the language simple through audio and text representations, they can be used to present complex ideas to younger children and complement what the other is attempting to portray (Ainsworth, 2008).

A popular representation to present abstract complex ideas is through animations. It has been argued (Tversky et al., 2002), that the use of animation adds little benefit to a student’s learning. Based on their meta-analysis of existing studies, students performed better because the animations presented provided more information than their static equivalents, therefore, deemed incomparable. However, each study discussed concluded the group of students using the animation to aid their learning exceeded the performance of those using the static graphics. Therefore, it could be argued animations allow for a more detailed description and enable interactive learning. Animations within the application are used to illustrate to the user how the set of instructions interact with each other. In TALES, animations appear slow and clear to ensure the user can interpret each individual instruction in the program sequence.

3. The TALES Application
The objective of this study was to design and implement an application to introduce programming to children in pre-school education as shown in figure 2.

![Figure 2 - TALES screenshot](image)

K-12 education refers to a child’s education from kindergarten to 12th grade, this is typically children between 5 and 18 years old.
The iPad application (shown in figure 2) prompts a user to construct a program using a visual programming language (VPL) to complete levels. The user receives rewards and unlocks subsequent levels on successfully creating a program sequence that completes a task. The VPL is represented using colourful and engaging instruction blocks. Each instruction block belongs to one of four types and these types are used to validate a user’s input once a sequence has been executed. The four types are as shown in table 1. The language has been designed so that a Play object must appear at the beginning of a sequence and Actions appear after Object blocks as an Action belongs to an Object.

<table>
<thead>
<tr>
<th>Instruction Block Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play Block</td>
<td>The Play will appear at the beginning of every sequence to represent the start. It should only appear once in a sequence. The play block appears as a green square block with a white arrow head.</td>
</tr>
<tr>
<td>Object Block</td>
<td>An Object block represents a character. A user can animate an Object by placing Action blocks subsequently to an Object block. Objects are labelled using letters of the alphabet.</td>
</tr>
<tr>
<td>Action Block</td>
<td>An Action block determines how the Object (character) will be animated. Figure 5 shows yellow Action blocks which allow the character to move, up, down and right.</td>
</tr>
<tr>
<td>Control Block</td>
<td>Control Blocks control how each section of the sequence is executed. Two examples of Control blocks are conditionals and loops.</td>
</tr>
</tbody>
</table>

Table 1 - Instruction Block Types

3.1 Level Structure
The app takes the user through a series of progressively more challenging levels of the game. Each level presents the user with new knowledge to absorb (as shown in Table 2), following which the user is asked to complete a short task that applies this knowledge. The user is able to execute their program (figure 3) which plays a short animation that corresponds to their program sequence, providing the user with immediate visual feedback. Small, simple pieces of information (figure 4) are presented to the user to ensure each concept is understood. In addition, concept-relevant feedback is provided when a user incorrectly runs the program to reinforce their understanding of the original concept before they re-attempt the level. The type of feedback presented to the user is dependent on whether a parse or a code error has occurred (figure 5).

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Level 1 introduces the Action Blocks. A Play and an Object block are presented to the user as an existing and un-editable part of the sequence. The user has the choice of four Action Blocks to append to the program sequence (Up, Down, Right and Left).</td>
</tr>
<tr>
<td>2</td>
<td>Level 2 introduces one Object Block and informs the user that Action Blocks change the position of an Object (character). The program sequence still presents the user with an un-editable Play Block, however a user can now place an Object Block anywhere in the sequence in addition to the Action Blocks.</td>
</tr>
<tr>
<td>3</td>
<td>Level 3 introduces the Play Block. The user is no longer given any pre-set blocks in their program sequence.</td>
</tr>
<tr>
<td>4</td>
<td>Level 4 introduces two Object Blocks. Users are taught that Actions are only applied to the Object Block they appear subsequently to.</td>
</tr>
</tbody>
</table>

Table 2 – Level Progression
The user receives star rewards once they have completed a level. The user will always receive at least one star and the number of stars is determined by the number of attempts taken to complete each level (fewer attempts leads to more stars). Rewarding users in relation to their number of attempts introduces a gamification element to the application therefore rendering it more engaging.
To include additional blocks in a program sequence the user can select instruction blocks (top left section in figure 3) to appear in the program flow window (top right section in figure 3). The user is able to alter this sequence by dragging items in the program flow window to the bin to remove completely or by dragging and dropping blocks to amend the program sequence order.

In addition to background research, the app design was additionally informed by a small number of children from the target age group who were invited to suggest ideas for the various app levels and, as a consequence, the design of several levels were based on those suggestions. Figure 6 illustrates one of the suggested levels (to create a cake) used within the game.

![Can you help make a cake?](image)

**Figure 6 - Instructions from Level 4**

### 3.2 Main Colour Scheme

The choice of colour scheme was a crucial decision in the design process as colours can be associated with a person’s behaviour (Doherty, 2010). As the game is designed for young children, it needed to be engaging and fun; therefore, a warm colour was chosen as these are generally associated with stimulation and happiness. In addition, research (Doherty, 2010) shows that users’ cognitive performance increases when exposed to warmer colours.

The colour yellow was chosen as the application’s primary colour over other warm colours such as red and orange, as red is more commonly associated with danger (Saif, 2011) and recent studies (Saif, 2011) showed that yellow has a more positive connotation than orange. Characters are animated when a user executes their program and have been designed to be colourful and ‘cute’ to allow for a positive learning environment.

![Main Character in TALES](image)

**Figure 7 - Main Character in TALES**

### 4. User Evaluation Study

#### 4.1. Method

A group of children between the ages of four and six years old were invited to participate in an observational study in which the children used the app, followed by a short group discussion which allowed each participant to express their opinion regarding the application. Informed consent was obtained from each participant’s guardian before the study began. In addition, the purpose and structure of the study along with their right to withdraw was explained to each child and their verbal assent acquired. The focus group was conducted in a home environment to ensure naturalistic results. Participants were recruited based on their age and were of mixed ability. To ensure the data collected is kept confidential each user was assigned a unique number which they were able to use to log into the system. Only the participant’s progress data and their unique number is saved, therefore leaving each user anonymous. The participants will be referred to as shown in table 3.
<table>
<thead>
<tr>
<th>Participant</th>
<th>Age</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participant A</td>
<td>6 years old</td>
<td>Male</td>
</tr>
<tr>
<td>Participant B</td>
<td>6 years old</td>
<td>Female</td>
</tr>
<tr>
<td>Participant C</td>
<td>5 years old</td>
<td>Female</td>
</tr>
<tr>
<td>Participant D</td>
<td>4 years old</td>
<td>Male</td>
</tr>
</tbody>
</table>

Table 3 - Participants.

Quantitative data about each user’s progress was captured automatically by the app and the data was sent to an external source. Each child was provided with an iPad with the application pre-installed and assigned a participant number to use to log in for the purposes of the study. In addition to the automatically captured quantitative data, hand-written observational notes about each participant’s interaction with the application were taken throughout the session of playing using the app.

4.2. Data Collection

For the purposes of the user study, TALES records a user’s progress to an external database including correct and incorrect program sequences, time to completion and completed levels. If an internet connection is unavailable, the data is stored locally until an internet connection is available. The incorrect program sequences focused on two types of errors, parse and code and were recorded/coded retrospectively by the observer following the procedure shown in figure 8. A parse error occurs when a sequence doesn’t conform to the rules of the VPL, whereas a code error occurs when the sequence of instruction blocks is valid but the sequence does not complete the level (e.g. the character will not reach the food as shown in figure 3).

![Figure 8 - Validation Process](image)
4.3. Quantitative Results

Each time a user attempted or completed a level, the program flow sequence was stored in an external database complete with any relevant error types.

The graph in figure 9 shows the percentage of each error type per level. As parse errors are not possible in level one they have been excluded from the average.

The graphs in figures 10 and 11 display each user’s time to complete and the number of attempts per level, respectively. A comparatively dramatic increase can be seen in the figures relating to level 2 with an equal percentage of code errors to parse errors. The introductory level was completed in the least amount of time with the least amount of attempts by the majority of participants.
The program flow for each parse error has been investigated and the main issues users faced have been detailed in the table below.

<table>
<thead>
<tr>
<th>Level the error occurred</th>
<th>Reason for the parse error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 2</td>
<td>Participant placed two Object blocks in the program flow without continuing the sequence with Action blocks.</td>
</tr>
<tr>
<td>Level 2</td>
<td>Participant placed Action blocks before the Object block.</td>
</tr>
<tr>
<td>Level 2</td>
<td>Participant appended an additional Object block to the end of a correct sequence.</td>
</tr>
<tr>
<td>Level 3</td>
<td>Participant didn’t include a Play block at the beginning of the sequence.</td>
</tr>
<tr>
<td>Level 3</td>
<td>Participant didn’t include a Play block or an Object block at the beginning of the sequence.</td>
</tr>
<tr>
<td>Level 4</td>
<td>Participant included both Object blocks correctly, although didn’t include the Play block at the beginning of the sequence.</td>
</tr>
<tr>
<td>Level 4</td>
<td>Participant attempted to append an Action block to two Object blocks, each Object block requires its own Actions.</td>
</tr>
</tbody>
</table>

Table 4 - Program Flow Errors

4.4. Observation Results
When the children were presented with the application they seemed to be excited to get started. Therefore, a couple of the children started the levels before logging into the system. This did not pose a problem as the system logs a user’s progress locally and uploads it once they have logged in. All participants seemed to understand the first level without any problems, although there was a bit of trial and error to find out how far the giraffe would walk based on a single Action block.

Participants seemed to enjoy the star rewards and felt a sense of achievement when they were received and were happy when they progressed to a new level. Participant A regularly expressed their joy when they received three stars for completing a level.

Participant B would remove blocks rather than rearrange if there was an error in their sequence, as they appeared to be unaware of the drag and rearrange feature in the program sequence.

The second level seemed more difficult for the majority of users. Users were eager to progress past the instructions to the actual game, and therefore didn’t read the instructions. In addition Participant D (the youngest participant) was confused about which button needed to be pressed in the Feedback view. Participant D also made the most parse errors and often expressed frustration with the application.

![The OBJECT has to go BEFORE ACTION blocks](image)
The majority of participants became more focused whilst completing later levels, appearing to spend more time thinking about the solution before attempting it, and would express their delight on completing another level.

All the users touched the Play block in the program at least once to try and execute the program instead of the one in the menu bar and none of the users used the undo or redo functions. The older participants in the group didn’t require any assistance, whereas the youngest participant only started to enjoy the game once the aim of each level was read to them verbally. However, the language used within the application was understood by all participants.

From the discussion it was discovered the participants felt like they learnt something new. When asked what they liked most about the application, several participants answered level 3 as they felt that was the easiest level. Participant B found the application ‘nice’ and ‘challenging’ although at times confusing, in particular the number of steps needed to get to the end. Participant D didn’t feel as though they would play the game again as they found it too difficult, whereas Participant C enjoyed playing the game and would play again. All participants expressed the view that their favourite level was Level 4 as they found it challenging and fun.

5. Discussion
This and the following sections discuss the results presented above and evaluate the application in terms of the primary objectives of the study, suggesting improvements based on those findings.

5.1. Does the application teach the user basic programming concepts?
The application presents the user with new knowledge and requires them to complete a task based on that knowledge. Whilst observing users with the application, it was noted that they made frequent mistakes regarding the program sequence which were typically code errors. The discussion that followed and the data logged by the system revealed this was due to the user’s inability to gauge the distance attached to each Action block throughout all levels. Therefore, the vast number of code errors are due to the participants’ trial and error approach to the program sequence. Taking this into account, we can start to look at the parse errors  that occurred which, as shown in figure 9, are significantly fewer. Only 33% of the errors that occurred were due to a parse error and, of those, 73% were generated by one participant. This suggests the remaining participants understood the rules of the VPL.

The results show, a decrease in the number of attempts for all participants in the final two levels (figure 10), however, the time to complete increased in the final level. This could suggest that participants were taking more time to solve the problem in later levels as they began to understand the concepts more. Although parse errors did appear in later levels, the number of parse errors steadily declined (figure 9) and the observations indicated that participants seem to learn from their earlier mistakes and rectify their mistakes quicker than for previous levels, despite the level of difficulty increasing.

The initial introduction of Object blocks in Level 2 confused several participants as more errors occurred at this level (shown in figures 10 and 11). As discussed in the observation study results section, participants were eager to progress and skipped the instructions, therefore, allowing insufficient time to absorb the new knowledge. A possible solution to this is discussed in the next section. Users did not disregard the level introduction in later levels following Level 2. In addition, after Level 2 the amount of time to complete increased (figure 10) as the number of attempts (figure 11) stayed consistently low: to ensure the information was understood and they could complete the level with fewer mistakes, users took greater care when reviewing the instructions at the start of each level.

Overall participants seemed happy with the application and seemed confident in using it. The majority of participants expressed they had learned something new and found the game enjoyable. Although the positive results are indicative, it is important to confirm them by performing a user study involving a larger sample size in order to conclusively establish that the application consistently teaches programming concepts to its users.

5 Parse errors are generated when the program flow sequence will not parse in the current order.
5.2. Suggested improvements based on findings
Several issues were raised during user testing, outlined below with suggestions for potential solutions.

As discussed, users were unsure about the distance each Object would move in relation to the Action blocks they chose, leading to frequent code errors. This problem could be resolved easily with the addition of a grid being placed on top of the existing background as a guide, enabling the user to count the number of cells the giraffe would need to move to reach the desired position.

Younger users were less inclined to read the instructions at the beginning of each level. The addition of audio instructions could aid the user in their learning experience, removing the need for additional assistance. To ensure confident users weren’t hindered by the addition of audio instructions, this would be a feature the user could activate and deactivate as and when they required. Another solution to the tendency to skip instructions could be to present the user with a short quiz, testing them on the new information presented, before allowing them to continue.

A number of the participants struggled to differentiate between buttons and images in the feedback pop up. A simple solution to this would be to redesign the images without the raised effect that misleadingly suggests they are actionable buttons.

Participants attempted to play the animation using the Play block in the Program Flow. This could be resolved by using separate icons for the Play block and the Play button, or alternatively allowing the Play block to also validate the program as many participants automatically chose the Play block before the Play button.

Some of the participants had a much higher number of attempts on level 2. This could be resolved by offering the user a hint about the next possible block. The game was designed using a 2D array to represent a grid, therefore, this could be achieved using a search algorithm to find a pathway to the End from the Object’s current position and suggest the next block to the user.

6. Future Work
In addition to the improvements suggested in the previous section, an obvious improvement would be to extend the range of levels within the game to include further programming concepts, such as conditional and loop statements (Control blocks), thus increasing the user’s knowledge of basic programming concepts. A further idea is to incorporate a Project Mode within the game. The aim of this would be to enable users to create their own additional game levels and animations using the skills they have acquired from the main game.

Further investigation through user testing would support the suggested improvements such as experimenting with different versions of the Play block and Play button. In addition, to ensure educators are comfortable using TALES as an educational tool in their classrooms, further user studies are required using primary and early year’s teaching staff.

7. Conclusion
This paper presents an app developed to assist young children to grasp programming concepts early in their educational experience. Overall, the application was a success and through user testing it was found to increase interest in STEM subjects (notably programming) amongst the young users involved.

Since this project began, educational technology aimed towards computing continues to evolve. The introduction of the Raspberry Pi Zero (Raspberry Pi, 2015) in late 2015, continues to make computer science more accessible and affordable to the masses, rendering it all the more important to equip young people with the concepts needed to exploit these technologies and to forge productive careers.

It is intended to continue the development and research to improve TALES considerably in the future. Ultimately, it is hoped to release it as a fully functioning application available through Apple’s App Store.

8. Acknowledgements
The authors would like to thank Marianna Obrist and the members of her research group for the equipment used in this study, the parents and children who took part in testing the application, whose
contributions were crucial to developing the TALES application, and Judith Good for her continued encouragement in the preparation of this paper.

9. References


129
Whither with 'with'? – new prospects for programming

Nicolas Pope
Computer Science
University of Warwick
nw pope@gmail.com

Elizabeth Hudnott
Computer Science
University of Warwick
lizzy.hudnott@gmail.com

Jonathan Foss
Computer Science
University of Warwick
jonny@dcs.warwick.ac.uk

Meurig Beynon
Computer Science
University of Warwick
wmb@dcs.warwick.ac.uk

Abstract
The role that dependency can play in developing programs is a longstanding topic of interest. The idea that spreadsheet principles are the key to new more accessible ways of programming continues to be a focus for current research. This paper further elaborates an idea whose development has been documented in several previous PPIG meetings: that programming can be best understood by construing human and automated agency in observational terms. Its main focus is on illustrating and discussing a new concept: a with-construct that has been introduced in a recently developed environment for making such construals (the 'MCE'). Preliminary work indicates that introducing this concept has a transformative impact both on the practice of making construals and its relationship to conventional programming.

1. Introduction
This paper could be regarded as a sequel to a paper presented at the 6th PPIG meeting in 1994 [6]. The title of that paper, by Beynon and Joy, was "Computer Programming for Noughts-and-Crosses: New Frontiers". Its main purpose was to introduce an approach to agent-oriented modelling in which the interfaces that mediate the interaction of each agent are expressed as networks of dependencies between observables and to show how this might be applied to developing programs. By way of illustration, that paper explored how programs for playing noughts-and-crosses – and variants of noughts-and-crosses that were characterised as 'OXO-like games' – could be flexibly developed in this way. In keeping with the spirit of PPIG, the emphasis in the paper was on the way in which the networks of dependencies, as represented by sets of definitions (aka "definitive scripts"), could be interpreted in cognitive terms. A basic cognitive function in playing noughts-and-crosses would be 'recognising the grid geometry' for instance; a more sophisticated one would be evaluating the merits of playing in a specific grid square. The development of scripts to express the agency in a program to play an OXO-like game reflected the way in which such cognitive capabilities could be construed as layered: one capability being prerequisite for another (cf. the script extracts in Figure 3 below).

Appending the words 'New Frontiers' to the core title of the paper was in part intended to be facetious – surely the idea that programming a simple game could be so challenging as to trespass on the boundaries of programming practice was absurd, and, more pretentiously, prescient – reflecting the idea that programming was not delivering to its full potential and in great need of an alternative vision. By comparison with conventional programming, agent-oriented modelling with definitive scripts clearly offered prospects for development more closely matched to human cognitive functions and with far greater openness and flexibility to change (cf. [6,5]). It was also an approach that had quite the opposite effect from conventional practices: rather than modelling the application domain by abstracting and simplifying, it engaged in the first instance and in its essence with the concrete experience of the modeller, and encouraged reflection that exposed the cognitive complexity that underlies the simplest tasks. The practical implications for giving digital support were intimidating: though each individual observable in a script might have an intuitive counterpart and hence be simple to interpret, specifying and managing the dependencies between observables and taking account of the open-ended range of perspectives on agency generated unprecedented problems. For instance, it was hard to imagine how to express the observation involved in surveying the grid to identify a winning
move, or developing a program that exploited a standard minimax strategy, on account of the number
of observables involved and the subtlety of their interrelationship, conceptual, spatial and temporal.

This paper discusses how the principles proposed in [6] are currently being developed and supported
in such a way that the prospects for more serious application to programming are much enhanced. The
four main sections which follow relate to: the conceptual framework that has now been adopted – that
of ‘making construals’ [3,2]; the with-construct that has recently been introduced into the environment
for making construals (the ‘MCE’); the significance of the with-construct from a programming
perspective; and the prospects and challenges for future development and application.

2. Making construals

The emphasis in thinking about computing, and the central place of programming within it, owes
much to Turing’s seminal account of ‘a mind following rules’ [13,16,4]. In placing the emphasis on
‘computational thinking’, there is a danger of neglecting an equally significant preoccupation for the
human mind: that of ‘making sense of a situation’. The premise here is that an excellent working
understanding of a situation is a prerequisite for being able to act by following rules. Informally, it is
this sense-making activity to which the term ‘making construals’ used throughout this paper refers.

In this paper, the term “construal” is used in a distinctive way to refer to an interactive physical
artefact that embodies our ‘working understanding’ of situation in terms of three primary interrelated
notions: agency, observation and dependency: a concept that is the central theme of many previous
publications from the Empirical Modelling project [21]. In any situation, agency relates to what we
believe to be responsible for changes we observe, observation to the specific entities to which we
believe agents respond and through which their state-changing action is mediated, and dependency
to the way in which changes to observables are perceived by agents to be concomitant. There is a
distinguished role in this sense-making activity for the personal agency, observation and dependency
that is perceived by the maker of the construal, but the understanding of the situation encompasses
putative agency, observation and dependency that is projected onto other agents of which the maker
can have no direct experience. Understanding the way in which agency, observation and dependency
interrelate establishes a template that is characteristic of a situation. In making a construal of a
situation using contemporary technologies in which computing is an essential ingredient, the
aspiration is to develop an interactive artefact that embodies counterparts of the agency, observation
and dependency that interrelate according to this same template. In a computational environment, the
situations that arise must be crafted so that agency, observation and dependency exhibit highly stable
patterns of interrelationship that can be reliably reproduced.

In [9], the psychologist and philosopher of science David Gooding introduced the notion of a
construal to account for the way in which Michael Faraday achieved the ‘working understanding’ of
electromagnetism that led him to prototype the first electric motor. This is a precedent for applying
the term ‘construal’ to hybrid constructions that have both a physical aspect and an associated cloud
of interactions and interpretations. The spreadsheet is another example of such a blend of physical and
mental ingredients that serves a sense-making purpose in which the concepts of agency, observation
and dependency are more explicitly represented. Previous work shows broad potential application for
making construals with this distinctive emphasis on agency, observation and dependency [21], but this
has been constrained by the technical limitations of the approach described in [6].

The most familiar context in which the term ‘construal’ arises is in natural language understanding.
An expression such as ‘the polar bear is chilling’ can be construed in several ways. We may imagine a
polar bear lying in the shade of an iceberg on a sunny day, or a cartoon polar bear with sun glasses
and a pint of beer lolling on a beach, or a hostile bear standing on its haunches towering above us
ready to pounce with ‘arms’ and claws outstretched. Observables, dependencies and agents are clearly
relevant to understanding these scenarios, though these are of a more complex and subtle nature than
are illustrated in Figure 3. Whether or not the polar bear is seen as ‘chilling’ is an observable. Its
status is determined by dependency on more prosaic observables: Is the sun shining? Is the bear in the
shade? Are the bear’s limbs configured in a benign way? Different construals reflect different kinds of
agency: Is it the bear or the observer who experiences the chilling? At a meta-level, other varieties of
observation, dependency and agency may be significant. A teacher might use this expression as a way
of assessing natural language understanding: the capacity of the student to observe different meanings being dependent on their level of familiarity with English.

Construal is an important ingredient in problem-solving. Creating artificial contexts for solving puzzles is a good way of exposing characteristic features of construal. In a cryptic crossword puzzle, each clue is a phrase that at face value has a natural language meaning, but – with some ingenuity – can also be construed as specifying a word to be entered into the grid. There are then standard conventions to guide the solver. The solution is always given in a non-cryptic form in the initial or final segment of the clue. Words such as ‘leading’, ‘embrace’ or ‘scrambled’ that have literal meanings may in fact refer to the disposition of letters in the solution. By way of illustration, consider the phrase ‘Force a writer to absorb Republican material’, which is the clue to a 6 letter word in a Guardian newspaper crossword [17]. A plausible solution is ‘police’: a writer (construed as Edgar Alan ‘Poe’) absorbs (construed as ‘contains’) Republican material (construed as the letters ‘lic’ extracted from the word Republican), the first word in the clue – force – being construed as a non-cryptic reference to the police force.

The nature of the activity involved in arriving at this solution illustrates several distinctive qualities of making construals. At its core, making construals is a subjective activity that relies upon associations that can be experienced by the maker on account of their general knowledge and cultural background (the fact that Poe is a writer, and that the police can be described as a force are well-known in the UK and USA). These associations can be loose and informal (how many different combinations of letters might qualify as ‘Republican material’?) and cut across semantic categories (Poe is perceived both as an author and as the string of letters ‘poe’). The plausibility of a construal depends in general on conventions for interpretation (how well has the crossword solver – or indeed the crossword setter – observed the accepted code of practice). The justification and evaluation of a construal is not amenable to abstract logic – it relies upon a process of exposition in which proposed connections have to be presented in conjunction with informal explanations and critically examined (consider what steps lead us to conclude that the answer to the above clue is ‘a force’ rather than ‘a material’). To accept the validity of a construal is to make a pragmatic judgement that takes account of all manner of observations about connections being proposed (for instance, Roe is a writer [12] – though not quite as celebrated as Poe, ‘ubl’ is a substring of “Republican”, and there is an online reference under a Guardian headline from December 2014 citing “a rise in the rouble ... as forcing people to live within their means” [19] – but rouble would hardly be deemed to be a convincing solution). In general, the status of construal as right or wrong, good or bad, appropriate or inappropriate, is likewise to be construed as context-dependent (for example, though ‘police’ is in some sense a good and appropriate proposed solution, it is actually incorrect, as reference to [18] (‘spoiler alert’) will show. That is to say, it is not consistent with the answers to other crossword clues). In practice, the way in which the maker is guided towards a construal can be very personal matter that may depend on the process of construction itself (‘police’ is an alternative solution that would not have been devised had other clues been solved first). This interplay between the construction process and the personal perspective of the maker of a construal is even more vividly highlighted by the famous NYT crossword published prior to the 1996 US election [20], where the answer to one clue could be construed to be either of the presidential candidates “Clinton” and “Bob Dole”. This evoked amazement or indignation from solvers that reflected their temperament, political affiliation and the solution they found.

In building on [6], the aspiration is develop principles and instruments for making construals that can do justice to construing in its broadest sense. Though the central focus of the paper is on a technical construct that can create a more expressive instrument, the concern is much broader than issues of programming language design.

3. Introducing the with-construct

Making construals is the central theme of the EU Erasmus+ CONSTRUIT! project [22]. A core aim of CONSTRUIT! is to create an online environment for making construals (the ‘MCE’). The latest version of the MCE [23], whose principal architect is Nicolas Pope (cf. Figure 1), incorporates several new features that promise to transform the practices that were first prototyped in [6]. One feature in particular has most significant implications for specifying families of definitions that correspond
closely to the more complex modes of observation encountered in making non-trivial construals. This is a "with-construct" that makes it possible to replicate patterns of dependency without needing to explicitly clone observables and create bloated scripts with perhaps many thousands of definitions.

The basic principle behind the with-construct is that where an observable A is defined by an (acyclic) network of dependencies in which other observables B and C occur, it is possible to introduce a variant of the observable A that makes use of the same dependency network that specifies A but incorporates new definitions for the observables B and C. Syntactically, this is expressed:

\[ \text{newA is A with B is newB, C is newC} \]

The with-construct has a wide range of potential applications that both have the effect of making scripts much more concise and making it possible to express more complex forms of observation. A key element in this is being able to generate a set of instances in the form of a list defined via

\[ \text{list_of_newA is A with B ranging over list_of_newB, C ranging over list_of_newC} \]

The nature of the with-construct and its full implications for making construals are yet to be explored. The illustrative examples represent some of the possibilities to be further discussed and developed.

Where applications are concerned, several broad categories of use have been identified to date. Each of these modes of use of 'with' is illustrated in scripts that can be found in the project repository within the MCE [23, 24]. A recent version of the MCE can be invoked at the url cited below [23]. A basic introductory tutorial for the MCE together with instructions for loading example construals can be found via a webpage prepared for use by teachers [24]. Example uses of 'with' include:

- **prototyping in what resembles an object-oriented style**: This is the most orthodox application of the with-construct, to derive variants from a prototype: "X is a Y with a Z like this ...". To explore this, invoke the MCE and load the script at tutorials/scoping/arrow as described above.

- **generation of loci that reflect sophisticated forms of observation**: For instance (see Figure 2a), if the observable 'mercury' is a circle that represents Mercury in a simple planetary model, and the position of 'mercury' depends on the time-based observable 'tick' then

\[
\text{smallMercury is mercury with radius is smallRadius}
\]
\[
\text{traceMercury is smallMercury with tick ranging over (tick - memory) up to tick}
\]

Parametrically defined geometric figures can also be defined in this way. For instance (cf. Figure 2b):

- the ellipse \((a \cos(\theta), b \sin(\theta)) | 0 \leq \theta \leq 360\) and
- the sine curve \((\theta, \sin(\theta)) | 0 \leq \theta \leq n \times 360\)
creating generic configurations of mathematical interest: A good application for making construals has been creating artefacts that can help to expose abstract mathematical concepts and results. For instance, the Marriage Theorem is a basic theorem in matching theory which gives a simple criterion for a bipartite graph to have a perfect matching. In the past, it has been relatively straightforward to make a construal that captures the relevant observables for a specific small bipartite graph (say m by n, where m=n=4), but exceedingly tiresome to handle larger examples, and impossible to generalise for larger m and n without introducing complex macros. The with-construct makes it possible to construct generic observables to represent bipartite graphs and matchings in ways that are cognitively more congenial and better capture the spirit of mathematical abstraction.

- expressing modes of observation that involve surveying many states: as when examining several game states (cf. the listing below), or generalising geometric configurations featuring in games, such as the OXO-like games discussed in [6], which include 'cubic', a 3D version of noughts-and-crosses.

```
end_of_game is owon || xwon || draw;  ## the criterion for the game to be over
winlineix is lin_w with lin is alllines[ix];
winlines is winlineix with ix is 1..8;  ## which of the winning lines has two Os, if any
gapinlin is 1 if lin[1]==0 
    else (2 if lin[2]==0 
    else (3 if lin[3]==0 else 0));  ## where there is no entry on line lin
playonlinix is gapinlin with lin is alllines[ix];
playonlines is playonlinix with ix is 1..8;  ## where you can play on a winning line
alllinesindices is alllines with
    s1 is 1, s2 is 2, s3 is 3,
    s4 is 4, s5 is 5, s6 is 6,
    s7 is 7, s8 is 8, s9 is 9;
winindex is _index if winlines[_index] else 0;
iswinindex is winindex with _index is 1..8;  ## indices of lines with two Os, if any
wline is max(iswinindex);
when ((player==o) && wline>0 && !end_of_game) {
    boardstate[alllinesindices[wline][playonlines[wline]]] = o;
}
## if player O is to move and there is a line with two Os and the game is not over
## make a winning move by placing an O in a gap on such a line
```

Listing 1: An experimental script to reflect human observation when seeking a winning move
4. The with-construct from a programming perspective

What is the significance of \texttt{with} from a programming perspective? In [8], where the relationship between ‘making construals’ and proposals for new approaches to programming by Bret Victor [14] and Chris Granger [10] is discussed, much emphasis is placed on the distinction making construals and programming. This seems to be appropriate, not least because the program-like behaviours that are realised through what was characterised in [6] as ‘agent-oriented modelling with definitive representations of state’ have clear limitations. Framing behaviours in terms of interactions with definitive scripts is ill-suited to programming dynamical systems, does not address the challenges of achieving computational efficiency, and (the \texttt{with}-construct notwithstanding) does not suit large-scale computational applications where anonymous data variables cannot be avoided. But where definitive scripts were recognised in [6] as a powerful way of making the interface that mediates an agent’s \textit{action} intelligible, the \texttt{with}-construct may be seen as serving a complementary function: enabling the context for an agent’s \textit{observation} to be conveniently transformed. In this way, it becomes possible to make more direct links between programming and observation than before.

Listing 1 above is an experimental script that illustrates some of the significant features of introducing the \texttt{with}-construct. It is drawn from [2], where it can be studied in context. This script exploits many instances of \texttt{with} to describe the observation required by player O when making a winning move in noughts-and-crosses. The use of \texttt{with}'s may be compared and contrasted with the directly translated extracts from the original script for the OXO grid (cf.the ‘outline of the OXO-model’ presented in ‘Listing 1’ as an Appendix to [6]) that are shown in the left hand panel and in the bottom half of the right hand panel of Figure 3 below. Though these extracts illustrate a mode of observation much simpler than that expressed in the above listing, it requires definitions that exploit maker-defined functions (such as \texttt{noofpieces()}) formulated in a conventional procedural programming style. This tends to subvert the notion that making construals through modelling with definitive scripts is conceptually simpler than traditional programming. As the comments in the Listing 1 show, each of the \texttt{with}-definitions admits a conceptually simple interpretion in observational terms. In a similar manner, the definition of the observable \texttt{xwon} in the top half of the right hand panel of Figure 3 expresses the notion of surveying each of the possible winning lines (‘alllines’) in a fashion that is faithful to human observation and is in sharp contrast to the clumsy procedurally expressed function that is used to check the status of each of the potential winning lines in the original OXO-model.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{oxo-grid.png}
\caption{Representing the OXO grid without and game status with and without \texttt{with}'s}
\end{figure}
The value of linking programming with observation is being recognised in teaching programming. In his account of work with kindergarten children, Ivan Kalas [11] alludes to Blackwell’s characterisation of programming (cf. [7]), whereby “the user is not directly manipulating observable things, but specifying behaviour to occur at some future time”. He then highlights the importance of scaffolding the transition from specifying action through direct manipulation of observables to full automation.

Similar principles are illustrated in the work of Dave White [15] on teaching programming in an ‘unplugged’ fashion by encouraging learners to conceive systematic ways of carrying out tasks such as ‘drawing a grid’. White has developed some ingeniously crafted environments that support the transition from walk-through to automatic execution of such systematic procedures. His work highlights the critical role played by the mode of observation of a situation in specifying programmed activity within it. Figures 4a and 4b below show how the different modes of observation of a grid of this nature can be expressed in the MCE using the with-construct.

![Figure 4: Conceiving a 3 by 3 grid (a) in terms of lines and (b) in terms of squares](image)

A construal of 'giving change' inspired by Phil Bagge's Coins example in [1] shows how this notion can be generalised to transform a strategy for teaching a learner to observe coins in such a way that they are eventually able to give change into a 'program' for giving change that is flexible enough e.g. to cope with missing denominations and can deal with different currencies. This example also exhibits the use of recursively scoped with-definitions. A presentation, built into the MCE [23], that introduces this construal can be accessed online by loading the script givingchange/commentary2 from the project manager of the MCE.

5. More about the with-construct

The fundamental respect in which realising program-like behaviours through making construals differs from traditional programming is that it makes a clear separation between two aspects of the computer's role:

- the computer performs dependency maintenance, ensuring that the relationship between observables is maintained so that it is closely matched to the observations of significant entities in the application domain. This dependency maintenance is a background process that provides the stage on which the state-changing actions of the agents within the application are played out.
- the computer automates the state changing actions that are characteristic of the application by redefining observables explicitly following a prescribed algorithmic pattern. The algorithmic specification is much simpler than that in a conventional program, since a single redefinition of an observable affects the values of all dependent observables without the need for explicit updates.

These two aspects correspond to what was characterised in [6] as 'definitive representations of state' and 'agent-oriented modelling' respectively. The powerful advantage of the above separation of concerns is that it makes it possible to integrate modes of agency that in traditional program are conceptually quite distinct. As explained in detail in [3], after making an appropriate construal of the application domain, implicit dependency maintenance sets the stage for redefinition of observables to
represent state-changing actions that might be interpreted, according to the situation, at the discretion of the maker, as ‘program design’, ‘implementation’ or ‘use’.

Prior to the introduction of the with-construct, the benefits of being able to blend design, implementation and use in this fashion were compromised by the fact that scripts of definitions were statically defined and could not vary in size dynamically as observables in the application domain varied. As a simple example, in making a construal of a simple game such as Conway’s Life [25], where the growth of the population of cells cannot be predicted, there was no easy way to meet the need for an unbounded number of observables. This meant that, whilst the model of cell population was in some respects far richer than that supported by (say) a functional programming implementation of Life (for instance, modelling each game state explicitly and enabling the observer to intervene in the simulation to add or remove cells or to adapt the rules on-the-fly), it could not emulate the scope for unbounded extension afforded by lazy evaluation.

As far as the nature of the with-construct is concerned, there are clear parallels with prototype-based object-oriented development and with the replication of patterns of dependency that can be achieved (e.g.) through cutting-and-pasting rows in a spreadsheet. The with-construct can also be regarded as performing a scoping function somewhat resembling that available in JavaScript, in that it changes the context in which observables in a script are interpreted. Long experience of trying to integrate standard object-oriented notions into modelling with dependency networks suggests that the role that dependency plays in this context is semantically and logistically crucial. On this basis, the parallel with spreadsheets is probably the most significant. The extent to which this is corroborated by our practical examples will be an interesting topic of discussion. The PPIG audience may be familiar with other potential parallels to be considered.

A further aspect of the with-construct that may be of particular interest to a PPIG audience concerns its most effective representation. At present, whilst it is not too painful to trace the stream-of-thought that leads to the construction of a script that includes several with’s, it is not so easy to reconstruct this stream-of-thought, which is closely linked to a sequence of specific kinds of observation, by casual inspection of the script (cf. Listing 1 above!). This is in contrast to the directness of the connection between an observable in a construal and its counterpart in the referent.

A further consideration is that, in order to make most effective use of the potential for replication that with affords, it seems to be necessary to allow the component observables introduced in a with-clause to have independent definitions. This is in danger of undermining a highly significant feature of definitive scripts as so far conceived: the fact that the value associated with an observable or component of an observable can always be inferred from a single definition. This principle, which is in some sense a counterpart – in a semantic frame that has inherent a notion of ‘the present moment’ – of referential transparency in a declarative programming language, makes a major contribution to the intelligibility of scripts. If overriding of component definitions is permitted, it becomes essential to structure scripts so as to make it possible to retrieve the authoritative as-of-now definition of all observable components. For instance, when the script fragment below is interpreted sequentially:

```plaintext
a is 5;
alist is a with ix is 1..10;
alist[2] is 3*a;
```

the current value of the second component of the list alist is 15, whilst all other components are 5. Some syntactic convention is then needed to ensure that the overriding that occurs in this context can be easily detected when reviewing the script.

The need for independent definitions of the observables generated by a range scope can also be seen when observables such as buttons are generated in this way. It is essential to be able to distinguish between clicking on different buttons, and to be able to associate different actions with clicks accordingly. For instance, it is evident that in Figure 1 it is necessary to generate different actions according to which of the squares in the grid is selected on a mouse click. Figure 6 below (in which the syntax ‘::’ has been introduced as a synonym for with) illustrates work-in-progress on refinements of the MCE which help to address the concerns raised above.
6. Acknowledgements

Thanks are due to Dave White for introducing us to ISPY, to Jane Waite for helpful advice on the 'giving change' construal and to Mike Joy for acting as the Coordinator of the CONSTRUIT project. We are also grateful to three anonymous reviewers for their helpful feedback. This project has been funded with support from the European Commission under the Erasmus+ programme (2014-1-UK01-KA200-001818). This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

7. References


15. White, D. ISPY and PUSH-PYTHON. http://ispymath.com/


17. https://www.theguardian.com/crosswords/prize/26741


21. The Empirical Modelling website: go.warwick.ac.uk/em

22. The CONSTRUIT! project website: www.construit.org


24. Getting Started with JS-Eden

https://www2.warwick.ac.uk/fac/sci/dcs/research/em/construit/year2/c15/forteachers/

The End-User Programming Challenge of Data Wrangling

Maria Gorinova
Computer Laboratory
University of Cambridge
m.gorinova@gmail.com

Karl Prince
Judge Business School
University of Cambridge
k.prince@jbs.cam.ac.uk

Sallyanne Meakins
Papworth Hospital
University of Cambridge
sallyanne.meakins@nhs.net

Alain Vuylsteke
Papworth Hospital
University of Cambridge
a.vuylsteke@nhs.net

Matthew Jones
Judge Business School
University of Cambridge
mrj10@cam.ac.uk

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

Abstract
We present a case study of requirements for “data wrangling” capabilities in a healthcare application context. Data wrangling is an increasingly common requirement for data scientists, policy makers, market researchers, intelligence analysts, and other professions where existing data must be used in ways that were not envisioned when it was first collected. We characterise data wrangling as a programming problem, in which aggregate data must be restructured in ways that remain consistent with its semantic origins or ontological referents. We recommend the table as a lowest common denominator representational device, affording both direct manipulation and programming by example. We describe work in progress, in which we have identified new opportunities for clinical end-users to interact with the content of a customisable information system, through a focus on tables as an approachable analytic tool.

1. Introduction
Much of the work of the professional data scientist is concerned with ‘data wrangling’ – organising data into a form that will allow it to be used for statistical analyses or visualisations. Data journalist Simon Rogers, as with many data analytics consultancies and other less public-minded specialists, finds this the most time consuming and technically challenging aspect of the work, up to 80% of a typical data analysis project (Rogers 2013). In his guide for the budding data journalist, Rogers suggests that this might involve copy and pasting values between Excel tables, before converting to comparable units, getting rid of unnecessary columns, merging cells, and changing it into a format compatible with visualisation systems.

Much of this work is currently manual, although tools such as TextWrangler can automate the extraction of numerical data embedded within XML or other text formats, and mashup techniques such as Yahoo Pipes can be used to scrape data and aggregate data from websites. Recognising and exploiting regularity in textual data has been a topic of concern at PPIG in the past (e.g. Church 2008, Blackwell 2001). However, it is often the case that the data is already in a structured form, simply the wrong structure for the task at hand. The data of concern may be stored in database, or even simply embedded within a large table, requiring queries to extract it, joins to combine it, or ‘pivot’ operations to unpack and reorder a nested structure.

In this paper, we report early results from a project ‘in the wild’, in which data wrangling has been identified as a central issue for end-user professionals. Our collaboration involves researchers from a business school, a computer science department and an NHS hospital, with the goal of helping hospital clinicians to make better use of patient data. Our ultimate objective, as with other fields of data analytics and data journalism, is to create visualisations and conduct statistical investigations as an aid to improved policy and care. However, results from our first phase of fieldwork indicate that data wrangling is one of the most substantial technical obstacles to this goal. In the remainder of this paper, we describe the problem we have observed, discuss the specific technical opportunities that have been highlighted by our investigation of this domain, and present some initial design principles that we are now exploring as the basis for novel end-user programming tools appropriate to the users we have been meeting.
2. Background and previous research

2.1. Research context

The research that we report in this paper is part of a project entitled Repurposing Clinical data for quality improvement in Critical Care (ReCliC), funded by the Health Foundation. The goal of the ReCliC project, as expressed in its title, is to explore ways in which data can be used more effectively in clinical practice. We take a specific focus on the clinical context of critical (or intensive) care – a relatively complex healthcare context, involving greater degrees of monitoring and intervention than most healthcare work. One consequence of this complexity is that larger quantities of data are generated in a critical care context, and hence that automated approaches to working with that data are particularly likely to be profitable.

The customisable clinical information system that is the focus of the current research has previously been studied in the context of end-user programming research (Morrison & Blackwell 2009). In a previous study, we investigated the potential for end-user programming tools to offer additional value to such systems by hiring a professional programmer to carry out extension work, under the instruction of a clinical end-user (Blackwell & Morrison 2010). In that study, we observed that the most pressing need for system extension was with regard to structuring the data in new ways, allowing for aggregated reports to be generated. In this work, drawing on a wider variety of informants from additional institutions, we have again identified the problem of reorganising and restructuring aggregated data.

The current research focuses on five NHS Critical Care sites across the UK, all employing the same clinical information system (CIS). At each site we engaged with key users responsible for supporting the clinical information system as well as servicing requests for data. The roles of key users varied significantly as well as their range of technical and clinical skills. This included a nurse, pharmacist, consultants, IT staff and an information analyst. We also engaged with a wider group of end users who typically enter data into the clinical information system but also request data for various purposes. End users included both clinical and non-clinical staff.

The research methodology is based on conducting semi-structured interviews with research subjects as well as observations of how the clinical information system is used in routine practice. Additionally, we have attended user group meetings where challenges faced in using the clinical information system are discussed and users share learning. The qualitative data collected from these interactions are coded for relevant themes, discussed amongst the research team and refined through a repetition of these steps.

Future phases of this research will explore clinical potential for novel statistical analyses, including time series trends, sample comparisons, correlations and clustering. We expect that end-user programming perspectives will also be relevant in this future work. However, in the current study, we focus purely on the end-user challenge of gaining access to data, and organising it into a form from which statistical analyses might be conducted – the challenge of ‘data wrangling’.

2.2. Previous research in data wrangling

Data wrangling tasks have previously been identified as a candidate for end-user programming techniques. For example, tools such as PADS (Fisher & Walker, 2011) and FlashExtract (Le & Gulwani, 2014) allow users to extract structured data from semi-structured data, by either describing with the aid of a description language how the data should look like (PADS) or demonstrating it through an example (FlashExtract). Other tools focus on improving ways of connecting content from several different sources into a single Mashup, through programming by example techniques (Tuchinda, Szekely, & Knoblock, 2008). Further to that Vegemite uses a spreadsheet-like interface, together with direct manipulation and programming by demonstration techniques to achieve that (Lin, Wong, Nichols, Cypher, & Lau, 2008).

The research strategy that we have chosen is to treat data wrangling as a transformation problem, in which one set of data must be transformed into another. Based on the observation by Rogers (2013) that much routine data wrangling is carried out in spreadsheets, and on recent observation of the value that tabular representations provide in end-user data analytics (Sarkar et al 2014), we focus on
approaches to transforming one table of data into another. Previous approaches include systems that use menus, demonstration or examples to specify a spreadsheet or table transformation program.

Menu-driven tools allow the user to create programs by specifying a sequence of operations through a menu. One such tool is AJAX (Galhardas, Florescu, Shasha, & Simon, 2000), which uses an SQL-like language and focuses on data cleaning – the process of resolving inconsistencies in the data, performing entity resolution and correcting errors. Other systems, such as OpenRefine (Verborgh & Wilde, 2013) allow some commands to be specified graphically, but mostly users need to write them in a command language. Potter's Wheel (Raman & Hellerstein, 2001), on the other hand, has purely menu-based interface, used by the end-user to describe a sequence of operations, which could then be saved and applied to new data in the future. It relies on users constructing transformations gradually, i.e. having a clear idea of what sequence of operations they have to execute. This implies a need for strong understanding of the semantics of the operations, rather than just their effect.

Wrangler (Kandel, Paepcke, Hellerstein, & Heer, 2011) and its improved version, Proactive Wrangler (Guo, Kandel, Hellerstein, & Heer, 2011) take the Potter's Wheel approach one step further, by extending the transformation language and introducing an interface that allows the user to demonstrate what their intent is. This starts by selecting columns or rows relevant to the transformation, and is followed by suggestions from the system regarding what the intended transformation might be. The suggestions given by the Wrangler and Proactive Wrangler systems address the problem of the user needing a clear idea of the operations they need to go through. However, neither of these systems manages to completely solve this problem, still requiring the user to understand the particular semantics of the backend transformation language.

FlashRelate (Barowy, Gulwani, Hart, & Zorn, 2015) and previously ProgFromEx (Harris & Gulwani, 2011) allow users to give examples of the spreadsheet transformations they want to perform. The transformations are then automatically synthesised. The advantage of this approach is that users only need to understand the effect of transformations, not their semantics. However, a resulting problem with these approaches is that if the user does not understand the semantics, they find it difficult to verify that they have created the right program, to correct possible errors in the synthesised program or understand how it might be reused in future.

Our aim is to ease data wrangling tasks by unifying existing approaches such as Wrangler and FlashRelate, prompting the user to specify an example of the effect of the desired transformation program, automatically synthesise a suitable program, and allow the user to understand and amend it when needed.

3. Preliminary findings
In this section, we discuss findings from our field research into the organisational factors that set the context for reuse of clinical data, and analyse the characteristics of the data itself from an end-user programming perspective.

3.1. Organisational factors in data reuse
Our initial field research identified two constituencies of people who might be regarded as ‘end-users’ from the psychology of programming perspective. Neither is formally trained in IT (apart from the product training provided by the CIS supplier, together with any local or user-group support). However, one group has acquired primary responsibility for the data wrangling task. Morrison and Blackwell (2010) presented a case study of one such individual, who was described in that publication as a ‘professional end-user.’ In the current analysis, we describe these individuals as ‘key users’. The other group wishes to make use of the data, but currently has no tools that they find suitable for extraction of data from the system. We call these (data) ‘end users’.

In the remainder of this section we describe the ways that these two groups interact with the system within their organisational context. There are also organisational constraints on their work, sometimes reflected in their formal posts. Key users are often people who have been assigned specific responsibility for system operation, while data end users are more often clinical staff or hospital managers who need access to data for other purposes. Some of these staff are paid for the time they spend analysing data or interacting with the CIS, while others engage with the system outside of their
primary duties. However, there is not a straightforward mapping from the category of paid professional duties to the key user / data end user distinction. In some sites, the key user is a clinician or manager whose primary responsibilities are in clinical treatment or hospital management rather than IT systems.

In studying the typical process of interaction, we found that data end users requested data for a variety of purposes, including regular reports for clinical and management purposes, data for audit purposes (often submitted for national audit requirements), and data for research purposes. Key users would agree any data request queries with end users, taking into account not only their understanding of what data was stored in the clinical information system but also how likely they were able to extract particular data out of the system. Investigating this process highlighted particular issues for both key users and end users.

The first stage we identified in servicing data requests is the negotiation between the key user and end user. Key users indicated that end users are often not very clear about what data they require and there is a need to elicit more specific information before the exact specifications are understood. End users had not often thought through queries and key users see part of their job as facilitating the development of the ‘right’ question. Part of this negotiation is for key users to make end users aware of what is pragmatically possible within the constraints of the clinical information system. Since key users know the CIS best, they have a sense of how queries need to be stated to be feasible and so need to negotiate end user expectations accordingly.

A significant issue highlighted by key users attempting to address requests for data is the limited functionality of the CIS in providing the means to extract data. A distinctive (and desirable) feature of the CIS is that the recorded data is fully customisable. Data fields and formats can be defined to suit local clinical practice, to accommodate particular combinations of monitoring and measuring equipment, and to support local innovation (Morrison, Blackwell & Vuylstekke 2010). As a result of this substantial degree of customisability, the data extraction tool provided with the product (called ‘Query Wizard’) needs to be highly generic, allowing for queries that match and extract any possible combination of data types and values. This extremely high degree of abstraction and customisability results in a cognitive dimensions profile closer to that of a professional programmer than typical end-user tools.

Documentation is limited, and those key users who do use the tools are self-taught, because formal training is not readily available. The need for data extraction has steadily increased in recent years so that those key users who have developed skills over time have offered to teach others in sites across the national health care provider. So cumbersome is the clinical information system data extraction process that when investigating small data sets e.g. a single or small group of patients, users typically choose to, or are often encouraged to, read data off the screen of the user interface intended for everyday observation and data entry, rather than attempt an automated data extraction process.

Aside from the difficulty in using extraction tools, an additional concern is that any data extracted using the tool is in a structure that does not lend itself to immediate analysis. Data is distributed in such a way that it is necessary for key users and/or end users to first manipulate the data set so that it is arranged in rows and columns that can be used to address the initial data request. In order to do this users turn to an intermediate tool i.e. spreadsheet software, to first organise data into suitable views. This is particularly necessary when dealing with large, more complex data requests. Given the limitations of the extraction tools, such queries need to be sub-divided into queries key users estimate will be executable. Each of these sub-queries is then aggregated and manipulated in a spreadsheet to build up the overall data query result.

Over time, key users have developed other ways to overcome data extraction issues. For example, one key user has kept a separate data set in a spreadsheet (manually updated at intervals) which enables her to more quickly query basic data in a format that requires less, and often no, manipulation after the query process.

Another way users have tried to overcome these issues is to circumvent the use of the CIS data extraction tools altogether. A subset of data collected in the CIS has to be submitted to the national
intensive care audit database ICNARC. This data is either extracted from the CIS automatically, or (sometimes) partially manually re-keyed into the required audit data set. Despite the challenges that this extraction process itself introduces, the ICNARC data set has a well-defined structure (as required by national audit specifications) which lends itself to being more easily queried than the CIS. Both key users and end users will use this data set in their local contexts to perform queries which they regard as a simpler process by comparison to using the CIS. Evidently the key constraint is that this is a subset of the entire CIS database and so can only support particular queries.

As alluded to previously, a constraint for key users is the limited level of skills they may have to perform data related tasks. While it is rare for key users to have data-related training, a few have made efforts to develop specific programming skills, such as SQL, to extend their ability to extract and manipulate data in more useful ways. In effect this enables them to bypass the CIS and the related data extraction tools, allowing them to directly access the database. Such efforts are still limited, however, because key users have a relatively basic understanding of the CIS database structure, thus placing a constraint on the effectiveness of data extraction.

A related issue limiting the ability of key users to service data requests is that they must typically combine such tasks with a wider set of responsibilities. Work on CIS tasks has to be balanced with, for example, clinical or other IT related work. This limits the amount of time key users can dedicate to working with the CIS as they prioritise more urgent work accordingly. Key users often have to make expeditious decisions as to whether it is possible to service a data request. If a request appears too complex and time consuming to extract and manipulate the data, it is likely to be deferred or rejected when the wider responsibilities of the key user are considered more urgent.

We observe that the consequence of this prioritisation is that end users may limit data requests over time. Knowing that key users are busy, or that a data request is likely to be too complex, results in end users either not asking the question at all or perhaps agreeing to simplify the question to match the time and abilities of the key user as well as the capabilities of the CIS, pre-empting the negotiation they are likely to have with the key user in agreeing the data request.

3.2. Data semantic roles

As mentioned in the previous section, a distinctive (and desirable) feature of this CIS is that the recorded data is fully customisable (Morrison, Blackwell & Vuylsteke 2010). This is achieved through use of an extremely general database model. The main data store is a single table of time-stamped entries, each consisting of an attribute-value pair. The set of possible attribute values is completely configurable in each installation of the system, defined as the set of ‘variables’ that can be recorded, reported, graphed or acted upon.

Each entry in a table represents a single observation – the type of observation or measurement that has been made, and the observed result or value of the measurement. This logic is compatible with the logic of the critical care unit, in which the clinical team are constantly making observations or measurements of the patients. However the practical application of these observations is constrained by a number of practical issues that arise from the characteristic ways data is collected for analysis. These different natural categories of data observation can be compared to the ‘roles of variables’ identified in conventional code (Sajaniemi & Prieto 2005), for example:

1. Statistical invariants: Apart from the changing observations, most other entries in the database are single values, not expected to change. We are not asserting that these things can never change, simply that they are invariant for the purposes of clinical reasoning. Examples include data of birth, height, eye colour etc.

2. Timebase resampling: Various observations are collected at differing intervals. One aspect of a patient's condition might be recorded once a day, another aspect three times a day, another every hour, and another (automatically measured) every five minutes. Whenever it is necessary to compare or relate these observations, some kind of interpolation is necessary, whether as simple as using two values as close to each other as possible (constant nearest neighbour), linearly interpolating between points on either side, or fitting a regression model to account for sources of variance.
3. Persistent legacy data: When new variable types are added to the local CIS configuration, it is not possible to delete the previous variables (because this would invalidate existing queries and reports). Data associated with those variables is therefore retained in the database, with local practices used as work-arounds to prevent confusion (for example, renaming out-of-date attributes with the letter ‘z’ as a prefix, so that they are not inadvertently selected when attribute values are offered in alphabetical order).

3.3. Data ontology

Our focus in this project on a specific domain has highlighted the ways in which the degree of abstract generalisation supported by customisable tools might be unhelpful. This builds on observations with regard to design-time abstraction (Blackwell et al. 2008), and also on the suggestion that external representations can be more usable if made less abstract, so that users are able to reason about more specific interpretations (Stenning and Oberlander 1995). These considerations are aspects of the cognitive dimension of abstraction, which we believe is a central issue in the respective approaches of key users and data end users. In this section, we therefore offer some further analysis of the types of abstraction that have resulted from the highly customisable CIS system.

In the relational model, data types are ontologically undifferentiated, with attention paid only to the compatibility of machine data types, the specification of whether a value might be used as a relational key, and possibly special viewing or editing facilities for free text or binary objects. We speculate that in an end-user context such as the repurposing of patient data, this extreme degree of abstract generalisation is unhelpful. For example, the name of a patient “Adam Smith” and the name of a surgical procedure “coronary artery bypass graft” are both text fields. However, there is no clinical context in which it would be meaningful to compare these two values to each other, or to count the relative numbers of each. Our hypothesis is that manipulation of data will be easier for end-users to achieve if values such as these cannot be confused, by means of detecting and maintaining the ontological referents of different types of data in the database.

The design of programming standards to reflect ontological categories is reminiscent of the People, Places and Things design standard promoted by Apple’s Taligent spin-out in the 1990s (Cotter & Potel 1995). Taligent failed in part because of failure to anticipate that most software ‘objects’ would not be ‘things’, but rather purely engineering abstractions such as lists, buffers, wrappers, sockets and so on (Blackwell 1993). More seriously, perhaps, they were unable to enforce a business model in which a single understanding of ontology could be imposed on potentially incommensurable knowledge systems (Star & Bowker 1999). Attempts to create ontologically-justified data type protocols continued in initiatives such as the Object Management Group’s Business Object Model (BOMSIG) (Zamir 1998), or continuing Semantic Web work of the W3C Web Ontology Language (McGuinness & Van Harmelen 2004). Arguably, all of these share the fallacy identified by Umberto Eco as the "Search for the Perfect Language" (1995).

A lighter-weight approach to ontology and types can be found in Scaffidi’s Topes proposal (Scaffidi et al 2008), as well as in the unit inference approach to spreadsheets developed by Abraham and Erwig (2004). Ontological differentiation might recognise common sense categorisations, such as TIME IS ORDERED, CAUSALITY IS UNIDIRECTIONAL, and PERSONS ARE INDIVIDUALS. These categories also aid statistical reasoning. For example, it is reasonable to investigate whether the time of day for surgery has any consistent effect on the outcome of an operation. It is less reasonable to investigate whether the time of day for surgery has a consistent effect on the patient’s date of birth. By taking the ontological referents of data into account, the tabular structure of the data can help to guide statistical enquiries.
4. Potential design strategies

We propose a design for a system, which guides the user through constructing a simple example explaining their intent, automatically synthesises a likely spreadsheet transformation program which matches this example, and then executes the program, having the rest of the spreadsheet as an input. Below, we present our approach by describing its three dimensions, following best practices of the program synthesis community (Gulwani, 2010): the way the user shows what their intent is, the space of possible programs, and strategy for searching the space of possible programs.

4.1. Specifying the intended program

One problem with many menu-driven interfaces for data wrangling is the necessity for the user to have clear idea of the meaning of each of the suggested transformation steps. Naturally, this means a steep learning curve, making it hard for novices to use the system. In the context we have been studying, it is apparent that CIS users such as nurses and doctors often lack the time and access to a technical person who might help them learn how to use new software. Rather than menu-driven specification, we have therefore focused on the development of a demonstration-based approach, in which users interact directly with the data to select the items that are of interest. The interaction method that we have created, responding to this requirement, is called Data Noodles, and is described in a companion paper (Gorinova, Sarkar and Blackwell 2016).

4.2. Transformation language

The transformation of spreadsheets, selecting and restructuring them in response to the user’s actions, is achieved with a Domain Specific Language (DSL) largely inspired by the Potter's Wheel and Wrangler's transformation languages. It implements reshaping transformations, such as Fold and Unfold, substitution of missing values, and we plan to add support for data value interpolation, filtering, splitting and merging operations. We choose to use a language adapted from the two above, as this previous work has found them sufficiently expressive for common transformations, and furthermore, providing greater readability than more ‘fine-grained’ languages in which the program is a collection of constraints determining the value of each cell in the output spreadsheet (Barowy et al., 2015; Harris & Gulwani, 2011).

A key requirement for the CIS context is an interpolation operation, which fills in missing values of a column, by estimating them based on some function. In the context of electronic health records this is particularly useful, as often timestamps of different continues parameters do not match, e.g. heart rate could be recorded every minute, but respiratory rate every five minutes. Thus, if a clinician wants to analyse a dependency between a patient’s heart and respiratory rates, they either need to delete all extra heart rate entries (which will work only if the two time-series have some data points taken at the same time) or they could speculate about the way parameters behave between measures. Our interpolation operator would offer a range of options for describing this speculated behaviour, such as ‘copy the last known value’, ‘copy the nearest known value’, ‘linearly interpolate’, etc.

4.3. Program synthesis

To synthesise programs in the DSL described above, we have been using the PROSE SDK (Polozov & Gulwani, 2015). PROSE is a .NET framework which provides tools for defining the syntax and semantics of a domain-specific language and can then synthesise a ranked set of programs satisfying some input-output example specification. To be able to synthesise spreadsheet transformation programs with PROSE, we define the following:

- Syntax: a text file defining the syntax of the transformation languages as a context free grammar.
- Semantics: a static C# class defining the operational semantics of each symbol of the grammar. That is an actual implementation of what each of the spreadsheet transformation language operations does.
• **Witness Functions**: a static C# class defining the *inverse semantics* of each symbol in the grammar. A witness function deduces information for a parameter of a function, given some information about the entire function. For example, consider the column renaming function \( \text{Rename}(S, \text{OldColumnName}, \text{NewColumnName}) \), which returns \( S' \) – the spreadsheet \( S \), but with the name of the column \( \text{OldColumnName} \) changed to \( \text{NewColumnName} \). A witness function of \( \text{Rename} \) for \( S \) is then:

\[
W_S(S') = \text{Rename}(S', \text{NewColumnName}, \text{OldColumnName})
\]

• **Scoring Functions**: a static C# class defining a *scoring function* of each symbol in the grammar. That is, we inductively define a score of the entire transformation program, based on its components. For example, we adopt an Occam's razor approach -- the longer a program is, the lower its score is, as we believe that the most probable program the user wanted, is the simplest one that explains the input-output example.

The above definitions allow PROSE to find spreadsheet transformation programs that are likely to describe the user’s intent, based on the example the user has provided.

**5. Discussion**

We have presented a context in which users face a very specific end-user programming problem arising from the need to restructure data from one loosely constrained context (a configurable information system) to another (exploratory statistical analysis). It appears that many of the challenges they face could be addressed by an example-based interaction approach, in which they would demonstrate the data format that they need, and that this demonstration could be used to synthesise a data transformation program. We have presented an overview of a program synthesis approach that could be used for this purpose, if made sufficiently accessible.

In a companion paper (Gorinova et al 2016), we propose an interaction paradigm called Data Noodles, that we hope will be accessible to users, while offering sufficient power to achieve the data transformations appropriate to this application domain. In future, we plan to integrate ontological representation elements into this paradigm, in a way that will help constrain the statistical analyses that might then be carried out with such data.

**6. Acknowledgements**

This research is funded by the Health Foundation.

**7. References**


Abstract
The importance of the education in informatics, also in non-vocational curricula, was recognized in the Italian school system many years ago. The introduction of the new discipline in a complex and articulated educational organization is still a work in development and its implementation may differ a lot across schools. Hence, computing background of bachelor students is really diverse. The teacher in this situation has to manage very different levels of skills and students often have to cope with failures and frustration. So, also motivation and emotions could have a role in determining performance. In this work, we present an early stage analysis of the connection between motivation, emotions and performance in initial learning of programming for bachelor students in Applied Mathematics in Verona. Performance in programming correlated positively with believes on control and negatively with anger and hopelessness. This finding supports the relevance of intervention programs promoting efficient motivational strategies and a positive emotional climate during learning of programming.

1. Introduction
Procedural methods of solving exercises are very common in primary and secondary school, particularly in scientific and technical disciplines. All the same, learning how to solve a problem in an algorithmic way and using a programming language is a non trivial task, even at the end of secondary school.

A course in programming requires fundamentals skills in the use of computers and assumes basic general notions about informatics. Many students have difficulties in thinking about problems in an algorithmic way: difficulties in understanding the problem and rigorously characterize it, problems in describing its solution informally, problem in coding it in a programming language (Wing, 2006), (Katai, 2015).

Some students reveal a lack of motivation, which is a relevant component of a successful learning activity (Jenkins, 2001). As matter of fact, in the secondary school with the word "informatics" people frequently refers to digital literacy or to the use of the information technology, rather than to computational or algorithmic thinking. Nevertheless, other scientific subjects, as mathematics or physics, may be seen as problematic and even abstruse, but hardly useless.

The role of motivation on learning outcomes is widely supported in educational psychology. However, a few number of empirical studies reported in the literature concerns computer programming courses. First year programming students in two university showed low level of intrinsic motivation (Jenkins, 2001). Students’ learning motivation in learning programming seems to be connected with self-efficacy, which is demonstrated to be an antecedent of performance (Law, Lee, & Yu, 2010). The study assessed the relationship between motivation and self-efficacy, but did not actually observe performance. Kumar (Kumar, 2014), assessing affective learning - that is motivation, attitudes and emotions associate to
a learning experience - connected to the use software tutors to learn programming concepts, showed that no differences in the level of affective learning was found considering sex, demographic group, and specificity of the major. Emotions seems to influence students’ learning motivation, even if the effects of emotions on performance are complex, and more research is needed to disentangle the causal relationships of emotions with other variables inside a learning situation (Mega, Ronconi, & De Beni, 2014). The role of emotions in learning programming is not so highly studied yet, even if an attempt to understand emotional experience of learning to program is described by (Good, Rimmer, Harris, & Balaam, 2011). The authors underline the importance for the learner to assess emotions during the programming course experience, mainly because it increase emotional awareness. However, this study report qualitative data useful to understand students’ feeling and thoughts concerning the use of this kind of assessment, and few information concerning the relationship between achievement emotions during learning experience and performance in the topic.

Control-value theory (Pekrun, Goetz, Frenzel, Barchfeld, & Perry, 2011; Pekrun & Perry, 2014; Raccanello, Brondino, & Pasini, 2014) defines achievement emotions as those emotions linked to learning activity or outcomes, differentiated for valence (positive and negative emotions) and activation (activating and deactivating emotions). This theory also considers some motivational antecedents of emotions, specifically people’s beliefs concerning their control on the situation and the value they attribute to the task. The model considers the role of emotions, and of their motivational antecedents, in predicting performance.

The aim of this research is to investigate the relationships among motivation, emotions, and performance to better understand which factors can affect both students’ emotions and performance.

2. Method
2.1. Participants
We consider students of the Bachelor program in Applied Mathematics enrolled at University of Verona attending the first year. The sample group consists of 64 students (49% males, mean age 20) enrolled in the first year. A subsample of these students was asked to fill in a self-report questionnaire on antecedents of achievement emotions and achievement emotions, and 18 students accept to participate (mean age was 20.3, 52.6% males). This subsample was used for this preliminary correlational analysis.

2.2. Teaching method
In the current academic year, we introduced a "light" version of eXtreme Apprenticeship (XA) in the practical part (Laboratory). This teaching methodology has been described in (Vihavainen, Pakula, & Luukkainen, 2011) for Programming in Finland and also in other contexts in Italy (Dodero & Di Cerbo, 2012), (Del Fatto, Dodero, & Lena, 2015), (Del Fatto, Dodero, & Gennari, 2015), (Del Fatto, Dodero, & Gennari, 2014). Its application in Verona is presented with more details in (Solitro, Zorzi, Pasini, & Brondino, 2016).

Extreme Apprenticeship is inspired to the Cognitive Apprenticeship (CA) learning model (Collins, Brown, & Holm, 1991). In CA the teacher gives students a conceptual model through the interactive presentation of working example. Then students solve exercises under the guidance of an experienced instructor; also they can receive hints to be able to discover the answers to their questions themselves. Finally they should be able to master tasks by themselves.

In the XA approach the hours devoted to frontal lessons are reduced. XA practical activities consist of a number of exercises of increasing difficulty that students have to solve actively interacting with the tutors and classmates. Suggestions are permitted, but no explicit or direct correction is allowed.

Programming exercises have also a positive impact on the motivational side (see, for example, (Brondino et al., 2015),(Bergin & Reilly, 2005).

In Verona we tried to applied these principle to our teaching, but we had to consider a working setting which is still "traditional": non continuous support by instructors, limited availability of laboratories,
and other technical concerns. The acceptance of the new learning method was not compulsory; hence a group of students (1/3) decide for the traditional way of studying.

The rest of them (2/3) took part actively to a practical training in the initial stage (the first two months of the course). We asked the students to solve practical programming exercises of increasing difficulty at university or at home. A few issues come out: a lack in the comprehension of the requirements in a few cases; the basics of the language (Python) are generally good; the correctness of the solution is achieved at most in the 70% of cases.

Finally the students have taken a test divided in two parts: a general part on the fundamental notions of programming; a practical part with three exercises about: the comprehension of an algorithm; the definition of an algorithm starting from its formal characterization; the analysis of a simple problem and the definition of the solution. The practical part may contain suggestions, and the teacher can answer to a few questions on the textual interpretations of the exercises, but there is no support during the examination.

2.3. Measures

Motivation. (Perceived control, task value, and self concept). We assessed students’ beliefs on perceived control, task value, and self concept with 12 self-report items. Items had to be rated on a 5-point Likert scale (1 = not at all true of me and 5 = very true of me). For perceived control (four items, e.g., I’m certain I can learn what we do in programming course) and task value (four items, e.g., I think programming is useful for me to learn), items were adapted from the Patterns of Adaptive Learning Scales, PALS (Midgley et al., 2000) and from Lichtenfeld and colleagues’ work (Lichtenfeld, Pekrun, Stupnisky, Reiss, & Murayama, 2012). For self-concept (four items, e.g., I get good results in programming), items were adapted from Goetz and colleagues’ work (Goetz, Cronjaeger, Frenzel, Lüdtke, & Hall, 2010)

Achievement Emotions. We assessed 10 achievement emotions with the Achievement Emotions Adjective List, AEAL (Pekrun & Perry, 2014) (Racanello et al., 2014), which consists of a list of 30 adjectives related to three positive activating emotions (enjoyment, hope, pride), two positive deactivating emotions (relief, relaxation), three negative activating emotions (anxiety, shame, anger), and two negative deactivating emotions (boredom, hopelessness). Items had to be rated on a 5-point Likert scale (1 = not at all and 5 = completely). Students were asked to evaluate how much they feel in that way related to attending the course.

Performance. Performance was operationalized in terms of the score obtained in the general theoretical part of the partial exam (G_score) and the score obtained in the exercises (Ex_score).

3. Results

Descriptive statistics. Analyzing students’ performance on the two parts of the partial exams, considering the whole sample, we observed that the 70% of the students had a positive evaluation in the general part; in the three exercises students got respectively 73%, 62%, and 55%. Mean values and standard deviations of perceived emotions are shown in Table 1. The highest level is perceived for hope (3.3) and the lowest for shame (1.4). Positive emotions showed a higher average level than negative ones (positive: 2.96, sd = .68; negative: 2.00, sd = .42) and this difference is significant (paired sample t-test: t(18) = 4.65; p < .001) with a large effect size (Cohen’s d: 1.1).

Relations among motivation, emotions and performance. Considering the subsample of students who participate to the assessment of affective constructs, we computed bivariate correlations among motivational antecedents, achievements emotions, and the two measures of performance (G_score and Ex_score). G_score was negatively correlated with one negative emotion, specifically boredom (r = −.48, p < .05). G_score was positively correlated with motivational antecedents, specifically perceived control (r = .53, p < .05) and self-concept (.58, p < .05). At the same time, Ex_score was negatively correlated with two negative emotions, specifically anger (r = −.62, p < .01) and hopelessness (r = −.54, p < .05).
Table 1 – Mean and Standard Deviation for each emotion (N = 19)

<table>
<thead>
<tr>
<th>emotion</th>
<th>valence</th>
<th>activation</th>
<th>mean</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enjoyment</td>
<td>positive</td>
<td>activating</td>
<td>3.0</td>
<td>0.8</td>
</tr>
<tr>
<td>Hope</td>
<td>positive</td>
<td>activating</td>
<td>3.3</td>
<td>0.73</td>
</tr>
<tr>
<td>Pride</td>
<td>positive</td>
<td>activating</td>
<td>3.1</td>
<td>0.83</td>
</tr>
<tr>
<td>Relief</td>
<td>positive</td>
<td>deactivating</td>
<td>2.6</td>
<td>0.76</td>
</tr>
<tr>
<td>Relaxation</td>
<td>positive</td>
<td>deactivating</td>
<td>2.9</td>
<td>0.95</td>
</tr>
<tr>
<td>Anxiety</td>
<td>negative</td>
<td>activating</td>
<td>2.5</td>
<td>0.69</td>
</tr>
<tr>
<td>Anger</td>
<td>negative</td>
<td>activating</td>
<td>2.0</td>
<td>0.74</td>
</tr>
<tr>
<td>Shame</td>
<td>negative</td>
<td>activating</td>
<td>1.4</td>
<td>0.52</td>
</tr>
<tr>
<td>Boredom</td>
<td>negative</td>
<td>deactivating</td>
<td>2.1</td>
<td>0.55</td>
</tr>
<tr>
<td>Hopelessness</td>
<td>negative</td>
<td>deactivating</td>
<td>2.0</td>
<td>0.63</td>
</tr>
</tbody>
</table>

4. Conclusion

Results concerning performance are encouraging, showing quite respectable performance, probably due to XA methodology (Solitro et al., 2016). However, we observed that a considerable part of the students decide to postpone the subject and give priority to more "traditional" subjects. This choice may be caused by low level of motivation and general learning overload. The higher level of positive emotions experienced by students in comparison with negative ones seems to highlight that XA methodology stimulates a positive affect, which in turn produces good results. To verify this hypothesis more research is needed, also comparing the XA methodology with traditional one, not only on performance but also on emotion undergone during the learning experience.

Looking at results on motivation and performance, this pilot study showed that students who perceived a high level of perceived control and self-concept, which are considered motivational constructs, had a better performance.

The results of the preliminary analysis on correlations among performance, emotions and motivation on this sub-sample of 18 participants were promising. Even if the sample is small, analyses underline the link between some emotions and performance, confirming that results reported by the literature are true also for programming courses. The relationship between anger, hopelessness and performance confirm that the more you feel angry and hopeless when attending the course, the worst is your performance (Pekrun et al., 2011).

An interesting result is the medium size negative correlation between boredom and performance; this means that, considering the general score, students who perceived high level of boredom, while attending the course, performed worse than students who perceived low level of boredom. Few researches concern the effect of boredom on academic performance, maybe because it looks like a "silent" emotion, not evident from the teacher’s perspective. Nevertheless, this result confirms some empirical evidence suggesting that lack of achievement values relate to boredom(Pekrun, Goetz, Daniels, Stupnisky, & Perry, 2010). Generally, empirical studies report a positive correlation between boredom, anger and hopelessness. Also in this pilot study these relations are confirmed (boredom and anger: .56; boredom and hopelessness: .45), and the model which connect these three emotion among them and with performance should be deeply explored.

These preliminary results highlight the importance to assess motivational antecedents and achievement emotions during the programming course, in order to help students to improve their performance, and, probably, to decrease dropout. A complete analysis of students performance, motivations and emotions will be possible only after the end of academic year.

A larger sample is needed to better understand the model of relations. For the future we plan a more systematic approach to the methodology and, if possible, a cooperation that also involves other universities.
5. References


Software and How it Lives On -
Embedding Live Programs in the World Around Them

Antranig Basman  
Raising the Floor  
amb26@ponder.org.uk

Luke Church  
University of Cambridge  
luke@church.name

Clemens Klokmose  
Aarhus University  
clemens@cavi.au.dk

Colin Clark  
OCAD University  
cclark@ocadu.ca

Abstract

Virtues beyond those that traditionally motivate live programming are needed to support lively and unbounded communities of authors collaborating by creating and using shared artefacts. We will argue for the importance of each element of an artefact’s design to be externalizable, and introduce terms describing the function of parts of a fully capable live externalizable system (the res potentia and res extensa). We critique the standard presentation of live programming, situating it within a wider set of authorial values. We introduce the quantity of divergence of a programming language or system and explain the desirability of minimising it. We survey some existing systems through this taxonomy and speculate how future systems could improve on them.

1. Introduction

Liveness in software, as described by (Tanimoto, 2013), (Ungar & Smith, 2013), and others, is essential for bringing the affordances of reshaping software to everyone who uses software. Liveness by itself, however, is not sufficient to produce durable, sharable software artefacts supporting lively and unbounded communities of authors. We consider that the values of liveness are better understood when considered against the experience of networks of authors, rather than single ones. From this point of view, the strict values of liveness itself (that is, level of immediacy of perceived updates to the system) are less important than the authorial affordances of liveness, in terms of being able to use the system to author itself. In section 4.4.4 we critique (Tanimoto, 2013)’s taxonomy of levels of liveness, and point out that levels of liveness that he considers “higher” (the predictive levels L5 and L6) can be achieved without necessarily achieving the letter of liveness he considers prerequisite (immediate editing liveness at level L4). In sections 1.1 and 3, we situate what we consider underlies the real value of liveness in terms of the authorial cycle experienced both by single authors and networks of authors.

In this paper, we will introduce some new virtues which need consideration, as well as some new qualities, categories of design elements, and ways of looking at designs that they induce. These virtues support externalization — the ability for elements of a design to have mobility within a space of authors, to be shared and operated on by a variety of tools which provide “elite affordances” (version management, durability, unit and integration testing1) to those not traditionally considered part of the software engineering elite. We will say that a design, system or language fails to be properly externalizable if it, at runtime, exhibits divergence — a discrepancy between its bookkeeping, runtime state and the state with which it can be externally authored. We develop a taxonomy of system state based on (Kauffman, 2011)’s presentation of the terminology in (Whitehead, 1929), consisting of the res extensa and the res potentia. This taxonomy accounts for the authorial purpose and fate of system state in order to promote designs with smaller divergence.

1.1. Liveness and the Authorial Cycle

Here we outline the conceptual background for this paper, with a brief account of liveness and Whitehead/Kauffman’s taxonomy.

(Ungar & Smith, 2013)’s axiom of live programming is that “the thing on the screen is the thing itself.” There should be as little discrepancy as possible between the artefact shown to users and the system

---

1These are examples of affordances that are all taken for granted as available in any professional software development context, but are traditionally absent in user programming systems.
which can be used to build it. This implies, firstly, that a system can be authored in place — that is, authoring a system should not require intrusively tearing it down and rebuilding it, resulting in loss of access to the system during the interval it is being worked on. Secondly, it implies that all of the tools required to build and modify the system are delivered along with the system, and can be accessed through an interface appearing next to or on it.

Liveness has been enthusiastically accepted as essential by a small section of the programming community, but has not yet brought about a revolution whereby every piece of software that can be used, can also be modified, customised and shared. These are values that we take for granted with physical artefacts and tools, but are not yet widespread for their informational equivalents. In this paper we try to identify an authorial barrier which impedes the adoption of live systems, and consider ways that it might be removed.

In order to identify the nature of this barrier, we borrow some terminology from (Kauffman, 2011; Whitehead, 1929), originally used to describe the universe itself — this is natural since we expect that the function of software is to hold a mirror up to nature. In Whitehead’s model, reality consists of two Realms, the Possible and the Actual, in which Actuals give rise to Possibles, which in turn give rise to Actuals. We adopt this workflow as a paradigm for live programming, in which Possibles (considered as the design space of the system) give rise to Actuals (considered as running systems) through the process of execution. Then, the process of authorship, directed at the running system, as well as the process of Observation in general (interaction of the system with its context, the outside world) feeds back into the realm of Possibles.

In section 2 we’ll introduce the key quality of externalizability in a design, which enables a particular, intimate and bidirectional relationship between Actuals and Possibles, and in section 3 split the realms of Actuals and Possibles into particular domains named from (Kauffman, 2011)’s treatment, which names Actuals as constituting the res extensa (extended matter, the realm of that which is), and Possibles as the res potentia (potential matter, the realm of that which might be), which we will split into potentia I & potentia II.

2. Externalizability, Documents and Models

For elements of a designed software artefact to be easily shared and authored by a variety of different tools, live or otherwise, it is essential that they are externalizable — that is, for these elements appear with the affordance of documents. Documents, in our presentation, are transparently addressable by a natural coordinate system, and represent a format in which an exhaustive summary of state in one part of a running system can be transported from place to place. A natural coordinate system is one that allows units of the implementation to be addressed in their position within the document’s structure in such a way that combinations of expressions from different authors can be aligned and overlaid in a semantically meaningful way.

Documents with a basis for this idiom are described in (Fielding & Taylor, 2000)’s presentation of the REST idiom as an interpretation of the meaning and function of the web as operated by the HTTP protocol. In REST, an exhaustive representation of a resource is transferred during a conversation, rather than the more prevalent “API” or “message passing” style of conversation where transmissions consist only of answers to limited questions with an essentially arbitrary semantic. Fielding considers that documents should have a publicly intelligible textual semantic, but does not venture further into considerations of their support, via internal coordinates, for authorial networks.

We’ll use the term model (in the sense of the MVC community, rather than the sense of the model-checking community) to refer to the part of a design imaged when a document is transferred. All elements of the res potentia, and some elements of the res extensa (those that we will simply name models) consist of “model material” and are hence directly serializable as documents in this sense.

---

2Hamlet: Act 3, Scene 2 — “playing, whose end . . . is, to hold . . . the mirror up to nature”
The traditional, meagre externalisation of a program in the form of its source code is unsuitable to participate in the lifecycle we outline in section 1.1, since source code is a very fragile notation for authoring by means other than a human being sitting in front of a text editor. It has only a very weak correspondence in structure with that of the running program, and cannot be reliably processed by tools other than its one intended audience — the programming language’s compiler and its closely bound tools. One tradition, *Literate Programming* (Knuth, 1992), somewhat extends the audience for part of this text to include authors at design time. Specially formatted elements, embedded alongside elements of the source text, are marked for processing by a special tool chain which may either publish them as a static document describing the system, or else appear in a live interface assisting an author who is in the context of selecting between the elements designated by the source text. However, this text only has a unidirectional role in the authorial cycle — it consists just of human-readable text and can’t assist the system to operate on itself.

3. Actuals and Possibles in Software Systems

The web as seen in its “Web 1.0” incarnation, as it originally emerged in the 1990s, traffics purely in documents in their own right — those which represent a rendered web page, which are serialized as HTML and in model form are represented as the DOM. We call such a “document in its own right”, following Kauffman’s presentation of (Whitehead, 1929), a *res extensa* (representing *what is*). It consists of a fully actualised system — in this case, the web page itself, displayed in a browser.

However, an executing application, rather than simply a markup document, requires more sophistication than this — whether we are considering general user applications, or the “Web 2.0” AJAX-enabled web applications that emerged in the 2000s. In a dynamic system, we need to describe the space of entities that may come into existence, in addition to those that are currently running. In traditional software engineering, this is achieved through the ability to define classes, types, or other kinds of implementation unit which can then be instantiated. We call this kind of material *potentia I* (representing *what can be*), forming the first category of the *res potentia*.

The second constituent of the *res potentia*, which we call *potentia II*, represents *what is to be*. This is the registry of user expressions — that is, the collection of instructions they have given in order to designate that elements of *potentia I* should be instantiated, and the addresses they should be instantiated at.

In existing programming systems, extensa, potentia I and potentia II are entangled together, and few to none of them are externalizable. Although some systems separate the extensa from potentia I, none to the knowledge of the authors separate potentia II from the others. The process of defining and instantiating objects occurs at the same level of system design, and the processes of designating an object to be instantiated and initiating the instantiation process itself are not separated. Some examples of these situations appear in section 7.1.

4. Divergence

The Web itself, considered as simple system presenting only documents using an extensa (separate from its dynamic functionality added on by scripts), can easily meet our criteria for externalizability. However, the requirement to present arbitrary applications, rather than simply content, creates significant problems. Any realistic system, in the pursuit of animating its *extensa*, must make use of significant amounts of internal book-keeping information — which in practice it will be hard to imbue with the semantics of documents. This information is held in hard-to-access implementation elements such as the compiler’s runtime, the virtual machine or other libraries. This causes the state held in the extensa to *diverge* from a usably externalizable representation.

This problem is very likely insurmountable in its totality. However, certain choices of language and system design will tend to exacerbate or reduce the system’s divergence. Systems with lower divergence will tend to be easier to author and work with through a variety of tools (for example, textually-directed tools, version management tools, or live tools presented at the system’s own interface or remotely), since these tools can be targeted at the externalized, document form of the design, and hence address a greater
proportion of the real, executing design.

4.1. Divergence from the Stack
The divergence in a system or language typically also manifests itself as an obstruction to liveness itself — since the runtime or external tool has as a task to update the divergent material in some kind of consistent way with respect to the authorial updates. For example, (Church, Söderberg, Bracha, & Tanimoto, 2016) cite that a hazard in imbuing their highly dynamic system with liveness is the task of updating (or if impossible, destroying) any stack frames which are in the process of executing through an updated class (potentia I element). This highlights that the program stack is one of the most prominent divergent elements of most current language designs, and one whose impact has to be strongly restricted. An ideally non-divergent system would eliminate the program stack entirely as a runtime phenomenon — a tall challenge for a system which requires to remain intelligible and efficient.

4.2. Divergence from Object Representation
Dynamic and live programming languages typically expose facilities where the interfaces of objects can be customised at an instance level. Whilst Smalltalk (Black et al, 2011) frowns on this somewhat (methods are defined on classes, not on instances — however, method lookup can be dynamically intercepted), Self (Ungar, Chambers, Chang, & Hoelzle, 1991) provides primitives for dynamically adding and deleting slots from objects. After this has been done, the in-memory representation of the object has diverged from the one resulting from its hierarchy. Were the object destroyed and reconstructed, the authorial information would be lost. Both Self and Smalltalk provide a scheme for serializing and restoring objects and prototypes into persistence, known as the “object transporter”. The files written by the object transporter do not qualify as documents in our taxonomy, since they have no natural coordinate system and cannot easily be manipulated by tools other than the virtual machine which produced them. The Pharo (Pharo, 2016) and Squeak Smalltalk (Squeak Project, 2016) projects cooperate by providing access to a DVCS system named Monticello (Bryant & Putney, 2016), which system is capable of interpreting the output of the object transporter on those particular Smalltalk dialects. The fact that this is a platform-specific facility rather than a generalised one highlights the fact that the virtual machine contents have not been externalized in the sense we have described, and that general text- or document-based version control systems such as git are not easily applicable. A Smalltalk or Self program is essentially a hermetic machine image — mixing the functions of an extensa (in that it is running) and a potentia (in that it is authorable).

4.3. Divergence from Event Listeners
Essentially any programmatic (as opposed to declarative) authorial action targeted at the extensa may result in divergence, but we’ll round off this list of examples with a final prominent case — that of registration of event listeners. Traditionally, events represent a multicast publish/subscribe pattern which has some kind of incarnation in essentially every programming system, especially those which present some form of user interface. Event listeners represent a primary source of divergence because they are traditionally very hard to coordinatise. Programmatic systems tend to treat listener equality or identity based on the equality of function (or slot) handles which naturally leads to problems once part of a system enters serialisation outside a running VM. This issue can often lead to design puzzles in itself — for example, if a particular method or slot is added multiple times as a listener to an event, should it be notified multiple times? If it is then removed, should just one instance of the listener be removed, or should it be all matching ones?

4.4. Strategies for Minimising Divergence
No unified strategy for opposing divergence in all its forms can be mounted — since these forms are highly diverse, and do not stem from a single cause; instead, they each result from particular engineering tradeoffs, physical and cognitive ergonomics. Considering our sources of divergence above, we could consider, for example, the following kinds of strategies for reducing their design impact:
4.4.1. Fighting Divergence from the Stack

To reduce the impact of the stack as a source of divergence (that is, its tendency to interfere with our ability to author a running system as if it consisted of pure, externalized state), we could consider strategies like these:

Firstly, we could prohibit trying to author the system whilst the stack is not quiescent. This might for some applications, represent a significant loss of liveness (pushing us down from L4 to L3 or lower in (Tanimoto, 2013)’s taxonomy). However, the strict definition of L4 liveness has problems with respect to a fully collaborative system, which may be in progress with edits from multiple users concurrently. If we make the design choice (an expensive one) to fully isolate these transactionally, we may be able to finesse L4 liveness by claiming that the system “does not visibly have an execution in progress” if it is observed from outside the context of the relevant user’s transaction.

Secondly, we may adopt execution strategies which tend to make for short stacks, or, more radically, abolish the use of call stack entirely. There is some precedent for this in ancient and modern asynchronous or message-passing architectures. For example, highly distributed systems such as Erlang partition work out to many thousands of “processes” (execution sites which may or may not correspond to distinct physical execution sites). As a result, Erlang programs tend to have much shorter, “broader” stacks. There is a clear relationship with our first stack elimination strategy if we imagine that a “process” takes the form of a forked avatar of a substantial part of the state in the system (during a transaction), which is capable of being joined back to it if the transaction commits successfully. Other connections are with the “spatialised computation” of (Kulkarni et al., 2008)’s “Galois” framework, and with the rising tendency of JavaScript (and other) developers to implement architectures using Promises. Promises by convention (though not by necessity) resolve asynchronously, cutting the stack at the point they evaluate.

The costs of these strategies are to dilute the important virtues which the machine stack was designed long ago to embody. Firstly, the stack as an “explanation” for the system’s current actions is a truly vital affordance when authoring — for example when debugging, or in the case of an error. Naturally tools and libraries could compensate for this loss, but they would have to exist, be ubiquitous and cheap, and of high quality — and it would be hard for them to be as ubiquitous and cheap as something which is forcibly embodied in the machine’s own execution model. Secondly, the stack is a vital target for important classes of optimisations. A function call could never be inlined if it never occurred in the first place.

4.4.2. Fighting Divergence from Object Representation

It is possible that Smalltalk’s discouragement of instance-based object modification stems from a taste against divergence. It certainly points the way towards a family of strategies for dealing with this problem — since the problem, after all, stems from the possibility that an entity with a “less utterable name” (an instance) diverges from one with a “more utterable name” (a class/prototype). If the design of the system forces the user to give a more utterable name to a unit as part of the authorial action of modifying it (or its class), divergence is headed off — since the crucial affordance required by document semantics is that the document is transparently coordinatised (section 2). In our taxonomy of state from section 3, forcing the authorial action to be ascribed to some named category of entity helps the system push responsibility for the edit from extensa up to potentia I.

Another strategy which assists here is to segregate the state held at a unit (object/component) into material which explicitly is considered “model state” (which is mutable, but has edits ascribed to potentia II), and the remainder, which is considered immutable, to which edits must be ascribed to potentia I. This increases the costs of understanding the system’s design, given it represents a lack of orthogonality in its affordances, but resolves the problem in authorial meaning. This is the strategy currently adopted by the Infusion system.
4.4.3. Fighting Divergence from Event Listeners
Event listener divergence can only effectively be combated by removing the affordance to add or remove event listeners as a direct authorial action. As with other cases of divergence, we need to produce idioms for pushing the association of events with their targets up into potentia I. This implies that lifetime and reference for listeners are both managed by the system, in response to some form of declarative dialect encoding the relative positions and identities of the target and source. We can generalise this category of problem to the problem of maintaining any kind of binary relationship between pairs of objects in the system. The natural choice is to bring the relationship into existence at the time point where the later of the two objects has been constructed, and tear it down when the first of them is destroyed.

4.4.4. Queen of Sheba Adaptation
In Chapter 16 of his *Fusus al-Hikam* (The Bezels of Wisdom, written c. 1229) (Al-Arabi, c. 1229), the Sufi philosopher Ibn al-Arabi describes an encounter between Solomon and the Queen of Sheba (Bilqis) in which he causes her throne to appear before her, apparently displaced by thousands of miles in an instant. She comments on seeing it, “It is as if it were the same”. al-Arabi analyses this incident from the point of view of doctrine of the “renewal of creation by similarity”. Under this model, the contents of the universe may be being destroyed and renewed countless times each second through the will of Allah. This process is not evident to us because we, the observers, are destroyed and renewed along with it.

This suggests a simple test as to whether a system is divergence-free, as well as a cheaper means for systems to undergo adaptation in the face of authorial instructions than to implement full L4 liveness in the sense of Tanimoto. If we can destroy a component of the system and then regenerate it from its potentia I and potentia II records, in either an identical condition, or a changed one in order to reflect potentia II adaptations, the system must be divergence-free. If, in addition, we can arrange the system such that no observer could perceive the system in the intervening time (which capability is mostly implied by our treatment of transactions in section 4.4.1), we could say that the system (inefficiently) is both fully live and divergence-free. This scheme might involve less implementation effort than a more ambitious system that could compute the minimal differences between the two configurations of the *extensa* before and after the change, even if the latter would execute much more efficiently. This treatment suggests that the Tanimoto taxonomy has an anomaly at the point where the L4 liveness level is defined — in that we could easily imagine systems delivering L5 and L6 predictive capabilities even where the strict letter of L4 had not been observed. One such system is the Entelechy system described in (Church et al., 2016).

4.4.5. Tools or System
We will visit these kinds of considerations further when we look at particular examples of systems in later sections. However, we will note that they place a huge burden on the development of powerful and highly usable tools in order to compensate for the affordances which the divergence sources were originally designed to meet. In practice, appeals to the existence of such tools are far more frequent than the tools actually coming into being. However, it’s clear that pushing forward the standard model of live programming, where “the tools are part of the structure of the system itself”, has to be the model for resolving these problems — whilst, at the same time, avoiding the standard trap of live programming, whereby the running system itself becomes a hermetic, self-sufficient ecosystem with little reference to things which lie outside it. These “tools”, when we build them, must be every bit as externalizable and divergence-free as the “system” that they arguably operate on.

4.4.6. Homeostasis rather than Execution
In order to enable a harmonious authorial cycle, we need to create programming systems in which the extensa, potentia I & II are cleanly separated. In particular, separating potentia I and potentia II creates a very different idiom for the function of the running system — rather than being given the task of *executing* the user’s expressions (properly supplied as potentia II elements), the system instead is engaged in a process of *homeostasis* — that is, constantly choosing actions at its disposal in order to minimise what discrepancy there is between the extensa and the potentia from time to time. This model
of homeostasis also allows for *predictive homeostasis* — speculatively anticipating authorial expressions in order to achieve L5 and L6 liveness in Tanimoto’s terms.

### 4.4.7. Mocks and a Taxonomy of Effects

A crucial requirement for the speculative execution of an L5/L6 live system is a clear schema identifying which implementation units have side-effects and which do not. Effects issued against the external world cannot be called back — effects issued against our own implementation might possibly be. Ideally, as well as being able to read metadata identifying an effect-laden component, the running system would also be able to locate a variant component that had some value of substitutable behaviour that was free of the side-effects. In the testing community, these implementations are known as “mocks”. In traditional software development, mocks and test cases are considered part of a separate arena of expression and activity. Here we close the loop on another of the “elite affordances” that we identify in our introduction, unit and integration testing. A fully competent authorial system would make no sharp distinction between test cases and real execution, between mock implementations and real ones — since a speculative mock implementation executing at the behest of the future should be equivalent to a real implementation executing today.

The test cases for each artefact would be part and parcel of its packaging in the system, together with metadata allowing the system to identify one or more mock implementations that could be substituted for it in cases where its side-effects need to be limited as a result of speculative execution.

### 5. Webstrates

Webstrates (Klokmose, Eagan, Baader, Mackay, & Beaudouin-Lafon, 2015) is a Web framework designed to explore a software vision based on Alan Kay’s early conceptualizations of interaction with a computer being interaction with a dynamic medium (Kay & Goldberg, 1977). Webstrates persists all changes to the Document Object Model (DOM) of any page, called a webstrate, served from the Webstrates server, and synchronizes these changes between clients of the same page. This includes changes made to inline JavaScript and Cascading Style Sheets (CSS). In effect Webstrates turns the DOM into a collaborative and malleable medium. Development happens inside the browser either through the developer tools of the browser, or through authoring webstrates that transclude other webstrates through the use of *iframes*. Klokmose et al (Klokmose et al., 2015) demonstrate how transclusion of webstrates can be used to create dynamic application-to-document relationships between webstrates, and allow users to collaborate on the same documents with personalized and radically different user interfaces.

#### 5.1. Externalizability

Development with Webstrates follows the dogma that all important application state is be stored in the DOM. If this dogma is followed, application state will persist over a page reload and will synchronise between clients with the same webstrate open. The DOM can be serialised to HTML, this means that an application running on Webstrates can be serialised, deserialised and resumed in the same state (in principle, see section 5.3). It also means that the application state can be modified in its “dead” serialized from, and be the subject of some elite affordances like version management.

#### 5.2. Actuals and Possibles

Webstrates uses the conventional JavaScript language and runtime, and the relationship between the actuals and possibles is therefore mostly conventional. Yet, part of the res extensa is expressed in the DOM, and is therefore inspectable, modifiable and serializable, and in Webstrates, collaboratively editable. The dynamic behavior of a webstrate is inlined as JavaScript stored in `script` nodes. Inlined JavaScript can be (collaboratively) edited at run-time, but must be re-evaluated by the client (either through reloading the page or using `eval`).

#### 5.3. Divergence

Webstrates on the one hand exhibits a low degree of divergence, as most application state is represented in the DOM that can be losslessly serialized to human-readable and human-editable HTML. However, on the other hand the JavaScript runtime (objects, prototypes, functions, event handlers etc.) is not
present in the DOM, but is instantiated from JavaScript that typically is stored in script nodes. Run-time changes to e.g. a prototype object will not be serialized back into the JavaScript code it was instantiated from in the first place. This means that the run-time state of two clients of the same webstrate can differ, which can be exploited to create collaborative applications with relaxed WYSIWIS³.

6. Dynamo

Dynamo (www.dynamobim.org) is a programming ecosystem for computational design, it is mostly used within computational architecture. It is a hybrid visual/textual and dataflow/imperative language using a conventional node and arc notation with dataflow semantics; it can also be seamlessly converted into DesignScript, a textual language for exploratory programming. Dynamo’s primary uses are for exploring geometric forms and for automating Building Information Management (BIM) workflows, such as computing furniture needs for a building. As such it has a very strong connection to Autodesk Revit. Dynamo is an L3/L4 live system, with the level of liveness controllable by a checkbox in the interface. The default behaviour is that any changes to the program will execute as soon as they can. The system will re-execute automatically in response to slider changes — and more ambitiously — to changes in elements that are used within Revit.

6.1. Externalizability

State is messy in Dynamo. The computational graph carries state that is typically associated with the execution of a program. However, many of the variables in the graph are really pointers to objects in the “world” of Revit. This means that the state is shared between the Dynamo virtual machine and the Revit document model. Approximately the intent is that the program is represented within the Dynamo document, the state in the Revit document, and the Dynamo “document” also stores pointers as to how to link the two together. This aligns to the design agenda that the Dynamo project embodies, that it is a mediator of distributed “cognition” amongst computational modes of representation (Dynamo), architectural ones (Revit) and the unknowable representation in the users. However as this structure is reified into software architecture, the boundaries get messy.

6.2. Actuals and Possibles

The separation of the software architecture along the lines describes above, whilst making accounting for state complicated, fairly naturally enforces the separation between Res Potentia I and Res Potentia II. The definition of what is possible is in two places: the core data types in the virtual machine (numbers, strings, maps), and the more complicated object types that contain things useful to architects (points, lines, planes, doors, walls, chairs). These are defined on the target side of a Foreign Function Interface boundary. In Dynamo’s case this is a C# wrapper over the Revit API or a geometry kernel. These makes these definitions technically and socially separated from the nodes that use them. This strategy has been sufficiently successful that the project has progressively moved towards eliminating the possibility of defining objects on the Dynamo VM side. Res Potentia II, using the objects, on the other hand, is all on the Dynamo VM side, with the construction calls corresponding roughly to nodes in the graph.

The Extenisa of a system employing this distributed style of programming where the state is shared across many technical boundaries is much more challenging to design. The Extenisa represents a totality of the state across the ecosystem, including the representation of objects that are only partially computationally accessible (not everything in Revit has an API that Dynamo can use). This represents a substantial ongoing challenge in programming language design: how to design systems that do not expect to have a totalist perspective over the worlds they manipulate, or even over the potentia for those worlds.

6.3. Divergence

With a system that embodies this distributed Potentia and lack of total knowledge, divergence ceases to be only an inconvenience and becomes a primary usability problem. In Dynamo we address this in a couple of ways. The virtual machine employs an indirection layer, names are bound to transitory “shells”: these can dynamically change which objects they actually point to. These shells are computationally

³What You See Is What I See
efficient, but also allow a decoupling of the virtual machine’s view of identity.

This allows us to employ a Queen of Sheba identity scheme. On each partial execution, we generate a “trace” of new shells associated with the call sites of the program. We then model the structural correspondence in the layout of these shells compared to the previous trace. Heuristics such as comparing the cardinality of the arrays are used to determine if they structurally match. If they do, the shells adopt the identity of the previous objects. If the correspondence can’t be matched, new objects are created for the shells, and any that don’t have remaining shells are garbage collected. This strategy has been reasonably successful at managing divergence in a way that isn’t too surprising to users using the system — however it remains to seen how successful it will be as the computational complexity of the graphs being manipulated grows.

7. Infusion

Infusion currently takes the form of a configuration dialect expressed in terms of JSON structures, organising the activity of short, publicly named, mostly side-effect-free functions written in JavaScript. Rather than considering Infusion specifically as either a language or a system (see (Gabriel, 2012) for the interesting history of this distinction) we prefer to consider it as an integration domain, following the thought of (Kell, 2009).

Much of the design of Infusion has been motivated by the minimisation of divergence, although this term had not been coined for most of its history. Infusion draws up a taxonomy of application function, for example events and their listeners, model material and relationships between it, that allows this function to be authored in terms of documents (following the sense of section 2).

The Nexus (The GPII Team, 2016) is an experimental system which fully externalises the affordances of a running Infusion system in terms of simple JSON payloads exchanged over HTTP and WebSockets protocols. To the extent that the Nexus is functional, it represents a system in which does indeed fully separate the three areas of expression — however, it has many implementation gaps that need to be filled before it is ready for purposes beyond simple technology demonstrations.

7.1. Potentia I & II in Infusion

Infusion does not yet clearly allow the user to separate potentia II expressions from their extensa, but this feature is close enough to being real that we can helpfully illustrate what this distinction is in terms of Infusion features.

Listing 1 shows a traditional programming-language use of Infusion to construct an object, which illustrates several problems. In this snippet, both the intention (to create an instance of type fluid.thing) as well as the result (the instance itself) are private: the fact that fluid.thing has been requested is knowable only to the runtime at that moment (or else some highly sophisticated tool capable of parsing arbitrary JavaScript and matching it against a complex schema of possible function effects), and the result is stored in an inaccessible area — a function scope, from which it is not accessible by any code elsewhere in the system, and is not even accessible by traditional debugging tools unless the stack frame for the function happens to be activated at that moment.

A somewhat less divergent incarnation of this expression is shown in listing 2. The first argument to fluid.construct holds the global address of the component to be constructed, expressed as a set
of path segments\textsuperscript{4}. The second argument holds the description of the component, expressed in an easily serialisable JSON-equivalent form. This expression could be treated as simply a specification of the potentia II material designating the instantiation of the component, and form a separate workflow to actually initiating the instantiation itself. Unfortunately in the current framework, this function call is treated as a directive to synchronously instantiate the component.

To complete the externalization picture, listing 3 shows the same expression as issued from outside the system via the HTTP Nexus API. Here we are tantalisingly close to having the potentia II expression separable from the executing system itself. All we would need to do is keep a record of such requests incoming into the system in a readily indexable form, or perhaps place an HTTP proxy layer in front of the system which achieve this. Such things have so far not been done.

7.2. Stack divergence in Infusion
Infusion currently makes no efforts to fight divergence from the stack. Since our tooling is quite immature, we rely heavily on standard debugging tools in which the presence of a long and detailed stack trace is essential in order to understand the action of the system. An upcoming rewrite will improve the system’s treatment of asynchronous resolution, which will result in loss of quality in the authorial experience described in section 4.1 unless it can somehow be compensated by development of powerful tools to help visualise the trajectory of the system.

7.3. Reducing listener divergence in Infusion
Infusion successfully eliminates divergence resulting from listener definition and registration. Listeners definitions are attached to component definitions, and have their lifetime automatically scoped to the lifetime of the component holding the definition (which may be different to the component to which the listener is attached). Since each definition is assigned a unique address within the system’s space of definitions, the identity of the listener is stable (with respect to the potentia), despite being honoured by different function handles in the running system from time to time (the extensa). In addition, listener definitions can be manually assigned a namespace which enables them to be targeted by further authorial expressions — that is, a particular listener definition can be overridden by supplying its namespace when supplying a further definition which already matches both in target component and event name. Listing 4 shows a simple setup with three components, A, B and C where construction of B will register a listener attaching one of C’s methods to one of A’s events. Destroying B will tear down this listener, and reconstructing B would regenerate it, in accordance with the “Queen of Sheba” dynamics described in section 4.4.4. Since at present Infusion has no transaction system governing overall component instantiation (it has transactions governing only the model skeleton attached to components), this destruction and recreation would be sadly evident to all.

\textsuperscript{4}Note that this externalization of component addresses raises its own hazards. The privacy/opacity of traditional addresses served a useful authorial purpose — it was impossible to “accidentally” (or indeed at all) overwrite one author’s component with another. The shared inhabitation of a global address space requires careful management of a hierarchical space of names — just as in similar shared spaces as the DOM. This is to be done via conventions allocating well-known component root addresses to different authors — by schemes similar to those used to avoid collisions in package names in global namespaces such as the Java package system.
fluid.defaults("examples.ppigListeners", {
  gradeNames: "fluid.component",
  components: {
    A: {
      type: "fluid.component",
      options: {
        events: {
          fireIt: null
        }
      }
    },
    B: {
      type: "fluid.component",
      options: {
        listeners: {
          "{ppigListeners}.A.events.fireIt": "{ppigListeners}.C.hearIt"
        }
      }
    },
    C: {
      type: "fluid.component",
      options: {
        invokers: {
          hearIt: "fluid.log"
        }
      }
    }
  }
});

// Sample usage:
// Construct overall tree
var that = examples.ppigListeners();
// causes logged message "I send it" via C’s method mediated by B’s binding
that.A.fireIt.fire("I send it");
// destroy B, thus tearing down its binding
that.B.destroy();
// No effect since event binding has been removed
that.A.fireIt.fire("I send it again");
// Note that B cannot in this design be easily recreated since we do not have separated potentia II records

Listing 4 – Declarative registration of divergence-free listeners in Infusion

8. Conclusion
We have critiqued the values of live programming, showing that many of their aims can be met without necessarily adhering to the letter of liveness, and that they can be situated amongst wider considerations of authorship for software systems in general, supporting cycles of authorship and use by networks of authors working in different times and places. We have introduced a quantity named divergence, a measure of the discrepancy between the internal bookkeeping state required to maintain a running, live system, and the state by which it can be authored by means of externalizable documents. We have linked the authorial cycles that we wish to support both to the system of Potentials and Actuals envisioned by Whitehead in his Process and Reality, as well as to the system of maintenance of the universe described by Ibn al-Arabi in his The Bezels of Wisdom.

We have considered three real systems (Webstrates, Dynamo and Infusion) from this point of view, and shown how much work still remains before a substantial fraction of the aims of open authorship can be achieved.

9. References


Building Software is Not[^1] a Craft

Antranig Basman
amb26@ponder.org.uk

Abstract
Software construction, yet to establish its role in the world, inhabits an invidious position between mathematics, engineering and craft — losing out by the comparison to each. I draw a distinction between different levels at which the term ‘craft’ could be applied and list four key virtues desirable for crafted items at a higher level. The current products of computer scientists or software engineers are deficient in these four virtues, but not necessarily so. I propose a shift in the value system underlying our work — recognising that the function of material is to bring communities into contact. I will argue that software is material to the extent it promotes an ecology of artefacts which cooperate harmoniously in the lives of their owners. Rather than the workmanship of certainty, we should promote the workmanship of risk, turn from correctness to truthfulness, and consider time horizons of generations rather than at most a couple of years.

1. Introduction
This paper will take a different course in assessing the nature of software as craft, materials and media than is traditional in academia. A common setup would have us note some grounds for analogy — for example, that constructing software requires some skill, is error-prone, and requires labour — just like a craft. We would then muster the resources of critical theory and subtle argument in order to extend this analogy, all the while begging the question of whether we are dealing with a craft at all. As (Wittgenstein, 1953, §414) has it,

You think that after all you must be weaving a piece of cloth: because you are sitting at a loom — even if it is empty — and going through the motions of weaving.

This kind of error is rife — for example, (McCullough, 1996, p. 119), a seminal work on the craft status of software construction, argues

But to say that computing is limited because it consists of no more than these few tasks is like saying painting is no good because all you can do is apply color to canvas.

Instead, I will consider the software that we actually have — and compare it, in terms of the value that its artefacts offer to society at large, to the highest craft practices that humanity has achieved. I will argue that a change in the values that we bring to our work can bring about better roles for software artefacts in our lives. Craft artefacts are produced within a cultural context in which they can seem natural, and can be holistically integrated into a life or a household. They are connected to human tactility, usefulness, ergonomics, beauty and durability — all virtues that I will consider as defining true craft artefacts. These artefacts take part in an ecology of similar artefacts in similar roles, as well as other cooperating artefacts in the lives of their owners. I will propose a system of values in software construction that promote the authorial stories which structure these ecologies — how they pass from hand to hand and take part in networks of creative accumulation and reuse. This in turn feeds back with implications on what we understand by the nature and properties of the “material” which software should be made of.

2. Craft Status of Software Artefacts
The craft status of software development remains limited, in the crucial senses which mark the phenomenon and results of craft as highly desirable to society. In a looser sense, software development

[^1]: Yet
shares some of the aspects of a craft — in the same way that a DIY enthusiast, journeying to their local hardware store in search of cheap, highly standardised components with poor manufacturing values can be said to be indulging in a craft as they seek to make some more or less successful home improvements. But these aren’t the values we should appeal to as we position our “craft” amongst its more established cousins.

3. Virtues of Craft
As William Morris remarked as he began to establish the English Arts and Crafts movement in 1880, “Have nothing in your houses that you do not know to be useful, or believe to be beautiful” (Morris, 1880). A true craft produces artefacts that are

**Beautiful** - giving pleasure in use and contemplation

**Durable** - long-lived, stable, and if degrading, degrading gracefully

**Ownable** - can be the possession of their owner, living amongst them in their household

**Handleable** - fulfil a function which sight and touch fit into an ecology of similar artefacts

In contrast, the outputs of computer science that we see in practice are ugly, brittle, masterful and obtuse. Let us consider our virtues in turn.

3.1. Beauty
There is an aesthetics in computer science, but it is not for the user and rarely even for the developer. The model for beauty with any currency is imported wholesale from mathematics — an algorithm can be elegant, a proof beautiful. In contrast, any actual piece of software, especially one which performs a useful function, is without fail fantastically ugly, reflecting the haphazard and throwaway values of real software culture. As Simon Peyton-Jones remarked at the PPIG keynote of 2015, of open source - “It’s where people go to contribute, but it’s not beautiful”. A pivotal early essay on the craft of programming, (Dijkstra, 1977) gives a fundamental role to what is called “the beauty of a program”, associating it with such inhuman virtues as the brevity of a proof of its correctness. However, (DeMillo, Lipton, & Perlis, 1979) have recognised that “Formal verification of programs, no matter how obtained, will not play the same role in the development of computer science and software engineering as proofs do in mathematics.” A piece of software which possesses desirable, useful beauty will exhibit this beauty as it is worked on throughout its long life. I have seen no sizable or long-lived program that exhibits such a thing.\(^1\)

3.2. Durability
The lack of beauty in actual software is intimately tied up in the lack of possibility for any kind of durable expression — and still less, of any durable function. It is well-known (Programmers’ Stack Exchange, 2014) that software will spontaneously “rot” as a result of a progressive violation of its environmental assumptions. Sadly, it does not rot gracefully in the manner of a forest log, but disgracefully in the form of simply failing to install or run correctly or indeed at all. The only hope one has for running an old piece of software is to create a complete replica of its original home (a “virtual machine”) correct in every detail, and keep it captive there. If software were a lifeform, it would be an impossibly brittle and ill-adapted one that would mark in every case an evolutionary dead-end.

Unfortunately, our field does not so far have the courage to face up to this problem. Indeed, one of its leading lights, Alan Perlis, in (Abelson & Sussman, 1985) speculated whether software might not inherently lack durability, comparing it to a “soap bubble”. I can’t infer from Perlis’ remark whether he believed software should lack durability or simply that we lack the understanding to make it so. I suspect the latter — and if most software didn’t possess the beauty of a junkpile or shantytown rather than a soap bubble, we could be more sympathetic to its lack of durability.

\(^1\)With the possible exception of Donald Knuth’s \TeX \text{\textregistered} typesetting system, which being almost exclusively the product of a single mind, has not entered a traditional software maintenance process.
3.3. Ownability
Because it is not durable, but also through further causes, software can never effectively be owned by its user. Software is less ownable now than it has ever been — with the prevalence of “leasing” models for software, and increasingly aggressively pushed “updates” which will sometimes interrupt vital human activities in order to deprive them temporarily or permanently of the use of the artefact that they just believed was theirs (Hoffer, 2016). This is not the behaviour of a tool or a product, but of a tin-pot aristocrat transplanted into the user’s home.

3.4. Handleability
Software is legendarily opaque in form and function. It is well-attested that the vast majority of users are unaware of or unable to exploit the majority of configuration or function which is in theory available to them (Spool, 2011). The output of a craftsman, whether it takes the form of a spoon, bowl, basket or chair, even if it is of an unfamiliar form, typically appeals to an immediately obvious use value. Part of the problem could be ascribed to the novelty of software function, but users have little incentive to invest effort in becoming familiar with their software, because of our previous point — they have no control over its form or any ability to resist change. At any moment the familiar may once again be swept away by the unfamiliar.

4. Our Lack of Materials
Some of the discourse around software “craft” centres on the property of its “material”. We argue that the core of the deficiencies of software can be ascribed to the lack of anything which properly qualifies as its material. A true material is the result of a natural process, accumulated and shaped for years if not millennia. When one applies craft to a natural material, one is engaging with the natural accident of this process when meeting difficulties of resistance. By contrast, when meeting resistance in crafting software, one is primarily tangling with the obtuseness of one’s colleagues and their insistence in retaining authorial power for themselves.

Before software could become a craft, we need to construct its material. The durability of a true material, as well as enjoying the sheer ability of continued survival, also has the aspect of the ability to participate in creative networks. A product made of a material can be passed from hand to hand, at each point being seen in a different role or being the starting point of a fresh creative cycle. We’ll return to the nature of our materials in section 9.

5. Workmanship Leading to Craftsmanship
(Pye, 1968) makes a crucial distinction between grades of workmanship — the “workmanship of risk”, where the outcome is at risk at every stage, and under the direct control of the worker, and the “workmanship of certainty” where variability has been eliminated through a industrial process. Pye is clear that only the former kind of workmanship can lead to true craft, whereas computer science exclusively values the latter — in its use of proofs, abstractions, implementation insulation, and other techniques employed to take power away from the user, by erasing variability and control from production. Pye is also clear to value the contributions of engineers, who alone are responsible for defining the nature of materials which lead to real results, as opposed to theoreticians and scientists who accept such materials as a fait accompli without assigning or recognising credit. (Gabriel, 2012) is also clear to point out that it is most often the case that principled engineering work long precedes formalisation.

(Lakatos, 1976) on certainty:

“Certainty” is far from being a sign of success, it is only a symptom of lack of imagination, of conceptual poverty. It produces smug satisfaction and prevents the growth of knowledge.

This leads the way to characterising what constitutes the “risk” in our discipline. (McCullough, 1996, p. 212), himself following Pye, asks

…must a true medium entail sufficient risk and irreversibility to demand the rigor and
devotion that have always been necessary for great works? Can a computer with its undo and save as functions ever demand sufficient concentration on our part to enable serious, expressive works to come forth?

Two major routes by which we could or do face risk in the presence of “undo” are discussed in the following sections.

5.1. Risk through Authority
The first lies in the authoritarian structure of our current authoring idioms. In practice, only a very limited number of creative choices are delegated to the author dignified by the reductive term “end user” — and only these can effectively be undone. The greater part of the choices are reserved to the designers and programmers of the tower of language, platform, application beneath the user’s feet — and these decisions, from the user’s point of view are irrevocable and can never be “undone”. In practice, the discourse around design typically stresses the removal of affordances from the user that the designer considers inappropriate or violate design guidelines from their point of view.

An interesting example of this kind of “dialogue against design risk” emerged during the workshop conversations supporting (Computing Community Consortium, 2015). One participant warned that user interfaces should not be made too configurable, otherwise the helpdesk tasked with supporting the user with their software might be rendered incapable of helping them — because the user had reconfigured their interface so radically that the helpdesk staff couldn’t recognise it.

This is a form of “positive risk” that we are not exposed enough to. We should try to promote authorial systems and design communities which allow owners of artefacts to run more such creative risks, whilst minimising their costs.

5.2. Risk through Unmanageable Complexity
Another route to exposure to risk in the presence of “undo” lies in the potential for the complexity of artefacts to exceed our ability to manage or comprehend them. This is a risk for amateurs and professionals alike — if we can no longer comprehend the authorial affordances that we have, and reliably distinguish the virtues of one version of an artefact from another, it matters little whether all versions of the artefact that have ever been authored have been perfectly stored and can be instantly retrieved. (Eick, Graves, Karr, Marron, & Mockus, 2001) report “anecdotal evidence of systems that have reached a state from which further change is not possible”.

This can be described as a form of “negative risk” that we are all already exposed to — our challenge is to structure languages and frameworks to minimise the chances of artefacts being “lost” to this risk.

6. Lutyens’ Latch and Barn’s Spoon
As an example of the kind of craft I am appealing to, I firstly exhibit the example of this door latch (Figure 1) designed by Edwin Lutyens, a prime proponent of the Arts and Crafts movement. Latches like these can be found throughout his restoration of Lindisfarn Castle which he undertook starting in 1902, as well as some of his other designs. It self-evidently enjoys all the virtues we have listed — it is highly beautiful (even achieving a kind of wittiness²), and its method of construction, based on readily available and widely understood materials, is a direct guide to its function. In the unlikely event that it requires maintenance (having endured intact for over a century), this can be achieved through widespread means.

Next, as a contemporary example, I exhibit a ‘cawl’ spoon (Figure 2) carved from rippled sycamore by a modern representative of the Arts and Crafts tradition, Barn the Spoon, at his shop in Hackney Road, London. Again it enjoys all of our virtues as well as participating in the vernacular both at the level of its design and its construction — this spoon was carved in the presence of its buyer at Barn’s shop, creating a direct link between the worlds of authors and users.

²Note that this wit is only interpretable with respect to the ecology of function represented by latches in general.
A lack of durable and natural materials, as well as a suitable craft culture, represent the primary deficits in computer science which prevent it from constructing crafted items such as these.

**Figure 1** – Door latch designed by Edwin Lutyens

**Figure 2** – Spoon designed by Barn the Spoon

### 7. Spectrum of Forgiveness

Whilst the software that we have is constituted of a material far more brittle and less malleable than the least forgiving of the human crafts (for example, ceramics), being a product of the mind, it has the potential to be far superior to the most forgiving (for example, knitted textiles). The following table illustrates the affordances that we can expect from different materials.

**Ceramics**
- Once the product’s form is fixed, it cannot be further altered in a graceful way.
- It is brittle and once damaged cannot be gracefuly repaired (a glued repair is always obvious, and a damaged surface can never be reformed like the original)

**Wood**
- With some effort, an item can be reformed or repaired, for example by sanding, varnishing, repolishing etc.
- For some items, a broken element can be removed (unglued, unscrewed) and replaced with a new part which functions and appears as well as the original

**Textiles**
- Most items can be repaired, sometimes indefinitely, by patching or darning
- In some cases the original raw materials can be fully recovered by unravelling the garment and a completely fresh one constructed

In contrast, we have

**Computer Science of the Present:**
- A product may spontaneously disintegrate without warning, suddenly becoming wholly unusable
- No modifications may be made by the owner after delivery of the software, even through amateurish affordances such as superglue
- The raw materials for software are endlessly distant from the owner, separated by a series of barriers operated by successive priesthoods (compilers, linkers, integrators, hosting services)

Whereas nothing inherent in Computer Science prevents it from being

**Computer Science of the Future:**
- Any product, either in form, function, or both, may be preserved indefinitely in a working condition
Any owner of software may modify and maintain it using only the affordances to hand — or else share or receive such modified versions from peers or communities of interest.

The raw materials for software are constantly immanent — any user of software, simply through an act of perception, may choose to see it either as raw materials, finished product, or anything in between.

From its current position as the world’s least forgiving craft, computer science could surpass all those currently known.

8. Forces Producing the Craft We Have

We have the craft that we do, because of the power structures and mentalities embodied in our communities. As (Blackwell, 2010) notes:

Even supposedly universal “general purpose” computer languages, although assumed to be culture-free, become imbued with metaphors of power and control that are easily recognized when inspected from outside the perspective of their designers and users.

Computer scientists, in theory charged with planning and directing the formalisms underlying the practices of software construction, have no appreciation of how software is actually made and on whose behalf. In turn, software engineers adopt methodologies that retain power for themselves at the expense of their colleagues and users, rather than appreciate that design and usability concerns should be the paramount ones governing their work. These usability concerns should be those which emerge when artefacts are deployed as part of their owners’ households and ecologies of living, rather than reflecting a process of “abstracting away the user” (Blackwell, Church, & Green, 2008).

8.1. The Term ‘Craft’ in Our Lineage

We can trace back the threads of this discourse as far back as we like. For example, (Dijkstra, 1977)’s original essay on software craft status insisted that software production should cease to be a craft, rather than become one - “a craft was applied, where a scientific discipline was needed”. How could Dijkstra have come to make such a profound category error? It is because he already inhabited a completely self-sufficient discipline, answerable only to its own aesthetic and formal standards. Dijkstra is already capable of looking at the only “users” he has known, professional physicists and engineers, with a kind of amused contempt for their tendency to “write a three-page program in an afternoon”. The thought that even less-well formally oriented citizens, the general public, could gain access to the affordances of crafting the software that they need, was unthinkable — these are people “the majority of whom it is totally unrealistic to expect that they can still acquire a scientific attitude”. Those who imagine that ordinary citizens could become thus empowered are condemned as “foolish”. These are the roots of the kind of disciplinary imperialism that within a few decades would bear fruit as the pernicious “computational thinking” movement of (Wing, 2008).

8.2. Mistaking the Past for the Future

But how could Dijkstra have come to believe that software production was a craft in the first place? The answer lies in the opening paragraphs of this paper. He had become confused by the simple fact that the software production process required special insight and skills as a result of its hopelessly unreliable tools and materials. In fact, in most cases the “craft” of his and our day doesn’t even aspire to the level of hardware store, flat-pack DIY furniture values that we described. It is as if we bought a kit of parts for a flat-pack cupboard, and found them hopelessly ill-fitting, some pieces missing and the instructions written in an unrecognisable language with no illustrations. This is certainly the experience even of a consumer of what should be the most standard, well-understood and fully reduced to practice products of computer science — parsers. Anyone who has proposed to design a new programming language and expected to pick up a parser technology for it “off the shelf” will respond to this point.

3This similarity of workflow and outlook is why I fail to particularly distinguish between computer scientists and software engineers in this essay.
Rather than, as Dijkstra claims, software production needing to cease to be a craft, it has not even got to the starting blocks where it could qualify as one. His appeals for a “scientific discipline” and a “scientific attitude” are wholly misplaced. As is well known, computer science is not a science (Abrahams, 2013). Its subject matter is not a domain governed by natural laws despite the belief of some leading lights of the field, e.g. (Hoare, 2014) but instead the communities of real human beings, their needs, cognition, and psychology. Properly understood, Dijkstra’s plea is simply one for reliability — but reliability does not signify the end of craft, but merely its beginning. Imagine a discipline of carpentry operated by chisels which might unpredictably bounce off perfectly ordinary-looking pieces of wood, or whose tips might spontaneously disintegrate. This is the world of tools and materials which computer science has made for itself. Acquiring reliable tools doesn’t mark the beginning of a science, or Dijkstra’s “production-line worker” — instead “Reliable tools are the beginnings of spiritual freedom”(Matthews, 2016).

9. What Could Materials Be?
What could the materials be like, which might be the medium for the craft we want?

9.1. Material and Immaterial — Supporting Authorial Stories
There’s potential confusion and paradox in this area, since from a traditional point of view, these materials are “immaterial” since they aren’t directly constituted as matter in the world that we can grasp with our hands. (Gross, Bardzell, & Bardzell, 2014) have drawn up three principal models for the materials of software — broadly speaking, these are i) the traditionally physical, as embodied in tangible user interfaces, ii) the metaphysical, embodied in the relationships between information, and iii) the traditions, relations and communication that artefacts enable between people. Of these, I am most closely in sympathy with the third model — material can be anything it needs to be, either material or immaterial, that enables the open, unbounded authorial stories that promote participation in society. Traditional materials support the growth of traditions — as artefacts are passed from hand to hand, they inspire others not only by the stories embodied in them but by the fresh stories and uses they may participate in. Today’s software materials support only “dead-end” authorial stories. To adapt an artefact built with today’s languages in a way other than its authors intended, one must first “fork” the source code of the artefact in order to modify it. Version control techniques only provide a paper trail by which this process could be audited, rather than a transparent means for communities downstream of the fork to remain in contact with the upstream.

How could we build software that allows us to “change things without changing them”? Just as with the apparent distinction between the ‘material’ and the ‘immaterial’, this paradox will lose its force through a change of perspective. The paradoxes of today will be the unexamined assumptions of tomorrow.

9.2. The Density of Materials
(McCullough, 1996, p. 196-197) cites a quality of materials that underpins their suitability for craft — their density. Each state of the material is closely surrounded by a dense collection of neighbouring states that behave similarly. As McCullough puts it, “Density supports engagement not only through continuity but also through variety”. However, today’s software materials, traditionally consisting of source code text in an editor buffer, could not be more sparse. The “nearest neighbouring program” to any given one is separated from it by a vast ocean of syntactically invalid or crashing variants.

A movement exploring such density at the hardware level is known as “circuit bending” (Ghazala, n.d.) — these groups of enthusiasts assemble components from low-powered electronic circuits into random configurations and expose them to random electronic signals (within a safe range). When applied to games consoles of the 1980s, for example (Belojevic, 2014), a surprising amount of interesting behaviour results — recognisable sprites or game elements will appear on screen and sometimes animate. This behaviour shows that the space of allowable system states is dense within the overall space of available states, and is inconceivable in a system of modern hardware or software. One approach to increasing the density of valid authorable states is simply to make them more findable via assistance
from powerful IDEs or code completion techniques. This can carry us a certain distance, but more funda-
mental approaches to rethinking the structure of software materials can make the space of these states
intrinsically dense and naturally navigable. I’ll refer briefly to some such approaches in the next section.

9.3. Some concrete approaches to our new materials
(Basman, Church, Klokmeose, & Clark, 2016) explores some concrete strategies for bringing true ma-
teriality to software, by expanding the repertoire of available moves for building authorial stories. A
traditional distinction between artefacts is between those which are ‘live’, that is, comprise currently
executing systems, and those which are “dead”, taking the form of textual documents or directly au-
thored files on disk. The central dogma of computer science, that live systems are only derived from
dead ones, and not vice versa, is one of the many inflexibilities that prevent communities trying to share
software artefacts from remaining in touch with each other. A side-effect of bridging the gap between
live and dead systems, by corresponding them more closely, is to increase the density of their authorable
states (with respect to invalid ones) — since these states are naturally coordinatised as a result of the
correspondence.

10. Resolution
Our field is making only tiny and halting steps towards craftsmanship. It is increasingly popular to show
relevance to craftspeople by an appeal to our ability to work in their spaces or with their materials (see
(Victor, 2014) or other “Maker” movements), but the only result will be to export our own aesthetics
of frustration and power-worship into already functional communities. The experience of (Blackwell,
Aaron, & Drury, 2014) exhibits the behaviour of craftspeople when provided with pieces of “techno-
junk” such as a USB keyboard which doesn’t work with a USB socket and other similar obstructions:

experiences were . . . “frustrating” — a word that was used consistently and repeatedly in
the artists’ reports
with the result that
busy artists are equally likely to respond to obstacles by shifting their effort to other projects
in which they are making rapid progress

Computer scientists show little recognition of the frustration that their products cause in practice, and
instead imagine that the world would be a better place if more people thought and worked like them —
witness the rise of “Computational Thinking” (Wing, 2008) as the vehicle for this imperialism. Rather
than the virtues we list above, the highest virtues of computer science are the tedious accountancy of
efficiency and correctness. These will create a civilisation of bureaucrats rather than craftspeople.
In some quarters, there are signs of a change of heart. (Blackwell et al., 2008) turn against the tide of
computational thinking towards a respect for the individual human, and (Blackwell & Collins, 2005)
admit the possibility that “an end-user programmer may well prefer to accept the results of an imperfect
execution” — an appeal to the “workmanship of risk”.
However, there are few concrete signs of the durable yet forgiving materials that must underlie true
crafts. There are some simple, house-clearing tasks that we should turn our resources to, before we are
prepared to achieve the higher virtues:
• To guarantee to every user that, once given an artefact, they can always choose to use it in its
current form, regardless of “improvements”, updates, or environmental changes
• For every piece of software to be worked on by means of itself, rather than through the use of
specialised tools only available to an elite at a different site, at a different point in its history
• Given a user’s preferred way of working and seeing, for any system that they start to work with, it
adapt to whatever extent it can to meet those preferences
The latter three of our virtues are something that can be worked on in our lifetimes — and if we succeed
in constructing better, more forgiving materials, our descendants might possibly use them to construct
something which is beautiful.
Acknowledgements
I am grateful to Colin Clark for insightful comments on previous drafts of this essay, and for many of the connections drawn within it.

11. References
API Usability at Scale

Andrew Macvean  
Google Inc  
amacvean@google.com

Luke Church  
Google Inc  
lukechurch@google.com

John Daughtry  
Google Inc  
daughtry@google.com

Craig Citro  
Google Inc  
craigcitro@google.com

Abstract
Designing and maintaining useful and usable APIs remains challenging. At Google, we manage hundreds of externally visible web APIs. Here, we report on our experiences and describe six on-going challenges: resource allocation, empirically-grounded guidelines, communicating issues, supporting API evolution over time, usable auth, and usable client libraries at scale.

Introduction
Application Program Interfaces (APIs) are the way in which programmers access functionality created by others. Every substantive program written in every language today makes extensive use of APIs. It’s no surprise that their usability often dominates the experience of programmers.

APIs have been used for decades to connect software modules on a single machine. With the rise of web services and microservice architectures, web APIs have become the primary user interface for many types of services being offered online. While the paper is not centered only on web APIs, we do focus on them (as opposed to standard libraries) a few times. When we do, we are generally referring to representational state transfer (REST) APIs over HTTP. However, the structure of the API (REST vs., say, Simple Object Access Protocol [SOAP]) and the communication protocol (e.g., HTTP vs. local connection) are orthogonal for the discussion.

Despite their importance, there is comparatively little work on understanding API usability. A special interest group was held at the 2009 ACM CHI conference to discuss this problem, the result of which was the founding of apiusability.org (Daughtry, Farooq, Myers, & Stylos, 2009). As of the time of writing, some 50 publications have been listed on this site. A recent paper by Myers and Stylos (2016) provides a summary of the existing work and outlines the various techniques in use for understanding and improving API usability.

In this experience report, we describe some of the major challenges that we have faced building usable APIs at Google and discuss some of our approaches to those challenges. We aim to show the kinds of problems that Google faces as a major API provider and to start a broader conversation as to how we might go about addressing these challenges.

1. Challenge 1: UX Resource Allocation
Traditional usability evaluation techniques can be applied to APIs in order to measure, understand, and improve usability (Clarke, 2004), (Stylos et al., 2008). One of the most cited techniques is running API usability lab studies. This is effective, but not scalable, as participants with professional software experience and researchers with a blend of human-computer interaction, domain and technical knowledge are limited resources (Farooq & Zirkler, 2010). Replacing lab studies with design reviews has been explored by both Microsoft (Farooq & Zirkler, 2010) and Google (Macvean, Maly, & Daughtry, 2016); our experience so far suggests this can augment lab studies, but not replace them.

We are beginning to explore having software engineers run their own studies for APIs, which Google has employed successfully in the past for other domains (Baki et al., 2013). This reduces the need to find qualified researchers but doesn’t address the time commitment or the burden of recruiting professional programmers as participants.

Stylos and Myers (2007) described heuristics for how to prioritize research of API design decisions;
namely, prioritizing for decisions that developers make often and that severely affect API use. We are using two sources to find the APIs which create the most friction for our users: (1) API exploration usage data, and (2) API Customer Satisfaction surveys.

We can identify invalid API requests by looking for responses with a 4XX status code (i.e. an error in the client request, as opposed to, say, an issue caused by the server of the API. For example, a 404 or 429 HTTP response code), and we can compute the percentage of these responses across APIs. The problem with this approach is that API log data doesn’t distinguish between calls that are placed when the developer is learning the API and calls that are run in production, making it difficult to distinguish between a hard-to-use API and existing buggy code. However, nearly all modern web APIs have a ‘try it out’ feature available online, such as the Google API Explorer\(^1\) or Swagger UI\(^2\). When we examine log data coming from such a tool, we know that a human is on the other side.

Our hypothesis is that by exposing the Explorer data as usability metrics to the producers of those APIs, we can shine a light on problems and elicit positive change. As a first step, we explored a set of metrics for a small set of APIs. We can, for example, look at a success funnel on a per API basis, see Table 1.

In isolation, these metrics are not actionable. While we want success rate to be higher, what constitutes a good success rate? As a next step, we took a look at the 26 most used APIs in Explorer in a one week period, see Figures 1, 2 and 3.

We now have actionable information. Each graph exposes APIs that are outliers and need significant attention, while also showing us which APIs have further room for improvement. In Figure 1, we see that there are three APIs for whom almost everyone experiences a 4XX error at least once. We now know which APIs to focus on improving and can set a realistic numeric goal based on the less error-prone APIs (e.g., 40%). When we look at Figure 2, we see that many of our APIs average two 4XX errors per session. If you make an error, then the response should include enough information for you to fix the mistake. So, we know that we should look at the APIs in the > 3 range and try to bring them down close to 2. Figure 3 shows us the percentage of sessions that experienced success, where success is defined as achieving a syntactically correct (2XX) request. Again, we see a few APIs that are underperforming and room for improvement on many others.

There are three downsides to these metrics, which must be remembered during interpretation. First,  

\[^1\]https://developers.google.com/apis-explorer/  
\[^2\]http://swagger.io/swagger-ui/
while we know errors are a signal for usability problems, we don’t know exactly what usability issue(s) led to the erroneous request. Second, the problems they are experiencing could be usability problems with Explorer itself (the tool) as opposed to API usability issues. Third, we know from other research we have conducted that not every use of Explorer has a goal of achieving a 2XX response; for example, you may be trying to intentionally replicate a bug, or exploring the error messages that occur when you include invalid parameters. In practice, these three problems do not detract from the utility of the method. As long as the metrics guide follow-up evaluation, human investigation can eliminate false-positives while also better understanding the specific usability problems.

While we don’t include representative data here, we can also change the granularity of our metrics to expose problems on a per-method basis as opposed to a per-API basis. This can expose potentially confusing methods within a single API as well as providing UX researchers and software engineers more specific targets for lab studies or design reviews.

Our next step is to broaden the scope of the data we include in the analysis to include all of the Google APIs exposed via Explorer and to expose the metrics to API producers. Having access to so many APIs will allow us to set benchmarks that the API producers can compare their API against (e.g., more than one standard deviation above or below the mean). We are also conducting exploratory analysis to establish whether we can reliably distinguish between the different types of usage we see in Explorer. Splitting off the data from those, for example, intentionally getting errors to replicate a bug, would help with the validity of the data, as well as opening up avenues for other interesting analysis.

In parallel to our API log-file analysis, we are also exploring the application of customer satisfaction surveys to our APIs. Using a simple Likert-scale question (Likert, 1932), which asks users to rate their overall satisfaction with an API on a 5-point scale between ‘Very dissatisfied’ and ‘Very satisfied’, we can begin to quantify user sentiment towards a particular API. By distributing surveys across our suite of APIs, we also begin to get a picture of how users perceive our services, and, as with the Explorer data, the ways in which one API may stack up against others in order to set benchmarks for improvement. Furthermore, this lightweight approach to surveying allows tracking against time, allowing us to, for example, gauge the impact of a new version of an API with the old.

Surveying API consumers is difficult, given the fact that much of their interaction with the API will
be spent within their IDE or text editor, writing code. We are exploring distribution of our surveys via an unobtrusive intercept survey method (Müller & Sedley, 2014), letting us target API consumers reading documentation. This approach may introduce a selection bias, as using the documentation is not a requirement for using an API, and users viewing the documentation may not have any experience with the API. None the less, when distributed equally and consistently across all of our APIs, we can uncover interesting trends.

Much like with our API Explorer data, comparing sentiment across APIs provides a reference point for improvement. It allows us to show a team managing a given API where their API(s) score in relation to the set of all Google APIs, a set of comparable APIs, or the set of APIs within a given product suite. By exposing outliers, we also know which set of APIs to study to identify successful patterns and which set need to be revised.

Customer satisfaction surveys also overcome one of our problems with the Explorer data; they give us a chance to ask for follow-up qualitative data that helps us interpret the quantitative data. When a respondent indicates that they are not satisfied with the API, we use an open-ended form asking for more details, allowing us to uncover pain points, misconceptions, and feature requests. In aggregate, this exposes cross-cutting issues within a set of APIs. Authentication emerges as a consistent friction point for our users, and is discussed in Section 5.

In this section, we’ve summarized two approaches we are using to uncover and understand API usability problems. This allows us to direct limited UX resources to more expensive lab studies where they are most needed. Our next step is to see what happens when we expose these metrics to the teams responsible for each API. We hope that providing them with a clearer picture of where they stand with respect to their peers will elicit self-correction and self-management over time. In the next section, we explore methods for discovering these issues before an API is released.

2. Challenge 2: Empirically-Grounded Guidelines

We aim to increase success and satisfaction among developers using our APIs by updating the APIs themselves. Any metrics we can define based on the API surface itself allow us to iterate without expensive lab studies in the process. This approach has been explored by Scheller and Kuhn (2015), who developed a framework for evaluation of an API, identifying measurable factors that impact an API’s usability. Similarly, Rama and Kak (2015) established a set of structural issues affecting an API definition and mapped them to a set of measurable metrics. While these methods are grounded in literature, large scale validation remains to be done. Validation often suffers from a lack of empirical data, a problem well understood in the software engineering community (Kaner & Bond, 2004).

We have started a program of work on API analytics, with the goal of empirically validating structural measures of API usability. Our primary goal is to get this information to API producers during the API design and review process, when it’s easiest to make changes.

As usability itself is something that is difficult to objectively measure, we chose to use API error rates as a proxy for usability during our preliminary analysis. Thus, the usability of an API is captured as the ratio of client side erroneous requests (4XX) / total requests to the API. In other words, we posit that APIs which result in more client side errors have a poorer usability.

For this analysis, we chose to use API Explorer usage data. This has the obvious shortcoming that users are not fully crafting their own API requests, as they are supported by a GUI based form rather than writing the code themselves. However, this does have the added benefit that we know a human is behind the call, as discussed in the preceding section. We chose to conduct our analysis at an individual method level, rather than, say, evaluating an API as a whole. This is due to the fact that individual methods differ so drastically in structure, that collating across an APIs entire set of methods would likely result in misleading results. For our analysis, we chose the following structural factors:

1. Number of parameters in the request (required and optional).
2. Number of required parameters in the request.
3. Number of different types of parameters (integer, string, float, etc).
4. Number of methods in the overall API.
5. Whether the API required a request body (True|False).
6. HTTP method for the API request (GET, POST, PATCH, etc).
7. Whether the request mutated server state (True|False).

Using a corpus of one year’s worth of API Explorer data, we built a linear regression model to assess how much of an API method’s 4XX response (client side error) rate could be predicted based on the aforementioned structural factors. We excluded methods which had less than 500 requests in the year, due to the low frequency of traffic. This left us with data on 724 different API methods.

In building our model, we got a coefficient of determination \((R^2)\) score of 0.256. While in some respect this may be interpreted as a poorly fitted model, given the many potential factors that can impact an APIs usability (both structural and otherwise), we are encouraged by the ability to predict as much as 25% of the variance.

While these result are early in nature, and further work (discussed below) is required before we begin to draw strong conclusions, this data leads to two preliminary conclusions. Firstly, objectively measured structural factors do indeed have a measurable impact on the error rate of API requests, a key determinant of an API’s usability. This helps to validate both the explorations presented within this paper, and the prior work in the field. Secondly, this technique, coupled with other algorithms such as outlier detection, can focus the attention of API usability practitioners. Those methods which stand apart, both positively or negatively, give cause for further investigation. This can help focus resource allocation, much like the techniques discussed in Section 1.

While these results are positive in nature, we acknowledge that there are limitations to this work. First, we wish to replicate this with our live API logs, thus, evaluating its applicability to ‘real’ API request data. Additionally, we used a small subset of the possible structural information. We will continue to expand our measures, building upon the literature on API design, software engineering metrics, and our own first hand experience working on API usability. Importantly, future analysis should evaluate the predictive value of each of the individual structural factors, and their role in the overall model. Furthermore, we acknowledge that while API request error rate is an important factor of API usability, it is certainly not the only metric to consider. We wish to extend our analysis to other metrics, including, but not limited to, API customer satisfaction as measured by API surveys, and ‘errors on the path to success’, a measure of how many erroneous requests an individual makes before achieving a successful call. Finally, linear regression is just one such model we can explore, with its own shortcomings as a way to predict variance. Future work will continue to explore other approaches to machine learning.

3. Challenge 3: Communicating Issues

The techniques above produce a considerable volume of data about API usability. Just as there are substantial resource constraints on the researchers studying the usability of the APIs, the engineers who design and implement the APIs are also busy. Making the information about API usability available to them in a way such that they can easily comprehend and use it is crucial to building better APIs.

Clarke (2004) describes one approach for doing this, using a Cognitive Dimensions profile and showing the difference between the ideal properties of an API and the ‘as designed’ properties. We have used a similar tool for summarizing dimensional profiles, however we have found that whilst they provide a good synopsis, challenges remain in communication.

Problems with API usability are often made of many small ‘papercuts’. Each individual problem by itself may not be too severe, however the overall result over many interactions is that the user makes relatively
little progress. To address this we use friction diagrams, such as Figure 4, which are generated based on observation of a user working with an API in a usability study. These diagrams show the progression of time from left to right. The timelines can be duplicated to highlight specific issues, as they are here. The bottom row within each timeline indicates ‘productive working time’, the middle row indicates ‘experiencing a problem’ and the top row indicates ‘not making progress or distracted’. For example, in this diagram, we were explaining two issues, one was that there are infrastructure problems with the tooling at that point in time, and the second is that the design pattern in use resulted in developers being repeatedly confused about the handling of state in the API. Whilst each individual interaction wasn’t too bad, the overall effect is considerable.

The second technique that we have adopted is to use the outcome of a simplified Cognitive Walkthrough as an explanatory model rather than a predictive one. Figure 5.(a) highlights the simplified version of the Cognitive Walkthrough that we have used for summarizing the interaction that a user might have with an API, derived from (Blackwell, 2008).

This gives us a frame that we can use to explain to engineers the challenges the users are having in interacting with the API. For example, Figure 5.(b) is used to explain a UX problem caused by an apparently missing method in an API (the delete method on the File class). The user has observed the list of available methods, but can’t find what they were looking for. Figure 5.(c) shows UX problem caused by the user needing to perform many subgoals into order to achieve the task (to write a line to a file).

These forms of communication have proven reasonably effective at helping engineers understand UX issues, and work on redesigning their APIs. This then leads to another challenge; once engineers have decided how they wish to adjust the APIs, how can they do that in a way that assists users?

4. Challenge 4: Supporting Change

In general it is difficult to change the APIs of a service that is used in the wild. APIs are primarily consumed by computers acting on behalf of their users. Even a small change to the API can result in outages of these dependent systems. APIs, then, have two groups of users: programmers who write the code against APIs, and the computer programs that run this code on behalf of their users. The combination makes safely evolving an API a substantial sociotechnical challenge.

Within Google the source code is largely stored in a single system (Potvin & Levenberg, 2016) which allows every use of an API to be located. Whilst it is still a considerable effort, this makes it possible to determine that all usages of an API have been upgraded before the previous version is disabled.

Externally, things are more challenging, as even APIs that have been marked as obsolete for a number of years show considerable usage. Google’s OAuth 1.0 API sent out a deprecation announcement in April of 2012 (Feldman, 2012). The primary use case wasn’t turned down until May 2015, and even then many users had not migrated to the new API (Denniss, 2015). The rest of the API was kept alive even longer, and will be shut down in October 2016 (Agarwal & Chun, 2016), over four years after the original deprecation notice. Here again we see a core difference between the defaults for a human and computer – once a process using an API is automated, it will continue using that API, and it requires a developer to spend additional effort to stop it.

The challenge of evolving APIs impacts more than deprecation schedules. At the API special interest group at CHI in 2009, many participants reported that they had seen public APIs withheld from users far longer than they should have been, due solely to the fact that evolving clients is difficult (Daughtry et al., 2009). Thus, designers are encouraged to make the API ‘perfect’ before release.

The use of typed APIs is a considerable benefit here. For example, a gRPC3 backed system allows the tooling to warn, or prevent compilation, when an API has been changed in a manner that is incompatible. In some cases it can even provide sufficient information that the tooling can rewrite the old APIs into a

3http://www.grpc.io/
Figure 4 – An example friction diagram, highlighting where and when a user experienced friction during a period of work with a Google API. The bottom row within each timeline indicates ‘productive working time’, the middle row indicates ‘experiencing a problem’, and the red row indicates ‘not making progress or distracted’.
Figure 5 – Explanatory models for effectively communicating API usability issues.

new format, for example Gofix⁴.

It may be possible to extend the potential scope of the types of changes that can be addressed this way using the kind of techniques outlined by Church et al. (2016). However, despite these tools, designing systems that support effective dissemination of changes across multiple organizations with legacy code remains a serious challenge and one that limits our ability to act on broad empirical evidence about API usability.

Now we have looked at the general challenges that we have faced in this area, let’s look at two specific questions - usable authentication and client library generation.

5. Challenge 5: Usable Auth

Time to Hello World (TTHW) is one metric regularly cited within industry as a key determinant of success for an API. The goal is to reduce friction during the getting started phase such that a new developer can quickly and seamlessly get to a successful Hello World state (or with web APIs, a 2XX response to their request). However, time is only one factor, and one must be careful not to optimize too heavily on a quick 2XX call, at, say, the detriment of long term success. Thus, within Google, we have been advocating that teams instead focus on Pain to Hello World (PTHW) and have been working with teams to understand how they can apply our study design to expose a more holistic picture of the getting started experience.

During task-based API usability studies, we carefully evaluate the user’s journeys, tracking the resources used, context switches, misconceptions, difficulties, mistakes made, feature requests, and task time. Highlighted in Figure 6, is one user’s unsuccessful attempt to make a 2XX call with a Google API. Each box represents a different context (code editor, documentation, API tooling, etc), and each line represents

⁴https://blog.golang.org/introducing-gofix
a context switch. Lines are color coded to represent the phase of the journey. During the red and green phases on the journey, which saw many frequent cycles between documentation and our API management tooling, this user struggled to understand which type of auth (authentication or authorization) was required for the API they were using. Through a number of these evaluations, spanning multiple Google APIs, and approximately 60 participants, we have seen auth emerge as one of the largest painpoints new and experienced developers face alike, somewhat confirming the results of (Biddle, 2014).

To help remediate these problems, we designed and launched a credentials wizard flow within the Google API Console that provides you with the correct credential given the answer to a few questions, namely, which API you are using, in which type of context you intend to use it (e.g. an Android application vs. web-application), and what type of data your API will be accessing (e.g. user vs. application data). This has helped with the problem, guiding users to the correct auth mechanism while educating them on key decision points which impact auth type. However, it does so by papering over a larger issue; namely, API auth is hard. Even if you manage to get it set up, it is unlikely that you understand how it works, and there is a good chance you are still doing something insecure (e.g., exposing a sensitive credential on GitHub).

When dealing with APIs, you may have to deal with API keys, OAuth clients, and service accounts, with each broad category including sub-items meant to improve security. Some APIs require working with multiple types. The complexity has arisen for good reasons and has helped to improve API security, however it has come at the detriment of developer experience. Most of the attempts to improve API auth usability that we have seen tackle only a subset of the design space. Such local optimization has led to more proliferation of credential types, documentation, restrictions, and failure paths.

We believe that the API authentication design space is ripe for improvement. While there may be more local optimizations that helps, we believe a holistic design would have impact that is orders-of-magnitude greater over time.

6. Challenge 6: Usable Client Libraries at Scale
An attractive feature of web APIs is that they’re accessible in any language with an HTTP library – from an AWS Java SDK to a Github client in Emacs Lisp. However, this flexibility comes at a cost: it’s hard to design an API that feels equally usable across all potential client languages. In particular, we’re faced with several core design decisions: Do we want a hand-written client library for each language and API, or are generated client libraries sufficient? If we hand-write our client libraries, how can we ensure that we’re consistent as we move from language to language or API to API? Generating client libraries makes it easy to apply a fix across all APIs, but makes it equally easy to invalidate user code or samples across all those APIs. How do we balance the two?

If developer time were free, hand-writing client libraries for each API and language has significant advantages because we would have control over what was optimized for each language/API pair. In fact, one could even imagine multiple libraries for a given language based on disparate idioms (eg callback-
based and promise-based libraries for Javascript). This also affords the possibility of matching particular API features to language constructs, for example using Python’s `with` statement to model database-like transactions.

Providing libraries in multiple languages across hundreds of APIs simply doesn’t scale. For instance, Google currently supports client libraries in 10 languages across many of our APIs. The problem isn’t just developer time or the number of APIs that Google supports, even a small team building a single API backend can’t be expected to understand common idioms, conventions, and packaging across all languages.

While hand-written code is usually idiomatic, it can also be idiosyncratic. Surveys suggest (StackOverflow, 2016) that many developers end up working across at least two languages; having inconsistencies when using two different languages to talk to the same API can be frustrating and error-prone. A common example in this category is date/time-handling libraries, dealing with the inconsistencies of date libraries from language to language is a common pain point for developers. Generating code makes it possible to have consistency along two axes: talking to multiple APIs from a single language, and using the same API from multiple languages.

At Google, we have examples of both auto-generated and hand-written client libraries, and we’ve performed user studies on PTHW for both. In one example, we compared and contrasted the experience of developers working with the gcloud-node (hand-written) and google-api-nodejs-client (auto-generated) client libraries within the context of two Google Cloud Platform APIs. Note that the hand-written libraries also have documentation and samples written specifically for that library, whereas the auto-generated libraries refer to language-agnostic documentation. In our studies, participants were asked to implement some canonical functionality with either the auto-generated or hand-written library. We then compared and contrasted the experience, looking not only at the success and productivity of the study participants (i.e., how successfully they could complete the study tasks), but also any misunderstandings or errors that occurred during the session, the number of resources (such as documentation) required, and the distribution of time during the session (e.g., time spent reading documentation vs. debugging code, etc).

Although at a small scale, we saw interesting differences in the PTHW journey of the participants. Those working with the hand-written libraries were more successful in completing the development tasks, made less erroneous assumptions about the surface of the API, and required fewer resources in order to successfully complete the tasks.

The contrasting experience highlighted the merits of hand-written libraries, which can be carefully designed to ensure language idiomaticity and nuanced functionality specific to the APIs. In this case, gcloud-node, our hand-written library, is optimized for a specific subset of Google Cloud Platform APIs, while the auto-generated node library can be used with any Google API. We believe that the need for autogeneration of client libraries is clear, but we still have some work to do before the overall developer experience is up to par for that of hand-written libraries.

### 7. Conclusion

In this work we have summarized issues that we have faced whilst developing and maintaining production APIs. There are more questions and open areas of research than answers, and we’ve shared some of our approaches to these.

Researchers in industry are in a good position to explore improving designs via usage data. However there are many other concerns that could benefit from the broader research approach taken in academia. Some examples include: what would empirically justified guidelines of API design based on data look like? How might these guidelines be derived from multiple sources of data? How do they relate to existing design patterns? How should these guidelines and results best be presented to developers? How might we longitudinally validate the effectiveness of guidelines? How should change of APIs best be managed and communicated both to developers and to users?
There is clearly much more to be done in this important space.

8. Acknowledgements
We would like to thank Sanjay Ghemawat and Dan Li for their advice and guidance on our research, as well as for providing feedback on the drafts of this paper; and Tao Dong and Harini Sampath who also gave feedback on our drafts.

9. References
Stylos, J., & Myers, B. (2007). Mapping the space of API design decisions. In IEEE symposium on visual languages and human-centric computing (pp. 50–60). IEEE.
Investigating Domain Specific Visual Languages for Interactive Exhibitions

Andrew Stratton, Andy Dearden, Chris Bates
Cultural, Communication & Computing Research Institute
Sheffield Hallam University
A.Stratton@shu.ac.uk, A.M.Dearden@shu.ac.uk, C.D.Bates@shu.ac.uk

Abstract
Quando is a visual programming tool designed to enable Cultural Heritage Practitioners (CHPs), who do not have experience of computer programming, to create program scripts for interactive exhibits for galleries and museums. This paper reports preliminary findings from a qualitative investigation of the responses of CHPs to the Quando environment.

1. Introduction
In Cultural Heritage there is increasing desire to create digital interactive exhibits, intended to increase visitors’ engagement and understanding and increase the ways in which public spaces are viewed and used. Adoption is seen as slow and limited with ‘Over 60 per cent of arts and cultural organisations ... constrained ... by a lack of staff time and funding’ and ‘over 40 per cent report a lack of key technical skills…’, (Nesta, 2013).

CHPs typically commission experts to create software solutions; reducing the opportunity for CHPs to change/update the interactive experience and limiting their involvement.

Tools for CHPs
Current tools for creating digital interactivity are often technically demanding, especially for new technologies that might inspire visitors. Platforms, such as meSch, see Wolf (2015), are being created to support CHPs in specifying Interactive Digital Content without requiring programming language skills. Using meSch tools, high level ‘recipes’ allow CHPs to add/remove content and conditions to interaction description/s. This allows CHPs to create and/or modify the content of technically complex interactions at a cognitively suitable level, e.g. manipulating visual icons representing multi-media content, associating them with visitor data such as demographics or interests, e.g. the AtlantikWall exhibition, Museon (2016), requirements allow visitors to choose specific interests at the start of the exhibition and subsequently interacting with exhibits to trigger associated multi-media presentations. The meSch approach often requires a networked server for content delivery. Joining the recipes to the execution environment is dependent on high level technical skills, and the range of interactive behaviours available for a CHP is limited to the pre-defined library.

An alternative, lightweight, approach is proposed, where Quando, a set of visual ‘block’ based tools, allows CHPs to design interaction using a Visual (Domain Specific) Language in a similar approach to End User Programming, see Ko et al. (2011) for a summary. CHPs work at a higher conceptual level, concentrating on matching visitor interaction to content, i.e. using a domain specific language, see Fowler (2010). Quando seeks to offer a simpler approach for developers of the visual blocks that the CHPs will use. To explore this approach, CHPs were interviewed and asked to perform tasks in order to investigate the appropriateness of Quando in describing interactions.

2. Visual Language Design
The Quando language design follows an event matching approach. Rule blocks are created to describe events occurring in the environment of the exhibit. These events may represent temporal events (e.g. a regular event occurring every \( n \) seconds), or visitor triggered events (e.g. approaching a sensor). In order to respond to events, Rule blocks act as ‘containers’ for Action blocks; where the Actions are usually visible/audible effects for visitors, i.e. presenting multimedia content.

Quando uses Google’s Blockly language (Frazer, 2016), which was designed for education purposes. Blockly has been chosen because, unlike Scratch, (Resnick, 2009), it allows customisation of blocks and allows output to different programming languages to be generated.
Quando blocks are intended to allow CHPs to describe behaviour visually, without learning a traditional programming language. The blocks are used to generate a textual programming script for execution on an appropriate platform. A library of Blocks is designed using terms that are intended to be familiar to CHP’s rather than traditional imperative programming statements.

Using the current library of Blocks in Quando, a variety of visitor interaction behaviours can be described. These behaviours can be categorised by increasing interaction depth/complexity:

- **Static** - Fixed content is displayed
- **Sequence** - A regular, timed, sequence of content
- **Responsive** - Responding to an anonymous Visitor’s choices, using a stateless model
- **Session** - Based on a Visitor’s preference/demographic/interests/choices during a visit

Two further behaviours, not yet implemented, would support Historic behaviours using details of a Visitor’s previous visit/s and Inter-site behaviour based on a ‘partner’ site’s data for a visitor.

Prior to developing support for these more complex interactive behaviours, an investigation (see Section 3) has been conducted to explore whether a block based language would be acceptable for this user population, and to explore issues that might influence future development.

Quando blocks have focused on textual Identifiers, derived from the AtlantikWall exhibition interaction requirements, Museon (2016). The AtlantikWall requirements included video/audio/text and image presentations, for which Action blocks were defined, e.g.

![Figure 1 - Example Blocks with content selection by drop down list](image1.png)

Rule blocks were designed to distinguish between different types of event matching. Temporal blocks include ‘Every’ and ‘Do for’ blocks with CHP selected timings. Interaction Blocks were designed that show integration with recent technology, in this case through use of the Leap Motion controller (Bachmann et al. 2014) which uses two infrared cameras to detect visitors’ inputs using touch free hand movement.

Note that the behaviour associated with an exhibit may be described by multiple independent rule blocks, each triggering actions when its conditions are met.

![Figure 2 – Example Temporal and Interaction Rule Blocks](image2.png)

The Blocks were developed iteratively, starting with a device specific example script and resulting in a Block definition including interface configuration and a generative script, e.g.

```javascript
var HAND_COUNT = _hand_count;
quando_editor.defineRule(
    name: 'When Hands', next:false, previous:false, 
    interface: [ [name: HAND_COUNT, title: ''], number:1], 
    javascript: function(block) {
        var statement = quando_editor.getStatement(block, STATEMENT);
        return "When Hands:(+= when editor.getNumber(block, HAND_COUNT) ++,"\n    + " statement())"; 
    });
```

![Figure 3 – Example Javascript Block definition](image3.png)

### 3. Investigation

CHP Participants were recruited by recommendation and cold calling. The investigation focused on Participant’s background, interactive use of the Quando tool and subsequent feedback. For interactive
use, twelve behaviours were used; seven Static/Sequence, two Responsive using the Leap Motion and three Session behaviours. For each question, participants were asked to interpret/edit the behaviours and rate their confidence in their answer. A semi-structured interview followed with participant answers recorded using audio and video which was transcribed after the interview.

An outline description of example behaviours is given below:

- Participants were shown the behaviour below and asked to recreate the same sequence in the editor. A paper printed copy was also visible to the participants.

  ![Figure 4 – Experiment 1.06 Sequence Creation](image1)

- The responsive behaviour below was shown and participants asked to modify the behaviour: ‘Change the sequence to show the carving for 2 hands and the jewellery for 1 hand.’

  ![Figure 5 – Experiment 2.01 – Modify Responsive](image2)

Screen and Video/Audio was recorded of participants using the Quando editor in a full screen browser, to reduce navigation issues. The Leap motion was mounted at the top of the screen; reducing errors by increasing the distance from the participant.

### 4. Initial Observations

At the time of writing, five interviews have been conducted with CHPs using Quando for about twenty minutes. Two were experienced Curators, one a Museum Events/Exhibition Coordinator, one a Museum Software Developer and a Museum Researcher/Freelance Exhibit Designer. All CHPs have Postgraduate qualifications; three in Cultural Heritage, one each in Design and Software Engineering; only one of the participants had software development skills. Reviewing this early interview data, provides some interesting observations:

- Feedback was mostly positive - ‘I’m really impressed with it. I want it’ and ‘It would be brilliant for us to have this’.

- Simplicity was commonly identified - ‘once you’ve done it a couple of times, it’s very easy to use’, ‘it’s quite straightforward…once you understand the principles’, ‘(it’s) quite logical’.

- Some feedback was negative, ‘it’s quite alien to me…I’ll immediately find a barrier when it’s something…technological’. The same participant was also positive about the touch free interaction - ‘I liked the interactive element, the way it reflected ambidextrous behaviour was really good in getting across the point. I liked that…as for the others. I’m not sure’.

- The colours were negatively identified with, ‘I’m not too keen on the colours’ and ‘don’t like the colours’. Blockly limits text within blocks to white text and has a restrictive colour model, saturation and value are shared across all and pastel colours cannot be used.

- Blockly platform behaviour caused difficulties with dragging and dropping blocks. A small yellow line is used to indicate the target of a drag; this needed explaining to most participants. Also, when blocks are attached in sequence, dragging a block will also drag any blocks ‘attached’ below it; most participants tried to undo this behaviour when it happened.

- When asked about programming styles for specifying common actions, participants universally disliked repetitive content descriptions, e.g. where the same action blocks are repeated within multiple rules – e.g. see below (where the Play Video block is repeated), though there was awareness of it’s value - ‘it depends where you’re going with it next’.
The Rule/Action Blocks were mostly liked - 'I also like how the actions fit within the rules'.

Participants identified the benefits of the approach for non-programmers – ‘for me with no IT training, it's really useful if I could setup digital interactive exhibitions’, ‘I feel quite confident that I could put something rudimentary together now…I haven’t had to outsource and it means I can go and do it again’ and ‘it’s nice that you don’t have to code it’.

There were concerns about how well the approach and the tools would scale to more ‘complex’ interactions – ‘I wonder how difficult, how complex, you can get with it though’.

Children were identified as a likely audience, in particular the touch free interaction – ‘that kids could interact with’ and ‘it (leap motion) would work particularly good with schools’.

There was an expectation that training would be required – ‘I would be quite happy to go on a training course’ and ‘if they (museum staff) have the training’.

5. Future Work
The use of visual tools for CHPs to describe visitor interactions is promising and worth pursuing. More complex behaviours need to be considered as well as alternative visual tools.

Further artefacts will be designed to explore CHP conceptual models, and appropriate authoring tools, of interest based interactions, including session, historic and geographic based interactions, i.e. interactions that depend on current and/or previous visits.

An analysis of the Quando concepts using Cognitive Dimensions (Blackwell et al. 2001) is also planned.

References


Programming: Further Factors that Influence Success

Keith Quille  
Department of Computer Science  
Maynooth University  
keith.quille@nuim.ie

Susan Bergin  
Department of Computer Science  
Maynooth University  
susan.bergin@nuim.ie

Abstract
This paper revisits five previous studies spanning a decade, which were used in the development and validation of a computational model named PreSS (Predict Student Success) to predict student success in an introductory programming module with 80% accuracy. PreSS was initially developed over three studies from 2004 – 2006, recording data from 240 students across multiple institutions incorporating three studies. These studies examined 25 factors and six machine learning algorithms, resulting in a model that used only three factors and the naïve Bayes machine learning algorithm (coupled with multiple other statistical techniques) called PreSS. The authors completed two subsequent independent studies over the academic years 2013- 2015 that were used to investigate if the PreSS model was still valid on a modern data set while also collecting new data (factors) that were not considered during the initial development of PreSS. These two studies successfully validated PreSS, even with a decade separating the student profiles and landscapes.

This paper has two main objectives; the first objective is an investigation on factors gathered during the initial development of PreSS that were not used in the final model. This is important as several factors were found to be significant at the time but were excluded as their associated sample size was small and the goal was to develop the most generalizable model possible. PreSS is arguably a universal model for two reasons. 1) PreSS is independent of the programming language used. PreSS has been so far been exposed to 6 different programming languages that include: Java, C#, Processing, Python, Visual Basic and C++ and still maintained the same high level of accuracy. 2) PreSS is also independent of any specifics of the cohort sampled such as gender bias, institution bias, age bias etc. The second objective of this paper is an examination of additional data collected in two recent independent studies, to determine if incorporating some or all of these new factors could improve the accuracy of PreSS.

This paper reports on 90 experiments using data from all five previous studies, examining 126 possible factors. The work successfully identified 16 factors that when used in combination with the original PreSS factors either produced significant increases in prediction accuracy or exhibited noteworthy findings. Some of the 16 factors produced substantial gains in accuracy (in some cases in excess of 8%) or when integrated into the PreSS model, revealed interesting substitutions of factors.

1. Introduction
Computer Science (CS) non-progression rates in Ireland are alarming with a large number of students failing to progress each year. Currently non-progression rates are 25% in CS, which is significantly higher than the national average of 16%. In two recent reports (2010 and 2016 respectively) that cover all Irish higher level institutions, CS was found to have one of the largest, if not the largest rate of non-progression across all National Framework of Qualification (NFQ) levels, from level 6 to level 8 (Liston et al. 2016; Mooney et al. 2010). In addition, CS is one of only two fields of study, where the non-progression rate has increased since the first report in 2010. It is well acknowledged that a main contributor is students struggle to succeed in their initial programming module, a staple in most first year CS courses. Early identification of students who are at risk of non-progression is often hindered by the very high student-lecturer ratio (100:1 or greater) and often lecturers are not aware of the student’s progress until after the initial assessment. Numerous approaches have been trialled to varying degrees of success to improve learning and assessments outcomes. Our institution is no different with numerous initiatives implemented to improve outcomes, including the development of an automated adaptive assessment systems, (Traynor et al. 2006), novel teaching approaches such as problem based learning (O’Kelly et al. 2004) and more recently a Programming Support Centre run by peer tutors (Nolan et al. 2015). As computer programming is not a traditional Leaving Certificate subject or topic, there are no prior indicators of a student’s ability available to lecturers at an early stage, to enable the introduction of appropriate interventions. This often renders interventions, or appropriate methodologies inadequate as their introduction may have been too late in the course delivery to make a significant difference.
2. Background

2.1. PreSS

PreSS is the result of work completed over three years, between 2004 and 2006. This work consisted of several parts: three studies (a pilot study, a main study and an epilogue study), the identification of early predictors of performance in an introductory programming module, an investigation of multiple machine learning models for predicting student performance and finally the development of a computational model called PreSS (Predict Student Success) (Bergin and Reilly 2006; Bergin 2006).

The three studies were part of a longitudinal study, which was completed over four institutions during the three years. These institutions consisted of a University, two Institutes of Technology, and a Community College thus spanning all levels of Higher Education, from the National Framework of Qualifications (NFQ) level 5 to level 8. The pilot study recorded data using a questionnaire, which in turn lead to the development of the instruments that were used in the main study. The main study recorded a substantial amount of data from 123 students during the academic year of 2004-2005. Four paper based instruments were used to collect the data in the main study, a background questionnaire, a programming self-esteem questionnaire, a self-efficacy questionnaire and a motivation and learning strategies questionnaire (Bergin 2006; Bergin et al. 2005; Bergin and Reilly 2005). An epilogue study with 21 participating students was also conducted after the main study and was used in further analysis and validation of findings from the main study.

In Bergin’s main work 25 factors were examined, obtained from the data collected using the four instruments in the main study. Bergin used multiple statistical techniques when examining factor selection for use in the prediction model. Stratification was used as the initial step to ensure that all the data is in a state of homogeneity. Ten-times ten-fold cross validation (10FCV) was also implemented which is an acknowledged method of avoiding over fitting when accessing how the results of statistical analysis will generalise to a larger independent data set, Witten and Frank (2005). This method is more suitable than other conventional methods of statistical model validation, as the validation set is not represented in the training set, thus reducing bias and therefore over fitting of the accuracy prediction. Initially, Bergin examined over 40 logistic regression models using combinations of the 25 factors on the data from the main study. This resulted in a single prediction model that was marginally superior. This prediction model used three factors: programming self-esteem, mathematical ability (based on a high school mathematics exit examination) and the number of hours per week that a student plays computer games.

The pre-processed programming self-esteem data consisted of ten questions (Bergin 2006) and was based on the Rosenberg self-esteem questionnaire but modified to reflect a student’s perception of their programming ability (Rosenberg 1965). Cronbach alpha values used to estimate the internal consistency of psychometric questionnaires, were compared to investigate if the modified Rosenberg questionnaire had an equivalent or greater internal consistency than the unmodified questionnaire, thus showing that the modified questionnaire is a valid questionnaire to measure programming self-esteem. The Cronbach alpha values for the unmodified Rosenberg self-esteem questionnaire were in the range of 0.82 to 0.88 and for the modified questionnaire, the alpha value was 0.91 (Bergin 2006). Principle Component Analysis (PCA) was used to reduce some attributes that consisted of multiple data points to one value which accounted for as much of the variance in the multiple data points as possible. PCA was used to reduce the 10 questions of the programming self-esteem attribute to one principal component (Bergin 2006). Bergin examined multiple prediction models using six machine learning algorithms: Nearest Neighbour, Decision Tree’s, Naïve Bayes, Logistic Regression, Support Vector Machines (SVM) and Backpropagation Networks (Bergin and Reilly 2005) with the results of this work presented in Table 1 and revisited in a study (Bergin et al. 2015).
Table 1: Comparison of machine learning algorithms from Bergin’s study

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>78.28</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>76.47</td>
</tr>
<tr>
<td>Backpropagation</td>
<td>75.46</td>
</tr>
<tr>
<td>Support Vector Machine</td>
<td>77.49</td>
</tr>
<tr>
<td>Decision Tree (C4.5)</td>
<td>74.49</td>
</tr>
<tr>
<td>K Nearest Neighbour (K=3)</td>
<td>71.64</td>
</tr>
</tbody>
</table>

The programming self-esteem and mathematical ability factors were found to have a positive relationship with performance while the number of hours a student plays computer games was found to have a negative effect. Bergin analysed the results of six different machine learning algorithms and selected naïve Bayes as the algorithm of choice for multiple reasons; firstly it produced the highest accuracy of the six algorithms examined, but also naïve Bayes is a probabilistic classifier, where the algorithm computes a probability of how confident it is for each prediction. For example, a student who was classified as a strong programmer and where naïve Bayes is not confident in its prediction may need further investigation. This is a very useful tool in the case of the PreSS model, as misclassifying a strong student as weak is a significantly less adverse, than misclassifying a weak student as strong. This coupled with the fact that naïve Bayes is computationally inexpensive and thus is appropriate in a real-time web-based environment, which was outlined as possible future work of Bergin and in section 2.2.

2.2. PreSS#

PreSS# is a web based educational system, with the PreSS model at its core was developed during the academic year 2013-2014 (Quille et al. 2015), to continue Bergin’s future work that was outlined after the development of PreSS. Bergin’s model, used a paper based data collection method, and thus it was time consuming to gather and input the data, coupled with this the data collected also contained some partial and dirty data entries. If PreSS were to be used across multiple institutions with the results of the predictions expected in real time, the paper based collection method would be unsuitable. PreSS# could be used as a tool in an educational environment that would allow educators to make informed decisions about methodologies, differentiation and interventions at an earlier stage that will enable both weak and strong students to achieve their highest potential in real time. Additionally a large scale multi institutional study could be completed using PreSS#, examining the current effectiveness of the PreSS model.

The development of PreSS# consisted of four main phases. The independent development and validation of a PCA and naïve Bayes sub system, the integration of these two subsystems, into the main web application, the development of an intuitive, secure, mobile responsive interface and front end, and finally the validation of PreSS# using the same data that Bergin used in her main study. Both the PCA and naïve Bayes sub systems, were developed as an independent library of classes, which allowed for integration into PreSS# and any other future research and development. Both sub systems were independently validated using data and outcomes from Smith 2002 and Souza 2014. The PreSS# front end employed several techniques to ensure ease of use, rapid development techniques, security and services. Further details can be found in Quille et al. 2015. The system was validated using the same input data as Bergin’s main study (Quille et al. 2015) and the results are presented in Table 2.

<table>
<thead>
<tr>
<th>System Validation</th>
<th>Accuracy %</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreSS# (.NET)</td>
<td>80.39</td>
</tr>
<tr>
<td>PreSS (WEKA)</td>
<td>80.39</td>
</tr>
</tbody>
</table>

Table 2: Results from PreSS validation, (Quille et al. 2015)

A two tailed t-test for a binomial distribution was completed to confirm the validation of PreSS#, concluding that there was no statistical differences between the prediction accuracies of .NET and PreSS (using WEKA), with a \( p \text{ (value) } = 1.0 \) and a \( t \text{ (value) } = 0.0 \).
2.3. Study: Programming, Factors that Influence Success Revisited and Expanded

After the validation of PreSS#, the authors completed two independent studies during the academic years of 2013 – 2015, which were used to validate PreSS on a modern data set, (Quille et al. 2015). This study was conducted for two reasons. Firstly, the landscape, population and the student profile has significantly changed over the last decade since PreSS was developed. Secondly, validation of any study is important as all too often studies are undertaken and never repeated, thus the value of their findings remains questionable.

The two studies named: S14 (conducted in the 2013-2014 academic year) and S15 (conducted in the 2014-2015 academic year) were hosted by a Community College completing an Advanced Certificate in Computer Science. This is a similar institution to one of the institutions investigated in Bergin’s main study. S14 consisted of 34 students with a 5:29 ratio of female to male students. The divide of successful students compared to non-successful students was 16:18 and the ratio between mature students (students under the age of 23) and non-mature students (students 23 years of age or older) was 12:22 respectively. The second study S15 consisted of 26 students. Four females and 22 males took part in the study. The overall final results showed the divide of weak to strong students at 9 to 17 and the ratio between mature students and non-mature students was 3:23 respectively. These statistics are highlighted to show the variation between the student cohorts even though they were conducted in the same institution. There were also further differences between Bergin’s main study and S14, S15. The majority of students in the main study used Java as the introductory language, where as in S14 and S15 the language used was C#. This was the first time that C# was used with PreSS and to the authors knowledge, in any study to predict programming performance. C# was not commonly used in education or industry until sometime after PreSS was developed.

PreSS# was used to automatically collect the data, ensuring the data was clean and no data samples were missing due to the interfaces automatic validation techniques. An additional survey was completed alongside both, S14 and S15 studies, with the goal of capturing additional novel factors which may add value. The survey questions were selected based on hypothesis that some factors, which may have not existed or significantly impacted the model during Bergin’s study, were worthy of examination and could contribute to increasing the PreSS model’s prediction accuracy, details are provided in Appendix 1.

The results of this investigation were obtained by training the PreSS model using Bergin’s 102 data samples from the main study conducted a decade previous. The process used in this investigation was identical to the process used by PreSS, using the same machine learning toolbox (WEKA) and statistical methods to compute the predictions. Both prediction accuracies using S14 and S15 were independently compared to Bergin’s accuracy using a Welch’s t-test and a binomial distribution to calculate the variance. The accuracy achieved in the S14 study produced a p value of 0.8592 and a t value of 0.1778 thus showing that there was no statistical difference between the two accuracies. The accuracy achieved in the S15 study produced a P value of 0.5669 and a T value of 0.5748 thus showing that there was no statistical difference between the two accuracies. This means that even though the PreSS model was developed nearly a decade ago, similar levels accuracy are achievable in a considerably changed landscape today.

This result was very significant as both S14 and S15 studies consisted of very different student profiles with age, results and gender ratios differing significantly. Furthermore the student profiles from both these studies, differed significantly to Bergin’s student profiles. This was partly due to the time lapse between studies, institution representation (Community Colleges represented less than 7% of institutions in Bergin’s main study) and programming language used. This considered, PreSS still produced statistically similar accuracies using S14 and S15. This implies that even though Bergin’s PreSS model was developed over ten years ago, it is still valid, universal and not biased towards an institution, student profile, programming language, era or any other known factor, thus always achieving the same high level of accuracy.
3. Motivation
From unused factors in the final PreSS model, Bergin discussed the inclusion and combinations that may have value and were worth additional investigation, one in particular was gender and its relationship to the factor, hours spent playing computer games. Bergin noted that, removing the number of hours students spent playing computer games improves the classification of female students suggesting that its inclusion is not constructive in predicting female student performance. Bergin also noted that developing a prediction for each institution used in the main study resulted in higher classification accuracy for each individual institution. Finally Bergin noted that some factors investigated were only available in smaller sub sets of the main study data, due to restrictions in the data collection method and although showed value in the small sub sets could not be verified on a large data sample, thus were not included in the final PreSS model.

In addition new factors were captured by the additional surveys administered alongside S14 and S15, multiple investigations were now required to examine if the addition to or the replacement of the original PreSS factors with these new factors, captured during S14 and S15 may lead to an increase in the models prediction accuracies. Although these factors are not present on Bergin’s main study data, and only on the two smaller studies, S14 and S15, if improvements to the model can be shown here, they should warrant inclusion in future studies to investigate their true value.

The aim and objectives for this work is to examine if the remaining $\approx 20\%$ error in prediction accuracy of PreSS can be reduced, whilst still keeping the PreSS model universal. To do this an examination of the unused factors from Bergin’s original work, (from the large amount of data collected in the main study) and additional new factors collected from the two studies S14 and S15 (Quille et al. 2015) will be carried out. Where any improvements or value is found, these factors will be included in an upcoming large study.

4. Main Study Revisited
4.1. Research Design
The authors revisited the data in Bergin’s main study. There were a total of 123 students that participated with 113 recorded data points (factors). These 113 recorded data points consisted of questions and normalisation or summations/differences of combinations of questions. An example of this was a variation of the hours spent playing computer games factor used in the PreSS model. This attribute was the hours spent playing computer games during the duration of the introductory programming module. Bergin also recorded the amount of hours spent playing computer games before the commencement of the introductory programming module. These two questions also had a third data point, the difference between the two. This is just to show that Bergin not only examined a large amount of questions, but also examined outputs using combinations of questions. These 113 data points were not all recorded for every student due to some institutions not being able to complete each or all of the four instruments used, due to time or administration constraints. Due to the large quantity of data points, a method was developed to identify data points that may have value or be of interest for further investigation. Standard deviation and therefore variance was calculated for each data point over all the students that it was recorded for. The rationale for this method was based on the hypothesis that if the variance was small for a data point, it may not have value as a predictor attribute due to the fact that if each student had a similar value (due to low variance) when that value was used in a naïve Bayes classifier, the probability for that attribute would be similar in each case adding no value to the predicted output. This was further confirmed when the variance was calculated on the three attributes used in PreSS, each having a relatively large variance, which was then used as a baseline variance for selection of further data points for inclusion in this study.

This resulted in the identification of multiple data points that were to be re-examined. These data points were then developed into experiments for further investigation. Thus 65 experiments were performed which included the addition and multiple exhaustive combinations and substitutions of the data point, with the original PreSS model’s attributes and then this new model was statistically analysed.
using a student’s t-test to examine if the new prediction accuracy produced, yielded a significant increase in accuracy above the PreSS model. Noteworthy findings were also recorded.

Although in total, there was 113 data points recorded in Bergin’s main study, these data points were not recorded for every student as previously mentioned. This would result in a biased finding if, for example data point x was only recorded for 50 students and the prediction result of this new model which included x in some variant was obtained using a sample size, n = 50, and then the prediction accuracy was compared to the original PreSS prediction accuracy with a sample size, n = 102. To ensure an unbiased statistical comparison, the original PreSS model was re-run only on the students that were used in each investigation, resulting in a baseline prediction accuracy to be statistically compared with the new model incorporating the new data point. A Students t-test was used to examine if a significant difference was found in accuracy between the baseline accuracy and the experiment accuracy.

4.1. Experiment Results
Table 3 shows, of the 65 experiments conducted using the main study, the 12 experiments that resulted in a significant gain in accuracy or showed value when combined in some way with the PreSS model.

<table>
<thead>
<tr>
<th>Experiment ID</th>
<th>New Model Attributes</th>
<th>Sample size (N)</th>
<th>Accuracy Increase %</th>
<th>p value</th>
<th>t value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mathematical grade, hours spent playing computer games and what a student believed their final overall grade would be at the beginning of the module.</td>
<td>56</td>
<td>8.93</td>
<td>&lt; 0.001 ***</td>
<td>13.917</td>
</tr>
<tr>
<td>2</td>
<td>Mathematical grade, programming self-esteem and the difference in hours spent playing computer games before the commencement of the course with that of the hours spent playing computer games during the course.</td>
<td>107</td>
<td>7.50</td>
<td>&lt; 0.001 ***</td>
<td>12.200</td>
</tr>
<tr>
<td>3</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and what a student believed their final overall grade would be at the beginning of the module.</td>
<td>56</td>
<td>7.20</td>
<td>&lt; 0.001 ***</td>
<td>11.013</td>
</tr>
<tr>
<td>4</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and a student’s anxiety, fear and or uneasiness of examinations and assessment, and a focus on the negative while completing them.</td>
<td>68</td>
<td>2.94</td>
<td>&lt; 0.001 ***</td>
<td>5.293</td>
</tr>
<tr>
<td>5</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and a student’s feeling on their own ability on programming concepts, design of programming logic and completion of lab assignments.</td>
<td>110</td>
<td>1.82</td>
<td>0.0010 ***</td>
<td>3.165</td>
</tr>
<tr>
<td>6</td>
<td>Mathematical grade, programming self-esteem and the number of hours a week the student worked in a part time job.</td>
<td>111</td>
<td>1.81</td>
<td>0.0024 **</td>
<td>3.064</td>
</tr>
<tr>
<td>7</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and a student opting for more challenging / relevant course material when given a choice, even at the cost of a higher grade if less challenging material was chosen.</td>
<td>68</td>
<td>1.40</td>
<td>0.010 *</td>
<td>2.606</td>
</tr>
<tr>
<td>8</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and a belief the student has, that if they work hard that they will succeed and if they do not work, that they will fail and it will be the student’s own fault that they did.</td>
<td>68</td>
<td>1.40</td>
<td>0.010 *</td>
<td>2.606</td>
</tr>
<tr>
<td>9</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and a student’s belief that they will be successful in the course before it commences.</td>
<td>68</td>
<td>1.40</td>
<td>0.010 *</td>
<td>2.606</td>
</tr>
</tbody>
</table>
Table 3: Bergin’s main study unused factors that have shown value where
* Significant at $p < 0.05$; ** Significant at $p < 0.005$; *** significant at $p < 0.001$;

<table>
<thead>
<tr>
<th>Experiment ID</th>
<th>Factor Description</th>
<th>$t$-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and a normalisation of experiment ID 5 representing intrinsic questioning.</td>
<td>68 1.40 0.010 * 2.606</td>
</tr>
<tr>
<td>11</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and a normalisation of factor: “A student penuing a high grade in a class as their main motivation / learning outcome” representing extrinsic questioning.</td>
<td>68 1.40 0.010 * 2.606</td>
</tr>
<tr>
<td>12</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and gender.</td>
<td>107 0.00 0.1200 1.550</td>
</tr>
</tbody>
</table>

4.2. Summary & Discussion

From a total of 65 experiments run on Bergin’s main study, 12 combinations of factors have emerged which may have added value.

From both experiments ID 1 and 3 from Table 3, the factor was: the early expectation of the final grade that a student will achieve in the introductory module. This factor of a students expected grade was recorded early in the course delivery and when directly added into the PreSS factors resulted in a very significant increase in accuracy which was at just over 7%. Furthermore when this factor replaced the programming self-esteem factor, (thus removing the programming self-esteem factor completely), the overall accuracy compared to the control accuracy showed the highest increase in accuracy for this study of just under 9%. This is a noteworthy result, which could contribute to the overall goal of increasing the prediction accuracy of PreSS.

In experiment ID 2, from Table 3, the addition of the difference in time spent playing computer games prior to starting the course to the time they spent playing computer games during the course, produced a very significant increase in accuracy, up to 8%. This although, seems related to the original PreSS factor of hours spent playing computer games during the course, the substitution of this difference as a factor may have value over the original PreSS factor and is worth further investigation.

From experiment ID 4, Table 3, which was multiple self-ranking questions summated to give an indicator of a student’s anxiety, fear and uneasiness towards formative assessment and examinations. When this factor was added to the PreSS factors, the resultant model produced a significant increase in accuracy of almost 3%. This like in experiment ID 5, Table 3, may be an indicator of programming self-efficacy. The multiple experiments, id’s 7 – 11, Table 3, also showed a significant in accuracy although the overall gain may be classified as minimal. None the less, they may add value to an updated model and for this reason further investigations should be completed using these data points.

In experiment ID 5, Table 3, a student’s feeling on their own ability on programming concepts, design of programming logic and completion of lab assignments, added a significant increase to the models accuracy. The authors hypothesises that this may be related to a student’s programming self-efficacy and due to the large increase in accuracy would warrant further analysis on a modern data set. The same additional analysis should be applied for the factor: the amount of hours in part time employment while completing the course, experiment ID 6, Table 3. Although the addition of this factor to the PreSS factors did not yield any value to the model, when it replaced hours spent playing computer games a significant increase in accuracy was found.

Age may have value in an updated PreSS model as it may help to further increase the models accuracy. Age was examined in all three studies. In Bergin’s main study, the size of the data set that had age recorded was quite small and only had two of the four institutions represented, also in this dataset the
age was recorded as a dichotomous value thus in these experiments age did not yield any significant increases, which was expected due to the quality of the data set.

Gender as a factor in this study (ID 12) corresponded with Bergin’s findings, which showed no significant increase in accuracy when added to the PreSS model, but did have value when predicting the performance of only female students, when the hours spent playing computer games factor was replaced with gender. Unfortunately when the hours spent playing computer games factor was removed, the prediction accuracy for male students decreased. Thus if a suitable replacement factor for the hours spent playing computer games could be found then the introduction of the gender factor may have some real value specifically for predicting the performance of female students.

5. S15 additional survey experiments

5.1. Research Design
The authors had collected additional factors that were hypothesised to add value to the PreSS model (Appendix 1) during two studies S14 and S15 respectively. Using the added 12 questions, it resulted in 25 experiments. Only the survey from S15 was used, as the survey administered during the S14 study was administered towards the end of the introductory module, thus was regarded as the pilot survey. With the use of the PreSS# web based system for data collection, no student data entry had missing or dirty data, so all the 26 students from S15 were included in each experiment. As with the further analysis of Bergin’s main study, 10FCV was used to obtain the prediction accuracy. A Students t-test was used to examine if a significant difference was found in accuracy between the baseline result and the experiment result as per section 3. In each experiment the accuracy was recorded.

5.1. Experiment Results
Table 4 shows the experiments where the findings were of value or showed a significant increase in prediction accuracy which is then discussed in section 6.

<table>
<thead>
<tr>
<th>Experiment ID</th>
<th>New Model Attributes</th>
<th>Sample size (N)</th>
<th>Accuracy Increase %</th>
<th>P value</th>
<th>t value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and age (actual age in years)</td>
<td>26</td>
<td>7.7</td>
<td>&lt; 0.001 ***</td>
<td>15.175</td>
</tr>
<tr>
<td>2</td>
<td>Mathematical grade, programming self-esteem and gender.</td>
<td>26</td>
<td>3.85</td>
<td>&lt; 0.001 ***</td>
<td>7.204</td>
</tr>
<tr>
<td>3</td>
<td>Mathematical grade, programming self-esteem, hours spent playing computer games and age (dichotomous age value for mature and non-mature students)</td>
<td>26</td>
<td>3.85</td>
<td>&lt; 0.001 ***</td>
<td>7.204</td>
</tr>
<tr>
<td>4</td>
<td>Mathematical grade, hours spent playing computer games and the amount of time spent on social media.</td>
<td>26</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4: S14 and S15 study additional survey factors that may have shown to have value where
* Significant at p < 0.05; ** Significant at p < 0.005; *** significant at p < 0.001;

5.2. Summary & Discussion
From a total of 25 experiments conducted on S15, 3 factors have emerged, that added value to the PreSS model using some variation of factors resulting in increases in the accuracy of the model or noteworthy findings.

When age was added as a factor to PreSS on S15 where age was represented in years, experiment ID 1, from Table 4, and in dichotomous form, experiment ID 3, from Table 4, both produced significant increases in accuracy over the original PreSS model. The authors believe that this may have value as an additional factor in the PreSS model based on prior experience. That is that mature students
may rate their programming self-esteem lower than its true value, resulting in an false positive prediction, due to this skewing PreSS’s strongest performing factor, programming self-esteem. Research has shown that there is a positive relationship between age and programming ability/attainment, (Morrison and Murphy-Hill 2013). Therefore age may be able to increase the performance of the programming self-esteem factor if it is shown that mature students may negatively influence this predictor, and age is an appropriate factor to correct this.

The addition of gender did not yield an increase in accuracy when introduced into the original PreSS model. However, experiment ID 2 (Table 4), yielded a significant increase in accuracy at ~ 4% on the S15 dataset. Given the uptake of female students in computer science is low, any research that leads to a greater understand or identifies factors in predicting success for female students warrants further investigation.

Although the time spent on social media when added to the PreSS model factors, experiment ID 4, Table 4, did not return an increase in accuracy, in nearly all combinations; it did however produce an interesting outcome when it was substituted for the programming self-esteem factor. It resulted in an equivalent level of accuracy as the original PreSS model. This result in itself may not be of much value when trying to increase the accuracy of the model, but this direct input of hours spent on social media replacing programming self-esteem, may be masking some measure of programming self-esteem. Also for a web based system, if this hypothesis holds true, the use of social media as a factor may speed up the real time predictions, as the programming self-esteem factor requires statistical pre-processing in the form of PCA, whereas social media factor raw value is a direct input into the naïve Bayes algorithm.

6. Conclusions

It is well acknowledged that an early prediction system of student performance in an introductory programming module would be invaluable to educators for developing methodologies and administering interventions as early as possible. PreSS to the author’s knowledge (including an extensive review of the literature), is one of the highest prediction systems developed with nearly 80% accuracy, administered at the earliest stage (of four weeks into the module) which has also been validated over a decade. Other prediction models that include the use of early aptitude tests, examinations or programming tests have only shown value at around eight weeks into the module at the first midterm or have not shown any significant prediction value at all (Porter & Zingaro 2014; Tukiainen & Mönkkönen 2002; Evans & Simkin 1989). Even with the high prediction accuracy of nearly 80%, the focus of this study was to identify factors that were recorded in Bergin’s main study and in S14 and S15, that were not included in the PreSS model for various reasons but may have value as an additional factor or a replacement factor in an updated PreSS model that may improve upon the 80% prediction accuracy.

With each of these factors investigated and accuracies recorded, it must be noted that the results presented must be interpreted with caution. The main concern for this is due to the size of the data sets used in some experiments and the period in which they were conducted. That said some factors have made some significant increases in the accuracy when combined with the PreSS model. Some hypotheses have been made about the contributing reasons for the presented increase in accuracy.

1. Final grade predicted by the student at beginning of the module

The replacement of the modified Rosenberg Self Esteem factor with a students predicted final grade at a very early stage of four weeks into the module, produced the largest increase in accuracy of just under 9%. The author’s hypothesis is that this has a direct relationship to a student’s programming self-esteem. A noteworthy point is when the modified Rosenberg self-esteem factor was added to the model with the student’s final grade prediction, the models accuracy only increased by 7.2%, cautiously suggesting that this may be a conflicting or superior measure of a student’s programming self-esteem.
2. Age
The addition of age to the model produced the second largest increase in accuracy of just under 8%. This may again be related to programming self-esteem, as the authors believe from experience that mature students generally rate their own programming self-esteem lower than the actual value, with recent research confirming that mature students in fact find it easier to engage with programming related task compared to non-mature students (Lockood et al. 2016), and that there is a positive relationship between age and programming attainment (Morrison and Murphy-Hill 2013). This factor seems to assist the programming self-esteem attribute suggesting that it may correct the under rated value that some of the mature students submit to the model.

3. The difference in the amount of hours a student spent playing computer games before the course to the amount of hours a student spends playing computer games during the course
The hours spent playing computer games during the course has already been included in the PreSS model, but when this factor is replaced with the difference in hours a student spent playing computer games before the course commences to the hours spent playing computer games during the course, the accuracy in the model increase by 7.5%. The author’s hypothesis is that this could be related to the effort that a student may put into the course or how much time they may spend studying or practicing programming outside of the normal course hours.

4. Social Media
Even though the hours spent per day on social media did not improve the accuracy of PreSS, it produced one of the most interesting findings as it was able to replace the programming self-esteem factor while keeping the accuracy at just under 80%. If this factor could replace the modified Rosenberg self-esteem questionnaire that requires four weeks of programming exposure before administration, this factor could allow the PreSS model to be used before the commencement of the course, also as it is not programming specific like the current modified Rosenberg questionnaire, it may also allow the PreSS model to be used in other education or industrial domains. This would make PreSS not only universal in an introductory programming course, but in a much large education / industrial space. Secondly this could significantly contribute to research currently in this space if a direct relationship is confirmed, that is that the time spent on social media may be indicative of a student’s programming self-esteem or that students with lower self-esteem gain more from online social media maybe forming a relationship with the time they spend using it. This hypothesis aligns with current research in this space (Steinfield et al. 2008; Shaw and Gant 2002).

These early results warrant a further study with the inclusion of all of these factors on a large longitudinal study, to validate these findings.

7. Epilogue
Based on this work, a large scale study has commenced in the academic year of 2015-16. This international study is currently being conducted in 11 institutions, with the aim of improving upon the PreSS model, with hypothesised increases in the model’s accuracy with the introduction of the identified factors from this research. Thirty four questions are being used in the study with over 700 participants. A study of this magnitude will be of significant value to the community and may provide substantial evidence for the work presented here. PreSS# was used to administer the study (Quille et al. 2015).
8. References


Appendix 1

The additional survey questions administered during the S14 and S15 studies:

1. Age Bracket?
2. Gender?
3. How many hours per day would you play computer games on a mobile device?
4. If you play games on a mobile device, what genre of games do you play the most?
5. How many hours per day would you play computer games on a console, PC or laptop?
6. If you play games on a console, PC or laptop, what genre of games do you play the most?
7. How many hours per day would you use the internet (not including social media or messaging services)?
8. What would your primary use of the internet consist of (not including social media or messaging services)?
9. How many hours per day would you use a social networking service?
10. If you do use social networking, what particular service do you use the most?
11. How many hours per day would you use a messaging service?
12. If you do use messaging service, what particular service do you use the most?
Abstract
Children are now learning programming as early as in primary school or even earlier. The various programming constructs and patterns of use involved in coding require different levels of cognitive development, and young children are not ready to tackle all levels yet. It is well known that some aspects of programming are accessible to young children. We suggest that even more aspects may be accessible through the use of storytelling. In the course of this discussion, we survey literature relevant to the question of how storytelling and teaching programming can be mutually supporting. Then we describe relationships between six forms of programming and five levels of narrative development. Finally, we discuss three prominent issues regarding children’s coding and storytelling.

1. Introduction
Computer science education is expanding in primary and secondary schools in many countries, such as the US and the UK. There is a great deal of interest in having more children learn computer science earlier for a variety of reasons. One reason is that it promises to be a subject that will see a lot of job growth in the coming decades, and exposing children at an earlier age makes them comfortable with key topics and more likely to pursue the subject as adults. Another reason is that coding is an avenue towards computational literacy (Wing 2006).

There is some debate as to what age is appropriate for beginning computer science instruction. Duncan et al (2014) argued that multiple factors should contribute to the decision, including what tools are to be used and whether knowledgeable teachers are available. Young children may simply not be at a stage of development that allows them to grasp some computing concepts. However, young children can understand some basic concepts of computing, and these concepts have parallels with storytelling. In this paper, we examine the obstacles that prevent young children from understanding key concepts and then discuss several projects that combined storytelling and basic programming at early ages. Then, we examine the stages of cognitive development related to narration and how they may support teaching basic programming concepts. Finally, we consider three related issues: how school curricula should incorporate children’s coding, the incorporation of game elements into storytelling and coding, and criteria for an ideal programming environment for children to use in storytelling.

2. Children’s Limited Powers of Abstraction
In Piaget’s theory of cognitive development, children reach different levels of abstract reasoning ability as they age (Piaget and Inhelder 1969). Children begin life in the sensorimotor phase and pass through the preoperational and concrete operational reasoning phases before arriving at the formal operational reasoning phase, which starts at approximately eleven years of age.

Lister (2011) mapped these Piagetian stages to programming abilities and stated that full programming ability is not present until the final stage of development, the formal operational stage. Specifically, Lister applied these stages to Neo-Piagetian theory, which states that individuals go through the four stages of development numerous times, regardless of age, as they encounter new problem domains. While any individual can have trouble learning to program at any age, this theory suggests that typically developing children will not be able to fully program before the age of eleven.

This notion is supported in programming education research as well. Rader, Brand and Lewis (1997) tested children in grades 2 to 3 and 4 to 5 on their ability to program by demonstration in the Cocoa language. The complete list of tasks included individual actions, rule order comprehension, picture matching, object interaction, subroutines, and properties. The majority of 2nd and 3rd graders could not complete anything past the individual actions, while the 4th and 5th graders understood all but...
properties. Our interest is in children younger than 2\textsuperscript{nd} graders, but if older children are unable to grasp these concepts, younger children will almost certainly be unable to grasp them too.

While full programming may be out of reach for young children, researchers have still devised and tested programming environments for them that use some aspects of programming. Morgado (2005) discusses these systems in great detail in his thesis work. We will highlight only some of them below.

TORTIS is one of the earliest examples, developed during 1974-1976. Radia Perlman created TORTIS as a tangible programming system that grew out of the LOGO project (Morgado 2006). LOGO itself was a children’s programming language by Papert and others but required some ability to read (Papert and Solomon 1971; Papert 1980). In TORTIS, children built programs out of physical blocks that would then control a robot that could move forward, backward or rotate, lower and raise a pen, toggle a light, and beep. The TORTIS environment supported command repetition and up to four user-defined functions. Because TORTIS was entirely icon-based, children as young as three used and enjoyed the system.

Similar to TORTIS, Microworlds JR is an icon-based implementation of LOGO (LCSI 2004). Children construct programs by clicking and dragging icons onto a program track. Through commands, children control an on-screen turtle, and can instruct it to move, draw, stamp its shape, play music, and more.

In Cocoa, TORTIS, and Microworlds JR, students control character movement directly or through commands. Both are limited in their ability to support storytelling, however. Other systems, such as Alice (Conway et al. 1994), or Kokopelli’s World, a system we ourselves are developing (Thompson at al 2016), more explicitly support storytelling, such as with dialog support and custom animations. Before we discuss storytelling environments, however, let us look at the value of storytelling.

3. The Value of Storytelling
Children enjoy stories from a very early age, and storytelling as a form of instruction has many benefits. People pay more attention to stories than other text and are better able to remember information in the form of a story (Willingham 2004). The power of storytelling has already been featured in several computer science education research projects. Kelleher and Pausch (2007) used storytelling to motivate girls to learn coding in the Alice environment. Howland and Good (2015) explored the use of storytelling and natural language in programming in their work with Neverwinter Nights and the FLIP language.

Older computer-based storytelling environments for children include Spinnaker Software’s Story Machine (InfoWorld 1983) which immediately animated story events typed in English on the keyboard, Programming by Rehearsal, which combined a drama metaphor with graphics and the SmallTalk language (Gould and Finzer, 1984), and PLAY, which provided an iconic scripting facility with an animation facility using the drama metaphor (Tanimoto and Runyan, 1986).

A few research projects have both targeted young children and used storytelling as a vehicle for programming.

4. Young-Age Storytelling Programming Environments
Scratch Jr. is an attempt to adapt the blocks-based Scratch programming environment to K-2 students (Flannery et al 2013; Resnik et al 2009). It focuses primarily on story creation. Scratch Jr.’s design goals were to offer an environment with a low floor and a high ceiling that can engage students at a young age but still support creative solutions as more complex languages do. Programming in Scratch Jr., as in Scratch, involves connecting blocks together to form sequential commands. Blocks’ labels are icons, to enable pre-literate users to interact with the system. Most commands involve the manipulation of one of several on-screen 2D sprites, and nearly all have immediately visible effects. These sprites represent the characters children use to tell their stories. Stories/programs can also be shared between students, and Scratch Jr. offers classroom support for teachers to facilitate multiple students using the language simultaneously.
ToonTalk is an earlier example of a system designed for young children (Kahn 1996). ToonTalk provides a constraint-based system and presents itself like an open-ended Lego sandbox. Children program agents by adding blocks, all represented by icons instead of text, to a work space and programming by demonstration. While ToonTalk could in principle be used to tell stories, it presents itself more as a game where users can manage a town and create their own games to share with friends.

Magic Forest is another icon-based programming environment that allows users to edit or create new games by placing “stones” (commands) inside of “scrolls” that are attached to objects (Andrade 2007). The scrolls dictate the object’s behaviour. Like ToonTalk, Magic Forest presents itself as a game making and playing environment more than a storytelling platform. Storytelling is an integral part of game-making, however, and tools to build games can also be used to craft stories.

The role of animated graphics in storytelling environments is important both as a means of semantic feedback to children and as a source of humour and richness. This could be seen even within the relatively primitive environment of Spinnaker Software’s Story Machine (InfoWorld 1983). With more modern graphical affordances such as those described by Pahud (2016), one can expect children’s motivation to be stronger.

Storytelling has been used by these systems primarily to contextualize and motivate programming actions. From a young child’s perspective, storytelling and programming may be quite similar tasks. There are separate theories on childhood narrative development, however, and thus there may be something to be gained from using these theories to re-examine how we present programming tasks to children.

5. Narrative Development Levels
Stadler and Ward (2005) presented five levels of narrative development they observed while listening to stories told by 3-to-5 year old preschool children: labelling, listing, connecting, sequencing, and narrating.

In the labelling stage, children primarily talk about objects or character that they clearly have at their disposal. There are few actions and no expression of story flow.

   No, that’s not my cat. That’s my cat. That’s her cat. This is, and this is bee. Here’s my girl.

In the above example, the cats, bee, and girl are mentioned but not related to each other. Neither do any of the characters perform any actions.

In the listing phase children’s stories typically resemble long lists, but start to include attributes and actions of objects.

   My picture is a XX. And it have, and it has kids with music. And there’s some guy who’s teaching them how to do music. And then trying to make it. Some of ‘em are not listening cuz that one’s who’s being, like (gestures) are doing that. This one’s doing that. And so he broke the wire with the call the phone. (claps) He break it and the guy’s drinking some soda. And they’re doing their music concert. And the end.

Here, connecting words appear, such as “and” and pronouns (it). Verbs are being used as well, like “teaching”, “broke”, and “drink”.

In the third stage, connecting, objects and actors more commonly interact with one another, though there is still little to no sense of temporality until the fourth stage, sequencing.

   I have a garden by my house. And, it, um, I have a dog. And my dad puts her poop in the garden. Yeah, because that’s the only place we can put it. So he puts it in the garden. And we have some little pink flowers growing in there. And, um, they, um, my grandma and gramp came over. And they were going to check one day. And then we saw those red flowers and they were blooming. And, um, um, my mom always
goes to the garden. And she takes a watering can and waters them so they grow. They grow, but not too often in the spring.

In the connecting phase, verbs more commonly have subjects and objects, like “my dad puts her poop in the garden.”

Sequencing sees the introduction of stories that exhibit temporal ordering or cause and effect relationships.

On my birthday, I was holding my cat. And then my Mama took a picture with my brother holding it. And I was holding his head. And it was Jessica, my big sister’s cat. And her name is Callie. But she doesn’t have front nails. And she’s very little, because Cindy took her to the doctor. And then the doctor cut all her nails out. But it didn’t hurt at all. She couldn’t feel a single thing.

Here, the child is able to stay focused on one topic for the whole story, and the last three sentences are each a result of the preceding sentence.

In the last stage, narrating, children begin planning their stories on higher levels, and incorporate relationships such as foreshadowing. At this level, children have to understand what their story is going to say from beginning to end.

As she looked up, she saw her fairy godmother. And the fairy godmother said, “‘No wonder you’re so sad. I must make you a coach.’” And she did. And Cinderella said, “‘Don’t you think my dress?’” “‘It’s wonderful!’” her godmother said. And she looked again. “‘Oh, good heavens, my child, you couldn’t go in that.’” So Bibbety, Bobbety Boo. There stood Cinderella in the most perfect gown. And Cinderella said, “‘This is wonderful. It’s like a dream.’” And the prince danced with the charming Cinderella. And the king said, “‘That prince danced with that girl all night. So I think that means he found the girl that he wanted to marry.’”

In this example, the child had a clear notion of how the story would conclude from the start, and events flow logically and consistently.

If children conceive of programs and stories in the same way or similar ways, then there may be value in teaching programming concepts in a way that mirrors the levels of narrative development. Here we note that Stadler and Ward are not the only ones who have presented theories of cognitive narrative development. Their model could be considered an extension of earlier work done by others, including Applebee (1978) and Vygotsky (1962). Vygotsky’s stages in particular have several similarities to Stadler and Ward’s stages. However, to avoid clouding the discussion, we only examine one of these models in detail, and choose to focus on Stadler and Ward’s as it is the most modern. The next section adopts a definition of programming and shows how forms of programming could map to these levels of narrative development.

6. Relating Programming to Cognitive Narrative Development

Blackwell defines programming in broad terms in “What is Programming?” (2014). While Blackwell’s definition is by no means the only one, his definition attempts to address more varied and modern forms of programming, such as end-user programming, while still remaining broadly applicable. He describes programming as composed of five types of activities: requirements, specification, design, coding, and debugging. As mentioned by two of our reviewers, Blackwell’s definition of programming does not specifically include one type of activity particularly common among young programmers: recoding, also referred to as reuse or remixing. Dasgupta et al define remixing as “the reworking and combining of existing artefact’s” (2016).
Table 1 shows a matrix of narrative development levels (with one for each column) and programming activities (with one for each row). We use the letter $x$ to mark potentially interesting relationships, while letters $a$ through $h$ mark relationships we feel are supported by prior studies.

<table>
<thead>
<tr>
<th></th>
<th>Labelling</th>
<th>Listing</th>
<th>Connecting</th>
<th>Sequencing</th>
<th>Narrating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>a</td>
<td>x</td>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>Specification</td>
<td></td>
<td></td>
<td>x</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td>d</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td>f</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Debugging</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>Reuse</td>
<td>x</td>
<td>h</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 - Relationships between cognitive narrative development levels and programming activities.

Program Requirements
Blackwell defines the requirements component of programming as “decid[ing] the intended result of executing the program.” This includes specifying the overall purpose of a program, along with its key stakeholders and gross functionality. Similar steps occur when creating a story, even at low development levels like labelling or listing. At these stages, children are still clear about who or what is in the story, and what the story will literally be about; for example, “This is a story about my cat.” Requirements rarely involve the “how” and as such are a good fit for early narrative stages, which rarely involve actions or intent.

(a) For Flannery et al in their paper on Scratch Jr. a habit they commonly observed in both Scratch and Scratch Jr. is for children to create lots of characters with no scripted behaviour (2013). This occurred most often with younger, less experienced students who were lost or not engaged. This sort of behaviour is similar to what one would see in the labelling or listing stage of narrative; creating numerous characters or nouns with little attention spent on actual actions.

(b) There is also a higher-level component to requirements: One has to consider one’s audience. Knowing what an audience wants requires a degree of empathy that younger children may not be capable of until later stages of development, such as the narrating stage. A study of 4th graders using the LaPlaya blocks-based environment found that they rarely created programs for anyone other than themselves (Hansen et al 2015). The study did not prompt them to make programs for others, so it may be that they are capable of doing so, but it does not appear to be something they do instinctively even by the 4th grade.

Program Specification
Specification involves “identifying when [the program] will be executed, and allowing for variation in different circumstances.” At the specification stage one must decide what form the program’s input and output will take, as well as the overall relationship between that input and output. Context becomes more important as well. In the narrative levels, actions start to appear in the listing and connecting phases, and cause-and-effect relationships appear in the sequencing stage. The relation between inputs and outputs cannot be properly understood before familiarity with the cause-and-effect concept.

(c) Rader, Brand and Lewis, when observing the 2nd and 3rd graders who were unable to use Cocoa to perform complex commands, found the children could describe high-level ideas but were unable to break those down into a series of actual actions the system could recognize. Neither could the children specify the correct order for a series of actions. These students may have been succeeding at the requirements level but failing at the specification level. Since they also could not sequence commands correctly, the sequencing narrative-development stage may be an upper limit for specification.
Program Design
The design stage is the one in which the programmer “chooses from a set of technical features that may support [the desired] behaviour.” At this stage “how” becomes the central question. Choosing an effective design strategy requires one to be familiar with multiple strategies already, and be able to evaluate which is most effective. From the narrative perspective, this requires one to be aware of the relationship between actions (sequencing) and to have a clear notion of the purpose of the story (narrating).

(d, e) Herbert Simon (1969) popularized the notion of design as a complex, scientific process. We can consider design as a form of complex problem solving. Design has a very high skill ceiling that can accommodate an unlimited degree of advanced reasoning. As such, design activities most likely map to higher levels of narrative development as well.

Coding
Coding can be as straightforward as “entering abstract control commands as well as data.” But as described in the previous sections, coding can involve a wide range of abstract reasoning capabilities. We argue that actions involved with coding can map to any stage of the narrative development process.

(f) A study examining the habits of children using Scratch found that children greatly prefer a bottom-up programming approach, where they immediately started executing commands and then edited them until they got the desired behaviour (Meerbaum-Salanit et al 2011). Experience with this and other systems already discussed, such as TORTIS, indicates that children are capable of performing coding tasks at a low level of abstraction without much planning. Coding has a high ceiling, however, so particular tasks map to the narrative development levels depending on their degrees of complexity.

Debugging
In debugging one must “anticipate and account for departures from the intended behaviour.” Because one must already have a program, and have a clear sense of what the intended behaviour is, debugging is a fairly high-level activity. As such, we feel it falls more in line with sequencing and narrating narrative stages. A notion of cause and effect is essential to discovering what is causing a bug, and fixing a bug requires a complete understanding of one’s program or story to avoid adding new errors in the process.

This is an idealized vision of debugging behaviour, however. In practice, beginning programmers, and especially children, may be able to identify bugs without being able to reason through them yet. Children may blindly change code in an attempt to resolve an issue they do not understand, or follow coding standards told to them but which they do not yet fully appreciate. While these sorts of debugging methods should not be encouraged, they are more likely to map to earlier stages of narrative development.

(g) A survey of UK Code Clubs, with over 2200 children responding in total, found that children were least confident in their ability to debug code over general programming, variables, conditionals, Booleans, and broadcasting (Smith 2014). These were children who had completed an average of six Scratch projects. That children at this level were the least confident about their ability to debug suggests it is at least perceived as one of the more advanced types of programming behaviour.

Reuse
Retelling is a fundamental aspect of all storytelling, and as such reuse can appear in many forms throughout narrative development.

(h) Naïve forms of reuse can be seen at early stages of programming and narrative. In the labelling and listing stages, children seldom create new characters, but rather include entities they know from their own lives or from other stories they have heard. Even in later stages of narrative development, children are copying techniques and themes they have seen elsewhere before (Hill and Monroy-Hernandez 2013).
7. Discussion
The matrix above highlights the many potential relationships between narrative phases and forms of computer programming. Now, we raise three broader questions that arise as consequences of attempting to integrate storytelling and the teaching of programming.

As educators and technologists attempt to empower children to program at earlier ages, an important question is how school curricula should change to accommodate this. As we have attempted to show, there are potential synergies between storytelling (and therefore language arts more generally) and coding. Arguments can also be made for creating synergies between coding and other subjects: mathematics, science, art, music, and physical education. By focusing first on achieving an integration of coding and storytelling, we may be able to begin to design new curricula at the beginning of the child's formal education and work up from there.

How does interactivity fit into storytelling? Game-creation has been a popular approach to computer science education in recent years, and often students would rather create story-like experiences with interactive possibilities than traditional single story-line narratives. What are effective ways to combine the benefits of storytelling and game-making in the context of computing education?

What would an ideal programming environment for storytelling look like? Perhaps there is no single environment that could be both general enough and tailored to the special needs of storytelling that it could be called ideal. However, as we've suggested in this paper, designers ought at least to consider a theory of child development in terms of narrative, and work out how children could use their system effectively at various development stages. Ideally, such a system would meet the following criteria: fosters programming and narrative skills simultaneously; makes programming feel as natural at an early age as storytelling; is accessible to each child regardless of individual skill level in programming or narrative; and accommodates any child’s reading ability, from illiterate to fluent.

8. Conclusion
We have discussed some of the issues that prevent young children from becoming full programmers, as well as several systems intended to empower children in spite of their deficiencies. We also examined systems that use storytelling as their main approach. This direction deserves additional research. We have explored the potential relationship between the development of children’s narrative skills and six forms of programming activity. A system that could leverage the power of storytelling and advance a child’s understanding of both programming and narrative simultaneously could be a powerful educational tool. We are currently developing one such tool, called Kokopelli’s World, and describe it elsewhere (Thompson et al. 2016).

9. Acknowledgements
The authors would like to thank the reviewers for their helpful comments. This research was supported in part by grant P50HD071764 from the Eunice Kennedy Shriver National Institute of Child Health and Human Development (NICHD) at the National Institutes of Health (NIH). Thanks also go to Virginia Berninger for leading our center and encouraging the development of Kokopelli’s World.

10. References


Human language and its role in reference-point errors

Craig S. Miller
School of Computing
DePaul University
cmiller@cdm.depaul.edu
July 30, 2016

Abstract
A reference-point error occurs when a programmer writes code that mistakenly refers to one element when the intention is to refer to an element structurally related to it. I review these errors and their relation to the use of metonymy in human communication. Using a working example, I draw upon cognitive theories of human communication and problem-solving to explore three accounts of why these reference errors occur in novice programming. The first account involves a deficient mental model, the second assumes a misconception of the notional machine, and the third considers implicit, proceduralized habits of communication. I conclude with learning objectives for students that address these sources of difficulty.

1. Introduction
Human language has long served as a basis for analyzing the errors and misconceptions of novices as they learn to program (e.g. Clancy, 2004; Bonar & Soloway, 1985; L. A. Miller, 1981). In many cases, novice mistakes involve conflating the human language meaning of a word with its meaning in the programming context. For example, Spohrer and Soloway (1986) discuss how the English meaning of the word ‘OR’ may be misapplied in the construction of a Boolean expression. Other researchers have taken a broader view. For example, Tenenberg and Kolikant (2014) discuss how practices of human communication in general may shape novices approach to programming. Despite these efforts, less understood is the underlying cognitive mechanisms for which human language may shape novice performance when learning to program. There is some uncertainty as to whether language knowledge is a cause of the mistakes or whether it too is just a reflection of underlying mental representations.

The goal of this paper is to explore possible mechanistic relationships between human language and novice programming behavior. It does so with a pointed focus on just one aspect of programming, namely that of reference specification. Moreover, it just focuses on one type of error, albeit one that has been repeatedly identified and then systematically studied. Here the novice behavior involves reference-point errors and their parallel to metonymy in human communication. Metonymy in human language occurs when the speaker explicitly names one element with the intention of referencing a related element (see Lakoff & Johnson, 1980, for an overview of metonymy in the context of other figurative language). In previous work I have presented how reference-point phenomena in novice programming matches those in human communication (C. S. Miller, 2014) but have yet to explore possible sources of the phenomena in detail.

In this paper, I draw upon cognitive theories of human communication and problem-solving to suggest plausible accounts of why reference errors occur in novice programming. Of course, reference errors that parallel the use of metonymy only represent a very small portion of novice difficulties as they learn to program. Yet, reference errors have been identified as a significant source of difficulty for students (Goldman et al., 2008). Perhaps more importantly, the clear focus allows us to explore possible theories in greater detail and work us toward a more coherent and systematic theory. If successful, it may provide some direction for analyzing other categories of novice errors.

2. Metonymy and Programming
A reference-point error is an expression that refers to an entity that is not the programmer’s intended target. Often the intended referent and the actual referent are elements in a computer data structure, but
they could also be file references, database elements or conceptual constructions. While all programmers may make reference-point errors, scholars have noted that novice programmers have particular difficulty with them. For example, Du Boulay (1986) describes “confusion between the subscript of an array cell and the value stored there.” Holland, Griffiths, and Woodman (1997) describe cases where students conflate a whole object and an identifying attribute of the object. In these cases, a student may write a coded reference that indicates an attribute when the specification required a reference to the object itself. Reference-point errors have also been noted in exercises where students are asked to give precise instructions to the instructor acting as a robot (Davis & Rebelsky, 2007).

In previous work, I analyzed reference-point errors in terms of metonymy (C. S. Miller, 2014). Metonymy is a rhetorical device used in human-to-human communication, often to emphasize a particular attribute or to aid identification. For example, consider the phrase: “Open the ice cream and serve two scoops.” In this phrase, the intention is not to literally open the ice cream but rather the container holding the ice cream. The speaker may be emphasizing the ice cream in order to distinguish it from sorbet or some other frozen dessert. The container does not need to be mentioned since the recipient of the request can readily infer that it is the container of the ice cream that needs to be opened.

Use of metonymy in human language parallels some reference-point errors produced by novice programmers. Consistent with the observations by Holland et al. (1997), students are more likely to specify an attribute in place of an object (and vice versa), when the attribute is an identifying property such as the name. Elsewhere I have presented multiple experiments that systematically produce this effect and discussed its relationship with metonymy (C. S. Miller, 2012, 2014). Finally, other accounts (Ragonis & Ben-Ari, 2005; Vahrenhold & Paul, 2014) note student confusion between an object’s identity and its attributes, although these observations are not framed in terms of human-to-human communication.

The goal of this paper is not to provide a comprehensive account of reference-point errors, nor even an account of all such errors based on metonymy. Rather, we use this kind of reference error as a particular case that has a clear specification, directly parallels metonymic constructions in human language and is documented in diverse reports. This direction does not just consider cognitive mechanisms but allows us to explore external factors that shape mental models and habits of communication, considerations advocated by Tenenberg and Knobelsdorf (2014). Similarly use of language may influence how students construct their mental models (Holmboe, 2005; Diethelm & Goschler, 2015).

2.1. Working Example
In this section, I will provide an example for further reference throughout the paper. While the syntax has been simplified to make it more analogous to diverse programming contexts, the construction is derived from previous study (C. S. Miller, 2014). The working example will also illustrate the phenomena presented in this study (also consistent with other reports Davis & Rebelsky, 2007; Holland et al., 1997).

The example assumes a set of objects (described by attributes and values) and an API for manipulating them. This API provides a simple retrieval of an object based on an attribute and value of the object. The retrieved object can then be added to a collection object. Below is an example of its correct use:

```python
obj = catalog.find("name", "peach")
cart.add(obj)
```

Note that the retrieval requires the explicit specification of the attribute (i.e. “name”) and its value (i.e. “peach”). The retrieved object can then be passed to the `add` method as a parameter. For now, we assume that the API has been reasonably presented to students, although, as we shall see, as students commit a reference-point error, they are essentially incorrectly using the API.

While the API is not the programming language, it offers instructions that students need to understand for successful programming. In this context, the API arguably extends what Du Boulay (1986) calls the notional machine. As Sorva (2013) explains, the notional machine includes what students need to learn for successful programming in a particular context. Later in this section we will see an example that does not require an API for eliciting similar reference errors.
Given the API, a reference-point error occurs when the student tries to add the item by using one of its attributes (e.g. name) instead of the object itself. Below is such an example:

```python
cart.add("peach")
```

In another version of the error, a student may try to access the object using the attribute as a principal identifier and thus omit the explicit reference to the attribute name. In this example, the student may be using the string “peach” as the fundamental means for accessing the object rather than realizing that “peach” is just another attribute value for the object.

```python
obj = catalog.find("peach")
cart.add(obj)
```

As already discussed, students are more likely to commit these reference-point errors when the attribute is an identifying property (e.g. name, title, label) instead of a descriptive property (e.g. color, texture) as correctly demonstrated below:

```python
obj = catalog.find("color", "yellow")
cart.add(obj)
```

Note that this initial presentation of the working example does not provide the whole context for which it would be applied. For example, we have (so far) not considered how students are taught that the add operation requires an object reference. Also, the task instructions may bias the student towards a reference-point error. Let us consider the following task instructions:

1. Add the peach to the cart
2. Add the peach object to the cart
3. Add the object whose name is peach to the cart

The first of these task instructions is effectively using metonymy while the remaining instructions indicate a distinction between the object and the attribute. Presumably students would be more likely to commit a reference-point error with the first of these task instructions. However, previous work reports students making these mistakes even when the task instructions explicitly indicated the attribute and its value such as the wording found in the last task instruction (C. S. Miller, 2014).

An alternate version of the task may ask students to write a general function for adding by name any object to the cart. Below is a correctly coded example:

```python
def addToCard(item, cart):
    obj = catalog.find("name", item)
    cart.add(obj)
```

Different contexts such as this task may produce different results and our analysis should provide some insights on how student answers may vary depending on the context.

While the working example in this paper uses an API, reference difficulties may occur without an API. Below is an example, where the task is to write a function that serially searches an array of objects, returning true if there is an object that matches its name attribute:

```python
def isNameInList(item, list):
    list.each do |obj|
        if obj.name == item
            return true
        end
    end
end
```
This next example shows this example with a reference-point error consistent with the use of metonymy:

```ruby
def isNameInList(item, list)
  list.each do |obj|
    if obj == item
      return true
    end
  end
  return false
end
```

In this example, the intended referent is the `name` attribute for the `obj` object, but the code refers to the whole object in its place. While reference-point errors have yet to be studied extensively in this non-API context, a forthcoming paper (C. S. Miller & Settle, 2016) reveals similar reference-point errors including those consistent with previous findings.

3. Overview of theories

The source and even the definition of metonymy in human communication is subject to some debate. For example, some scholars question whether metonymy represents a shift in reference or simply involves a shift in meaning (Rebollar, 2015). More generally, such as in the construction of noun phrases, there are opposing views as to whether speakers deliberately construct a reference to aid the recipient in efficiently identifying the referent or whether its construction is more a product of what is active in the speaker’s working memory (Gatt, Krahmer, van Deemter, & van Gompel, 2014).

For this paper, the goal is not so much to resolve these debates but to explore possible sources for the errors seen in programming. I nevertheless draw upon the diverse theories of metonymy in human language and possible mechanisms that underlie it. Such treatment will allow us to consider how these mechanisms might relate to the reference-point errors we see in novice programming. These mechanisms thus hypothesize one source or even a combination of sources when a student constructs an incorrect reference.

As we look at sources, we will look beyond the cognitive mechanisms employed by the student, and include background knowledge and context, which arguably inform student behavior (Tenenberg & Knobelsdorf, 2014). As we shall see, all draw upon external domain knowledge that is arguably involved in creating and resolving metonymic references for human communication (Croft, 1993).

3.1. Mental Models and Verbal Reasoning

We first consider how a student’s mental model of an object may lead to reference-point constructions that parallel the use of metonymy. A mental model is person’s conceptualization of how a system or artifact works in the environment. Norman (1983) has shown how differences between a human user’s conceptual model and an accurate working model can account for errors in the context of human-machine interaction. Similarly, Sorva (2013) discusses how differences between a student’s mental model of the notional machine and an effective conceptualization of it can account for novice programming errors.

In this paper, I propose possible differences between a student’s model and an effective model in order to account for reference errors. I draw upon mental model theory as it has been applied to deductive reasoning in the form of syllogisms (Johnson-Laird, 1986; Polk & Newell, 1995). In these problems, people are given premises (e.g. “All clowns are artists”, “no mechanics are artists”) and asked to produce a valid conclusion (e.g. “no mechanics are clowns”).

Johnson-Laird (1986) theorized that people work with situation models and their ability to produce valid conclusions depends on their ability to reason with multiple alternative models. Failure to consider alternative models accounts for many of the errors observed in the studies. In a departure from the
Deficient model
peach | color: yellow, size: large, shape: sphere
apple | color: green, size: medium, shape: square
pear | color: green, size: large, shape: pear

add peach to cart

Effective model
obj1 | name: peach color: yellow, size: large, shape: sphere
obj2 | name: apple color: green, size: medium, shape: square
obj3 | name: pear color: green, size: large, shape: pear

add obj1 to cart

Figure 1 – Two mental models encoding the task of adding an object.

Johnson-Laird account, Polk and Newell (1995) hypothesize that people construct situation models as an immediate product of processing linguistic input. Characterized as “verbal reasoning,” the ability to draw valid conclusions depends on successfully encoding and reencoding the premises in an annotated situation model. Both the mental model theory of Johnson-Laird and the verbal reasoning theory make use of situation models instead of logical inference rules to account for human performance.

A consequence of verbal reasoning is that people often make assumptions that are not warranted or produce models that are incomplete. Moreover, their structure may facilitate particular inferences or operations while omitting other possibilities, unless further encoding or reformulation occurs. Since people have well practiced routines constructing mental models as they comprehend linguistic content, they are likely to apply such routines to diverse domains. Here I explore how a plausible encoding of a computer programming task could lead to the reference-point errors presented in our working example.

In the style of Johnson-Laird’s models, Figure 1 depicts two mental models that encode the example problem from the previous section. They include both the goal (i.e. “add the peach to the cart”) and a model of various objects, one of which is the targeted object. The first (deficient) model would produce a reference-point error by referring to the object by its name rather than the object itself. The second (effective) model is a possible reencoding, which properly distinguishes between the name of the object and a reference that refers to the whole object.

Let us consider the first, deficient model in more detail. While the actual details would vary among students and contexts, application of the model could lead to a reference-point error if it has the following characteristics:

1. Each object consists of a list of values (e.g. yellow, large), each with a named attribute.
2. Each element has a singular value that primarily identifies the object from a set of objects.
3. The identifying value does not have an explicitly named attribute.

Technically the identifying value (e.g. ‘peach’) is just another attribute of the model, but its privileged representation could lead to a reference error. If students applied this model, they may write code that just references attribute and not the whole object:

```python
cart.add("peach")
```

Following Polk and Newell’s verbal reasoning hypothesis, the construction of this deficient mental model is likely a consequence of human language and communication. First, the name of the attribute is significant. Knowing its identifying role gives it a privileged state as reflected in the model. Second,
the model may have also been constructed by presenting objects using metonymic constructions. The following examples use metonymic constructions to reference objects in the domain:

1. In this example, we will add a peach to the cart.

2. This literal representation of the pear object shows that it has the attributes of color, size and shape.

3. Note that the peach and the pear have the same size.

In all cases, the name attribute is used to refer to the object without explicitly indicating the attribute. A literal interpretation would then lead to the construction of the first (deficient) model show in Figure 1. More generally, because we routinely use metonymy to describe our world, this language may produce mental models that reflect the literal interpretations of task instructions. Reference-point errors result as a straight-forward application of literal interpretation.

Of course a student may construct a more accurate model that correctly distinguishes between the identifying attribute and the object itself. Such is the ‘Effective model’ in Figure 1. This model more accurately represents how objects are actually constructed. From this construction, a student could reason that the name is one of several attributes that does not technically identify the needed object for the operation of adding to the cart (although an API could be designed and implemented to allow such an operation). The presentation of this particular model commits to naming the connecting symbols (i.e. ‘obj1’, ‘obj2’ and ‘obj3’), although this detail may be unspecified in a hypothesized working model.

3.2. Communication design

Another possibility is that the student knows that the name is organized by attribute but believes that the computer can resolve a reference that omits the relationship between the object and the identifying attribute value. In this case, the student’s mental model may explicitly distinguish between the name attribute and the reference to the entire object. Yet, the student knowingly refers to just the name value with the view that the identifying nature of its name allows the computer to effectively infer that the whole object is intended.

Here the novice programmer may be intentionally employing principles of communication such as those presented by Grice (1975). Perhaps most relevant is Grice’s maxim on Quantity, where the amount of information should be no more or less than what the recipient requires. If the student believes that the system can effectively resolve the reference without explicit mention of the attribute, then it is a reasonable communication principle to omit the attribute. Moreover, for human communication, the so-called literal expression is not necessarily more efficient than an expression based on metonymy. It has been well argued that the literal interpretation is not necessarily processed first by human listeners (Recanati, 1995).

Knowingly referencing the attribute in place of the whole object suggests that the student believes that some provision has been made so that the computer can successfully resolve the attribute reference to the object itself. Pea (1986) calls this type of mistake a “hidden mind superbug” if the student performs as if the computer has a human-like mind that can successfully infer the intention of the student programmer. Pea warns that students do not necessarily believe that the computer literally works like a human mind. In fact, if asked, the student may disavow a hidden mind. For this reason, Pea suggests that this bug may be a product of unconscious knowledge, something we consider in the next section.

The explicit assumption that the computer can infer the intended referent may seem naive, unless we consider that designers of computing technology often consider the likely intentions of its users or programmers. In this specific case, it would not be unreasonable to provide a method in the API that gives identifying attributes, such as name, a privileged method for referring to objects when adding them to the cart. In any case, as Sorva (2013, p. 8:7) notes, “the novice needs to learn what the notional machine does for them on one hand, and what their responsibility as a programmer is on the other.”
3.3. Acquired habits of communication

We finally consider the case where a student relies on implicit communication knowledge for constructing the reference. In this case, the student may possess an effective mental model (such as that presented in Figure 1) and have no (explicit) expectation that the computer can successfully resolve the point of reference. Instead, the reference error occurs by drawing upon implicit, procedural skills, plausibly obtained through the practice of human-to-human communication. At one time, the source of this practice may have been the same as used to produce Grice’s maxims (Grice, 1975) or processes for selecting attributes, such the computationally efficient method proposed by Dale and Reiter (1995). In such cases, the student does not need to be aware that he or she is drawing on these practiced skills.

A proceduralized application of knowledge can be modeled with a production-based system. Productions are associative rules that match internal representations (including those that correspond to mental models) and perform an action, either to the internal representations or as a perceptual/motor operation. Production systems have been offered as cognitive architectures, such as Soar (Newell, 1994) and ACT-R (Anderson, 1993). In these models, productions can be acquired through deliberate practice. Once acquired, explicit access may not be available.

Here I consider a simple production-based model that would produce a reference-point error. Figure 2 depicts a production that selects an attribute from an object and uses it to write code. The elements in brackets (e.g. `<label>`) are variables and can match any symbol. Its condition matches one object with any attribute and the goal of adding that object to the cart. The production is very general and could match in many ways. However, assuming an activation-based process for selecting a production, the production only matches the attribute with the greatest activation and directly places it in the expression.

If the model assumes that identifying attributes are most salient, carrying the greatest activation, it accounts for why reference-point errors occur more frequently with identifying attributes than those that are just descriptive. In a model, salience could be based on perceptual properties as well as any preprocessing that highlights the attribute. Such preprocessing may draw upon attribute selection algorithms such as those proposed by Dale and Reiter (1995) in human-to-human communication.

This third account does not depend on a particular mental model—the production could match either representation in Figure 1. It also does not assume any explicit understanding of the notional machine. Instead, it relies on default habits, plausibly acquired or at least reinforced in human-to-human communication.

4. Discussion

The three accounts of reference-point errors stem from three different sources: a deficient mental representation, a misunderstanding of what the system can do, and a reliance on implicit habits for communication. Since these sources are not mutually exclusive, it is possible that student mistakes arise from any of them, given the right context.

A goal for future work is the development of strategies for diagnosing which account is producing a reference error for any given circumstance. For example, it may be possible to manipulate task instructions to encourage the construction of an effective mental model. If a defective mental model is a root cause, better instructions should reduce the frequency of errors. As another example, priming selected elements may elicit different habits and thus show support for the third account.

In the absence of an effective strategy for identifying the source of errors, the three accounts nevertheless...
suggest necessary prerequisites for successfully producing a well-formed reference. These accounts allow us to devise learning objectives for students. For successfully constructing a reference, students need to learn the following:

- Fully encode attribute/value pairs for modeling an object.
- Acquire a rigorous model of the notional machine.
- Obtain practiced routines (i.e. productions) for extracting the appropriate elements from the mental representation.
- Employ a validation step to ensure proper alignment.

The first three correspond to each of the three accounts. Fully encoding the object in terms of its attributes and values gives students explicit access to the identifying attribute name. Understanding the limits of the notional machine—the API in our example—informs their obligation to fully specify the reference. Obtaining a practiced routine ensures proper inclusion of the needed attribute label and increases its likelihood to be selected among other competing habits. This practice also enables students to appropriately chunk components to make better use of working memory (Soloway & Ehrlich, 1984). Finally, the last learning objective could be deployed as a defense against any of the error accounts.

Experimenting with instructional interventions that focus on any particular objective may also provide indirect evidence of where students have most difficulty. For example, an exercise may ask students to draw representations of the objects and check them against correct answers. If such an exercise improves student performance on specifying references, it suggests that the underlying mental model had been at fault.

Another issue for further work is whether these accounts are relevant for other disciplines. As an example, Zandieh and Knapp (2006) discuss the role of metonymy in mathematical understanding. They report that a student may express a derivative as the tangent line rather than the slope of the tangent line. Like the analysis here, instructional materials may have led students to develop a deficient mental model that ultimately affects their ability to solve problems. On the other hand, unlike almost any other discipline, problem solving in computing requires human-to-machine communication, an activity whose difference from the human-to-human kind can cause its own problems.

5. References


Programming language theory: Thinking the unthinkable (Work in progress)

Tomas Petricek
Computer Laboratory
University of Cambridge
tomas@tomasp.net

Abstract
Our thinking is shaped by basic assumptions that we rarely question. Such assumptions exist at multiple levels. Foucault’s episteme grounds knowledge within a particular epoch; Kuhn’s research paradigms determine how scientists of a given discipline approach problems and Lakatos’ research programmes provide undisputable assumptions followed by a group of scientists.

In this paper, we attempt to uncover some of these hidden assumptions in the area of programming language research. What are some of the hidden assumptions that we never question and that determine how programming languages are designed? And what might the world look like if we based our thinking or scientific method on different basic principles?

1. Incommensurability
Science is often distinguished from other human activities by its progressive nature. It has standards for identifying improvements and methods for accumulating them into an ever improving body of sound knowledge. Such traditional view has been challenged by many philosophers. The most significant blow to the idea of progressive science is incommensurability—the idea that two theories do not share a common basis that would allow evaluating them using a common metric. Incommensurable theories do not only have different basic assumptions, but they also ask incompatible questions.

In this paper, I first describe different sources of incommensurability described by philosophers and I review how the incommensurability surfaces in programming language research. Finally, in the most speculative part of the paper, I consider what might theories incommensurable with current main-stream programming language research look like, what questions would they ask and attempt to answer.

1.1 Episteme and human knowledge
Michel Foucault’s concept of episteme captures an incommensurability at the most fundamental level of human knowledge. An episteme defines the assumptions that make human knowledge possible in a particular epoch. It provides the apparatus for separating what may from what may not be considered as scientific. In other words, it determines claims about which we can say whether they are true.

Foucault gives an example of the incommensurability between the earlier episteme of the Renaissance and the later classical episteme. The former is characterized by signs and resemblances while the latter is characterized by ordering and categorization. Natural historian Comte de Buffon (of the classical episteme) refers to the work of Ulisse Aldrovandi (of the Renaissance episteme):

Buffon was to express astonishment at finding in the work of a naturalist like Aldrovandi such an inextricable mixture of exact descriptions, reported quotations, fables without commentary, remarks dealing indifferently with an animal’s anatomy, its use in heraldry, its habitat, its mythological values or the uses to which it could be put in medicine or magic.

As Foucault explains, Aldrovandi’s report was not incorrect. It linked things in accordance with the system of signs and similitudes, which was incommensurable with the system of ordering and categorization that was assumed by Buffon due to the classical episteme. This quote gives a vivid example of the effect that grounding in a different episteme gives – Aldrovandi’s work attempts to answer questions that Buffon (or modern reader) cannot even conceive.4

---

1 For an extended discussion, see e.g. Niiniluoto (2015)
2 Two most influential philosophy of science works introducing the idea are Kuhn (2012) and Feyerabend (1993)
3 Foucault (2001), the quoted example appears on page 43
4 The classical episteme of de Buffon is incommensurable with our modern thinking, but for other reasons

223
1.2 Paradigms and research programmes
Foucault’s episteme considers the whole of human knowledge and may appear too remote for a discussion about particular field of modern computer science. However, related concepts in philosophy of science capture common assumptions in a given scientific discipline or sub-group of scientists.

According to Kuhn\(^5\), normal science is governed by a single paradigm. The paradigm sets standard for legitimate work within the science it governs. A paradigm is not explicitly defined. Instead, it is formed by the methods and assumptions that a scientist learns during her training. When a paradigm becomes insufficient for solving ordinary scientific problems (puzzles), a paradigm shift replaces the predominant paradigm with a new one that is incommensurable with the old one. As with Foucault’s episteme, the new paradigm considers different questions to be worth a scientific enquiry.

At a smaller scale, shared assumptions and methods used by a group of scientists have been captured by Lakatos\(^6\) as research programmes. A research programme recognizes that, even in regular science, some laws and principles are more basic than others. Faced with an experimental failure, a scientist never blames such hard core assumptions, but instead addresses the issue by modifying some of the additional assumptions provided by the protective belt of the theory. Due to the different hard core assumptions, the work arising from different research programmes is to some degree incommensurable.

2. Episteme and paradigms in programming
Do episteme, paradigms and research programmes affect how programming language research is done? In this section, I consider two examples that answer the question in affirmative.

2.1 Mathematization of computer science
Mathematical methods are a foundational part of modern computer science. However, the history of the discipline suggests that this was not a logical necessity. Early programmers were often seen as anything from clerical workers to chess players and artists. In fact, early study noted that majoring in mathematics was not found to be significantly related to performance as a programmer.\(^7\) The focus on mathematics was, however, a good tactical move for early academic computer science:

*The rise of theoretical computer science was anything but inevitable. (...) Advocates of theoretical computer science pursued a strategy that served them well within the university, but increasingly alienated them from their colleagues in the industry.*\(^8\)

Mathematization of computer science established it as a legitimate academic discipline and differentiated it from industrial computer engineering. An essential part of the development was the concept of algorithm, which provided aspiring scientists with a practical agenda for advancing the discipline. Using the terminology of Kuhn, computer science became a normal science that is preoccupied with puzzle solving activity. The newly founded research paradigm determines which questions are scientific (various questions about algorithms) and how answers should be sought (through formal methods).

It is hard to imagine computer science where algorithm is not a foundational concept, but explaining that as a necessity would be misleading. Rather, a historical coincidence made algorithm a core part of our paradigm. It is even harder to question the idea that mathematics can find relevant answers to the questions posed by programming. This is because mathematization has become a part of our modern episteme. But how might a computer science look when our episteme or research paradigms change?

2.2 The Algol research programme
Paradigm shifts are rare and the change of episteme even more so. At a smaller scale, much of the modern academic programming language research has been influenced by the Algol research programme. The goal of the programme is to utilize the resources of logic to increase the confidence in the correctness of programs.\(^9\)

\(^5\) Kuhn (2012)
\(^6\) Lakatos (1978)
\(^7\) Ensmenger (2012), p129
\(^8\) ibid., p117
\(^9\) ibid., p257
The Algol research programme defines the hard core, together with a sufficiently open ended agenda. Indeed, much of the theoretical programming language research aims to prove programs correct through the use of formal logic. The methods differ, but the hidden assumption that formal proof provides the correct methodology is widely shared. Any experimental failures (such as the fact that proving programs correct is still difficult after 50 years of the existence of the Algol research programme) are attributed to protective belt – we do not yet have sufficiently powerful formal methods, the problem is not properly formally specified and programmers in the industry are not using the right tools.

3. Thinking the unthinkable

I outlined how programming and science more generally are affected by assumptions that are implicit in research programmes, scientific paradigms and also the episteme of the current period. I considered concrete examples from programming language research to illustrate that those are not just abstract philosophical concepts.

As suggested by Ensmenger\textsuperscript{10}, the establishment of theoretical computer science rooted in mathematics was not an inevitable development and it is conceivable that computer science would evolve differently, building on principles other than algorithms and formal logic. Similarly, in programming language research, the predominant Algol programme is not the only one. A largely incommensurable research programme was defined by Smalltalk where “programming was not thought of as the task of constructing a linguistic entity, but rather as a process of working interactively with the semantic representation of the program, using text simply as one possible interface.”\textsuperscript{11}

The most interesting aspect about research programmes, scientific paradigms and episteme is that the competing programmes or paradigms and episteme that replace the old ones are incommensurable with the old ones. This means that they do not share the same assumptions, goals and ways of thinking. The rest of the essay speculates on what might programming research arising from a different paradigm or episteme look like.

3.1 Taxonomies of programming ideas

Theoretical computer scientists attempt to extract mathematical essence of programming languages and study its formal properties.\textsuperscript{12} Now consider an episteme that instead aims to explore the design space and build a taxonomy of objects that occupy the space.\textsuperscript{13} It considers the entities as they are, rather than trying to extract their mathematical essence. What would be the consequence of such way of thinking that attempts to relate and organize programming ideas in taxonomies, rather than abstracting?

In programming language research, many novel ideas that defy mathematization are left out because they are too “messy” to be considered through the predominant formal perspective. If the episteme made us seek relationships, those would all become within the realm of computer science. For example, we would be able to discover similarities between live coded music and formula editing in Excel\textsuperscript{14} as both of those represent a form of programmer interaction with immediate observable feedback.

Science should not merely observe, but also “twist the lion’s tail” and conduct experiments to probe the properties of the nature\textsuperscript{15}. If our focus is on building taxonomies, the nature of relevant experiments will also differ. Rather than measuring properties of simple models, experiments designed to reveal relationships need to highlight interesting aspects of a behaviour in its full complexity. They need to reproducibly demonstrate relationships that are not immediately obvious, similarly to how thought-experiments in sciences and philosophy highlight an intriguing aspect of a theory. An interesting format that appeared recently is the presentation of programming research in the form of screencasts.\textsuperscript{16}

\textsuperscript{10} Ensmenger (2012)
\textsuperscript{11} Priestley (2011), p294
\textsuperscript{12} An occasional attempt to treat programming differently has been made, for example, by Noble (2002)
\textsuperscript{13} This way of thinking is similar to Foucault’s classical episteme that focused on the construction of taxonomies
\textsuperscript{14} Thanks to Sam Aaron (author of live-coding environment Sonic Pi) for numerous inspiring discussions on this topic
\textsuperscript{15} The quote has been attributed by many philosophers of science to Francis Bacon, founder of modern scientific method
\textsuperscript{16} This is the format used by the Future Programming workshop (\url{http://www.future-programming.org})
Another consequence of the focus on taxonomies in the classical episteme was the creation of museums, which present the studied objects neatly organized according to the taxonomy. A computer scientist of such alternative episteme might follow similar methods. Rather than finding mathematical abstractions and presenting abstract mathematical structures, she would build (online and interactive?) museums to present typical specimen as they appear in interesting situations in the real-world.\(^{17}\)

### 3.1 Crossing the vertical gaps with metaphors

Does modern computer science have something to gain from the Renaissance episteme centred around signs and resemblances? Work on programming languages is done at three layers. At the top layer, language or library designs follow some intuitive ideas, which are turned into an actual program (middle layer). Formal reasoning is done about a simplified mathematical model, which is the lower layer. Programming research operates horizontally – relating different formal models or various implementations – but does not easily cross between the layers. Theoretical work at the bottom is rarely linked with the informal top layer.

Signs and resemblances provide the missing link between layers. In object-oriented design, an object is a metaphor for an object in the real-world,\(^{18}\) but metaphors can be found in many areas of programming. We use them to conceive an idea and use our intuition at the higher level to guide design at lower levels. Since our episteme does not consider such resemblances important, we then hide them (for brevity, or out of disinterest) from our published narrative. Metaphors often remain present only through naming. In John von Neumann’s First Draft of Report on EDSAC\(^{19}\) (which described modern computer architecture) individual units are called “organs” suggesting a biological metaphor for the system structure. Even more obvious metaphors are hidden in the plain sight. For example, when and why did we start calling programming languages “languages”?

If resemblances and metaphors played fundamental role in our scientific thinking, we would not just gain interesting insights from them, but we would also ask different questions (which appear secondary or unscientific when considered through the perspective of our current way of thinking). What research agenda would computer scientist (or a programming literary critic) of an episteme centred around metaphors ask? Looking at the problem of concurrent programming as an example, one might try to design programming language models for concurrency by studying how simultaneity is expressed in different forms in narrative.\(^{20}\) What are the postmodern alternatives to threads where concurrent processes occurring in the same temporal interval are described in successive textual parts of the narrative?\(^{21}\)

### 4 Conclusions

Overall way of thinking that is captured by episteme or paradigms has been changing throughout the history and we can expect it to continue changing. Ideas grounded in different episteme or paradigms are often incommensurable, meaning that they have different basic assumptions and consider different questions as scientific. Just like Buffon was astonished when reading the work of Aldrovandi, we may wonder which of our current scientific achievements will appear as exact anatomical descriptions and which will appear as fables, and magic to the thinkers of the future.

Were we to think about programming in terms of taxonomies, much of the present work that explores interesting aspects of, say, programming language design space will still be relevant. The readers might be astonished why we focus on irrelevant technical details rather than trying to present the most unique interesting aspects of the work in their full richness. Were we to think about programming in terms of similarities and metaphors, formal mathematical models will become just one of many forms of metaphors available when understanding programs, but future thinkers might be astonished by our attempts to find more and more remote abstractions that gradually lose more and more of the similarity with the original idea and become merely a work of art or fiction.

---

17 The Design Patterns book by Gamma et al. (1994) can perhaps be seen as an example of this approach
18 Noble (2002)
19 von Neumann (1945)
20 Sedlacek (2011) looks at economics (mathematicised similarly to programming) through a perspective based on myths
21 This unsophisticated style of narrative is mockingly referred to as “Meanwhile, back on the ranch.” See Margolin (2016)
5. References
Gamma, E., Vlissides, J., Johnson, R., Helm, R. (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley
Towards spreadsheet tools for end-user music programming

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

Abstract
SheetMusic, an early-stage prototype, explores how spreadsheets can be used as accessible end-user tools for music programming and data sonification. This design probe uncovers many interesting questions: what are the primary advantages of the spreadsheet paradigm in this context? Should such a tool be regarded as a musical instrument, or as a way to create musical ‘programs’ with emergent runtime behaviour? How can musical experiences be ‘programmatic’? How sophisticated should provisions for scripting the tool be? How can time be represented? Each issue is considered in turn, drawing on previous work in live music coding, end-user programming, and the current SheetMusic implementation.

1. Introduction

SheetMusic is an exploration of how formulae with musical sound effects can be integrated into the spreadsheet paradigm. Figure 1 illustrates the current, early-stage prototype. Apart from the play/pause controls, it is largely indistinguishable from a regular spreadsheet. Indeed, the prototype can be used as a plain spreadsheet without any musical capability at all. It is implemented as a web page, and each cell can contain arbitrary Javascript code, consequently all native Javascript functions are available, and further libraries can be loaded in from the console.

However, this spreadsheet is also imbued with music and sound-generation capabilities, by virtue of a simple library of note and sequence synthesis functions built on top of the web audio API. The design of this library is one of the central, emergent, and open design questions raised by this exercise. Currently, functions are available which play a specified note (e.g., `p('c 5')`), or a sequence of notes (e.g., `s(['c 5', 'g 4', 'a 4', 'e 4'], 1)`), or which loop a sequence of notes defined by a range on the spreadsheet (e.g., `pLoop('viola','B1','B8')`). As with regular spreadsheets, arguments can be passed directly or can be references to other cell values. Further library functions provide easy access to arrays containing the notes of chords and scales. A global `tick` variable increments...
once every tick; the time elapsed between ticks can be set by editing the global `tempo` variable. At each tick, the spreadsheet is re-evaluated. The `tick` variable is accessible in the spreadsheet, so the formula `if(tick%2==0) p('snare') else p('kick')` would produce a simple drum beat consisting of alternating kick and snare sounds. Cells are also re-evaluated upon edits, as in a regular spreadsheet. This implementation of time borrows directly from spreadsheet stream processing convention, and will be discussed in greater detail.

The precise specification of music/sound-generating library is still emergent. As each cell can be used as an environment for arbitrary Javascript code, it may be argued that the library is irrelevant as the user can embed any desired functionality directly into the spreadsheet, or inject it into the runtime environment through the browser console. However, rich standard library support (or the lack thereof) is as much a part of the experience of programming in a certain language as the language syntax itself. This fact is not lost on the programming community; standard library support is what allows, for instance, the programming language Python to boast of having “batteries included”. Consequently, future versions of SheetMusic will still retain the property of being arbitrarily-scriptable, but it should be possible to create sophisticated musical programs without extensive custom Javascript code, showcasing clearly the advantages of the spreadsheet paradigm.

This direct use of spreadsheets provides two major advantages for music coding, beyond the mere fact that implementation can be live (Tanimoto, 1990), allowing for direct feedback when the program is edited. The first is that the spreadsheet paradigm is well-known to be an excellent interface for novice end-user programming. The second is that the grid formalism allows for rich secondary notation to be expressed through the layout of the spreadsheet. This is already common in business applications, where a single sheet may contain several separate ‘regions’ of cells with strong semantic connotations, separated by blank cells or highlighted in different colours, even though this separation is unnecessary for the correct functioning of the spreadsheet.

2. Related work

Manhattan (Nash, 2014) is a grid-based music sequencer which shares many properties of spreadsheets. In particular, the interface is laid out in rows and columns of cells, where cells can contain a number of object types, including formulae. Where it deviates from spreadsheets is that some of the layout is used to enforce a notion of time; each column represents a parallel stream of execution, and going downwards through rows within a column indicates the flow of time; lower rows are ‘after’ higher rows in the sequence. This basic control flow can be enriched using standard programming constructs such as conditional branching and loops, through formulae. Unlike spreadsheets, each cell can contain multiple formulae. Other grid-based languages, such as Piet and Al-Jazari (McLean, Griffiths, Collins, & Wiggins, 2010) exist, but these are unlike spreadsheets, as the ‘cells’ of the grid cannot freely store data and code interchangeably, and control flow is intimately linked to the layout of the cells. Visual dataflow languages such as Texture (McLean & Wiggins, 2011) also have some commonalities with spreadsheets.

Sonic Pi (Aaron & Blackwell, 2013) is a live coding platform for music, implemented as an embedded Ruby DSL, with sounds generated by the SuperCollider synthesis server. As novice end-users form part of the core audience for Sonic Pi, it is coupled with an IDE meant to mitigate the complexity associated with programming. Sonic Pi is nonetheless a textual programming language, with the control flow of music corresponding directly to the control flow of program execution. Other textual languages include Impromptu (Sorensen & Gardner, 2010), Tidal (McLean, 2014), and ChucK (Wang, Cook, et al., 2003).

3. Programming language or musical instrument?

A central design question is whether SheetMusic is intended as a composition tool, a musical instrument, or a programming language. This is a false trichotomy (Blackwell & Collins, 2005), but is a simplified expression of the question: *what sorts of artefacts are users expected to produce with this tool?*
I would argue that variable output is the defining characteristic of a program. A program whose output is fixed is merely a clever way of compressing that output data. In live music programming, it is possible to represent a musical piece by explicitly coding individual notes as a sequence, but this is not the standard approach. Instead, programming constructs such as data structures, conditionals, loops, and subroutines are employed, which has a number of advantages: it makes the structure of the piece explicit, it compresses the notation required to represent the piece (reduces verbosity), and increases the flexibility of the program (reduces premature commitment). That being said, if the output of running this program is fixed and independent of input, it is essentially equivalent to a hard-coded sequence of notes. It is more interesting to consider cases where the musical output of a program is variable, and dependent on its input. A program whose output is largely unaffected by a change in its input is less ‘programmy’ than one whose output is affected more. A closely related concept is cyclomatic complexity (McCabe, 1976), which quantifies the number of linearly independent execution paths through a program.

Moreover, I would argue that the defining characteristic of a musical instrument is a player with intent and agency to musically affect the output of the instrument. Consequently, live music programming is an instance of playing a musical instrument, as even though the intermediate source code instances may be ‘fixed’ programs with invariable output, the player has the agency to change the program. This distinguishes musical instruments from mere playback mechanisms; we would not consider someone pressing the ‘play’ button on a record player to be playing a musical instrument, but someone hitting ‘play’ and ‘pause’ rapidly in order to create a slicing effect has made a musical instrument out of the record player through intent and agency to musically affect the output.

SheetMusic could be used to store a sequence of notes (used as a composition environment), or play a sequence of notes with agency (used as a musical instrument). However, since it also has the potential to be used for designing music as an interactive, reactive, data-dependent experience (used as a programming language), it is this latter use case that the design will be focused towards.

4. Applications

Two immediate application scenarios present themselves. The first application scenario is custom sonifiers for data; a few SheetMusic formulae could instantly create ‘auditory scatterplots’ or line graphs, where data values are mapped to pitches and played in rapid succession – known to be highly perceptually effective for several types of analysis (Flowers, Buhman, & Turnage, 2005). Pitch mappings could be tailored specifically to the data domain (e.g., a change in octave or key at some domain-specific critical threshold). This would enable simpler multi-modal data analysis, as well as improve accessibility to data in spreadsheets for the visually impaired. Another application is as a prototyping tool for music in interactive games such as role playing games. Music and sound are often linked in complex ways to game state, from trivial applications such as sound effects for player actions, up to more complex applications such as selecting soundtracks with different moods depending on the player’s location or status in the game world. SheetMusic could be used to compose, test, and share such state-dependent musical experiences, as a subset of game variables could easily be captured as cell values which are then readily available for musical interpretation through the library functions.

5. Representing time

In SheetMusic, time passes independently of the spreadsheet, communicating its current value to all the formulae in the spreadsheet once per ‘tick’. Notes and effects relying on sub-tick durations can currently be expressed through stretching/squeezing and offset parameters. However, since this is an inconvenient notation, it is expected that just as in setting the time signature and tempo of a musical score, the tempo will be adjusted so that the duration of a tick corresponds to the smallest duration required to concisely capture the majority of the piece. For instance, a musical piece set to a tempo of 40, but consisting mainly of quavers, might be recast more ‘comfortably’ as a piece at tempo 80 consisting mainly of crotchets – this is a matter of taste, convention, and interpretation, which does not have an effect on the music denoted literally by the score.
The idea that time elapses independently of the spreadsheet layout, and that formulae recalculate either on a publish/subscribe or polling basis is the main convention for stream processing in spreadsheets, as implemented in Microsoft Excel’s native Real Time Data (RTD) feature\(^1\), as well as in several popular stream processing add-ins.

This can be contrasted against Nash’s Manhattan, which sacrifices much layout flexibility by committing columns to denote *tracks* which execute in parallel, and rows to represent *time slices* which execute sequentially going downwards. This approach was taken to build directly on the chronological grid-based design of music sequencers, as the aim was to introduce greater spreadsheet programming capabilities to users of such sequencers. SheetMusic takes the opposite approach as it has the opposite aim; to bring greater musical programming capabilities to users of spreadsheets. By decoupling time, SheetMusic frees the layout of the spreadsheet for use as arbitrary secondary notation. Griffiths’ Al-Jazari decouples time from the grid differently – in that system, agents are programmed to explore the grid and ‘play’ cells they occupy. McLean’s Texture is a 2D playground which employs proximity to infer control flow – syntactic elements placed close together are automatically connected.

### 6. Conclusion

This paper has presented an exploration of spreadsheets as end-user music programming tools, illustrated through an early-stage prototype called SheetMusic. The primary advantages of the spreadsheet paradigm in this context are direct manipulation, liveness, and the ability to use the layout of the spreadsheet as secondary notation. SheetMusic can be regarded as a musical instrument, but has potential as a tool for creating highly data-dependent musical ‘programs’ with emergent runtime behaviour.

### 7. Acknowledgements

Thanks to Alan Blackwell and Luke Church for discussions on the topic. The author is supported by an EPSRC industrial CASE studentship sponsored by BT Research and Technology, and also by a Robert Sansom scholarship from the University of Cambridge Computer Laboratory.

### 8. References


\(^1\)https://support.microsoft.com/en-us/kb/285339
A gaze-directed lens for touchless analytics

Abhishek Chander  
Computer Laboratory  
University of Cambridge  
ac839@cam.ac.uk

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

Abstract
We present the design and implementation of a gaze-directed lens tool for the interactive exploration of quantitative chart visualisations. In view of visual analytics as end-user programming, we present lenses which enable the selective display of labels, as well as interaction with parameterised uncertainty. We describe an algorithm for smoothing the movement of the lens. We report a controlled, within-subjects experiment on 11 participants demonstrating that participants were as adept at moving the lens with their gaze as they were using a mouse, and furthermore, that the gaze-directed interface promoted greater inspection of the key regions of the graphs.

1. Introduction
Visual analytics can be viewed as an instance of end-user programming. Here, the “programs” being written are not represented as textual source code, but rather the instances of charts, graphs, and other visualisations which arise as (often transient) results of exploring a dataset using a visualisation tool. The process of interacting with the tool is analogous to the act of programming, in the sense that each transient visualisation embodies a procedure for transforming data.

Traditional mouse-and-keyboard interfaces, or even touchscreen interfaces, are problematic in a number of situations. For instance, shared public display walls, where each individual user cannot be given mice and keyboards, and touchscreens cannot be implemented for reasons of cost and robustness; or operating theatres, where sterility is a primary concern (O’Hara et al., 2014; O’hara, Harper, Mentis, Sellen, & Taylor, 2013; Perry, Beckett, O’Hara, & Subramanian, 2010); or where the target end-user has impaired motor skills which prevents them from using conventional input mechanisms. In the era of ubiquitous analytics, similar situations demand tools for the visual exploration of data. In the touchless scenarios previously described, eye movement-based interaction provides a promising solution (Jacob, 1990). Lenses are interactive overlays which serve a variety of functions for data analytics, including zooming, filtering, manipulation of representation, and changing the level of detail (Tominski, Gladisch, Kister, Dachselt, & Schumann, 2014). Consequently, it seems appropriate to apply eye-tracking to the movement of lenses for visual analytics.

Eye-tracking is an important approach to investigating the psychology of programming. For instance, it has been used to assess program comprehension (Bednarik & Tukiainen, 2006), syntax highlighting (Sarkar, 2015), and visual attention (Bednarik & Tukiainen, 2004) amongst other things. With respect to the view of visual analytics as end-user programming, and that eye-tracking is an important methodology for program comprehension understanding, this work makes the following contributions:

1. We present a purely gaze-directed lens tool, designed such that eye movements can be multiplexed for use either to inspect the graph or to move the lens.

2. We describe an algorithm for dynamically smoothing raw eye-tracking coordinates which provides results superior to simple exponential averaging or PID control.

3. We report the evaluation of our tool through a controlled study of 11 participants, showing that it is as easy to manipulate as a mouse-based lens and promotes inspection of key regions.
2. Related work

Much previous work has applied eye-tracking to the movement of lenses, albeit not specifically for visual analytics. A central problem from which gaze-directed analytical tools suffer is the *gaze multiplexing problem*, sometimes known as the *Midas touch* problem: eye tracking applications have to guess whether the user is reading, intends for a lens to move, or engage a selection, etc., and act accordingly. It is tricky to infer intent from eye movements alone; for instance, when trying to read a label placed along the edge of a gaze-driven lens, the user might inadvertently move the lens itself because the centre of their gaze has shifted.

To address the problem of gaze multiplexing, state-of-the-art solutions often rely on temporal multiplexing (i.e., changing modes based on dwell time) to infer intent, at the cost of interactional fluidity (Ashmore, Duchowski, & Shoemaker, 2005; Lankford, 2000). Others still use multimodal input (e.g., through head position, touch, and keyboard) to improve the experience and reliability of lens tools (Stellmach, Stober, Nürnberger, & Dachselt, 2011; Stellmach & Dachselt, 2013; Spindler, Büschel, & Dachselt, 2012; Lehmann, Schumann, Staadt, & Tominski, 2011; Kumar, Paepcke, & Winograd, 2007), at the cost of touchless interaction.

![Flat magnification lens.](image1)

*Figure 1 – Flat magnification lens.*

![Our prototype, showing a 2x magnification lens. The toolbar, which can be hidden, shows various lens parameters.](image2)

*Figure 2 – Our prototype, showing a 2x magnification lens. The toolbar, which can be hidden, shows various lens parameters.*
Figure 3 – A magnifying lens for reducing label clutter. Top: all labels turned on. Bottom: Labels shown only within lens.

3. Flat Lens Design

Figure 1 shows a magnification lens. Our magnification lens is flat; i.e., it linearly scales the underlying axes. Thus, it avoids the interpretability issues caused by fisheye lenses, which warp the underlying axes and are thus less suitable for scientific analysis (Carpendale, 1999). However, this also loses a key advantage of fisheye lenses, namely that no part of the graph is obscured, despite extreme warping at the lens edge. Apart from magnification, we also implemented a label lens (Figure 3) which only shows labels for points within the lens, reducing clutter in dense plots.

To achieve linear magnification without warping or obscuring parts of the graph, we use a two-part lens: an inner box indicates what part of the graph is being magnified, and a translucent outer box contains the magnified graph. The lens is necessarily translucent, otherwise parts of the graph would be obscured. The underlying graph is also visually de-emphasised by using a grey stroke so that the magnified graph, which uses a black stroke, is more prominent.

It is also necessary for these boxes to be contained in one other and to have a common centre, because otherwise trying to inspect parts of the magnified graph would move the lens, changing the region being magnified. With a common centre, gaze location signals the user’s unambiguous intent to magnify a region and inspect it, greatly alleviating the gaze multiplexing problem.
4. Gaze data smoothing

The eye tracking data is extremely jittery and unusable in its raw form, even if the calibration is good. When the user is staring at a fixed point on the screen, deviations in the received gaze coordinates can have an amplitude of up to 50 pixels. A lens configured to be centred around the raw coordinates in real time would be unusable.

Exponential averaging is a well-known technique for smoothing jittery eye-tracking data (Wojciechowski & Fornalczyk, 2014). However, basic exponential smoothing was unable to meet our requirements as low values of $\alpha$, the memory coefficient, produced smooth outputs, but incurred a large latency when moving the lens across the graph. Higher values of $\alpha$ caused jittery output when attempting to focus within the lens. Our implementation of a Proportional-Integral-Derivative controller with Ziegler-Nichols tuning (Ziegler & Nichols, 1942) also produced unsatisfactory results.

4.1. Dynamic Exponential Smoothing (DES)

We modified our approach to use a dynamic value for $\alpha$, scaled by the difference between consecutive values of the gaze location. In the following, $x_t$ denotes the $x$-coordinate received from the eye-tracker at timepoint $t$, and $s_t$ denotes the output of our smoothing algorithm, on which the lens is centred:

$$\beta_t = \frac{|x_t - s_{t-1}|}{\text{chart width}}$$

$$\alpha_t = \begin{cases} 
\beta_t & \text{if } \beta_t < 0.05 \\
\min(1.0, f \cdot \beta_t) & \text{otherwise.}
\end{cases}$$

$$s_0 = x_0$$

$$s_t = \alpha x_t + (1 - \alpha) s_{t-1}, \ t > 0$$

A similar set of equations (with $\beta$ dependent on chart height instead of width) determines the smoothed $y$-coordinate. The variable $f$ is an “expansion factor,” which controls how quickly $\alpha$ grows as the gaze location moves further from the current smoothed location. This technique results in smoothing behaviour which is much better than basic exponential averaging or PID control in terms of maintaining smoothness, as well as quickly acquiring distant targets.

Figure 4 shows the effects of different smoothing strategies on a trace of the $x$-coordinate of users’ gaze when looking at points on the left, centre and right of the screen. Observe how DES catches up with large shifts in user’s gaze (i.e., the user is looking at another part of the graph and so the lens must move) while maintaining stability during small shifts (i.e., the user is inspecting different parts of the lens contents but does not wish the lens to move). Simple exponential smoothing suffers either from lag or excess jitter, and PID control suffers from both lag as well as “overshooting” the mark for large shifts in gaze location.

DES, like PID, is based on the principle that large shifts in the quantity being controlled should cause the controller to accelerate faster than small shifts (i.e., second-order control), but unlike PID, DES is altered to better fit the specific requirements of gaze data smoothing. Stellmach et al. (Stellmach & Dachselt, 2013) present a similar idea, where lenses have invisible concentric ‘zones’—the lens is not moved when gaze coordinates lie within the innermost zone, allowing uninterrupted examination of the lens contents; within the intermediate zone, the lens is moved faster as the gaze drifts further away from the innermost zone; within the outermost zone, the lens is positioned absolutely. DES is an evolution of this idea, but framed in terms of the user’s gaze location instead of in terms of lens coordinates, and also provides smooth, continuous control rather than shifting control based on hard boundaries, which can introduce unpredictable threshold effects for the user.
5. A lens for interacting with approximate computations

In previous work, we have shown how it is possible to enable users to interact with and control uncertainty associated with individual data points, where that uncertainty has arisen out of an approximate computation process, such as sampling (Sarkar, Blackwell, Jamnik, & Spott, 2014). The motivation is that it can be much faster, and consequently more interactive, to rapidly render a graph based on approximate values. However, it is an open question as to how these approximate values may be interactively refined. For instance, we have previously proposed that data points may be displayed with error bars, which can themselves be dragged to adjust the level of uncertainty associated with a point; reducing the size of an error bar triggers recomputation to a higher degree of accuracy (Sarkar, Blackwell, Jamnik, & Spott, 2015).

This section describes a lens built as a tool for interaction with simulated uncertainty. The simulation can be used on any data set, because the inaccurate initial values are generated randomly from the true data. The core idea here is that the position of the lens directly indicates user attention; points on which the user’s gaze dwells are recomputed with lower errors.

Figure 4 – Comparison of DES against basic exponential smoothing (top) and PID control (bottom).
5.1. Error model

We add Gaussian noise to create errors in the true $y$-values.

\[ e_i \sim \text{Gaussian}(0, \delta(y_{\text{max}} - y_{\text{min}})) \]  \hfill (5)

\[ \hat{v}_i = v_i + e_i \]  \hfill (6)

where $y_{\text{min}}$ and $y_{\text{max}}$ are the minimum and maximum $y$-values in the dataset respectively, which scale $\delta$ to the current dataset; $\hat{v}_i$ is the approximation to $v_i$, the true $i^{th}$ value. The value $\delta$, which defaults to 0.1, can be varied to increase/decrease the maximum error throughout the dataset. This produces a “noisy” version of the dataset as shown in figure 5.

Points within the lens are recomputed iteratively. Error is reduced exponentially; in each iteration we set

\[ \hat{v}_i := \theta \hat{v}_i + (1 - \theta)v_i \]  \hfill (7)

where $\theta$ is set to some fraction between 0.9 and 0.99, with higher values corresponding to slower convergence. We stop iterating and set our approximate value to be equal to the true value after 100 iterations, or after the onscreen distance between approximate and true positions of the data point becomes smaller than half a pixel. The visual effect of this procedure can be seen in Figure 6.
6. Evaluation
We designed a controlled experiment to compare the usability of our gaze-directed lens tool versus a conventional mouse-driven interface. Our primary questions were, in the context of exploring graphs using lens interfaces:

1. Is it feasible (i.e., not inferior in terms of time required) to use a gaze directed lens as a direct replacement for a mouse?

2. Does an eye-tracking system improve exploration of a graph in terms of the proportion of time spent inspecting key regions?

6.1. Experimental design and procedure
To answer these questions, we conducted a study of 11 users, undergraduate students at the University of Cambridge, while they performed analytical exploration tasks, each separately using the mouse and eye tracker to guide the lens. The user studies were conducted using a Tobii1 X120 remote eye tracker (Figure 7). The user sat in front of the eye-tracker. The experimenter sat to the side of the user, using an external keyboard to navigate through the questions.

We selected 30 openly-available datasets of univariate time series (Government, 2010; Quandl, 2015). Using our datasets, we framed unambiguous questions such as “What is the highest ever price of oil?”, “When did the market crash?”, etc. The participant was expected to seek peaks, troughs, and inflexion points on the graph in order to answer our questions. These questions could only be answered using the lens tool, as they either required a label value to be read off with the label lens, or a subtle feature to be discerned with the magnification lens.

We annotated each graph with “key regions,” such as peaks, troughs, and inflexion points, which the user would need to consider in order to answer the question (Figure 8). We logged when gaze/mouse coordinates fell within these key regions, so as to compare investigation of key regions with the eye tracker versus with the mouse.

The level of difficulty was consistent across questions, and the question order was randomised for each participant. Each participant answered 30 questions; 15 using the eye-tracker to guide the lens, and the other 15 using the mouse. Half of the users began with the eye tracker; the other half with the mouse. This allowed us to perform a within-subjects comparison. The participants were instructed to read the question, investigate the graph and speak out their answer, upon which the experimenter would proceed to the next question.

---

1. Tobii is a company that specializes in eye-tracking technology.
6.2. Results

Our timing data was log-normally distributed (Figure 10). Using log-normalised times, a non-inferiority test (two one-sided test) gives a confidence level of 94.4% using an epsilon value of 0.2 (for testing equivalence, a 90% confidence level yields a 0.05 significance level). This demonstrates that the gaze-directed interface is strictly not inferior in efficiency when compared to using a mouse (Figure 9). In fact, on average, participants spent 0.115s less per question with the eye-tracker than with the mouse.

Participants spent a median of 2.56s more time investigating key regions with the eye tracker than with the mouse (Wilcoxon rank sum test, $p = 8.63 \cdot 10^{-5}$). Thus, the eye-tracker promotes investigation of the key-regions of the graph using the lens. However, the total time per question was lower using the eye tracker, indicating that when using an eye tracker, a greater proportion of the time spent answering the question was investigating the key regions. The median proportion of time spent investigating the regions using the mouse was 35.2%, whereas using the eye-tracker, this increased to 48%.

6.3. Evaluation of uncertainty reducing lens

To evaluate our lens for interacting with approximate computations, we conducted a think-aloud study with each user after they had completed the primary study. The users investigated using the uncertainty reducing lens on one of the datasets (namely, the dataset in Figure 5), without any prior information about the function of the lens. The users were requested to investigate the lens tool until they had formed a hypothesis regarding the function of the lens.

\[\text{http://www.tobii.com}\]
Findings
The audio recordings from 11 users was transcribed and the average time the users spent on the dataset was 3 minutes and 51 seconds. The responses of the users are categorised as follows:

1. “Error”: 5 out of the 11 users correctly stated that the tool is reducing the error, or “noise”.

2. “Averaging”: 4 out of the 11 users mentioned that the tool is performing some sort of averaging (“mean” or “aggregate”) of the points. This might be due to the fact that the points were distributed bidirectionally using a normal distribution (section 5), and upon investigation, converged to their true values which on average could mean going up or down, as in figure 5.

3. The remaining 2 users gave physical descriptions (e.g., “the points are moving around”) of what they observed as the lens was used on the data points, but were unable to give precise hypotheses of the cause of the movement of data-points.

Of our 11 participants, 5 (45%) of the users correctly identified the purpose of the lens without any prior description of approximate computation.

7. Conclusion
We view visual analytics as a form of end-user programming. We have presented the design for a gaze-directed lens for data exploration, which uses concentric, flat, translucent lenses to overcome the warping associated with fisheye lenses, which render them unsuitable for scientific analysis. We presented a design for how a lens might be used to control uncertainty arising from approximate computation. We have also presented dynamic exponential smoothing, a smoothing technique for gaze data which surpasses exponential averaging and basic PID control.

We conducted a user study of 11 participants, and found that our gaze-directed interface is as efficient as a mouse-based interface for analytical inspection of graphs. Furthermore, use of the eye-tracker promoted interactivity with the lens tool, by increasing the proportion of time spent by the user investigating the key regions of the graph. Finally, evidence from the think-aloud study suggests that the lens can be used as an intuitive interface for interacting with parameterised uncertainty.

8. References


Abstract
An important task while reading program code is finding the location of relevant sections. When reading non-code documents, readers often rely on spatial memory and indications of the document hierarchy inherent in headings to build up an understanding. However, documents involving code are structured in a non-linear way, without the benefit of headings. Using the results from an observational study, we describe a tool that uses spatial memory for finding previously read sections of code. We propose that this tool will reduce the amount of time spent navigating through code and thus assist comprehension.

1. Introduction
Previous studies indicate that programmers mainly read program code on electronic devices (Sutherland, Luxton-Reilly, & Plimmer, 2015). Electronic devices do not provide the same affordances that paper provides for reading. One such affordance is the ability to find previously read material based on what the reader remembers of the context. The approximate location of the material is an important part of this context.

Reading program code for understanding is a more difficult task than reading other types of documents. Programmers do not read a program from start to finish: instead, they first identify a starting point they think is important and then follow the flow of the program logic (Roehm, Tiarks, Koschke, & Maalej, 2012). This requires the programmer to jump around the code files and functions as they read. This non-linear reading style increases the cognitive demands on the reader (Crisp & Johnson, 2007).

In a previous study, we observed programmers reading program code on paper. From this study, we argued that programmers use annotations to support their wayfinding (Sutherland et al., 2015). Annotations serve as way-marks, allowing a reader to quickly find material they have previously read (Marshall, 1997). Using external cognition as a model to explain this, annotations serve to spatially constrain the search space, and allow the reader to offload some of the cognitive effort required (Rogers, 2004). The navigational benefits of annotations have been observed for both text-based annotations (Storey et al., 2009), and freeform annotations (Sutherland et al., 2015).

However, annotations were not the only form of navigation support that we observed. During the same study (Sutherland et al., 2015), we observed some programmers placing pages in specific locations on the surface (see Figure 1). These programmers would pick up paper from the location, read, and then return the paper to the same location. When asked, the programmers were able to tell us what was at each location. This suggests that the spatial context plays a role in code navigation.

In this paper, we describe our current work on investigating spatial navigation for program code. First, we review related work on spatial memory during reading, and how this influences comprehension. Then we include some unreported results from the previous study. Finally, we describe our current Integrated Development Environment implementation.

2. Related Work
Early studies compared paper to online reading and found a number of differences (O’Hara & Sellen, 1997). For example, paper provides many affordances not available on a computer screen (O’Hara & Sellen, 1997). One major advantage of reading on paper relates to the reader’s ability to remember things spatially. Using paper allows the reader to lay out pages in space which helps provide an
overall sense of the document structure. This allows for visualising large amounts of data, quick referencing between documents and concurrent reading of different pages. Readers are able to take in the whole page at a glance, including both what is on the page and where things are relative to each other. Having fixed layout provides explicit cues about the document. These cues supported search and re-reading activities (O’Hara & Sellen, 1997). A second major advantage is paper allows rapid, effortless navigation through the document. Readers can quickly navigate between sections using one or two hands as needed, allowing navigation to overlap with other activities (O’Hara & Sellen, 1997).

While technology has improved, screen-based reading does not yet provide the same spatial memory or rapid navigation benefits. There have been attempts to use multiple devices (e.g. tablet PCs (Morris, Brush, & Meyers, 2007)) to provide these benefits. These devices allowed some rapid navigation but are still limited. In addition, the bulkiness of the devices constrained their use.

Central to these investigations was the premise that changing the form of the text does not influence comprehension. However, studies on how people understand literature challenges this premise. Recent research suggests that moving from a fixed layout (e.g. a page) to a flowing layout (e.g. scrolling text) may fundamentally change how people interact with documents (Mangen & Kuiken, 2014). The change in format may be causing dislocation for the reader as the way-marks in the document are no longer fixed (Mangen & Kuiken, 2014). There are no studies on whether programmers are similarly dislocated when reading code.

Reading on a screen has been found to result in lower comprehension when reading for understanding (Mangen & Kuiken, 2014). High comprehension readers build a mental model of what they are reading (Cataldo & Oakhill, 2000). This model is linked to where they read things and uses the reader’s spatial memory. If the layout of the document changes then the reader is unable to rely on spatial memory to locate previously read materials (Cataldo & Oakhill, 2000). A second related reason is when people read, they encode not just the text but also the surrounding context (Long & Spooner, 2010). With paper, the reader encodes the text, its location, and the items around it. With a scrolling reader there is less contextual information to encode.

3. Study Design
In our study, we observed 13 experienced programmers as they read a short C# program (six classes) printed on paper. The participants were instructed to read the program so they could explain it to another programmer. The participants were allowed to manipulate the paper in any way they desired. The interested reader is referred to Sutherland, et al. (2015) for the full details of the study design.

4. Findings on Spatial Navigation
Nine of thirteen participants arranged stacks of paper around the desk in specific locations. By specific location, the participants could tell us what was in each location just by glancing at the top page in the pile. While reading, these participants would pick up a stack of paper and either move it to the bottom centre location or hold it in the air while they were reading. When the participant finished reading they would return the stack to its previous location.

Figure 1 shows an example of a participant’s layout. The pages around the left, top and right are the individual code files. In this example, the participant has highlighted the name of each code file. The bottom centre shows the participant current working files. The file they are currently reading is centre right; the page centre left is for working notes. Notice the space at the top: this was the location of the current file they are reading.

As this behaviour was not the focus of the study we did not query the participants about it. We only have comments from seven of the participants on their use of the surface for laying out the pages. These comments show a consistent rationale behind why they did this.

Four of the participants started distributing the pages based on a conscious choice: they deliberately arranged the stacks of paper to allow quick access. One participant stated, “I knew I needed to flick between files so I wanted to do so quickly” and “by arranging like this I know the location of each file”. Another participant stated, “I placed the files this way because I know they are related and I would need to move between them”. Thus the arrangement of the stacks allows quick navigation.
The remaining three participants did not originally layout the pages. But by the end of the study they had changed their approach. One participant stated, “Originally I kept losing my place, I knew what I needed but I had to keep searching for it. Then I started placing the pages back in same place and I could find them much faster.” Another participant stated “I didn’t care where the files were but then I figured which ones needed to be where.” Both comments show the participants found they could reduce time by having the stacks in specific locations.

We also considered the approaches taken to locate files during the task. While there were a variety of responses, we group them into three broad categories. The first group of participants were those who “knew” where each file was. These participants had chosen or memorised the location of each file and made sure they returned it to the same location when finished (two participants). The second group added annotations to the top of the first page to identify the class (four participants), either by highlighting the class name or writing the class name at the top of the page. They would then quickly glance at these annotations to identify the pile. The final group located pages by what the front page looked like (two participants). By the end of the reading task, they had a rough mental image of each front page and used these images. One of these participants stated, “I just know what it looks like now.” We did not ask the final participant how they identified each pile.

One final related observation is the space needed for reading. Five participants left a space for reading (three deliberately, one by accident and one unknown as to why). One participant stated the reason for this space was “so I can put things and it won’t mess up the other piles.” Another participant stated “I like to rest the paper but I didn’t want to accidentally pick up other pieces of paper.” The other four participants did not leave a space (three by accident and one unknown). One participant stated “why would I need a space for reading, I’m holding the paper as I read.” The other two participants said they did not make a choice to do this but just moved paper as needed.

5. Current Prototype
Spatial layout is not currently available in Integrated Development Environments (IDEs), the workbench for most programmers. However there has been some research on using spatial layout for debugging (DeLine, Bragdon, Rowan, Jacobsen, & Reiss, 2012). Also, IDEs provide a significant amount of support for code navigation with search functionality and hierarchical code displays.

In order to investigate whether this functionality would be useful in an electronic environment, we have designed and are currently developing a prototype. We have deliberately embedded spatial layout functionality within an IDE so that the IDE support continues to be available. Figure 2 shows the current prototype we are developing. The user can open multiple code editors on the display surface, close them and move them around. The current prototype embeds these editors as scrolling windows but the intention is to change them into a paging display, with an indicator as to the number of pages in the code file.

1 This was the first participant and we were not aware of this behaviour yet
To evaluate the prototype we will perform a between subject evaluation. Each participant will be asked to read and explain a small program in either the new prototype or the standard version of Visual Studio. Planned comparison variables are the length of time needed and accuracy of comprehension. We will also observe the participants’ behaviours as they read.

6. References


Comprehension and Composition of Flowcharts

Unaizah Obaidellah
Faculty of Computer Science & IT
University of Malaya, Malaysia
unaizah@um.edu.my

Abstract

The common problem of the students’ failure in programming courses is due to the lack of problem solving skills. This is regarded as the lack of ability to understand the problems, divide them in sub-portions, and integrate them as a complete solution. The main purpose of this paper is to present preliminary findings of students’ comprehension and composition abilities focusing on problem solving activities related to interpreting code into flowcharts and flowcharts into code. The group of students tested were undertaking their second programming course. Initial findings from this study showed that the students possessed relatively weak problem solving skills even if they scored well in their fundamental programming course.

Keywords: Program comprehension, flowchart, data structure, problem solving

1. Introduction

Computer programming is a fundamental course of the computer science curriculum. A trend of high drop out and failure rate in introductory computer programming courses in universities is a universal problem (Lahtinen, Ala-Mutka, & Järvinen, 2005; Mayer, 1981; Mhashi & Alakeel, 2013; Milne & Rowe, 2002; Mow, 2008; Tan, Ting, & Ling, 2009). The majority of these research works performed a survey study that focuses on students’ perceptions of difficulty in understanding and implementing both low-level (i.e., syntax, variable) and high-level concepts (i.e., OOP, debugging). A general conclusion produced from these studies showed that learning programming is a difficult task to master by novice programmers. This difficulty is attributed by various aspects including teaching methods, study methods, student’s ability, motivation and nature of programming (Gomes & Mendes, 2007; Renumol, 2009; Sweller, 1994).

The study methods adopted by many students such as incorrect study methodologies and less intensive programming exercises contribute to this difficulty. Furthermore, the student’s abilities and attitudes towards programming such as the lack of problem solving skills required to interpret a particular problem statement, and limitation in mathematical and logical knowledge necessary to transform textual problem into successful solution code, are among the factors that reduce the students’ motivation to study programming. In addition, a pessimistic mindset associated with programming that spreads among the students sets an established reputation of programming courses being a difficult subject.

Success in programming is further attributed by factors such as prior computing experience, strong mathematical skills, learning style, gender and culture (Butler & Morgan, 2007; Lister et al., 2004; Mhashi & Alakeel, 2013; Piteira & Costa, 2013; Renumol, 2009). Although, these factors, taken independently or dependently, arguably influence a student’s programming ability, many previous research outcomes indicated that one of the major reasons students have difficulties programming is due to their lack of skills to solve problems. This is demarcated by insufficient set of abilities necessary for solving problems such as abstraction, generalization, transfer and critical thinking (Gomes & Mendes, 2007). These abilities are associated with limited understanding of how a program is executed (Milne & Rowe, 2002) which includes the ability to understand a problem description, separate it into sub-portions followed by an implementation before assembling these pieces into a complete solution. Lister et al. (2004) asked individual undergraduate students to study short
computer programs and answer multiple choice questions in an experiment that examines the processes involved in producing the solutions. Their study proposed that the inadequate knowledge and skills relate to the students ability to read code rather than writing which concluded that students often fail to analyse a short piece of code. However, this result is limited to reading code and choosing from a set of pre-defined answers. This approach may limit the results as students may use different mechanism for tasks such as writing code. Thus, considering the proposed future work by Lister et al (2004), the present study aims to evaluate students’ ability reading and writing code with respect to flowcharts (see below for flowcharts consideration). Both of these tasks can be classified as program comprehension (i.e., understanding what a program does such as aims and processes) and program generation (i.e., writing program code for a given programming problem). Robins, Rountree, & Rountree (2003) proposed that these tasks operate on different models. By this, novices tended to create program plans, while experts retrieve them, requiring explicit attention during planning and problem solving among novices. Thus, Robins et al (2003) proposed some level of dependent relationship between program comprehension and generation although they noted that these tasks may not be well correlated.

Given this unclear relationship, the proposed research idea is reported as part of an ongoing study in programming education. This work-in-progress report is presented as part of the main goal to better understand novice learners’ lack of skills at solving problems in the context of advanced programming course (i.e.: Data Structure) and to provide recommendations (i.e.: learning model/framework of cognitive model) for novices to aid learning. As discussed, the preliminary focus of this study is an extension of Lister et al. (2004), where students are evaluated on their competency reading and translating flowcharts into program code and vice versa. Based on our teaching experience, this evaluation is considered important because often students report sufficient understanding of programming concepts but fails to demonstrate writing its equivalent programming code. This raise questions whether this issue is influenced by weak fundamental or advanced programming concepts, cultural impact or other unclear reasons. Thus, better understanding on this lack of skills would contribute to the development of improved problem solving strategies.

Our motivation for the preliminary study described in this report was to assess the students’ composition and comprehension skills among those who passed the prior fundamental programming course. Specifically, investigation focuses on whether these skills would benefit novice learners to facilitate the solution process of a programming problem on a data structure topic (i.e.: linked list). In this context, comprehension refers to the students’ understanding of the given flowchart or program code and recognizing its relevant purpose. Composition refers to the students’ ability writing code with respect to the given flowchart or producing flowcharts from a specified set of functional code. Thus, this preliminary study will inform about students competency level at solving advanced programming problems. The proposed evaluation is different than those reported in existing literature as the student participants are assumed equipped with relatively strong fundamental programming concepts (i.e.: variables, control structure, arrays, methods, OOP) commonly taught in the first programming courses as demonstrated by good grades acquired by majority of the students in their final examination results of the fundamental programming course.

2. Flowcharting and code comprehension

In programming, flowcharts are commonly introduced to early learners to facilitate the understanding of a problem and segments of the solution processes. Flowcharts are useful to aid the creation of solutions to problems or facilitate the construction algorithms. In addition, flowcharts would serve as a foundation in drafting solutions for a programming problem. However, this tool is less practiced in many cases especially by novice learners. Students jump straight into writing the solution code. Our observation showed that students often attempt to solve a problem almost immediately after a short contact with the problem description, even before careful analysis and understanding of the question requirement. This problem conforms with the study reported by Almeida (2004).

Another empirical study by Shneiderman & Mayer (1979) investigated the effects of flowcharts on students’ performance in comprehension and debugging tasks between two different universities. A closer inspection showed that better performance was reported for the university which training
emphasized the use of flowcharts although generally, no difference in performance was found. However, Shneiderman & Mayer (1979) concluded that flowcharts may serve either as an advantage where solution is an aid that translates the process from syntax to semantics or a disadvantage where flowcharts were taken as an alternative representation which interferes with the students existing semantic mental structure. As such, what types of supplementary representation help programming learners to build their internal semantics was raised as a question worth further investigation. Furthermore, various levels of abstraction can occur in a flowchart construct due to the richness of consideration for a programming solution. It will be interesting to investigate whether some level of generality occurs between reading code and producing the corresponding flowcharts and vice versa.

2.1 Hypothesis
It is assumed that if a student can consistently demonstrate an understanding of flowcharts by naming an appropriate function name for the given problem followed by writing the equivalent program code, then it is estimated that a student has adequate problem solving skills. Similarly for understanding code followed by drawing the flowchart. However, if a student can demonstrate comprehension at naming the method or flowchart, but not able to write the program code, it is reasonable to conclude that such students lack the knowledge and skills for problem solving. Similar principle is applied if a student is not able to either, name the method, draw the corresponding flowchart or write the appropriate program code.

3. Data collection
Data for this study was collected under exam conditions. The linked list data structure topic was tested. The students were taught this particular topic and completed a tutorial and a lab session prior to the test. Generally, the students learned the use of standard notational description of the basic shapes of flowcharts in their fundamental programming course. During teaching, the students were shown how these flowcharts were used in linked list.

A total of 90 students contributed data to this part of the study. All students answered two questions – the first question on flowchart and the second on programming code presented in Java where necessary. The difficulty level for each question was considered equal as the two are developed based on slight variations of one another. This means the answer for question 1 is somewhat identical to the given flowchart from question 2. Similar principle applies to the programming code posed in question 1.

The open-ended questions required students to study a flowchart or a short source code of a method (i.e., getElement and contains methods) and provide answer in two parts, a) name the operation of the method or flowchart, b) draw the corresponding flowchart or write the code also in Java. Score out of 10 marks was calculated - one each for naming the code/flowchart and four marks for each correctly interpreted question. As for part (a) - naming the code, a full mark is given if the answer closely represents the processes defined by the code/flowchart showing that a correct answer is provided. Thus, the given method name answered need not exactly match the marking scheme. Ambiguities are resolved by validating with the corresponding answers provided in part (b) where a full mark is given for part (a) if the code/drawing in part (b) are correct. The four marks for part (b) were divided by percentage of correctness with 25% for 1 mark and increase by an additional mark for every further 25%. Zero mark is given if an answer is unrelated or incorrect.

All students who undertook this test had taken a fundamental programming course studied in Java in their first semester. A large majority of them scored grade A in their final examination. They were undertaking the Data structure and algorithm course as advanced programming, a continuation programming course in their second semester of their first year at the time of the test.
4. Performance data analysis and discussion

The scores of the students are shown in Table 1 and 2. Figure 1 and 2 show the corresponding graph representation.

<table>
<thead>
<tr>
<th>Q1</th>
<th>Flowchart Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>0</td>
</tr>
<tr>
<td>Comprehension</td>
<td>Very weak</td>
</tr>
<tr>
<td>Poor (0)</td>
<td>1</td>
</tr>
<tr>
<td>Excellent (1)</td>
<td>0</td>
</tr>
</tbody>
</table>

Comprehension. Poor = 0 mark; Excellent = 1 mark
Composition. Very weak = 0; Weak = 1; Average = 2; Satisfactory = 3; Excellent = 4 mark

Table 1- Relationship between comprehension of code and composition of flowchart for Question 1

Figure 1- Students’ success rate at understanding the code and drawing the flowchart for Question 1. The comprehension level is represented by poor(0) and excellent(1) performance.

In question 1, the students’ performance analysis in terms of their ability to comprehend or understand the code ( remarked by excellent comprehension in Table 1 and Figure 1) showed that majority of the students who named the method correctly could draw excellent flowchart equivalent to the given code. Similarly, those who did not manage to name the method correctly ( remarked by poor comprehension) remained successful at drawing the flowchart, although the number of students for this dropped by half. However, a paired t-test between the code comprehension and flowchart composition did not show a statistically significant results. Thus this may not support our hypothesis fully that those who are competent at understanding the program code in terms of its functions and name the appropriate method name are able to compose the flowchart without difficulties.
Analysis of question 2 showed that 25 students who could not interpret the flowchart (i.e.: not able to name the process workflow) adequately were weak coders as they only managed to write approximately 20% of the correct source code, thus, scored a 1 mark. Surprisingly, majority of the students who managed to name the process workflow of the flowchart were only able to write satisfactory codes. This means a large portion of the written source code were correct (i.e.: 75% correct), but contained inaccurate lines with respect to the given algorithm. Only four students showed excellent codes relative to the flowchart of question 2. A paired t-test comparison between flowchart comprehension and code composition showed a significant results, p<.05. This shows that there is a strong relationship between flowchart comprehension and code composition indicating that good proficiency reading flowcharts would at least improve a student’s competency at implementing program codes.

Based on this preliminary investigation, between these results, it is reasonable to conclude that understanding programming code followed by drawing its equivalent flowchart seems easier compared to the opposite task. In other words, the statistical distribution of the code writing performance showed the mean towards excellent composition for the excellent comprehension group and with the mean towards very weak composition for the poor comprehension group, respectively. The students can be considered better problem solvers when translating code into flowcharts. However, their problem solving skills require more comprehensive knowledge, practice and skills necessary for understanding a flowchart before converting them into the respective code. A paired t-test between both questions compared between each composition level was non-significant.

![Figure 2 - Students’ success rate at understanding the flowchart and writing the code for Question 2. The comprehension level is represented by poor(0) and excellent(1) performance.](chart.png)
5. Conclusion

This preliminary study investigates first year Computer Science students’ level of comprehension and composition by analysing tasks related to interpreting code into flowcharts and flowcharts into code. These students were tested on linked list, a data structure topic they studied in their second programming course (CS2). The early findings showed that they are still weak in coding and solving programming questions although a large number of these students did well in their first programming course. This finding is consistent with that reported by Lister et al. (2004) who proposed that students who are weak at reading code are potentially also weak in problem solving. This translates to an interpretation that if students are unable to read code well, it is unlikely that they will do well in writing novel codes. The reasons for this lack of skills may be attributed by vast possibilities such as lack of comprehensive conceptual understanding of the data structure topic, students do not master the flowcharting knowledge and weak at coding or uses inappropriate syntax while solving a programming problem. The initial result of this study found that these students perform better in translating source code into flowcharts more than the opposite task. This suggests a few possibilities including 1) the flowchart notation may be harder to understand than the Java syntax, at least for the problem studied, 2) the nature of Java as an imperative language enables a short Java program as that tested in the present study to be automatically divided into short segments which can be individually examined. However, a flowchart taken as a large undifferentiated whole which has to be understood all at a time may be a contributing factor for the differences in both tasks. 3) the Java language in the form of syntax uses conventional lexemes for control structures and other operations allows immediate understanding of the corresponding semantics. In contrast, flowcharts do not have lexical representation for familiar control structures apart from the standard notational shapes (i.e., rounded rectangle for start/end program, rectangle for processing, diamond for decision, parallelogram for program input/output) that is less informative than the kinds of control/selection structures represented in codes (i.e., for loop). The lexical representation have to be inferred from the connectivity between the shapes which can cause further difficulties as the parts of a flowchart can be spatially rearranged as long as the connectivity maintains. For this reason, a flowchart reader cannot necessarily rely on all types of loops structured in the same pattern. Thus, generally, this interpretation suggests that the flowcharts are a lower-level notation than the Java codes and the use of flowcharts for the type of task used in this study may be a causal role in the task being harder. Given this limitation, the next logical research step is to improve the notation used in this study. Instead of using flowcharts, a higher-level notation should be considered. The results of the present study could be better analysed using a more standardized coding scheme such as Good and Brna (2004), that was developed by analysing levels of abstraction in program summaries that allow participants to express their understanding in their own words. The coding scheme developed based on Pennington’s (Pennington, 1987) showed how descriptions of programs are classified. Another limitation of this study is that only one question for each type of translation was tested. Thus, the present results only apply to the questions chosen. Hence, limits the generalization of the results. Less emphasize on flowcharting given during lecture and practical exercises could have contributed to the student’s ability in comprehending the flowcharts as expected.

This inconclusive result stipulates further extensive study. It is proposed that the following is considered in the upcoming study:

- Develop measures to specify what and how problem solving approaches can assist beginner learner to develop their semantic knowledge.
- Evaluate the effectiveness of visualization in programming education by engaging learners in active learning activity.
- Determine whether students who perform in comprehension of code and flowcharts would do well in other programming activities such as commenting code, completing incomplete portion of code, memorization, debugging, modification, restructuring a random process flow of algorithm.
- Specify what learning activities facilitate novice learners to develop their problem solving skills.
• Identify what cognitive processes are involved in problem solving and coding by testing and comparing existing learning models.
• Analyse the levels of abstraction in programming comprehension according to the coding schemes proposed by Good and Brna (2004).

6. Acknowledgement
This research was in part supported financially by the University of Malaya Research Grant (RP030C-14AET) and Fundamental Research Grant Scheme (FP062-2014A). The author is grateful for all colleagues from Faculty of Computer Science, University of Malaya, who provided insight and expertise that greatly assisted the research.

7. References


8. Appendix

Questions were given to students in printed copies where they were asked to answer in empty spaces allocated in the question paper. Here, the spacing is altered to fit the conference format. A short description and notation of the basic shapes (i.e., rounded rectangle for start/end of a program, rectangle for processing, diamond for decision, parallelogram for program input/output) used to represent standard flowcharts was given prior to the questions.

**Question 1**

Given are codes for `OperationX(int index)`. Answer the following questions:

```java
public E OperationX(int index) {
    Node<E> temp = head;
    for(int i=0; i<index; i++) {
        temp = temp.next;
    }
    return temp.element;
}
```

- a) What is the name of the Operation X?
- b) Draw the flowchart for `OperationX(int index)`.

**Question 2**

Study the following flowchart.

- a) What is the name of the Operation Z?
- b) Using the method name you give in 2(a), convert the flowchart given above into code. Write the method signature with an appropriate method name.
Assessing Novices’ Program Comprehension based on Linked List Diagrams

Unaizah Obaidellah
Faculty of Computer Science & IT
University of Malaya, Malaysia
unaizah@um.edu.my

Abstract
Learning data structure can be difficult for low performing or novice students. Using diagrams during problem solving may benefit their program and algorithm understanding. We aim to evaluate the effectiveness of different types of linked list diagrams in assisting these students to solve specified problems in terms of naming operations and writing code for selected list operations of the data structure. Twenty eight novice computer science undergraduate students took part in this assessment. The metric used was students’ accuracy to name the list operations and to write their corresponding code. Findings revealed that the problem solving process was best supported in the presence of some amount of diagrammatic notation that describes the operation under consideration.

Keywords: diagrams, novices, problem-solving, data structure, program comprehension

1. Introduction
First year computer science students frequently encounter difficulties to master computer programming courses. The difficulty is most evident when they have little or no prior experience of computer programming. Among others, this problem is caused by one or a combination of inadequate experience, practice, motivation and/or problem solving skills. Winslow (1996) suggested that it takes around 10 years to become an expert in computer programming. In most cases, reaching this level of expertise is unlikely during their undergraduate degree program, as the commonly practiced duration for a CS undergraduate program range within 3-4 years across universities internationally.

The programming courses are offered in various levels and complexity throughout the degree program. Often, it is compulsory for the students to undertake a few of these courses, each designed to prepare them with specific necessary skills. Generally, students are required to accomplish the fundamental programming courses prior to registering the more advanced courses such as algorithm design and data structures. In some institutions, these courses offered during a term of 3-4 months are considered a short duration to enable these students acquire adequate mastery. As a result, many of these students (considered novices due to their limited programming proficiency) fail to excel even in the fundamental programming course. However, practices in some universities allow students with marginal passing grades to register for the more advanced courses. Considering their weak understanding, problem solving and practical skills, efficient learning method is necessary to facilitate competency and continued motivation. Thus, using diagrams is a potential approach for this purpose due to its characteristics, being visually stimulating, reduces complexity of abstract ideas and motivates longer attention for most learners.

In the programming domain, learning with diagrams has been demonstrated useful to support program and algorithm understanding. The concept described as program visualization is defined by Myers (1986) as a conventional textual approach of specifying programing with the aid of graphics used to illustrate some aspect of the program or its run time execution. Myers (1986) further described that ‘code visualization’ illustrates actual program text in the form of graphics (i.e. flowchart) or adding graphics to show the algorithm of a program. The use of graphics to show algorithm also exhibit an abstract (i.e. conceptual and theoretical) form of how a program operates. For example, 2D displays such as flowcharts, block structured programs have been used to facilitate program comprehension (Good, 1996; Smith, 1977). Some of these studies used diagrammatic representation for data structures (Brown & Sedgewick, 1984; Myers, 1980).
Novice programming students often claim that they acquire strong conceptual understanding but fail to write fully functional code. Various studies (Adelson, 1981; Pea & Kurland, 1984) remarked this as a result of poor ability in problem solving where students are unable to breakdown a problem into systematic and well-designed sub-problems. Our experience assessing weak (low performing) programming students on their ability to write computer programs show that they often memorise and recite codes and reproduce (with or without slight modification) them in the exam that tests similar questions. Their reproduction reflects superficial (weak) understanding as assessments of their knowledge, requiring higher level thinking skills, produce poor results. On the contrary, practicing “deep” learning strategies (involves close attention to meaning of the content being learned and relate them to learner’s existing knowledge) may potentially reduce cognitively passive learning behaviour (i.e., review or highlight class notes and program examples; listen to lecture) among novices or low performing programming students. More active learning behaviours associated with deep learning enable students to breakdown complex processes in gradual steps, an approach that facilitates the problem solving process. Considering this advantage, we estimate that the use of diagrams in learning programming has some potential to enhance novices understanding in computer programming.

The work reported in this paper builds on the above and extends research in the programming domain with the use of diagrams. We focus on using diagrams to evaluate the effectiveness of learning linked list (a topic of the data structure course) among novices. In this context, we define novices as low performing programming students who marginally passed the fundamental programming course (scored 50% in their final examination assessment) and those who are re-taking the course due to a previous failure in the data structure course. Most of these students demonstrate insufficient fundamental conceptual knowledge and programming skills. We aim to test the extent of using the linked list diagrams in assisting low-performing students in solving a data structure topic. The data structure topic on linked list is chosen because the topic is relatively easy to be conceptually represented in a notational approach that shows the relationship contained between the elements. Each statement of the program can be represented as a notation. It is argued that learning from diagrams is useful given that the notational and relationship between the elements is understood by the learner.

Our experience teaching low performing students using the conventional approach by explaining the conceptual relationship with the corresponding code are less effective for this group of students as proven by high number of failure rates among some of them. Therefore, we devise a strategy that adopts the classical linked list diagrammatic notation as part of our method in teaching the linked list topic. Students are taught the relation between the algorithms of list operations with the linking operations that can be shown in the diagrams. In this study, we intend to compare three variations of linked list diagrams and evaluate its usefulness among these novices in solving a particular problem. The problems posed in a quiz setting required the students to study the given diagrams and answer questions by naming the corresponding operation before writing their code represented in a standard linked list notation (i.e., nodes and links).

Three versions of diagrams were designed (Fig. 1) which differ in the amount of complexity and information provided. The diagrams are expected to assist the process of solving the given programming problem. However, each diagram posed as a main question is posited to provide different level of assistance in helping the novices to solve the problem posed. We are interested to evaluate if a particular structure of diagrams (if any) is more beneficial for the novices. Better understanding on which type(s) of linked list structure is most useful in teaching the novices would improve the existing teaching method.

The diagrams illustrate complete or partial procedures (algorithms) for common operations of standard linked list concepts (i.e., add an element first in the list, add element at an index location, remove the last element in the list), taught in the classroom and similar to those found in textbooks. Each portion of the illustration can be translated into their corresponding code. Respectively, the following versions of the diagrams represent these operations: add an element first in the list, add an element at a particular index and remove the last element from the list.

The first version, “partial and labelled” diagram only showed a portion of the notation (i.e., the nodes and arrows to represent elements of a list and their relationship) with labels to describe part of an
unnamed operation (i.e., operationX). The second version, “complete and unlabelled” diagram showed a full notation only without meaningful labels to describe another type of list operation. Students were required to label the diagrams prior to writing the corresponding program code. The third version, “complete and labelled” diagram showed a full notation with labels describing a different operation. No “partial and unlabelled” diagram was given as information provided by this type of representation is considered insufficient and less meaningful to support effective cognitive processes.

We predicted that the “complete and unlabelled” (version 2) diagram would assist the low performing students more than version 1 and 3 diagrams due to the additional fill in the blanks task that would set as an advance organizer for task that follows (i.e., writing code). The version 3 diagram is expected most difficult and version 1 as intermediate in facilitating novices in solving programming problems associated with the linked list of the data structure course. The version 3 diagram is expected to provide least code cue support to assist the students in writing the code unlike version 1 and 2, due to the missing fill in the blanks activities.

2. Method

Participants. Twenty eight undergraduate computer science students (6 male, 22 female), aged between 19-24 years old (Mean age=20.4 yr.; SD=1.63), from a public university in Malaysia participated in the assessment in partial fulfilment of a course requirements. The participants who registered for the Data Structure course were mostly low-performing students as they were either re-taking the subject due to previous failure to pass with a minimum of 50% of the course assessment (C-) or taking the subject first time upon similar marginal passing of the previous fundamental programming course (i.e., C-).

Materials. Two open ended questions follow each diagram to assess the student’s comprehension level. These questions asked students to name the operation and write the complete program for the operation named. The instruction specified them to “write the correct method name to replace operationX” in the method signature of their program. In the first and second version, students were asked to complete the program by filling in the blanks of the underlined missing statements (as of fill...
in the blanks task). The second version requires students to label the numbered parts of the diagrams by filling in the blanks task prior to writing the full program. However, in the third version, students were required to write the complete code without any underlined cues given. Different operations were being tested for each version. These operations refer to adding an item at a particular position (i.e. addFirst for the first version, addAtIndex for the second version, removeLast for the third version).

**Design.** All students performed all conditions by completing all questions for the three versions of the diagram - “partial and labelled”, “complete and labelled” and “complete and no labelled”, once each.

**Procedure.** All students answered the questions in a quiz environment within one hour during a lab session in a paper-based test. They were informed about the quiz one week before it took place. No access to a compiler, learning resources or peer discussions was allowed during the quiz. Although there was no restriction in the program language used to write the code, all students wrote their programs using the Java language following that used in teaching. The students were familiar with all of these notations and diagrams as they were used in the teaching part of the course. Prior to the quiz, the students completed their exercises including tutorial and lab assignments using these diagrams.

**Coding.** All answers were coded blind to condition by a research assistant. The score given was one mark for each correct answer. The distribution of the full marks for each part of the question is as follows: 1) one mark if a method is named correctly, 2) 10 marks if all code statements are complete and correct. The total mark of the quiz is 33 marks (11 mark for each question).

No additional minus mark (penalty) was deducted for any wrong statements written for questions on writing code. As writing the program code occurred on paper, minor syntax errors (i.e., missing semicolon or open brace) that do not affect the answers being evaluated were disregarded. A final score was calculated as the total number of correct statements on each main question.

### 3. Results

Table 1 shows the results of means and standard deviation for naming method, writing code and total overall score for each diagram version.

<table>
<thead>
<tr>
<th>Type of diagram</th>
<th>N</th>
<th>Name operation</th>
<th>Write code</th>
<th>Total score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>M*</td>
<td>SD</td>
<td>M*</td>
</tr>
<tr>
<td>Partial and labelled</td>
<td>28</td>
<td>0.78</td>
<td>0.42</td>
<td>6.58</td>
</tr>
<tr>
<td>(version 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete and unlabelled</td>
<td>28</td>
<td>0.93</td>
<td>0.27</td>
<td>6.01</td>
</tr>
<tr>
<td>(version 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete and labelled</td>
<td>28</td>
<td>1.00</td>
<td>0.00</td>
<td>3.32</td>
</tr>
<tr>
<td>(version 3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Total means scores (M) are out of 1 for name operation, 10 for write code and 11 for the total score.

**Table 1. Descriptive statistic of the results.**

Generally, the results for version 1 and 2 are similar. The mean results of the total scores for version 1 and 2 diagrams may suggest that these types of diagrams are more effective in facilitating novices during the solution process. Similar pattern also showed for writing the code for the corresponding named operation. However, all 28 students correctly named the operation illustrated in the version 3 diagram (M=1.00, SD=0.00). A t-test comparison between the diagrams for the total marks was
significant, p<.001, between version 1 and 3, and version 2 and 3. Table 2 shows the number of students who scored full marks in each task (i.e., name operation and write code). Nine students wrote complete and correct code for version 1 diagram, while only two and three students for, respectively, version 2 and 3. The overall results indicate that the version 1 diagram is more effective than versions 2 and 3. One third of the students were able to write the code better in version 1 (Table 2). Version 3 diagram was the least effective for writing code although all students were able to name the operation correctly.

<table>
<thead>
<tr>
<th>Type of diagram</th>
<th>Number of student</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name operation</td>
</tr>
<tr>
<td>Partial and labelled (version 1)</td>
<td>21</td>
</tr>
<tr>
<td>Complete and unlabelled (version 2)</td>
<td>25</td>
</tr>
<tr>
<td>Complete and labelled (version 3)</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 2. The number of students scoring full marks for each task and overall score

4. Discussion

The aim of the work is to evaluate whether different types of linked list diagrams are useful to facilitate the process of writing program among novices. The problems posed in a quiz setting required the students to study the given diagrams before they answer questions by naming the corresponding operation before writing their program.

The higher mean scores (total marks) for the partial with label (version 1) and complete without label (version 2) diagrams suggests that these representation are potentially most useful in assisting novices during their problem solving process. In terms of writing code, both of these diagrams employed the fill in the blanks tasks, which could have cued the participants to think of the correct steps of the algorithm. This is because information on these diagrams such as the labelled nodes and their association may serve as advance organizers for the learner. A pattern of highest score for version 1 (total score, writing code and number of student scoring full marks) diagram may suggest its effectiveness more than the others. This could be due to less complicated logic and/or fewer lines for the code for the add first operation (i.e. adding an element first in the list) shown in version 1 diagram (Fig 1(a)).

Although all students correctly named the operation for the version 3 diagram, their ability to write the corresponding program were not equally demonstrated. This means the students were able to analyse the processes for the remove last operation (i.e. removing the last element in the list), but lack effective solution strategies. Another indication is that these students attained necessary conceptual understanding of the list operation considered. Fewer number of students who scored full marks in the questions on version 3 diagram and in scores for writing the code, indicated that writing the complete program for the operation illustrated in the diagram is more difficult without the presence of the underlined parts (as fill in the blanks question), unlike that available for version 1 and 2 diagrams. Thus, the lack of code cues might have made the task with diagrams 3 much more difficult. Although this may seem speculative, perhaps, the availability of the underline portion for the statements has some relation with the amount of confidence these novices attain. This is considered because the amount of complexity of logic involved can be considered similar with the add first operation (version 1).

Findings for the version 2 diagram follows our prediction, which hypothesised that requiring students to label the numbered parts of a diagram prior to writing the code would assist their problem solving
process. It is expected that this practice (i.e., parts labelling) would form a mental schema for the operation depicted in the diagram, where the labelled portion serves as fillers for the slots given as underlined spaces in the writing code task. However, a pattern of lower marks for the writing code task, the total marks, and reduced number of students scoring full scores than version 1 diagram do not fully support this prediction. A potential reason why version 2 diagram was not as effective as version 1 diagram may be due to more complicated logic these novice student have to evaluate, thus reducing their capability to prioritize solution strategies and/or eliminate irrelevant criteria when solving the question. Although two groups of low performing students participated in this study - retakers and non-retakers (low performing pre-requisite course students), the results suggests that the student’s familiarity of the diagrams does not seem to affect their performance.

An alternative interpretation of the present results could be discussed in terms of ‘which exercise was the easiest’ with respect to the use of these diagrams. Diagram 1 was considered easiest to write code. Similarly, writing code for diagram 2 was easier than diagram 3. Although an opposite pattern showed for diagram 3, the complete and labelled notation to describe the remove last operation was easiest for the participants to evaluate for naming the operation.

The limitation of the present assessment only considers one version for each operation. Thus, it is useful to verify the present findings with different versions of the diagrams with variation of operations including control groups in the group of participants. Therefore, due to the no task variation, the present results could have been heavily influenced by the operation described in each diagram, as such, the results are less generalizable. In addition, the present work can be further improved in various ways, such as comparing the effectiveness of these diagrams between high and low performing students and comparing its effects with standard conventional program without any use of diagram.

5. Acknowledgement
The research described in this paper was funded by the FRGS 062-2014A and RP030C-14AET. We thank the reviewers and associate editor for their comments which improved this manuscript.

6. References


Preconceptions of novice learners about program execution

Sylvia da Rosa
Instituto de Computación
Facultad de Ingeniería - UDELAR
darosa@fing.edu.uy

Abstract
The ontological problem of the nature of programs is fundamentally based on the question “what is a computer program?” On the one hand, the algorithm is the abstract part of a program (the text) and on the other hand, the organization of data and the execution of instructions by a physical device (the machine), are the concrete part of the dual nature of programs. This question has important educational implications. In this paper we describe activities carried out to investigate novice students' preconceptions about how a program is executed by a computer.

1. Introduction
In this paper we describe the first stage of a research study of novice students' preconceptions about how a program is executed by a computer. The research is part of a project aiming to promote the exchanges between the field of Teacher Training and areas of scientific and technological research. The project seeks to enrich teachers in their professional role. For computer science this is a key issue considering the increasing importance of learning programming concepts at earlier levels. Especially, it is noted that in teacher training programmes the most important contribution should come from Computer Science departments in universities [2,3]. The activities were carried out with two groups of students. The first group was aged 13 to 15 with the activities taking place at Liceo La Paz 2 (high school La Paz 2). The second group was aged 16 to 18 and the activities took place at Instituto Tecnológico Informático (Institute of Computer Technology). Both locations are in Uruguay. The teachers of both groups took part in the facilitation of the activities within their classrooms. The content of the research activities was proposed by our research group in computer science education from the Instituto de Computación of the University (Computer Science Institute).

The aim of this research is to gather information about how student's preconceptions relate to their experience of simulating a computer running a program. To limit the scope of the investigation at this early stage, it is proposed to use a simple program to exchange the values of two variables (memory cells) using the assignment statement. In the activities the students were asked to:

- simulate the exchange using concrete material (glasses with liquids),
- write an algorithm for the exchange (in natural language),
- simulate the behaviour of a computer running the algorithm,
- anticipate the results of small Python programs for the exchange,
- run the programs and compare their predictions.

The main question is to make the students experience the process of writing an algorithm (the abstract part of a program) and play the roles of the different hardware components when running the program (as a physical object).

The theoretical motivation lies in the problems inherent to the definition of the ontological status of the notion of computing and the dual nature of computer programs [1]. The ontological problem is fundamentally based on the question “what is a computer program?”. We could defend that a computer program is an abstract entity, a mathematical entity, with independence of its physical implementation, or we could sustain that a program is a concrete entity, a physical entity, this is, the physical machine that actually computes that which the program develops.
The tension between the abstract/concrete duality (mathematical/physical) of programs influences the way that the students think. In [8] the authors addressed the question “What are novice students’ conceptions of computer programming” and found the following categories, among others:

- computer programming as writing text, in which to program is understood as to write a text in a foreign language, the computer’s language,
- computer programming as describing actions, in which there is awareness of the relation between the program text and the actions that take place when the program is executed.

The authors point out that the correspondence between text and action is difficult to capture and programming is regarded as an almost mystical way of thinking. The assignment statement is a simple instruction that clearly reflects this correspondence. Indeed, from the point of view of the program text, the assignment gives a value to a variable (a mathematical variable), and from the point of view of the program as executable object, the assignment alters a memory cell (a physical object). This point is clearly expressed in [4], as the authors write; “... our subjects seem to have guessed..., that the questions were about change rather than mathematical equality...” (page 11). How to help students in understanding programming variables has largely been a concern [4-7]. Some authors emphasize the physicality of the variables associated with the memory cells [9-11]. In those, roles of variables in program implementation are classified by the relationship between the variable and the value stored during the execution of a program. For example, a “walker” is a variable that is used to traverse a data structure a "follower" is a variable that is used to store the contents of a variable, prior to the execution of a sentence.

The paper is organized as follows; Section 2 describes two activities with concrete material. Section 3, describes the simulation, where the students play roles as computer components. Section 4 describes students' work with Python programs. These activities are described as they were planned, and in Section 5 some results of students' work are analysed. Sections 6, 7 contain acknowledgements and references respectively. The worksheets provided to students are included in Section 8 (Appendix).

2. First part
The first part of the study includes two activities, described below.

2.1 Activity 1
We work with drinking glasses and liquids of different colours. The glasses are divided into 4 identical sets of 8 glasses each, and the students are divided into 4 groups. Each group of students works with a set of glasses. Each group will have 3 to 4 students. The groups work independently.

In the first activity, the groups are shown two glasses, as shown in Figure 1. The task consists in exchanging the liquids without altering or spilling them. That is to say: the red liquid must be poured into glass[2], and the green liquid into glass[1].

![Figure 1](image.png)

The groups are expected not to be able to complete the task. The students are asked why this is the case and what they would need to complete the task. The aim is to raise their awareness about the need to have a third empty glass to exchange the liquids.
2.2 Activity 2

In the second activity, a sequence of glasses is presented (more than two) as shown in Figure 2; some are empty (in white) and others full (coloured). Some are made of glass, others of plastic, and others of acrylic (marked respectively “G”, “P”, and “A” in the figure). The red and green liquids are corrosive, therefore they can only be poured into glasses made of glass. The task is the same as in activity 1: exchange the green and red liquids without mixing them.

![Figure 2 - Initial state of the glasses](image)

In this case, the groups are expected to succeed and to use a third auxiliary glass. Then, the groups are given the following questionnaire. Every member of each group must provide his/her responses in writing, to ensure that all of them have participated in the activity (in the ellipsis, the values are 1 or 7 as the auxiliary glass chosen by the group).

2.2.1 Questionnaire on activity 2

1. Describe how you completed the task, mentioning glasses by their numbers.
2. Why did you choose glass [...] and not glass [8] or [6], for instance?
3. Why did you choose glass [...] and not glass [...]? 
4. Draw a sequence of glasses so that the task could not be completed.

2.2.2 Writing the program

We work with the sequence of instructions resulting from the first question of the questionnaire above. We ask students: could we use glass 1 instead of glass 7 and vice versa? (Glass 1 has the same characteristics as glass 7). Is it the same if we begin with glass[5] or glass[2]? The different sequences are written down, starting with glass[5]:

<table>
<thead>
<tr>
<th>auxiliary glass used is glass[7]</th>
<th>auxiliary glass used is glass[1]</th>
</tr>
</thead>
</table>

We then explain that the person writing the program (the user) is interested in the final result; exchanging the contents of glasses 2 and 5. The user does not mind whether the computer uses glass 1 or glass 7 as the auxiliary glass, moreover, does not know the state of the glasses. Therefore, the user will call this glass “x”. Additionally, to check that the contents have been exchanged, the user needs to verify the initial state and the final state of the glasses. Therefore, the user provides the computer with the following general program:

1. show state of glasses
2. pour contents of glass[5] into glass[x]
4. pour contents of glass[x] into glass[2]
5. show state of glasses
3. Second part

In this second part, our work was inspired by “Computer Science without a computer” (csunplugged.org). Students continue working in groups of 4, where 3 of them will act as the different components of a simplified computer. The fourth student will be the user. The user gives the above program to the student acting as the CPU. The other students act as the ALU/Memory and the Display.

The purpose of this simulation is to give the students a small taste of what computers do and about the role played by the different hardware components when running a program, and how they interact [12].

The aim is to highlight the fact that computers simply follow the instructions of a program, which does not mean that they “understand” what they are achieving in the process. This is the first step in a process of awareness students must go through about the importance of writing a program correctly; if the program has a mistake that leads to an unwanted end result, the computer will still execute the instructions as they were written.

3.1 Activity

Students are divided as shown in the image.

There is a table, with the row of 8 glasses from Activity 2, and a piece of cardboard to hide or show them. Only the ALU has access to this. The CPU receives the program and each student receives a worksheet with instructions (see Appendix). The roles can be described as follows:

- The CPU (Central Processing Unit) receives the program, reads the instructions one by one and communicates them to the others as applicable. Each time the execution of an instruction is verified, the instruction is ticked off, and the next instruction is executed. This is repeated until all the instructions have been ticked off.

- The ALU (Arithmetic Logic Unit) student executes the instructions received from the CPU, one by one. The glasses are hidden behind the piece of cardboard. After executing a “show” type instruction, the ALU turns the cardboard so that only the CPU can see where the glasses are (this should remain hidden from the Display and the User).

- The Display has a sheet with a drawing of the empty glasses, which they modify according to the orders received from the CPU, and which is then shown to the User.

- The User gives the program to the CPU and checks whether the exchange was successful or not. This student stands facing the Display. This position must be kept at all times, since the Display will communicate with the User by showing an image.
Students must nod to each other to communicate the completion of the instructions. We have written the program instructions on slips of paper, which are placed on the table so that only the CPU student can see them. For each instruction of the program, CPU choose the slip of paper and pass it on to ALU or to Display. We first work with an example so that students can play their roles using their worksheets and material. The initial state of the glasses is shown in Figure 2.

3.2 Running the program
Once the example has been shown successfully, students take their positions and the User gives the program above to the CPU.

The ALU and the Display must always face the CPU, who nods to indicate that some action is needed. They must do the same to show they have completed the actions requested.

- The CPU reads the first instruction.
  1. show state of glasses.
- The CPU selects the corresponding slip and gives it to the ALU.
- The ALU shows the CPU the state of the glasses by turning the cardboard.

The CPU has the following table and fills it in with empty/coloured/full information (full means that the glass contains another liquid).

<table>
<thead>
<tr>
<th>glass[1]</th>
<th>empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>glass[2]</td>
<td>green</td>
</tr>
<tr>
<td>glass[3]</td>
<td>empty</td>
</tr>
<tr>
<td>glass[4]</td>
<td>full</td>
</tr>
<tr>
<td>glass[5]</td>
<td>red</td>
</tr>
<tr>
<td>glass[6]</td>
<td>empty</td>
</tr>
<tr>
<td>glass[7]</td>
<td>empty</td>
</tr>
<tr>
<td>glass[8]</td>
<td>full</td>
</tr>
</tbody>
</table>

- The CPU looks at the Display and shows him/her the table (the Display must always face the CPU).

The Display worksheet states that when the CPU shows a table, the Display must colour the glasses in the first row of her/his worksheet accordingly and then show it to the user (who is always facing the Display).

- The Display colours the glasses according to the information on the table and shows them to the User. The Display nods to notify the CPU that the action is complete.

- The CPU ticks off the instruction and reads the second instruction:
  2. pour contents of glass[5] into glass[x]

In this case, the CPU looks at the ALU, selects the corresponding slip and gives it to the ALU. The ALU knows, according to his/her worksheet, that he/she must:

- look for an empty glass made of glass
- replace the “x” with the number of the glass chosen in all the slips where “x” occurs
- pour the contents of glass[5] into it

- The **ALU** performs this action hidden from the others, but tells the **CPU** the action is complete (nodding).

- The **CPU** ticks off the instruction and reads the next instruction:


selects the corresponding slip and gives it to the **ALU**.

- The **ALU** pours the contents of glass [2] into glass[5], and nods to indicate the action is complete.

- The **CPU** ticks off the instruction and reads the next one:

  4. pour contents of glass[x] into glass[2]

selects the corresponding slip and gives it to the **ALU**.

- The **ALU** replaces the “x” with the number of the glass previously selected and pours its contents into glass[2]. Then the **ALU** nods to notify the **CPU** that the action is complete (nodding).

- The **CPU** ticks off the instruction and reads the next one:

  5. show state of glasses

This time a new column in the table is completed by the **CPU**, shown to the **Display**, who in turn colours the glasses of the second row of her/his worksheet and shows the picture to the **User**.

When the **Display** indicates the action is complete, the **CPU** ticks off the last instruction and the program ends.

The **User** should see, at the end, the two drawings of the initial state and the final state as in Figure 3 (see Appendix).

**4. Third part**

The final activity consists of asking the students to anticipate the results of small Python programs (the students have been instructed in basic Python). They are then asked to run them and compare those results with the predictions they made. The students worked individually and wrote their answers down. The activity is described as follows.

A variable **v** is assigned the value ['w', 'w', 'g', 'w', 'w', 'r', 'w', 'w'], w, r, g for white, red and green respectively. Recall that v[i] is the value in position i, for 0 ≤ i ≤ 7 (for example, v[2] is 'g').

For each program in the table below, answer the questions.

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>print v</td>
<td>v[0] = v[2]</td>
<td>print v</td>
<td>1. Which is the displayed result of each program when executed?</td>
</tr>
</tbody>
</table>
5. Preliminary analysis

The aim of this research is on the one hand, to make the students experience the process of writing an algorithm (the abstract part of a program) and play the roles of the different hardware components when running the program (as a physical object). On the other hand, to gather information about the preconceptions of students regarding the process of a computer running a program. Finally, to offer new insights about how student's preconceptions relate to their experience of performing the actions themselves. Since the study described in this article was performed very recently, I am offering the following conclusions as a preliminary discussion around the facts found so far. An in-depth analysis of the results will be carried out as part of further research.

Thirty-seven works of the students were analysed. Regarding the last activity (Third part, Section 4), just two students answered question 1 as expected, pointing out the value of the variable \( v \) displayed in each print sentence, as shown in the table below:

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>['w', 'w', 'g', 'w', 'r', 'w', 'w']</td>
<td>['g', 'w', 'r', 'w', 'w', 'g', 'w', 'w']</td>
<td>['w', 'w', 'g', 'w', 'w', 'r', 'w', 'w']</td>
</tr>
<tr>
<td>['g', 'w', 'r', 'w', 'w', 'g', 'w', 'w']</td>
<td>['w', 'w', 'g', 'w', 'w', 'r', 'w', 'w']</td>
<td>['g', 'w', 'r', 'w', 'w', 'g', 'w', 'w']</td>
</tr>
</tbody>
</table>

Most of students wrote different values of the variables \( v[0], v[2] \) and \( v[5] \). Examples:

<table>
<thead>
<tr>
<th>in all the cases</th>
<th>( v[0] = 'g' )</th>
<th>( v[2] = 'w' )</th>
<th>( v[5] = 'r' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>v[0] = 'r'</td>
<td>v[2] = 'g'</td>
<td>v[5] = 'w'</td>
</tr>
<tr>
<td>Case 2</td>
<td>v[0] = 'g'</td>
<td>v[2] = 'w'</td>
<td>v[5] = 'r'</td>
</tr>
</tbody>
</table>

There are some limitations to the possibilities to draw conclusions about questions 1 and 2, due to the variety of responses. Instead, these give rise to further questions: Which is the print sentence that students consider? Is the assignment interpreted in reverse for some sentences? Is \( v[2] \) the auxiliary variable? Why? Which are the points of the activities of First and Second part (successfully done) that can be improved to impact in Third part? How can the questions of Third part be better formulated? How do students interpret the differences between their predictions and the results when running the programs? These points will be included in future activities of the project.

Regarding question 3, it is quite clear that more or less half of students believe that the interchange does not take place without print sentences. Indeed, seven students answered that without “print” sentences, the exchange is not performed, and eleven students believe that an error should occur (and expressed orally that this is because there is no print). This result is, to some extent, coherent with the simulation (Second part), where students tended to show (print) the glasses (turning the cardboard), after each of the sentences of the exchange (as if only showing the exchange, it takes effect).

Regarding activities 1 and 2 of First part (Section 2), although both were successfully done, there is a significant gap between the actions students are able to perform and their ability to answer questions that require thinking or imagining. For example, in question 1 of the questionnaire on Activity 2 (Section 2.2.1), where they were asked to write step by step the exchange just made, some students needed to perform the action again to be able to write it down. Likewise, in question 4 many students show difficulties to draw a sequence for which it is not possible to make the exchange: either they did not understand the question or they did not know how to provide an answer, despite having responded immediately and without doubt why the activity 1 (Section 2.1) could not be performed.
At the end of the activities, it was stressed that the simulation reflects the behaviour of the computer in a simplified form and is, in some cases, inaccurate. However, others are properly simulated; an algorithm is transformed into internal codes, the instructions alter physical objects, and the result is displayed to the user in a format that is understandable to human beings. The cells of the memory can be available for program instructions or not, as well.

Finally, I argue that, although the activities were conducted for a research study, I find that it is possible and beneficial to conduct similar exercises as learning activities. Students were highly motivated and enthusiastic which was documented in video recordings and photographs.

6. Acknowledgements
Thank you to Néstor Larroca and Patricia Añón for helping in doing the activities in their classes. The comments of the referees are gratefully acknowledged.

7. References
## 8. Appendix

### Worksheets

#### 8.1 CPU

Your job as the Central Processing Unit (CPU) is to execute the program. This means reading each instruction, choosing, from the slips available, one that matches the instruction, and sending the slip to the “ALU/Memory” and “Display” classmates as follows:

- the ALU/Memory should be given instructions regarding:
  - information about the state of the glasses
  - changing the contents of the glasses
- the Display should be given instructions about showing the user the drawing of the state of the glasses.

The ALU and the Display will be looking at you, awaiting orders, and they will let you know that the action is complete by nodding. You should do the same when you have finished. The table is to be completed with the state of the glasses when the ALU shows them to you. For example, if the instructions are:

- **show state of glasses**
- **pour contents of glass[5] into glass[x]**

Read the first instruction, choose the corresponding slip and give it to the ALU. The ALU turns the cardboard, then you must look at the glasses, complete the table and notify the ALU that he/she can return the cardboard to its original position by nodding.

<table>
<thead>
<tr>
<th>glass[1]</th>
<th>empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>glass[2]</td>
<td>green</td>
</tr>
<tr>
<td>glass[3]</td>
<td>empty</td>
</tr>
<tr>
<td>glass[4]</td>
<td>full</td>
</tr>
<tr>
<td>glass[5]</td>
<td>red</td>
</tr>
<tr>
<td>glass[6]</td>
<td>empty</td>
</tr>
<tr>
<td>glass[7]</td>
<td>empty</td>
</tr>
<tr>
<td>glass[8]</td>
<td>full</td>
</tr>
</tbody>
</table>

Turn to the Display and show him/her the table (the Display must always look at the last column of the state of the glasses). When the Display nods indicating that the instruction has been executed, tick off the instruction and read the next one. To execute it, nod to the ALU and pass on the slip with the next instruction: glass[5] to glass[x]. When the ALU nods, tick off instruction 2 and read the next one. This is repeated until you have ticked off the last instruction.

#### 8.2 ALU/Memory

Your job as the ALU/Memory is to execute the instructions given by the CPU about the glasses on the table. You must look at the CPU all the time, awaiting instructions. If the CPU gives you a slip with an instruction on it, you must:

- If you read “glass[n] to glass[x]” (n between 1 and 8), find the suitable glass (empty and made of glass), and when you find it, cross out the “x” and write the number of that glass (the same
glass must be used for all the instructions that include “x”). Pour the contents of glass[n] into the glass you have chosen as glass “x”.

- If you read “glass[n] to glass[m]” (n and m are numbers between 1 and 8), pour the liquid from glass[n] into glass[m].

Every time you complete an instruction of this kind, signal this to the CPU by nodding.

If the CPU gives you a slip with a “show” type instruction, turn the cardboard and let CPU see the state of the glasses. When the CPU signals that he/she has seen it by nodding, return the cardboard to its original position.

8.3 Display and User
Your job is to wait until the CPU tells you to what to show the User. You must look at the CPU all the time, awaiting instructions. You receive the following drawings:

Figure 3 - Initial and final state of the glasses
Abstract
This study investigates the effect of brain type (SQ minus EQ or E-S Theory) on first year students’ results of learning computer programming. It is anecdotally believed that a relatively high proportion of Computing students have high Systematizing (S) but lower Empathizing (E) personality factors and that this may be a contributing factor in student choice of subject, learning styles, and aptitude for learning to program. Two years of data are reported here, with EQ and SQ-R questionnaires being completed before any first year undergraduate teaching. The brain type of each student is calculated and compared to their final programming unit performance. There is some evidence that choice of degree subject may be related to brain type, but no evidence is found of a correlation between a male brain type (Type S or Extreme Type S) and programming aptitude.

1. Introduction
The main impetus for this work was to follow up a study conducted by Wray (2007). Wray’s hypothesis was that using two personality questionnaires from the Autism Research Group at Cambridge (http://www.autismresearchcentre.com/) the Systemizing Quotient (SQ) and the Empathy Quotient (EQ), the scores either together or individually would result in a correlation with a measure of programming ability. Wray observed that SQ minus EQ is highly correlated with the programming test scores of 19 students (all male) on the BSc (Hons) Telecommunications Systems Engineering at the Royal School of Signals. The two questionnaires were administered at the end of the course, after the programming test, thus Wray postulated that the SQ-EQ could be a predictor for programming aptitude, if delivered before a programming course began.

Thus our study is being conducted to see if SQ minus EQ is a predictor of programming aptitude with a larger sample, taken from a broader participant base and including both males and females. We administered the EQ and SQ-R questionnaires to the Bournemouth University first year Computing undergraduate students in their first week at university over three years 2013, 2014 and 2015, with the intention of correlating these results against their first year Programming unit results.

Our secondary, but no less important reason, for undertaking this research is the desire to explore and advance our teaching and learning strategies for programming. That both learning to program and the teaching of programming at undergraduate level are difficult and that the failure rate is high is evidenced by practical experience (both ours and many others) and by the volume of literature that surrounds the issues (Jenkins 2002, Robins, Rountree et al. 2003, Bennedsen and Caspersen 2007, Bornat, Dehnadi et al. 2008, Watson and Li 2014, Watson and Li 2014). In developing predictors of students’ performance on undergraduate programming courses, both as a means of understanding student achievements and identifying weaknesses would mean we could potentially change teaching strategies in a way that would support a wider range of students to achieve success in programming.

2. Background
2.1. Empathizing and Systemizing
Following findings that autism occurred more often in families of physicists, engineers and mathematicians and that autistic conditions are associated with scientific skills, the theory of empathizing and systemizing modes of thought has been proposed (Baron-Cohen, Wheelwright et al. 1997, Baron-Cohen 1998, Baron-Cohen, Wheelwright et al. 2001). Empathizing is defined as the drive to identify other’s feelings and emotions and to respond with an appropriate emotion. Whilst
systemizing is defined as the drive to understand rules governing the behaviour of a system and the drive to construct systems that are lawful (Baron-Cohen 2002, Goldenfeld, Baron-Cohen et al. 2005).

Forced-choice, self-administered questionnaires tests were developed to measure the two modes of thought the Empathizing Quotient (EQ) and the Systemizing Quotient (SQ) (Autism Research Centre, Baron-Cohen, Richler et al. 2003). The EQ is a 40-item questionnaire designed to assess cognitive and affective empathy giving the respondent a score from 0 to 80 (Baron-Cohen and Wheelwright 2004). The SQ-R, a modified version of the original SQ, has 75 items (Wheelwright et al., 2006) designed to measure the respondent’s drive to systemize, resulting in scores between 0 and 150. On both the EQ and SQ-R, participants are asked to respond ‘definitely agree’, ‘slightly agree’, ‘slightly disagree’ or ‘definitely disagree’ to each item, and approximately half the items are reverse scored to avoid response bias (Baron-Cohen, Richler et al. 2003).

It is the difference between an individual’s empathizing and systematizing scores that has been proposed to lead to a useful distinction in the understanding of different ‘brain types’ (Goldenfeld, Baron-Cohen et al. 2005). The E-S theory proposes that the difference between empathizing (E) and systemizing (S) categorises the individual ‘brain type’ as Type S (S > E), Type E (E > S) or Type B (E = S), where B stands for balanced. Two extreme types have also been identified Extreme Type S (S >> E) or Extreme Type E (E >> S) (Baron-Cohen, Knickmeyer et al. 2005). ‘Brain type’ is used throughout this paper as a shorthand way of referring to the difference (as identified by Baron-Cohen et al which is essentially continuous rather than categorical) between participants’ EQ and SQ scores.

It has been suggested that the brain type can be related to gender, Type E typifies the female brain and Type S the male brain, and E>>S typifies the extreme female brain and the S>>E the extreme male brain. Not all women have a female brain, and not all men a male brain, just that more females have a Type E brain (Baron-Cohen 2002).

Findings have also concluded that EQ and SQ scores can be related to university students’ degree choice, that higher SQ scores are found for both males and females studying physical science degrees than those on humanities degrees (Wheelwright, Baron-Cohen et al. 2006, Billington, Baron-Cohen et al. 2007, Focquaert, Steven et al. 2007).

<table>
<thead>
<tr>
<th>Physical Science</th>
<th>Female</th>
<th>AVG</th>
<th>59.9</th>
<th>SQ-R</th>
<th>44.7</th>
<th>n</th>
<th>159</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>STDEV</td>
<td>19.4</td>
<td>11.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>AVG</td>
<td>65.4</td>
<td>35.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STDEV</td>
<td>17.8</td>
<td>11.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biological Science</td>
<td>Female</td>
<td>AVG</td>
<td>52</td>
<td>48.5</td>
<td>290</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STDEV</td>
<td>19.2</td>
<td>11.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>AVG</td>
<td>62</td>
<td>41.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STDEV</td>
<td>17.8</td>
<td>11.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Social Science</td>
<td>Female</td>
<td>AVG</td>
<td>51.2</td>
<td>48.7</td>
<td>181</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STDEV</td>
<td>19.7</td>
<td>10.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>AVG</td>
<td>61.9</td>
<td>41.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STDEV</td>
<td>18.8</td>
<td>11.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Humanities</td>
<td>Female</td>
<td>AVG</td>
<td>48.4</td>
<td>48.7</td>
<td>408</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STDEV</td>
<td>18</td>
<td>11.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>AVG</td>
<td>53.7</td>
<td>40.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STDEV</td>
<td>20.6</td>
<td>11.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: (Wheelwright, Baron-Cohen et al. 2006)  
Table 2: (Billington, Baron-Cohen et al. 2007)
2.2. The Wray Experiment
Wray administered five tests to 19 male students on the BSc(Hons) Telecommunications Systems Engineering at the Royal School of Signals. The programming test was administered at the end of the students’ course in 2006 and the remaining four tests: self-rank (from programming is easy to programming is hard), EQ, SQ and Dehnadi-Bornat tests were administered five months after the completion of the course (and the programming test) (Wray 2007).

Wray found a moderate correlation \(r = 0.44, p = 0.056\) between test scores and SQ (Figure 1), and a moderate negative correlation \(r = -0.45, p = 0.052\) between test scores and EQ (Figure 2).

\[ \text{SQ minus EQ resulted in a high correlation } r = 0.67, p = 0.002 \text{ as can be seen in Figure 3.} \]

From these results Wray hypothesised that the use of SQ-EQ could be a predictor for programming ability, thus if the SQ and EQ questionnaires had been completed before the students began studying programming they would have indicated resulting programming test results.

3. The Bournemouth University Study
The Bournemouth University study aimed to explore the relationship between student brain type and a student’s final Programming unit grade. Following the Wray prediction we would expect to find a correlation between the male brain types (S and S>>E) and aptitude for learning to program.
3.1. The Programming Unit

The first year Programming unit (module) is an introduction to programming, assuming no previous experience. The unit covers a basic introduction to programming covering: variables, loops selection, arrays, file reading and writing, object oriented concepts and interface development. The current language used to explore these topics is Java. This unit is taught as part of a common first year syllabus to students who can be on one of six different degrees: Business Information Technology (BIT), Computing, Computer Networks (CN), Forensic Computing and Security (FCS), Information Technology Management (ITM) and Software Engineering (SE).

The unit is taught using a typical structure of a 1 hour lecture and a 1 hour seminar per week over two semesters (26 weeks). Students also have an optional extra of a weekly 5 hour block of support which is staffed by PhD students, any first student may attend for as much or as little time as they feel they need to support the current work they are doing.

3.2. The Programming Assessment

The students undertook a range of assessments throughout the year, during the first semester they did a number of small programming tasks, designed to both build student confidence and to promote coding on a regular basis, all tasks together were 15% of their final programming grade. Students undertook two online, multiple choice tests, one in the first semester and one in the second, each test was 10% of the students overall grade. In the second semester students had a larger single assignment that was a design, build, test and document assignment that was worth the final 15% (giving the coursework a value of 50%). The final written exam was the other 50% of the students’ grade. Results for the two complete years 2013-14 and 14-15 are below; the 2015-16 academic year is not yet finished at time of writing.

<table>
<thead>
<tr>
<th>2013-14</th>
<th>2014-15</th>
</tr>
</thead>
<tbody>
<tr>
<td>CW1 64.1</td>
<td>CW1 61.3</td>
</tr>
<tr>
<td>Test1 70.2</td>
<td>TEST1 67.9</td>
</tr>
<tr>
<td>SEM1 66.5</td>
<td>SEM1 63.9</td>
</tr>
<tr>
<td>Test2 58.1</td>
<td>TEST2 58.9</td>
</tr>
<tr>
<td>CW2 55.5</td>
<td>CW2 50.9</td>
</tr>
<tr>
<td>SEM2 55.2</td>
<td>SEM2 52.0</td>
</tr>
<tr>
<td>CW AVG 60.9</td>
<td>CW AVG 57.8</td>
</tr>
<tr>
<td>EXAM 51.1</td>
<td>EXAM 54.4</td>
</tr>
<tr>
<td>UNIT 56.0</td>
<td>UNIT 56.1</td>
</tr>
</tbody>
</table>

Figure 4: 2013-14 Programming Unit Results

Figure 5: 2014-15 Programming Unit Results
3.3. Method
Students were issued with the EQ and SQ-R questionnaires during their first session in Induction Week (prior to any formal teaching by the university) during three consecutive academic years 2013-14, 2014-15 and 2015-16. The students were given a brief introduction to the research and asked to voluntarily participate; a number chose not to and some students returned forms that they either didn’t sign (for identification and consent) or didn’t fully complete. Students who did not complete the final exam were also removed from the study as they were deemed not to have completed the unit.

For the 2013-14 cohort of 222 students we had 143 students who completed all assessments and the survey, 20 female and 123 male respondents. For the 2014-15 from a cohort of 236 students we had 153 surveys, 25 female and 128 male.

3.4. Questionnaire Results
A first analysis was carried out to see if the SQ-R and EQ averages and standard deviation were consistent with those found by other researchers when exploring EQ and SQ-R scores by gender and degree subjects.

<table>
<thead>
<tr>
<th></th>
<th>BU 2013-14</th>
<th>BU 2014-15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SQ-R</td>
<td>EQ</td>
</tr>
<tr>
<td>Female</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVG</td>
<td>59.50</td>
<td>44.30</td>
</tr>
<tr>
<td>STDEV</td>
<td>15.4</td>
<td>9.6</td>
</tr>
<tr>
<td>Male</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVG</td>
<td>63.91</td>
<td>40.51</td>
</tr>
<tr>
<td>STDEV</td>
<td>19.5</td>
<td>10.7</td>
</tr>
</tbody>
</table>

Table 3: EQ and SQ-R Scores by gender

At first glance the scores, especially those for the female students (both SQ-R 59.9 and 61.23 and EQ 44.7 and 43.48), are in line with those for the Physical Sciences found by both the Wheelwright and the Billington studies. The male SQ-R scores are similar (compared to 65.4 and 65.46), but both years’ male EQ scores are higher (compared to 35.9 and 35.59), this may be because of the broader nature of the six degrees supported by the first year Programming unit, such that those on the business degrees (BIT and ITM) may more closely resemble those on Social Sciences.

Following the method defined by Wheelwright et al we used the following formulae to identify the five defined brain types. Firstly to derive S and E: \( S = (\text{SQ-R} - \langle \text{SQ-R} \rangle) / 150 \) and \( E = (\text{EQ} - \langle \text{EQ} \rangle) / 80 \). So subtracting the population mean (denoted by \( \langle ... \rangle \)) from the scores, then dividing this by the maximum possible score for each questionnaire (150 for SQ-R and 80 for EQ). D is then calculate as \( D = (S - E) / 2 \), which is the difference between the normalized SQ and EQ scores, thus the measure of the difference between an individual’s empathizing and systemizing scores, so it allows us to determine an individual’s brain type. A positive D score indicates a Type S or Extreme S, a negative score indicates a Type E or Extreme E and a score close to zero indicates a brain Type B (Wheelwright, Baron-Cohen et al. 2006)
When plotting brain type against the final programming grade for 2013-14 (Figure 6) there is no evidence of any correlation ($r = -0.065$, $p = 0.440$).

Likewise when plotting the subsequent academic year, 2014-15, there is no evidence of any correlation ($r = -0.095$, $p = 0.243$).

Thus the overall correlation of calculated brain type and the final scores of both years showed no relationship between the two.

Given that brain type has been suggested to relate to gender, the male and female populations were explored further. In plotting the female populations brain type against grade there did appear some indication of a negative trend (figures 8 and 9), this however was not significant given the population size of 20 in 2013 ($r = -0.247$, $p = 0.294$) and 25 in 2014 ($r = -0.147$, $p = 0.483$).
Both indicated a negative trend, thus suggesting a more Type E brain achieved a higher programming grade than a Type S. Although neither correlations of brain type and result being significant for either year, the EQ results were plotted against grade also to explore the relationship further. The correlations for 2013-14 ($r = 0.403, p = 0.0781$) and for 2014-15 ($r = 0.182, p = 0.384$) whilst showing no significance both indicate a positive trend so are potentially worthy of further investigation.
4. Discussion
The results to date (there is still the 2015-16 year to explore) would suggest that brain type is not a predictor of programming aptitude as was suggested by Wray (2007). On the contrary, an opposing tendency has been observed that indicates a Type E brain or that a higher EQ score alone could be a predictor of programming ability.

A notable difference between the two studies, Wray’s and ours, is the point at which the EQ and SQ-R questionnaires were administered. Wray administered them at the end of the first year course; we administered them at the beginning. The results found by Wray therefore could have been influenced by students’ undertaking their computing degree. The learners may have become attuned to both language and behaviours that then bias their responses to these questionnaires. Whilst the test-retest reliability of the questionnaires has been shown to be high in children (Auyeung, Wheelwright et al. 2009), it would be interesting to see if our students’ scores remain stable across the course, or whether immersion in a particular discipline could have an impact on them.

The first year Programming Unit is almost entirely taught by females, the lecturer and the three seminar tutors are all female. There are two male PhD students who staff the extra support sessions, these however are optional and are based on the student request for help. The teaching style and approach, being delivered by females could specifically favour the female students with a female brain type. The brain type of the lecturer could have been an influencing factor on the results as it is reasonable to assume that the Type E lecturer of this study could have positively influenced and encouraged Type E students.

Our Computing students EQ and SQ-R scores are similar to those found by other researchers exploring university degree choice and EQ and SQ-R scores, most notably for the females. Our first year Computing Framework underpins six different degree routes, from software engineering to business IT, how can we then categorise the whole student body as being a physical science or social science when we do potentially have both?

Whilst we did endeavour to survey the entire cohort of students for all three years, this didn’t happen. Some students were not present during the first session of induction week, some students elected not to participate and others failed to complete the questionnaires fully. Brain type could have an impact on early attendance, on willingness to participate with the study or on attention span for completing questionnaires.

The programming ability of the students on entry to university is not uniform, whilst the unit taught programming from the very basics, assuming no prior knowledge; many students had already encountered programming. This prior knowledge was in some instances very positive; students had programmed for many years (for study, pleasure and/or remuneration) and were very confident and skilled in developing code, and enjoyed the process. Others, however, had already developed a dislike of programming (many students report “I hate programming”) and an opinion that it was a
very difficult subject, such that they were nervous and concerned about the unit. Such pre-existing skills and emotional responses to the subject could well cut across any potential brain type aptitude for programming that might be exhibited.

What is meant by programming aptitude and ability for a first year undergraduate student and how it is measured, could also have an impact upon the results. Wray used a specifically written 10 question test designed to test understanding of function and method calls in object-oriented Python to evaluate his participants programming ability (Wray 2007). Our study used the complete assessment diet of the first year students, including small coding exercises, on-line multiple choice tests and a final written exam. Using a broader spectrum of assessment of ability could potentially obfuscate the impact of brain type on final grade, as skilled programmers are not necessarily skilled at undertaking coursework or exams. Simpler, cleaner measures of programming ability may yield more interesting results.

The results found by Wray suggest a link between brain type and programming aptitude, but the potential uniformity of the 19 male, army personnel, who had elected to take further study during their army career may account for the finding. Our university students, enrolled on one of six different undergraduate courses, both male and female, could be expected to be a much more inconsistent and diverse group, for whom multiple factors would impact upon their programming ability. The conditions that the participants find themselves in not being the least of those factors, settled in a structured army life and building a career, compared to a complete change of both home life and daily educational structure. Brain type could well be expected to have an impact upon how an individual coped with such change and a university student’s ability to succeed in it.

5. Future Work

The third year of the survey data needs to be collated, once the 2015-16 academic year is complete. With three years of data complete the students could be split by degree title, it would be interesting to explore if there were any differences for students on the BIT degree compared to SE degree.

In the second year of the Computing Framework, students can elect which units to study (they have no choice at first year). The comparison of brain type to the second year programming results could be completed to see if there is a relationship. Also contrastingly this second year unit is taught by a male only teaching team.

Evaluating brain type against only those components that involved the writing of code may indicate a relationship, so removing other forms of coursework that may themselves require a particular aptitude.

Re-testing students at different points in their degree would give some indication as to whether undertaking the course may have had an impact upon their questionnaire responses. It would be interesting to see if the results remained stable over time.
6. References


Abstract
The use of video feedback is popular, even usual, in fields involving social behaviour and interaction or physical performance. In other academic subject areas, the use of video as feedback is, as yet, uncommon. The work of others in this field covers group work, generic feedback, small numbers of students, samples and trials. We believe this may be one of the first studies on returning individual personalised feedback to a sizeable number (over 300) first year undergraduate students taking Computer Programming, or any other academic subject, for every assessment submitted on the unit.

Student engagement with feedback is often lacking and in that case, a valuable learning opportunity is missed. Previous work using audio as feedback showed 80% of students would prefer audio to written feedback. However, the separation of submitted programming code from audio comments limits ease of reference to the work. The next natural step was to use video screen capture to augment the student experience by improving easy reference to work by simultaneously providing contextually relevant narrative and visually referring to elements of the work.

1. Introduction
Professionals in Higher Education (HE) are always chasing the ultimate learning experience for students. Empirical evidence elicited from previous cohorts of our students, showed that when receiving written feedback, students frequently claim they cannot understand the message conveyed in feedback; they understand where the errors lie but not how to correct them; or they pass over feedback completely in favour of instant validation from the grade.

It became clear that the marking team were writing the same comments, for the same students, week after week, as students were not engaging with feedback. Other studies recorded similar observations (Ackerman & Gross, 2010), and that students only cared about the mark given, (Mutch, 2003; Starbuck & Craddock, 2012) or indeed that students didn’t even collect their assignment (Carless, 2006; Handley et al., 2007; Mutch, 2003). The question of the extent to which students were reading and engaging with the feedback, and how to inspire them to do so, naturally arose.

The objectives for this work became to 1) improve student engagement and learning from feedback and thereby 2) improve the likelihood of application of learning from feedback to future work, on a Computer Programming unit. We are focussing on objective 1) improving student engagement and learning from feedback

2. Background of Video and Screen Casting Feedback
Previous work by authors included a study of the use of audio feedback (Atfield-Cutters & Jeary, 2013). The publication included a thorough review of written feedback and current work on audio feedback. Therefore, neither of these areas are covered in depth here, but they are augmented here with a review of research in the use of video feedback and screen casting. Only work applicable to non-performance, or social interaction based assessments is included since our interest lies in assessment of academic subjects, in particular, computer programming.

Case Studies
Case studies in the use of video as feedback are rare. Amongst the few the variety is wide ranging and no one seems to have exactly repeated a previous exercise, not even where there was intention to do so,
such as in the ASSET Project at Plymouth, UK (Gomez, 2010), which was an attempt to replicate the original ASSET Project work (Crook et al., 2012) completed at Reading, UK.

The majority of studies done in non-performance based contexts are on the use of generic feedback, that is the same piece of media returned for review to entire cohorts or classes of students covering common aspects without reference to individual student work. Generic feedback has the advantage of reuse, unlike feedback of an individual nature. Certainly from a staff perspective that advantage may have the potential to be outweighed by the benefits of individual feedback. These studies generally agree that the use of video improves the communication of feedback by providing greater clarity, and a more positive message.

The ASSET Project at Reading University, UK (Crook et al., 2012) created a system for the sustainable management of videos created for feedback on assessments. Plymouth also adopted ASSET (Gomez, 2010) with the intention of replicating the positive results but circumstances caused the two projects to diverge over time. These studies involve a variety of academic subjects however, our study focuses on a single unit in Computer Programming.

The approach at Reading was to create generic feedback and not for individual students. Even though, by nature, the feedback could not hold the personalised element students often claim to desire, the response was still very positive. Which raises the question of whether perceived personalisation comes from demonstration of the student’s individual work, or the presence, albeit virtual and asynchronous, of the staff discussing the assignment with the student?

Most studies begin cautiously and partially replacing written feedback with video feedback. Prior to our work on video feedback we used a set of written headings designed to enable consistency across markers, and our written notes were made under each heading along with a grade for that section e.g. Professionalism, Structure, Functionality, Testing. Whilst we only delivered video feedback to random samples of students these notes accompanied the video feedback. This measure ensured marking consistency, from the student perspective, across the cohort. Jones (2014) used a rubric highlighting the sections that applied to the work and this accompanied the video as feedback. Parton et al (2010) made a gradual switch, as the first assessment feedback was written, the second written plus a video and finally a video on its own. Similarly, Henderson and Phillips (2015) also began with written feedback on student’s first assignment and introduced video later.

Jones’ (2014) and Borup et al’s (2014) students were distance learners, with potential increased need for social contact with their tutor (Borup et al., 2014; Palloff & Pratt, 2007). Our full time, with-attendance students who see the staff all week, have many opportunities for interaction, and potentially less need for social contact with tutors through video feedback.

Often studies only involve small numbers of students, as we did at first. Parton et al (2010) used a flip camcorder to return feedback to 12 graduate level students over a summer short course on research methods, delivered by email. Their positive response may in part be due to student interest as they studied learning and teaching based subjects. Moore and Filling’s (2012) study is possibly the closest to our own, with individual recordings made for a single academic subject, in this case English Literature, but again numbers are only 45 students.

With student numbers of only 26, Henderson and Phillips (2015) delivered individual feedback as a tutor talking head discussing their work, making their feedback the closest to a face to face conversation. This decision was driven by a belief in the greater social connection (Borup et al., 2014) Students invested in the relationship with staff are more likely to engage and consequentially learn at deeper levels (Thompson & Lee, 2012). However, initial anxiety on the part of students was observed when facing their tutor, especially if they expected a poor result. Similar to our situation, students already knew their grades before viewing feedback.

Another reason for disregarding screen casting was that there were concerns that markers might become bogged down in the minutiae of low priority errors if viewing the detail of the work (Henderson & Phillips, 2014). Thompson and Lee (2012). The negative aspect of this is that students found it difficult to find examples being referred to in the work (Henderson & Phillips, 2014; Thompson & Lee, 2012). However, screen casts make reference to the work easy in (Rodway-Dyer, Knight, & Dunne, 2011) and
it may be that to maximise the potential benefits of video feedback, future work could consider recording a screen cast enhanced by the face of the tutor on screen as well.

With so many variables involved, and so many studies still at the trial stage, we have not found a study with a strong correlation to our own.

3. Context
In the UK national context, since before our studies began to the present day, the National Student Survey (NSS, 2015) has shown a significant difference, across the Higher Education (HE) sector, in students’ satisfaction with both a) the course (average 85.7%), and b) the teaching (average 86.5%), versus c) the feedback on assessments (average 71.20 %). This potentially shows a lower satisfaction with assessment feedback received than other aspects of the course and teaching. HE staff continuously take initiatives to improve student satisfaction in assessment and feedback, however, the data demonstrates a lack of impact so far, which has also been highlighted by other authors (Hyde, 2013; King, McGugan, & Bunyan, 2008; Yelland, 2011).

Local Context
All 300+ first year Computing students at Bournemouth University enrol on a Computing framework, including degree titles such as, Business Information Technology, Forensic Computing, Computing and Information Technology Management. These students study the same six first year units, including Programming. Our students are a mix of male and female, native English and those for whom English is a foreign language, and those with and without additional learning needs. A very small number of students may be repeating the unit for the second time. Whether there is correlation between any category and the results of student perspective of video feedback have not yet been analysed, although the data is available for future work.

During the first semester students upload three exercises per week. Each week, half of the students are marked on one of the exercises submitted, which is selected at random. Therefore, from the student perspective, one in six exercises uploaded is randomly selected for marking. To ensure students don’t miss out on being marked they must submit all exercises.

When students collect feedback they look at their individual ‘gradebook’ inside the VLE. The students journey to the VLE gradebook means they see the grade achieved first, and then must make a further click to view feedback. Informal observation determined that students were not engaging with feedback. Thompson and Lee (2012) explain that, lack of engagement with feedback may be a strategic move to balance home, work and study. Therefore, for there to be a perceptible improvement, the process must require less time and/or effort, or it must be deemed more pleasant and/or useful by students.

Thus our objective became to encourage students to look beyond the grade for the submission and to 1) engage with feedback so that 2) their learning could feedforward to the next piece of work.

4. Case Study
For this empirical study the unit of analysis is a first year undergraduate student on a computer programming unit in a British university carrying out assignments for formal assessment. The aim is to determine the perceived efficiency and effectiveness of student engagement with formal feedback delivered as video screen capture of a review of the work with audio narrative by the marking tutor. The case study was carried out using the work of Yin (2008) as a guideline.

During the first two years of study a random selection of one third of the students in each cohort, received video feedback on each submission. Therefore, some students never received feedback in this mode, some may receive feedback by video several times. During this period all students were receiving written feedback, including those who additionally received video feedback. In the third year of study all students are receiving video feedback for every piece of work they submit for assessment, and no written feedback is supplied.

Delivery
Students access feedback via the VLE, where previously storage of media had been an issue, as it was for Thompson and Lee (2012). Originally advice was taken to upload videos to YouTube, and set them
to be unlisted so that only a unique link could locate the video (i.e.: it cannot be found by searching). The link was embedded into the student’s area of the VLE. Concerns regarding information about student work being stored externally to the institution were a moot point, and deemed to be an acceptable risk at the time. By the third year of study the University provision for media storage had improved dramatically and now all feedback videos are stored in house via Panopto, thus reducing risk of access from external sources. To connect Panopto to the VLE links are embedded into each student feedback area, and permissions are set to allow only the relevant individual to access the video.

Recording
Originally the simplest way of recording the videos was to use SnagIt - software with basic editing facilities and the ability to choose a section of the screen to record, similar to Screencastomatic used by Jones (2014). This enabled e.g. only the programming code to be shown without unnecessary clutter of other parts of the screen. Panopto negated the need for a separate delivery and recording software, but currently has no function to focus on a section of the screen and now the screen is included in the recording in its entirety. There were no editing facilities built in although the menu item was in place ready for the addition. This was not considered a problem since editing was rarely used previously.

5. Benefits of Screen Cast Feedback
The propositions for using screen cast video as the ‘norm’ are the same commonly recognised benefits of video screen cast feedback in small scale case studies.

1. Students will perceive a benefit arising from the ability to reference their work
2. Students will perceive a benefit from the audio, as previous work shows (Atfield-Cutts & Jeary, 2013), due to the a) additional nonverbal element, b) the increase in volume of information and c) perceived personal and friendly tone.
3. These benefits will (Objective 1) increase engagement with feedback as screen cast video thus potentially (Objective 2) increasing the chances of learning being fed forward to future work.

6. Student perspective
During the period 2013-2015 students selected to receive video feedback were given a link to a Survey Monkey survey to record their responses if they wished to. During 2015-2016, students were asked to fill out a similar survey during December of 2015 and a new version was deployed via Mentimeter in March 2016.

<table>
<thead>
<tr>
<th>Percentage of 2015-2016</th>
<th>Vastly improved</th>
<th>No different</th>
<th>Not as good or Much worse</th>
<th>Percentage of all students 2013-2016</th>
<th>Vastly improved</th>
<th>No different</th>
<th>Not as good or Much worse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personal</td>
<td>86.30</td>
<td>8.22</td>
<td>5.48</td>
<td>Friendly</td>
<td>92.16</td>
<td>5.23</td>
<td>2.61</td>
</tr>
<tr>
<td>Friendly</td>
<td>86.30</td>
<td>9.59</td>
<td>4.11</td>
<td>Personal</td>
<td>90.20</td>
<td>7.19</td>
<td>2.61</td>
</tr>
<tr>
<td>Clear</td>
<td>86.11</td>
<td>8.33</td>
<td>5.56</td>
<td>Helpful</td>
<td>89.54</td>
<td>7.19</td>
<td>3.27</td>
</tr>
<tr>
<td>Helpful</td>
<td>84.93</td>
<td>8.22</td>
<td>6.85</td>
<td>Engaging</td>
<td>88.89</td>
<td>7.19</td>
<td>3.82</td>
</tr>
<tr>
<td>Useful</td>
<td>84.72</td>
<td>8.33</td>
<td>6.94</td>
<td>Useful</td>
<td>88.82</td>
<td>7.89</td>
<td>3.29</td>
</tr>
<tr>
<td>Engaging</td>
<td>82.19</td>
<td>9.59</td>
<td>8.22</td>
<td>Clear</td>
<td>88.86</td>
<td>8.55</td>
<td>3.29</td>
</tr>
<tr>
<td>Encouraging</td>
<td>76.71</td>
<td>16.44</td>
<td>6.85</td>
<td>Encouraging</td>
<td>84.97</td>
<td>11.76</td>
<td>3.27</td>
</tr>
<tr>
<td>Fair</td>
<td>76.71</td>
<td>17.81</td>
<td>5.48</td>
<td>Enjoyable</td>
<td>78.29</td>
<td>19.08</td>
<td>2.63</td>
</tr>
<tr>
<td>Enjoyable</td>
<td>72.22</td>
<td>22.22</td>
<td>5.56</td>
<td>Entertaining</td>
<td>76.32</td>
<td>17.11</td>
<td>6.58</td>
</tr>
<tr>
<td>Entertaining</td>
<td>66.67</td>
<td>23.61</td>
<td>9.72</td>
<td>Fair</td>
<td>75.82</td>
<td>21.57</td>
<td>2.61</td>
</tr>
<tr>
<td>Time</td>
<td>60.27</td>
<td>20.55</td>
<td>19.18</td>
<td>Time Consuming</td>
<td>60.13</td>
<td>28.10</td>
<td>11.76</td>
</tr>
</tbody>
</table>

*Table 1 - How do you feel about your video feedback compared to traditional written feedback?*
For the majority of students, receiving feedback by video is a new experience and across all cohorts (2013-2016) only 2 students had ever received video feedback regularly before from prior educational institutions. The students’ positive attitude towards the new style feedback is demonstrated when asked how they feel about video feedback versus written feedback.

Proposition 1
More than 74% of students using video feedback as a matter of course (2015-2016 only) claim it is easier to identify their errors and 72% find them easier to understand (Table 2). That figure increases to over 84% when the results from earlier cohorts are taken into account. Whilst more than 77% of the cohort believe they benefit from improved learning opportunities over written feedback. The perceived improvement in usefulness and helpfulness, are also likely indicators of ease of reference to work at over 84% each (Table 1).

Proposition 2
Students consider video feedback to be more personal and friendly with over 86% (Table 1) finding some improvement over written feedback. This validates Parton et al’s (2010) results that students felt a closer connection to staff after the second assessment with video feedback (83%) than the first assessment with written feedback (25%). When aggregated with results from previous cohorts the result is over 90% (Table 1). One would not expect the fairness of marking to change just because of the media used, but certainly the student perception is that there is a significant improvement, possibly the result of improved clarity or improved rapport with staff. Thompson and Lee (2012) claim the auditory element is the most important reason screen casting was successful for them.

Proposition 3 - Objective 1
All students across all cohorts accessed feedback successfully. Over 74% believe it will improve the chances of them reviewing the feedback more thoroughly than if the feedback was written. Student perception was sought to ascertain whether the improved engagement could facilitate improved learning (Table 2). Indeed, when using video feedback as a matter of course 77% of students believe they find it easier to learn from video feedback. This may be, at least in part, because the task of identifying errors and understanding them are regarded as significantly easier.

For 82% of students, our first objective has been fulfilled; that is improved engagement with feedback when receiving video feedback as a matter of course (Table 1). The notion that it is also more helpful and useful may be related to engagement. The whole student experience is recognised as more enjoyable and entertaining. Many even find it less time consuming to review feedback.

<table>
<thead>
<tr>
<th>Percentage of students (%)</th>
<th>Much Easier or Easier</th>
<th>Neutral</th>
<th>Harder or Much Harder</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2015-16 only</td>
<td>2013-16</td>
<td>2015-16 only</td>
</tr>
<tr>
<td>To understand</td>
<td>72.97</td>
<td>84.42</td>
<td>14.86</td>
</tr>
<tr>
<td>To identify errors</td>
<td>74.32</td>
<td>85.71</td>
<td>9.46</td>
</tr>
<tr>
<td>To revise from</td>
<td>79.73</td>
<td>81.82</td>
<td>8.11</td>
</tr>
<tr>
<td>To watch (v reading)</td>
<td>81.08</td>
<td>87.66</td>
<td>5.41</td>
</tr>
<tr>
<td>To identify future</td>
<td>55.41</td>
<td>75.97</td>
<td>27.03</td>
</tr>
<tr>
<td>improvements</td>
<td>71.62</td>
<td>84.42</td>
<td>10.81</td>
</tr>
<tr>
<td>To understand errors</td>
<td>56.76</td>
<td>69.48</td>
<td>25.68</td>
</tr>
<tr>
<td>To revisit</td>
<td>77.03</td>
<td>86.36</td>
<td>10.81</td>
</tr>
</tbody>
</table>

Table 2- How easy do you find it to make use your video feedback, compared to traditional written feedback?
Proposition 3 - Objective 2

In concordance with our second objective (to encourage the application of learning from video feedback to future work) 55% of the current cohort believe it is easier to identify future improvements (Table 2). That figure rises to more than 75% of students when data from earlier years are included. Over 84% believe it will improve the chances of them improving future work with recommended changes. Whether or not this occurs in reality requires analysis of individual videos in conjunction with subsequent work.

![Figure 1 - Do you think you would like to see video screen cast feedback on assignments in future? – survey results](image)

**Student Perspective Conclusion**

Students, past and present, would like to see video feedback used again in future (Figure 1). There is a decrease in percentage of students giving positive reactions in all aspects, when regarding the cohort receiving video feedback as a matter of course (2015-16 only) against the aggregated results across all cohorts. This may just be

a) the result of larger numbers

b) an anomaly of the 2015-16 cohort

c) the result of students receiving feedback by video screen cast many times over. This may be an indicator of the novelty factor having a positive influence in previous years.

Future results needs to be monitored in case it is the beginning of a downward trend. These conclusions are drawn entirely from quantitative data. There is much qualitative data still to be analysed.

7. Staff Perspective

Trials began with one member of staff which later increased to two. Since the start of the academic year 2015 the whole programming teaching team have been involved (4 staff), as well as other staff brought in to assist with assessment of the largest and final piece of work.

Some staff experienced initial anxieties regarding recording their own voice. One described it as ‘stage fright’. For most, these anxieties dissipated through experience over varying amounts of time. In addition to the new regime this emotional response may contribute to the slow start most staff find (Hyde, 2013; Thompson & Lee, 2012), but on the whole the process sped up with practise to the point where generation of feedback was the same, or faster, than the written version.
Most staff did not concern themselves too much with the ‘performance’ aspect and did not worry about the recording being perfect. However, for those that did, the burden of marking was vastly increased. There became a need to go through the student work first and to make notes before recording, thus effectively going through the submitted programming code twice. Sometimes the replaying of videos to check content, was also felt necessary.

Finding somewhere quiet to work is a well acknowledged issue (Henderson & Phillips, 2014; Thompson & Lee, 2012). On the whole our staff share offices and often take marking elsewhere to complete. As a result, birds tweeting and clanging of doors and chatter, are common soundtracks to our recordings. Apologies for colleagues walking in noisily, or a pet walking across the keyboard when working from home, become part of the conversation and indeed, is thought to add to the friendly style. After a while the voice needs a rest but as one member of staff pointed out, by the time that happens it is time to move away from the computer for a while anyway.

Some positive aspects noted by other authors were also echoed by some of our staff. These included that it was less tiring, easy, enjoyable and faster (Henderson & Phillips, 2015; Hyde, 2013). You can be more specific and show students how to fix their own code or use a better technique, directly, without having to direct them to a generic example. Video feedback is an opportunity to communicate with students about their own work and to build rapport (Thompson & Lee, 2012).

Certainly staff attitudes were more positive when marking smaller assignments e.g. just code, rather than the final and largest assignment which involved code, plus design and testing documentation.

8. Conclusions
We have learned that our students are happy to engage with video as feedback as a matter of course and feel that the common practise of precautionary supplemental written notes are no longer necessary. The majority of students, across all cohorts involved in the study, believe it is easier to engage with, and identify potential future improvements, from video feedback than from traditional written feedback.

Of the original propositions, 1) Students do perceive a benefit arising from the ability to reference their work, such as ease of identity of errors and understanding of them, 2) Students do perceive the benefits of the audio, 3) Students do believe their engagement with feedback will be improved compared to written feedback, thus also increasing the chances of learning being fed forward to future work.

Since students are likely to lack the knowledge of how to maximise the potential with video feedback for a number of years to come it is important that staff encourage interaction, such as rewinding and note taking, and alteration of a copy of the work whilst viewing (Thompson & Lee, 2012). As academics, it is our responsibility to encourage experimentation to facilitate finding the best strategies for them. We may wish to offer suggestions, but explaining this is a new realm for everyone may embolden students to take the lead.

Rotherham (2008) even claims that giving students richer feedback will save time in the long term. Students will take more notice of feedback; need less repeated feedback; and require less critical feedback, in future as their work improves. Video feedback should therefore be viewed by staff as a long term investment.

9. Future Work
The conclusions here are drawn entirely from quantitative data. There is much qualitative data still to be analysed. In addition, there are many other aspects to be examined.

Working out how students watch their feedback might be both insightful and influential. Not only where and when do they watch, but who they share their videos with, and who do they discuss them with. It is possible that they don’t watch the video to the end or perhaps they view videos multiple times. We also have the data to pursue analysis of categories of student, such as gender, prior qualifications, English language proficiency, additional learning needs and interests as indicated by the chosen degree title.
We can attempt to optimise the content of the videos by looking at styles of presentation, structure, duration, levels of detail and examine whether the intended message is conveyed successfully to the student (Henderson & Phillips, 2015).

Enabling the student side of the conversation is not a mechanism we have considered to date. Would there be additional benefit by making that possible as was done in Reading with their new communities of learning (Crook et al., 2012), or have we already maximised potential by creating feedback on an individual basis versus their generic version? We may wish to include the assessors face on screen and determine if there really is benefit to bringing the feedback closer to the face to face meeting desired by students (Henderson & Phillips, 2015). This relies on the willingness of staff to take part in such a study.

Studies have had positive results using generic videos. Does personalisation come from demonstration of the student’s individual work, or the presence, albeit virtual, of the staff discussing the assignment? When supplying written feedback staff kept lists of common comments to save retyping, making efficiency savings by virtue of the copy and paste facility. There is the potential for technology to enable assessors to record a video explaining a problem with individual work, and then to select a generic video on how to fix it, automatically dropping it into the timeline of the recording in a similar way.

However, with all these potential areas to follow up, there are two likely next steps.

1. Analysis of the quantitative data already collected with regard to objective 1.
2. A study into whether screen cast video as assessment feedback provides improved opportunities for students to use learning gained during feedback review, in future work (objective 2).

Students have often moved their whole lives to begin their HE program, or are managing additional stress on top of a previously full life. Due to their age and maturity, change of lifestyle, and/or displacement from home, students are at a point in their lives when even the most level headed is emotionally vulnerable. Students believe, trust, and indeed expect, that staff know how to construct useful feedback, which also leaves their self-esteem at least intact, if not lifted. We must continue to strive to deserve their faith in us. Positive results means progress towards enabling students to fulfil their potential, and a step closer to the ultimate learning experience for our students.

References


Teaching Software Testing with a Mutation Testing Game

José Miguel Rojas and Gordon Fraser
Department of Computer Science, The University of Sheffield, UK
{j.rojas,gordon.fraser}@sheffield.ac.uk

Abstract
Software testing is crucially important in a world dominated by software. Software testing is also inherently difficult and requires theoretical expertise and practical experience. However, standard testing techniques are often perceived as boring and difficult, and thus do not feature as prominently in programming education as they maybe should. In order to address this problem, we aim to make testing education more interesting with gamification. The Code Defenders game uses gameplay elements to engage students in the testing process in a competitive and fun way. Our hope is that if students perceive writing tests as a fun activity, they will later become better software engineers, with better testing skills, and with more inclination to apply thorough testing. In this work-in-progress paper we describe our initial experiences with Code Defenders, as well as open challenges on the way to making testing education fun.

1. Introduction
It is common wisdom that software needs to be thoroughly tested: Insufficient testing is known to cause project failures, large economical damage, and can threaten people’s lives. Many large and successful software companies have made testing an integral part of the day-to-day jobs of their developers. However, this does not generally hold, and often software developers are not well educated to actually do testing. Indeed there are reports that testing plays a much less prominent role in the daily activities of software developers than one would hope it to do (Beller et al., 2015).

One possible root cause for this lies in how software developers are educated: Programming courses in higher education and outside it generally focus more on the creative aspects of coding, rather than the analytical process of testing. While some basic testing is usually taught, education rarely manages to get people “test infected”; that is, it does not convey testing as an enjoyable task (Barriocanal et al., 2002; Patterson et al., 2003). At face value, this is not surprising, as systematic testing is a comparatively boring task in the face of more exciting alternatives, such as writing new code.

We aim to make testing education more interesting by applying gamification principles. We want to engage students with testing by using the human competitive nature as incentive, and by mapping testing tasks to gameplay components. We hope that if students perceive writing tests as a fun activity, they will become better software engineers, with better testing skills and more inclined to apply thorough testing.

As a first step towards this goal, we have developed Code Defenders (Rojas & Fraser, 2015), a game that makes use of a testing technique known as mutation testing. Players of the game can participate in one of two roles: As attackers, they aim to create mutants, which are subtle modifications of the program under test. The more difficult it is to reveal a behavioural difference between the original program and a mutant, the better the mutant. As a defender, players aim to write tests that reveal these behavioural differences, i.e., kill, these mutants. Both roles encourage and train an understanding of software bugs and how they are detected by tests, and how to create good tests.

Initial studies revealed that students engage willingly with testing using Code Defenders. However, there remain numerous challenges on the way to making Code Defenders a viable approach to teach testing.

2. The Code Defenders Game
Code Defenders uses elements of mutation testing to gamify software testing. This section therefore introduces mutation testing before going into details on the gameplay.
2.1. Mutation Testing

Mutation testing is a software engineering technique aimed at guiding the development of test code and at evaluating its quality (DeMillo et al., 1978). Given a program under test, mutation testing involves the automated creation of a set of variants of the program, called mutants, which differ from the original program by small syntactical changes. The underlying premises are that software developers tend to write almost-correct programs (known as the competent programmer hypothesis) and that small syntactical changes can be representative of complex real-world program faults (known as the coupling effect). The resulting set of mutants can be used to quantify the quality of existing tests by measuring how many of the mutants are detected: A test is said to kill a mutant when the outcome of executing it on the original program is different from the outcome of its execution on the mutant. Mutants that are not killed can guide further test generation efforts. It is possible for a mutant to be semantically equivalent to the original program, in which case there exists no test that could distinguish it from the original program. Detecting equivalent mutants is an undecidable problem and typically requires human intelligence.

2.2. Gameplay

Code Defenders implements a gamification approach to mutation testing. The main components of the technique are built into an intuitive gameplay that allows players to produce mutants, develop tests and reason about mutant equivalence.

In its current version, Code Defenders is a two-player game, and players use the Java language and the JUnit testing framework. Each player takes the role of either the attacker or the defender. The goal of the attacker is to create subtle mutants of a class under test that are hard to kill by editing the source code of the class under test. The goal of the defender is to create strong JUnit tests that kill the mutants created by the attacker. The game is played in rounds of attack and defence, where the attacker takes the first turn of play. Once the attacker has submitted a mutant, the turn is passed to the defender, who then tries to create a test with the right assertions to detect the mutants. A round is completed when the attacker has submitted a new mutant and the defender has countered with a new test. The game then proceeds until the number of rounds chosen when creating the game have been played. Upon completion of each round, a mutation analysis is performed to determine whether the newly created test has killed any live mutants. A special scenario arises when the defender suspects a mutant to be equivalent: Instead of providing a new test, the defender can trigger an equivalence duel, challenging the attacker either to accept that the mutant is equivalent, or to produce a killing test to prove that it is not equivalent.

The point scoring system of Code Defenders is intended to encourage attackers and defenders to produce good quality mutants and tests, and to decide the winner of a game. A mutant scores points for the attacker when it survives long in the game. The more rounds it survives, the more points the attacker scores. On the other hand, a test scores as many points for the defender as mutants it kills. In the case of equivalence duels, the attacker can score extra points by submitting a killing test for the mutant claimed equivalent, but can lose the points scored by that mutant otherwise. More details on the design, implementation and the point scoring system of Code Defenders can be found in an earlier publication on the gamified mutation testing system (Rojas & Fraser, 2015).

3. Initial Findings

We performed an initial evaluation using a controlled study with undergraduate and graduate students from the University of Sheffield. In this section, we present some initial findings from this experiment.

3.1. Evaluation Study

The study consisted of two 30-minute sessions in a computer room, where 38 participants were asked (using a random task assignment) to write tests manually for two small, yet challenging Java classes or to play Code Defenders on them—either attacking or defending. In each session, one third of participants wrote tests manually and two thirds played the game. All participants used the two different classes.

A training phase preceded the two main sessions of the experiment. As part of the training, a brief tutorial on unit testing and mutation testing was presented, followed by an introduction to Code Defenders.
Through short, intuitive tasks, the participants were guided to familiarise themselves with the main interfaces of the game. Furthermore, to conclude the training phase, all participants played an actual Code Defenders game on a simple example class.

In total, 32 games were played and 26 manual testing tasks were completed. On average, 3.8 rounds were played on each game, leading to 121 mutants and unit tests for two Java classes used as subjects. On the other hand, the manual testing tasks resulted in 145 unit tests. While a quality evaluation of these tests is necessary to fully assess the comparison, these preliminary results suggest that the gamification approach implemented by Code Defenders was able to engage students in the task of writing unit tests.

We are currently working on the analysis of the mutants and tests produced during the study. The research questions we are interested in answering through the study involve (a) the kind of mutants developers create on Code Defenders and how they compare with mutants created using state-of-the-art mutation operators, (b) whether using Code Defenders leads to stronger tests than manual testing, and (c) how effective developers are at killing mutants and at detecting equivalent ones.

3.2. Student Feedback

At the end of the study, the students were asked to fill in an exit survey reflecting on their experience with Code Defenders. The survey covered demographics and addressed qualitative aspects of the study from the participants’ perspective.

52% of the participants were undergraduate students, 37% were Master’s or PhD students and the rest were either professional developers or academics. All participants are in Computer Science or Software Engineering-related fields, have a diverse degree of experience programming in Java but generally little to no industrial work experience (66%). The majority (76%) had used JUnit or a similar testing framework before and understood well or very well the concept and usage of mutation testing, although most admittedly only rarely or occasionally writing unit tests when programming.

When asked about their experience with Code Defenders, although a small group had difficulties understanding the game (6), all participants claimed having enjoyed playing it and a vast majority (92%) agreed that writing unit tests on Code Defenders is more fun than writing unit tests while coding. Interestingly, we observe a strong tendency that creating mutants is more enjoyable than creating tests. Whereas these responses indicate that the testing task is more engaging for participants when it is performed in a gamified scenario, supported by most participants acknowledging that they would even play the game for fun in the future, there is still some disagreement about whether writing tests in a professional IDE would have led participants to produce stronger tests, possibly due to the sometimes long idle time when waiting for opponents to play their turns.

The ways in which Code Defenders could be improved were also addressed in the participants survey. A list of suggested features was presented to participants, who were asked to indicate whether they thought each suggested feature was important, just nice to have or rather irrelevant. Incorporating the notion of code coverage by highlighting the code covered by each test seems to be the most important aspect to improve in the Code Defenders. Almost as important seems to be the creation of a single-player mode that would avoid having to wait for an opponent to join the game and play turns promptly. Improving the capabilities of the code editor, a multi-player mode and support for other programming languages are, to a lesser degree, also important and nice to have features from the participants’ perspective.

To capture more aspects that were not covered explicitly in the survey, participants also had the opportunity to suggest improvements for the game in two categories we consider crucial: user interface and point scoring system. Besides some editor capabilities, there were several comments on the point scoring system: The way in which Code Defenders rewards players was determined in an ad-hoc manner with two purposes in mind: fairness and engagement. Some interesting suggestions involve a) making the scoring system time-dependent, i.e., a test scores less points if it took longer to write it; b) different number of points depending on the complexity of the mutants; and c) allowing players to gamble on the number of points to be scored by making trials before actually submitting mutants or tests. Some
participants also mentioned the need for a leaderboard to compare themselves to other players.

Participants were finally given the opportunity to make general suggestions to improve Code Defenders. One of them is particularly humbling and actionable: “Fix bugs”. Others involve features we are planning to explore in the near future, such as (a) limiting the time available per turn, (b) allowing defenders to run a new test on the original code before submission, and (c) improvement of the equivalence challenge protocol in the hard mode, given that it is a bold move at the moment to claim a mutant is equivalent when its code is not revealed to the defender.

4. Open Challenges
We envision that Code Defenders can be useful in at least two contexts: Testing education, and crowdsourcing test development. In both scenarios, the human factor will play a crucial role in the potential success of the approach. Although our preliminary experience with Code Defenders suggests developers enjoy playing the game and also prefer the gamification approach over manual testing, several open questions and challenges remain:

- Breaking code (i.e., creating mutants) tends to be seen as a more fun task than defending it (i.e., writing tests). How to achieve similar motivation levels for both attackers and defenders?

- How to overcome the long waiting times in between turns? Would alternative mechanisms to the current round-based, such as an asynchronous gameplay for example, be more suitable?

- Our initial empirical evaluation involved two relatively simple classes with no dependencies beyond standard data structures. Will the game continue to be practical and enjoyable when used on more complex, real-world programs?

- Code Defenders is currently a two-player game only, but we are working on a multi-player mode that will allow several attackers and several defenders to play the same game. This poses several challenges to the current design of the game. Will competitiveness among sides (attackers vs. defenders) and teamwork within sides complement or conflict?

- Code Defenders can serve as an educational resource to support instructors in delivering theoretical software testing concepts such as fault-detection effectiveness or code coverage, and it could also provide meaningful practical experience to the students through one-on-one or multi-player games in the classroom or even as a self-tutoring tool in a single-player mode. However, the actual educational value of these alternatives need still to be assessed in the field, and leads to follow-up questions such as whether Code Defenders can be used for assessment and how to prevent players from cheating the game in the classroom.

5. References


