

Proceedings of the
31st Annual Workshop
of the
Psychology of Programming Interest Group
(PPIG 2020)

Summer Edition: 17-21 August,
Winter Edition: 1-4 December,
online



Edited by
Mariana Marasoiu, Colin Clark, Philip Tchernavskij, Ben Shapiro, Clayton Lewis and Luke Church

Editors' note

Due to be hosted for the first time in North America, at OCAD University's Inclusive Design Research Centre in Toronto, Canada, the year's theme was Cultivating the Margins.

However, as with many other conferences needing to adjust to the new circumstances of the COVID-19 pandemic, PPIG 2020 went online as well.

At the core of the PPIG annual workshops is the fostering of a community of people interested in all aspects of programming. We aimed to maintain and support these goals in the transition to the online setup as well.

The single event initially planned for July was replaced with two one-week-long virtual events in August (17th to 21st) and December (1st to 4th). During each of the weeks, we met for two to three hours a day for presentations, discussions, panels, and social activities.

The organisers wish to thank all who participated and made both editions of PPIG 2020 such a success!

Mariana Marasoiu, University of Cambridge

Colin Clark, OCAD University

Philip Tchernavskij, OCAD University

Ben Shapiro, Apple Inc. and University of Colorado Boulder

Clayton Lewis, University of Colorado Boulder

Luke Church, University of Cambridge

PPIG 2020 Call for Papers - Cultivating the Margins

The speed, growth, and increasing entanglement of computational systems is actively changing our work, social, political, and creative lives. Yet the apparent success of these systems in reshaping social and economic landscapes has also come with enormous costs—putting fair and stable employment, the veracity of information, respectful use of data, and democratic participation at risk. Perhaps the ways we create and study these systems, and the normative assumptions and values that are embedded within them, need to be reconsidered in a new light?

This critique has been present at PPIG for a while, from End User Programming, Live Coding, to even the idea of studying the psychology of programmers. We've always been a community that invites other perspectives on what it means to program, and we want to continue to extend this interest.

This year's theme prompts us to reflect upon what we're missing—the practices, theories, people, and technologies that have been excluded, set aside, or overlooked by mainstream programming research. What are the edges and limits of programming and programmed systems? Who has agency to participate in their creation and study, and who is relegated to the passive role of user or research subject?

What new or overlooked possibilities are growing at the margins of programming, away from the prevailing industrial and technoscientific values of speed, efficiency, measurability, and scale? What would an ecology of programming look like, in which we have a responsibility to cultivate communities, invite diverse perspectives, and grow a plurality of approaches and epistemologies?

The Psychology of Programming Interest Group (PPIG) was established in 1987 in order to bring together people from diverse communities to explore common interests in the psychological aspects of programming and in the computational aspects of psychology. “Programming”, here, is interpreted in the broadest sense to include any aspect of software creation. As always with PPIG, we accept the widest range of submissions on a variety of topics, such as:

- Programming and human cognition
- Programming education and craft skill acquisition
- Human centered design and evaluation of programming languages, tools and infrastructure
- Team/co-operative work in programming
- End user programming
- Distributed programming, programming distribution
- Gender, age, culture and programming
- New paradigms in programming
- Code quality, readability, productivity and re-use
- Mistakes, bugs and errors
- Notational design
- Unconventional interactions and quasi-programming
- Non-human programming
- Technology support for creativity
- Music(al) programming
- Liveness and interactivity in programming

PPIG 2020 Programme & Proceedings Index

Summer Edition, 17-21 August

(timings are given in British Summer Time / UTC+1)

Monday, 17 August

16:00 - 16:30 **Welcome to PPIG!**

16:30 - 18:00 **Icebreaker & getting to know each other**

Tuesday, 18 August

16:00 - 18:00 *Panel discussion:* **Programming, Communities & Creative Power**

Invited speakers:

Alan Blackwell and Jason Lewis, *software for indigenous communities*

Clemens Klokmose, Midas Nouwens, *nanoscience*

Clayton Lewis, *home automation as support for people with disabilities*

Colin Clark and Michelle D'Souza, *platform cooperatives*

Philip Tchernavskij, *software for citizen natural science*

18:00 - 19:00 **Sightseeing** - StreetView tour of Toronto

Wednesday, 19 August

16:00 - 17:00 *Paper presentations*

Parallel Program Comprehension 8

Eric Aubanel

What the Mouse Said: How mouse movements can relate to student stress and success 17

Natalie Culligan and Kevin Casey

Designing an Open Visual Workflow Environment 26

Charles Boisvert, Chris Roast and Elizabeth Uruchurtu

17:00 - 18:00 *Roundtable discussion:* **Reflections on Programming Otherwise**

18:00 - 19:00 **Musical Soirée** (or **Matinée**, depending on your location!)

Thursday, 20 August

16:00 - 17:00 *Paper presentations*

Assessing a candidate's natural disposition for a software development role using MBTI 32

Daniel Varona and Luiz Fernando Capretz

On personality testing and software engineering 39

Clayton Lewis

17:00 - 18:00 *Panel discussion: Friendly editors*

Invited speakers:

Sarah Chasins

Cyrus Omar

Titus Barik

18:00 - 19:00 **Conference reception** - make and bring your own dish!

Friday, 21 August

16:00 - 16:45 *Doctoral Consortium short talks*

Validation of Stimuli for Studying Mental Representations Formed by Parallel Programmers During Parallel Program Comprehension 42

Leah Bidlake, Eric Aubanel and Daniel Voyer

Purpose-First Programming: Scaffolding program writing and understanding to align with purpose-oriented identities 45

Kathryn Cunningham

16:45 - 17:45 *Panel discussion: Between science and software engineering*

Invited speakers:

Ross Church

Chris Martin

17:45 - 18:30 PPIG Prizes & Close

Winter Edition, 1- 4 December

(timings are given in British Standard Time / UTC)

Tuesday, 1st December

16:00 - 16:30 **Welcome to PPIG!**

16:30 - 17:30 *Paper presentations:*

An Analysis of Student Preferences for Inverted vs Traditional Lecture 47

Brian Harrington, Mohamed Moustafa, Jingyiran Li, Marzieh Ahmadzadeh and Nick Cheng

Exploring the Coding Behavior of Successful Students in Programming by Employing Neo-Piagetian Theory 57

Natalie Culligan and Kevin Casey

Developing Testing-First Labs For a Less Intimidating Introductory CS Experience 66

Brian Harrington and Angela Zavaleta Bernuy

17:30 - 17:50 *Doctoral Consortium presentation*

A principled approach to the development of drum improvisation skills through interaction with a conversational agent 75

Noam Lederman

18:00 - 19:00 **Virtual Guided Tour**

Wednesday, 2nd December

16:00 - 16:20 *Doctoral Consortium presentation*

Programming “systems” deserve a theory too 78

Joel Jakubovic

16:20 - 18:00 *Panel discussion:* **Digressions on expression**

Invited speakers:

Steven Githens

Willie Payne

Lee Tusman

Sepideh Shahi

Tony Atkins

18:00 - 19:00 **Musical Soirée** (or **Matinée**, depending on your location!)

Thursday, 3rd December

16:00 - 16:40 *Paper presentations:*

Understanding the Problem of API Usability and Correctness Misalignment 82
Tao Dong and Elizabeth Churchill

Integrating a Live Programming Role into Games 108
Steve Tanimoto and Krish Jain

16:40 - 18:00 *Board game play session*

Undecided? - A board game to study intertemporal choices in software project management 122
Christoph Becker, Tara Tsang, Rachel Booth, Enning Zhang and Fabian Fagerholm

18:00 (optional: continued game play)

Friday, 4th December

16:00 - 16:40 *Paper discussion*

Undecided? - A board game to study intertemporal choices in software project management 122
Christoph Becker, Tara Tsang, Rachel Booth, Enning Zhang and Fabian Fagerholm

16:40 - 17:40 *Breakout groups discussion: On computing and communities*

17:40 - 18:00 PPIG Prizes & Close

Parallel Program Comprehension

Eric Aubanel

Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick
Canada, E3B 5A3
aubanel@unb.ca

Abstract

Parallel programming keeps growing in importance, driven both by changes in hardware and the increasing size of data sets. Hundreds of parallel languages have been proposed, but very few have taken hold beyond the language developers themselves. One reason for this is usability - that is the degree of ease with which one can develop and maintain parallel programs that are both correct and reach the desired level of performance. The few studies of parallel language usability have not been informed by a theoretical framework. Existing theoretical models of program comprehension need to be extended to parallel programming to help address the challenges of developing new languages, programming frameworks, development tools, and pedagogy. The contribution of this article is to motivate research on parallel program comprehension, and to suggest a way forward by expanding the two-level program/situation model of program comprehension to include a model of program execution and by applying the extensive work on human reasoning by Johnson-Laird to understand how people reason about parallel programs.

1. Introduction

The cognitive psychology of computer programming has been well studied since the 1970's, and has led to deep insight into how programmers design, build and understand software (Détienne, 2001). This knowledge is vital for the development of software engineering tools and techniques, for the design of programming languages, and for computer science education. Program comprehension is relevant to many programming tasks. When implementing a design, programmers need to read and assess what they have written. Design also frequently involves reuse, which requires comprehension of the code to be reused. Other programming tasks require program comprehension, such as modification to add features, improve performance or software quality, and of course debugging. In order to comprehend a program, a programmer constructs a mental representation based on the program text and the programmer's knowledge (Détienne, 2001).

Theories about the mental representation of computer programs have informed the research and development of software engineering tools (Storey, 2006). Existing theoretical models of program comprehension include knowledge stored in long term memory (language syntax and semantics, programming schemas) and the development of mental models of programs in working memory. These components were brought together in von Mayrhauser and Vans's Integrated Code Comprehension Metamodel (1994).

The mental model theories of program comprehension are based on theories of natural language text comprehension (Détienne, 2001). One important question is whether a text is represented mentally by its propositional structure or by its meaning. Johnson-Laird has made a strong case that it is the meaning that is represented, in the form of mental models (P. N. Johnson-Laird, 1983). For computer program comprehension the propositional structure is referred to as the program model and the meaning is represented by the situation model. This two-part model has been successfully applied to the comprehension of procedural and object-oriented programs (Détienne, 2001), but has not been studied for parallel programs. Understanding a parallel program requires additional work, such as reasoning about multiple streams of execution and awareness of execution at the machine level.

Parallel programming keeps growing in importance, driven both by changes in hardware and the increasing size of data sets (Asanovic et al., 2006). Hundreds of parallel languages have been proposed,

but very few have taken hold beyond the language developers themselves. One reason for this is usability - that is the degree of ease with which one can develop and maintain parallel programs that are both correct and reach the desired level of performance. The few studies of parallel language usability have not been informed by a theoretical framework (Mattson & Wrinn, 2008). Sadowski and Shewmaker's (2010) survey found that the existing literature on usability of parallel programming languages was inconclusive and that there were significant challenges in measuring usability.

The existing theoretical models of program comprehension need to be extended to parallel programming to help address the challenges of developing new languages, programming frameworks, development tools, and pedagogy. The contribution of this article is to motivate research on parallel program comprehension, and to suggest a way forward by expanding the two-level model of program comprehension to include a model of program execution and by applying the extensive work on human reasoning by Johnson-Laird to reasoning about parallel programs.

Sections 2-4 review three types of mental models. Section 2 reviews the mental model theory of program comprehension, and concludes with an illustrative example to discuss the extra work required in parallel program comprehension and introduce the idea of an execution model. Section 3 discusses the importance of machine models in parallel programming, and how they are related to the concept of notional machines. Section 4 briefly presents Johnson-Laird's understanding of human reasoning with mental models. Finally, Section 5 presents a proposal for an execution model component of program comprehension and how it might be used in reasoning about parallel programs.

2. Mental Models in Comprehension of Computer Programs

The experimental study of computer program comprehension dates back to the early '80s (Bidlake, Aubanel, & Voyer, 2020). Détiéne provides a thorough analysis of work up until the late '90s in her book *Software Design – Cognitive Aspects* (2001). She classifies work on program comprehension into approaches that use schemas, representing domain and programming knowledge, problem solving approaches, and the mental model approach. According to Détiéne (2001, ch. 6), "It appears that the mental model approach is the one that explains most completely the processes employed and the representations constructed in the course of understanding a program."

The mental model approach began with work by Pennington (1987). Pennington's experiments revealed that programmers construct a program model using control flow structures, when reading a program for the purpose of comprehension. When the comprehension stage is followed by a modification stage, which requires comprehension of the meaning of the program, a situation model is constructed. The situation model represents the program's data flow and goals. Any competent programmer can construct the program model. The situation model is constructed when the meaning of the program is important, and is more difficult to construct than the program model. The data flow is not as obvious as the control flow, and the function of the program is the hardest to discover.

Later work expanded Pennington's model by considering the effect of expertise, programming paradigm, and task (Bidlake et al., 2020). While both novice and expert programmers show no differences in the construction of the program model, novices do have more difficulty in constructing the situation model (Burkhardt, Détiéne, & Wiedenbeck, 2002). Object-oriented program comprehension does not proceed in the same way as procedural program comprehension, in that both program and situation models are constructed in parallel. Burkhardt et al. expanded the program model to include a macrostructure consisting of the control flow between functions. They expanded the situation model for OO programs to take into account objects and their interrelationships. The majority of program comprehension studies use program understanding, also known as read-to-recall as their task (Bidlake et al., 2020). The read-to-recall task is to remember program code after a study period, either by answering questions about the code or paraphrasing it. Other tasks used in studies can be classified as read-to-do, which includes modification, debugging, and classifying programs (Bidlake et al., 2020). As suggested in Pennington's study and confirmed in later work, the development of the situation model is more likely given a read-to-do task, where understanding the meaning of the program is important.

2.1. Illustrative Example

There are many parallel programming models, suitable for parallel execution using vector instructions and threads on multicore processors and graphics processing units, and processes across processors in a cluster. We use OpenMP as an easy to understand and popular shared memory programming model in our illustrative examples. Consider the nested loops in Figure 1 written in the C programming language, annotated with an OpenMP parallel directive (Liao, Lin, Asplund, Schordan, & Karlin, 2017).

```
double a[len][len];
\\ ...
#pragma omp parallel for private(j)
for (i = 0; i < len - 1; i += 1) {
    for (j = 0; j < len ; j += 1) {
        a[i][j] += a[i + 1][j];
    }
}
```

Figure 1 – C/OpenMP simple race condition example

We can examine this code as a miniature program comprehension exercise, and identify the components of the two-level program/situation model. We'll start by ignoring the OpenMP `pragma`. The program model contains both a micro- and a macro-structure (Détienne, 2001), but here only the former is relevant. The microstructure represents the surface details and the control flow of the program: two nested `for` loops updating elements of a two-dimensional array `a`. This model is built automatically by any programmer with syntactic/semantic knowledge of the language. The situation model has static and dynamic components. Here the dynamic situation model represents the data flow of the program and the static situation model represents the goal of the program. Construction of the situation model is optional, and takes more effort. It involves tracing updates to the matrix, where elements of each row are replaced by the sum of their value and their lower neighbour. In this small code sample this is also the goal of the program.

The OpenMP `pragma` tells the compiler to parallelize the outer loop by forking threads and assigning contiguous blocks of iterations to them. All variables are shared among threads by default, except for the iteration counter `i` (implicitly) and the counter `j` (using the `private` clause) of the inner loop, which are private. This adds to the program model the text itself, but not its meaning, other than its identification as a compiler directive. It adds to the situation model the parallelization of the outer loop and the knowledge that the iteration variables `i` and `j` are private to each thread, that `a` is shared, and that there is an implicit barrier at the end of the outer loop. Analysis of the data flow must now take into account multiple threads. This analysis reveals a problem, namely a data race. A thread working on row `i` could read from a value in row `i+1` while a thread working on row `i+1` is writing to the same memory location, leading to incorrect results.

Whereas the above analysis of the comprehension of the non-parallel code is based on mental structures for which there is considerable experimental evidence, there is no evidence that the analysis in the previous paragraph reflects how programmers think about parallel programs.

Full parallel program comprehension might even seem impossible. In the words of Skillicorn and Talia (1998), "An executing parallel program is an extremely complex object." There may be hundreds of threads executing concurrently, and threads may communicate with each other synchronously or asynchronously. The interleaving of memory accesses by multiple threads may change from execution to execution, which can lead to nondeterministic results in a faulty program. How can comprehension of such complex execution happen? We propose that the programmer builds mental models of representative cases of parallel execution, and then reasons about the correctness and meaning of the code using these models. We further propose to introduce a new component to the memory model theory, namely the execution model, which represents the execution of a program. In the example above, the execution

model would represent the execution of multiple threads and how they lead to a data race by reading and writing to the same locations of the `a` array.

3. Notional Machines and Machine Models

Texts written in a high-level programming language must be translated into machine instructions in order to be executed on a computer. This has important implications for the programmer's mental model. Any educated programmer knows that a single high-level instruction may be translated into multiple low-level instructions. However, as the programmer traces through some code and executes it in their working memory, they do so on an abstract mental representation of a machine that can execute the high-level instructions directly. This abstract machine has been called the **notional machine** in the context of computer science education. (Du Boulay, 1986): "A notional machine is a characterization of the computer in its role as executor of programs in a particular language or a set of related languages." (Sorva, 2013, p. 2). In this definition 'computer' refers to both hardware and system software (compiler/interpreter, operating system). There can be multiple notional machines for the same language, at different levels of abstraction. For example, it's possible to mentally execute a C program without considering how memory is divided into stack and heap. It's also possible to track the memory management with a lower-level notional machine.

Notional machines, together with the literature on knowledge mental models, are valuable for computer science pedagogy. Experiments have shown that novice programmers' mental models are inadequate (Sorva, 2013), and computer science instructors commonly observe students' superstitions about the behaviour of the notional machine. A challenge in educating programmers is to help them build viable notional machines, so that they can accurately reason about programs and simulate them mentally.

We argue that something akin to notional machines is relevant for expert program comprehension. This knowledge can be called a machine model. It adds the cost of execution to the programming schema knowledge that experts possess, including the contribution of compilers, operating systems, runtime software, and hardware. For instance, it allows programmers to assess the overhead of function execution and the desirability of function inlining. For a programmer performing incremental parallelization of a sequential program, knowledge of the machine model allows them to reason about whether parallelizing a loop is worthwhile, based on the cost of the parallelization overhead. The machine model also supports the dynamic aspects of the mental model of the expert programmer, when faced with a comprehension task. While comprehension relies to a large extent on static knowledge of programming plans, it also can involve the dynamic aspects of the mental model, particularly data flow. Mental execution may be required if the programmer is unfamiliar with a programming plan, either because the plan doesn't follow the rules of programming discourse or because the programmer has not seen the plan before (Détienne, 1990). For parallel programming, reasoning about the execution of the program is crucial in assessing correctness and performance, as in the example in Figure 1.

Previous work has not emphasized the dynamic aspects of program comprehension. We argue that it would be fruitful to move comprehension of data flow from the situation model into a separate component called the execution model. We believe that the construction of the execution model in working memory depends on the machine model that is used, but we will focus on the former in what follows.

4. Mental Models for Reasoning

According to Johnson-Laird there are at least three types of mental representations:

1. Propositional representations (strings in a natural language)
2. Images, which are perceptions from a particular point of view
3. Mental models, "which are structural analogues of the world" (P. N. Johnson-Laird, 1983)

Mental models can be manipulated, and can be used to reason without formal logic. The structural analogues are usually two or three-dimensional icons. As Johnson-Laird explains in a 2010 review, "A

visual image is iconic, but icons can also represent states of affairs that cannot be visualized", such as "the abstract relations between sets that we all represent." (P. N. Johnson-Laird, 2010). These icons are not restricted to a single point of view but can be manipulated.

Johnson-Laird's main concern is with understanding how people reason using mental models. His research has convincingly demonstrated that humans don't reason using formal logic (the "doctrine of mental logic"). He makes an important point about the complexity of reasoning, which is relevant to the comprehension of parallel programs: "Almost all sorts of reasoning,..., are computationally intractable. As the number of distinct elementary propositions in inferences increases, reasoning soon demands a processing capacity exceeding any finite computational device,..., including the human brain" (P. N. Johnson-Laird, 2010). Humans deal with this complexity by constructing representative mental models.

Consider the following sentence (P. N. Johnson-Laird, 2004):

> The cup is on the right of the spoon

It might be reasonable to postulate that the meaning of this sentence is represented by the reader in a mental language. What if we add three more sentences:

> The plate is on the left of the spoon.
> The knife is in front of the cup.
> The fork is in front of the plate.

and ask the question: what is the relation between the fork and the knife? Answering this question requires spatial reasoning, not reasoning about language. A mental model for these four premises can be given as:

> plate spoon cup
> fork knife

This model can then be used to give the answer to the question: the fork is on the left of the knife.

Consider now the same four premises with a small change to the second one:

> The plate is on the left of the cup.

These premises can be represented with at least two models, the model above and this one:

> spoon plate cup
> fork knife

The answer to the question is still the same (the fork is on the left of the knife), however the reasoning is more difficult, because more than one model needs to be considered. The extensive literature on reasoning with mental models is likely to contain insights into how programmers reason about code.

5. Execution Model

Comprehension of the illustrative example in Figure 1 requires applying knowledge of how the loop iterations are assigned to threads (the C/OpenMP machine model). Comprehension of the parallel aspect of this code requires reasoning about the data access pattern of the threads, in other words the parallel data flow. This comprehension can be done independently of the understanding of the meaning of the program, which forms the static part of the situation model. The programmer, faced with the task of verifying the correctness of this code, could conceivably focus on the parallel data flow without

bothering to understand the meaning of the code. In contrast, the essence of the situation model in text understanding is the situation of the text, that is its meaning.

We propose to carve out a separate mental execution model, which would include the dynamic aspect of the situation model, namely the data flow. The separation of the execution model could account for the separate understanding of the behaviour of the the program from its meaning. This behaviour occurs at two levels: data flow in the program text and data flow in the computer. The latter behaviour is key to understanding parallel programs. It is also key to understanding the performance of any program, for example the flow of data between levels of the memory hierarchy. The first level bears more discussion. While the data flow of a program can be considered between variables in a program, it is often considered in the context of one or more data structures. Part of the comprehension of a non-trivial program is understanding the data structures, which are not evident in the surface of the text. They form part of the understanding of the underlying algorithms used in the program. Data structures do not seem to have been considered in models of program understanding (Détienne, 2001, ch. 6, footnote p. 94).

The execution model can be divided into three layers of abstraction. At the top, it represents the behaviour of the operations on the data structures. In the middle, the data flow between variables in the program text. At the bottom, the data flow in the processing elements of the computer.

The comprehension model in Table 1 modifies Burkhardt et al.’s version of the program/situation model (2002), which accounts for object oriented programs, by moving data flow from the situation model to the execution model, and adding two aspects of data structures to the execution and situation models. The identification of the data structure refers to the program’s meaning, and is part of the situation model. The behaviour of the data structure, that is its mutation by one or more threads of execution, is part of the execution model. The parts of the execution model that are unique to parallel programs are discussed in the following example.

Program Model	Execution Model	Situation Model
Microstructure:	Data structures (behaviour)	Data structures (identification)
Program statements	Data flow (text)	Program goals
Control flow	Data flow (machine)	Domain objects (classes)
Macrostructure:	Parallel programs:	Relations between objects
Functional structure	Decomposition	Communication between objects
Control flow between functions	Communication	

Table 1 – Three-part comprehension model.

5.1. Another Illustrative Example

Consider another C/OpenMP example (Mattson & Meadows, 2014) in Figure 2, where `p` is a struct containing a `*next` pointer and a pointer to some payload that will be used in the execution of the `process()` function.

Ignoring the parallel directives for the moment, the following comprehension process can be sketched. The program model would identify a loop that continues as long as pointer `p` is not `NULL`, the initialization of pointer `p` and its updating in the loop, and its passing to function `process`. The situation model would contain the meaning of this code as the processing of data that is stored in a linked list. The role of the execution model could depend on whether the programmer is a novice or an expert. An expert would not likely have to mentally invoke the dynamic aspect of the linked list; its static meaning should be sufficient. A novice who was unfamiliar with linked lists, might need to trace execution of the linked list, that is invoke the execution model, on their way to understanding its meaning.

The execution model is more pertinent to the parallel version of the code. Adding parallel directives doesn’t change the meaning of the code, so the situation model is unchanged. It does require understanding how the code executes in parallel, hence the execution model is vital. This understanding

```

#pragma omp parallel{
  #pragma omp single{
    node * p = head;
    while (p) {
      #pragma omp task
      process(p);
      p = p->next;
    }
  }
}

```

Figure 2 – C/OpenMP parallel linked list example

includes the forking of threads at the beginning of the code, followed by the restriction of the code's execution to a single thread. Each iteration of the `while` loop involves creation of a new OpenMP task which is scheduled by the OpenMP runtime on another thread. In other words, one thread dispatches the tasks, which are executed by multiple threads.

Decomposition and communication are two key parts of the execution model which are found only in comprehension of parallel programs. The former is an essential part of any parallel program (Aubanel, 2016). The example in Figure 2 exhibits a trivial decomposition into tasks. Data decomposition also features prominently in parallel computing. The example of Figure 1 exhibits decomposition of the two-dimensional array into blocks of rows. While communication is not explicit in shared memory programs such as these, it can be present implicitly in the ordering of memory accesses of the threads. Communication should form an explicit part of the execution model of distributed memory programs, such as those that involve message passing.

5.2. Reasoning with the Execution Model

A mental model represents a possibility (P. Johnson-Laird, 2001). What are the possibilities for the execution model for the example in Figure 1? Knowledge of the semantics of the OpenMP `parallel for` directive reveal that the default scheduling is to partition the the iterations of the outer loop into contiguous blocks, one per thread. The execution model would represent the decomposition of the 2D array into blocks of rows and the dependence between blocks arising from `a[i][j] += a[i + 1][j];`. It also needs to represent the dynamic behaviour of the threads. A simplifying (but not generally correct) assumption is that threads start at exactly the same time and proceed in lock step. This means that a thread working on the last row of its block would use values in the next row which had already been updated by the thread working on the next block, yielding incorrect results. This single possibility can be used to identify the data race.

Programmers can't possibly follow the execution of multiple threads, especially since the relative timing of the threads can vary from one execution to the next. Instead, programmers construct mental models for representative cases, each corresponding to an interleaving of a particular number of threads. Reasoning gets more difficult as the number of mental models increases (P. Johnson-Laird, 2001). The code in Figure 3 (Liao et al., 2017) represents a more subtle race condition, where `xa1` and `xa3` point to two elements of an array, where `xa3 - xa1 = 12`.

As hinted by the note, there is a dependence between iterations 0 and 5, where `xa1[521]` and `xa3[533]` point to the same location. This is not a problem if iterations 0 and 5 are handled by the same thread; a race condition occurs if they are handled by different threads. Assuming the same static scheduling of the loop iterations, this means that iterations 0 and 5 would be handled by the same thread if the total number of threads is less than 36. This requires mental models for two possibilities: number of threads < 36 and ≥ 36 . The programmer could miss the race condition by using a single possibility of less than 36 threads, which could easily happen if they are used to working with a small number of threads.

```

#define N 180
int indexSet [N] = {
//Note: indexSet[5] - indexSet[0] = 533-521= 12
    521, 523, 525, 527, 529, 533,
    547, 549, 551, 553, 555, 557,
// omitted code here ...
};
#pragma omp parallel for
for(i=0; i< N; ++i){
    int idx=indexSet[i];
    xa1[idx]+=1.0;
    xa3[idx]+=3.0;
}

```

Figure 3 – C/OpenMP tricky race condition example

6. Acknowledgements

The author acknowledges the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2018-04811.

7. Conclusion

Understanding a program involves more than determining its meaning. It also involves understanding its dynamic behaviour. This is particularly important for the comprehension of parallel programs, where the behaviour of multiple streams of execution must be understood. We have argued that adding an execution model to the current mental model theory would account for this understanding. This includes data structures and their decomposition in the mental representations of parallel programmers.

Program comprehension involves knowledge stored in long term memory in addition to the mental representations created in working memory during comprehension. Understanding a parallel program requires a mental model of the parallel system, which could be viewed as a notional machine for expert programmers. Different parallel programming languages have different machine models (Aubanel, 2016; Skillicorn & Talia, 1998). The impact of a language's machine model on the comprehension of its execution would be worth studying. Languages that have a high level of abstraction may make it difficult to understand what is happening at the machine level, which is necessary in order to reason about performance. Languages at a lower level of abstraction may involve a tradeoff between exposing execution at the machine level and increasing the cognitive load of having to understand multiple streams of execution.

How do programmers actually reason about a parallel program's behaviour? They likely use their knowledge about the programming language's machine model to construct representative cases. This could be similar to how people reason about natural language texts, and could lead the way to understanding what makes one program harder to understand than another, and what kind of mistakes even expert programmers make. Understanding how parallel programmers reason, and the challenges they face, could lead to the development of representations that would aid in comprehension. This could include diagrammatic representations, to show the decomposition of data structures and the communication between tasks. Without knowing anything about programmers' mental representations, it's hard to predict whether a proposed representation would be helpful.

8. References

- Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., ... others (2006). *The landscape of parallel computing research: A view from Berkeley* (Tech. Rep.). Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.
- Aubanel, E. (2016). *Elements of parallel computing*. CRC Press.

- Bidlake, L., Aubanel, E., & Voyer, D. (2020, July). Systematic literature review of empirical studies on mental representations of programs. *Journal of Systems and Software*, 165(110565).
- Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2), 115–156.
- Du Boulay, B. (1986, February). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1), 57–73.
- Détienne, F. (1990). Expert Programming Knowledge: A Schema-based Approach. In *Psychology of Programming* (pp. 205–222). Elsevier.
- Détienne, F. (2001). *Software Design–Cognitive Aspects*. Springer Science & Business Media.
- Johnson-Laird, P. (2001). Reasoning with Mental Models. In *International Encyclopedia of the Social & Behavioral Sciences* (pp. 12821–12824). Elsevier.
- Johnson-Laird, P. N. (1983). *Mental Models*. Cambridge University Press.
- Johnson-Laird, P. N. (2004). The history of mental models. In *Psychology of reasoning* (pp. 189–222). Psychology Press.
- Johnson-Laird, P. N. (2010, October). Mental models and human reasoning. *Proceedings of the National Academy of Sciences*, 107(43), 18243–18250.
- Liao, C., Lin, P.-H., Asplund, J., Schordan, M., & Karlin, I. (2017). DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '17* (pp. 1–14). Denver, Colorado: ACM Press.
- Mattson, T., & Meadows, L. (2014). A “Hands-on” Introduction to OpenMP. Retrieved 2020-03-25, from https://extremecomputingtraining.anl.gov/files/2016/08/Mattson_830aug3_HandsOnIntro.pdf
- Mattson, T., & Wrinn, M. (2008). Parallel programming: can we PLEASE get it right this time? In *Proceedings of the 45th annual Design Automation Conference* (pp. 7–11). ACM.
- Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19(3), 295–341.
- Sadowski, C., & Shewmaker, A. (2010). The last mile: parallel programming and usability. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (pp. 309–314). ACM.
- Skillicorn, D. B., & Talia, D. (1998, June). Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2), 123–169.
- Sorva, J. (2013, June). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2), 1–31.
- Storey, M.-A. (2006, September). Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3), 187–208.
- von Mayrhauser, A., & Vans, A. M. (1994). Comprehension Processes During Large Scale Maintenance. In *Proceedings of the 16th International Conference on Software Engineering* (pp. 39–48). Los Alamitos, CA, USA: IEEE Computer Society Press.

What the Mouse Said:

How Mouse Movements Can Relate to Student Stress and Success

Natalie Culligan
Department of Computer
Science
Maynooth University
natalie.culligan@mu.ie

Kevin Casey
Department of Computer
Science
Maynooth University
kevin.casey@mu.ie

Abstract

Stress in students may be a useful indication for when a student is struggling and in need of academic intervention. Investigating differences in student behaviour in stressful and comparatively less stressful environments could be helpful in understanding the processes involved in learning to code, and combatting the high levels of drop-out and failure in undergraduate computer science. In this paper we will discuss the mouse movement data gathered from Maynooth University Learning Environment (MULE), our in-house, browser-based pedagogical environment for novice programmers, during the time period February to May of 2019. This included 5 supervised, scheduled lab sessions and two in-lab examinations. The data was used to examine 21 different measurements of student behaviour, for example, by measuring efficiency of the mouse path, or the time between mouse click-down and mouse click-up. These features were used to build a Deep Neural Net that classifies sequences of mouse movements as being either from a more stressful environment or a less stressful one by training the classifier on data from examination situations and regular weekly lab situations, with the goal of comparing how students behave in environments with different levels of student comfort. The classifiers had an average accuracy of 61.9% but was more successful with students who performed poorly in their lab examinations. To further examine this connection between mouse movement, stress and student outcome, a second classifier was built to classify students as being in the high or low 50% of lab-exam grades in the module, with an accuracy of 69%.

1. Introduction

In this study we use data collected by Maynooth University Learning Environment, or MULE (Culligan, Casey 2018). MULE is an online, browser-based pedagogical desktop environment which has been used in multiple first-year coding modules. We received clearance from the University Ethics Committee to collect mouse movements from students as they learn to code from the 29th of February until the 3rd of May in the Introduction to Programming II module (taught in Java) with 250 students completing the module. The students were informed about the use of their data and were asked to consent at the beginning of the semester. All students who completed the module chose to participate in the study.

Using the mouse movement data collected by MULE, a Deep Neural Net (DNN) binary classifier was built to detect if a sequence of mouse movements is from a stressful (in-examination) or less stressful environment (in-lab). The classifier is not universal. It needs to be trained on a student's own data and does not work on all students. This was expected, as stress and comfort are subjective and not all students will experience stress in the same way during an examination. Students may also have different mouse use "styles", which makes it harder to generalise mouse behaviour caused by stress. We must also consider that some students are not stressed during an examination and may even be less stressed than in a normal lab situation.

The classifier works very well for some students and poorly for others, with an average increase of 11%-12% over the accuracy baseline of 50%, an average accuracy of 61.9% for classifying both in-lab and in-examination sequences. The classifier was moderately successful but interestingly the classifier was more successful for students who did poorly in the module. To further explore this, we built a second DNN to investigate if the mouse movement data could be used to classify students as being in the top or bottom 50% of module grades. This classifier was more successful than the stress

classifier, classifying students as being in the top or bottom 50% of the module Continuous Assessment grades with an accuracy of 69%, over an accuracy baseline of 50%.

In this paper, the following questions will be explored in relation to the gathered mouse movement data.

1. *Are there differences in mouse movement behaviour of students between lab and exam situations, and can this be a first step in a classifier for stressed students?*
2. *Are there differences in student mouse behaviour and stress in students in CS1 between students who perform well in-lab examinations and written exams, and those who perform poorly?*

The null hypothesis for these questions are as follows:

1. *The results from the Deep Neural Net for classifying sequences of mouse movements sequences as being from stressful or not stressful environments performed no better, or not significantly better than random chance.*
2. *The results from the Deep Neural Net for classifying individuals as being in the top or bottom performing 50% of students performed no better, or not significantly better than random chance.*

2. Motivation and Related Research

Stress in students may be a useful indication for when a student is struggling and in need of academic intervention. Intervention for students experiencing unusual amounts of stress could be helpful in combatting the high levels of drop out and failure in undergraduate computer science (Beaubouef et al, Biggers et al, Giannakos et al, Hembree et al, Kinnunen et al). This is the first of our studies into student behaviour as they learn to code, and in this study we focus on mouse movement. There are studies that suggest that mouse movement is linked to stress and mood (Sun *et al.*, Wahlström, *et al.*, Yamauchi). In this paper, we are interested in examining student mouse movement in stressful and less stressful environments to try and gain insight into behaviours that indicate stress, and investigate if this is related to student performance.

2.1. Stress Levels in Students

Computer science courses have been reported to have low levels of retention in comparison to other subjects (Giannakos *et al.*, Kinnunen, *et al.*). Research suggests that student comfort is a useful signifier of student success and retention (McCracken *et al.*, Tenenberg, *et al.*, Wilson and Shrock), and that stressful situations such as examinations can cause a student to perform below their ability (Beilock and Carr).

Beilock and Carr discuss the connection between anxiety and a loss in academic performance, and suggest that situation-related worries – such as examination stress or anxiety – can result in a loss of focus on task at hand as the working memory is occupied. Alternatively, it has also been suggested that over-attending to performance, overthinking tasks usually performed automatically, can lead to underperforming in an uncomfortable or stressful situation. Beilock *et al.* discuss how a more stressful or anxious state can also affect tasks that are usually performed in an automated fashion, without the subject thinking too much about it – their paper mentions soccer players' dribbling. We propose that mouse movement could be considered in a similar manner.

Connolly *et al.* found that in their study of 86 computing undergraduate students, 44.4% reported not feeling relaxed when using computers, suggesting that research into this area would be beneficial to a significant portion of the student population.

Bergin and Reilly examined 15 factors in predicting if a student is likely to pass or fail. One of the most statistically significant factors in predicting success was comfort level, in relation to how the student felt about the course. This was measured through cumulative responses to questions about the students' understanding and difficulty completing lab assignments.

2.2. Mouse Movement and Stress

There is prior evidence of a link between student stress and comfort level and their mouse movements. Sun *et al.* constructed a Mass Spring Damper model for the human arm - essentially a model for approximating arm motion and stiffness which could be fed with data from mouse movements. Using arm stiffness as a proxy for stress in the user, the authors report that their method was tested across a variety of prescribed stress tasks. The classifier worked when generalised but was more effective when trained and tested separately for each user.

Yamauchi claims there is both psychological and neurological evidence to suggest that mouse trajectories can be used to assess affective states, such as anxiety. The results of their study show that temporal features, such as speed of mouse movement, and spatial features such as direction change were both indicative of the user’s state of anxiety. The researchers in this paper ran a separate analysis for male and female users and found different indications of state anxiety, with female subjects being more inclined to use a less efficient mouse path when anxious, and male subjects being more likely to change their mouse velocity.

Kapoor *et al.* use a specialised pressure mouse with additional sensors to detect frustration in subjects as they attempt to complete a towers of Hanoi puzzle computer game. The game includes an “I’m frustrated” button for the users, which is used to associate behaviour with frustrated state. The resulting classifier can predict frustration at an accuracy of 79%, outperforming the random classifier (58%).

3. Research Design

The goal of this study was to examine the relationship between student mouse behaviour, student outcome, and comfort level in students in CS1, an introduction to programming module. Using the data from MULE, we constructed a Deep Neural Net binary classifier to classify sequences of mouse movements as being from a stressful environment or a less stressful one.

Data Type	Description
userID	The anonymous ID assigned to the student
dumpID	The ID of the dump from student session to the database
sessionID	An ID assigned to the session when a student logs in until they log out
Time	Timestamp of when the event took place, not when it was stored
Type	Mousemove, mouseup or mousedown
X	X co-ordinates of the mouse’s current position
y	Y co-ordinates of the mouse’s current position

Table 1: Mouse movement data features

MULE was used to collect mouse movement data from students as they learned to code in an authentic learning environment. To use the system, the students sign in through their Moodle accounts from any internet browser on any machine, they do not need to be in the university computer labs. The system is a desktop-like environment simulated within the browser, where they can view assignments from a designated application, use a text editor to write code for the assignments, and compile, run and automatically evaluate their code, receiving a grade and automated feedback if their code has errors. The students use the mouse to navigate the system, to open assignments, open the code editor, and to save, compile, run and evaluate from drop down menus. As the student works, the system automatically stores their mouse movements, along with a timestamp and an anonymised user key to allow for cross session comparisons. Stored mouse movements are sent to the database every 30 seconds, or as soon as the user tries to log out or close the system tab. The system collects mouse movement data as shown in Table 1. Anonymised data on students’ performance in the module was also collected, specifically how they performed in the written examination, in weekly labs and in-lab examinations. The total number of students who completed the second semester was 250, of which 196 are included in this study. We removed data from students who did not participate enough for their data to be used in the study, including:

1. *Students who did not take both in-lab examinations*
2. *Students who did not complete the course*
3. *Students who participated in less than two lab sessions*

Students have labs for 3 hours once a week for 12 weeks per semester. The students began using the system in the first semester of the academic year 2018/2019 and used the system for the rest of the academic year. The mouse movement data set we are examining in this paper is from the second semester, from the 29th of February until the 3rd of May. This time period includes 5 regular weekly labs and 2 in-lab examinations. We compare mouse data from students in a regular lab situation versus mouse data from an examination situation, to examine the differences between coding when in situations with different levels of comfort. Both situations are in the same physical space, but with different rules. The students are not allowed to speak to each other, ask for help from demonstrators or look back at their previous work during the examination situation, but are encouraged to do so during regular labs. One of the authors worked as a demonstrator in the labs where this research took place to ensure the coding environment was working correctly, and to assist the students.

We recorded mouse data from students as they worked in scheduled labs, scheduled examinations, and outside of these times. The data from outside of the lab is not discussed in this paper. Data outside scheduled labs and examinations may be the result of users other than the signed-in student and/or very different mouse set up (touch screen, touch pad, or different desk size, for example). Students may also be working in very different situations due to environmental noise, distractions, or caretaking responsibilities, for example.

The mouse data from each student is divided into sequences to be assessed by the classifier. Each sequence begins with any mouse movement and ends with a mouse click-up, and any sequence that is longer than 1450ms is rejected to avoid evaluating sequences from when the student is idle. This time limit was chosen through trial and error, and found the classifier worked best with sequences under this time limit. Tests are run on each sequence to find various metrics for the users' behaviours. Metrics include SequenceSpeed, ClickTime and Efficiency. Each sequence also has an identifier, as in-lab, in-examination or out-lab. Once we have the metrics for each of the sequences, they are used to train and test the Deep Neural Net.

We used a total of 21 different features in our classifier.

Features

1. *AngleVariance1:*

Finds all the different angle changes from one movement to the next (with precision of 2 digits) within a sequence and returns the total number of unique angles.

2. *AngleVariance2*

Same as above, but the total number of angles returned.

3. *AngleVariance3*

The ratio of total angles to unique angles.

4. *VarianceDistance1*

Finds the optimal distance between every set of two mouse movements to 1 decimal place and returns the number of all unique distances.

5. *VarianceDistance2*

Same as above but returns the number of all distances.

6. *VarianceDistance3*

The ratio of all unique distances and all distances in the sequence.

7. *Overshoot-x*

Measures how far a user "overshoots" with the mouse in the direction they are moving the mouse in, along the X axis. If a user moves from point a to point b within a small window of time, point b being where they click the mouse, if at some point during this journey they move further along the x-axis than where they ended, this is recorded as an *Overshoot-x*.

8. *Overshoot-y*

Same as *Overshoot-x*, but along the y axis.

9. *Overshoot*

The square root of *Overshoot-x* and *Overshoot-y* squared and added.

10. *OvershootDirectionAngle*

Finds the angle of the overshoot.

11. *SequenceSpeed*

The total distance travelled divided by the total time.

12. *SequenceDuration*

The time duration of the sequence.

13. *DistanceTravelled*

The true distance travelled during the sequence.

14. *OptimalDistance*

The distance in a straight line between the start and end points of the sequence.

15. *Efficiency*

Optimal distance divided by total distance travelled.

16. *Direction*

The direction from the first point in the sequence to the last.

17. *DirectionAngle*

The direction angle between the starting point and the ending point of the sequence.

18. *AngleDifference*

The absolute value of *DirectionAngle* subtracted from *OvershootDirectionAngle*.

19. *ClickTime*

The time between click down and click up.

20. *Hesitate*

The amount of time the mouse stalls before the user clicks.

21. *ClickRatio*

This is *Hesitate* divided by *ClickTime*

Yamauchi's paper '*Mouse Trajectories and State Anxiety: Feature Selection with Random Forest*' found that speed and direction were indicators of a subject's emotional state. Our features are chosen to examine this connection, with features such as *DistanceTravelled* and *ClickTime* relating to speed, and *DirectionAngle* and *OvershootDirectionAngle* relating to direction. The paper also discusses tracking direction change, x-overshoot, y-overshoot, which we replicated in our experiment with features such as *Overshoot-x*, *Overshoot*, *DirectionAngle* and *DirectionAngle*. Beilock et al discuss how a more stressful or anxious state can also affect tasks that are usually performed in an automated fashion. We investigated this with the features *VarianceDistance1*, *VarianceDistance2*, *VarianceDistance3*, to give us insight into how much the subject changed their speed, and the features *AngleVariance1*, *AngleVariance2* and *AngleVariance3* to investigate how often the subject changed direction, perhaps due to confusion or indecisiveness as a result of stress or discomfort.

As per Sun *et al.*, we trained our classifier per user, instead of building a generalised stress classifier. Our initial experiments involved a general classifier using a large subsection of the data from all students, but this classifier did not perform significantly better than random chance. To build a classifier for a user, we selected all the sequences from in-examination, and then a random selection of sequences of an equal amount from in-lab, or vice-versa, depending on the imbalance of data categorised as in-lab or in-examination. The features we get from the mouse movements of each student are then used to train and test a deep neural net, built in Python using TensorFlow (Abadi, Martín, *et al.*).

For most students, we have much more in-lab data than in-examination, so we take a random sample of the in-lab data equal to the size of the in-examination data. We used TensorFlow's *DNNclassifier* module, with 3 hidden layers of 10 units, a batch size of 5 and 2000 epochs. The classifier outputs a 1 if the mouse movement sequence is classified as in-lab and 0 if the sequence is classified as in-exam. When running the classifier for each student, we wanted to ensure that the results were not due to chance, or a "lucky" selection of test data from the total data set. To combat this, we selected a subsection of the data as test data, and rejected it if it was not 50/50 in-lab and in-exam, again to avoid

good results that are just the result of a classifier only choosing one classification, regardless of feature input. To check that the variance for the classifier results was low, and we were not reporting outliers, the classifiers were run in sections of ten, and the variance within results was checked. The variance for all users was 0.05 or less, with one exception that had a larger variance of 0.13. We performed multiple sets of ten, checking the variance on the cumulative results. For each student, the classifier was run 60 times, with a different random division of training and test data with no increase in variance over 0.016 between the first 10 and the final 60.

4. Discussion of Classifier Performance

The classifier works very well for some students and poorly for others, with an average increase of 11% to 12% over the accuracy baseline and an average accuracy of 62.9% for classifying both in-lab and in-examination sequences. However, for some students that performed poorly in their lab examinations, we found the classifier could work 30% over baseline. On examination of the results, it became apparent that the classifier was more successful with the students who performed poorly in the module than those who performed well. One of the possible reasons for student stress during exams is that they may be unable to use their usual method of solving coding problems. Some students will take previously written code, copy it and rewrite it to complete the given task. During exams the students no longer have access to their previous code. They may panic when they find they cannot use their usual strategy (though they are informed beforehand of the format and rules of the exam), or they may be experiencing additional strain on their working memory. This strain may come from the extra work now being performed by the student. For example, they can't copy a while loop from previous work, so instead they struggle to remember how to write one. The student is not comfortable and familiar with the computer science concepts needed to construct the code to solve the exam question and has been relying on 'tinkering', a technique used by students as described by Perkins *et al.* and Jadud.

4.1. Stress Classifier

When examining the results of the classifiers, differences between the high-performing and low-performing students became apparent. Table 2 shows the average classifier of two groups, the top 50% of grades and bottom 50% of grades. This was done for Continuous Assessment, written exam and total module grade, and repeated with the top and bottom 40%, 30%, 20% and 10%.

	Module High	Module Low	Written Exam High	Written Exam Low	CA High	CA Low
50%	61.875%	62.7913%	61.3518%	62.627%	60.8315%	63.1473%
40%	60.8037%	62.6183%	61.1019%	61.949%	60.7157%	63.8293%
30%	60.1556%	62.6878%	60.9173%	62.6955%	59.7906%	63.9333%
20%	60.0027%	62.5255%	59.8544%	62.8666%	59.7657%	63.4562%
10%	58.8089%	62.8847%	58.2153%	62.643%	59.4642%	65.3041%

Table 2: Comparison of the high and low performing students

In all groups, and with all three grade types, the lower grades group have more successful classifiers, with the difference becoming more pronounced as we look at smaller subsections. We suspect that the reason students in the lower-grade groups are easier to classify is because these students may experience additional strain when writing code, perhaps due to exam anxiety, or a lack of comfort with the material. In the paper "*On the causal mechanisms of stereotype threat: Can skills that don't rely heavily on working memory still be threatened?*", Beilock, *et al.* claim that while overloaded working memory does not directly affect procedural skills because it is not reliant on working memory, over-attention to procedural skills does impact the subject's performance – a worried student may overthink their behaviour, causing changes in their mouse movement.

4.2. High Low Grade Classifier

We were interested in the possible connection between mouse movements and student grades, from the apparent relation between classifier success and the students' performance in the module shown in Table 2. We suspected that the results indicated a relation between mouse movements, specifically indications of stress in exams, and student grades. There is previous work (Casey) to suggest that low-

level keystroke data can be used to improve grade classifiers, so we wanted to examine if mouse movement data could also be used. To investigate this, a trio of DNN classifiers were created to predict the outcome of students in:

1. *Continuous Assessment (coding exercises, and lab exams),*
2. *End of year written exams*
3. *The module overall.*

The DNN uses the same configuration as the stress classifier. We tried other configurations, including increasing the number of hidden units, but found this was the most successful setting. The DNN classifies each student into one of two categories – either the higher or lower 50% of the class, divided by the results in order. For this dataset we calculated the average of each of the features in the table for in-lab and out-lab. We found this gave the best results, possibly because the indicator of a student who does well or poorly is the difference, or the similarity of the behaviour between regular labs and exams, in line with the findings that the students who did poorly were more easily classed by the classifier.

Grade	Higher 50%	Lower 50%	Classifier Results
Written Exam	63% and over	61% and under	0.588333333
Module Total	59% and over	58% and under	0.656666667
Continuous Assessment	54% and over	53% and under	0.693333333

Table 3: Results of classification

Like the previous classifier, the high/low classifier was run 60 times, each time randomly selecting the training set and the testing set. Like the stress classifier, the randomisation was written to ensure that the testing data set would always be 50% from each classification, to avoid misleadingly high or low results from a classifier only choosing one classification.

5. Discussion of Research Questions

1. Are there differences in mouse movement behaviour of CSI students between lab and exam situations, and can this provide insight to the different comfort levels experienced by students in these environments?

The DNN classifier was mildly successful, implying that there is at least a weak link between mouse movement and comfort level. Students may still be stressed in lab situations, but because the classifier was more successful with students who did poorly in their lab examinations, we believe this is evidence that the classifier works as an indicator of stress – we believe that students who are taking examinations that they are not doing well in are more likely to be experiencing stress than others. We can reject the null hypothesis, as the classifier is more successful than a random chance classifier.

2. Are there differences in student mouse behaviour and comfort level in students in CSI between students who perform well in-lab examinations and written exams, and those who perform poorly?

The classifier is more effective with students who perform poorly than those who perform well. We would expect students who do poorly in the module to be more stressed in examinations than students who are comfortable with the material and are performing well. To examine this further we built a second DNN classifier and found that we were able to classify students into high/low performing groups with 69% accuracy. We reject the null hypothesis as the high/low classifier is more successful than random chance.

6. Conclusions and Future Research

In this paper, we have reported on the construction of a moderately successful Deep Neural Net that classifies sequences of mouse movements as being from a stressful or less stressful environment. While other researchers have published work on the connection between mouse movements and stress, to our knowledge this is the only study of mouse movements and stress that uses mouse data gathered

outside of closed experimental environments. From the analysis of the results, we found a connection between mouse movements and a student's grades, especially grades for practical coding assignments.

The classifiers in their current state are not a useful mechanism for detecting stress in students, or for predicting if students will be in the high or low 50% of grades. However, in the construction of these classifiers, we have found mechanisms that will contribute to the construction of models of successful students, and classifiers for students in need of academic intervention. This study is part of a larger project to examine the relationship between student behaviour when learning to code and student success and retention. Our coding environment gathers data beyond mouse movement, including keystrokes, compilation and run results, and returned errors. Other research in this area has used data such as keystrokes to predict student outcome (Casey), and from our work in this paper, which suggests a connection between student success and comfort-level, we believe this data will give further insight to student behaviour in stressful situations. Further work can be done in relation to the mouse analytics performed so far. We are currently refactoring our recording of mouse data so that we can capture additional data in order to attach more meaning to mouse sequences. This would, for example, allow us to distinguish between a mouse sequence that led to a file being saved, versus a mouse sequence that led to a compilation of student code.

We believe there is huge potential for study of this data, which is gathered from an authentic learning environment, as students learn to code. With continued research, we plan to build a larger model of the behaviour of novice programmers as they learn to code, with the potential for an integrated classifier in our coding environment that will alert course coordinators to a student in need of intervention. We hope the construction of a model of successful students will be a useful way to inform and build curriculums that best help students achieve their potential.

5. References

Abadi, Martín, *et al.* (2016) Tensorflow: A system for large-scale machine learning. 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16).

Beaubouef, Theresa, and John Mason. (2005) Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin* 37.2 103-106. DOI: <https://doi.org/10.1145/1083431.1083474>

Beilock, Sian L., and Thomas H. Carr. (2005) When high-powered people fail: Working memory and “choking under pressure” in math. *Psychological science* 16.2 01-105. DOI: <https://doi.org/10.1037/e537052012-380>

Beilock, Sian L., *et al.* (2006) On the causal mechanisms of stereotype threat: Can skills that don't rely heavily on working memory still be threatened?. *Personality and Social Psychology Bulletin* 32.8 1059-1071. DOI: <https://doi.org/10.1177/0146167206288489>

Bergin, Susan, and Ronan Reilly. (2005) Programming: factors that influence success. *ACM Sigcse Bulletin* 37.1 411-415. DOI: <https://doi.org/10.1145/1047344.1047480>

Biggers, Maureen, Anne Brauer, and Tuba Yilmaz. (2008) Student perceptions of computer science: a retention study comparing graduating seniors with cs leavers. *ACM SIGCSE Bulletin*. Vol. 40. No. 1. ACM. DOI: <https://doi.org/10.1145/1352135.1352274>

Casey, Kevin. (2017) Using keystroke analytics to improve pass-fail classifiers. *Journal of Learning Analytics* 4.2 189-211. DOI: <https://doi.org/10.18608/jla.2017.42.14>

Connolly, Cornelia, Eamonn Murphy, and Sarah Moore. (2008) Programming Anxiety Amongst Computing Students—A Key in the Retention Debate?. *IEEE Transactions on Education* 52.1 52-56. DOI: <https://doi.org/10.1109/te.2008.917193>

- Culligan, N., & Casey, K. (2018). Building an Authentic Novice Programming Lab Environment. Irish Conference On Engaging Pedagogy
- Giannakos, Michail N., *et al.* (2017) Understanding student retention in computer science education: The role of environment, gains, barriers and usefulness. *Education and Information Technologies* 22.5 2365-2382. DOI: <https://doi.org/10.1007/s10639-016-9538-1>
- Hembree, Ray. The nature, effects, and relief of mathematics anxiety. *Journal for research in mathematics education* (1990): 33-46. DOI: <https://doi.org/10.2307/749455>
- Kapoor, Ashish, Winslow Bursleson, and Rosalind W. Picard. (2007) Automatic prediction of frustration. *International journal of human-computer studies* 65.8 724-736. DOI: <https://doi.org/10.1016/j.ijhcs.2007.02.003>
- Jadud, M. C. (2006). An exploration of novice compilation behaviour in BlueJ (Doctoral dissertation, University of Kent). DOI: <https://doi.org/10.1080/08993400500056530>
- Kinnunen, Päivi, and Lauri Malmi. (2006) Why students drop out CS1 course?. *Proceedings of the second international workshop on Computing education research*. ACM. DOI: <https://doi.org/10.1145/1151588.1151604>
- Lister, Raymond. Concrete and other neo-Piagetian forms of reasoning in the novice programmer. *Proceedings of the Thirteenth Australasian Computing Education Conference-Volume 114*. Australian Computer Society, Inc., 2011. DOI: <https://doi.org/10.1215/9780822381525-005>
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B. D., ... & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education* (pp. 125-180). ACM. DOI: <https://doi.org/10.1145/572139.572181>
- Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1), 37-55. DOI: <https://doi.org/10.2190/gujt-jcbj-q6qu-q9pl>
- Sun, D., Paredes, P., & Canny, J. (2014, April). MouStress: detecting stress from mouse motion. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 61-70). ACM. DOI: <https://doi.org/10.1145/2556288.2557243>
- Tenenberg, Josh D., *et al.* (2005) Students Designing Software: a Multi-National, Multi-Institutional Study. *Informatics in Education* 4.1 143-162.
- Wahlström, J., *et al.* (2002) Influence of time pressure and verbal provocation on physiological and psychological reactions during work with a computer mouse. *European journal of applied physiology* 87.3 257-263. DOI: <https://doi.org/10.1007/s00421-002-0611-7>
- Wilson, B. C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: a study of twelve factors. *Acm sigcse bulletin*, 33(1), 184-188. DOI: <https://doi.org/10.1145/364447.364581>
- Yamauchi, Takashi. (2013) Mouse trajectories and state anxiety: feature selection with random forest. *2013 Humaine Association Conference on Affective Computing and Intelligent Interaction*. IEEE, 2013. DOI: <https://doi.org/10.1109/acii.2013.72>

Designing an Open Visual Workflow Environment

Charles Boisvert, Chris Roast, Elizabeth Uruchurtu

Dept. of Computing, Sheffield Hallam University
Sheffield, United Kingdom, S1 1WB

{c.boisvert | c.r.roast | e.uruchurtu}@shu.ac.uk

Abstract

This paper presents open piping, a box-and-wire programming environment, then uses Cognitive Dimensions of Notations to analyse its interaction design and identify its weaknesses. Physics of Notations gives a complementary perspective to propose solutions which we present by example. We also discuss the respective uses and benefits of Cognitive Dimensions and Physics of Notations in this work.

Keywords: Computer science education; Data Science; Functional Programming; End-User Programming; Notational Design

1. Project background and motivations

Open Piping is an open-source visual functional programming environment, based on a box-and-wire model, intended for data processing applications.

Our ambition is to propose a graphical tool for user-defined data processes¹, with the transparency and flexibility needed to ensure that users can easily define the processes they want to operate on data, while also retaining control of these processes to use them in new environments.

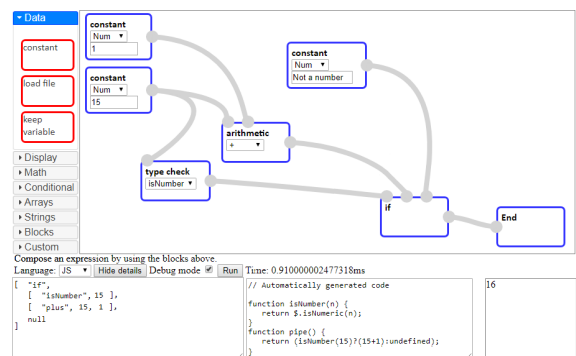


Figure 1 – Open Piping interface with an example workflow

Fig. 1 shows the Open Piping interface and an example data flow. A more complete description of the system is given in (Boisvert, Roast, & Uruchurtu, 2019).

Four elements motivate our work: the systematic improvement in access to programming brought by the growing ease of use and learning of programming tools; the rise of data processing, underpinned by functional programming and the growth of big data and data analytics; the possibility of modelling functional computation visually; and finally the access barriers to this visual programming paradigm.

A systematic improvement in access to programming. Usability breakthroughs mark the progress of all computer science, including programming. One remarkable advance is the wide range of programming learning and novice developer environments, such as MIT Scratch, using a jigsaw puzzle metaphor to represent the combination of individual statements (Resnick et al., 2009).

The rise of data processing. As simple applications have become more accessible, computation has shifted to new domains, and to programming languages that support multiple paradigms, like R, Clo-

¹<http://boisvert.me.uk/openpiping>

jure, or Python which add functional programming to imperative, object-oriented and event based development. Yet, the jigsaw puzzle metaphor favours an imperative perspective on programming: the programming paradigms computing education tools support best, are becoming less used in practice.

Modelling functional computation visually. Lambda calculus' mapping to directed acyclic graphs provides a visual model, summarised table 1. The graph, or box-and-wire model, can read as a data flow.




Notation	Represents	Graphical equivalent
x	Variable	
$\lambda x.f$	Abstraction (function f has parameter x)	
fx	Application (function f is applied to variable x)	

Table 1 – Basic elements of untyped λ -calculus and their representation as box-and-wire

Access limitations to visual functional programming Visual box-and-wire environments are common (Le-Phuoc, Polleres, Tummarello, & Morbidoni, 2008; O'Reilly, n.d.), including some in commercial (Instruments, accessed 2019) and scientific (Hull et al., 2006) use. But in many cases, the value of the tools is limited by a lack of open, accessible implementations of the processes they define and intermediate technologies. Take the case of the popular Yahoo pipes (O'Reilly, n.d.): when support ended in 2015, users only option was a complex export process.

However, data analysis applications require mastery of complex systems to apply mathematical techniques and represent information in non-trivial domains, and this needs to be supported by design.

2. Improving a Data Flow Visualisation

As prior research (Roast, Leitão, & Gunning, 2016; Blackwell, 2006, 2001) shows, visualisation is not easy to represent in ways that end-users spontaneously understand. Users', particularly novices, need carefully designed presentation and interaction devices.

2.1. Cognitive Dimensions of Notations

Cognitive Dimensions of Notations is a framework of design heuristics (Green & Petre, 1996). The common language it provides is frequently used to evaluate the usability of programming languages and interfaces (Hadhrawi, Blackwell, & Church, 2017; Ennals & Gay, 2007; Morbidoni, Polleres, Tummarello, & Le Phuoc, 2007).

Evaluating the notations used in a complex information artefact, such as Open Piping, with this framework requires a lot of judgement. As an example, let us can compare two evaluations of tools developed on principles comparable to Open Piping.

(Morbidoni et al., 2007) propose Semantic Web Pipes, a functional language and pipe editor to prototype semantic mash-ups; while (Green & Petre, 1996), introducing the framework in their analysis of visual programming environments, consider two functional environments, Labview and in Prograph.

Evaluating the viscosity of their tool, the first estimate that the underlying functional paradigm guarantees 'almost literally the principle of encapsulation and decoupling'. For the same dimension, Green and Petre instead test the viscosity by attempting a minor change to test code in each of the two functional environments, Labview and in Prograph. They choose to focus, not on the language, but on manipulating the code at the interface, explaining that 'boxes had to be jiggled about', and are not satisfied that the language supports appropriate abstractions.

The list of dimensions has varied a little since the framework was proposed. Here, we use the dimensions suggested by (Green & Petre, 1996), listed table 2 (next page).

Cognitive Dimensions provide a useful vocabulary to evaluate the usability of Open Piping, and supports identifying its limits more precisely. Below, we propose an evaluation of Open Piping against each cognitive dimensions.

Dimension	Characteristic
Abstraction gradient	What are the minimum and maximum levels of abstraction exposed by the notation? Can details be encapsulated?
Closeness of Mapping	Does the notation correspond to the problem world?
Consistency	When some of the notation has been learnt, how much of the rest can be inferred?
Diffuseness / terseness	How many symbols (how much space) the notation requires to produce a certain result
Error-proneness	Does the notation induce user mistakes?
Hard mental operations	How much do the notations impose hard mental processing?
Hidden dependencies	Are dependencies visible or hidden?
Juxtaposability	Is every part of the notation visible at the same time?
Premature commitment	Are there strong constraints on the order in which the user must complete the tasks to use the system? Are there decisions that must be made before all the necessary information is available? Can those decisions be reversed or corrected later?
Progressive evaluation	How easy is it to evaluate and obtain feedback on an incomplete solution?
Role-expressiveness	How obvious is the role of each component of the notation in the solution as a whole?
Secondary notation	Can the notation carry extra information by means not related to syntax (e.g. layout, colour, or other cues?)
Viscosity	How much effort is required to make a single change?
Visibility	Can required parts of the notation be identified, accessed and made visible?

Table 2 – Cognitive Dimensions, after (Green & Petre, 1996).

Abstraction gradient Open Piping supports the re-use of code by creating new blocks, and of data by setting variables. The management of these abstractions becomes difficult if the user doesn't anticipate: that is, re-usable code identified late in a project needs to be redefined to set it as an (abstraction supporting) block.

Closeness of Mapping and Consistency Consistency and Closeness of Mapping are the system's strongest point, as the functional model is represented faithfully by the visual objects.

Diffuseness / terseness A lot of blocks can be necessary to specify even simple computations, as each function call and each operator is one block.

Error-proneness End-users can easily drag the wrong block, or the wrong link from output to input; though these mistakes are easily undone.

Hard mental operations Constructs which use expressions as input (such as lambda expressions) are very difficult to compute mentally. The number of blocks also create visual clutter and make mental operations harder.

Hidden dependencies Blocks represent functions and operations: end-users need to be familiar with their effect, including that of complex operations (e.g. lambda extraction).

Juxtaposability and Role-expressiveness How much of the computation is visible at the same time depends on its complexity: how many blocks are in use and whether it uses any user-defined blocks. Blocks are clearly annotated with the function they represent; the 'wires' relating them are more easily confused as they are not marked with information. Wires can also cross.

Premature commitment User-defined blocks are created within a separate window, and so the end-user

needs to plan ahead that their computation belongs in a new block.

Progressive evaluation Solutions can easily be tested throughout the development process.

Secondary notation User-defined blocks can be named by the user. Spatial positioning of blocks has no incidence on the result and is also chosen by the end-user, although the blocks are presented with inputs at the top-left and outputs on the right-hand side, so blocks' disposition is intended for a top to bottom, left to right reading order.

Viscosity Minor changes (e.g. adding a box) require a lot of adjustment; user blocks are also difficult to redefine, as discussed above in premature commitment and abstraction gradient.

Visibility The notations are clear to end users, but the visualisation relies on a limited range of three objects: boxes, lines, and discs marking input or output.

Cognitive Dimensions support a useful discussion to identify of the tools' weaknesses. Following it we propose to reduce the abstraction gradient and increase viscosity by allowing a user manipulation to select a subtree in an expression, and make it into a custom block. But it is not always as clear how to address the points identified through Cognitive Dimensions. To that end, we propose to turn to another approach: Physics of Notations.

2.2. Physics of Notations

Physics of notations proposes a theory of design for notations based on maximising cognitive effectiveness (Moody, 2009; Van Der Linden & Hadar, 2018). To that aim, it defines nine principles to analyse and develop notations. Compared to Cognitive Dimensions, Physics of Notations ambitions to be a more complete theory, which offers to explain why notations succeed, as well as simply describe them.

More prosaically, for this work it offers two clear advantage over cognitive notations: its principles are focused on visual notations, and all of them offer actionable points to improve the effectiveness of notations. Using these principles, we propose an alternative, fig. 2 which addresses many of the weaknesses of the design identified 2.1 (next page).









Type	Symbol and colour	Note
Boolean		Chosen to match the colour and shape of Scratch booleans
Number		Match the colour and shape of Scratch numbers
String		A large quotation mark
List		Square brackets, as in JSON and arrays in many programming languages
JSON data		Curly brackets, as in JSON objects
Expression		Expression, formed of related blocks, are needed as inputs to some blocks, for instance lambda-extraction or conditionals
Block		To allow blocks as output of and input to other blocks (in programming terms, functions as data)
Any		Used when any data type is allowable as input, for example, in a type checking block

Table 3 – Data types used by open piping and their notational representation

Semiotic clarity recommends that all semantic constructs find a visual expression. Boxes and wires represent input and output to functions, but data typing is an important semantic construct that should also be made clear: identical wires give no information about the data transmitted. By associating each main data type to a colour (for wires) and a symbol (for inputs and outputs), we express more of the

important semantic information within the notation. Each data type used, and the proposed colour and shape to denote them is presented for reference in table 3 above.

Perceptual discriminability (ensuring symbols are easy to tell apart) can be seen in table 3, where a small number of colour and symbols are used, making them highly distinguishable. Blocks have an identical shape, except for blocks processing expressions as discussed below, but input blocks and blocks carrying out operations can be distinguished by colour.

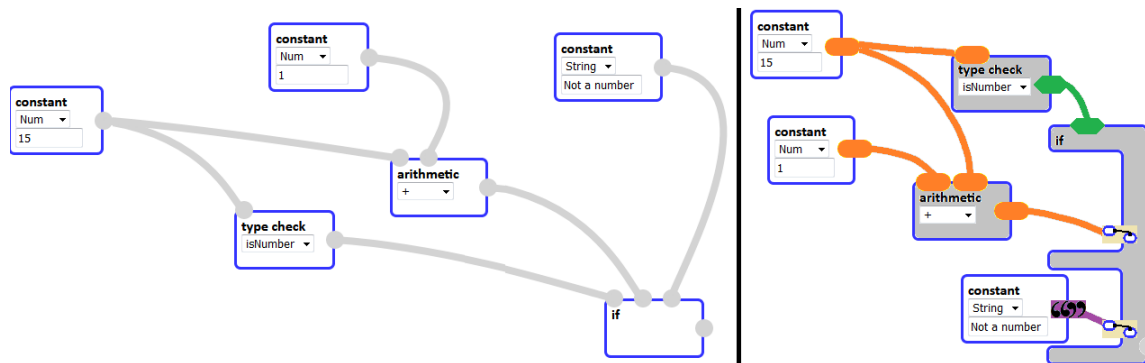


Figure 2 – A workflow before (left) and after (right) revision for cognitive effectiveness

Visual expressiveness (using the fullest range of visual variables possible) point to fully exploiting visual variables - for example, as above, with colour and shape. This can also be done by giving the blocks that receive expressions as input, a special shape, to indicate their exceptional character - an "open jaw" shape that can also visually include some of the expression.

Semantic transparency recommends that the appearance of notations suggests their meaning: this is done by composing function boxes with icons for input and output parameters, and by positioning inputs at the top or left and outputs at the bottom or right, following the common reading order.

Graphic economy is also respected as symbols remain few and easily recognised. We also propose to allow the output symbol to glide freely around the right and bottom side of a block, and inputs around its left and top sides - including not imposing an argument order, so that wires rarely cross over in complex expressions, reducing visual clutter.

Complexity management is supported by encoding more information in pre-attentive ways - through colour, size and shape. Allowing the argument inputs and output to "float" along the box's border also reduces complexity. The shape of blocks handling expressions as input may not reduce complexity, but it signals it to the viewer.

Fig. 2 applies the principles of Physics of Notations to one workflow. The final three principles of Physics of Notations are *Cognitive Integration*, *Dual coding*, and *Cognitive fit*, which address, respectively, the coordination of documents, the use of written annotations, and the adaptation to diverse audiences. Our proposed solution fig. 2 does not make use of these points.

3. Future work and reflection

A further evaluation will be needed to consider the effectiveness of the redesign proposed in section 2.2 above. But the two frameworks showed an interesting complementarity when using them simultaneously.

Cognitive Dimensions and Physics of Notations complemented each other usefully to carry out this analysis and find design options. The Cognitive Dimensions heuristics provided insights in dynamic aspects of the visualisation with its dimensions of abstraction gradient and viscosity, pointing immediately to both problem and solution. It then helped highlight many weaknesses of the design, but provided fewer actionable points to improve it. By contrast Physics of Notations focused attention on identifiable improvements to the visualisation. In particular, as (Roast & Uruchurtu, 2016) point out, Physics of No-

tations centrally asks the question: 'what constitutes and defines the semantic domain being visualised?' (Roast & Uruchurtu, 2016). This semantic focus brought the insight that the visualised domain must include typing, while other principles provided means to do so.

Cognitive dimensions also provides a validation of the redesign suggestions. The proposed redesign goes some way to solving several of the most egregious problems identified with Cognitive Dimensions: it is less error-prone, more tolerant of change, and gives important clues in support of hard mental operations. This shows it is valuable to exploit both frameworks in combination: Cognitive Dimensions helps consider notation abstractly and include dynamic aspects of use, while Physics of Notations focus on visual ways of expressing meaning and on identifying that meaning to be expressed.

4. References

- Blackwell, A. F. (2001). Pictorial representation and metaphor in visual language design. *Journal of Visual Languages & Computing*, 12(3), 223 - 252. Retrieved from <http://www.sciencedirect.com/science/article/pii/S1045926X01902071> doi: <https://doi.org/10.1006/jvlc.2001.0207>
- Blackwell, A. F. (2006, December). The reification of metaphor as a design tool. *ACM Trans. Comput.-Hum. Interact.*, 13(4), 490–530. Retrieved from <http://doi.acm.org/10.1145/1188816.1188820> doi: 10.1145/1188816.1188820
- Boisvert, C., Roast, C., & Uruchurtu, E. (2019). Open piping: Towards an open visual workflow environment. In *Conference proceedings of 2019 international symposium on end-user development (is-eud)*.
- Ennals, R., & Gay, D. (2007). User-friendly functional programming for web mashups. In *Acm sigplan notices* (Vol. 42, pp. 223–234).
- Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2), 131–174.
- Hadhrawi, M., Blackwell, A. F., & Church, L. (2017). A systematic literature review of cognitive dimensions. In *Ppig* (p. 3).
- Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M. R., Li, P., & Oinn, T. (2006). Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(suppl 2), W729–W732.
- Instruments, N. (accessed 2019). *What is labview*. <http://www.ni.com/en-gb/shop/labview.html>. (Accessed: 2019-30-04)
- Le-Phuoc, D., Polleres, A., Tummarello, G., & Morbidoni, C. (2008). Deri pipes: visual tool for wiring web data sources. *(Eds.): 'Book DERI pipes: visual tool for wiring web data sources' (2008, edn.)*.
- Moody, D. (2009). The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering*, 35(6), 756–779.
- Morbidoni, C., Polleres, A., Tummarello, G., & Le Phuoc, D. (2007). Semantic web pipes. *Rapport technique, DERI*, 71, 108–112.
- O'Reilly, T. (n.d.). *Pipes and filters for the internet*. <http://radar.oreilly.com/2007/02/pipes-and-filters-for-the-inte.html>. (Accessed: 2016-10-10)
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... others (2009). Scratch: programming for all. *Communications of the ACM*, 52(11), 60–67.
- Roast, C., Leitão, R., & Gunning, M. (2016). Visualising formula structures to support exploratory modelling. In *Proceedings of the 8th international conference on computer supported education* (pp. 383–390). Portugal: SCITEPRESS - Science and Technology Publications, Lda. Retrieved from <https://doi.org/10.5220/0005812303830390> doi: 10.5220/0005812303830390
- Roast, C., & Uruchurtu, E. (2016). Reflecting on the physics of notations applied to a visualisation case study. In *Proceedings of the 6th mexican conference on human-computer interaction* (pp. 24–31).
- Van Der Linden, D., & Hadar, I. (2018). A systematic literature review of applications of the physics of notations. *IEEE Transactions on Software Engineering*, 45(8), 736–759.

Assessing a candidate's natural disposition for a software development role using MBTI

Daniel Varona
CulturePlex Laboratory
Western University
dvaronac@uwo.ca

Luiz Fernando Capretz
Faculty of Electrical and Software Engineering
Western University
lcapretz@uwo.ca

Abstract

Over the past decade, there has been a marked interest in understanding the personal traits of software developers and their influence on the process of assigning people to roles, as has been evident from the growing number of related publications on this topic. This study is part of a larger research project focussed on identifying the elements associated with the candidate's personal traits and how these traits better fit with particular software development roles. Our goal in this study is to complement the current approach to assigning roles, which is based on an individual's capacity to fulfill a role's functional competencies profile during the assignment process. Our approach helps to support the assigning of people to software development roles by providing a set of tools, based on Myers-Briggs type indicators, to assess a candidate's natural disposition. To do this, we modeled the results obtained in a previous study on software developer preferences for tasks associated with software industry roles. As a result, we obtained a set of rules to be considered at the time of assignment— relationship values between MBTI type indicators based on preferences— and then mathematically formalized a coefficient to evaluate the natural disposition of candidates during the allocation process.

Keywords: Software Project Staffing, Role Natural Disposition Assessment in Software Industry, MBTI, Assigning People to Role in Software Projects, MBTI and Software Projects

Introduction

There is no doubt that the software industry is key to all spheres of society: economic, social, political and infrastructural. Due to its wide range of applications, it is difficult to conceive of an industrial branch without a software component. Behind that technical coverage, there is a project team, the group of people who make it possible. Like any other industry, software needs raw materials.

In the case of software development, the raw material is expressed as technical and non-technical competencies, and as the communication skills required from developers, among others, that contribute to enhancing the synergy of the team. Considering that software projects tend to be planned in shorter periods of time each time, the synergy of the team gains a significantly greater value. In terms of the contribution of each member to other members of the

team, the software industry is an ecosystem where the success of the projects depends chiefly on proper role assignment and adequate project team staffing from the beginning of the project.

Over the past decade, academics have become more interested in the relationship between natural disposition and the suitability of software development roles and have begun researching new approaches that can complement the current process of assigning roles in software projects. These approaches include studies on personality traits (Varona, Capretz, Piñero, & Raza, 2012) (Varona, Capretz, & Raza, A multicultural comparison of software engineers, 2013) (Capretz, Waychal, Jia, Varona, & Lizama, 2019) (Lizama, Varona, Warchal, & Capretz, 2020), and team staffing (Mazni, y otros, 2016) (Aritzeta, Swailes, & Senior, 2007) based on team roles, to name just two examples. The present research continues work done in a previous study (Capretz, Varona, & Raza, Influence of personality types in software tasks choices, 2015) conducted by the authors, which sought to describe the distribution of MBTI indicators among software developers and to identify their preferences for certain software related tasks that were historically linked to roles such as Project Leader, Analyst, Designer, Programmer, Tester, and Maintenance related roles.

While it is true that these more defined roles are closely tied to heavy software development methodologies, and that most projects are currently implemented following flexible development methodologies, we must point out that in such cases roles do not dissolve but are mixed. Ones acquiring responsibilities subscribed to other, and therefore, the needs that the role demands of the individual who performs it remains untouched. One may acquire responsibilities that have been assigned to another person, but the demands on the individual of the originally assigned role remain unchanged. Therefore, the need for a proper human resources allocation from the very beginning becomes even more important.

This study aims to identify a set of patterns based on software practitioners' preferences for certain software tasks and the distribution of their MBTI-type indicators gathered in (Capretz, Varona, & Raza, Influence of personality types in software tasks choices, 2015), that can be used to model a natural disposition coefficient towards the role a candidate is given. And together, both tools support the decision-maker while role assignment subprocess in the process of human resources acquisition for software projects.

Method

Firstly, we converted the software practitioners' preferences exhibited in (Capretz, Varona, & Raza, Influence of personality types in software tasks choices, 2015) into "If - then" rules that we then processed using R. This resulted in 1500 rules that we summarized using R features for

rules summarization. With the resulting set of rules, it was possible to identify a set of patterns which are related in the Results and Discussion section.

Next, and also based on the software practitioners' choices of software tasks and their preference priorities over the studied roles, we determined the correlation values between the MBTI type indicators and the studied software roles. Table 1 in the Results and Discussion section shows the correlation values.

Lastly, taking into account the identified correlation values between MBTI type indicators and software development roles, we proceeded to mathematically formalize a coefficient to evaluate the natural disposition in role candidates.

Results and Discussion

Following the same order presented in the Methods section, we then proceeded to present the patterns we had identified. Our goal is for these patterns help the decision-maker at the time of assigning people to software development roles. For better organization we grouped the patterns by roles, as can be seen below.

- To better assess a candidate's natural disposition for the Project Leader role, the candidate should meet the following criteria that characterize current successful software practitioners performing as Project Leader:
 - There must be a predisposition towards ST or NT mental functions
 - There must be a predisposition towards EJ attitude functions
- To better assess a candidate's natural disposition for the Analyst role, the candidate should meet the following criteria that characterize current successful software practitioners performing as Analysts:
 - There must be a predisposition towards extroversion E_ _ _.
 - There must be a predisposition towards an extroverted judging attitude function _ _ _ J if it is met that IS_ _.
 - There must be a predisposition towards an extroverted perceiving attitude function _ _ _ P if it is met that IN_ _.
 - There must be a predisposition towards the following mental functions¹: NT, ST, and SF; no prioritization needed between them.
 - The following type indicators must be prioritized in this exact order: ESTJ, ESTP, ISTJ, ISFJ, INTJ, ESFP, and INTP.

¹ The attitude pair analysis is omitted here as it seems to tend to the IJ and that might be perceived as a contradiction to the above stated conditions whereas is actually it harnesses the mental pairs and allow this extroverted energizing attitude.

- To better assess a candidate's natural disposition for the Designer role, the candidate should meet the following criteria that characterize current successful software practitioners performing as Designers:
 - There must be a predisposition towards an extroverted perceiving attitude function __ __ P if it is met that I _ T _.
 - There must be a predisposition towards an extroverted judging attitude function __ __ J if it is met that IS__.
 - There must be a predisposition towards the following mental functions: NT, ST, and SF; no prioritization needed between them.
 - The following type indicators must be prioritized in this exact order: INTP, INTJ, ISTJ, ISTP, ESTJ, ENTJ, ESTP, and ESFP.
- To better assess a candidate's natural disposition for the Programmer role, the candidate should meet the following criteria that characterize current successful software practitioners performing as Programmers:
 - There must be a predisposition towards extroversion E__.
 - There must be a predisposition towards the following mental functions: ST, and SF; no prioritization needed between them.
 - There must be a predisposition towards the following attitude functions: IJ, and EJ; no prioritization needed between them.
 - The following type indicators must be prioritized in this exact order: ESTJ, ESTP, ISFJ, ENTJ, ESFP, and ISTP.
- To better assess a candidate's natural disposition for the Tester role, the candidate should meet the following criteria that characterize current successful software practitioners performing as Testers:
 - The ISTJ type indicator must be prioritized.
- To better assess a candidate's natural disposition for the Maintainer role, the candidate should meet the following criteria that characterize current successful software practitioners performing as Maintainers:
 - There must be a predisposition towards an extroverted perceiving attitude function __ __ P if it is met that ES__.
 - There must be a predisposition towards the ST mental function.
 - There must be a predisposition towards the following attitude functions: EP, IJ, and EJ; no prioritization needed between them.
 - The following type indicators must be prioritized in this exact order: ISTP, ISFJ, ENTJ, ESTJ, ISTJ, ESFP, and ESTP.

Considering the patterns outlined above, we proceeded to relate the type indicators to the functional roles under investigation, as can be seen in Table 1. The blank cells indicate that the corresponding type indicator and functional role are not related. In contrast, the cells that denote a relationship between the corresponding type indicators and functional roles exhibit a value expressing the type indicator's natural disposition coefficient for the functional role.

To find the value that expresses the natural disposition of a type indicator with respect to a role, it was necessary to formalize a coefficient to that effect, which we call the natural disposition (*ND*) coefficient. Equation I show the *ND* coefficient modeling:

$$\text{Equation (I): } ND = \frac{PT}{TP}$$

Where *PT* is the number of patterns for a given role a type indicator satisfies between all identified patterns for that role, and *TP* is the total number of identified patterns for the analyzed role.

MBTI Type Indicators	Functional Roles					
	Project Leader	Analyst	Designer	Programmer	Tester	Maintainer
ISTJ	0.5	0.6	0.75	0.5	1	0.75
ISFJ		0.6	0.5	0.75		0.5
INFJ				0.25		0.25
INTJ	0.5	0.4	0.5	0.25		0.25
ISTP		0.2	0.75	0.5		0.5
ISFP		0.2	0.25	0.25		
INFP		0.2				
INTP		0.6	0.75			
ESTP	0.5	0.6	0.5	0.75		1
ESFP		0.6	0.5	0.75		0.75
ENFP		0.2		0.25		0.25
ENTP	0.5	0.4	0.25	0.25		0.25
ESTJ	0.5	0.6	0.5	1		0.75
ESFJ	0.5	0.4	0.25	0.75		0.25
ENFJ	0.5	0.2		0.5		0.25
ENTJ	1	0.4	0.5	0.75		0.25

Table 1 Relation between the MBTI type indicators and the studied functional software roles

The *ND* coefficient takes values between 0 and 1, resulting in the natural disposition of the given MBTI indicator for a given role. Consequently, the possibilities for assigning an individual with a certain MBTI type indicator to a given role can be easily sorted and related to the roles for which the type indicator is a better match. Table 1 itself represents a decision matrix for each type indicator and their associated roles given its natural disposition coefficient

for each of them. The decision-maker can use the matrix as a tool for reference when assigning people to roles.

Conclusions

In the present study we identified a set of patterns from the MBTI type indicators of currently successful software practitioners performing in each of the roles studied, as well as their preferences towards software task choices, also linked to the competence profile of the roles studied.

Based on the patterns identified, it was possible to formalize a coefficient to assess a candidate's natural disposition in relation to a given software project role. This was done by considering that candidate's MBTI type indicator, expressed in terms of the relation between the number of patterns the candidate's MBTI type indicator satisfied from the total number of identified patterns for the target role.

We defined a decision matrix connecting the MBTI type indicators with the target roles using the values assumed by the natural disposition coefficient, which can be used by the decision-maker as a support tool when assigning people to roles in software projects.

The new approach described in this study complements those currently available within the specialized literature focusing on the assignment of people to roles in software development. It also enriches the methodological framework around the assignment as an object of study in the software engineering field of study.

References

- Aritzeta, A., Swailes, S., & Senior, B. (2007). Belbin's Team Role Model: Development, Validity and Applications for Team Building. *Journal of management Studies*, 44(1), 96-118.
- Capretz, L. F., Varona, D., & Raza, A. (2015). Influence of personality types in software tasks choices. *Computers in Human Behavior*(52), 373-378.
- Capretz, L. F., Waychal, P., Jia, J., Varona, D., & Lizama, Y. (2019). Studies on the software testing profession. *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.
- Lizama, Y., Varona, D., Warchal, P., & Capretz, L. F. (2020). Unpopularity of the Software Tester Role among Software Practitioners: A Case Study. En *Advances in RAMS Engineering. In Honor of Professor Ajit Kumar Verma on His 60th Birthday* (pág. 465). Springer International Publishing.
- Mazni, O., Bikhtiyar, H., Mazida, A., Azma, Y., Fauziah, B., Haslina, M., & Norida, M. D. (2016). Applying fuzzy technique in software team formation based on Belbin team role. *Journal of Telecommunication, Electronic and Computer Engineering*, 8(8), 109-113.

- Varona, D., Capretz, L. F., & Raza, A. (2013). A multicultural comparison of software engineers. *World Transactions on Engineering and Technology Education*, 11(1), 31-35.
- Varona, D., Capretz, L. F., Piñero, Y., & Raza, A. (2012). Evolution of software engineers' personality profile. *ACM SIGSOFT Software Engineering Notes*, 37(1), 1-5.

On personality testing and software engineering

Clayton Lewis
Coleman Institute for Cognitive Disabilities
University of Colorado
clayton.lewis@colorado.edu

There is current interest in applying methods from the psychology of personality to software engineering. For example, in a recent review, Graziotin et al. (2020) say,

“Software engineers are knowledge workers and have knowledge as main capital [98]. They need to construct, retrieve, model, aggregate, and present knowledge in all their analytic and creative daily activities [89]. Operations related to knowledge are cognitive in nature, and cognition is influenced by characteristics of human behavior, including *personality, affect, and motivation* [47]. It is no wonder that industry and academia have explored psychological aspects of software development and the assessment of *psychological constructs at the individual, team, and organization level* [30, 61]. [emphasis added.]

They are quite critical of much current work, including that using the discredited Myers-Briggs Type Indicator (MBTI, which remains popular in the literature), and other work that attempts to adapt what the authors feel are well validated methods from other areas of social science in ways that are scientifically unsound. The authors then present what they see as the methods needed for sound work in the area, emphasizing that doing the work correctly requires an enormous amount of work. They conclude,

“The software engineering community must value psychometric studies more. This, however, requires a cultural change that we hope to champion with this paper. ‘Spending an entire Ph.D. candidacy on the validation of one single measurement of a construct should be, not only approved, but encouraged.’ [43] and, we believe, should also become normal.”

We are skeptical that the psychometric methods the authors advocate can usefully be applied in software engineering. Further, we see the potential for substantial harm from efforts to apply them.

As an illustration of the issues, we’ll use the hypothetical psychometric study included as a methodological appendix to Graziotin, et al. (2020). The study aims to illuminate “individual perception styles of source code”:

“Our fictitious construct is the ‘individual perception styles of source code’ that, through a literature review (or, perhaps, after a grounded theory study), we believe is mainly composed of, or highly related, to the following five constructs: code curiosity, programming paradigm flexibility, learning disposition, collaboration propensity, and comfort in novelty.”

Some premises of such an investigation are that there is a population of source code perceivers whose attributes can be studied, and that these individuals have characteristics like “code curiosity” or “learning disposition” that can usefully be measured. The nature of psychometric methods requires that investigators have access to many members of the population to be studied.

The nature of software engineering is that it is complex in a great many different ways, that all influence “source code perception”. What tools are being used to examine source code? What language is the code in (how does perception of Python code relate to perception of an Excel macro)? Is the code being viewed in the context of coding, or debugging, or testing, or maintenance? Is the perceiver already familiar with the code base, or are they a new team member? Are they experienced or inexperienced in the particular task being examined? Are they working individually, or in a group code review? We think it highly unlikely that enough data could be gathered in any one of the situations defined by the cross product of these distinctions to support a proper psychometric analysis. That is to say, undertaking a psychometric analysis of this construct really means that one thinks that “source code perception” isn’t actually influenced by these things in important ways.

Does this argument mean that all empirical study of phenomena as complex as software engineering is futile? We think it does mean that empirical study that is *purely* empirical cannot succeed. That is, research has to be guided by ideas about the mechanisms at work in the phenomenon, mechanisms that can be identified in action across varying situations. Cognitive dimensions analysis is an example of a framework that can offer that (see Lewis, 2017, for related discussion).

We have concerns about the constructs proposed as “related” to source code perception, as well. To undertake this psychometric investigation means that one believes that things like “programming paradigm flexibility” are stable characteristics of people. But given the complexity of software engineering situations, and of people, we see no reason to expect that. Rather, we expect that people’s behavior, and preferences, will be influenced by the tools they use, their past experience, their current social environment, their personal goals, and much more. There just isn’t such a thing as “programming paradigm flexibility” independent of these contextual factors.

We understand that the constructs in this example from Graziotin et al. are made up, so it is pointless to argue about the specifics of them. But we agree with Graziotin et al. that they illustrate the logic of many studies that might actually be undertaken, or have been undertaken.

An advocate of psychometric research might respond, “Well, if you are right, our studies will fail, but if you are wrong, our studies will reveal that there really are these stable characteristics of software engineers, with consequences that cut across the distinctions among situations that you are concerned about. So you should let us get on with the work, and we’ll see who is right.”

Of course, people are free to investigate whatever they like, and we can’t forbid them, even if we wanted to. But we can register our doubts that the large investments needed will prove justified. In particular, we strongly dissent from the suggestion that “Spending an entire Ph.D. candidacy on the validation of one single measurement of a construct should be, not only approved, but encouraged.” We would certainly not encourage our students to do such a thing.

Our concerns go farther. The persistence of the discredited MBTI in the literature suggests that psychometric ideas are *seductive*. People often want to believe that there are simple, knowable characteristics of people that will allow us to deal more straightforwardly and objectively with our fellow creatures. Rather than embracing human diversity, we often want to reduce people to numbers: we should hire this candidate rather than this one, because they scored better on this assessment, or we should assign this person this role rather than that one, because they show higher “programming paradigm flexibility”.

That seems easier than trying to assess their job experience. The appeal of this vision is such that it often overrides methodological scruples, as Graziotin et al. document. The research community needs to push back.

We think that some potential applications of psychometrics to software engineering are actually *harmful*, as well as wrong, adding urgency to the push back. The belief that software engineers have stable psychometric characteristics that we can measure, and that predict their aptitude for particular roles, leads naturally to making decisions about hiring, or promotions, on the basis of these measurements. Indeed, Graziotin et al. envision just these applications, and the potential for harm accompanying them:

“Improper development, administration, and handling of psychological tests could harm the company by hiring a non-desirable person, and it could harm the interviewee because of missed opportunities.”

For Graziotin et al., these potential harms call for *proper* “development, administration, and handling of psychological tests.” But we argue that *there can be no such thing* as proper development and application of such tests, because of the complexity of people, software engineering situations, and their combinations.

Acknowledgement. I thank Luke Church for useful comments.

References

- Graziotin, D., Lenberg, P., Feldt, R., & Wagner, S. (2020). Behavioral Software Engineering: Methodological Introduction to Psychometrics. arXiv preprint arXiv:2005.09959.
- Lewis, C. (2017). Methods in user oriented design of programming languages. In Proc. PPIG 2017- 28th Annual Workshop. Online at <https://ppig.org/papers/> .

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

Leah Bidlake **Eric Aubanel** **Daniel Voyer**
Faculty of Computer Science, Faculty of Computer Science, Department of Psychology
University of New Brunswick
leah.bidlake@unb.ca, aubanel@unb.ca, voyer@unb.ca

Abstract

Research on mental representations formed by programmers during program comprehension has not yet been applied to parallel programming. The goal of this proposed study is to validate the stimuli that will be used in subsequent studies on mental representations formed by expert parallel programmers. The task used to stimulate the comprehension process will be verifying the correctness of parallel programs by determining the presence of data races. Responses to the data race question will be analysed to determine the validity of the stimuli. Participants will also be asked what components of the program they used to determine whether or not there was a data race and their responses will be collected for use in future work.

1. Introduction

In recent years, the research on program comprehension has declined dramatically and as a result, newly developed or popularized languages and paradigms including parallel programming have not been a part of the research (Bidlake, Aubanel, & Voyer, 2020). Parallel programming has introduced new challenges including bugs that are hard to detect, making it difficult for programmers to verify correctness of code. One type of bug that occurs in parallel programming is data races. Data races occur when multiple threads of execution access the same memory location without controlling the order of the accesses and at least one of the memory accesses is a write (Liao, Lin, Asplund, Schordan, & Karlin, 2017). Depending on the order of the accesses some threads may read the memory location before the write and others may read the memory location after the write which can lead to unpredictable results and incorrect program execution. Data races are difficult to detect and verify as they will not appear every time that the program is executed. To detect data races programmers must understand how a program executes in parallel on the machine and the memory model of the programming language.

2. Research Goal

To date, no empirical research on program comprehension or mental representations of parallel programmers has been conducted. The lack of research in this programming paradigm means that there are no existing data sets or stimuli to draw from. We will create a stimulus set to be used in subsequent research on mental representations formed by parallel programmers. The research goal of the proposed study is to validate the stimulus set.

3. Method

Given the feedback received when this proposal was presented at the PPIG Doctoral Consortium 2020, we will be conducting a pilot study with 10 participants. The results of the pilot will be analysed to determine if any of the parameters need to be adjusted for the validation study.

3.1. Participants

Participants will need to have experience programming in C and using OpenMP 4.0 directives to implement parallelization. One hundred participants will be recruited using social media including Facebook and LinkedIn and email. Participants will receive a \$10 gift card as an incentive; this will be administered using Rybbon. Participants will be informed in the consent form that the incentive is only available in select countries. Participation will be voluntary and the protocol will be approved by the research ethics board at UNB.

3.2. Materials

The programs from the DataRaceBench 1.2.0 benchmark suite (Liao et al., 2017) were used as inspiration for the programs written by the first two authors, who are expert programmers. The programs will be written in C using OpenMP 4.0 directives with no comments or documentation. There will be 80 programs in total, all containing a parallel region. In our original study proposal the programs were divided into data race and no data race categories and then subdivided into reading input from a file and reading input from the command line. The feedback received from the doctoral consortium led us to simplify the programs so that all data was contained within the program. Forty of the programs will contain a data race and forty of the programs will not contain a data race. This will produce 4000 data points per data race (40 programs x 100 participants), exceeding the power recommendation proposed by Brysbaert and Stevens (2018) for our linear mixed model analysis.

The length of the programs will be measured by counting the lines of code. The mean length of the programs with data races will be the same as the mean length of the programs without data races.

3.3. Procedure

Participants will complete the tasks online. The experiment will be developed using PsychoPy 3, an open source software package, and Pavlovia will be used to host the experiment online. Qualtrics will be used to develop and administer the consent form at the beginning of the experiment and the questionnaire at the end of the experiment.

The stimuli will be presented to each participant in a random order. Participants will be given a time limit of 30 seconds for exposure to the stimuli. The exposure will end when the time limit has been reached or when the participant responds to the data race question. There were concerns raised at the doctoral consortium regarding the exposure time and mental strain of tracing code. To address these concerns we have ensured that variable names used in the stimuli match typical programming conventions (i.e.: variables *i*, *j*, and *k* are used for loop counters) and were consistent between stimuli to reduce the mental load (i.e.: variables used for arrays in all stimuli are *a*, *b*, and *c*). The stimuli were also simplified by removing the code to read in data from either a file or the command line.

The following measures will be taken for each stimulus: correctness of response, response time, and level of confidence in their response. Their level of confidence will be measured using a visual analogue scale representing a 100mm line with "not confident" as the left side extreme (score of 0) and "very confident" as the right side extreme (score of 100). Participants will click at the location of their answer and the program will record the distance along the line (0 to 100) as the measure of confidence.

Originally we proposed to ask a summary question for a subset of the stimuli. It was suggested at the doctoral consortium that a summary may not provide insight into the mental model of the participants. Instead, questions that specifically elicit how participants are thinking about the code, what parts of the code they are looking at, or what parts of the code are most relevant for the task, would provide more information that relates to their mental representations. In response to this we decided to ask the question "What cues or program components did you use to determine whether or not there was a data race?" instead of writing a summary. Thirty of the stimuli will include the additional task of answering this question. The participants will not be given a time limit for writing their answer. The question will take place after the data race question and the level of confidence rating are completed. After finishing the data race experiment, participants will complete a questionnaire to document their background and level of expertise (Feigenspan, Kastner, Liebig, Apel, & Hanenberg, 2012). Specifically, they will be asked about their: age, gender, year of study, level of education completed, years of programming experience, number of programming courses completed, self estimated level of programming expertise and parallel programming expertise, perceived level of programming expertise compared to their peers, and perceived level of parallel programming expertise compared to their peers.

3.4. Analysis

The results of the pilot study will be used to determine if any parameters of the study need to be adjusted. The exposure time and level of difficulty of the stimuli may need to be adjusted if the accuracy rate is

low and participants are using all of the allotted time exposure.

The participants' responses to the question for select stimuli will be subjected to an informal analysis as a preliminary examination of mental representations, however, the emphasis will be on stimulus validation. The responses to the data race task will be used to validate the stimulus set. Ideally we want to have an accuracy rate of approximately 90% for both data race types (yes, no). If the task is too difficult we expect there will be a higher number of no responses compared to yes responses to the data race question. We predict, with expertise as an independent variable, a positive correlation between level of expertise and confidence and a negative correlation with response time. The variables relevant to expertise will be used with the data race type (yes, no) as predictors in exploratory mixed linear models.

4. Conclusion

The results of the proposed study will indicate the validity of the stimulus set and provide direction for future studies. A valid stimulus set would allow us to move forward with our research on mental representations of expert parallel programmers.

5. References

- Bidlake, L., Aubanel, E., & Voyer, D. (2020, July). Systematic literature review of empirical studies on mental representations of programs. *Journal of Systems and Software*, *165*, 110565. doi: 10.1016/j.jss.2020.110565
- Brysbaert, M., & Stevens, M. (2018). Power analysis and effect size in mixed effects models: A tutorial. *Journal of Cognition*, *1*(1). doi: 10.5334/joc.10
- Feigenspan, J., Kastner, C., Liebig, J., Apel, S., & Hanenberg, S. (2012, Jun). Measuring programming experience. In *20th IEEE International Conference on Program Comprehension (ICPC)* (p. 73–82). IEEE. doi: 10.1109/ICPC.2012.6240511
- Liao, C., Lin, P.-H., Asplund, J., Schordan, M., & Karlin, I. (2017). Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (p. 11:1–11:14). ACM. doi: 10.1145/3126908.3126958

Purpose-first programming: Scaffolding programming learning for novices who care most about code's purpose

Kathryn Cunningham
School of Information
University of Michigan
kicunn@umich.edu

Abstract

Becoming “a programmer” is associated with gaining a deep understanding of programming language semantics. However, as more people learn to program for more reasons than creating software, their learning needs differ. In particular, end-user programmers and conversational programmers often care about code's purpose, but don't wish to engage with the low-level details of precisely how code executes. I propose the creation of scaffolding that allows these learners to interact with code in an authentic way, highlighting code's purpose while providing support that avoids the need for low-level knowledge. This scaffolding builds on theories of programming plans.

1. Semantics-focused activities don't meet the needs of certain learners

Introductory programming activities often echo the epigram “*To understand a program you must become both the machine and the program*” (Perlis, 1982). Commonly-used code tracing exercises (Sorva, 2013) ask students to simulate the execution of a program: determining the order in which lines of code run, which values are created and modified, and when the program starts and stops. Computing education researchers have created theoretical hierarchies of programming skills that describe code tracing as a primary skill that students should be taught before they learn to write code or explain the purpose of code (Xie et al., 2019).

However, for many programming learners, writing code or being able to read code and determine its purpose are far more important than understanding code semantics. In thinkaloud interviews about code tracing activities, I found learners who wanted to use code to solve problems but didn't view themselves as “a programmer” or didn't feel they can think “like a computer”. These learners rejected code tracing activities, describing them as in conflict with their self-beliefs (Cunningham, Agrawal Bejarano, Guzdial, & Ericson, 2020). At the same time, these learners expressed a value for learning about code that solved “real” tasks, not toy problems.

2. An approach that emphasizes code's purpose

To meet the needs of these learners, I propose **purpose-first programming**, an approach to learning programming skills that starts from the user's need for programming, rather than the demands of programming language syntax and semantics. Instruction will focus on common programming patterns in the domain of choice, called plans (Soloway & Ehrlich, 1984). Purpose-first programming learners assemble and tailor plans, so that they can write useful programs in an area of interest right away, without needing to understand every detail of how code works.

2.1. Purpose-first programming scaffolds make programming easier

2.1.1. Code is grouped by plan

Plans are highlighted to allow learners to “chunk” (Gobet et al., 2001) programs more easily, hopefully leading to faster problem-solving. In purpose-first programming, plans are drawn from a specific domain. As a result, information about the plans can be described in domain-specific terms that facilitate the application of the plan to solve problems.

2.1.2. Plans consist of fixed code and “slots” that can contain objects or code

“Slots” are the only areas of a plan that can be changed. This decreases the complexity of editing code and draws learners' attention to the most important parts of code.

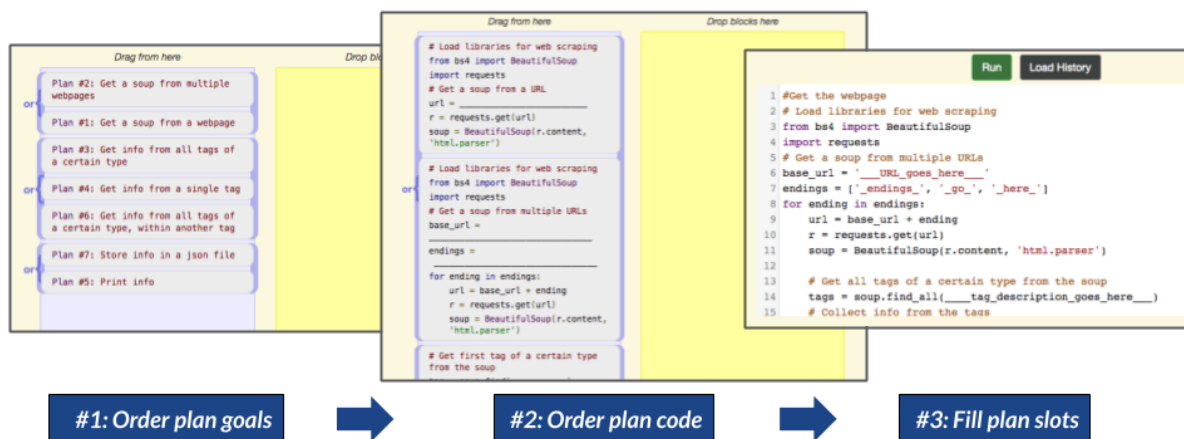


Figure 1 – Purpose-first code writing supports assembly and tailoring of plans.

2.1.3. Subgoals break plans into pieces to guide plan integration and tracing

Subgoals (Morrison, Margulieux, & Guzdial, 2015) provide a smaller unit of cohesive code, and a subgoal label clarifies the contribution to the plan's purpose. By tracing the input and output to each subgoal, learners can trace purpose-first code at a higher level of abstraction.

2.2. Designing a purpose-first programming proof-of-concept

I will design a proof-of-concept curriculum that teaches five plans from the domain of web scraping with BeautifulSoup. Learners will view examples of complete programs, learn about each plan individually, and then write, debug, and describe code that uses combinations of plans not previously seen. Activities will provide purpose-first scaffolds (see support for code writing in Figure 1).

2.3. How do learners describe the effect of purpose-first programming on their motivation?

2.3.1. Evaluating expectancy of success and value for tasks

I will perform semi-structured interviews with 6-12 novice programmers after they complete the purpose-first programming curriculum. According to the Eccles Expectancy-Value Model of Achievement Choice, motivation for a task is explained by expectancy of success on the task and value for the task (Eccles, 1983). The interviews will explore learners' feelings of success during activities in the curriculum, as well as expectations of success with similar learning in the future. The interviews will also ask learners about aspects of their value for this type of learning activity, including their enjoyment, alignment with future goals, and alignment with self-identity.

3. References

- Cunningham, K., Agrawal Bejarano, R., Guzdial, M., & Ericson, B. (2020). I'm not a computer: How identity informs value and expectancy during a programming activity. In *Proceedings of the 2020 international conference of the learning sciences*. International Society of the Learning Sciences.
- Eccles, J. (1983). Expectancies, values and academic behaviors. *Achievement and achievement motives*.
- Gobet, F., Lane, P. C., Croker, S., Cheng, P. C., Jones, G., Oliver, I., & Pine, J. M. (2001). Chunking mechanisms in human learning. *Trends in cognitive sciences*, 5(6), 236–243.
- Morrison, B. B., Margulieux, L. E., & Guzdial, M. (2015). Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research* (pp. 21–29). ACM.
- Soloway, E., & Ehrlich, K. (1984, September). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), 595–609.
- Sorva, J. (2013, July). Notional machines and introductory programming education. *Trans. Comput. Educ.*, 13(2), 8:1–8:31.
- Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., . . . Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, 29(2-3), 205–253.

An Analysis of Student Preferences for Inverted vs Traditional Lecture

Brian Harrington

Dept. of Computer and
Mathematical Sciences
University of Toronto
Scarborough

Mohamed Moustafa

Dept. of Computer and
Mathematical Sciences
University of Toronto
Scarborough

Jingyiran Li

Dept. of Computer and
Mathematical Sciences
University of Toronto
Scarborough

Marzieh Ahmadzdeh

Dept. of Computer and
Mathematical Sciences
University of Toronto
Scarborough

Nick Cheng

Dept. of Computer and
Mathematical Sciences
University of Toronto
Scarborough

Abstract

The benefits of inverted lectures are well documented, including improved retention and a focus on active, student-directed learning. However, not all students prefer the inverted lecture model. In this study, we provided students with both inverted and traditional lectures in the same introductory CS course. Students were asked to attend both styles of lecture, and at the end of the course, they were asked to compare the lectures to each other as well as to other course components such as assignments and readings.

Analyzing the responses of 243 students, we found no obvious preference trend with respect to grades. However, we did find a preference for traditional lectures among international students, as well as a very strong preference for inverted lectures among female students.

1. Introduction

Inverted or “flipped” classrooms are very popular in large introductory computer science courses, and have been shown to benefit students by improving engagement and focusing the classroom on student centered learning (Horton, Craig, Campbell, Gries, & Zingaro, 2014; Strayer, 2012). However, there are down-sides to inverted lectures as well, and the inverted lecture format may not appeal to, or be appropriate for, all types of students (Abeysekera & Dawson, 2015; Mason, Shuman, & Cook, 2013; Gannod, Burge, & Helmick, 2008).

In this study, we offered students a hybrid course, with both inverted and traditional lectures simultaneously covering the same material. Students were encouraged to attend both lecture types, but no marks in the course were directly tied to attendance. At the end of the course, students were asked to rank the various course components and choose those which they felt were most beneficial to their learning experience. This gave us an opportunity to directly compare student perceptions of the lecture styles on their own educational development.

Our initial hypothesis was that the inverted lecture model would be particularly beneficial to students who were otherwise struggling with learning to program and finding the core content overwhelming, and therefore that we would see a discrepancy in preference for lecture type by final course grade. We also believed that female identifying students would be more likely to appreciate the interactive and interpersonal nature of the inverted lectures, and that we would therefore see a corresponding preference for inverted lectures among female students. Finally, we believed that our international student population, who anecdotally reported having less group work and discussion based classes in their academic history, would display a corresponding preference for traditional lecture over the inverted model.

We therefore have the following research questions for this study:

- RQ1: Do lecture preferences vary by course grade?
- RQ2: Is there a difference in lecture preference by gender?
- RQ3: Is there a difference in lecture preferences among international students?

To form our initial hypotheses, we used anecdotal evidence and personal opinions of the authors.

Our initial hypotheses, based on anecdotal evidence and personal experiences are as follows:

- H1: There will be a negative relationship between final course grade and preference for inverted lectures.
- H2: Female identifying students will show an increased preference for inverted lectures.
- H3: International students will show a decreased preference for inverted lectures.

These hypotheses were validated through discussion of past experiences and anecdotal evidence among the teaching team and teaching assistants for the course. While there was some disagreement as to the hypothesised causes and effect sizes, particularly with respect to our international students (discussed further in Section 5.1), there was a general agreement that these were the most likely expected outcomes.

The development of the semi-inverted course model, where students simultaneously take traditional and inverted lectures on the same core material was also an interesting experience. Therefore, in Section 3, we provide a brief experience report of developing the course, along with some of the pedagogical decisions and constraints that may have impacted the study.

All of the worksheets and lecture slides used in this course are available for anyone who wishes to attempt to replicate this study, or to implement this classroom model themselves at <https://uoft.me/PPIG2020>.

2. Background

The use of the inverted classroom as a method of teaching, sometimes referred to as a ‘flipped classroom’, has increased in popularity in recent years. In 2013, 29% of higher education faculties in the United States had implemented some form of inverted learning and 27% were planning to do so according to (Bart, 2013). A substantial amount of existing research is available detailing the methodologies and implementations of inverted classroom (Aliye Karabulut-Ilgu & Jahren, 2018). The modern resurgence of inverted classroom is largely technologically focused, with content being distributed online (Lockwood & Esselstein, 2013).

Inverted classrooms have previously been shown to increase collaboration and discussion between students (Herold, Lynch, Ramnath, & Ramanathan, 2012; Strayer, 2012) with an increase in active learning (Mason et al., 2013; Timmerman, Raymer, Gallgher, & Doom, 2016), and improved student outcomes (Horton et al., 2014). There are conflicting findings on the effects of inverted lectures on student outcomes. Some studies have found that students in inverted lectures outperform those in traditional classrooms (Ossman & Warren, 2014; Schmidt, 2014; K. Yelamarthi & Drake, n.d.), some found no significant difference (B. Love & Swift, 2014; Mason et al., 2013; Olson, 2014; Swift & Wilkins, 2014; S. B. Velegol & E.Mahoney, 2015), while a few showed that the inverted classes had worse outcomes (J. P. Lavelle & Brill, 2013; McClelland, 2013). Inverted lectures introduce some challenges to both instructors and students. A common component of the modern flipped classroom is the distribution of online video lessons covering course content. The increased technical requirements of online videos can add significant overhead to course preparation for the first semester that the course is taught (V. Kalavally & Khoo, 2014; Gannod et al., 2008), but subsequent semesters would have reduced course preparation

time (Herold et al., 2012). Creating high quality videos is time consuming (Stephenson, 2019) and poor quality or inappropriate length videos can have a negative impact on student engagement (Olson, 2014). The inverted lecture can also result in increased interaction demands during the course (R. M. Clark & Besterfield-Sacre, 2014).

Some research has pointed to issues regarding motivation in inverted classrooms. Since the effectiveness of the model is reliant on students completing pre-class assigned work, more time is required for students to watch videos or complete readings prior to lecture, which causes problems for students with poor motivation or time management skills (Abeysekera & Dawson, 2015). Many inverted classroom models use quizzes at the beginning of lecture as a means of forcing students to arrive prepared (Toto & Nguyen, 2009). However, this can lead to increased student anxiety, a diminished sense of trust and a sense of patronizing (Herold et al., 2012; Mason et al., 2013). Some research has shown that students who are used to traditional lecture structures can be resistant to accepting an inverted model, particularly if they arrive from a cultural context where classroom interactivity is less common (Gannod et al., 2008; A. Amresh & Femiani, 2013; Bland, n.d.).

3. Course Structure and Development

In this section, we provide the details of our hybrid model alongside a brief pedagogical explanation of the reasoning behind its development.

The initial goal of our hybrid lecture model was to create a way to run our lectures that had some of the benefits of the flipped classroom, but with a lower barrier to entry. In particular, we wished to see if it was possible to achieve some of the self-direction and student focused learning aspects of an inverted lecture without making major changes to the logistics of the course or needing to develop large amounts of online resources.

A secondary desire was to keep as much of the experience ‘in-the-room’ as possible, as The University of Toronto Scarborough already has a large commuter community, and we wanted to keep the students focused on being physically present in the classroom, without feeling like they were really working remotely and only coming to class for administrative purposes.

A third goal of the project was to develop a teaching model that was not overly paternalistic; many inverted classroom models have marks awarded for attendance or mandatory in-class quizzes that primarily serve to ensure that students come to lecture. While this may be beneficial for many students, our teaching team felt that students should be treated as adults, and be given as many opportunities as possible to choose their own method of learning and level of commitment.

Prior to the project, the course consisted of three hours of lecture per week, organized into a two hour and a one hour block, on different days of the week. There were also one hour weekly tutorials, run by undergraduate teaching assistants, and drop in practical sessions where students could get help with weekly exercises or class material. The course had two term tests and a final exam as well as three larger programming assignments, weekly exercises and quizzes held in tutorial.

For the hybrid model, we converted one hour of lecture time to an inverted model. So students would still receive two hours of traditional lecture, unchanged from previous course offerings, aside from a reduced amount of live coding examples. The inverted hour covered material from the previous week (this was necessary as some course sections had their 1 hour block earlier in the week than their 2 hour block), and consisted of worksheets that were handed out at the beginning of the hour, and submitted electronically via the MarkUs submission system (Magnin et al., 2012). Each worksheet was worth 0.5% (10 weekly worksheets for a total of 5% of the course grade).

Attendance was not taken in either lecture, and it was possible for students to complete the worksheets without attending the inverted lecture. However, students were informed that sufficient help would be offered during the inverted lecture to guarantee that if they attended and worked during the hour, they would receive full marks on the worksheet. Furthermore, several of the worksheets were specifically

Implement the program outlined on your specification page. Your code must consist of at least one function which raises exceptions for bad input, and global code which calls the function(s) and deals with bad input appropriately. When you think you have everything working properly, swap with your partner, and see if you can find test cases that break their code. If your partner found a test case that breaks your code, fix your code so that it passes the test case. Repeat as necessary until you can't find any more errors. If you still have time remaining, challenge your partner to add extra functionality.

Figure 1 – Example Worksheet Prompt

designed for group work with a think-pair-share model. An example prompt for such a worksheet can be found in Figure 1.

Both traditional and inverted lectures were taught by the same teaching team, with students randomly assigned to lecture sections.

4. Analysis

At the end of the semester, students were asked to pick the top three course components that they found most beneficial to their learning experience, and to rank those components. We had administered a similar survey at the end of the previous year. The survey was completed online in the final week of the term in exchange for a bonus mark on the final exam.

The options available to students were:

- Reading - weekly online readings which were assigned
- Practicals - drop in practical sessions where students could complete weekly exercises, or get help with other programming questions
- Tutorials - weekly in-class tutorials, led by TAs, reviewing topics covered in lecture
- Assignments - three large programming assignments spread throughout the term
- Exercises - weekly programming questions, submitted online and auto-marked
- Inverted Lecture
- Lecture

In the 2015-16 version of the survey, the options were the same except that Inverted Lecture/Worksheet was not available for obvious reasons.

We developed a simple model to compare the results of these surveys, whereby the highest ranked component was given a score of 3, the second highest was given a score of 2 and the third highest a score of 1. Cumulative totals for each component were then computed, and compared as a percentage of total marks allocated.

This “pick your top three” method of relative evaluation was chosen rather than either directly questioning the students as to their preference for traditional or inverted lectures in order to be compatible with previously collected data, and to avoid leading or biasing questions. In pre-study evaluations with a different student group, we found students to be very susceptible to the wording of direct questions: “Did you prefer inverted lectures to traditional ones” vs “Did you prefer traditional lectures to inverted ones”, and providing students with a likert scale resulted in most students giving the same or very similar results for all course components.

Table 1 – Participant Demographics

	Total Count	Percentage
Total Participants	243	
Gender		
Female	61	25.1%
Male	154	63.4%
Other/Did not specify	28	11.5%
Student Status		
Domestic	104	42.8%
International	115	47.3%
Unknown	24	9.9%

4.1. Demographics

Out of a total of 264 students enrolled in the course, 243 agreed to participate in this study. 154 students identified as Male, 61 as female, and 28 chose not to specify. 104 of the students were registered as domestic students while 115 were registered as international, 24 students registration status could not be determined. The demographics are summarized in Table 1.

This study did not include racial demographics, but the department’s international student cohort overall was drawn 65% from China, 12% from other Asia Pacific countries, 6% from India/Pakistan, 5% from Europe, 6% from other countries in the Americas and 2% from Africa.

4.2. Comparison with Historic Data

A comparison of the point allocation for the course offered prior to the commencement of the project and for the improved semi-inverted version of the course can be found in Figure 2.

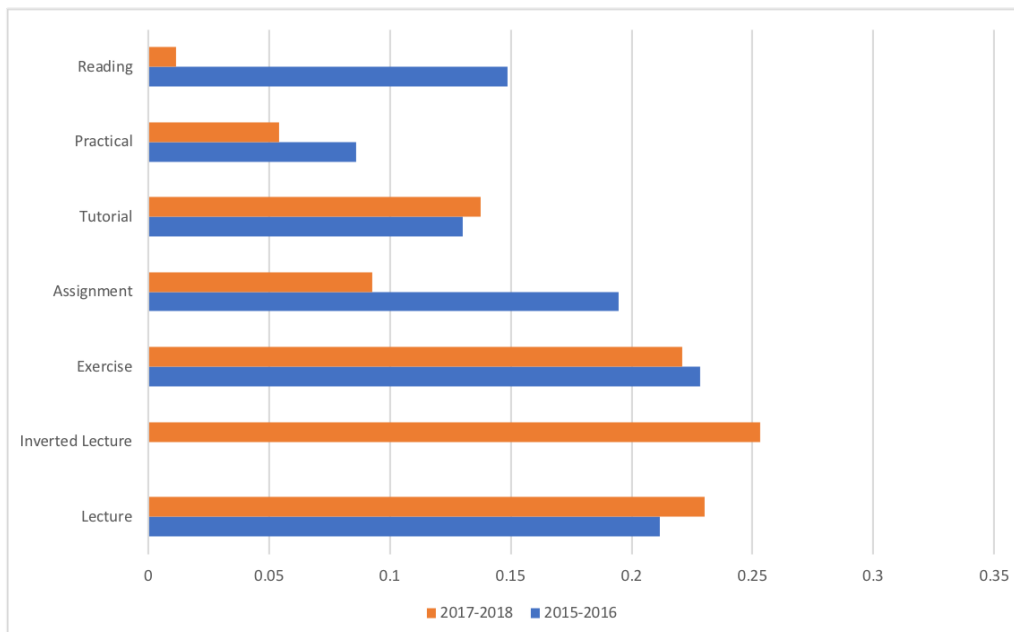


Figure 2 – Student survey results for traditional and semi-inverted offerings

While lectures were always a popular course element, second only to weekly exercises in terms of perceived learning benefit, the inverted lectures immediately became the most popular component, beating out both the traditional lectures and exercises.

Of particular interest is where the points now allocated to the inverted lecture appear to have come from. It would be natural to assume that the points previously allocated to lecture would now be split among the lecture and inverted lecture options. However, it seems that the points allocated to lecture remained relatively constant, whereas the points allocated to assignments dropped significantly and those allocated

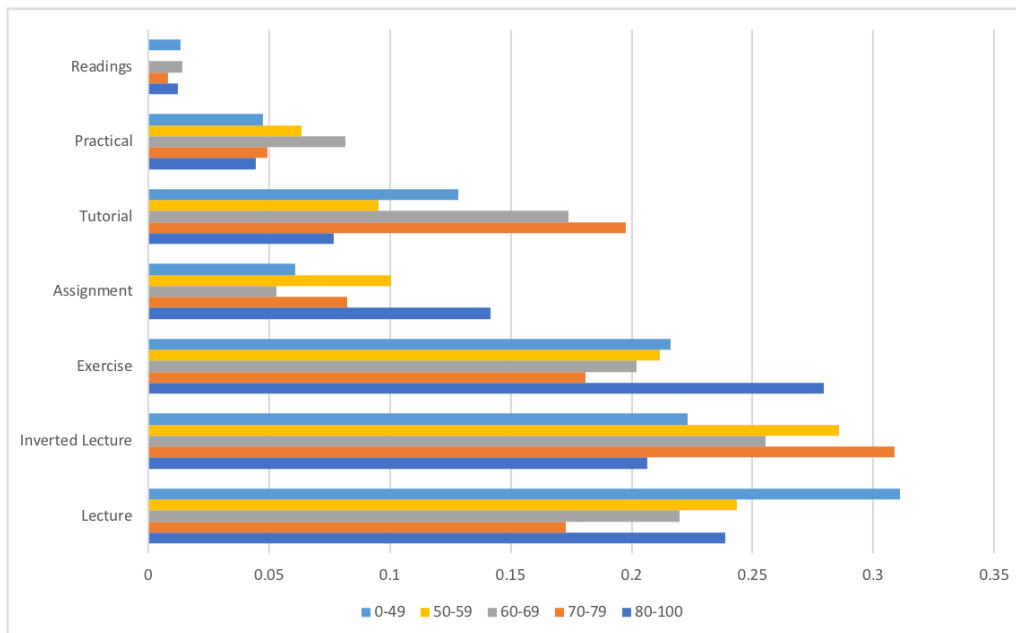


Figure 3 – Student survey results by final course grade

to readings dropped drastically. While it is possible that the assignments points were due to different assignments being used in the two years, the readings were kept constant.

We cannot draw any strong conclusions from the differences between preferences in the two cohorts, due to the course changing and shifting student preferences and attitudes over time. However, comparing the results of our survey to that of previous data indicates that the changes in preferences are likely due to the change in teaching model. Furthermore, this process does provide some insight into the changes in relative perceived importance of various course components.

4.2.1. Course Outcomes and Grades

Due to the difference in the final examination and assignment questions, it was not possible to directly compare course outcomes between the two years. The averages and grade distributions were not significantly different. However, since the graded material and grading scheme changed between years, a direct comparison is not helpful.

4.3. Sub-Group Analysis

While knowing the overall perceived benefits of the inverted lectures is interesting, we wanted to analyze further in order to see if the distributions changed for various sub-groups.

4.3.1. Course Grade

We first checked the distribution of preferences by final course grade. As shown in Figure 3, the inverted lecture appears to be least popular among students at the lower and higher ends of the grade spectrum (those with final marks below 50 or above 80), and more popular with students in the middle of the spectrum. The opposite effect is seen for the traditional lecture, where students in the middle of the grade spectrum responded less favourably than their counterparts. However, this result was not statistically significant, and does not appear to follow any clear trend. We therefore conclude that there was no evidence of a relationship between lecture preference and course outcome ($\chi^2 = 13.9432$, $df = 24$, $p = 0.9479$).

4.3.2. International vs Domestic Students

We next separated the data by international and domestic students. As seen in Figure 4, it appears that domestic students believed the inverted lecture was more beneficial than the traditional lectures, their international peers had the opposite opinion.

The differences between the two groups was not large, and after performing a Pearson’s chi-squared test with Yates’ continuity correction based on the number of students putting inverted lecture in their top three components, we found no statistically significant difference ($\chi^2 = 0.061719$, $df = 1$, $p = 0.8038$).

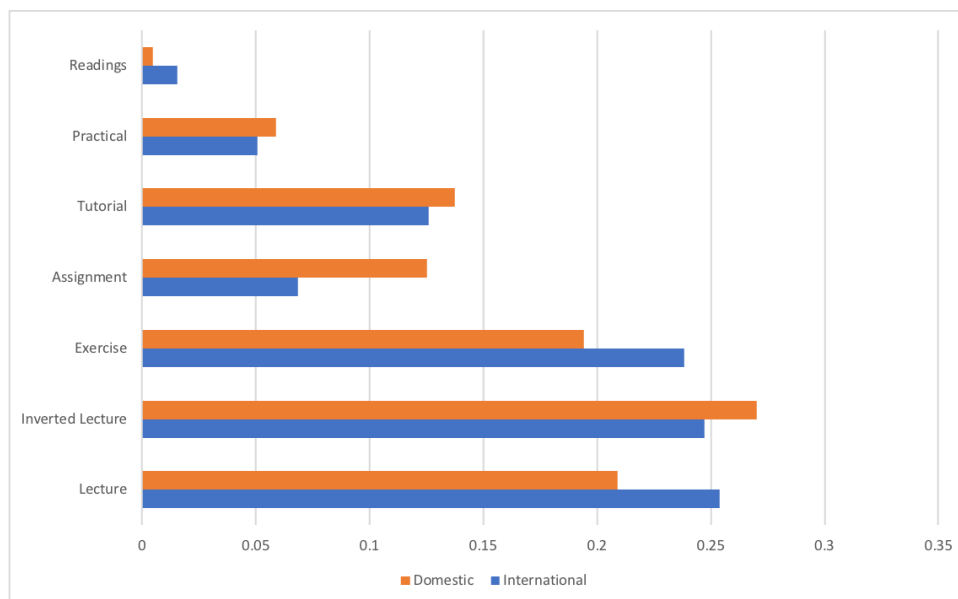


Figure 4 – International vs domestic student survey results

4.3.3. Gender

Grouping the data by gender produced very interesting results, as can be seen in Figure 5. Most of the course components were fairly closely ranked, but the traditional lecture was perceived as more helpful by male students, with the inverted lecture being ranked much more highly by female students. 62.3% of female students had inverted lecture as one of their top three components, compared to only 42.9% of males. Running a Pearson’s chi-squared test with Yates’ continuity correction based on the number of students putting inverted lecture in their top three components showed that there was in fact a statistically significant difference ($\chi^2 = 5.8551$, $df = 1$, $p = 0.01553$).

5. Conclusions and Future Work

In this study, we analyzed the student preferences for traditional vs inverted lectures. By developing a hybrid model which allowed students to simultaneously participate in both lecture types, covering the same core material, we were provided with a unique opportunity for direct comparison.

We found that students overall found the inverted lectures useful, pushing readings and assignments out of many of their top three course components. And while there was not conclusive data of the effect of the introduction of these lectures on student outcomes, that was beyond the scope of this project, and has been well studied elsewhere.

As for our primary research questions, we found the following results:

- **RQ1: Do lecture preferences vary by course grade?**

While there were slight differences in preference at various grade levels, there was no clear pattern, and the results were not statistically significant. We must therefore conclude that there is no evidence of a correlation between course grade and lecture preference.

H1: Our hypothesis was not supported.

- **RQ2: Is there a difference in lecture preferences among international students?**

International students did show more of a preference for traditional lecture, while domestic stu-

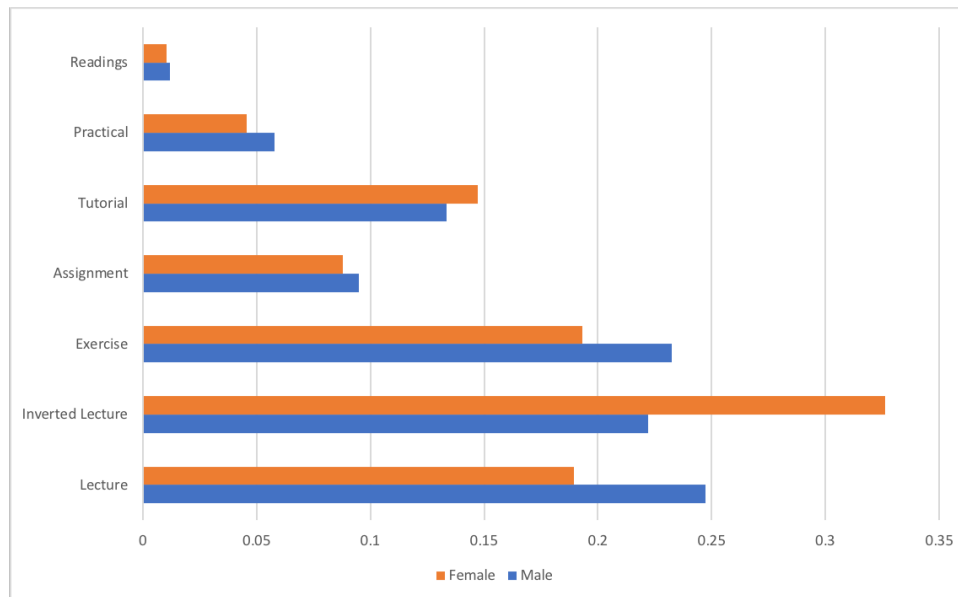


Figure 5 – Male vs female student survey results

dents ranked inverted lectures higher. However, this result was not found to be statistically significant.

H2: Our hypothesis was not supported.

- **RQ3: Is there a difference in lecture preference by gender?**

There appears to be a quite large and statistically significant difference in preferences by gender, with females showing a marked preference for inverted lectures over the traditional model, while their male colleagues showing a slight trend in the opposite direction.

H3: Our hypothesis was strongly supported.

In addition to allowing for interesting research findings, the hybrid model appears to have achieved its original goal of allowing for a low barrier to entry method for obtaining many of the benefits of inverted classrooms, without many of the drawbacks. We intend to maintain this hybrid model in future and hope to replicate this study, while improving the learning experience of our students.

5.1. Threats to Validity

This experiment was conducted on a single cohort of students at a single institution. Therefore the results should not be assumed to extrapolate to a global context without further research.

Reasonable attempts were made to control relevant variables in the course. The teaching team remained constant, with the same instructor teaching both sets of lectures with the same material in both the experimental and comparison years. However, it is entirely possible that some of the results could be attributable to the course or the instructors themselves. While we have provided all our course materials at the link given in the introduction, it is entirely possible that these results are indicative of preferences for our particular method of running inverted lectures, and would have been different for a different teaching team or lecture setup.

We have identified several threats to the validity of the study, which could also be the basis for future studies with more clearly refined controls.

Validity of Survey Results:

The primary purpose of the survey was to ask for student feedback in order to improve the course. While it is reasonable to assume that students would want to respond honestly, as it may affect future courses during their time at the university, it is possible that students could answer dishonestly.

Comparison across Cohorts:

Comparing survey results of the 2017-2018 cohort with those of the 2015-2016 cohort is naturally problematic as this is not a controlled environment, and it is possible that shifts in the student body or course instruction could have a major impact on the result we see in Figure 2. In future studies, we would like to more closely control for these factors and see if the results replicate.

Treating International Students as a Single Demographic Group:

This was a very contentious matter among the research team, as we only had authorization to collect international student status, and not more salient information such as country of origin, level of English proficiency, or pedagogical history. We ultimately decided to report on this data, because we had heard many anecdotal reports that our international student body, especially those from China and Asia-Pacific (who make up 77% of our international students), do not like inverted models. In future studies, we would like to produce a much more nuanced survey, focusing less on fee status, and more on linguistic and pedagogical history.

Survey Design - Using perceived benefit as a proxy for preference.

There are limitations to our “pick your top 3” model. However, it was chosen to be consistent with previous data, and to be as simple and easy to answer as possible. We report our results as preference, while the question itself asks which components students felt benefited them the most. In future studies, we could split this question to specifically ask which components students enjoyed most and which they felt they learned from the most, as it’s possible those two concepts are not directly linked.

Many of these threats to validity come from the simple fact that the primary goal of this project was to obtain naturalistic feedback on a pedagogical development. While all reasonable precautions were taken to ensure threats were mitigated, pedagogical development was our main concern. Future work will be needed in order to validate these results. However we feel that this work provides a model for future study and an interesting data point providing evidence of a difference in gender preference.

5.2. Data Availability

All of the relevant material for this project can be found at <https://uoft.me/PPIG2020>.

6. References

- A. Amresh, A. C., & Femiani, J. (2013). Evaluating the effectiveness of flipped classrooms for teaching cs1. In *Proceedings of frontiers in education conference* (p. 733–735).
- Abeysekera, L., & Dawson, P. (2015). Motivation and cognitive load in the flipped classroom: definition, rationale and a call for research. *Higher Education Research & Development*, 34(1), 1-14.
- Aliye Karabulut-Ilgu, N. J. C., & Jahren, C. T. (2018). A systematic review of research on the flipped learning method in engineering education. *British Journal of Educational Technology*, 49(3), 398–411.
- Bart, M. (2013, November). *Survey confirms growth of the flipped classroom* (<https://www.facultyfocus.com/articles/blended-flipped-learning/survey-confirms-growth-of-the-flipped-classroom/> No. Last Accessed: 23-04-2019).
- Bland, L. (n.d.). Applying flip/inverted classroom model in electrical engineering to establish life-long learning. In *Proceedings of asee annual conference & exposition*.
- B. Love, N. G., A. Hodge, & Swift, A. W. (2014). Student learning and perceptions in a flipped linear algebra course. *International Journal of Mathematical Education in Science and Technology*, 45, 317–324.
- Gannod, G. C., Burge, J. E., & Helmick, M. T. (2008). Using the inverted classroom to teach software engineering. In *Proceedings of the 30th international conference on software engineering* (pp. 777–786). New York, NY, USA: ACM.
- Herold, M. J., Lynch, T. D., Ramnath, R., & Ramanathan, J. (2012, Oct). Student and instructor experiences in the inverted classroom. In *2012 frontiers in education conference proceedings* (p. 1-6).
- Horton, D., Craig, M., Campbell, J., Gries, P., & Zingaro, D. (2014). Comparing outcomes in in-

- verted and traditional cs1. In *Proceedings of the 2014 conference on innovation & technology in computer science education* (pp. 261–266).
- J. P. Lavelle, M. T. S., & Brill, E. D. (2013). Flipped out engineering economy: Converting a traditional class to an inverted model. In *In a. krishnamurthy & w. k. v chan (eds.), proceedings of the 2013 industrial systems engineering research conference* (p. 397-407).
- K. Yelamarthi, S. M., & Drake, E. (n.d.). A flipped first-year digital circuits course for engineering and technology students. *IEEE Transactions on Education*, 58, 179–186.
- Lockwood, K., & Esselstein, R. (2013). The inverted classroom and the cs curriculum. In *Proceeding of the 44th acm technical symposium on computer science education* (pp. 113–118). New York, NY, USA: ACM.
- Magnin, M., Moreau, G., Varoquaux, N., Vialle, B., Reid, K., Conley, M., & Gehwolf, S. (2012). Markus: An open-source web application to annotate student papers on-line. In *Asme 2012 11th biennial conference on engineering systems design and analysis* (pp. 301–307).
- Mason, G. S., Shuman, T. R., & Cook, K. E. (2013, Nov). Comparing the effectiveness of an inverted classroom to a traditional classroom in an upper-division engineering course. *IEEE Transactions on Education*, 56(4), 430-435.
- Mcclelland, C. J. (2013). Flipping a large-enrollment fluid mechanics course—is it effective? In *Proceedings of the 120th asee annual conference & exposition*.
- Olson, R. (2014). Flipping engineering probability and statistics—lessons learned for faculty considering the switch. In *Proceedings of the 121st asee annual conference & exposition*.
- Ossman, K. A., & Warren, G. (2014). Effect of flipping the classroom on student performance in first year engineering courses. In *Proceedings of the 121st asee annual conference & exposition*.
- R. M. Clark, B. A. N., & Besterfield-Sacre, M. (2014). Preliminary experiences with “flipping” a facility layout/material handling course. In *Proceedings of the 2014 industrialand systems engineering research conference*.
- S. B. Velegol, S. E. Z., & E.Mahoney. (2015). The evolution of a flipped classroom: Evidence-based recommendations. *Advances in Engineering Education*, 4, 1-37.
- Schmidt, B. (2014). Improving motivation and learning outcome in a flipped classroom environment. In *Proceedings of 2014 international conference on interactive collaborative learning* (p. 689–690).
- Stephenson, B. (2019). Coding demonstration videos for cs1. In *Proceedings of the 50th acm technical symposium on computer science education* (pp. 105–111).
- Strayer, J. F. (2012). How learning in an inverted classroom influences cooperation, innovation and task orientation. *Learning environments research*, 15(2), 171–193.
- Swift, T. M., & Wilkins, B. J. (2014). A partial flip, a whole transformation: Redesigning sophomore circuits. In *Proceedings of 120th asee annual conference & exposition*.
- Timmerman, K., Raymer, M., Gallger, J., & Doom, T. (2016, Aug). Educational methods for inverted-lecture computer science classrooms to overcome common barriers to stem student success. In *2016 research on equity and sustained participation in engineering, computing, and technology (respect)* (p. 1-4).
- Toto, R., & Nguyen, H. (2009, Oct). Flipping the work design in an industrial engineering course. In *2009 39th ieee frontiers in education conference* (p. 1-4).
- V. Kalavally, C. L. C., & Khoo, B. H. (2014). Technology in learning and teaching: Getting the right blend for first year engineering. In *Proceedings of 2014 international conference on interactive collaborative learning* (p. 565–570).

Exploring the Coding Behaviour of Successful Students in Programming by Employing Neo-Piagetian Theory

Natalie Culligan

Department of Computer Science
Maynooth University
natalie.culligan@mu.ie

Kevin Casey

Department of Computer Science
Maynooth University
kevin.casey@mu.ie

Abstract

We have collected data from approximately 300 students in their third-level first year Introduction to Programming module as they learn to write code using our in-house pedagogical coding environment, MULE. This data includes performance in lab exams and pseudocode questions, and data on code compiled, code run, and code evaluated, which we call CRE data. Evaluations are automatically graded and feedback is provided to students on their code. The student can only evaluate their code in the scheduled lab place and times but can evaluate as many times as they wish without penalty. The pseudocode questions are used to examine the students' understanding of programming concepts, by removing the use of the compiler and comparing their performance in pseudocode questions to CRE data. Using a Neo-Piagetian framework, we examine pseudocode performance, lab exam performance and programmer behaviour in terms of CRE data. We investigate CRE data as signs of a student's progression through the three stages of Piagetian understanding and build a series of Deep Neural Net binary classifiers to test if this passively collected behavioural data can be used to detect students in danger of failing.

1. Introduction

Computer Science has one of the highest failure and dropout rates in 3rd level education (Bennedsen, & Caspersen, Corney *et al.* 2010, Lang *et al.*, Watson & Li). In this paper, we will investigate if students in introductory computer science courses are failing to reach the later stages of Neo-Piagetian understanding, and if we can investigate and observe signs of these stages through passive data collection, and the results of pseudocode tasks in the weekly practical coding labs. The research question for this study is:

- Can we observe signs of progression through the Neo-Piagetian stages of learning by examining passively collected data on students' coding behaviour?

The coding behaviour data we discuss in this paper is the order in which students compile, run, and evaluate their code. Evaluation provides the student with automatic grades and feedback. The students use the pedagogical coding system MULE to complete their weekly coding tasks. In this system, students are unable to run their code until they have successfully compiled and cannot evaluate their code until it has run successfully.

In the doctoral thesis "*Neo-Piagetian Theory and the Novice Programmer*" (Teague), the author states that "*Programming competence requires abstract reasoning skills and learning to program is about the sequential and cumulative development of those abstract reasoning skills in an unfamiliar domain.*" We wanted to introduce pseudocode questions into our first-year curriculum to encourage students to build mental models of programming concepts by requiring students to predict code output, without relying on the compiler. These pseudocode questions are English language representations of code that cannot be run with a compiler but represent programming concepts such as loops and arrays (Lopez *et al.*). With pseudocode, we can see if the students can abstract the concepts away from Java and apply what they have learned in class in a much more generalised way. This is useful as if the

students are able to do so, they are more likely to be able to reuse the skills and apply them in a variety of ways, instead of memorizing and replicating techniques they have used in the past.

In this paper, we will discuss our findings when investigating CRE data in weekly labs as students graduate from random/loosely guided “tinkering” to more intentional code-writing. While previous work has discussed “tinkering” as a viable method of learning programming, we will discuss if this is true throughout the first semester, or if CRE data that implies an over-use of tinkering is in fact an indication that a student is not developing a good mental model of fundamental programming concepts and is therefore in danger of falling behind.

2. Related Research

2.1. Student Behaviour when Learning to Code

There have been numerous studies that investigate novice programmer behaviour such as patterns of compilation and running of code and how it relates to student success.

Perkins *et al.*, investigate the different strategies that novice programmers adopt when learning to code, and describe what they term “stoppers”, “movers”, and “extreme movers”. “Stoppers” are novices who, when faced with a problem without a clear course of action, stop attempting to find a solution to the problem and appear to be unwilling to explore the problem any further. “Movers” are novices who will constantly modify and test their code when faced with a problem. “Extreme Movers” will also constantly modify and test their code but are different from movers in that they do not seem to learn from attempts that previously did not work, and they do not continue to work on solutions that fail the first time so do not end up “homing in” on a working solution. The authors do not specifically speak about how these different patterns relate to compilation and run behaviour, but the below papers do touch on it in direct reference to this study.

Two papers on the programming environment BlueJ (Jadud, 2005, Jadud 2006) discuss the behaviours of the authors’ students, and how similar their students’ behaviours are to those in the above Perkins *et al.* paper. They discuss their own “extreme movers”, which they describe as “tinkerers”, and how these students would sometimes allow their experimental code to accumulate, causing their code to become increasingly complex and, eventually, incomprehensible. The BlueJ studies found that 24% of all compilation events followed less than 10 seconds after a previous compilation, and half of all compilation events occurred less than 40 seconds after a previous compilation. Students spent more time working on their code after a successful compilation than they did trying to fix a syntax error. The authors found that students tend to program in large blocks, then spend time writing and compiling code in small bursts in order to fix syntax errors. Accordingly, multiple compilations may indicate a large number of syntactic problems.

In “*Studying the Novice Programmer*” (Soloway & Spohrer) the authors discuss the need for students to build plans. As mentioned above, students who tinker aimlessly create bugs, and without clear goals may fail to progress towards a working solution. The authors used natural language to investigate if students with plans, broken into small tasks, are more successful when programming.

In “*Analysis of Code Source Snapshot Granularity Levels*” (Vihavainen) the author discusses the ratio of “snapshots to submissions”, where a snapshot is a copy of the code taken every time the student saves, compiles, runs, or tests their code. Submissions are final versions of a program submitted for correction/grading, provided by a plugin for NetBeans that provides feedback and grading to the student. Using a Wilcoxon rank sum test, the authors found a statistically significant difference between the number of runs and tests for students with previous programming experience and those without. This difference continued to be visible throughout the course, although the behaviour of the participants was more alike in the final weeks of the course, perhaps implying that these behaviours are indicators of programming proficiency.

One of the research questions in the paper “*Evaluating Neural Networks as a Method for Identifying Students in Need of Assistance*” (Castro-Wunsch) is “*Are neural network (NN) models appropriate for the task of identifying students in need of assistance?*” The authors found that, yes,

neural networks predicted at-risk students at least as well as Bayesian and decision tree models, and had the advantage of being “pessimistic”, meaning that the neural networks were more likely to incorrectly classify students as at-risk, rather than incorrectly classify students as not at-risk. From this research, we decided to use neural networks as our classifier.

2.2. Neo-Piagetian Theory and Abstraction in Programming

There are also a number of studies that use Neo-Piagetian theory in examining student behaviour in computer science and discuss abstraction in relation to novice and expert programmers.

In “*Concrete and Other Neo-Piagetian forms of Reasoning in the Novice Programmer*”, (Lister) the author discusses the reasoning behind the use of Neo-Piagetian theory. Classical Piagetian theory considers the progress through different stages of learning to be a consequence of a biological maturing of the brain. Neo-Piagetian theory, on the other hand, considers this instead a result of gaining experience, and in particular, the ability to “chunk” knowledge within a certain knowledge domain.

Corney et. al (2011) describe a study in which almost half of the sample students were unable to answer a simple explain-in-plain-English question in the third week of their introductory programming course, showing that students were encountering problems much sooner than could be detected by traditional programming questions/examinations.

In “*Neo-Piagetian Theory and the Novice Programmer*” (Teague, 2015), the author found that the development of programming skills is both “*sequential and cumulative*”, and that behaviours associated with sensorimotor and preoperational reasoning are evident from very early in the semester.

The authors of “*Mired in the Web: Vignettes from Charlotte and Other Novice Programmers*” (Teague et al.) ask if a student can have different levels of ability for different tasks which test similar programming concepts – if a student can trace and understand code, can they also *write* that code? They also ask why some students do not seem to be able to understand code with abstractions and instead rely on tracing code with specific values. The study found that students who were still operating at the sensorimotor level in week 2 were often still operating the same way in week 5, and were lagging behind students who were operating at the preoperational level in week 2. They defined students in the preoperational stages by certain behaviours which they observed using think-aloud data from students. Preoperational behaviours were guessing, a fragile grasp of semantics, confused use of nomenclature, an inability to trace simple code, as well as general misconceptions. Errors due to cognitive overload and reluctance to trace were considered behaviours associated with both sensorimotor and preoperational. The ability to trace but not explain code, as well as a reliance on specific values, were signs of the preoperational stage. The authors note that students may achieve marks for guessed answers, but it is not until they listen to the students speak aloud their thought process that they were able to get a clear picture of the students understanding and ability.

Shneiderman and Mayer found that expert programmers were able to recall more of a program than novices when it was presented to them in normal order, but not when it was scrambled, implying that the experts were able to “chunk” information together when the code made sense. The authors proposed that experienced programmers construct functional representations of computer programs.

Adelson found that expert programmers’ memory chunks tended to be semantically or functionally related, while novices typically chunked by syntax. Semantic knowledge consists of programming concepts that are generalized, and independent of programming language, whereas syntactic knowledge is more precise and rooted in exact representations of concepts in specific programming languages. For example, a novice may think of a loop as a specific for loop in Java, but an expert planning a piece of code may simply think of a loop abstractly, as something that performs a needed function, without thinking about the exact type of loop, the details of the iteration, or the syntax associated with it (Bisant & Groninger, Wiedenbeck).

3. Methodology

For this study, we collected data from around 300 students as they completed their introduction to programming module in Java using MULE, our in-house, browser-based pedagogical coding environment (Culligan & Casey). This system resembles a desktop with both built-in applications for content and assignment delivery, and a code editor for completing, running, and evaluating code for

assignments. MULE also includes mechanisms for making sections of the material invisible to some users until some constraints are satisfied such as date/time and IP address – this was used to allow certain assignments to only be accessible in the scheduled lab times and locations. Within MULE, each attempt the student makes on an assignment is recorded, and the student can easily recover any previous attempt, allowing the student to “tinker” and experiment with their code without fear of losing any work. There is evidence to suggest that a certain amount playing/tinkering with code is an indication of student success (Berland *et al.*, Berland & Martin).

For 5 of the 10 mandatory computer lab sessions during the first semester of their computer science course, students were asked to predict the outcome of pseudocode snippets, along with their usual lab consisting of two programming questions, and some peer-programming tasks. The students were told that they are not awarded any marks towards their continuous assessment for answering the pseudocode questions. The students have access to most of the programming tasks before the lab and can write code, compile, and run it, but not evaluate it for continuous assessment grades. Some of the exercises are only accessible in the labs at the assigned times, so students must write, run, compile, and evaluate the code in the lab.

Although students were able to work on an assignment before assigned lab times, we chose to look exclusively at the data from lab times. Our reasoning is that students outside of labs can be in very different environments – some may have a quiet place to work undisturbed, others may be working in a noisy environment or may be frequently interrupted, so comparisons of their behaviour may be less insightful than those from a formal lab. For most of the semester, the students can only evaluate from inside the lab during the specified lab times, so the data from outside the labs would only have compile and run events.

We did not include data from students who did not participate in the weekly labs (missing more than four), as we wanted to investigate changes in behaviour from week to week and to look at at-risk students who are actively engaging in the course labs on a weekly basis (Castro-Wunsch). After removing students who did not complete 4 or more labs, we were left with 266 subjects. The gathered data is the patterns of student compile, run and evaluate actions:

- Compile: Students cannot run their code until it compiles successfully
- Run: Students cannot evaluate their code until it runs successfully
- Evaluate: The student’s code is assigned a grade, and feedback is provided.

This data was used to build Deep Neural Net binary classifiers, that would classify students as being in either the top 50% or the bottom 50% of the class lab exam grades on a week-to-week basis. Each weekly classifier would use the CRE data for each assignment for that week, and from all previous weeks. Below we discuss the results of statistical tests exploring correlations between student behaviour and outcome, and the classifier built to predict student outcome.

4. Analysis

When analysing the data, for every time a student performs a CRE action, we look at that action and the one before and record it as a “movement” - the student moves from a Compile to a Run, is recorded as C2R, or a Run to an Evaluate in R2E for example. When processing this data, we looked at each movement as a percentage of all actions a student took during that lab. From the previous studies on programming and Neo-Piagetian stages, we expected to see the following as signs of progression through the stages:

Sensorimotor Stage: Interacting almost randomly, with little understanding of the outcome, resulting in more C2C movements, less C2R movements and less participation and success with pseudocode questions.

Preoperational Reasoning Stage: The student is beginning to master writing compilable code, and can predict code outcome, resulting in higher amount of C2R movements and R2C movements, fewer C2C movements and more participation and success with pseudocode questions.

Concrete Operational Stage: At this stage, programmers have a good grasp of concepts allowing the programmer to write more complex code, resulting in fewer C2C movements, fewer C2R movements, more R2E movements and more participation and success with pseudocode questions.

The students in the study were divided into two groups: those in the top 50% of the class in lab exam grades, and those in the bottom 50%. The two data sets contain the percentages of total movements per week for each student. A sample of the student data for a week would look like the following:

C2C	C2R	R2C	R2R	R2E	E2C	E2R	E2E
0.33997	0.254913	0.127186	0.01639	0.130208	0.111902	0.003655	0.004159

Table 1: Example of an average sample of student weekly data

The following tests were then run on the two data sets:

- To examine if the differences between the two groups were significant, t-tests were used.
- Linear regression was used to find which movements were most related to lab exam outcome, on a week-to-week basis, to select which movement data would be used in the classifier.
- Finally, the data from the most significant movements each week are used to create a Deep Neural Net binary classifier, to classify each student as being in the top or bottom 50% of the class.

5. Results

To find if there were significant differences between the top and bottom 50% of the students, t-tests were used, the results of which are considered significant differences between the two groups if the result is less than 0.05. These results are in bold. The p-value results of the groups according to lab exam results are in Table 2, and the results of the groups divided by pseudocode performance are in Table 3. Lab 6 and lab 10 included lab exams, during which the students could not look at their previously written code from earlier labs.

	C2C	C2R	R2C	R2R	R2E	E2C	E2R	E2E
1	0.001214	0.470967	0.539369	0.448587	0.090213	0.070492	0.650004	0.24305
2	0.004757	0.02424	0.665045	0.674198	0.005787	0.060203	0.388143	0.260525
3	4.80E-06	5.04E-05	0.112107	0.585846	0.031448	0.498212	0.120838	0.280715
4	1.50E-06	3.16E-08	4.04E-06	0.032048	0.305453	0.031562	0.951828	0.748698
5	4.03E-10	1.47E-08	0.008756	0.100132	0.000341	0.004397	0.015933	0.10881
6	9.00E-14	7.94E-15	5.90E-11	0.019153	0.064127	0.122571	0.026066	0.940655
7	2.05E-06	6.28E-09	6.96E-07	0.561189	0.111728	0.140013	0.924634	0.579504
8	5.73E-13	2.60E-09	0.00853	0.00715	3.36E-05	2.50E-05	0.501948	0.04282
9	0.002878	0.187736	0.187655	0.386204	0.008422	0.0083	0.234538	0.389353
10	6.05E-06	4.70E-07	0.012479	0.683429	0.034407	0.082467	0.609155	0.758995

Table 2: Results of t-test on groups divided by lab exam results

There were significant differences between the two groups found in C2C every week, C2R most weeks, and R2E and R2C in 8 of the 10 weeks. In Table 3, we see that the results are similar results to the lab exam t-tests, the main difference being that the R2E movements are almost never significant.

	C2C	C2R	R2C	R2R	R2E	E2C	E2R	E2E
1	0.656744	0.010296	0.167739	0.096637	0.368741	0.27848	0.979841	0.062224
2	0.007839	0.01318	0.698311	0.79891	0.005272	0.518315	0.074192	0.384717
3	0.001236	0.000551	0.049986	0.141391	0.498473	0.615511	0.872643	0.817136
4	0.00078	0.000378	0.015029	0.089415	0.768301	0.095344	0.72989	0.085109
5	0.000306	0.001537	0.268965	0.043827	0.051354	0.050773	0.147545	0.185538
6	0.000562	0.001477	0.038343	0.018253	0.07908	0.088534	0.328674	0.182885
7	0.014912	0.00068	0.001243	0.371309	0.114922	0.140932	0.328346	0.52572
8	0.014178	0.279716	0.265897	0.414656	0.768084	0.718394	0.434119	0.501809
9	0.043973	0.184825	0.768122	0.165121	0.268245	0.31005	0.380506	0.77323
10	0.027376	0.009383	0.042855	0.350375	0.601622	0.888391	0.139526	0.910241

Table 3: Results of t-test on groups divided by pseudocode results

From our predicted behaviour of the Neo-Piagetian stages outlined at the start of the analysis section we expected to see students who did poorly in the exams displaying different behaviour in the C2C, C2R and R2C movements as more successful students moved onto preoperational reasoning stages. Higher achieving students have a consistently lower average percentage of C2C when groups are divided by lab exam results. The difference in C2C movements gets steadily larger from week 1 until week 7, when it slightly reduces. This is also true for the pseudocode results, with smaller margins of difference. The difference is smaller in the last weeks of the module, which may indicate that our students who do not do well are moving through the Neo-Piagetian stages but are not moving quickly enough for the course.

Higher achieving students have a consistently higher average percentage of C2R when divided by lab exam results. This difference peaks in week 7, for both lab exam and pseudocode results. Both groups have a similar percentage of R2C when divided by lab exam results, but the difference peaks in weeks 6 and 7, when the higher achieving students have a higher average percentage of R2C movements. This may be the point where successful students have reached preoperational reasoning, as an increase in R2C movements indicate the student is in the “tinkering” stage as described by Perkins *et al.*

Using the dataset containing the CRE percentages for each student for each week, Deep Neural Net binary classifiers were trained to classify students as being in the top 50% or the bottom 50% of grades for the lab exams. Linear Regression tests were used to compare the CRE actions and their relation to student performance in lab exams. This was used to select movement data to be used in the

Week	Average Classifier Success Rate
1	0.62
2	0.6
3	0.68
4	0.7
5	0.62
6	0.6
7	0.72
8	0.76

Table 4: Classifier results

Deep Neural Nets. Multicollinearity can be an issue for DNN, so a check was run on the features (where each movement was a feature) and we removed the most highly correlated CRE data and tested again. This was repeated until the remaining data was sufficiently nonlinearly related, when all features had a variance inflation factor (a test for correlation between independent variables) of less than 5. The resulting data set was used to train and test our DNN classifier. A classifier was built for each week of the semester, using the CRE data from that week, and from all previous weeks. The results are shown in Table 4. The results of week 9 and 10 are identical to week 8, as it uses the same CRE data after the multicollinearity tests.

5. Discussion

Other studies have referred to lab 4/week 4 (Teague) as the time around which students who are in danger of failing begin to perform badly or separate in behaviour from the other students. Of

course, what takes place at this point varies across different institutions and courses. Nonetheless we see that in line with this estimated timescale, the differing behaviour among students becomes more pronounced around lab 4, and at this point the classifier has a success of 70%. At this point, if students are consistently compiling without progressing to run, this is a sign the student is in danger. This is not hugely surprising. It implies that the student is failing to write compilable code, and we would expect that a student who cannot write compilable code would be in danger.

The percentage of R2C becomes more significant around week 4. A student who compiles code, then runs, but then goes back to compile, is most likely working on a semantic issue, rather than a syntactic one, as mentioned in the BlueJ papers (Jadud 2005, Jadud 2006). We suspect that the reason it becomes relevant to the students' overall performance in lab exam results is because week 4 is when most students should be beginning to master syntax and to abstract solutions, allowing them to construct more complex programs using multiple concepts together. The result is that we see successful students compiling successfully and rewriting their code until they reach a solution, causing successful students to have more C2R movements and fewer C2C movements. Students who are still struggling to write semantically correct code will have even more C2C movements as the assignments get more difficult.

Research Question: Can we observe signs of progression through the Neo-Piagetian stages of learning by examining passively collected data on students coding behaviour?

Yes, we have described the expected signs in CRE movements of progression through the stages of Neo-Piagetian learning, observed these signs in novice programmers and found these signs relate to student success. From our analysis section, we see that the CRE movements that are associated with success change as the semester progresses. Using a Neo-Piagetian framework, we examine these differences.

The three Neo-Piagetian stages in learning to program (Lister, du Boulay, Teague, Teague *et al.*):

(1) Sensorimotor Stage - interacting almost randomly, with little understanding of the outcome

A high percentage of C2C movements may indicate that a student is tinkering almost randomly with their code and is unable to write compilable code. From our analysis, we see that a lower amount of C2C movements, and a higher amount of C2R movements is associated with better performance in lab exams. This is similar to the findings in the paper (Vihavainen) which found a statistically significant difference between the number of runs and tests for students with previous programming experience and those without.

(2) Preoperational Stage – beginning to master syntax, deeper understanding and being able to predict behaviour from interactions

Students with a higher amount of C2R movements may be in this stage, as they become able to write compilable code, but are still be unable to predict the outcome of their code. As a result of this, the student will repeatedly “tinker” with their code, resulting in increased R2C movements. We found R2C movements became significant from week 4, indicating that students should reach this stage by week 4 if they are to be successful in the module lab exams.

(3) Concrete Operational Stage – can “chunk” (Shneiderman & Mayer) programming concepts and abstractions of the code's behaviour, allowing the programmer to write more complex code.

At this point, students should be able to write compilable code and successfully predict their code's outcome. Students at this stage should have fewer C2C movements, fewer C2R movements, and a higher percentage of R2E movements. This indicates that they have a good grasp of semantics and are able to predict code behaviour with less tinkering and playing with code. We would expect to see a higher correlation between outcome and R2E movements as students reach this stage, and while R2E is related to success at some points in the semester, the average difference between the two groups is consistently low. We strongly suspect that most students do not reach concrete operational stage until after their first semester (Teague).

6. Conclusions

We have found that C2C and C2R movements are important indicators of student performance in their first semester of programming. While the highest classifier success of 76% used data from throughout the 8 weeks, we had success with the week 4 classifier which had a success percentage of 70%, showing there is evidence of a student success or failure as early as week 4. This version of the classifier used all 4 weeks C2C percentages as the input data in predicting the student outcome. In future work, it would be worth looking at which specific assignments and topics are key clues in a student's eventual outcome.

We would expect that student coding behaviour would correlate to lab exam performance and pseudocode performance, if the coding behaviour in question indicates progress through the Neo-Piagetian stages of learning. We have seen that patterns of student behaviour contain indications from an early stage if they are likely to perform well in lab exams. We have discussed how this relates to previous work done in the area of Neo-Piagetian theory in the context of students learning to program. We have established a strong case for the connection between students' programming behaviour and their stage of Neo-Piagetian learning by showing the correlation between student CRE movements, and their lab exam outcomes, and we discussed the reasons behind those behaviours and how they relate to Neo-Piagetian theory.

Introduction to programming modules that emphasize only how to write code, and grade based primarily on written code may be problematic. Results of pseudocode assignments in a programming module can help us as researchers and educators to identify students who have not developed mental models of programming concepts, and are instead relying on "hacking", where students attempt to complete a coding assignment by writing code and testing input/output without planning and predicting their code's behaviour. Students who are "hacking" may still perform reasonably well in their weekly labs, and so may believe that they are keeping up and do not need to continue to work on their grasp of fundamental coding concepts. These students will then progress to more difficult modules without the programming basics required to engage with the material. This may be a scenario unique to computer science and a significant contributory factor as to why computer science failure rates are so high. In future work, we will examine how a novice programmer's pseudocode results and patterns of behaviour may relate to code complexity, as a reflection of their ability to "chunk" programming concepts, in order to combine them to create solutions for programming problems.

In conclusion, the most significant findings from this study are, firstly, that the divergence in behaviour between high and low achieving students takes place in week 4. Students who are not displaying signs of progression to the preoperational stage of Neo-Piagetian learning do not do well in their lab exams at the end of the semester. Secondly, we found that these differences in behaviour are less pronounced later in the semester – implying that the students who were behind in week 4 are capable of progression to preoperational stage, but crucially, not at the pace dictated by the module.

7. References

- Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & cognition*, 9(4), 422-433.
- Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *AcM SIGcSE Bulletin*, 39(2), 32-36. J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32-36, 2007.
- Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using learning analytics to understand the learning pathways of novice programmers. *Journal of the Learning Sciences*, 22(4), 564-599.
- Berland, M., & Martin, T. (2011). Clusters and patterns of novice programmers. In *The meeting of the American Educational Research Association*. New Orleans, LA.
- Bisant, D. B., & Groninger, L. (1993). Cognitive processes in software fault detection: a review and synthesis. *International Journal of Human-Computer Interaction*, 5(2), 189-206.

- du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of man-machine studies*, 14(3), 237-249.
- Castro-Wunsch, K., Ahadi, A., & Petersen, A. (2017, March). Evaluating neural networks as a method for identifying students in need of assistance. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 111-116).
- Corney, M. W., Lister, R., & Teague, D. M. (2011, January). Early relational reasoning and the novice programmer: Swapping as the “Hello World” of relational reasoning. In *Conferences in Research and Practice in Information Technology (CRPIT)* (Vol. 114, pp. 95-104). Australian Computer Society, Inc..
- Corney, M. W., Teague, D. M., & Thomas, R. N. (2010, January). Engaging students in programming. In *Conferences in Research and Practice in Information Technology*, Vol. 103. Tony Clear and John Hamer, Eds. (Vol. 103, pp. 63-72). Australian Computer Society, Inc..
- Culligan, N., & Casey, K. (2018). Building an Authentic Novice Programming Lab Environment. *Irish Conference On Engaging Pedagogy*
- Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1), 25-40.
- Jadud, M. C. (2006). An exploration of novice compilation behaviour in BlueJ (Doctoral dissertation, University of Kent).
- Lang, C., McKay, J., & Lewis, S. (2007). Seven factors that influence ICT student achievement. *ACM SIGCSE Bulletin*, 39(3), 221-225.
- Lister, R. (2011, December). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Conferences in Research and Practice in Information Technology Series*.
- Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008, September). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research* (pp. 101-112).
- Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1), 37-55.
- Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3), 219-238.
- Soloway, E., & Spohrer, J. C. (2013). *Studying the novice programmer*. Psychology Press.
- Teague, D. (2015). *Neo-Piagetian theory and the novice programmer* (Doctoral dissertation, Queensland University of Technology).
- Teague, D., Lister, R., & Ahadi, A. (2015, January). Mired in the Web: Vignettes from Charlotte and Other Novice Programmers. In *ACE* (pp. 165-174).
- Vihavainen, A., Luukkainen, M., & Ihtola, P. (2014, October). Analysis of source code snapshot granularity levels. In *Proceedings of the 15th Annual Conference on Information technology education* (pp. 21-26).
- Watson, C., & Li, F. W. (2014, June). Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 39-44).
- Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, 23(4), 383-390.

Developing Testing-First Labs For a Less Intimidating Introductory CS Experience

Angela Zavaleta Bernuy

Dept. of Computer and
Mathematical Sciences

University of Toronto Scarborough

angelazb@cs.toronto.edu

Brian Harrington

Dept. of Computer and
Mathematical Sciences

University of Toronto Scarborough

brian.harrington@utoronto.ca

Abstract

When introducing non-majors to programming in an introductory computer science course, the simple mechanics of code writing can be intimidating. Many students report feeling overwhelmed by the requirements of user interfaces and syntax guides before even writing their first line of code. In an attempt to combat this anxiety, we have developed a tool called *Code Detective*, which allows students to learn fundamental skills of computer science: testing, program description, debugging and tracing before ever having to write any code.

Code Detective starts by completely hiding the code, asking students to reverse engineer the specifications of each module from only the inputs and outputs. Over several weekly laboratory sessions, students are then introduced to program definition, documentation and testing, as more elements of the code are revealed. Students then learn tracing and debugging, all before actually being required to directly write or edit any code.

In this experience report, we discuss the development and deployment of Code Detective in an Introduction to Programming course for non-majors course at a large North American research university.

1. Introduction

Learning to program is perceived by many students as a very challenging academic task (Bennedson & Caspersen, 2007). It is a common understanding that students can get intimidated and overwhelmed when they are introduced to programming, especially if they have a fixed idea on their minds about the levels of difficulty of the subject.

Reports show that there are high failure rates when learning to program, and many students choose not to take computer science courses because they find the concept of programming to be intimidating (O'Donnell, Buckley, Mahdi, Nelson, & English, 2015). Educators have been trying to improve the overall students' satisfaction as it is connected with students' retention (Rybarczyk, 2020). As non-majors have different learning habits from traditional computer science students (Rybarczyk, 2020), increasing engagement during a non-majors class is always a challenge (O'Donnell et al., 2015).

Previous work about other strategies to introduce programming includes educational games, breadth-first approaches, testing first, among others. Educational games have been used to increase students' engagement and retention (Lee, Ko, & Kwan, 2013). Another approach is breadth-first, which includes teaching students everyday computer tasks like image editing, OS installation and building home computer networks in an undergraduate course (McFall & DeJongh, 2011). There has been work done in testing-first approaches to introductory programming. Marrero Settle conducted a course where students were required to implement their test cases before completing their assignments (Marrero & Settle, 2005).

This work explores an alternative way of introducing programming to non-major students that combines some elements of breadth-first and testing-first approaches, utilizing more traditional deductive reasoning skills in place of technical abilities that may be new and intimidating to novice programmers. The focus of this work is to present programming as a deductive logical process first, allowing students to think algorithmically, and become comfortable with fundamental concepts of programs and functions, before presenting them with actual code. In this way, students can begin by using tools and methods with

which they are already familiar while they are gradually introduced to more specific computer science concepts.

2. Code Detective

Code Detective is a tool that was designed as part of an Introduction to Programming course for non-major students at The University of Toronto Scarborough. The web tool was developed by a team of undergraduate computer science students. Code detective focuses on introducing non-major students to the concept of computing in a way that emphasizes skills they already possess, without forcing them to write (or initially, even read) code that they may find overwhelming. The programming languages taught in the course were Scratch and Python, and this tool was easily adaptable to both languages.

Code Detective consists of nine modules that are aligned with the material covered during the weekly lectures starting on the second week of the semester. Students worked in pairs on the Code Detective modules during two-hour weekly laboratory sessions supervised by teaching assistants. The teaching assistants were mainly tasked with providing guidance to groups as needed. At the end of each session, students were asked to present their solutions and explain their reasoning.

Each module consists of a series of questions about various programs with simple logic. In the first modules, the program's code is entirely hidden, only offering input and output on the screen where the students are asked to experiment with the program and deduce what it does, formally define the program's function, and develop a testing plan to determine if the program has any bugs. For later modules, students practice how to trace and repair code, without the need for writing any code. Only in the later modules are students asked to write or edit code.

The nine modules were created following a gentle, yet increasingly difficult pedagogy as follows:

2.1. Lab 0: Program definitions

Consists of twenty questions. Each question has a small program with a series of input/output boxes (check-boxes, text boxes, date selectors). The students need to provide some input, click the "run" button and keep track of the response. After experimenting with the program, they are asked to provide a formal definition for the function which includes: stating what the function does, valid input and expected output.

2.2. Lab 1: Program definition and introduction to algorithms

Consists of two parts, five questions each. For the first five questions, the students are given the definition of a program and a high level algorithm that would implement the definition in a flawed way. Their goal is to find the flaws in the algorithm by providing a list of test cases that will fail and write the fixed algorithm. For the last five questions, the students only get the definition of a program and they have to provide an algorithm to solve the problem.

2.3. Lab 2: Test dimensions and black-box testing

Consists of two parts, five questions each. For the first five questions, the students were provided with the definition of a program and are tasked with designing a testing plan. They need to provide the dimensions of the testing space, decide which are the important points on each dimension, and calculate the number of tests required for a full coverage testing. For the last five questions, they need to perform black-box testing of a given program and provide a list of failed test cases specifying the input, expected output, and actual output.

2.4. Lab 3: Tracing and white-box testing

Consists of two parts, five questions each. For the first five questions, the students are provided with a simple algorithm. Their task is to trace the code for a given input and enter the output. The students get immediate feedback from Code Detective and a live-count of failed attempts while entering their answers. For the last five questions, they need to perform white-box testing of a given Scratch program and provide a list of failed test cases specifying the input, expected output, and actual output.

2.5. Lab 4: Tracing and debugging

Consists of two parts, five questions each. For the first five questions, the students are provided with more complex code than the previous module as loops are introduced. Their task is to trace the code for a given input and enter the output. The students get immediate feedback from Code Detective and a live-count of failed attempts while entering their answers. For the last five questions, the students are given a broken program and their task is to debug and fix the code.

2.6. Lab 5: Refactoring and implementation

Consists of two parts, five questions in total. For the first three questions, the students get a working program that is not well designed. Their task is to refactor it by creating smaller, better designed modules without breaking the code. For the last two questions, the students get an incomplete program with missing code segments (missing blocks in Scratch). Their task is to implement the missing components to get the code working.

2.7. Lab 6: Efficiency, and implementation

Consists of two parts, five questions in total. For the first three questions, the students get a working program implemented inefficiently. Their task is to refactor the program without changing its functionality. For the last two questions, the students get the definition of a program with a set of specifications that they are required to implement.

2.8. Lab 7: Scratch-Python translation

Consists of five questions. Each question has five different Scratch working programs. The students need to translate the Scratch code into Python code. The students are required to test their translated code using an IDE and demonstrate their testing plan. This module is specifically designed for courses that cover more than one language in order to help students learn to work with a new language, but can also be used to help students understand how much of their learning is transferable to other languages.

2.9. Lab 8: Design and implementation

The students are provided with a partially completed Python code. They are required to read and understand the code, as well as to implement the missing documented functions.

Following the completion of the Code Detective modules, the students are required to work on a project of their choice where they had to implement a program either using Scratch or Python. Some of the most common projects were arcade games in Scratch and simple data management programs in Python.

3. Evaluations

Based on the anonymous course evaluations, some students shared that they enjoyed this course design because they did not have any prior coding experience and they were gradually exposed to the course content. Moreover, one student stated that *"this was good because it prevented me from getting scared off from programming forever"*. A couple of students shared that the structure of the course helped reduce intimidation as it helped them *"let go of the mindset that computer science is intimidating and instead makes us see that computer science can be for everybody"*, and that we created *"a learning atmosphere that was not intimidating as a person with no experience at coding!"*.

Many students stated that the course was still challenging and required time and effort to develop a deeper understanding of programming concepts. They agreed that Code Detective helped reinforce the lecture material and encouraged them to get more practice. On the other hand, students who had prior coding experience reported that they felt the class progress slow and they wished they were exposed to harder concepts.

To measure the success of Code Detective compared to previous deliveries of the same course, we looked at the drop rates from previous years. We found that the year in question had a 7% drop rate, around 23% lower than the previous years: 29% and 30% in the two years prior, even though the class size of 450 students was the same across the three years. While we cannot attribute this substantial reduction in drop rates to Code Detective directly, as other factors such as teaching staff and structure were not held

constant, the teaching team commented on the absence of the phenomenon, observed in previous course offerings, of students dropping after being unable to complete the first lab sessions.

4. Conclusion

Code detective played a key role in reducing students' sense of intimidation, and fostered a sense of accomplishment without resorting to paternalistic methodologies or games that may alienate some students. Introducing computer science as a deductive, logical, problem solving system first before introducing the technicalities of code writing made students feel that they were able to use the logic and reasoning skills they already possessed to solve problems, and gave them a gentler and less intimidating introduction to programming.

5. References

- Bennedsen, J., & Caspersen, M. E. (2007, June). Failure rates in introductory programming. *SIGCSE Bull.*, 39(2), 32–36. doi: 10.1145/1272848.1272879
- Lee, M. J., Ko, A. J., & Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. In *Proceedings of the ninth annual international acm conference on international computing education research* (pp. 153–160).
- Marrero, W., & Settle, A. (2005). Testing first: emphasizing testing in early programming courses. In *Proceedings of the 10th annual sigcse conference on innovation and technology in computer science education* (pp. 4–8).
- McFall, R. L., & DeJongh, M. (2011). Increasing engagement and enrollment in breadth-first introductory courses using authentic computing tasks. In *Proceedings of the 42nd acm technical symposium on computer science education* (pp. 429–434).
- O'Donnell, C., Buckley, J., Mahdi, A., Nelson, J., & English, M. (2015). Evaluating pair-programming for non-computer science major students. New York, NY, USA: Association for Computing Machinery.
- Rybarczyk, R. (2020). Non-major peer mentoring for cs1. In *Proceedings of the 51st acm technical symposium on computer science education* (p. 1068–1074). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3328778.3366901

6. Appendix: Code Detective Interface

Figure 1 – Code Detective Main Menu

All Labs

Lab 0

Black-box testing.



Lab 1

Fixing and creating algorithms.



Lab 2

Test planning and black-box adversarial testing.



Lab 3

Code tracing and white-box adversarial testing.



Figure 2 – Lab 0 Menu

Lab 0

Question 1

No Answer

Go

Question 2

No Answer

Go

Question 3

No Answer

Go

Figure 3 – Lab 0 Sample Question

Question 3

Value: 16

Output: true

Enter your answer:

Type your response here...

Save & Back to Question List

Figure 4 – Lab 1 Sample Question 1

Question

Input: Person, Driver's License, Car
Output: Person, Car

Check if a person is allowed to drive based on their driving license specifications.

```
If the person has a valid driver's license:
    let them drive
If the person needs glasses to drive and is not wearing any:
    do not let them drive
If the person has a learner's license:
    if the person is with an authorized driver:
        do not let them drive
    else:
        let them drive
```

Your Answer

Explain below why the algorithm on the left doesn't work

Write a fixed algorithm below

Submit

Figure 5 – Lab 1 Sample Question 2

Question

Input: Person, Word, Dictionary
Output: Page number

Find the given word in a dictionary.

Your Answer

Write an algorithm below

Submit

Figure 6 – Lab 2 Sample Question 1

Question

Inputs

age (integers 1-100), has_id (boolean)

Outputs

can_vote (boolean)

Expected Behaviour

Return True if and only if a person is allowed to vote (18 or older and has an ID)

Your Answer

```
# How will you divide the tests into multiple areas?

# How many tests do you need to thoroughly test this program?
Describe them.
```

Submit

Figure 7 – Lab 2 Sample Question 2

Question

Input

num_slices (integer >= 0), num_people(integer >= 0)

Output

slices per people

Behaviour

Return the number of slices per person given the number of people who want pizza and the slices available, a person can only get full slices. If there is not enough pizza, return: "Not enough pizza!"

Test Cases

Type your test cases below, the algorithm will automatically execute while you type.

Yellow background denotes a failed test case.

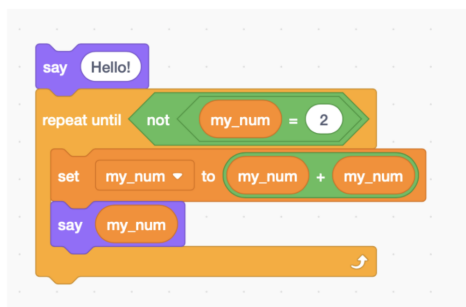
Input	Expected Output	Actual Output	
6, 2	3	3	Remove
2, 3	Not enough pizz:	Not enough pizz	Remove

Add Test Case

Submit

Figure 8 – Lab 3 Sample Question 1A

Question 1



Trace the Program

Part 1 Part 2

Let my_num = 1. Enter what Scratch will say, in the order in what it will be said.

Hello!

You have completed the part!

Save & Go Back

Figure 9 – Lab 3 Sample Question 1B

Question 1

```

say Hello!
repeat until not my_num = 2
  set my_num to my_num + my_num
  say my_num
  
```

Trace the Program

Part 1 Part 2

Total failed attempts 1

Let my_num = 2. Enter what Scratch will say, in the order in what it will be said.

That was not quite right... try again.

Hello

Figure 10 – Lab 3 Sample Question 2

Question 7

Check if a number is divisible by 6. Please enter your inputs in the following order, if there are multiple inputs, separate them using comma:

- my_num - integer

```

if my_num mod 3 = 0 then
  say No
else
  if my_num mod 2 = 0 then
    say Yes
  say No
  
```

Test Cases

Please separate multiple inputs using comma.

Input	Expected Output	Actual Output		
3	No	No	<input type="button" value="x"/>	<input checked="" type="checkbox"/>
4	Yes	Yes	<input type="button" value="x"/>	<input checked="" type="checkbox"/>
7	No	No	<input type="button" value="x"/>	<input checked="" type="checkbox"/>

Figure 11 – Lab 4 Sample Question

Question 1

```

repeat 3
  say my_num for 1 seconds
  change my_num by 2
repeat until my_num mod 3 = 2
  change my_num by 1
  say my_num for 1 seconds
  
```

Trace the Program

Part 1 Part 2

Total failed attempts 5

Let my_num = 4. Enter what Scratch will say, in the order in what it will be said.

4

6

That was not quite right... try again.

7

Figure 12 – Lab 5 Sample Question

Question 2

Carefully read the description below, and fix the program on Scratch. When you open the project, click on "Remix" to copy the project into your account and fix the changes there.

Implement the block perimeter. Hint: Try using the instruction "item # of ___ in shapes"

- Input: shape (triangle, square, pentagon, hexagon, heptagon, or octagon)
- Output: perimeter

[Go to Scratch](#)

Figure 13 – Lab 6 Sample Question

Question 2

Carefully read the description below and complete the question on Scratch. When you open the project, click on "Remix" to copy the project into your account and fix the changes there.

Task: Improve the code without changing the actual functionality.

- Input: word (string)
- Output: has_digits, has_symbols, has_vowels

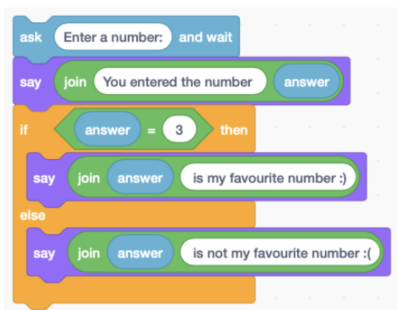
Check if the input has at least one digit, one symbol or one vowel. If it has any of these, change the appropriate variable to "Yes", if not, leave it as "No"

[Go to Scratch](#)

Figure 14 – Lab 7 Sample Question

Question 1

Read through the Scratch code below and re-write it in Python. You may use any editor you want, after you have completed the task, paste your Python code below.



Enter your Python code here...

[Submit & Go Back](#)

Figure 15 – Lab 8 Sample Question

Lab 8 Problem

In this lab, you will learn how to use Python to read/write CSV files. You can download the starter files below, once you have completed the lab, paste your completed lab within the textbox below and submit.

Starter Files

[lab8.py](#)[store_items.csv](#)

Submission

Enter your Python code here...

[Submit & Go Back](#)

A principled approach to the development of drum improvisation skills through interaction with a conversational agent

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

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

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

Abstract

Shedding is a term used to describe a musical conversation between drummers with the aim to improve their drumming vocabulary, gain confidence in real-time trading of musical ideas, develop an understanding for their original voice on the drum kit and enjoy the process of exploring creativity with a fellow drummer. However, in practice drummers have limited opportunities to play in real time with other drummers. This research explores shedding activity in the form of mixed-initiative interaction between a human drummer and a conversational agent. This paper focuses on a series of design studies and experiments to explore three novel refinements to the proposed shedding model.

1. Introduction

The proposed agent embodies an inference system allowing it to navigate through transformations of a *core phrase* chosen by the user (the starting point of every shedding interaction in our model) with a design aim of conversing with the human drummer in a way that is perceived as meaningful, musical and inspiring. The transformations involve the agent taking a core phrase and adapting it in various ways, for example, by making changes to elements such as orchestration, metric modulation and phase shift. These elements offer dimensions of development in linear drumming, where a range of transformations of each element can be explored by the agent and human drummer. This research focuses on creating a reflective drumming agent that inspires the user by having a conversation rather than by teaching specific grooves (Senn, 2018) or drumming concepts. This paper focuses in particular on how this central shedding model can be enhanced and deepened based on a novel characterisation of rhythmic grouping and accent patterns.

2. Previous work

Previous research has investigated creativity from a number of perspectives including computer modelling (Boden 1994, Cope 2005), communication (Davidson 2005) and perfection (Berger 1999). However, creativity in performance has received relatively little attention (Pinheiro 2010). More specifically, research into musical interaction activities with intelligent systems such as the *Continuator* (Pachet, 2003), *Controlling Interactive Music* (Brown, 2018) and *Monterey Mirror* (Manaris et al. 2018) present tools for contemporary music creation and co-creativity. However, our present work suggests a musical framework with a reflective agent that aims to elicit creativity by encouraging the human drummer to observe and refine their creative process.

3. Adding depth to the shedding model

The inference system employed by the agent uses the core phrase and rules of transformations in order to converse with the human drummer in ways that are perceived as meaningful. Following several design prototyping studies using Wizard of Oz, we were able to refine the initial transformation model with three elements: *linear drumming*, *grouping* and *external inspiration for core phrases*.

3.1. Linear drumming

Linear drumming is a monophonic drumming playing style, where drum instruments are hit exclusively one at a time. Combining the conversational shedding model with the linear concept amplifies the clarity of the system responses, making the transformations of the core phrase stand out for the human drummer. Moreover, linear drumming is very common in live shedding interactions and therefore adds a layer of stylistic authenticity to our model. The relative ease of monophonic drumming, as opposed to polyphonic, allows us to refine our design further by exploring one-bar drumming phrases that alternate unambiguously between *accented* and *unaccented* notes. The relationships between these two *types* of notes create clear grouping relationships that reflect the shape and conversational tone of the drummer.

3.2. Grouping

Although the idea of grouping as a recursive segmentation of musical phrases occurs in Jackendoff (2009), in this paper we propose an alternative characterisation of groupings based on accent patterns in the context of linear drumming. We explore two-bar system transformations that take advantage of this characterisation with the aim to promote thematic unity in the shedding process. Under this characterisation we analysed the accent patterns to identify groupings using the following procedure: i) the first accented note marks the beginning of a phrase and is therefore internally annotated with the number '1'. ii) each subsequent unaccented, or crucially, consecutive accented note is then numbered serially '2', '3', '4' etc. iii) new sections (marked with the number '1') start with each new accented note that follows an unaccented one. We posit that the shedding interaction will be perceived as more meaningful and inspiring to the human drummer if the system's transformations are sensitive to the accent patterns that emerge from the grouping characterisation.

3.3. External inspiration for core phrases

With the aim to make the shedding interaction more inspiring and promote learner's autonomy (Green, 2002), we have explored a model where the agent and human drummer can utilise the similarities in the hierarchical metrical grid (Jackendoff, 2009) between drumming and 30 second extracts from spoken art forms such as rap, speech and poetry. For example, rhythmic content can be borrowed from a rhythmic speech art form, transformed into one or more core phrases and used for shedding with the conversational agent. Our experiments drew on a wide variety of creative sources: i) a speech by Nelson Mandela ii) a song by Grime artist Stormzy iii) a freestyle rap by the artist Mos Def. We conclude that further research into the hierarchical metrical grid used in spoken art forms such as rap and poetry may offer creative new ways for learning stylistic improvised interaction through a musical instrument, such as the drum kit.

4. Conclusions

- Shedding is a promising basis for mixed-initiative interactions between a human drummer and a conversational agent.
- Early explorations have presented the potential of such interactions to improve drumming vocabulary, promote confidence in real-time trading of musical ideas, foster originality and promote exploration of creativity.
- Relationships between the accented and unaccented notes in linear drumming create clear grouping relationships that reflect the shape and conversational tone of the drummer.
- Shedding interactions will be perceived as more meaningful and inspiring to drummers if the system's transformations are sensitive to the grouping of accent patterns.
- Further research into the hierarchical metrical grid used in spoken art forms such as rap and poetry may offer transformative ways for learning stylistic improvised interaction through a musical instrument, such as the drum kit.

5. References

- Brown, A. R. (2018). Creative improvisation with a reflexive musical bot. *Digital Creativity*, 29(1), 5-18.
- Chester, G. (2006). *The New Breed: Systems for the Development of your own creativity*. Hal Leonard Corporation.
- Greb, B. (2008). The language of drumming.
- Green, L. (2002). *How popular musicians learn: A way ahead for music education*. Ashgate Publishing, Ltd..
- Jackendoff, R. (2009). Parallels and nonparallels between language and music. *Music perception*, 26(3), 195-204.
- Lerdahl, F. (2009). Genesis and architecture of the GTTM project. *Music perception*, 26(3), 187-194.
- Manaris, B., Hughes, D., & Vassilandonakis, Y. (2011, June). Monterey mirror: combining Markov models, genetic algorithms, and power laws. In *Proceedings of the IEEE Conference on Evolutionary Computation*.
- Pachet, F. (2003). The continuator: Musical interaction with style. *Journal of New Music Research*, 32(3), 333-341.
- Pinheiro, R. (2010) 'The creative process in the context of jazz jam sessions'. *Journal of Music and Dance*. 1(1), pp. 1-5. Available at: http://www.academicjournals.org/journal/JMD/edition/January_2011 (Accessed 04.06.2014).
- Senn, O., Kilchenmann, L., Bechtold, T., & Hoesl, F. (2018). Groove in drum patterns as a function of both rhythmic properties and listeners' attitudes. *PloS one*, 13(6), e0199604.

Programming “systems” deserve a theory too!

Joel Jakubovic
School of Computing
University of Kent
Joel.Jakubovic@kent.ac.uk

Abstract

It is comparatively easy to find new *languages* and derivative work within academic publishing, but somewhat harder for more general programming “systems” or “environments” which encompass more than that. This is a shame, since some of these have a dedicated following and have influenced programming and computing at large. We concentrate on some examples of such that, for various reasons, do not have much material written about them. We suggest some reasons to expect this, and draw attention to some characteristics (both “cognitive” and “technical”) marking them worthy of further study. We conclude with a sketch of further steps to make the most of these software artefacts from a research perspective.

1. Introduction

Programming language theory, implementation and evaluation is an expansive and well-established field. Even entire applications, “environments” or “systems”, developed *for the purposes of research*, contribute to progress by being discussed and evaluated within a certain academic scope (for example, HCI). However, there exist innovative, influential or otherwise noteworthy systems that nevertheless fall outside of these realms. As we expand on later, this is in part due to originating outside of the academic environment, and it is also pushed against by the nature of the publishing medium. We will lead in with three such systems to illustrate our later points.

1.1. Smalltalk

Smalltalk could be described as a self-sufficient desktop environment programmed in a language of the same name. It was influential on the rise of object-orientation and graphical user interfaces, and was itself intended to be widely embraced. However, it did not manage to achieve this goal, and lives on in its own niche on a par with other programming language ecosystems.

Whenever a language or IDE feature spreads across the mainstream, Smalltalkers have a reputation for pointing out that Smalltalk was able to do the same thing back in the ’80s. This is a plausible claim, owing to the deliberate high-level, flexible and totalising architecture of the system. Noting its following by a community of devotees, this is evidence of something special and valuable to learn from and build upon. However, despite having a literature presence in virtual-machine optimisation and related *implementation* technologies, insight into Smalltalk’s *design* seems to be mostly scattered around historical magazine articles (*BYTE Magazine Volume 6, Number 8, 1981*), web pages and blog posts.

Without a well-structured design discussion, it is only clear that there is “something about Smalltalk” that is worth improving on. In other words, it is difficult to distinguish which specific aspects of Smalltalk are essential to its value and which are incidental. We can identify characteristics such as *meta-circularity* and *self-sufficiency*, by which the software within lives in a world where “everything is Smalltalk”. Another identifying feature is that instead of transient “applications” and manual saving to “files”, the entire system state is persisted as a continuously evolving “image”. Smalltalk also embeds its programming language within a larger prescribed context of graphical interface and device utilities, encompassing the same roles (and scope) of an entire Operating System. Such prescription of a graphical interface is characteristic of what we are referring to as “systems”, but this last “encompass the world” aspect is more unusual.

1.2. HyperCard

HyperCard was a platform for exploring and creating interlinked sets of multimedia pages, often regarded as a precursor to the Web. Explicitly designed with the goal of end-user empowerment, with a slogan of “programming for the rest of us”, HyperCard was a popular platform both to share and to author among teachers, businesspeople, and other non-programmers. This was no doubt in part due to its (initial) default inclusion in Macs. However, the existence of other default apps (such as Terminal) *not* enthusiastically adopted by end-users means that HyperCard itself still stands on its own merit.

HyperCard applications were organised into “stacks”, with typical UI elements (text, pictures, buttons) added and edited by direct manipulation. Every such element could have a script attached to it in HyperTalk, a language designed to resemble English. HyperCard also distinguished itself in having separate “beginner” / “advanced” layers, allowing novices to make their way through straightforwardly *using* and maybe editing stacks superficially, before digging into more advanced authoring and scripting.

Unfortunately, despite its popularity and originality, HyperCard fell out of use due to corporate and marketing decisions and subsequent declining technical support, eventually falling out of compatibility with newer versions of the Mac OS. As HyperCard was primarily a *commercial* product, not much has been written about its design, but its popularity suggests that it is worth studying and possibly adapting for future software systems. Sample design characteristics for HyperCard include its layered design allowing progression from *using* to *developing* card stacks, and the fact that such developing is not considered something *far away* but instead occurs in the same user interface.

1.3. Flash

Adobe Flash was a widely popular browser plugin providing “rich” content (video, audio, and games) for the primitive early Web. Such applications were created using Flash Builder: an advanced graphical workspace for various forms of vector graphics and animation, scripted with an event-driven language called ActionScript.

Flash gave rise to an explosion of animations, games and websites across the early Web, often by individuals and within online communities. Because it was used so widely for websites and games, Flash can be considered a success case, to some extent, of “end-user” software development. Unfortunately, Flash was thought to be too insecure and poorly-optimised for the emerging mobile platforms of the 2010s. Apple’s decision not to support Flash on the iPhone ultimately spelled its decline and planned official demise at the end of 2020.

Following from our previous two examples, what are the interesting design characteristics of Flash? Its success may be attributed to the fact that it merely *filled a gap* and achieved rapid ubiquity as a result, even defining a new *medium* of interactive Web content. This is certainly true, but suggests this role could have been filled by any other provider of rich content support. Flash Builder, though, does seem to be an effectively designed tool for its audience. It is a graphical editor centering around a WYSIWYG “stage” metaphor, surrounded by many specialised sub-interfaces (for example, managing the broad and narrow details of keyframe animation), and with an attached scripting sub-editor. Are there more specific design principles hiding here than merely “use the right interface for the job”?

2. What to do?

2.1. Rational Reconstruction

The main point of focusing on such systems is that they exist outside of the academic literature, having been commercial or educational ventures rather than research artefacts. This means that theoretical analysis or design rationale, if it exists at all, may have never been published or be scattered around documents for internal use. In some cases, there may not be any such written material or what existed has been lost.

This means that it is currently hard to do rigorous follow-up work on these artefacts. If we wish to build on their successes and learn from their failures, in a scholarly way, there needs to be more than the artefact itself and documentation. They need to be made “legible” to academic literature by *reconstructing*

what their design and theoretical analysis *would* have been in such a counterfactual world.

2.2. Cognitive and Technical Dimensions

One component of such a “rational reconstruction” could be an analysis in terms of a common vocabulary of named characteristics. For example, the Cognitive Dimensions of Notation framework (Green & Petre, 1996) suggests a number of “dimensions” along which a user interface or “notation” may be measured, along with linkages and tradeoffs between different dimensions. This is certainly useful, but its emphasis on the *cognitive* aspect of *interfaces* might leave uncharted other aspects – what we suggest to be *technical* dimensions – of the “rest” of a software system.

For example, a programming language in which “code” is expressed in the same form as “data” is said to exhibit “homo-iconicity”. We could extend this to systems-in-general, perhaps, as a measure of how much *automating* a task resembles manually performing it. Similarly, “self-sufficiency” measures the extent to which a system can be improved and evolved from within, without relying on external tools. Smalltalk was designed to be quite high on this measure (intended to “obsolete itself”), while HyperCard and Flash (along with most other software) facilitate creation of *separate* artefacts (HyperCard stacks, Flash applications) and can only be changed via in-application preferences or by editing their source code.

Our desire for “technical” dimensions comes from a feeling that there are also properties of “the system itself” rather than merely its *interface*. If *cognitive* dimensions are about how a system is *received* or *experienced* by users, then *technical* dimensions are about how the system was *intended* or *designed*. However, this does rest on an assumption that separating the “interface” from the “system itself” is a meaningful or useful thing in the first place, and we invite further discussion of this.

Whatever the details, such a set of technical dimensions would introduce a common language for comparisons between systems, which is currently lacking. It would give systems research a shot at “characterising the design space” of possible systems (Church & Marasoiu, 2019) and enable the identification of currently unexplored gaps in that space.

2.3. The Medium is the Message

Another important thing to note is that academic publishing is by and large a paper (virtual or physical) medium; as such, the ease with which one can publish varies according to the medium of the subject matter. Programming languages, being strings of characters equipped with syntax and semantics, lend themselves very well to describing in a paper. However, our systems of interest is that tend to be graphical, interactive *systems* rather than *languages* alone. They define much more of a given computing environment than languages; they additionally state how they are graphically presented and manipulated. So, in order to properly present or discuss these systems in full, one needs to at least provide pictures and ideally provide a running version.

While videos or runnable artefacts can be part of academic submissions, they are still seen as “supplementary material” to accompany a written paper. It is plausible that this makes it less attractive to present general “systems” in academic publishing, and the non-academic nature of our examples is easier to understand in light of this. This is recognised in (Edwards, Kell, Petricek, & Church, 2019), which begins a conversation about how format, submission and peer-review ought to look for such interactive systems and work on their *design*. Such a model would support a “gallery of interactions” with (possibly simplified) re-creations of systems as one possible corrective.

3. Conclusions

In summary, there exist interesting holistic “systems” which seem to have been ignored in published research. To be able to do follow-up work on them, we need a deeper and more rigorous understanding. But their typical commercial or educational nature means there was no initial body of literature to seed such a process. Additionally, the (virtual) paper medium is not well suited to “systems” in the first place. We propose to re-create the design analysis in light of innovations like the Cognitive Dimensions framework, and taking advantage of more appropriate media like video or interactive essays. A Smalltalker

might say that “everything was already invented in the 1970s”; we agree that there was little explicit building upon the visionary work of the past. To remedy this, we need to take a fresh, academically rigorous look at such old work on programming systems.

4. References

- Byte magazine volume 6, number 8*. (1981). Retrieved from <https://archive.org/details/byte-magazine-1981-08/>
- Church, L., & Marasoiu, M. (2019). What can we learn from systems? In *Proceedings of the conference companion of the 3rd international conference on art, science, and engineering of programming*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3328433.3328460> doi: 10.1145/3328433.3328460
- Edwards, J., Kell, S., Petricek, T., & Church, L. (2019). Evaluating programming systems design. In *Proceedings of 30th annual workshop of psychology of programming interest group*.
- Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework. *JOURNAL OF VISUAL LANGUAGES AND COMPUTING*, 7, 131–174.

Understanding the Problem of API Usability and Correctness Misalignment

Tao Dong
Google LLC
Mountain View, CA, USA
taodong@google.com

Elizabeth F. Churchill
Google LLC
Mountain View, CA, USA
echurchill@google.com

Abstract

When developers have more than one API they could potentially use to solve a programming problem, it's often natural for them to start with the easier and familiar, often default, option. Yet, for some tasks, such as manipulating text in the presence of grapheme clusters (e.g., `g̃` and `ᄃᆞ`), the easier API could produce less correct and reliable results. We sought to measure the impact of such misalignment between API usability and correctness. Specifically, we conducted a controlled experiment which shows that user education has a limited effect on helping the programmer choose the appropriate API, when it's not the default and error cases are difficult to imagine. We discuss a few things programming language and SDK designers can consider in order to mitigate the impact of such misalignments.

1. Introduction

API usability researchers (Myers & Stylos, 2016) have shown that poor usability of APIs can lead to flaws in computer programs, and sometimes serious security flaws (Fahl et al., 2013). However, when there are multiple APIs the programmer can choose from to solve a given problem, using the most usable API does not necessarily lead to the expected outcome. What if the easier to use API is in fact more likely to produce incorrect or unreliable results under some circumstances? We call this problem “API usability and correctness misalignment.” We believe this is a challenge that has been under-examined by API usability researchers.

One of the areas where such a misalignment problem has manifested itself is text manipulation, specifically handling grapheme clusters in unicode strings. In many programming languages, text is often represented as a sequence of UTF-16 code units. However, some user-perceived characters need to be backed by two or more UTF-16 code units. In fact, this is quite common once you need anything beyond the ASCII character table. For example, the letter `g̃` is composed of a base character `g` and a combining mark `̃`, the Hangul syllable `ᄃᆞ` may be represented by a sequence of conjoining jamos `ᄃ`, `ᆞ`, and `ᆫ`, and the emoji `👉` is encoded as a *surrogate pair* (Microsoft, 2018) of two UTF-16 code units 55357 and 56395. In the Unicode Standard, such user-perceived characters are formally known as grapheme clusters (Davis & Chapman, 2020).

Grapheme clusters can easily break text manipulation code written without anticipating their presence, because the default String APIs in most popular programming languages are not aware of them, and they instead operate on fixed-length code units such as UTF-16. For instance, the following line of code in Dart will produce an unexpected result:

```
print('Hello👉, world!'.substring(0,6));
```

The programmer's intent here is to print the substring “Hello👉.” However, the end index “6” (exclusive) will include “Hello” and then the first half of the surrogate pair representing the 👉 emoji. As a result, this line of code will return a substring with a malformed trailing character: “Hello👉.” While this particular example is from the Dart programming language, the problem is shared by many others. One notable exception is Swift, a more recently launched programming language, first released in 2014.

For mature programming languages such as Java, Python, and Dart, a common approach to adding support for correctly processing grapheme clusters is providing an auxiliary grapheme-aware text manipulation library while keeping the default String API intact for backwards compatibility and performance reasons. Nonetheless, this seemingly practical and sensible solution also leads to a

misalignment between usability and correctness, forcing the programmer to choose between multiple text manipulation APIs with hard to test real-world implications. To languages that have taken this path to supporting grapheme clusters, the easiest to use text manipulation API is often the built-in String API due to its availability, familiarity, and accumulated documentation and community knowledge. However, it's often the less correct API to use, especially when the programmer cannot limit the kinds of characters included in the text they manipulate, as it's often the case when the text is originated from user input.

To understand the severity of such misalignment between usability and correctness in making API choices, we conducted a web-based controlled experiment in the context of text manipulation in the Dart programming language. Dart recently added a package called *characters* to support grapheme clusters while keeping its UTF16-based String API unchanged. In our experiment, participants were asked to examine a number of code snippets, written with Dart's String API, for common text manipulation scenarios and determine whether each snippet would produce the expected results. Our analysis of the experiment results led to three main findings:

- Prior exposure to the grapheme clusters problem did not help participants identify it in text manipulation code.
- Participants in the two treatment groups who received information about the grapheme clusters problem and the *characters* package early in the experiment, showed a significant improvement in their ability to make correct assessments of text manipulation code over those in the control group who did not receive such information.
- Nonetheless, more than half of the participants who received an explanation about the grapheme clusters problem still failed to apply that knowledge to assessing code snippets.

The results demonstrate a substantial limit of user education in both the durability and strength of its effect, when the usability and the correctness of API choices are misaligned and no immediate feedback could be provided to the programmer. Our findings suggest that modernizing the default String API to support grapheme clusters should be seriously considered by programming language designers, especially when the programming language is not yet widely used and can afford making breaking changes to the String API.

Nonetheless, when overhauling the default String API is infeasible, language and SDK designers should mitigate the usability and correctness misalignment by facilitating the programmer's choice making process. There are a few ways to achieve it beyond user education:

- Making the alternate API more readily available and frequently visible to the programmer through a tight integration with the programming environment.
- Making the alternate API a local default through proactive suggestions and warnings in coding contexts where the risk of choosing the wrong API is high.
- Helping programmers write test cases that cover edge cases where the alternate API should be used rather than the default API.

The rest of the paper is organized as follows. First, we review related work on API usability and choice architecture. Next, we go over the design of the experiment and main findings from analyzing the results. Last, we discuss possible explanations for the experiment results and the implications for programming environment design.

2. Related Work

We contextualize our work at the intersection of API usability and the psychology of making choices. Due to space limits, we describe most relevant work in these two areas and explain the intellectual gap our research addresses.

2.1. API Usability

The importance of API usability has started gaining recognition in the industry after more than a decade of advocacy by API usability researchers (Clarke, 2004). Myers and Stylos (2016) argue that unusable APIs can lead to bugs, performance degradation, and even significant security problems. They suggest adopting the human-centered design approach in the process of designing APIs. This

approach calls for evaluation of API design in accordance with various usability principles such as those adapted from Nielsen’s “heuristic evaluation” guidelines (Nielsen, 1994) and cognitive dimensions of notations (Blackwell et al., 2001).

Nonetheless, the challenge of making API choices is under-examined. The main paradigm in API usability research has been focused on the design of a single API or a single set of APIs intended to be used together. The common approach of API usability research usually involves observing how programmers learn and use the API in question, identifying user experience problems, and making recommendations on how to improve the API’s design and documentation (for example, Piccioni et al. (2013)). Some studies compared multiple designs of an API in order to select a more usable one (Stylos et al., 2006). In a rare instance, researchers paid attention to the problem of choosing from multiple APIs for the same purpose and found that the presence of two APIs with similar names caused widespread programmer confusion (Murphy-Hill et al., 2018).

Few studies have examined the programmer’s ability to make API choices, especially when those choices involve subtle trade-offs between usability and other goals of API design, such as correctness, backwards compatibility, and computational efficiency (Stylos & Myers, 2007). The prevailing paradigm in API usability research often implicitly considers those other goals as variables independent from the API’s usability characteristics. While this is a useful simplification, it does not sufficiently recognize the inherent tensions and dependencies between some of those API design goals. For example, a high-level API is often more usable, but it could lack flexibility or coverage for edge cases. How well can programmers understand and evaluate such tradeoffs? And how can API and tooling designers provide appropriate signifiers and constraints to support the programmer in making API choice decisions? We believe it is important to extend the focus of API research from a single solution to all solutions the programmer can choose from in order to satisfy a programming requirement. This shift of focal point is critical to address the complexity of evolving APIs already widely used in production.

2.2. Choice Architecture

There has been much research in the area of human decision-making and option selection in the face of alternatives. This domain of investigation is explicitly about designing choices, hence the name “choice architecture design,” coined by Thaler and Sunstein (2009). Our particular interest is in the ways in which API choices are presented to programmers, and whether it is possible to understand the ways in which defaults explicitly presented or tacitly assumed can lead to poor choices that have negative downstream consequences for users of the program in question.

It is well known from studies of choice architectures and user interface design that people usually stick with defaults, unless there are clear indications of costs and benefits of making alternative choices. Schneider et al. (2018) have shown that UI design influences choices, even unintentionally; they state “user interface, from organizational website to mobile app, can thus be viewed as a digital choice environment” nudging people to take certain actions over other by modifying what is presented or how it is presented. The example they cite is the Square mobile payment app which presents a “tipping” option by default; customers must select “no tipping” if they prefer not to give a tip which is additional effort and also triggers social “norming” and social belonging.

With respect to defaults specifically, Jachimowicz et al. (2019) in a meta analysis suggest two factors that partially account for the variability in defaults’ effectiveness: the contexts and domains where defaults were presented and their relative ease of implementation. Their findings also point to the importance of evaluating the intended population’s preferences when deciding when and how to deploy defaults. Huh et al. (2014) show that the observed choices of others can become social defaults, increasing their choice share. Social default effects are a novel form of social influence not due to normative or informational influence: participants were more likely to mimic observed choices when choosing in private than in public and when stakes were low rather than high.

The empirically observed difficulty of making rational choices, under the influence of defaults in particular and other choice architecture interventions in general, has been subject to several different explanations about human cognition and reasoning. One of those explanations invokes the notion of *bounded rationality*, in which Simon (1996) argues that the decision maker is often limited by the

availability of information and their cognitive ability to process all the information pertinent to the decision in order to make the optimal choice. As a result, the decision maker would “satisfice” – settling for a reasonable choice but not necessarily the best one. Another explanation points to the *failure of imagination*. Shackle (Shackle, 1964) contends that “Choice is amongst imagined experiences,” and Augier (2000), elaborating on Shackle’s theory, suggests that “Choice is amongst imagined experiences,” and “each alternative is considered with a variable amount of disbelief.” This point of view is particularly relevant to decision making in software engineering, as Somers (2017) has warned about the challenge of reasoning about program behavior in complex software systems.

It is worth noting that the aforementioned assignment of disbelief to different imagined scenarios is likely a process adaptive to subtle cues available in the decision-making context. Through a number of laboratory experiments, McKenzie et al. (McKenzie et al., 2018) demonstrate that a number of apparent “biases” in making choices stem from adaptive sensitivity to subtle contextual cues in the choice environment, which dynamically update the decision maker’s belief about the desirable course of action or the attribute distributions in the population. For example, the default option is often perceived as the recommendation from someone in a position of authority.

This perspective on dynamic belief and preference construction not only provides a different view of the psychology and rationality of decision making, it also suggests a different approach to choice architecture design. Whereas the traditional nudge approach tries to engineer specific decision outcomes, often by rerouting apparent biases so that they point in desirable directions, the authors suggest an alternate approach focused on facilitating the processes of decision making. For instance, McKenzie et al. (2018) called for overt messaging instead of covert “nudges” to support decision makers by increasing information salience in accordance to the information’s relevance and the decision maker’s attentional capacity. The above perspectives and results have informed our problem framing, interpretation of data, and development of mitigations for the API usability and correctness misalignment problem we examined.

3. Study Design

3.1. Overview

To measure the impact of the API usability and correctness misalignment, we conducted a web-based controlled experiment. In particular, we wanted to check how much of the problem can be mitigated by getting the programmer to read documentation about the problem, since API designers often resort to “user education” as the first and easiest to implement response to issues related to user experience.

The experiment was implemented as a scenario-based questionnaire using the survey software Qualtrics and administered over the Internet. The questionnaire had four main parts:

- **Screener:** Participants of the experiment were recruited from Dart’s user community without incentives. At the beginning of the experiment, each participant answered a few questions about their background and their knowledge about Dart in order to determine their eligibility.
- **Code snippet review tasks:** This was the main body of the questionnaire. The participant was asked to assess the correctness of code snippets written for six common text manipulation scenarios (see subsection 3.3). Participants in the treatment groups (more about experimental conditions below) received information about the grapheme clusters problem in the Dart String API after Scenario 1, simulating an experience of getting educated about the problem.
- **Reflect on assessment results:** After reviewing the code snippets in those six scenarios, participants in the treatment groups were given an opportunity to review and reflect on the correct assessments of the code snippets they examined.
- **Post-test questionnaire:** In the last part of the experiment, participants answered questions about their attitudes, preferences, and prior experience that might help contextualize their responses.

The full questionnaire design is available in the Appendix.

3.2. Experimental conditions

The experiment had a control condition and two treatment conditions. We randomly assigned the 183 participants to one of the three experimental conditions. The three conditions primarily differed in the amount of information about the grapheme clusters problem given to the participant after they evaluated the code snippet in Scenario 1, which was about extracting a substring (shown in Fig. 1).

Scenario 1

Imagine that you want to implement a function that deletes the last character from a string and returns the result as a new string. The string comes from user input in a text box.

You come across the following snippet online:

```
String skipLastChar(String text) {
    return text.substring(0, text.length - 1);
}
```

Note: the substring (int startIndex, [int endIndex]) method returns the substring of this string that extends from startIndex, inclusive, to endIndex, exclusive. For example, "hello".substring(0, 4) returns "hell".

Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)?

Yes

No

Maybe

Figure 1 - The first of the six code assessment scenarios participants went through in the experiment. The code snippet could produce incorrect results when the input text had grapheme clusters.

Specifically, the control group received no feedback at all after assessing scenario 1 and continued to review the rest of the code snippets. Both treatment groups received basic information about the fact that the Dart String API could produce incorrect results when extracting a substring from text that included grapheme clusters and an example of revising the snippet using the characters package. Treatment group 1 was given further explanation about the underlying UTF-16 representation of text in Dart’s default String API and why the API won’t handle grapheme clusters correctly. Table 1 summarizes the differences between the three experimental conditions.

Experimental Condition	Explanation Received after Scenario 1
Control	No explanation
Treatment 1	Full explanation
Treatment 2	Light explanation

Table 1 – The differences between the three experimental conditions.

3.3. Experimental tasks

As aforementioned, each participant was asked to review code snippets written for the following six text manipulation scenarios (see an example in Fig. 1):

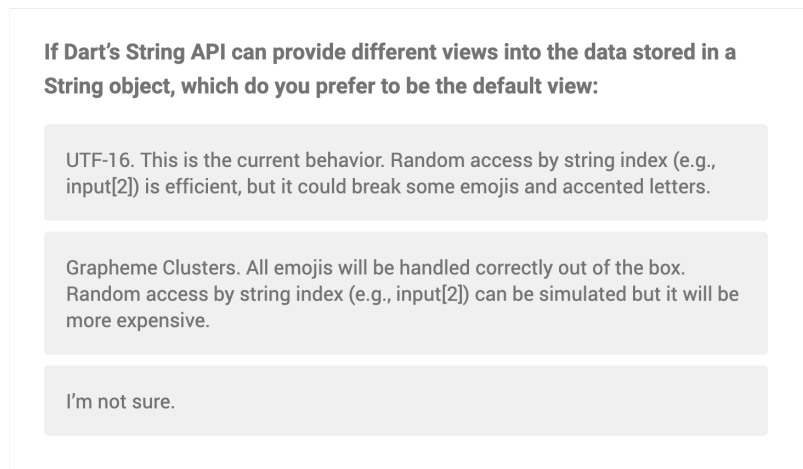
- 1) Extracting a substring
- 2) Validating email addresses using a regular expression
- 3) Checking character limit
- 4) Splitting a string on an emoji
- 5) Creating initials from a first name and a last name

6) Turning overflowed text into an ellipsis

The code snippets in the 6 scenarios all used the Dart String API to process their respective text input. Among them, Scenario 1, 3, 5, and 6 would produce unexpected results when grapheme clusters were part of the text being manipulated. Programmers could use the characters package instead as a remedy. The experiment used those scenarios to check false negatives – failures to identify potential programming errors. In contrast, Scenario 2 and Scenario 4 were designed to check false positives – the potential of participants overacting to the information provided about grapheme clusters and hence mistakenly dismissing the correct use of String API in those two scenarios.

3.4. Post-test questionnaire

The post-test questionnaire had two pages. The first page was focused on the participant's attitudes towards the grapheme clusters problem and their preferred default string data representation (see Fig. 2). This section of the questionnaire was only displayed to participants in the treatment groups, since the control group was not informed of the grapheme clusters problem in the experiment.



The image shows a screenshot of a questionnaire question. The question is: "If Dart's String API can provide different views into the data stored in a String object, which do you prefer to be the default view:". Below the question are three radio button options:

- UTF-16. This is the current behavior. Random access by string index (e.g., input[2]) is efficient, but it could break some emojis and accented letters.
- Grapheme Clusters. All emojis will be handled correctly out of the box. Random access by string index (e.g., input[2]) can be simulated but it will be more expensive.
- I'm not sure.

Figure 2 - One of the questions in the post-test questionnaire is about preferred string representation in Dart's String API.

On the second page of the questionnaire, the participant was asked to provide their background information, such as awareness of the grapheme clusters problem, prior knowledge about Dart's characters package, and familiarity with Swift's String API, which might help explain their behavior and attitudes shown in the experiment.

3.5. Hypotheses and data analysis

As mentioned, part of the experimental goal was to measure how much we can rely on programmer education (e.g., reading documentation) to help the programmer make sound API choices and hence mitigate the impact of the usability and correctness misalignment. Thus, the experiment was designed to test the following hypotheses:

- H1: Participants who had prior exposure to the grapheme clusters problem in text manipulation were more likely to identify the problem in code snippets.
- H2: Participants who were informed of the grapheme clusters problem early in the experiment (i.e., the two treatment groups) would be able to more accurately assess the correctness of the code snippets than those who weren't (i.e., the control group).
- H3: Participants who were given more in-depth explanations of the grapheme clusters problem early in the experiment (i.e., treatment group 1) would do better in identifying the problem in the code snippets than those given a basic explanation (i.e., treatment group 2).
- H4: The majority of participants would prefer a String API aware of grapheme clusters by default after reflecting on their own ability to make correct API choices.

4. Findings

In this section, we report the main findings from analyzing the results of the experiment by the order of the hypotheses stated above. Due to space limits, we omit the results about false positives in evaluating Scenario 2 and 4. Assessment results of Scenario 2 were also muddled by participants' lack of experience with regular expressions.

4.1. Prior exposure to the grapheme clusters problem showed a negligible effect

As expected, a minority of our 183 participants reported prior awareness of the grapheme clusters problem. Specifically, 53 said they had only heard of the issue, and 22 said they had run into the issue themselves. The analysis below was focused on participants' assessment of the code snippet in Scenario 1, before participants in the two treatment groups were informed of the grapheme clusters problem after this scenario, as described in the Study Design section. As a reminder, the correct response to the assessment question in Scenario 1 should be "No."

Among the 53 participants who had heard of the issue before, 26% made a correct assessment of the code snippet in Scenario 1. And among the 23 participants who claimed to have first-hand experience with the problem, 23% made a correct assessment. Both were only slightly better than the participants who reported no knowledge of the issue prior to assessing Scenario 1, as shown in the green portions of the bar chart in Fig. 3. A chi-squared test didn't show statistical significance in the differences between those without prior knowledge and those who were aware of the problem to some degree. Therefore, the data does not support H1.

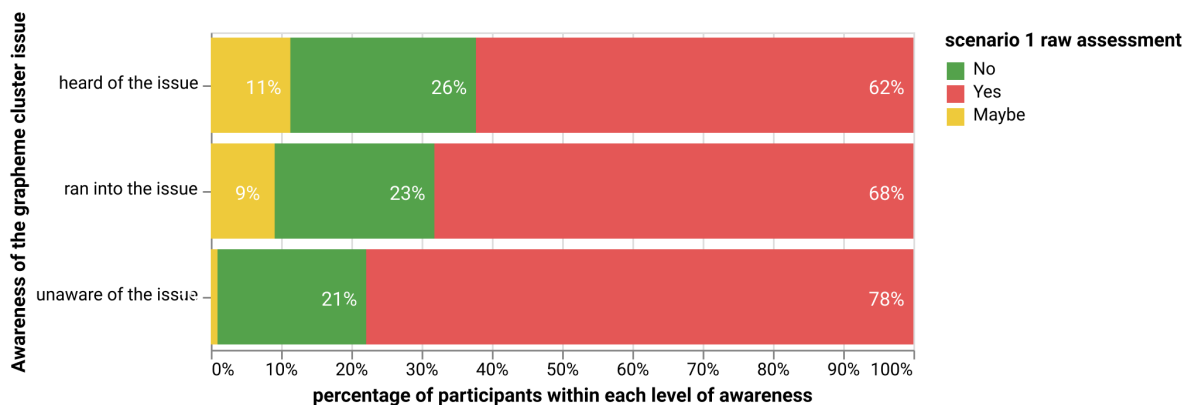


Figure 3 - Participants' assessments of the code snippet in Scenario 1 broken down by their prior awareness of the grapheme clusters problem. The correct answer should be "No."

4.2. Explaining the grapheme clusters problem upfront was useful but insufficient

In the experiment, both treatment groups were informed of the correct assessment of the code snippet in Scenario 1 and given different amounts of explanation about why the snippet could produce incorrect results. Since this information was given right before those participants went on assessing the rest of the code snippets, it's not surprising that they performed much better than the control group (see Fig. 4). For example, in Scenario 3 (counting the number of characters), only 15.5% of the participants in the control group were able to determine that the snippet could produce wrong results when grapheme clusters were part of the string, while 46.3% of treatment group 1 and 36.2% of treatment group 2 were able to do so. Moreover, chi-squared tests showed statistically significant differences across the three groups' assessment performance for the three scenarios (#2, #5, and #6), where the provided code snippets could have trouble processing grapheme clusters.

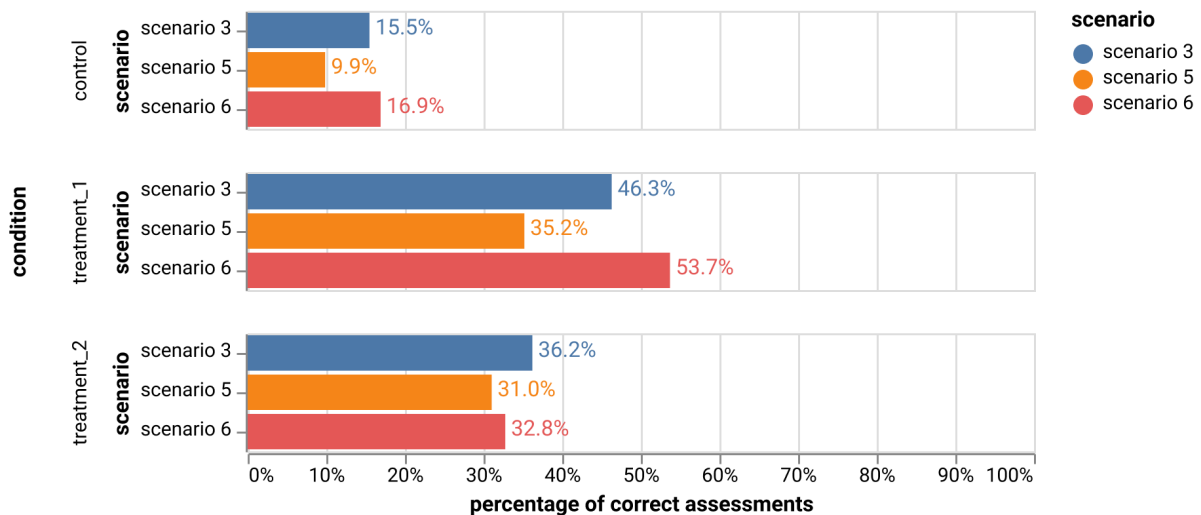


Figure 4 - The percentage of participants who made a correct assessment of the code snippet in Scenario 3, 5, and 6 across three experimental conditions

Disappointingly, both treatment groups still made more incorrect assessments than correct assessments. On average, treatment group 1 (received an in-depth explanation about the grapheme clusters problem) and treatment group 2 (received a basic explanation) achieved an average assessment success rate of 45% and 33%, respectively. In conclusion, the data does support H2, but the effect of providing the participants with the documentation of the grapheme cluster problem and the characters package is much weaker than many might have expected.

4.3. The amount of explanation only made a small difference

In the experimental design, we gave treatment group 1 a deeper and more thorough explanation in the hope that participants in this group could apply this knowledge to the assessment of subsequent text manipulation scenarios that were different from Scenario 1. The data shows that treatment group 1 did seem to perform better than treatment group 2 in every scenario where using the String API was problematic (see Fig. 4). The largest difference was observed in Scenario 6, where the code snippet needs to enforce a character limit in a text field and replace any overflowed text with an ellipsis. In this scenario, 53.7% of participants in treatment group 1 made a correct assessment, while only 32.8% of the treatment group 2 were able to do so. However, the difference between the two groups' performance in Scenario 6 was not statistically significant according to a chi-squared test ($p = 0.055$), so were the test results for the other two scenarios.

4.4. Participants lacked confidence in their ability to make correct API choices

As mentioned in the study design, participants in the treatment groups were given an opportunity to compare their own assessments of the coding scenarios with the correct assessments and read a brief explanation about each scenario. We were curious about how that opportunity to reflect on their assessment performance might influence their opinions about how the String API should be designed.

Responding to a multiple choice question in the post-test questionnaire, about half of those participants said they would prefer a string representation aware of grapheme clusters by default, given the trade off between correctness and a potential hit to computational efficiency (see Fig. 5).

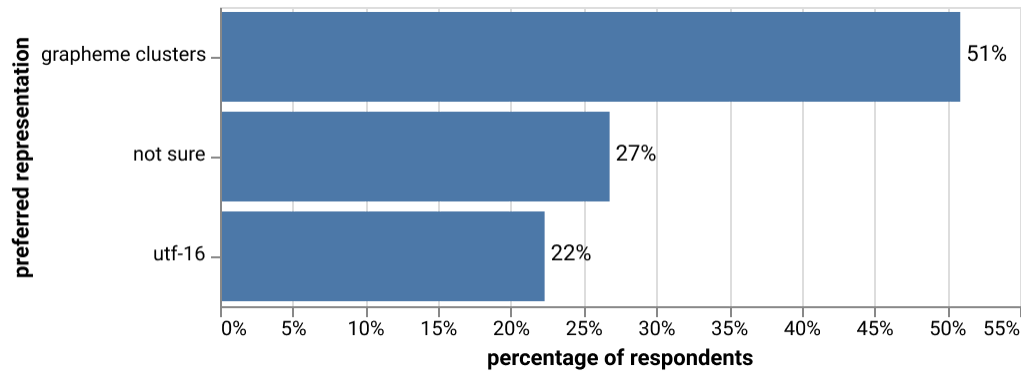


Figure 5 - Participants' preferred default string representation as stated in the post-test questionnaire

Some participants voiced concerns over their ability to be constantly on the lookout for subtle issues due to mishandling of grapheme clusters:

“Even when you understand how this currently works in theory, it is EXTREMELY easy to forget in practise. The current behavior WILL create many, many bugs.”

“...making the api catch those errors by default will improve the quality of apps since most developers don't handle emoji use cases until they have a real crash.”

For the minority who preferred UTF-16 as the default string representation (i.e., the status quo in Dart), some felt there was value to be consistent with other languages:

“Current behaviour is consistent with other language paradigms which allow low level manipulations.”

Others thought emojis were still rare in their particular use cases:

“In many use-cases, emojis probably won't show up. Let's say I'm developing an application for an insurance company, or maybe a medical info app. In these cases, it's highly unlikely to have emojis due to the professional nature of the app.”

To sum up, the data weakly supports H4, but a lot of participants were unable to make up their mind, reflecting the complexity of the trade-offs involved in modernizing a foundational API such as String in a mature programming language.

5. Discussion

In this section, we discuss what might have contributed to the difficulty of making API choices observed in the experiment, the implications for designing programming tools and environments, and the larger question of how priorities and values are framed and acted upon in software development.

5.1. Factors exacerbating the impact of misalignment

Our findings show that the misalignment between API usability and correctness can be difficult for the API user to deal with. The conventional wisdom might lead us down a path of more user education about the problem through documentation, but the results of our experiment, which in many ways simulated such an approach, suggest that it is insufficient. So why did our study participants, after they had been exposed to the grapheme clusters problem either before the study or during it, still experienced so much difficulty identifying it in the simple code snippets they examined? There could be several contributing factors, some of which we have touched on when describing related work:

- *Insufficient absorption and recall of information.* Although the explanation about the grapheme clusters problem and the characters package was provided to all participants in the treatment groups, there was no guarantee that every participant was able or willing to fully absorb this information in the first place and to recall that information later when it was needed in evaluating additional scenarios. This was essentially *bounded rationality* at work.

- *The failure of imagination.* It could be difficult to imagine an edge case that would produce incorrect results. Grapheme clusters is a complicated concept in unicode. Only some emojis and non-ASCII characters need to be represented in more than one UTF-16 code unit, and only specific types of text manipulations would break them. As a result, it's hard for programmers to come up with example strings to verify the correctness of their code.
- *The lack of learning transfer* (Perkins et al., 1992). Some participants might have understood why the first coding scenario was problematic and the grapheme cluster issue in abstract after receiving feedback, but they might not have developed the ability to apply this knowledge to the assessment of subsequent coding scenarios.
- *The habitual use of APIs and routinized operations.* Because manipulating text is an extremely common task in programming, the ways of solving different text manipulation problems are likely to have become routines if not habits. It is thus difficult to critically evaluate familiar code snippets used so many times before without issues.

When one or more of these factors are at play, the impact of the misalignment between API usability and correctness can be especially hard to overcome.

5.2. Implications for design

While our results suggest that it is important to avoid or remove this type of misalignment in API design whenever possible, it is not always feasible when evolving an existing SDK with many users and real-world applications built on top of it. Forcing a realignment of the API correctness and usability could break backwards compatibility and lead to ecosystem fragmentation.

Nonetheless, the SDK designer could still consider the following mitigations that specifically address the three factors contributing to the difficulty of identifying and managing the misalignment.

- *Making the choices visible and readily available.* For example, Flutter, a popular UI framework for Dart, has integrated the characters package in its SDK, so Flutter programmers don't need to manually import it to their projects. In addition, the Characters API is also made available on String objects and literals through extension methods. Therefore, in a Flutter UI programming context, it's much easier to get reminded of this new, more robust way of manipulating text through the IDE's autocomplete facility (see Fig. 6).

```
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: TextField(
        controller: _controller,
        onSubmitted: (String value) {
          // print out the number of characters
          print(value.);
        },
      ),
    ),
  );
}
```

characters
codeUnits
hashCode
isEmpty
isNotEmpty
length
runes
runtimeType

Figure 6 - Discovering the characters API is much easier after Flutter integrated it into its SDK.

- *Providing assistance in creating useful test cases.* This strategy intends to address the failure of imagination in making API choices. Specifically, programming tools need to help programmers write test cases that cover grapheme clusters by suggesting example input strings and showing warnings when existing test cases fail to cover edge cases. In addition to augmenting automated tests, a tool simulating diverse text inputs, such as emojis and non-ASCII characters, could help the developer discover text manipulation bugs early.
- *Breaking habitual use of API by making new local defaults.* When it's infeasible to change the global default, it might be possible to enact local defaults to mitigate risks of misusing APIs. For example, a UI toolkit often requires the programmer to create a callback function to handle user input. This particular context is highly susceptible to issues related to grapheme

clusters. The code editor could proactively complete a template for such callback functions where the grapheme-aware API is used in the generated code. Additionally, the code editor can use lints to warn programmers in potential high-risk text manipulation contexts.

5.3. Developer incentives and values

The need for processing grapheme clusters far predates emojis. In fact, they have been used to encode non-ASCII characters in many natural languages for almost 20 years (Davis, 2001). But why did this problem, a prime example of API usability and correctness misalignment, seem to draw little interest from the programming language design community until recently?

To answer this question, we need to critically examine software developers' incentives and the economics of designing and building software for a diverse user population. The value of being correct in the presence of grapheme clusters in text manipulation was probably considered less important than providing a straightforward, familiar, and efficient String API until two underlying economic forces became more prominent in the last few years:

- The rapid growth of both the proportion of Internet users who speak a language other than English and the proportion of non-English web pages (Wikipedia contributors, 2020)
- The rise of emojis in daily electronic communications (Danesi, 2016)

More recently designed programming languages, such as Swift, made the grapheme clusters a first-class citizen in its String API. This change of attitude mirrors the gradual rise of accessibility, which also suffered from *the failure of imagination*. For example, an able-bodied web developer could have trouble thinking through the experience of the site from the perspective of a visually impaired user. Modern developer tools provide facilities to simulate the experience of users with accessibility needs. We believe a similar approach can be fruitful in helping developers build awareness and empathy towards users with diverse text input needs.

6. Conclusions

In this paper, we proposed and examined the problem of misaligned API usability and correctness – the more usable API produces less correct results than a harder to use or less familiar API. We measured how well programmers can handle this type of misalignment through a controlled experiment in the context of manipulating unicode text that includes grapheme clusters, widely used to represent emojis and characters in non-English scripts. Those characters are not properly supported by the default String API in most mainstream programming languages, and they often require add-on libraries to properly process them. Our experiment results suggest that it is difficult for many programmers to identify instances where the default String API could produce incorrect results in common text manipulation scenarios, despite user education and warnings provided earlier in the experiment in the form of documentation. The findings lead to specific implications for designing programming environments that facilitate the process of making API choices in order to mitigate such misalignments.

7. Acknowledgements

We thank our study participants for their time. In addition, this study would not be possible without the support and feedback from Ian Hickson, Michael Thomsen, Lasse Nielsen and Leaf Petersen from the Flutter team and the Dart team at Google.

8. References

- Augier, M. (2000). Rationality, imagination and intelligence: some boundaries in human decision-making. *Industrial and Corporate Change*, 9(4), 659–681.
- Blackwell, A. F., Britton, C., Cox, A., Green, T. R. G., Gurr, C., Kadoda, G., Kutar, M. S., Loomes, M., Nehaniv, C. L., Petre, M., Roast, C., Roe, C., Wong, A., & Young, R. M. (2001). Cognitive

- Dimensions of Notations: Design Tools for Cognitive Technology. In *Lecture Notes in Computer Science* (pp. 325–341). https://doi.org/10.1007/3-540-44617-6_31
- Clarke, S. (2004). Measuring API usability. *Dr. Dobbs's Journal Windows*, S6–S9.
- Danesi, M. (2016). *The Semiotics of Emoji: The Rise of Visual Language in the Age of the Internet*. Bloomsbury Publishing.
- Davis, M. (2001, March 11). *Text Boundaries (Version 1)*. Unicode Technical Reports. <https://www.unicode.org/reports/tr29/tr29-1.html>
- Davis, M., & Chapman, C. (2020, February 19). *Unicode Text Segmentation (Revision 37)*. Unicode Technical Reports. <https://unicode.org/reports/tr29/>
- Fahl, S., Harbach, M., Perl, H., Koetter, M., & Smith, M. (2013). Rethinking SSL development in an appified world. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 49–60.
- Huh, Y. E., Vosgerau, J., & Morewedge, C. K. (2014). Social defaults: Observed choices become choice defaults. *The Journal of Consumer Research*, 41(3), 746–760.
- Jachimowicz, J. M., Duncan, S., Weber, E. U., & Johnson, E. J. (2019). When and why defaults influence decisions: a meta-analysis of default effects. *Behavioural Public Policy*, 3(2), 159–186.
- McKenzie, C. R. M., Sher, S., Leong, L. M., Müller-Trede, J., & Others. (2018). Constructed preferences, rationality, and choice architecture. *Review of Behavioral Economics*, 5(3-4), 337–360.
- Microsoft. (2018, May 31). *Surrogates and Supplementary Characters*. <https://docs.microsoft.com/en-us/windows/win32/intl/surrogates-and-supplementary-characters>
- Murphy-Hill, E., Sadowski, C., Head, A., Daughtry, J., Macvean, A., Jaspan, C., & Winter, C. (2018). Discovering API Usability Problems at Scale. *2018 IEEE/ACM 2nd International Workshop on API Usage and Evolution (WAPI)*, 14–17.
- Myers, B. A., & Stylos, J. (2016). Improving API usability. *Communications of the ACM*, 59(6), 62–69.
- Nielsen, J. (1994, April 24). *10 Heuristics for User Interface Design: Article by Jakob Nielsen*.

- Nielsen Norman Group. <https://www.nngroup.com/articles/ten-usability-heuristics/>
- Perkins, D. N., Salomon, G., & Others. (1992). Transfer of learning. *International Encyclopedia of Education*, 2, 6452–6457.
- Piccioni, M., Furia, C. A., & Meyer, B. (2013). An Empirical Study of API Usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*.
<https://doi.org/10.1109/esem.2013.14>
- Schneider, C., Weinmann, M., & Vom Brocke, J. (2018). Digital nudging: guiding online user choices through interface design. *Communications of the ACM*, 61(7), 67–73.
- Shackle, G. L. S. (1964). *General thought-schemes and the economist: the second Woolwich Economic Lecture delivered before the Woolwich Polytechnic on 3 March 1964*. Woolwich Polytechnic, Department of Economics and Business Studies.
- Simon, H. A. (1996). *The Sciences of the Artificial - 3rd Edition* (3rd ed.). The MIT Press.
- Somers, J. (2017, September 26). The Coming Software Apocalypse. *The Atlantic*.
<https://www.theatlantic.com/technology/archive/2017/09/saving-the-world-from-code/540393/>
- Stylos, J., Clarke, S., & Myers, B. A. (2006). Comparing API Design Choices with Usability Studies: A Case Study and Future Directions. *PPIG*, 17.
- Stylos, J., & Myers, B. (2007). Mapping the Space of API Design Decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*.
<https://doi.org/10.1109/vlhcc.2007.44>
- Thaler, R. H., & Sunstein, C. R. (2009). *Nudge: Improving Decisions About Health, Wealth, and Happiness* (Revised & Expanded edition). Penguin Books.
- Wikipedia contributors. (2020, August 22). *Languages used on the Internet*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Languages_used_on_the_Internet
- Cronqvist, H; Thaler, R (2004). Design choices in privatized social security systems: Learning from the Swedish experience. *American Economic Review*. 94 (2): 424–8.
- Dinner, I., Johnson, E. J., Goldstein, D. G., & Liu, K. (2011). Partitioning default effects: why people choose not to choose. *Journal of Experimental Psychology: Applied*, 17(4), 332.

- Jachimowicz, J. M., Duncan, S., Weber, E. U., & Johnson, E. J. (2019). When and why defaults influence decisions: A meta-analysis of default effects. *Behavioural Public Policy*, 3(2), 159-186.
- Johnson, E. J., Shu, S. B., Dellaert, B. G., Fox, C., Goldstein, D. G., Häubl, G., ... & Wansink, B. (2012). Beyond nudges: Tools of a choice architecture. *Marketing Letters*, 23(2), 487-504.
- Johnson, E.J.; Goldstein, D.G. (2003). Do Defaults Save Lives? *Science*. 302 (5649): 1338–1339.
- Kahneman, D., & Tversky, A. (2013). Choices, values, and frames. In *Handbook of the fundamentals of financial decision making: Part I* (pp. 269-278).
- Huh, Y. E., Vosgerau, J., & Morewedge, C. K. (2014). Social defaults: Observed choices become choice defaults. *Journal of Consumer Research*, 41(3), 746-760.
- Larrick, R.P. and Soll, J.B (2008). The MPG Illusion. *Science*. 320 (5883): 1593–4.
- McKenzie, C. R., Sher, S., Leong, L. M., & Müller-Trede, J. (2018). Constructed preferences, rationality, and choice architecture. *Review of Behavioral Economics*, 5(3-4), 337-360.
- Scheibehenne, B., Greifeneder, R. and Todd, P. (2010). Can there ever be too many options? A meta-analytic review of choice overload. *Journal of Consumer Research*. 37 (3): 409–25.
- Schneider, C., Weinmann, M., & Vom Brocke, J. (2018). Digital nudging: guiding online user choices through interface design. *Communications of the ACM*, 61(7), 67-73.

Appendix: Questionnaire Design

Screeners

Thank you for your interest in participating in this research study on the usability of Dart's API. To determine your eligibility, please answer questions on this page.

If you qualify, you can expect to complete the survey within 15 minutes. Please make sure you will not be interrupted. If you opened this survey on a mobile device, we kindly request you to fill it out from a desktop/laptop computer to ensure correct rendering of the content.

By submitting your responses in this survey, you acknowledge that you are at least 18 years of age and that Google and its affiliates may use your responses to improve Google's products and services in accordance with [Google Privacy Policy](#).

#	Question	Criterion
Q2	What is your level of experience with programming in Dart? <ul style="list-style-type: none">• No experience• Awareness• Novice• Intermediate• Advanced• Expert	Criterion: must be "Intermediate" or higher to be eligible.
Q3	How proficient are you in English? <ul style="list-style-type: none">• None• Beginner• Intermediate• Advanced• Native	Criterion: must be "Intermediate" or higher to be eligible.
Q4	How confident are you in using the following Dart APIs? Options <ul style="list-style-type: none">• Async, await and future• String• Dates and times• Math• Regular expressions Levels <ul style="list-style-type: none">• Not at all confident• Slightly confident• Moderately confident• Quite confident• Totally confident	Criterion: confidence in using the String API must be "moderately confident" or higher to be eligible.

Introduction

In this survey, we are measuring the experience of reading Dart programs for the purpose of improving its API. You will go through a number of coding scenarios and determine if a given code snippet can satisfy the requirements described in each scenario. Both your answers and the time you spend on answering each question will be recorded in this survey.

Please read the information in the survey carefully, which may help you better assess those coding scenarios. To ensure the validity of the study results, we request you to not share the survey with others. On behalf of the Dart team, thank you for your time.

Scenario 1 (substring)

Note: the purpose of this task is to introduce the grapheme clusters issue to participants in the treatment groups. Treatment group 1 will receive additional conceptual explanation of the problem.

Imagine that you want to implement a function that deletes the last character from a string and returns the result as a new string. The string comes from user input in a text box.

You come across the following snippet online:

```
String skipLastChar(String text) {  
    return text.substring(0, text.length - 1);  
}
```

Note: the substring (int startIndex, [int endIndex]) method returns the substring of this string that extends from startIndex, inclusive, to endIndex, exclusive. For example, "hello".substring(0, 4) returns "hell".

#	Question	Display Logic
Q7	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? <ul style="list-style-type: none">• Yes• No• Maybe	
Q9	What do you think could be missing or incorrect in this snippet, if any? _____	Show if answer to Q7 is "No" or "Maybe"

Scenario 1 feedback

Note: this block will be displayed to both treatment groups but not the control group.

The correct answer is "No", because the code snippet won't pass the test below.

Test code:

```
test("skipLastChar(text) removes the last character from the string", () {  
    var string = 'Hi 🇩🇪';  
    expect(skipLastChar(string), equals('Hi '));  
});
```

Test results:

```
Expected: 'Hi '  
Actual: 'Hi D???'  
Which: is different. Both strings start the same, but the actual value  
also has the following trailing characters: D???
```

It turns out Dart's String API doesn't handle some emojis and non-English characters (e.g., `ğ`, `𑍎`, and `𑍎`) well. The formal term for those characters is extended grapheme clusters. The good news is that Dart has an experimental package called `characters` to handle this kind of situation. Here is how to rewrite this snippet using the package:

```
String skipLastChar(String text) {  
  return text.characters.skipLast(1).toString();  
}
```

Note that the package provides an extension method which turns a String object into a Characters object.

#	Question	Display Logic & Rationale
Q11	<p>Before participating in this survey, had you run into, or heard about, any issues when using Dart's String API to manipulate text input that includes extended grapheme clusters such as emojis (e.g., 🍌 😊 🇩🇪) and accented letters (e.g., café)?</p> <ul style="list-style-type: none">• I had run into such issues before.• I had heard of such issues but never ran into them.• I hadn't heard of such issues before.• I'm not sure.	<p>Rationale: check if the respondents knew about this issue yet still gave the wrong answer.</p>

Additional explanation for treatment group 1

```
String skipLastChar(String text) {  
  return text.substring(0, text.length - 1);  
}
```

So what was the fundamental problem in this code snippet? There are a couple of things going on here:

- Dart's standard String class uses the UTF-16 encoding, which means that the string is stored in a sequence of 16-bits code units. The length property of String returns the number of those UTF-16 units.
- However, some emojis and non-English characters require more than one UTF-16 code unit to store. For example, `🇩🇪` is stored in 4 UTF-16 code units in a Dart String. Such characters are formally called extended grapheme clusters.

It is clear that when using Dart String’s substring method to remove the last “character” from 'Hi 🇩🇰', it will remove the last of the four code units representing the flag, leaving the resulting substring corrupted.

The Characters package was created to address this limitation of the String API. Please take a few minutes to take a look at the documentation of the characters package below:

[Insert screenshot: <https://pub.dev/documentation/characters/latest/characters/Characters-class.html>]

Transitional page

Next, you will read a few more coding scenarios and evaluate code written to satisfy those scenarios. If you cannot determine whether the code would work or not, please respond “maybe” instead of trying to run the code outside of the survey.

Some scenarios will provide correct answers immediately after you give your assessment, while others will show correct answers at the end of this exercise.

Scenario 2 (email validation)

Note: the purpose of this task is to check if the participants in the two treatment groups would overreact to what they just learned about the String API’s limitations in processing grapheme clusters. The snippet in this scenario is actually correct.

You’re asked to review code written to implement the following requirements:

- The code is a function that checks if the text input contains a likely email address.
- Since email addresses may include non-English characters these days, the check needs to be inclusive.
- After some testing, the following regular expression is considered good enough:
`['^@\.\.]+\@[^\.\.]+\.[a-z]*'`

```
bool containEmail(String email) {
  return email.contains(RegExp(r'^@\.\.]+\@[^\.\.]+\.[a-z]*'));
}
```

#	Question	Display Logic
Q16	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code and the regular expression have been checked)? <ul style="list-style-type: none"> ● Yes ● No ● Maybe 	
Q18	What do you think could be missing or incorrect in this snippet, if any? <ul style="list-style-type: none"> ● The code should use the characters package instead. ● The problem is unrelated to the characters package. Please describe the problem briefly: _____ ● I don’t know. 	Answer to the previous Q is “No” or “Maybe” AND the participant is not in the control group

	Next page	
Q19	The correct answer is “Yes”, this snippet can satisfy the requirements described in the scenario. In fact, this scenario requires the String API. First, regular expressions can be used to match text with non-English characters. Second, the characters package doesn’t support regular expressions.	Show if the participant is not in the control group.

Scenario 3 (count characters)

Note: the purpose of this task is to check if the participant would realize they need to use the characters package to correctly measure the number of characters in a string.

You’re asked to review code written to implement the following requirements:

- The code is a function that checks if the text entered by the user has exceeded a specific number of characters.
- The function returns a positive number of remaining characters if the limit hasn’t been reached, or a negative number of extra characters if the limit has been exceeded.

```
int checkMaxLength(String input, int limit) {
    var length = input.length;
    return limit - length;
}
```

#	Question	Display Logic
Q20	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? <ul style="list-style-type: none"> • Yes • No • Maybe 	
Q22	What do you think could be missing or incorrect in this snippet, if any? <ul style="list-style-type: none"> • The code should use the characters package instead. • The problem is unrelated to the characters package. Please describe the problem briefly: _____ • I don’t know. 	Answer to the previous Q is “No” or “Maybe” AND not in the control group

Scenario 4 (split string)

Note: the snippet was initially considered to be correct in order to test overreaction, but we later discovered a condition where this snippet would produce incorrect results. Therefore, data from this scenario was excluded from the analysis.

You're asked to review code written to implement the following requirements:

- The code is a function that splits a string at '★' and returns a list of substrings. For example, turning 'space★is★for★everybody' into ['space', 'is', 'for', 'everybody'].

Review the following code snippet and determine if it can correctly meet your requirements.

```
List splitStarSeparatedWords(String text) {
    return text.split('★');
}
```

#	Question	Display Logic
Q23	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? <ul style="list-style-type: none"> • Yes • No • Maybe 	
Q25	What do you think could be missing or incorrect in this snippet, if any? <ul style="list-style-type: none"> • The code should use the characters package instead. • The problem is unrelated to the characters package. Please describe the problem briefly: _____ • I don't know. 	Answer to the previous Q is "No" or "Maybe" AND not in the control group
	Next page	
Q26	The correct answer is "Yes", this snippet can satisfy the requirements described in the scenario. The snippet works because the String API's split method can appropriately handle emojis as separators. Moreover, the characters package doesn't have a split method or its equivalent.	Show if the participant is not in the control group.

Scenario 5 (create initials)

Note: the purpose of this task is to test if the participant would realize that the index operator can break grapheme clusters.

You're asked to review code written to implement the following requirements:

- The code is a function that creates initials from the first name and the last name the user enters in two separate text fields.
- For example, the function needs to generate initials such as "JS" from "john" and "smith".

```
String createInitials(String firstName, String lastName) {
    return firstName[0].toUpperCase() + lastName[0].toUpperCase();
}
```

#	Question	Display Logic
Q27	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? <ul style="list-style-type: none"> • Yes • No • Maybe 	
Q29	What do you think could be missing or incorrect in this snippet, if any? <ul style="list-style-type: none"> • The code should use the characters package instead. • The problem is unrelated to the characters package. Please describe the problem briefly: _____ • I don't know. 	Answer to the previous Q is "No" or "Maybe" AND not in the control group

Scenario 6 (text overflow ellipsis)

Note: the purpose of this scenario is to check if the participant still remembers the substring scenario where they were first exposed to the grapheme manipulation issue.

Your app needs to display a list of messages, one per line. You're asked to review code written to implement the following requirements:

- The code is a function that displays text overflow as an ellipsis when the message's length exceeds the given character limit.
- For the purpose of this survey, you can ignore the varying widths of characters.

```
String textOverflowEllipsis(String text, int limit) {
    if (text.length > limit) {
        return text.substring(0, limit - 3) + '...';
    } else {
        return text;
    }
}
```

	Question	Display Logic
Q30	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? <ul style="list-style-type: none"> • Yes • No • Maybe 	
Q32	What do you think could be missing or incorrect in this snippet, if any? <ul style="list-style-type: none"> • The code should use the characters package instead. • The problem is unrelated to the characters package. Please describe the problem briefly: _____ • I don't know. 	Answer to the previous Q is "No" or "Maybe" AND not in the control group

Review correct answers

Please review the expected answers to the questions in the previous code scenarios. This page also shows the correct code for coding scenarios where the snippets provided had issues.

Summary of correct answers

The table below shows the expected responses to the question "Will the code above correctly satisfy the requirements described in the scenario?"

Scenario #	Your Answer	Correct Answer
2 (email validation)	<Insert the participant's answer>	Yes
3 (count characters)	<Insert the participant's answer>	No
4 (split string)	<Insert the participant's answer>	Yes
5 (create initials)	<Insert the participant's answer>	No
6 (text overflow ellipsis)	<Insert the participant's answer>	No

Scenario 2 (email validation)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: Yes.
- This scenario requires the String API. First, regular expressions can be used to match text with grapheme clusters. Second, the characters package doesn't support regular expressions.

Scenario 3 (count characters)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: No. The code won't pass the test below.
- Test code:

```
test("checkMaxLength(String input, int limit) returns how many characters left
in the space", (){
  var input = "Laughter is the sensation of feeling good all over and showing
it principally in one place.";
  var limit = 140;
  expect(checkMaxLength(input, limit), equals(49));
  input = "Laughter 😄 is the sensation of feeling good all over and showing
it principally in one place.";
  expect(checkMaxLength(input, limit), equals(47));
});
```

- Test results:

```
00:00 +1 -2: checkMaxLength(String input, int limit) returns how many
characters left in the space [E]
  Expected: <47>
  Actual: <46>

package:test_api expect
test.dart 22:5 main.<fn>
```

- Correct code:

```
int checkMaxLength(String input, int limit) {
    var length = input.characters.length;
    return limit - length;
}
```

Scenario 4 (split string)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: Yes
- The snippet works because the String API's split method can appropriately handle emojis as separators. Moreover, the Characters class doesn't have a split method or its equivalent.

Scenario 5 (create initials)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: No. The reason is that the subscript interface as used in name[0] could grab a fraction of a character. For example, the code won't pass the test below, because the character "É" can be a combination of an "E" and an acute "'".
- Test code:

```
test(
    "createInitials(firstName, lastname) creates initials from a first name
and a last name",
    () {
        var firstName = "étienne";
        var lastname = "bézout";
        expect(td.createInitials(firstName, lastname), equals('ÉB'));
    });
```

- Test result:

```
00:01 +1 -5: createInitials(firstName, lastname) creates initials from a first
name and a last name [E]
Expected: 'ÉB'
Actual: 'EB'
Which: is different.
Expected: ÉB
Actual: EB
          ^
Differ at offset 1
```

- Correct code:

```
String createInitials(String firstName, String lastName) {
    var initials = firstName.characters.first.toUpperCase() +
        lastName.characters.first.toUpperCase();
    return initials;
}
```

Scenario 6 (text overflow ellipsis)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: No, the code won't pass the test below.

- Test code:

```
test(
  "textOverflowEllipsis(String text, int limit) displays an ellipsis for
  overflown text",
  () {
    var input = "🦖rhinoceros";
    var limit = 7;
    expect(td.textOverflowEllipsis(input, limit), equals("🦖rhi..."));
  });
```

- Test result:

```
00:01 +1 -3: textOverflowEllipsis(String text, int limit) displays an ellipsis
for overflown text [E]
Expected: '🦖rhi...'
Actual: '🦖rh...'
Which: is different.
Expected: 🦖rhi...
Actual: 🦖rh...
          ^
Differ at offset 4
```

- Correct code:

```
String textOverflowEllipsis(String text, int limit) {
  return text.characters.take(limit - 3).toString() + '...';
}
```

Attitudes and preferences

(Display logic: treatment groups only)

#	Question	Display Logic & Rationale
Q37	How important is it to handle emojis correctly in text manipulation tasks while developing apps? <ul style="list-style-type: none"> • Extremely important • Very important • Moderately Important • Slightly important • Not at all important 	Rationale: To measure perceived importance of the issue.
Q38	Which of the following statements best describes how you would approach text manipulation in Dart in the future? <ul style="list-style-type: none"> • I would stick with the String API unless I have evidence that something is broken for my app's users. • I would evaluate every use case of the String API in my project and decide when the character package should be used instead. 	Rationale: To measure potential behavior change

	<ul style="list-style-type: none"> ● I would use the characters package to manipulate text whenever it can be used. ● I'm not sure. ● Other: _____ 	
Q39	<p>How easy is it to determine when you need to use the characters package to write correct code.</p> <ul style="list-style-type: none"> ● Very easy ● Somewhat easy ● Neutral ● Somewhat hard ● Very hard 	Rationale: To measure confidence in making intuitive choices.
Q40	<p>In your own words, how would you summarize situations where the characters package should be used to properly manipulate text?</p> <p>_____</p>	Rationale: To measure, how well the user can come up with rules of thumb they can remember. A rubric needs to be created to grade this answer.
Q41	<p>If Dart's String API can provide different views into the data stored in a String object, which do you prefer to be the default view:</p> <ul style="list-style-type: none"> ● UTF-16. This is the current behavior. Random access by string index (e.g., input[2]) is efficient, but it could break some emojis and accented letters. ● Grapheme Clusters. All emojis will be handled correctly out of the box. Random access by string index (e.g., input[2]) can be simulated but it will be more expensive. ● I'm not sure. 	Rationale: To measure user preference of default string view.
Q42	<p>Why do you prefer the option you chose in the previous question?</p> <p>_____</p>	

Participant background

#	Question	Display Logic & Rationale
	You're almost done. We have just a couple more questions on this page.	
Q43	<p>Which types of apps have you developed using Dart/Flutter? (Select all that apply)</p> <ul style="list-style-type: none"> ● Business and productivity tools (e.g., calendar, to-do) ● Communication and social network ● Education ● Enterprise ● Financing and banking ● Health ● Games 	Rationale: Apps in categories such as "business and productivity tools" or "communication and social network" are more likely to have user input that includes EGC.

	<ul style="list-style-type: none"> ● Lifestyle, food, and drink (e.g. shopping, news, travel, fitness) ● Music and Video ● Utilities (e.g., weather, calculator) ● Sports ● Other 	
Q44	<p>Is English the primary language in your country?</p> <ul style="list-style-type: none"> ● Yes ● No 	Rationale: Developers in non-English speaking countries might be more aware of the EGC issue.
Q45	<p>Before participating in this survey, had you run into, or heard about, any issues when using Dart’s String API to manipulate text input that includes extended grapheme clusters such as emojis (e.g., 🍌 😄 🇩🇰) and accented letters (e.g., café)?</p> <ul style="list-style-type: none"> ● I had run into such issues before. ● I had heard of such issues but never ran into them. ● I hadn’t heard of such issues before. ● I’m not sure. 	<p>Logic: show if condition = control. Respondents in treatment conditions get this question right after scenario 1.</p> <p>Rationale: check if the respondents knew about this issue yet still gave the wrong answer.</p>
Q46	<p>Which of the following statements best describes your knowledge about the characters package for Dart before participating in this study?</p> <ul style="list-style-type: none"> ● I didn’t know this package existed before the study. ● I heard about it but knew little about what it does. ● I knew the package and the problem it solves, but I haven’t used it. ● I’ve used the characters package in my own projects. 	
Q47	<p>What is your level of experience with programming in Swift?</p> <ul style="list-style-type: none"> ● No experience ● Awareness ● Novice ● Intermediate ● Advanced ● Expert 	
Q48	<p>How often did you have trouble understanding something in this survey (e.g., questions, instructions, or descriptions)?</p> <ul style="list-style-type: none"> ● Always ● Often ● Sometimes ● Rarely ● Never 	
Q49	<p>What was unclear in this survey?</p> <p>_____</p>	Logic: show if the answer to Q48 is not “Rarely” or “Never”.

Integrating a Live Programming Role into Games

Krish Jain

Lake Washington
School District
Redmond, Washington 98052
USA
s-kjain@lwsd.org

Steven L. Tanimoto

Computer Sci. & Engineering
University of Washington
Seattle, Washington 98195
USA
tanimoto@uw.edu

Abstract

Web-based games can permit players to take on multiple roles, and in the past such roles have generally been defined in terms of characters in game narratives. In this report on early work, we propose adding a live-programming role to games that may involve the kind of problem solving that requires “thinking outside of the box.” The live programmer can be empowered by the game designers to bend the rules, within certain bounds. We demonstrate the concept using a prototype multi-role game in which players must bring Covid-19 outbreaks under control by performing a sequence of pre-designed actions. The live programmer is able to adjust parameters of the actions, and even disable actions or create new ones. We suggest that having the live programming role in such a game can foster learning about the game domain and structure in different way than usual game playing or modification. Such a live programming role may also be appropriate in some simulation environments and emergency management systems. Finally, we discuss several issues raised by the existence of the live programming role: player power and fairness, “live scripting” (one form of live programming), and characterizations of game sessions in terms of evolution of game state versus evolution of game state plus code versions (“full trajectories”).

1. Introduction

Live programming may occur in a variety of contexts. One is musical performance, in which there is typically one programmer writing code in front of an audience, and the code is controlling a music synthesizer, responding to changes in the code during the performance (Aaron & Blackwell, 2013). In another context, a programmer working on creating a dynamic web page writes HTML, SVG, or Javascript code in one pane of browser window, and in another pane, a graphic is shown that has been produced by the code. The objective is fast development of the graphics, so that the SVG can be incorporated into a new web page, especially a web application (JSFiddle-Staff, 2020).

Another kind of context for live programming is what we refer to in this paper as “live scripting.” Here, we have a programmer who is editing code that affects the running of an activity. For example, that activity might be an architectural CAD session, in which an architect is drawing a CAD model or a blueprint for a building. The programmer, who may be a different person from the architect, is editing a script that controls how an alignment tool works. For example, the tool might be reprogrammed to cause drawing elements to snap to various horizontal positions on the basis of a hierarchy of attraction preferences involving nearby objects and the current distances to them. The live programmer (live scriptor) here is playing a secondary, behind-the-scenes, role that is supporting the architect.

The meaning of the word “live” in such live scripting can be understood to be “concurrent with the activity being modified.” Here “activity” is not necessarily to be interpreted as the computer executing the edited code, as it typically is in live programming situations. The activity is more general and involves, in our architectural example, a session in which a human user (or several users) are interacting with a computational system to accomplish a task, such as designing a building. The live scripting is taking place while the designing is happening. If the scripting were not live in this sense, then it would have to be done in specific time intervals during which other activities such as drawing the blueprint were suspended or allowed to continue but determined to be independent of the scripting. We further discuss the notion of live scripting in section 4.9.

Live scripting has several potential benefits. One is the usual live programming benefit of reduced

latency between programmer action (e.g., editing) and programmer understanding of the consequences of that action. Another is allowing tooling limitations to be fixed without requiring the users of the tools to start their designs or other projects over again. Related to this is the possibility of real-time interaction between the programmer and the user, in the context of the session, so they can cooperate to achieve the joint goal of a successful design and improved tool.

This live-scripting form of live programming could play an important part in education and training, especially when the programmer needs to learn how a certain type of system works and can be modified. One of us has made the case that future systems for emergency management could benefit from facilities for live programming (Tanimoto, 2020). An important component of an emergency management system is a subsystem for conducting training exercises. A live-programming component could be used both to help create new training exercises and as the target of training exercise – i.e., for bringing a new live programmer on-board as an emergency response team member.

With that kind of scenario in mind, we developed a simple collaborative game that very roughly resembles an emergency management system which offers multiple user roles. Within that scenario is a special, unconventional role: a “live programmer.” (Now that we have made a distinction between live programming in general, and live scripting, we will revert to the more common terminology of live programmer for the game role to be described; it is really live scripting in the above sense.)

The live programmer is empowered to edit program code that affects the running of the game itself. The overall system, which includes the facility for live programming, is set up such that more traditional game play and live programming not only can happen concurrently, but such that the changes made by the live programmer can take effect immediately, sometimes altering the course of the game, and without requiring players to restart the game.

2. Related Work

The design of programming environments has a major impact on the experience of computer programmers (Edwards, Kell, Petricek, & Church, 2019). In addition, the applications context and social context also have an impact on the programmer experience, especially with novice programmers (Guzdial, 2015).

In addition, by supporting “live” programming, a development environment can enable an interactive programming experience that is “tighter” and that can enhance either the process or the end result of a programming session (Victor, 2012) (McDirmid, 2013) (Church, 2017).

Liveness can be incorporated in a variety of styles within an environment, from full programming to merely parameter tuning (Kato & Goto, 2016). For more literature related to liveness, see the references in some recent works (Petricek, 2019) (Kato, 2017).

Programming is important nowadays not only for creating software products, but to control or modify the behavior of systems that already come with a lot of software. In a domain such as emergency management, it can happen that new information-processing needs arise, or preconceived models need to be modified, and computer code may be the best vehicle for achieving the change.

Past work on emergency management systems (EMS) includes work from the decision-control systems standpoint (e.g., (Turoff, 2002)), and the commercial enterprise developers of EMSs such as WebEOC (Juvare-Inc., 2020) and intergovernmental agencies (World-Health-Organization, 2013).

3. The Game Context for Live Programming

Next we describe the specific game we developed that serves as a computer-based activity context for the live programming. The game is collaborative, providing explicitly named roles for multiple players, and allowing each player to join a session over the Internet. The basic game is turn-based. However, there is a special role of live programmer (LP) which is not strictly turn-based but more loosely synchronized with the activities of the other roles.

3.1. A COVID-19 Pandemic Management Scenario

We designed and implemented a game in which a collaborating team of players work to get an imaginary (and extremely simplified) version of the COVID-19 pandemic under control. The game is implemented within an online client-server software framework called SOLUZIONE, which is described in more detail in section 4.3. Figure 1 shows a screen shot from the beginning of a session. For some of the details about the game, please refer to Video 1 of the two associated with this paper. (URLs to the videos are given in the appendix.)

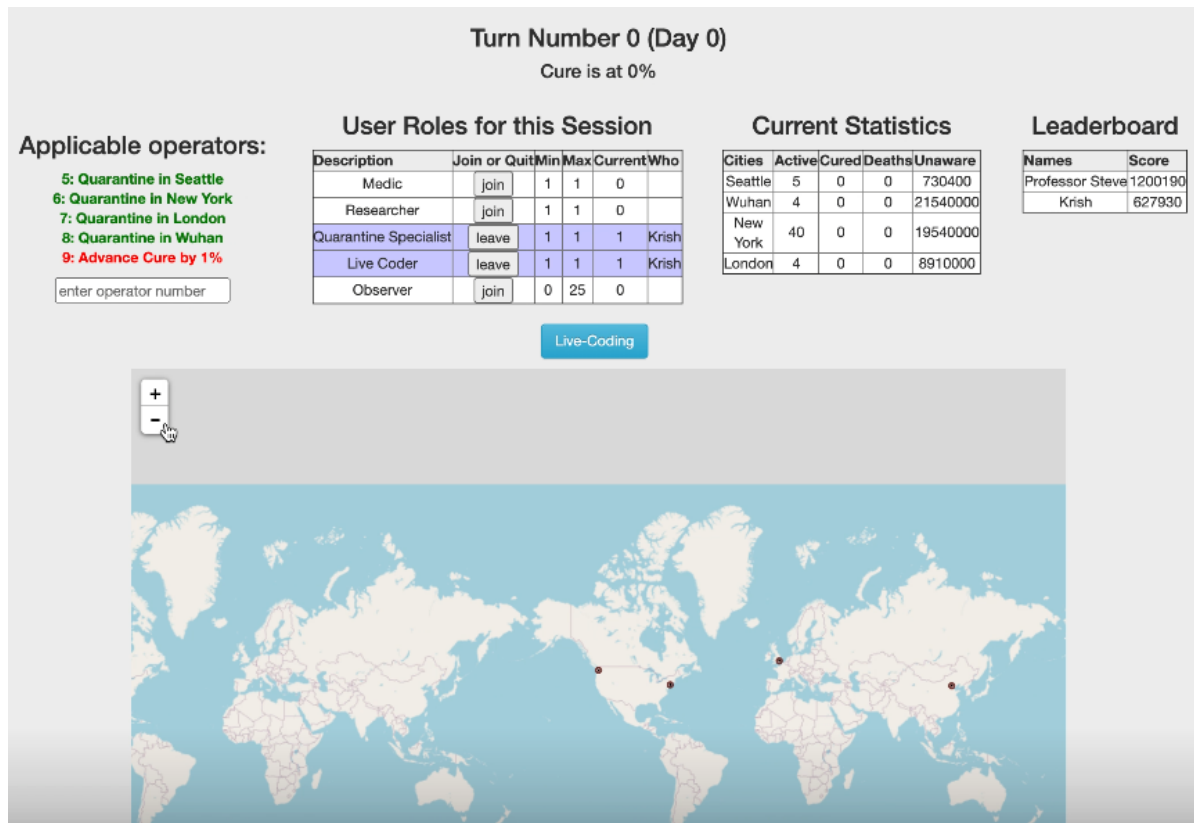


Figure 1 – Screen shot at the start of a game of pandemic management.

3.2. The Live Programming Role

We invented a special role for the game in order to begin to explore the possible ways in which live programming could be meaningfully integrated into games.

The progression of the game itself is controlled by a Python program (the formulation file) that operates within an enclosing client-server framework implemented in Javascript and Python. The formulation file specifies an initial state for the game and a set of game operators that can be applied by players to change the current state of the game.

The live programmer (LP) in the game is empowered to edit the formulation file such that the functions contained within operators change, and/or new operators are created. The effects of these changes occur at the very next turn in the game after the LP clicks on “Save.”

The framework requires the LP to have a basic knowledge of Python programming, as well as of the relationship between operators and states. As to not overwhelm a novice programmer, the framework presents two options: a novice version and a complete version. The first comes with suggested events for the LP to follow so that the LP becomes familiar with the code. After developing a thorough understanding of the framework, the LP can choose to use the complete version to edit all the code in the program. In the context of this simulation, the LP can present novel ideas into the mix to better predict outbreaks. For instance, the LP could split an existing city into two to further illustrate the idea of quarantine or

create a new operator for the other roles to act upon. After introducing a new parameter to cities, the change might not be immediately visible, but over multiple turns, it will be observed that the growth of the virus will slow due to increased borders.

For more details about how the LP role works in the game, as well as its technical implementation, please see Video 2 of the two videos associated with this paper. (Again, URLs to the videos are given in the appendix.)

The game has been play-tested on a very limited basis at the time of this writing. However, the trial game showed that the LP could respond to needs arising in the game at scripted intervals, to improve the rudimentary COVID-19 propagation model to more accurately predict outbreaks and help the team score better in the game.

If this game represented an actual emergency management system, and there were a LP who could make functional changes to the system in response to new need during a developing crisis, the system and its team of operators might be better able to manage the crisis.

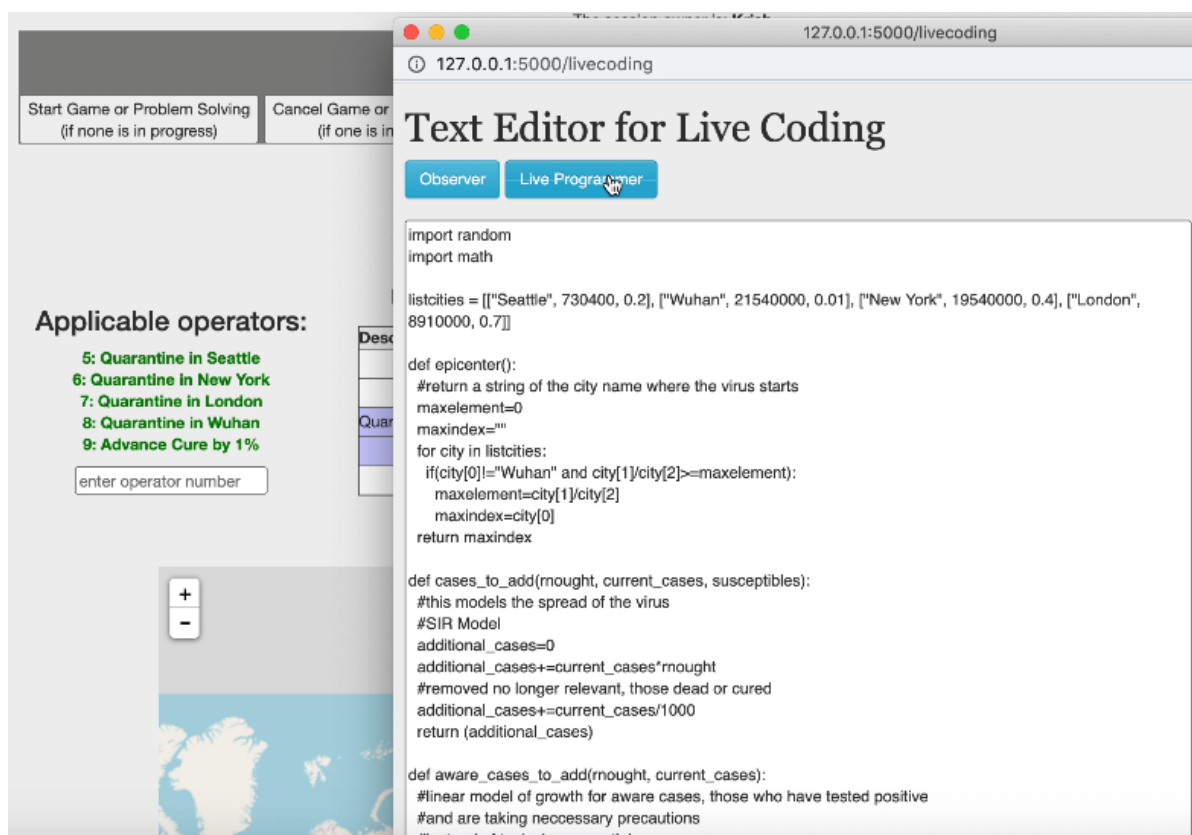


Figure 2 – Illustration of a live programmer's screen.

4. Future Work and Discussion

4.1. Future Work

We are considering several possible follow-on developments in this project: enhancements to the game to make it somewhat more realistic, studying the LP role in relation to other roles, and making live programming an essential part of playing and winning the game. In this next section we consider issues primarily related to the second of these (understanding and reshaping the LP role in relation to the other roles).

4.2. Discussion Overview

This section brings up a variety of issues germane to having a live programming (LP) role in a game. The first issue is the relationship between the LP role and the other game roles. This leads to a discussion

of how much power the LP has, and how it might be limited or expanded for various purposes. Playing the LP role typically requires more knowledge than playing other roles, and we next discuss what that knowledge is and how the requirements can be reduced. These same issues are affected by the purpose of the game in relation to application domains, which are then discussed. Then we return to the issue of live scripting vs. live programming generally, which was laid out at the beginning of the paper. Finally, we consider the issue of “divergence” which can be problematical in some live programming contexts; it turns out not to be a concern for us when our purpose is supporting LP for the sake of enriching game play, but it remains potentially problematical if the purpose of LP during the game is to improve the formulation of the game.

4.3. Computational Model

In order to discuss some of the more interesting issues associated with this work, we first present some background on the SOLUZIONE framework as well as a general model for a SOLUZIONE game that includes live programming, such as our game presented in this paper. One example of the model’s use is that it will help us explain the possible application of this sort of game to emergency management training.

The SOLUZIONE framework permits a game designer to specify (using the Python language) an initial game state and a set of operators, as well as a set of player roles. For example, we have a SOLUZIONE formulation for a 4-disk version of the Towers of Hanoi puzzle, with a single player role, “Solver.” The initial state contains a representation of a platform with three pegs, and 4 disks of different diameters, with holes in their centers, piled up on the leftmost (first) peg. There are six operators, with names such as “Move the topmost disk on Peg 1 to Peg 2.” A player takes a turn by selecting an operator that is applicable in the current state. The computer then applies the operator, which updates the state and the players’ displays (over the web, in their browsers). If the game has multiple roles, then turn taking is typically managed through the current state having a variable whose value specifies whose turn it is. A game session begins with the players adopting roles. SOLUZIONE does not prevent a user from taking more than one role, but normally each player will take one role in a session. When the session owner (the user who first connects to the game server after it has been started) starts the game, the SOLUZIONE system running on the server puts the game into its initial state. That specifies which player(s) may move first, based on the Python code in the game formulation file. Players take turns making moves until the game ends, typically when a “goal” state is reached. In the Towers of Hanoi, the goal state is the configuration of disks in which they are all piled up on the rightmost peg.

Thus a typical SOLUZIONE game session can be characterized as a sequence of moves: $\langle \mu_1, \mu_2, \dots, \mu_n \rangle$, which induces a corresponding sequence of game states $\langle \sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n \rangle$. Here σ_0 represents the initial state, and σ_i for $i > 0$ represents the game state that results at the completion of move μ_i . Every game state σ_j is the result of zero or more applications of game operators. (Each operator application is one move.)

The incorporation of a live programming role makes the basic SOLUZIONE model insufficient to characterize a game session. This is because after the game starts, the set of operators may change at any time, due to the live programmer’s editing. Existing operators may have their names changed, their preconditions changed, or their state-transformation functions changed. They may also be deleted. New operators may be added. In order to describe the evolution of the set of operators, we’ll use a sequence of code versions: $\langle \Xi_0, \Xi_1, \dots, \Xi_m \rangle$.

The symbol Ξ_0 refers to the problem formulation file at the beginning of the game, before a live programmer has made any changes. Each time the live programmer edits and saves the formulation file, the set of available operators changes from, say, the one specified by Ξ_i to the one specified by Ξ_{i+1} . The relative timing of LP code-saving steps and player moves is important. The *full trajectory* of a game session consists of a sequence in which game moves and LP save operations are interleaved (though not necessarily in a one-to-one fashion). Such a sequence can be represented as $\langle \zeta_0, \zeta_1, \dots, \zeta_\omega \rangle$, and each ζ_i is either one of the μ_j or one of the Ξ_k . These sequences are ordered by time, and the full tra-

jectory should have associated with it a sequence of time stamps, $\langle t_0, t_1, \dots, t_\omega \rangle$, such that in the game session event ζ_i happened at time t_i . The full trajectory, together with its sequence of time stamps, gives an accurate representation of what the players did during a game, insofar as the evolving game state and formulation changes are concerned. Such a record makes it possible to recompute the sequence of game states $\langle \sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n \rangle$, in spite of the live programmer having made changes to the operator definitions throughout the game.

When we discuss the possible application of these games to training emergency management personnel, the full trajectory is a fundamental object that might be evaluated to provide educational assessment relative to the desired training.

4.4. Live Programmer Role Relationships

We designed our game to have a live programmer (LP) role with the assumption that this LP role would be one more role for a team of collaborators. Although we tell players they are working together in a team, the software does not prevent them from either fighting over resources or getting in each others' ways; e.g., the live programmer changing something that another player has been counting on, such as the way funds are distributed when they arrive from governments. We have been assuming that a team wants to win and will try to avoid these conflicts.

Our assumption is not necessarily warranted — that the LP will make positive contributions toward the team's achieving the game goal of bringing the pandemic under control. The LP may have the ability, within the game's software, to help, but that does not mean the LP will know how to help or even actually wish to help. The LP could intentionally work against the efforts of the rest of the team, whether instructed to help or not. Also, the LP could break the smooth operation of the game by causing a semantic error, say, that sets the game into an extreme state. (Merely syntactic programming errors will block the activation of new code versions that contain them and are fairly harmless.)

This leads to the concern, whether the LP is well-intentioned, or sufficiently skilled, or not, of whether the other players trust the LP. If they are aware that the power of the LP is limited, this concern can be ameliorated. If they trust the intentions of the LP, and they can be confident in the skill of the LP, the concern can also be ameliorated. To reduce the worry on the part of other players, game designers can do three kinds of things, and we are considering doing these to our game: (a) putting more explicit limits on what the LP can do; (b) providing better training to the team, including helping teach the LP to be competent, and training the other players about how to communicate with and make requests to the LP; and (c) share the LP's power among players by rotating the LP role within a game. We consider each of these directions in more detail in subsections 4.5-4.7 below.

Before that, we discuss the particular role of “observer,” for which we haven't provided any operators for making moves in the game. Our “Observer” role was originally created to support audience members who might wish to see the code being edited by the LP, as has been popular in live coding performances with music (Sorensen, 2005). As implemented now, the observer role can serve that purpose, so online sessions with the game can have audiences. However, the observer role can also serve as a supplemental affordance for any player other than the LP. Thus the Medic is allowed to not only be the Medic but also be an observer of the LP, and see the code that the LP is editing. Finally, we have neither intended it nor tried it, but the observer role could support pair programming in which the observer acts as a code navigator (communicating with the LP via Zoom, Skype, etc., or in person), while the LP does the editing.

4.5. Limits on The Live Programming

In our current implementation of the game, the LP is free to edit any part of the problem-formulation file, which specified the initial state of the game, and the operators that are available to the players to make moves. Let us now consider placing easily understandable limits on what the LP can do.

The game operators could be partitioned into two groups: mutable and immutable operators, such that the LP can only edit the mutable ones; which operators are which could be made known to all players.

Thus players who have learned what the immutable operators do can be assured that they will not change and they will probably not lead to surprises.

Alternatively, the game's state variables could similarly be partitioned such that the LP's new code cannot alter the existing behaviors with regard to the protected variables. This would be more difficult to enforce while still keeping the existing LP programming mechanism intact, because an operator is generally allowed by the SOLUZIONE framework to update the state in an arbitrary way. However, if there were a "budget" variable in the game, players might be reassured to know that it could not be tampered with by the LP; or perhaps they would be disappointed to know that.

Even more restrictive would be a constraint that the LP only create new code for a specific mathematical equation that is part of the embedded simulation model. This function could be prevented from having side effects or from accessing any but pre-selected state variables. In this way, the players could rest assured that game logic would not change as a result of LP errors, creativity, or mischief, and yet the LP could have a well-defined means of contributing to the game.

A different form of restriction would be to limit the LP's changes to the turn-taking logic of the game. It might seem like a good idea to a team to let the medic perform a succession of vaccination operations before the researcher or quarantine specialist get to proceed with a new move, and this could be enabled on a one-time basis by the LP, or whenever some particular condition is true in the current state, such as a vaccine being available and millions still need it.

This last example illustrates one sense of the phrase "bending the rules." The initial turn-taking order gets modified in response to the needs of simulation, in a manner not expected in games. Another example of a LP bending the rules is modifying a requirement that the Medic and Researcher never overspend their budgets, such that players can go into the red financially, at least temporarily, to a certain extent. Postponing a deadline or allowing one to be missed is another example of this sort of update that seems inconsistent with the original game formulation but adds a degree of realism. Allowing larger groups of people to be vaccinated in one turn, or cured of the disease, is another example of bending the rules – altering the dynamics of the game in a manner unexpected at the start of the game.

An interesting possible variation of the limitation on LPs according to which game operators they might edit is the following (which we have not implemented but are considering). In our game, each operator is associated with one of the non-LP roles: Medic, Researcher, Quarantine specialist (M, R, Q). By structuring game turns to follow an interleaved order (LP, M, LP, R, LP, Q, ...) and restricting the LP edits to only operators of the role next up, players would have an understanding that the LP is regularly providing coding services for each of them. Furthermore, each non-LP could specify through a menu, which operator they would permit the LP to edit. This might not prevent mischief, but could give clearer expectations about the role relationships than with the LP having complete freedom.

One more way to limit what the LP can do is to limit edits on operators to only the pre-condition portion of operators. This is a predicate that defines the scope of applicability of the operator. For example, undertaking a Covid-related research study may only be allowable in the game when 2 million dollars are available in the current game state. However changing that precondition to allow the study when only 1 million dollars are available widens the scope of applicability without altering the portion of the operator that changes the current state.

Before we leave the subject of protecting a team from the potential ravages of a rogue LP, we note that having an uncooperative or ill-intentioned group member is a problem for cooperative teams of all kinds, and not to be blamed on the existence of an LP role. However, the difference in power between a normal game role and the LP role justifies some special emphasis on the issue.

4.6. What Players Need to Know

Incorporating a live programming role into a game imposes some additional demands on the person filling that role, and can also raise expectations on other players. Let's consider what the LP role may require and then discuss effects on other players.

In our game, the live programmer needs three kinds of knowledge: (a) programming in Python, (b) the structure of a SOLUZION problem formulation, including some elements of the “classical theory of problem solving,” and (c) the specific effects of the game’s operators. If we add limitations on what the LP is permitted to change or add, then the LP needs to be aware of these limitations, as well.

The Python source code in the game’s initial problem formulation is simple enough that advanced knowledge of Python is not required by the LP, in order to read and understand it. In addition, modifying this code is not expected to require advanced Python knowledge either. For example, new class definitions are not required, although modifications to the given State class, could be useful; adding a new state variable, through an assignment statement such as `new_state.num_virus_variants = 3` is allowed in Python, even if the State class’ `__init__` did not set up any `num_virus_variants` member variable. If an LP wishes to use advanced Python features, however, there is nothing that we have put in the game to stop that.

The LP needs to understand that our game is implemented within a software framework (SOLUZION) that requires all potential player actions to be implemented as “operators” that may transform a data object, known as the “state” to effect moves or progress in the game. Each operator has three components: a textual name used in the game’s human interface, a “precondition” function that determines whether the operator is allowed in the current state, and a “state-transformation function” that maps the current state to a new game state when the operator is used. Learning this structure is relatively easy, say, in comparison with learning to program. A new LP should either be given a 15-minute personal introduction to SOLUZION and the existing problem formulation code or watch a short video.

The third kind of knowledge needed by the LP is an understanding of what the existing formulation’s operators do, and how they do it. This can be learned through a combination of the tutorial (which should combine an introduction to SOLUZION with information about how the existing operators work), game play – to see the operators put into action by the players, and examining the source code of the problem formulation. Comments within this source code assist the LP in this regard.

While the knowledge requirements for the LP role may seem formidable, one can argue that they are not so bad. Programming ability and fluency is more and more common, as computational literacy is taken seriously by K-12 educators. The SOLUZION structure is intentionally simple, almost minimal in its requirements, and the given problem formulation file is already in the proper form, so an LP never has to come up with a formulation file from scratch. The existing game is quite simple, and does not involve complicated code in its operators. That said, in the future, we could further simplify what the LP needs to know through one or more of the limitations mentioned earlier, such as limiting the changeable code to one particular mathematical function used in modelling the pandemic’s spread.

What the non-LP players need to know is a little about their own roles (e.g., Medic, Researcher, Quarantine Specialist) in controlling a pandemic, how the basic game mechanics work (turn taking) and enough about the LP role to either trust the LP (if possible) or have a justified mistrust.

Game events must be understandable to all players. Certain events take place after a pre-specified number of turns have been taken – this could be a new outbreak of the disease, the discovery of a new variant, or a breakthrough in vaccine development. Players should also be able to read game state variables as shown on the screen and know generally what they represent.

Prompts to the LP need to be understandable, and understandable in terms of the problem formulation elements – current state, game operators including their preconditions and state-transformation functions. A LP might benefit from having a “cheat sheet” about how to get started in responding to such prompts.

4.7. Power Sharing

An alternative to greatly restricting the free-ranging power of the one LP in the game is to somehow distribute that power more evenly among players. Here are some ways we might go about re-designing the game for that.

Rotation of the role of Live Programmer could be incorporated and enforced, so each player who wishes to do so could have the chance to perform live programming as part of his/her regular turn. Such a policy might comport well with the earlier-mentioned constraint that such live programming be limited to editing only the operators associated with the role whose turn it is. This suggests all players would have to be able to program in Python to take full advantage of this policy.

This rotation approach is not very much different from saying that all roles in the game should be considered LP roles. Then the operators become simply a means of ordering edits into turns. A player would take a turn by making some edits as a LP and then applying an operator to signal the end of the turn. Whether players would be allowed to perform editing outside their turns is a design decision that might depend on how quickly players are expected to make changes, and how their edits might be restricted to operators they “own.” If their editing time intervals and program scopes are allowed to overlap, then conflict-resolution methods might be needed.

A game with this many LPs might require more elaborate scaffolding to keep them all on track. Once again, each could be limited to specific functions, operators, or formulas.

Finally, we could achieve an approximation of this sort of power sharing without making any change to the current game implementation as follows. We would ask each non-LP player to take on both a regular role (Medic, Researcher, Quarantine specialist) and an observer role. There is no limit on how many observers there can be. We then instruct each player to be part of the “programming committee” by using techniques of pair programming, triple programming, etc., to assist the LP in making the changes needed by the whole team. We might call this manner of playing the “co-programming” game strategy. Co-programming this way avoids certain editing/versioning conflicts that could arise if all players were LPs simultaneously editing the problem formulation file. That is another possibility, however.

4.8. Contrasting Application Domains

We imagine three types of applications for the techniques used in our game to combine the LP role with the rest of the application: (a) training of live programmers to modify existing code in the context of a running activity, (b) game design, during which the modifications introduced by the LP get immediately tested in subsequent turns of the play-testing session, and (c) entertainment.

Training a live programmer involves not only helping that person master the three kinds of knowledge listed in Section 4.5, but giving the LP experience in communicating with a team in the context of solving a problem. Emergency management is one such context (Tanimoto, 2013). Such communication requires that the LP not only understand much of the existing code, but be able to discuss its structure and functionality with team members who might have next to no programming knowledge. This could mean that the LP learns to help the non-LPs develop mental models for the code functionality and for the challenges of the coding process.

Game design differs in a fundamental respect from training live programmers. The actual sequence of state changes during game design is of much less interest to the team than obtaining the final version of the problem formulation file. The game-play session, with the live programming role, is a means to an end in which the final game formulation Ξ_m will represent a game that no longer needs a live programmer, since the tweaking of game rules will have been completed.

A third aim for prospective designers of our type of game is simply to create entertaining challenges for players. Entertainment aspects include game storyline, problem/puzzle solving, and the social aspects of collaboration, or possibly competition.

Our main interest is the first category of applications. Live programming may turn out to be important in domains such as emergency management, or large control-system software maintenance (e.g., transportation systems, nuclear power plants, space stations, etc.) An example from emergency management where a live programmer may be required is patching a system for earthquake response management to accept a new format of map data (say, produced by a new model of drones), so that the data can be integrated with the existing maps to show locations of drones or beacons. The software must keep

running to keep supporting current rescue missions, but the new functionality needs to be added.

The game context can be helpful for training live programmers, as the environment may feel safer or more welcoming, or just simpler as a first step. A veteran of the LP role in our game may have gained enough confidence to consider working up to a LP role in, say, managing an earthquake response or other emergency.

To create an effective training tool for emergency management (EM) through a reworking of our game, not only should an initial problem formulation provide a good starting point for an EM exercise, but the full game trajectory described in section 4.3 should be automatically captured and analyzed, so that pedagogical feedback can be given to the players about their responses to particular game situations. This should include an analysis of the code changes made by the LP to determine which were effective, which were ineffective, and why or why not. Such a game and analysis system could be augmented with in-game questionnaires to further interrogate players on their beliefs and the reasons for their in-game choices, leading to even more accurate assessments.

A complex phenomenon, such as the evolution of a Covid-19 pandemic, would be modeled much more accurately in a computer simulation based on high-fidelity data sets and mathematical formulas, than in a simple turn-based game. One can imagine that a complex, scientific computational model for the pandemic could be set up to have a live programmer making adjustments to it while running. That might make sense if the simulation requires long run-times during which some of the assumptions on which it is based turn out to change. In that scenario, the LP would need deep knowledge of the computational model, but would perhaps not have to know about any game roles or much about other users of the system. That scenario is also a possible application of our integrated LP role, but unless the running simulation is also driving policy decisions with real-life consequences, this kind of system doesn't seem subject to the same risks as emergency management systems face, and so the safety and security issues are less in focus.

The distinction between game and simulation, already tricky, is further eroded in a game with a LP role. After all, isn't a game just a simulation with rules about what players can do? If the rules can change or disappear, what's left might just look like a simulation. Of course, our LP role can also change the simulation, or make it disappear, too. But by incorporating an LP role, we admit to "fuzzing the rules of the game," and thus we blur the line between game and simulation.

Yet one more possible application of our style of game is to support general learning and meta-cognition. The players' task could be thought of as having to quickly learn a set of rules and then track the rules as they change. This could prompt students to reflect on how game structures guide their own thinking as they live their lives, and how they might reshape those guides as their lives unfold.

4.9. Characterizing Live Scripting

The distinction we made at the outset is that live scripting is a special type of live programming in which an ongoing activity which involves computation has human users whose experiences are being affected by the programming as it happens. We further discuss this distinction here.

The category of programming commonly called "scripting" typically refers to developing relatively small programs that work with larger, existing software items either to (1) "glue" them together, as a Unix shell script might invoke a data-processing program and then pipe its output to a graphing program, or (2) customize a large software application, such as Microsoft Word, adding new functionality through Visual Basic programs. Customizing applications such as Gnu Emacs is accomplished by writing scripts in Emacs Lisp. However, it requires a much greater level of technical proficiency to script Emacs effectively than to simply use it as a text editor. Efforts have been directed to providing scaffolding for would-be scriptors, e.g., in the form of customizable buttons in a Xerox Lisp environment (MacLean, Carter, Lövstrand, & Moran, 1990). That group stresses the need for a culture of software tailoring in addition to any special features in the environment.

In our game, the scripting can also benefit from a culture of tailoring, so that the live programmer can

feel fully supported by the rest of the team in making changes to the game’s formulation details. This culture is relevant regardless of whether game players are simply trying to win, to improve the game, or to learn about live programming or game structures.

In the context of collaborative design of software systems, including games, the argument for users and designers working together has been made by Bødker and Gronbaek (Bødker & Grønbaek, 1991). That teamwork situation is analogous to game players and live programmers working together to improve the effectiveness of game operators. Our use of the term live scripting is somewhat apt for the sorts of co-design in which users and programmers are testing and coding with a process of many very short cycles of editing and testing. However, we would argue that scripting is really live when the cycles form a continuous flow, and it is not necessary to explicitly restart tests whenever an update to the code is made.

An additional example of live scripting is a collaboration between an advertising layout specialist and a CSS specialist. They have a shared screen in front of them. The JSFiddle website (JSFiddle-Staff, 2020) is up. The CSS specialist is editing the CSS code as the ad specialist provides feedback about the look. In this case, the result will be CSS code that then goes into regular use on a corporate sales page. A variation of this involves a color-blind reader instead of the ad specialist, but again with the CSS specialist. The result of this session is a customized styling of a the material, optimized for the individual or for a relatively narrow set of viewers. In either case, the scripting is part of a close collaboration between concurrent participants one of who edits code.

Live scripting for design can be considered a subclass of co-design processes in which we have a specific LP role, and at least one other role whose job it is to test and/or use the computational affordances being changed by the LP, and in which liveness plays an important role. The liveness eliminates or reduces the need for the non-LP participant to repeat completed steps, due to having to restart the software. This definition may not always lead to a clear distinction between live scripting and other co-participation processes, but it seems to capture the nature of the relationships among the roles in our game, while also being applicable to other systems.

4.10. Divergence of Trajectories – Game play vs Program Execution

One of the sticky technical issues around live programming in a software development context is the “divergence” that can arise between the execution state (which typically can depend on past versions of the source code) and the current version of the source code (Basman, Church, Klokose, & Clark, 2016). If users wish to be able to re-create an arbitrary execution state from a session, it may be infeasible, because the program’s source code has been altered by live coding, and older versions have not been kept. Another aspect of divergence is that the continuing execution of a program that has been live-modified is no longer guaranteed to be representative of the code in its latest version. Thus the final code version Ξ_m from the session, cannot be accepted as an adequate formulation of the game solely on the evidence that the final game state σ_n from the session is a desired final state. The continuing execution is dependent both on code that exists and on code that existed in the past but no longer exists.

One way of preventing the loss of access to early execution states is to build into a system facilities for “remembering.” One is to start by backing up the original program formulation file, and then logging every live-coding edit and every user-input event, and time-stamping them in such a way that the complete session trajectory can be replayed. Another is to build-in state-externalization methods that allow check-pointing the evolving session state at an arbitrarily fine temporal resolution. Such techniques can be useful not only in addressing the divergence issue, but also to enable flexible error recovery when live programming results in undesired execution states, such as errors or unexpected deletion of state data.

In our game, as in other SOLUZION framework games, all relevant execution state for a game session is embodied in the instances of a game “State” class, which, defined in Python, is easily externalized as a JSON text string, although we have not yet had sufficient reason to implement that. The sequence of formulation file versions, represented in section 4.3 above as $\langle \Xi_0, \dots, \Xi_m \rangle$, could be written out to file storage, with a new file (with index i as part of the file name of the i -th version), after each LP

save operation. Again, we have not implemented this. But the fact that we could do this suggests that divergence in the sense of Basman et al should not be seen here as impugning the incorporation of the LP role.

If it is desired that the final formulation file Ξ_m be a means to produce the final game state σ_n without need of old formulation versions Ξ_0, \dots, Ξ_{m-1} , then an additional auto-replay mechanism should be added to the SOLUZIONE infrastructure to permit any move sequence prefix $\langle \mu_1, \mu_2, \dots, \mu_i \rangle$ to be used as a regression test on the latest Ξ_j , and if ever a desired intermediate state is not attained, then the latest formulation change should be undone and replaced with a formulation that achieves the correct state before proceeding further in the game. Such a process may be worthwhile in a game-design context, but might lead to player confusion if added to our game.

In our game scenario, as in musical live-coding, one can consider that the possible disparity (between the current game state and the game state that would exist if the the game were re-run from the beginning using only the latest versions of the operators) to be “part of the game” and not usually of concern, since the live programmer is trying to help control the evolution of the game state rather than produce a new piece of software.

Provided that an original problem formulation file is not thrown away, we maintain the possibility of playing again along any particular full game trajectory, even without a recording of the original session. However, to replay an entire game the same way, under evolving rules, the players, including the live programmer, would need to perform the same actions and edits as happened in the earlier game, and with very similar relative timing. Some ineffective edits and redundant saves could be skipped, but the essential code updates and player moves would have to take place again in order to produce the same state sequence.

4.11. Safety, Learnability, Empowerment, and Engendering Trust

We discuss a few more topics before closing: safety, learnability, technical empowerment, and how to engender trust within a team having a live programmer.

By empowering one user (or a subset of users) to modify a running system or the rules of an in-progress game, there is a danger that the system will break or become worse off than before any live programming. This issue has been discussed by the second author elsewhere (Tanimoto, 2020). Using a game context rather than a real emergency management situation can reduce the risk of real damage while facilitating education and training about live programming and effective collaboration.

The learnability issue is particularly important in the emergency management context because it is relatively likely that less-than-fully-trained personnel may have to come on board to help deal with an emergency. Liveness in a programming environment can help a programmer new to that environment in coming to understand the possible effects of various code changes and additions.

High technical empowerment for live programming means designing the base system in such a way that a live programmer can change much of the system functionality. One way to achieve this is for the base system to be written in a language such as Smalltalk that supports inspection and modification. In fact, a commercial data analysis system known as Analyst was written with this in mind at Xerox PARC in the 1980s (Thomas, 1997) (Xerox-PARC, 1987). Earlier, we discussed ways to limit such power, but it's worth mentioning that expanding the power is sometimes of interest.

Finally, trust and collaboration, especially important during emergencies, can be fostered through training exercises, including games. A live programmer (and in particular, a live scriptor), working in a collaborative situation, a game or not, should be trustworthy. The observer role (as a supplement to the roles present in the game) allows a player watch the code in real time, serving as a referee and a trusted adviser for all the changes that the live programmer is making. This also allows the observer to function as a deterrent to the live programmer abusing the premises of the framework. Often, the live programmer only has the technical expertise to code, but not the expertise to decide what pressing issues need to be fixed; in this specific simulation, the live programmer would probably lack the medical expertise needed

to decide what needs to be changed. Through observing the code, the observers can better inform the live programmer on the action that needs to be taken. There is much more to these issues, but we leave the discussion here.

5. Acknowledgements

The authors would like to thank the organizers and insightful reviewers, including Mariana Mărășoiu, Luke Church, Colin Clark, Philip Tchernavskij, and those anonymous, as well as the play-testers who tried out the game. We also thank the PPIG December attendees who provided feedback after our presentation.

Appendix

There are two “.mov” format videos that support this paper’s narrative. They are available at this time via the following URLs. Each is approximately 6 minutes in length. The first describes the collaborative game referred to in the paper, and the second describes how the live programming role works.

<http://xanthippe.cs.washington.edu/ppig20au/PPIG-Video1.mov>

<http://xanthippe.cs.washington.edu/ppig20au/PPIG-Video2.mov>

6. References

- Aaron, S., & Blackwell, A. (2013). From sonic pi to overtone: Creative musical experiences with domain-specific and functional languages. In *Proceedings of the first acm sigplan workshop on functional art, music, modeling & design* (p. 35–46). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2505341.2505346>
doi: 10.1145/2505341.2505346
- Basman, A., Church, L., Klokmose, C., & Clark, C. (2016). Software and how it lives on - embedding live programs in the world around them. In *Proceedings of the psychology of programming interest group annual conference 2016 (ppig 2016)*.
- Bødker, S., & Grønbæk, K. (1991, March). Cooperative prototyping: Users and designers in mutual activity. *International Journal of Man-Machine Studies*, 34, 453-478.
- Church, L. (2017). *Becoming alive, growing up*. Vancouver BC, Canada. (Invited keynote, LIVE 2017, workshop at SPLASH/OOPSLA)
- Edwards, J., Kell, S., Petricek, T., & Church, L. (2019). Evaluating programming systems design. In *Proc. 30th annual workshop of the psychology of programming interest group, ppig 2019*.
- Guzdial, M. (2015). *Learner-centered design of computing education: Research on computing for everyone*. Morgan and Claypool.
- JSFiddle-Staff. (2020). *Jsfiddle*. Retrieved from <https://jsfiddle.net>
- Juvaré-Inc. (2020). *Webeoc*. Retrieved from <http://juvare.com/webeoc>
- Kato, J. (2017). *User interfaces for live programming*. Keynote presentation at LIVE 2017, Vancouver, Canada.
- Kato, J., & Goto, M. (2016, 07). Live tuning: Expanding live programming benefits to non-programmers. In *Proceedings of ecoop live*. ACM. Retrieved from <https://junkato.jp/live-tuning/>
- MacLean, A., Carter, K., Löfstrand, L., & Moran, T. (1990, March). User-tailorable systems: pressing the issues with buttons. In *Proceedings of the sigchi conference on human factors in computing systems* (p. 175-182).
- McDermid, S. (2013). Usable live programming. In *Proc. 2013 acm international symposium on new ideas, new paradigms, and reflections on programming & software* (p. 53-62). ACM.
- Petricek, T. (2019). *Histogram: You have to know the past to understand the present*. Retrieved from <http://tomasp.net/histogram/>
- Sorensen, A. (2005). Impromptu: An interactive programming environment for composition and performance. In *Proceedings of the australasian computer music conference*.
- Tanimoto, S. (2013). A perspective on the evolution of live programming. In *Proc. 1st international*

- workshop on live programming* (p. 31-34). Los Alamitos, CA: IEEE Computer Society.
- Tanimoto, S. (2020). Multiagent live programming systems: Models and prospects for critical applications. In *Programming '20: Conference companion of the 4th international conference on art, science, and engineering of programming* (p. 90–96). New York: Assoc. for Computing Machinery. Retrieved from <https://doi.org/10.1145/3397537.3397556>
- Thomas, D. (1997). *Travels with smalltalk*. Retrieved from <http://www.mojowire.com/TravelsWithSmalltalk/DaveThomas-TravelsWithSmalltalk.htm>
- Turoff, M. (2002). Past and future emergency response information systems. *Communications of the A.C.M.*, 45, 19-32.
- Victor, B. (2012). *Inventing on principle*. video. Retrieved from <https://vimeo.com/36579366>
- World-Health-Organization. (2013). *A systematic review of public health emergency operation centers (eoc)*. Geneva, Switzerland. Retrieved 28 March 2020, from http://apps.who.int/iris/bitstream/10665/99043/1/WHO_HSE_GCR_2014.1_eng.pdf
- Xerox-PARC. (1987). The analyst workstation system. In *Xerox special information systems*.

Undecided? A board game about intertemporal choices in software projects

**Christoph Becker,
Tara Tsang, Rachel Booth, Enning Zhang**
Faculty of Information
University of Toronto
christoph.becker@utoronto.ca

Fabian Fagerholm
University of Toronto &
Department of Computer Science
Aalto University
fabian.fagerholm@aalto.fi

Abstract

Software projects teem with choices that have a temporal component – i.e. they involve uncertain future outcomes that are spread in time. In the field of Judgment and Decision Making, such decisions are called “Intertemporal Choice” situations. The board game “Undecided?” simulates a software project through a series of intertemporal choices made by a team decision. It is designed to be educational and fun, but its aim is to provide a platform for cognitive and social studies of decision making in software projects. This paper describes the game and outlines possible research designs.

1. Overview

Software projects teem with choices that have a temporal component – i.e. they involve uncertain future outcomes that are spread in time. In the field of Judgment and Decision Making, which draws on psychology, behavioural economics, neuroscience and other fields, such decisions are called “Intertemporal Choice” situations. In software projects, such choices surface in many areas including technical debt management, iteration planning, personnel development, requirements prioritization, test automation, refactoring, code documentation, and many other issues (Becker et al., 2018, 2017). The temporal distance at which various outcomes occur in such situations has a marked effect on how they are perceived, considered, evaluated, judged, and selected. But even within Judgment and Decision Making, there is no one robust theory about how exactly people really make up their mind in these situations – neither on the individual nor on the group level. We have performed initial behavioral studies to establish the relevance of intertemporal choice in technical debt decisions (Becker et al., 2019; Fagerholm et al., 2019), but these studies have not yet examined just how professionals make their judgment.

The board game “Undecided?” simulates a software project through a series of choices made by a team decision. In each round, the team makes a choice on a next “move” by selecting from a number of cards at their disposal. Each move has uncertain effects in four dimensions (internal quality, external quality, process, and team), some of which are spread in time. At periodic intervals, the team is faced with challenges – meeting thresholds for each dimension; handling unforeseen events; justifying their choices. The game is designed to be fun and educational – we anticipate it being played as part of a moderated workshop and used for a debriefing discussion. But it is also designed to be a platform we can use to study how teams consider the temporal aspects of their choices. The game materials will be made available at cost to anyone interested. An app for hybrid and online play is in development.

In this contribution, we present the game design and outline possible research designs. At PPIG, we organized a gameplay and discussion mini-workshop on multiple boards. Gameplay takes about 60-90 minutes. We then debriefed with the participants; provided a bit more context on the intertemporal nature of decisions and our initial ideas for study designs; and finally, discussed possible applications, extensions, educational applications, and study designs. Below, we introduce key concepts of intertemporal choice, describe the game design, and outline possible research designs for studies using the game to examine how people make intertemporal choices.

2. Intertemporal choices in programming, software engineering, and systems design

“Intertemporal choice” (Loewenstein et al., 2003) is the technical term psychology and adjacent disciplines – behavioural economics, neuropsychology, neuroeconomics (Loewenstein et al., 2008) and other disciplines studying Judgment and Decision Making (*The Wiley-Blackwell handbook of judgment and decision making*, 2015) – give to decisions between uncertain outcomes that are distributed across time. The choices may or may not be explicitly listed and distinguished; and their probability may be clear or less clear. In contrast to the common probabilistic connotation of *uncertainty*, the term

ambiguity describes situations where probabilities themselves are not certain (Camerer & Weber, 1992; March, 1978).

A few examples of software project choices that feature a strong intertemporal component are:

- *Upgrading the software development toolchain* is an effort to improve team and process performance in the medium to long run, but the immediate outcome is missed productivity over the short term, as the team spends time on building infrastructure instead of ‘billable hours’.
- *Test automation* is similarly an effort in improving the infrastructure used to develop, test and deploy software that does not immediately result in features or improved quality.
- *Professional development* of team members or the whole team has no direct impact on the software system under development, but it is certainly intended to have longer-term and far-reaching benefits for everyone involved.

In each of these examples, the delayed effect is assumed to be positive, while the immediate effect is generally seen as a cost to those making management-level decisions. (Whether that is appropriate is a different question, because it may ignore the innate value of such activities as education!) In other situations, the delayed effect is negative.

- *Bugfixing* under time pressure will often involve a choice between a quick solution that runs the risk of introducing technical debt and a thorough approach that carries a higher momentary cost in relation to the single bug, but introduces less technical debt to carry forward.

From the beginning of the software engineering discipline, there have been lamentations about the lack of long-term perspective (Becker, 2014; Naur & Randell, 1969; Neumann, 2012; Parnas, 1994); but the insights from psychology on this very subject have not been actively deployed. For example, the change of valuation of positive effects across time seems to differ from the change that negative effects undergo when they are pushed into the distance, with very interesting effects on so-called ‘mixed outcomes’ (Soman et al., 2005).

The classical, normative view of intertemporal choice takes a rationalistic stance: A discount rate is used to model the difference that time *should* make in the evaluation of possible outcomes from the perspective of an idealized agent. This results in a function that computes a value for an outcome depending on its distance in time. The classic model of discounted utility by Samuelson uses an exponential curve (Samuelson, 1937). Empirical results have often suggested that a hyperbolic curve is a better fit for human behaviour (Frederick et al., 2002). There are however significant arguments against the use of both, summarized in (Fagerholm et al., 2019). In addition, one prominent study (Zauberman et al., 2009) suggested that the concept of mathematical discount functions is entirely misguided, since the human brain does not process time in this way – instead, this study demonstrated that instead of *discounting* future events, the participants *perceived* events in time proportionally to their distance from the present. *Perception*, rather than the discounting of the future, was the explanatory model proposed and empirically validated

On a broader level, dissatisfaction with the ‘rationalistic’ models of decision making in general had led already in the 70s and 80s to the emergence of *naturalistic decision making* studies that focus on understanding how people think outside the confines of artificial experimental settings and narrow data collection methods such as the infamous survey questions deployed by Tversky and Kahnemann (Kahneman & Tversky, 1979). Most prominently, Klein led large ethnographic studies of decision making in natural settings to understand how highly performing professionals effectively deploy their expertise and knowledge (Klein, 1998). His focus was on what he terms the *macrocognitive system* of decision making – the entire system of environmental cues, roles, incentives and structures, knowledge, prior experience, work processes, tools, available information, time constraints, etc. which influences the cognitive processes – understood psychologically and socially – which comprise *decision making* as a situated process. This view is akin to the well-known view of cognition “in the wild” (Hutchins, 1995) which has been influential in HCI (Rogers & Marshall, 2017). These naturalistic decision making studies have led to profound insights into the nature of expertise and skills, and the cognitive processes at work in experienced professionals (Klein, 1998). In Software Engineering, the focus of attention has been firmly on the normative, rationalistic perspectives and the associated research program of

heuristics and biases (Mohanani et al., 2018), with some notable exceptions that found significant evidence for the value of NDM in SE (Zannier et al., 2007).

In our previous studies, we found significant evidence that professionals acted *as if* they discounted the future (Becker et al., 2019; Fagerholm et al., 2019) – though whether it was because of perception or because of something like a discount factor, we do not know. In an ongoing large-scale study, we examine the cognitive processes at work in such situations using *Cognitive Task Analysis* methods developed by Klein and others (Crandall et al., 2006). The game *Undecided?* is a next step: By creating a simulation environment that playfully embeds a range of intertemporal choice situations in a group decision making setting, we hope to create a platform that allows us to study intertemporal choice in SE from multiple angles using many different methods.

3. The game: UNDECIDED?

Overview

Undecided? is an educational game that seeks to provide players with a general and integrated understanding of project management as well as software, product, and systems development. It is a tabletop game that uses a game board to simulate project phases and cards to mimic the actions undertaken and obstacles encountered in the development process. Players work together to complete a project, hitting milestones and striving for objectives throughout project phases. While *Undecided?* focuses strongly on collaboration, individual players also work to achieve personal goals corresponding to their role on the project team. This creates the possibility of moderating the tension between group and individual goals by varying aspects of the background scenario that provides a narrative frame for the game.

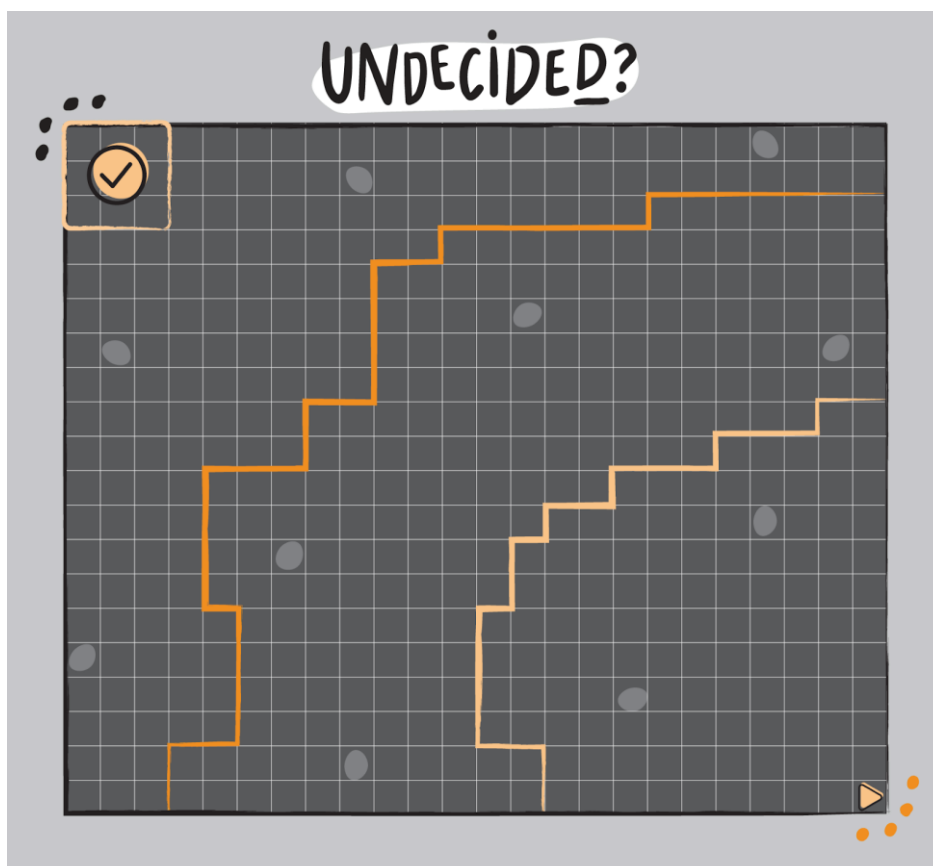


Figure 1 The start square is on the lower right of the board, the final gate on the upper left.

Undecided? is played on a rectangular board partitioned in three stages, with a start square in one corner and a target range in the opposite corner (shown in Figure 1). Each board facilitates play for one single team. Multiple teams compete by playing on adjacent boards. In each round of the game, the team draws cards and decides which of the cards they currently hold to play next. Playing a card means to place it

on the board, adjacent to a previous card, so that it covers eight squares. Crossing the gate from one stage to the next, shown by the colored lines, requires certain conditions to be fulfilled.

The team keeps a score on the following categories that represent the robustness of the project:

- **Internal** Quality of the software system under development
- **External** Quality of the software system under development
- **Process** Quality
- **Team** Strength

Each card – i.e. each possible action – will influence the team’s current score on some or all of the four dimensions. How much exactly is revealed after the move is performed, but the card design indicates visually what will happen, as illustrated by Figure 2. Note that *Major Toolchain Upgrade* is the only intertemporal choice in this example set. The four dots indicate that the benefit on *Process* will manifest over the following four rounds. The orange semicircles indicate that there is a *risk* of negative effects. In contrast, architectural design and feature development are relatively straightforward positive contributions to specific aspects of the project.

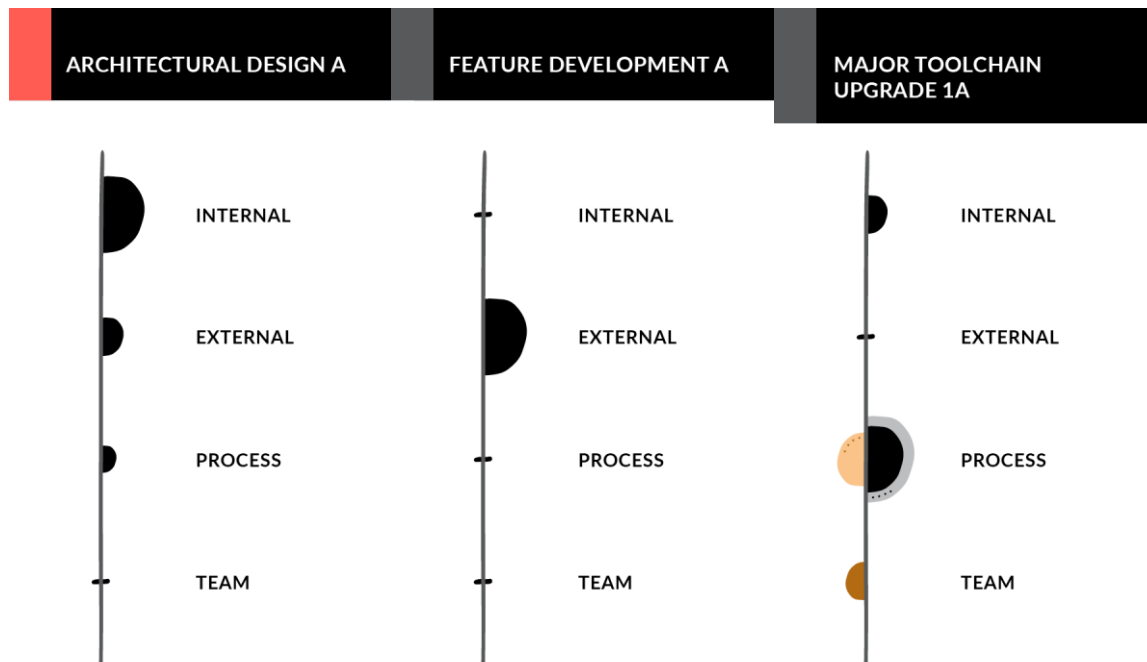


Figure 2 Selected action cards

Game play: Roles, Setup and Rules

A *game facilitator* is involved in the form of either a person who controls the game or an app (under development). The facilitator does not participate in play because they have access to secret information such as precise gate values and will determine how action cards are scored.

Player roles are determined before the game starts either randomly or through deliberation. Roles have various advantages when certain cards are laid, and outline role-specific goals that a player must meet to gain – or prevent the loss of – personal action points (AP).

Player roles are as follows:

- **UX Lead**
 - Focuses on External Quality
- **Team Lead**

- Focuses on Team Strength
- **System Architect**
 - Focuses on Internal Quality
- **Technical Lead**
 - Focuses on Process Quality

Role-specific goals are representative of the overall standards of project management and development, with AP awarded to players if various cards are laid during certain project phases. (Action Points can be used collectively in specific “Blank” Action Cards.) For example, a UX lead receives +4 AP if a user studies card is played in the first phase, as it is generally regarded as an important step to consider early on in development. If a user studies card isn’t laid in the first phase, the UX lead loses 4 AP to represent the impact of failing to determine user needs.

To set up, the Facilitator takes the Facilitator Notes. These are confidential: Players must not see them.

Players begin by reading their *Game Scenario*. The Facilitator reads their information packet based on the game scenario, keeping the contents secret unless otherwise directed. Players then set up the game board and shuffle the card deck. Players are assigned a role, either at random using a dice roll or through deliberation. Players then take their role description sheet. Next, the players draw 5 team cards from the deck and laid face-up in front of all players.

- If a “play immediately” event card is drawn at this point, return it to the deck and draw another card in its place. Examples of such cards are shown in Figure 3.



Figure 3 A sample of "Play immediately" cards

Once all elements are in place, gameplay begins. The team lead goes first. The first card laid must begin on the start square, at the bottom right of the board.

Core Gameplay Loop

Players take turns by moving clockwise around the table. Players begin their turn by drawing a card from the deck.

- If the card is a “play immediately” event card, the player lays the card on the board, follows the instructions, and ends their turn.

The player then consults their team to determine the next card to lay. Though collaboration is encouraged, the current player makes the final decision as to what card is laid.

- If a *Blank Action* card is in the hand, players can ask team members to contribute their AP points to the card. If the player is a senior-level team member, they can take as many action points as desired from team members. (Yes, there is a level of cruelty to that.)

The card is then laid on the game board so it is touching the edge of at least one other card. Cards can be laid in any orientation as long as it follows the grid pattern on the gameboard

- If the card is laid over an Event Square (grey circle), the current player partakes in the event, then ends their turn.

Cards can only be laid in the current project phase, unless the team wishes to attempt a *Gate Pass* and move onto the next project phase.

Gate Pass

A *Gate Pass* is initiated by a player laying a card across one of the orange borders on the game board.

All players, except senior-level players must then take part in a project review, constructing a narrative of their project up to the current point. The Senior-level player(s) then select the player with the best project story, and award them +3 AP if the gate pass is successful.

After the project review, the gate scores are revealed. If the team's scores exceed or equal the gate scores, they move into the next project phase.

If the scores fall short in any category, the player must leave the card on the board and turn it over. This card becomes a “dead” card and does not add to the team score, with the additional punishment of blocking part of the gate. The current player's turn is then over, and the team must continue to build their project in the current phase and attempt to pass again by repeating the process.

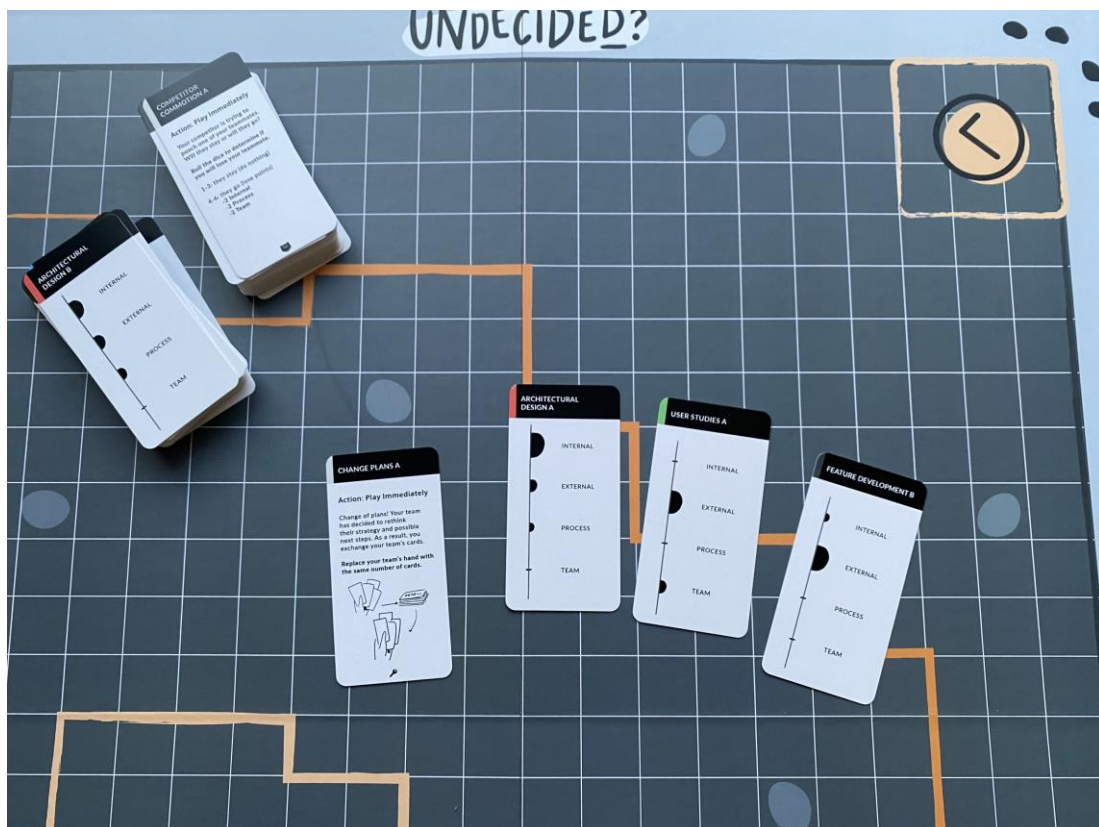


Figure 4 Partial view of an Undecided? board with selected cards

Winning/Losing the Game

The game is won when the team passes the final gate on the game board.

- After the final gate is passed, final scores are tallied.

The game is lost if:

- The team runs out of room to lay cards on the game board.
- The team runs out of cards in the deck
- The team makes a choice in a *Game Scenario* that triggers a loss state.

Figure 4 shows parts of a finished board with selected cards.

Cards and Scoring

Each move consists of playing a card. Each Card represents a prototypical action that the team can take. The team is scored on the four categories that represent the robustness of the project:

- **Internal** Quality of the software system under development
- **External** Quality of the software system under development
- **Process** Quality
- **Team** Strength

Points in each category are earned through the laying of cards on the game board.

When a card is laid, the facilitator uses their **score sheet** to determine how points are distributed.

- Some Event Cards are scored based on a dice roll. Dice rolls result in a success or failure of the event/action. In this instance, the facilitator must input the result of the dice roll into the score sheet for the score to be calculated.

The game scenario determines what scores are needed in each of the four categories for the players to progress through project phases.

The game facilitator uses a predefined spreadsheet to determine score values for the team. The facilitator is in charge of inputting the title of the played card into the sheet, and of reading the score values of that card aloud once they have been calculated by the sheet. A *Facilitator Notes* document provides specific instructions.

Scenarios and Major Events

The game is structured by the main narrative. Currently, we offer two narratives:

- In *Angry Cats*, the team is part of a start-up game studio and has just signed a contract with a prominent publisher to develop a new game and bring it to market.
- In *DysTalk*, the team has formed a start-up to develop a secure communication and networking product.

Each scenario comes with one major event that happens at an undisclosed point in the game. The event is triggered by external forces and will pose a significant intertemporal choice to the team outside the regular game play. But we shall not give away what it is – where would be the fun in that?

Moderation, online game play, and app development

As mentioned above, the game is based on some form of facilitation – in the initial deployment, that is a human moderator using a macro-enabled spreadsheet, but we are also developing an app that handles the game mechanics. The game cards are equipped with individual QR codes that can be scanned in order to make a move, and the app will take over the facilitation role once developed. We envision that the app may also

- Facilitate team competition,
- Incorporate educational content such as short explanatory videos,
- Facilitate a playful interaction across time between different teams, and
- Allow a board-less gameplay that facilitates remote group play with virtual cards.

To accommodate the pandemic circumstances, the visual game materials were imported to an online whiteboard to facilitate game play in parallel breakout groups, as shown in Figure 5.

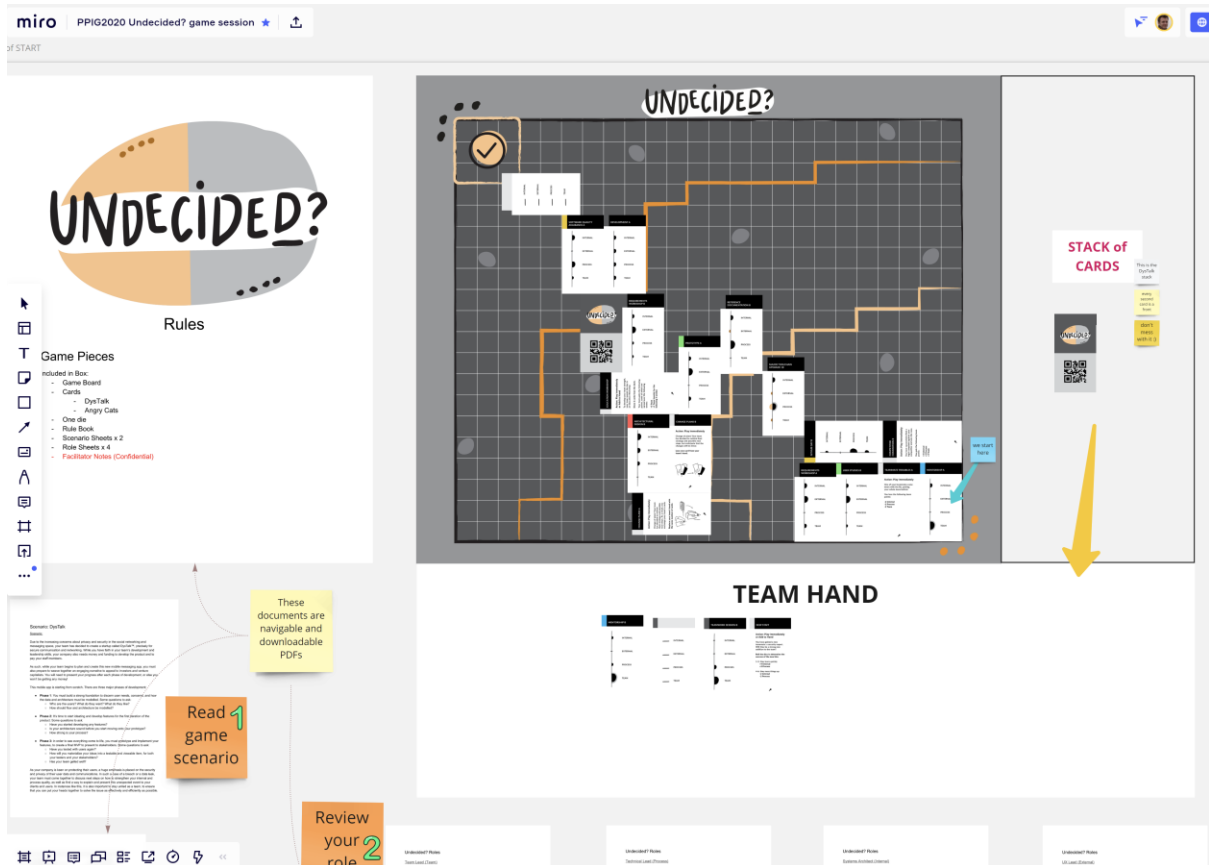


Figure 5 Segment of the online board game environment at PPIG 2020

4. Intertemporal Choices in *Undecided?*

Intertemporal choices in this game come in two different forms: routine and exception. The routine choices arise out of a subset of cards that carry intertemporal components, as outlined above. The exceptional choices arise out of the major game event that strikes the team along the course of game play and requires their attention.

The choices differ in several key dimensions.

- **Uncertainty and ambiguity.** The exact size of effects is uncertain – both spreadsheet and app employ simple randomizing functions. But there is also an element of ambiguity since the probability distribution remains blurred for the players. The visual design of effects as variously sized dots represents an intentional effort to keep game play from becoming a numbers' game. We do not want players to directly calculate the maximum number of achievable points and devise some heuristics, as that would circumvent what the main focus of interest is – the deliberation around the various dimensions and time. By developing such a heuristic, a player would in fact be playing a different game. In our prototype game play sessions, this worked quite well, but will need to be evaluated more rigorously.
- **Temporal and social distance.** Through their team roles, players are tied into a focus on certain dimensions. They become spokespersons for those dimensions, and they may in some situations not mind a loss in other dimensions. However, it is clear at all times that the team can only win collectively.
- **Large vs small outcomes.** Routine actions carry smaller weights, while the major event carries larger implications and highly ambiguous risks.
- **Explicit vs implicit choices.** The selection of cards provides a very explicit – and not entirely realistic – choice of possible actions. This is in some sense a limitation – NDM researchers such as Klein have argued convincingly that *comparative evaluation* of clearly enumerated alternatives

is often not the focus of real-world decision making under time pressure. However, some choices are more implicit, and some types of cards carry an open-ended scoring scheme that allows the team themselves to decide what action to take and how to label it.

- **Rhythm.** The regular routine choices come in each round, but some action cards are labelled “play immediately” and thus disrupt the cycle of deliberation, choice and status update. The major event is initiated by the facilitator, and not mentioned in the instructions for players, so it is designed to catch them by surprise.
- **Deliberation, choice, and reflection.** Each round will typically involve a phase of discussion resulting in a choice. Different configurations of roles can be deployed. However, some hierarchy is foreseen through the introduction of senior roles, and some choices can be taken by senior roles that affect others and can overrule others. Finally, the narrative reflection requested at each Gate Pass provides a retrospective that places the players into a narrative mode quite distinct from the deliberation and choice mode. Players can get quite creative in the story they tell at that point.

These design choices were made to facilitate a range of interventions and study designs to allow us to examine how individuals and groups reason; how variations in the setting may influence their choices; and to provide a situation that we can observe and retrospectively analyse using techniques from Cognitive Task Analysis.

To deliver educational value, we anticipate the game should be embedded in a workshop setting with a debriefing session that explores the nature of intertemporal choice and its role in software projects and includes a guided team reflection. It should also ask: *What is realistic about the game? What isn't? How is reality different? How and where are intertemporal choices hiding in our practice? Do we need to change the way we approach such choices, and how?*

The next section outlines a range of study designs we envision.

5. Research Designs

This section shortly outlines a range of envisioned uses of the game as a research platform and some of the possible choices and challenges in study design. We first outline a range of instruments, interventions and data collection methods that are of relevance in this setting, then outline possible study designs with concrete aims and combinations of instruments.

- **How to measure discounting:** As discussed in previous studies (Becker et al., 2018; Fagerholm et al., 2019), there are various ways of measuring the amount to which participants discount future outcomes, and a choice has to be made for each study which is most appropriate. In our recent studies, we opted for a more robust measure of the general amount of discounting over time per participant called *Area Under Curve*. It is worth mentioning that: (1) the amount of discounting varies wildly across participants in our studies (Fagerholm et al., 2019) and across studies in general (Frederick et al., 2002)(2) some of our participants do not exhibit discounting at all. It is therefore well worth examining the range of individual responses to explore possible reasons and forms of reasoning or the absence thereof.
- **Demographics** and differences in individual players should thus be explored. In our prior studies, we found no effects of education nor the amount or area of professional experience on discounting, but identified a significant effect – the larger the *range of* professional experience, the less participants discounted (Fagerholm et al., 2019), pointing to a possible role of empathy in how psychological distance affects discounting (Weber, 2006).
- **Time perception**, instead of discounting, has been proposed as an explanation for the appearance of discounting behavior (Zauberman et al., 2009). In that study, participants were asked to draw a line and to evaluate the length of lines in relation to time. A similar instrument could be deployed to establish individual differences in time perception before game play.
- **Think Aloud Protocol** analysis can be used to a limited degree in a group setting, but it can be useful in individualized game modes such as online game play.
- More generally, **Cognitive Task Analysis** offers an entire toolbox of methods including interviews using Critical Incident Method for retrospective interviews as well as observational methods that can be deployed non-obtrusively in game play settings.

- **Surveys and interviews** support an evaluation of the game from the perspective of participants in terms of educational value (short-term and long-term learning outcomes) as well as enjoyment value (short-term) and possible side-effects (e.g. team formation, interpersonal relations)
- Finally, we hope to explore the effects of **interventions** on intertemporal choice, guided by emerging insights such as the possible role of empathy in overcoming psychological distance; the role of specific education modules (such as technical debt) on discounting; or a range of priming effects. For example, the Empathy Toy® is “a blindfolded puzzle game that can only be solved when players learn to understand each other”, a playful way to activate empathy skills in players (*The Empathy Toy*, n.d.). Would its use before game play influence the choices made by participants?

Of course, the range of choices made for each concrete study will have to involve a careful configuration of designs, instruments and interventions fit for the purpose and methodological assumptions of the study. For example, a study could examine the role of group composition and incentive structures on reasoning strategies and discounting outcomes. The roles embedded in game design involve a few structural conflicts of interests, but they are built toward an overarching team interest. Different instructions for each team member, and different incentive structures, could be used to explore how they affect people’s reasoning and group dynamics. This would involve a very different design from a study to explore to which degree the sensitization to intertemporal choice arising from playing the game leads to future shifts in discounting behavior on the side of participants.

We currently aim to prioritize the following types of studies ourselves.

- In **Cognitive Task Analysis** studies, we aim to observe game play in groups and conduct retrospective interviews with groups and/or participants, supported by additional data collection instruments including pre- and post-game surveys and measures of discounting.
- In **Randomized Control Trials**, we hope to assess the effect of
 - o **Interventions** outside the game such as an empathy workshop or the delivery of an education module, as well as
 - o **Variations in game design** that vary the *architecture of choice* that shapes and configures how participants enter the choice situation and which information is presented to them. This can involve variations in the time scales and how they are presented – for example as weeks, months or sprints; variations in the language used to frame outcomes; variations in the emphasis given to gains and losses; or simply variations in the instructions given to individual players.
- We aim to explore **the game as intervention** embedded in a workshop, in industry or community settings. We envision the first rounds of this as Action Research to explore the possible values and effects of the game and associated workshop, and to identify future directions of game design and development. This could eventually lead to a more quantitative validation of the game’s value, which requires a clearly identified dependent variable such as the amount of discounting exhibited by a team or person. That needs to be measured through a validated instrument, for which our previous study design may provide a starting point (Fagerholm et al., 2019).

6. Conclusions

Choices involving uncertain future outcomes that are spread in time – so-called intertemporal choices – abound in software projects. Multiple factors impact how such choices are made, including the uncertainty and ambiguity of the options and outcomes, the combination of favorable and unfavorable outcomes at different points in time, and the psychological distance to people who are affected. This macrocognitive view of decision making situates the cognitive processes of decision-makers in the social environment.

We have previously examined intertemporal choices in software engineering and found evidence for extensive discounting, but also of large individual differences (Becker et al., 2019; Fagerholm et al., 2019). However, what determines the choices is still unclear. The board game "Undecided?" simulates a software project through a series of intertemporal choices made by players in teams. The game is educational and fun, and it aims to provide a platform for cognitive and social studies of decision

making in software projects. This paper describes the game and outlines possible research designs in which the game is used to shed more light on how decisions are made in software projects.

7. Acknowledgements

The design of *Undecided?* was partially funded by the Canadian Natural Sciences and Engineering Research Council under NSERC RGPIN-2016–06640. The game was designed while Fabian Fagerholm was a Postdoctoral Fellow at the University of Toronto.

8. References

- Becker, C. (2014). Sustainability and Longevity: Two Sides of the Same Quality? *Proceedings of the Third International Workshop on Requirements Engineering for Sustainable Systems Co-Located with 22nd International Conference on Requirements Engineering (RE 2014)*, 1216, 1–6. <http://ceur-ws.org/Vol-1216/>
- Becker, C., Chitchyan, R., Betz, S., & McCord, C. (2018). Trade-off Decisions Across Time in Technical Debt Management: A Systematic Literature Review. *Proceedings of TechDebt '18: International Conference on Technical Debt, Co-Located with the 40th International Conference on Software Engineering (ICSE 2018)*. <https://doi.org/10.1145/3194164.3194171>
- Becker, C., Fagerholm, F., Mohanani, R., & Chatzigeorgiou, A. (2019). Temporal Discounting in Technical Debt: How do Software Practitioners Discount the Future? *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 23–32. <https://doi.org/10.1109/TechDebt.2019.00011>
- Becker, C., Walker, D., & McCord, C. (2017). Intertemporal Choice: Decision Making and Time in Software Engineering. *Proceedings of the 10th International Workshop on Cooperative and Human Aspects of Software Engineering*, 23–29. <https://doi.org/10.1109/CHASE.2017.6>
- Camerer, C., & Weber, M. (1992). Recent developments in modeling preferences: Uncertainty and ambiguity. *Journal of Risk and Uncertainty*, 5(4), 325–370. <https://doi.org/10.1007/BF00122575>
- Crandall, B., Klein, G., & Hoffman, R. R. (2006). *Working Minds: A Practitioner's Guide to Cognitive Task Analysis* (1 edition). A Bradford Book.
- Fagerholm, F., Becker, C., Chatzigeorgiou, A., Betz, S., Duboc, L., Penzenstadler, B., Mohanani, R., & Venters, C. C. (2019). Temporal Discounting in Software Engineering: A Replication Study. *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–12. <https://doi.org/10.1109/ESEM.2019.8870161>
- Frederick, S., Loewenstein, G., & O'donoghue, T. (2002). Time Discounting and Time Preference: A Critical Review. *Journal of Economic Literature*, 351–401.
- Hutchins, E. (1995). *How a Cockpit Remembers Its Speeds*. 19(3), 265–288.
- Kahneman, D., & Tversky, A. (1979). Prospect theory: An analysis of decision under risk. *Econometrica: Journal of the Econometric Society*, 263–291.
- Keren, Gideon., & Wu, George. (Eds.). (2015). *The Wiley-Blackwell handbook of judgment and decision making* (<http://go.utlib.ca/cat/10376181>). Wiley-Blackwell.
- Klein, G. A. (1998). *Sources of Power: How People Make Decisions* (<http://go.utlib.ca/cat/8525456>). MIT Press.
- Loewenstein, G., Read, D., & Baumeister, R. F. (2003). *Time and Decision: Economic and Psychological Perspectives of Intertemporal Choice*. Russell Sage Foundation.
- Loewenstein, G., Rick, S., & Cohen, J. D. (2008). Neuroeconomics. *Annu. Rev. Psychol.*, 59, 647–672.
- March, J. G. (1978). Bounded rationality, ambiguity, and the engineering of choice. *The Bell Journal of Economics*, 587–608.
- Mohanani, R., Salman, I., Turhan, B., Rodriguez, P., & Ralph, P. (2018). Cognitive Biases in Software Engineering: A Systematic Mapping Study. *IEEE Transactions on Software Engineering*, 1–1. <https://doi.org/10.1109/TSE.2018.2877759>
- Naur, P., & Randell, B. (Eds.). (1969). *Software Engineering: Report on a Conference sponsored by the NATO Science Committee*. NATO Scientific Affairs Division.
- Neumann, P. G. (2012). The Foresight Saga, Redux. *Commun. ACM*, 55(10). <https://doi.org/10.1145/2347736.2347746>

- Parnas, D. L. (1994). Software Aging. *Proceedings of the 16th International Conference on Software Engineering*, 279–287. <http://dl.acm.org/citation.cfm?id=257734.257788>
- Rogers, Y., & Marshall, P. (2017). Research in the Wild. *Synthesis Lectures on Human-Centered Informatics*, 10(3), i–97. <https://doi.org/10.2200/S00764ED1V01Y201703HCI037>
- Samuelson, P. A. (1937). A Note on Measurement of Utility. *The Review of Economic Studies*, 4(2), 155–161. JSTOR. <https://doi.org/10.2307/2967612>
- Soman, D., Ainslie, G., Frederick, S., Li, X., Lynch, J., Moreau, P., Mitchell, A., Read, D., Sawyer, A., Trope, Y., Wertenbroch, K., & Zauberman, G. (2005). The Psychology of Intertemporal Discounting: Why are Distant Events Valued Differently from Proximal Ones? *Marketing Letters*, 16(3), 347–360. <https://doi.org/10.1007/s11002-005-5897-x>
- The Empathy Toy*®. (2020). <https://twentyonetoys.ca/pages/empathy-toy>
- Weber, E. U. (2006). Experience-Based and Description-Based Perceptions of Long-Term Risk: Why Global Warming does not Scare us (Yet). *Climatic Change*, 77(1–2), 103–120. <https://doi.org/10.1007/s10584-006-9060-3>
- Zannier, C., Chiasson, M., & Maurer, F. (2007). A model of design decision making based on empirical results of interviews with software designers. *Information and Software Technology*, 49(6), 637–653. <https://doi.org/10.1016/j.infsof.2007.02.010>
- Zauberman, G., Kim, B. K., Malkoc, S. A., & Bettman, J. R. (2009). Discounting Time and Time Discounting: Subjective Time Perception and Intertemporal Preferences. *Journal of Marketing Research*, 46(4), 543–556. <https://doi.org/10.1509/jmkr.46.4.543>