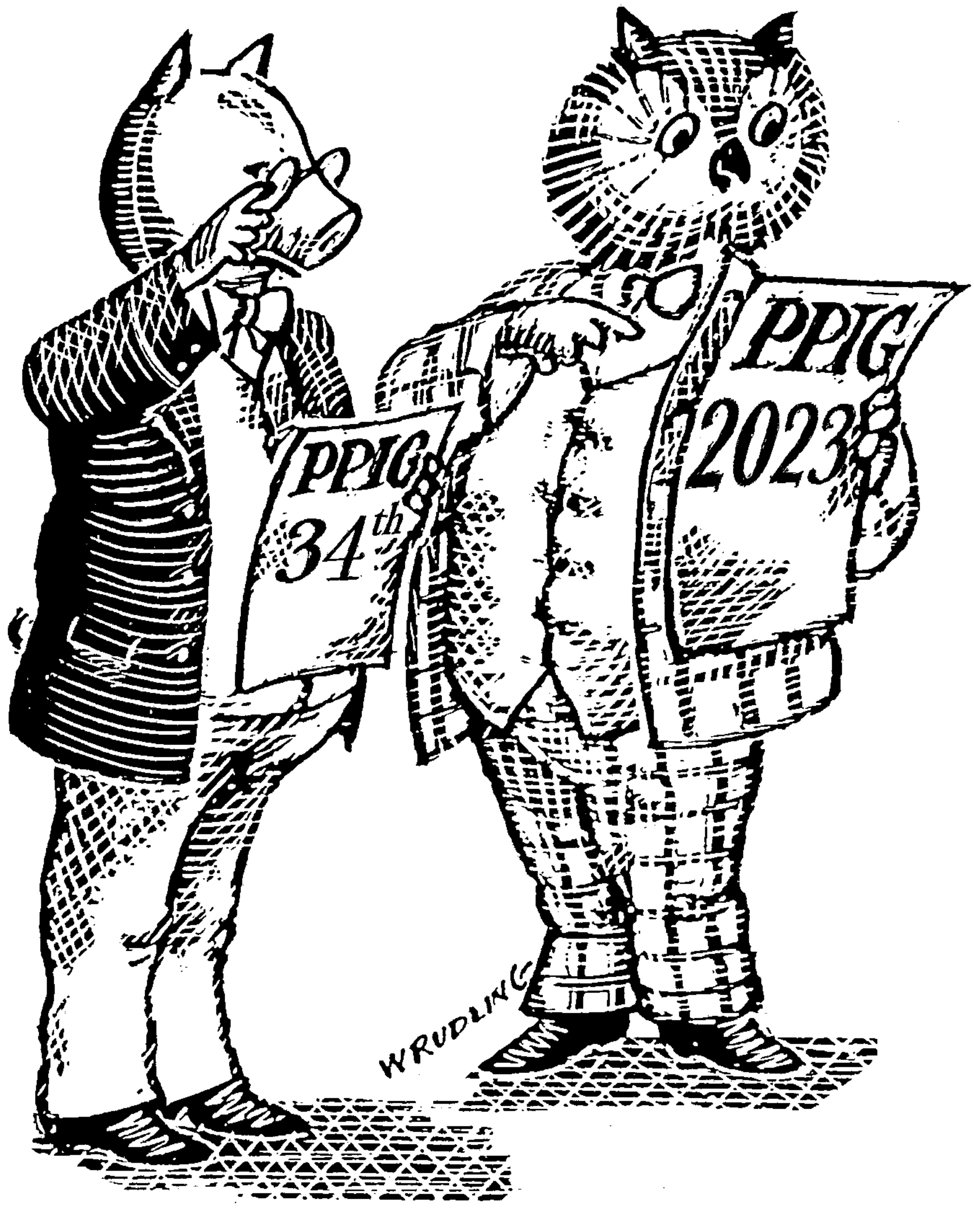
PPIG 2023

Proceedings of the 34th Annual Workshop of the Psychology of Programming Interest Group

21 - 25 August 2023

Lund University, Sweden & Online



Edited by

Emma Söderberg, Luke Church, Mariana Marasoiu

PPIG 2023
Programme & Proceedings Index

Monday, 21st August

10:00 - 12:00	Doctoral Consortium (invite-only meeting)	
13:30 - 14:00	*Fika & registration*	
14:00 - 16:00	*Demos & music - in-person* Bring your own instrument, cool tool or prototype!	
16:00 - 16:10	*Conference open - hybrid* PPIG 2023 Open and Welcome	
16:10 - 19:00	*Presentations session - hybrid*	
	Keynote: **Socio-Technical Smells: How Technical Problems Cause Organizational Friction** Adam Tornhill, CodeScene	4
	A brief history of the human centric study of programming languages Luke Church and Alan Blackwell	5
	Overcoming the Mental Set Effect in Programming Problem Solving Agnia Sergeyuk, Sergey Titov, Yaroslav Golubev and Timofey Bryksin	22
	Automatic, Suggestive Feedback in Algorithm Visualisation Exercises Artturi Tilanterä	37
19:30 onwards	*In-person* Vattenhallen & PPIG Dinner	

Tuesday, 22nd August

9:30 - 10:00	*Coffee and registration*	
10:00 - 12:00	*Hands-on activities - in-person* Visit to the Lund University Humanities Lab	
12:00 - 13:30	*Lunch*	
14:00 - 16:00	*Hands-on activities - in-person*	
16:00 - 18:00	*Presentations session - hybrid*	
	Keynote: **Bots in Software Engineering and their impact on development** Linda Erlenhov, Chalmers University of Technology	43
	How A Data Structure's Linearity Affects Programming and Code Comprehension: The Case of Recursion vs. Iteration Aviad Baron and Dror Feitelson	44
	Exploring cognitive waste and cognitive load in software development - a grounded theory Daniel Helgesson	60

Wednesday, 23rd August

PPIG @ AI Lund: Fika-to-Fika Workshop

This was an integrated workshop in collaboration with AI Lund (ai.lu.se).

Thursday, 24th August

Friday, 25th August

16:00 - 17:45 *Presentations session - online*

17:45 - 18:00 *online*
PPIG Prizes and conference close

Socio-Technical Smells:
How Technical Problems Cause Organizational Friction

Adam Tornhill
CodeScene
adam.tornhill@codescene.com

Keynote abstract

Successful software development requires that you keep code and people in balance so that one supports the other. It's a hard challenge since a piece of code doesn't reveal anything about its socio-technical context. Enter behavioral code analysis, an approach which combines code level metrics with data on how teams interact within the code. Armed with these techniques, we look to reduce organizational friction by focusing on a set of common challenges:

- Identify architectural coordination bottlenecks and understand the technical root causes.
- Visualize implicit dependencies between teams, act to decouple teams.
- Discover knowledge risks by measuring the Truck Factor. Learn how to mitigate it.
- Communicate the scaling risks inherent in Brooks's Law by showing data on how it impacts your delivery.
- Go beyond technical impact by knowing how bad code causes unhappiness, low morale, and increased attrition.

A brief history of the human centric study of programming languages

Luke Church
Computer Laboratory
Cambridge University
luke@church.name

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

Abstract

The study of programming languages focussing on the needs of the programmer has been a subject of intellectual enquiry for at least the past 50 years. We draw one through this history that is likely to be of particular interest to the Psychology of Programming audience highlighting recurring tensions of internal validity, generalisability and practical utility. We suggest this longer term perspective is useful for informing contemporary debates.

1. Introduction

The question of how to design languages for programming computers has been a matter of interest for a number of different disciplines in the past. Engineers working in compiler construction have considered the impact of the design of the languages on the technology needed to run programs (e.g. Waite and Goos 1984). Mathematicians have studied the ways in which the structure of programs affects the statements we can make about their behaviour (e.g. Pierce 2002). Psychologists and interaction design researchers have studied the design from a human perspective. In their landmark paper (Newell and Card 1985) observe that:

"Now programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use."

This paper offers an overview and reflections on the human focussed, scientifically inclined, study of programmers and their behaviour. It is not a historical overview of the literature in psychology of programming - readers interested in that overview can refer to (A. F. Blackwell, Petre, and Church 2019). Alternatively, a more light-hearted historical introduction to the field can be found in (A. F. Blackwell 2017).

2. The origins of programming

During the early history of electronic computing, the only way of interacting with a computer was to write, or control the execution of, a program. For more than a century, the study of the interaction with a computer was effectively the study of programming, from Ada Lovelace's famous 1843 notes on the programming of Babbage's Analytical Engine (Menabrea and of Lovelace 1843) to the 1947 report by Goldstine and Von Neumann on Planning and coding of problems for an electronic computing instrument (Goldstine and von Neumann, n.d.).

Although nothing corresponding to the modern user interface had been imagined, it is notable that these pioneers already focused on problems of notation, and also that they considered the extent to which it would be feasible to specify different classes of problem using these notations. This means that early research on interaction with programs can still be informative when discussing the properties of modern programming environments (see e.g. (Arawjo 2020)). Furthermore, due to the importance of programming in early computing, the study of the usability of programming was formative in the study of the usability of computing more generally. In the first part of this review we will describe the evolution of the perspectives and techniques that researchers have used to try and improve the usability of programming, as well as the venues in which this work was disseminated.

From the perspective of subsequent consolidation of cognitive science as an interdisciplinary field, we can see that early contributions often involved implicit cognitive claims about what people find easy or hard when programming, followed by the implications for design of those claims. Aspects of human performance and experience have been part of the argument from the beginning.

3. Early cognitive claims (1968-1973)

An early example of a cognitive claim made by a computer scientist came from Edsger Dijkstra. In a now famous letter to the Communications of the ACM, he stated his position that 'Go To Statement Considered Harmful' (Edsger W. Dijkstra 1968) . This work makes essentially cognitive arguments such as:

'*My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed.*'

Dijkstra then goes on to consider the implications for design of these claims such as:

'For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.'

The broader position that is outlined in the 'goto considered harmful' letter is that the structure of programs can be described by a number of 'independent co-ordinates' inherent to sequential processes by which progress in execution can be measured. Dijkstra argues that the 'unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress.'

The argumentation is presented as an expert experience report "For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce.", it does not present evidence to support this claim, or a definition of what 'quality of programmers' might be, or even a description of what the programmers being considered were doing. However on the basis of this experience report and several cognitive hypotheses, Dijkstra felt confident in recommending broad design changes to the majority of programming languages:

‘*I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine Code).*’

In 1971, (Evershed and Rippon 1971) follow a similar form of argumentation, considering their experience of the usability characteristics of various aspects of ALGOL and FORTRAN. On the basis of these characteristics they go on to make recommendations for a variety of design changes. They include an economic argument, which can be seen as a precursor of end-user programming: ‘with the esoteric content of programming eliminated, a much broader section of the population will become potential computer users’.

The scope of Evershed and Rippon’s work is broader than Dijkstra’s earlier work; it independently discusses several language features, however it doesn’t propose an overarching psychological hypothesis like Dijkstra’s ‘independent coordinates’.

Evershed and Rippon, as well as Dijkstra, argue from their experience rather than from documented empirical evidence. When (Sime, Green, and Guest 1973) published the results of a controlled experiment, it represented both a new approach to studying programming languages as well one of the first applications of what would become known as Human Computer Interaction (HCI) methods (Katz, Petre, and Leventhal 2001).

In their work, Sime, Green and Guest compare the behaviour of programmers when presented with two different forms of a conditional construct, the if-then-else form (named NEST) and the branch-to-label form (named JUMP). The authors constructed two micro-languages using the NEST and JUMP forms respectively. They then conducted a controlled experiment, in which participants were given a simple program description in English which they needed to translate into one form, and then after a week to translate the same English description into the other form. The experimenters recorded the number of problems that the participants did not correctly solve the first time, as well as the time to complete the tasks.

The experiment clearly identified that the NEST form is a statistically significant improvement over the JUMP form, resulting in increased task completion rate and fewer semantic errors where the program the participant submitted for execution did the wrong thing.

This experimental approach is borrowed from cognitive psychology. It follows the methodology of considering a particular property that is common in a number of languages, constructing a micro-language that stresses the property and conducting an empirical experiment for measuring the effect of those properties. The authors hope that

‘*By devising micro-languages exemplifying particular features of interest, knowledge can be gained which allows more informed decisions to be made when designing new languages.*’ - (Sime et al. 1973).

Green and others expanded on this strategy in subsequent work on e.g. the delimitation of scopes (Sime, Green, and Guest 1977). This arc of work not only set the foundation for later work in HCI, but also established an empirical perspective on the evaluation of candidate language features.

Just as Evershed and Rippon were hopeful of increasing the access to computers for a broader section of the population, Sime et al. were concerned with the importance of studying the usability of computers and of programming systems by ‘non-specialists’. In a practice that would be followed by many others, the population of non-specialists they used as test participants were university students.

4. Broadening interest (1973-1986)

After the publication of the Sime et al’s work there was a growth of interest in the application of empirical techniques, especially the relatively informal Software Psychology society convened by Ben Shneiderman (Shneiderman 1986). In his book *Software Psychology*, Shneiderman (Shneiderman 1980) outlines many perspectives on the problem of creating software. He defines software psychology, a term he attributes to Tom Love, as “the study of human performance in the using computer and information systems” - but concentrates on software development rather than software use.

Shneiderman outlines a number of the methodologies in use for this research including introspection, detailed analysis of recordings of activities, called protocol analysis, discussion of code corpora such as (Knuth 1971), and controlled experiments. Shneiderman also deconstructs the idea of ‘programming’ as an activity, instead describing it as being composed of the tasks of Learning, Design, Composition, Comprehension, Testing, Debugging, Documentation and Modification.

Much of the book is concerned with what would become the study of software engineering, including organisational and managerial concerns. The material concerned with the design of language features is primarily discussed in Chapter 4 - ‘Programming Style’. It documents experiments concerned with the impact of comments on comprehensibility, (unpublished work by Peter Newsted) and apparently contradictory results on the effect of variable naming schemes ((Weissman and University of Toronto. Computer Systems Research Group 1973)), null results on the effects of indentation on comprehension of FORTRAN programs ((“RELATING INDIVIDUAL DIFFERENCES IN COMPUTER PROGRAMMING PERFORMANCE TO HUMAN INFORMATION PROCESSING ABILITIES” 1977)) and a extended discussion of choices of control structures.

At the end of this chapter, Shneiderman’s advice for designers of languages is that they *‘should recognize the that specific features may have a statistically significant effect on performance and should thoroughly test alternative proposals’* ((Shneiderman 1980), p90)

It is clear from the number of studies referred to in this section that by 1980 there was a growing community of research around programming language usability, and whilst advice about specific features such as jumps or blocks could be given, the broad advice at this point was that it was important to test the usability properties of the various choices in languages.

The work between 1973 and 1980 was criticised by (R. E. Brooks 1980) and (Sheil 1981). Brooks raised concerns that the heterogeneous nature of the total population of programmers results in experiments needing extremely large samples to achieve significance. He highlights a specific source of variation associated with the use of timing measures, observing that the time to solve the problem is interleaved in a non-trivial way with the time to understand the question, and that the effect size is diluted by the participants needing to understand concerns not strictly relevant to the test, for example understanding a library used in the stimulus material, when the purpose of the experiment was actually instruction flow.

Brooks also mentions a range of issues with the external validity of the experimental designs; he is concerned that a common practice to reduce variation, using students as participants, results in studies where the sample is not representative of the larger population of programmers. Brooks observed that the programs in the stimulus materials at the time were all under 500 lines, compared to contemporary commercial programs which were in excess of a million lines. He questions the validity of the experimental measures used, suggesting that timing studies may not be representative of a real value property. For example, measuring only development time may exclude considerations of quality, and debugging exercises may not successfully entail that the participant understands the program and so on.

The broad direction of this criticism is that designing an experiment which has a statistically significant effect is hard, and even when it is achieved, it can be difficult to know if the observed effect generalises to the practice of programming and software engineering.

Brooks is outlining, in 1980, one of the central motivating concerns of our work: knowing how to use information about programmers to improve design in a way that has validity outside of the experimental context. Brooks' response to these problems was to advocate the development of models of the cognitive processes involved in programming to understand and manage the methodological issues:

"*What approaches, then, show promise? Any successful characterization of the program-programmer interaction will probably be based on a model of the process or processes used by a programmer in interacting with a program. The development of such theories or models of the cognitive processes involved in programming is, therefore, likely to be a prerequisite to progress on these methodological issues.*" (R. E. Brooks 1980)

In a lengthier critique (Sheil 1981) outlines not only concerns about methodology but also about the lack of impact of the psychological research on the computing community. Sheil begins by outlining the state of the art in 1981:

"*As practiced by computer science, the study of programming is an unholy mixture of mathematics (e.g. (Edsger Wybe Dijkstra 1997)), literary criticism (e.g. (Kernighan and Plauger 1974)) and folklore (e.g. (F. P. Brooks 1975)). However, despite the stylistic variation, the claims that are made are all basically psychological; that is, that programming done in such and such a manner will be easier, faster, less prone to error, or whatever*" (Sheil 1981), [citation format updated]

Sheil's argument, however, is not that these alternative styles are problematic, it is that the direct psychological study of languages as it was performed in 1981 was problematic.

Sheil points out several underlying patterns: that results that might appear strong for novices quickly vanish as the participants become more experienced, that it is easy to measure the effect of pathological designs but multiple practical alternatives are often difficult to distinguish, and that it is difficult to separate out the ways in which a treatment, such as the introduction of static typing, which

Sheil argues entails the introduction of structural typing, affects the subject performance. Sheil continues enumerating other methodological challenges to the experimental designs and reporting of the experiments published at that point, the strongest critique however is in the general handling and deployment of evidence in support of design.

"*Yet, as Shneiderman sadly notes in a retrospective of this work, "Flowchart critics cheered our results as the justification of their claims, while adherents found fault and pronounced their confidence in the utility of flowcharts in their own work" (Shneiderman 1980). It is in response to reactions like this that the use of psychology in computer science debates was earlier characterized as "ammunition.*" (Sheil 1981)

Sheil is concerned that, as the actually empirically supported conclusions are too weak, there is a tendency to overreach, blurring what is known with what is conjecture.

"*Another consequence of this dependency on computing is that behavioural researchers tend, possibly in an attempt to make their work appeal to computer scientists, to generalize far beyond their data… [discussing (Shneiderman 1980)] Detailed discussions of experimental results are interleaved with totally (empirically) unsupported opinions on programming style. Much of this material would be quite legitimate, intuitively based argument in a computer science debate. However its presentation as part of a discussion of empirical research completely blurs the distinction between data and intuition, inviting readers to reject data that do not support their preconceptions. This makes the entire empirical enterprise moot*". - (Sheil 1981)

And

"*The absence of a critical review process, coupled with the very considerable difficulty of research in this area and the constant tendency to drift into intuitively based argument and generalize far beyond what has been established, has created a pseudopsychology of programming*" (Sheil 1981)

Similar to Brooks, Sheil argues that the most pressing need is to establish a theory of programming skill.

"*The experimental investigation of such factors as the style of conditional notation is premature without some theory which gives some account of why they might be significant factors in programmer behavior.*" (Sheil 1981)

In a retrospective in 1986, (Curtis 1986) argues that these methodological critiques were coupled with a problem of the better designed studies tending to come after decisions had already been taken:

'*Computer science was little interested in weak empirical justifications for directions it had already taken, such as structured programming. Computer scientists cared more for deductive proofs than for the rejection of null hypotheses.*' (Curtis 1986)

Looking back, Curtis sees 1981 as a turning point, that in response to Sheil and Brooks future research of the field adopted a cognitive psychological approach rather than a human factors approach. However, Curtis somewhat laments that cognitive psychology was chosen. At this point in time, the first workshop on the Empirical Studies of Programmers was organised, and it was here where Curtis was presenting his retrospective.

5. Community of study: Empirical Studies of Programmers (1986 - 1997)

Although the most prominent spinout from the Software Psychology society was the ACM CHI series ((Shneiderman 1986)), a smaller group was convened to continue the specific focus on programming. Empirical Studies of Programmers (ESP)[1] first met in June 1986 in Rosslyn VA ((Shneiderman 1986)). The preface of the proceedings outlines the purpose of the research as guiding interventions to improve practice:

'*Broadly speaking, the basic assumption of researchers who study programmers is this: By understanding how and why programmers do a task, we will be in a better position to make prescriptions that can aid programmers in their task. For example, if we can understand how a maintainer, say, goes about comprehending a program, we should be in a good position to recommend changes in documentation standards that would enable the maintainer to more effectively glean from the documentation the necessary information. Similarly, recommendations for software tools and education should also follow.*' ((Soloway and Sitharama Iyengar 1987), pvii).

They outline a number of the challenges of carrying out experiments in this area, many of which persist to the current day. One is for the research to be multidisciplinary, requiring 'a healthy degree of sophistication in both programming and psychology in order to recognize what are the important research issues'. They also acknowledge a limitation of the work contained in the proceedings: that the research was examining 'programming in the small' whilst million line programs could be found in industry and 'software of such magnitude has not as yet received significant attention by researchers in the field'.

Curtis ((Curtis 1986)) shares this concern about the ecological validity of the studies carried out, suggesting that the series might be called "Empirical Studies of Student Programmers" or that the work will continue to be retrospective in focus on 'demonstrating already established cognitive phenomena' (p257), raising doubts as to the justification for experimental studies as a cost-effective method of influencing design.

Curtis argues for increased ecological validity in studies, considering not only the individual's cognition but also the broader organisational context in which programming takes place. (Soloway 1986) concludes the proceedings discussing an agenda for the continued importance of programming-in-the-small to identify 'baseline-issues', that is to catalogue the important and recurrent behaviours of programmers as well as to develop confidence in the research methodologies. He also goes on to suggest that controlled studies may 'not be that useful for initially studying programming-in-the-large. This type of methodology requires carefully worked out hypotheses be developed first, before the experiment.', going on to state that 'Almost by definition, programming-in-the-large violates the basic premises for a controlled study.' ((Soloway 1986), p266). Instead he suggests that alternative techniques such as 'talking-aloud' may be a richer source of data for theory building.

[1] http://www.ppig.org/news/2006-06-01/whatever-happened-empirical-studies-programmers

Even at this very early stage of the field a number of techniques and questions are outlined. For example, (Onorato and Schvaneveldt 1986) considers, in the proceedings for the first Empirical Studies of Programmers workshop, the difference in exploratory behaviour for communicating how to do something between programmers and non-programmers. This kind of question is now important in discussions of how to support 'end-user programmers' in an organisational context, as identified in both MacLean et al's pioneering Buttons project ((MacLean et al. 1990)) and Nardi's seminal text ((Nardi 1993)).

Similarly, (Spohrer and Soloway 1986) describe a project to understand high-frequency bugs, showing that there are bugs that occur very commonly in novice programs (such as off-by-one errors), but also that these are not the result of novices misunderstanding a language feature. They conducted this experiment by gathering a corpus using a rudimentary form of instrumented tooling. They augmented the operating system of the computer their participants were using (a VAX 750), and obtained a copy of each syntactically correct program submitted for compilation. They refer to this data as on-line protocols.

By the conclusion of the first workshop of the Empirical Studies of Programmers, many of the core methodological tensions (e.g. internal validity vs; ecological validity, and what form of evidence needs to be in place to have an effective impact on language design) were already established.

Over the course of the subsequent workshops these issues continued to be explored with the balance shifting towards the study of professionals (by ESP 5, 70% of subjects in studies were professionals, compared to 21% in ESP 1). The series concluded with what would have been ESP 8. The submissions were included in a special issue of the International Journal of Human-Computer Studies (IJHCS Volume 54, No 2, Feb 2001). As the Editorial of this final issue suggests (Katz, Petre, and Leventhal 2001), the need for research into the Empirical Studies of Programmers had grown as wider populations engaged in programming ranging from the growing business applications, to educational and end-user programming fields. However, after this 8th workshop, some of the core researchers moved into other communities.

The role of a US based workshop considering the empirical investigation of professional programmers has been taken up by the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) series discussed below. Human factors and HCI aspects have also been strongly represented at the Visual Language and Human Centric Computing (VL/HCC) IEEE symposium series, especially since the addition of "HCC" to the original focus on descriptions of novel kinds of visual syntax, which had an implicit concern with improving usability while seldom actually testing this ((A. F. Blackwell 1996)).

6. PLATEAU (2009 - present)

PLATEAU started in 2009 with the goal of considering and improving the efficiency of programmers by improving the usability of the languages and tools they develop with (Anslow, Markstrum, and Email: 2009). It was explicitly aimed at improving the visibility of the work on Human Computer Interaction to the programming language community. As such, PLATEAU tends to meet co-located with major software engineering conferences such as ICSE or OOPSLA/SPLASH.

PLATEAU has taken a software engineering perspective on the questions that ESP was interested in. It has a similar focus on professional and expert study (60% of studies that explicitly reported their sample, were of professional programmers compared to 23% at PPIG, discussed below).

The work that is published at PLATEAU includes corpus analyses, for example (Pritchard 2015) models the distribution of error messages from a system for novices learning to program as a precursor to systematically improving their usability. They compare their system for writing Python (Pritchard and Vasiga 2012) with data from Blackbox, which records data from the BlueJ Java editor (Brown et al. 2014).

PLATEAU also published empirical investigations into tools usage such as (Sadowski and Yi 2014)'s qualitative study into how developers at Google use tools for concurrent race detection, (Kabáč, Volanschi, and Consel 2015)'s controlled experiments with four professionals evaluating the ease of learning of DiaSuite, or (Campusano et al)'s controlled experiment investigating preference, developer productivity and comprehension in live programming robots.

Work at PLATEAU is often focussed around evaluation techniques such as (Hanenberg and Stefik 2015)'s call to build a community standard for the design of controlled experiments. (Kurtev, Christensen, and Thomsen 2016) outline a methodology for supporting the design of iterative or incremental improvements to programming languages. They observe that the 'evaluation' step of the incremental design of languages is often prohibitively expensive, especially if large scale empirical studies are used. Instead they propose a 'discount usability evaluation' method conducting a simple test observing participants completing basic tasks and categorising the problems they experienced by severity.

As well as these discussions and publications of empirical results on how programming is done, PLATEAU also provides a venue for higher level discussions such as (Ko 2016)'s description of the socio-technical roles of programming languages.

As would be expected given the focus on professional users, the PLATEAU community has in general been focussed on conventional uses of programming. This is in contrast to the sibling community of ESP and PLATEAU in Europe, PPIG, which has often provided a venue for the discussion of avant garde uses of programming such as bricolage programming (McLean and Wiggins 2010), choreography (Church, Rothwell, and Downie 2012) and education via robotics (Martin and Hughes 2011).

8. What works - the 'language wars' (2014 - present)

In 1981 Sheil criticised the construction of a 'pseudoscience of programming' through a combination of a lack of theory development, poor experimental practice and systematic overreach from weak empirical positions. Since 2014 there has been a resurgence of critique along a similar line, notably from Stefik and Hanenberg, that the standards of evidence used in programming research is inadequate, especially in contrast to the use of Randomised Control Trials (RCTs).

In the earlier critique, Sheil was primarily concerned with the integrity and scientific standing of programming language research, a concern that Stefik and Hanenberg also share. In ((Andreas Stefik et al. 2014)) they describe a systematic review of the papers in PPIG and PLATEAU, coded according to their own scheme for categories and using criteria from the campaign for evidence-based policy in education research in the USA known as the What Works Clearinghouse. As described in a critical evaluation of that campaign (an evaluation which the author claims was suppressed by the federal funders of WWC), in asking why it had attracted so much controversy, 'The answer to these questions can be summed up in two words: "math wars."' (Schoenfeld 2006).

The goal of the WWC in USA educational policy had been to ensure that educational initiatives resulted in quantifiable improvements, applying the same logic as the use of RCTs in publicly-funded health interventions. Although driven by a recognisable political agenda, education researchers are well aware of the futility of attempting RCTs in children's education, for factors that include the primarily political and social drivers of educational outcomes (e.g. race, gender, social deprivation) that political actors hope not to draw attention to when advocating curriculum change or technological intervention as a less costly panacea.

The desire for more clear cut quantifiable evidence in programming language design was also associated with a particular approach to programming education, hoping to demonstrate that Stefik's Quorum language (2017) had a more solid scientific basis than the more popular Scratch. Scratch had been introduced to schools from the more arts-based tradition of live, creative coding, rather than a focus on teaching conventional syntax, and this was a matter of concern for computer scientists who, as had occurred earlier with Dijkstra's polemic against BASIC, were concerned that the more easily-learned graphical syntax would damage students' understanding. Stefik originally implemented Quorum (and its predecessors) to support the needs of students with disabilities, but developed this into a campaign to make Quorum the first language for which every feature could be justified with a scientific study.

Advocates of the evidence-based approach, meeting the "what works" standards, focus on technical properties of programming language design, for example explicitly labelling the discussion of self-efficacy, tooling support, education or program comprehension as not related to programming language design ((Andreas Stefik et al. 2014), p228). Instead they concentrate on properties such as type systems, syntax and API design. In a direct challenge to established methods in Psychology of Programming, Stefik and colleagues concluded that only 1.1% of PPIG papers were both related to programming language design (by the above definition), and also met the standards proposed by the What Works Clearinghouse. They report that the corresponding number for Plateau is 14.3% and 16.7% for ESP.

They argue that this represents a fundamental weakness in programming language design research. However, they go on from there to suggest that this weak evidential base for language design may be an important cause of long-standing controversy that they describe as 'language wars' (Andreas Stefik and Siebert 2013)), which they argue have had substantial, negative, social consequences caused by the effort of creating, learning and adopting multiple languages. As an alternative they advocate for methodologies that contribute "what works" style evidence, such as RCT's of specific features. Examples of this approach based on RCTs are found in (Uesbeck et al. 2016), (Endrikat et al. 2014), (Andreas Stefik and Siebert 2013).

In the math wars in education research that the “language wars” phrase alludes to, the political drivers reflect conservative advocacy of labour utility, as imagined through the rote classroom “labour” of skill acquisition, practice drills, and quantitative assessment of the numbers of uniformly correct answers, as contrasted with a vision of education that emphasises creative experience and diversity of assessment. In education, these two poles are associated with alternative scientific agendas - on the creative side, qualitative and interpretive research that is politically informed, while on the utilitarian side, research emphasises quantitative assessment and RCTs to verify reductionist cognitive or perceptual models.

The same dynamics can be seen in programming language research, where a utilitarian labour focus emphasises the number of correct actions taken by the programmer implementing a well-defined specification, rather than programming as an exploratory, creative and diverse experience. As programming language research becomes more human-centric, in the recognition that the language is a user interface and that some account needs to be taken on the user, these two alternative perspectives have led to the language wars campaigners suggesting that programming language design should prefer some HCI research methods, but not others.

Large scale software application deployments do use RCT-like A/B tests to compare interface design alternatives, where there is a clear productivity measure that can be used by the company (e.g. numbers of sales or click-throughs). We note that these methods are especially relevant to *incremental* optimisation of programming language designs. This is in contrast to research such as that in the Visual Languages or Live Coding communities that focuses on novel paradigms, new styles of representation, and applications beyond traditional waterfall-style software engineering. Of course, the focus on novelty within VL/HCC may indeed miss opportunities for optimisation of existing language designs, which often have points of detail that could be improved through application of RCTs. A point of particular relevance for incremental RCTs is decisions that relate to the surface syntax of programming languages and environments. There are many details of syntax that have been chosen on an arbitrary basis, without clear evidence for the choice made. When supported by empirical evidence, we are able to determine, for example, whether a special assignment operator corrects frequent misconceptions that arise from misuse of the equals sign ((Mc Iver, n.d.)).

The natural desire of computer scientists to produce quantitative accounts of human behaviour ((A. F. Blackwell 2022)), combined with the particular kind of political drivers that expect educational policy to produce a mechanically trained, yet disempowered, workforce ((Hicks 2017)), combine in the software industry with those large companies whose business model require measurement and control of user’s attention ((Zuboff 2019), (Seaver 2022)). The result for the psychology of programming has been a constant struggle to take a well-informed approach to design guidance.

8. Cognitive dimensions (1989 - present)

The paper describing the broad layout of the language wars (Andreas Stefik and Hanenberg 2014) raises a number of questions that overlap with our research interests. Perhaps the largest point of difference is that they are particularly upset about the widespread use of Thomas Green’s Cognitive Dimensions of Notations framework ((Green 1990), (Green and Petre 1996), (Hadhrawi, Blackwell, and Church 2017)). The emphasis of CDs on the importance of the tool and environment suggested that reductive controlled comparison of syntax choice as the primary scientific agenda in language design might not actually be the most important question to investigate. The emphasis of CDs on the need for different solutions to different problems, insisting that a language should only be evaluated in relation to a particular profile of activities, also seemed contrary to the desire to identify objectively "best" features.

(A. Stefik and Hanenberg 2017) write:
For example, one common approach that we think lacks merit is the so-called cognitive dimensions of notations framework, a set of design principles conceived by Thomas R.G. Green in 1989 and expanded in a 1996 article. According to Google Scholar this influential article has been cited some 500 times, but Green's theory wasn't based on sound empirical evidence--by 1989 there had only been seven programming language design studies.

Emphasis on citation counts as a measure of scientific quality is not one that we would advocate, especially since the publication describing Stefik's own Quorum language ((Andreas Stefik and Ladner 2017)) has only been cited 23 times, which is less influential than we would hope for "the first language to use human-factors evidence from both field data and randomized controlled trials in its design" (the "language wars" paper has been cited many more times than the language itself, which certainly does not do justice to the original scientific or design objectives of Quorum).

The language wars authors were not the first to have criticised the intentions and scientific significance of Cognitive Dimensions. Green's work at the MRC Applied Psychology Unit, had been intended precisely to use empirical human factors techniques as a basis for engineering design, along with much other research that made the MRC-APU one of the founding centres of HCI research internationally ((Craik 1944), (Reynolds and Tansey 2003)). It was after 20 years of work on this problem that Green came to realize individual controlled experiments were not a practical or effective source of design guidance for the designers of new interactive systems. The intentions of Cognitive Dimensions were precisely to avoid the "death by detail" that made it infeasible to address design problems in this way, instead offering a theoretically informed "broad brush" vocabulary with which designers could be encouraged to discuss decisions and trade-offs that were cognitively relevant.

Some younger researchers have perceived this response to the need for "broad brush" design guidance as a lack of scientific rigour. In particular, those coming from human-factors engineering or business productivity contexts have hoped that problems of design could be reduced to more quantified performance formulas or objective observations of what is right. This was the driver for the "Physics of Notations" proposed by business school lecturer Daniel Moody ((Moody 2009)) - a critique that is even more highly cited than the above quote, even more critical of the supposedly scientific status of CDs (keeping in mind that CDs was always intended to be a resource for designers, not a scientific theory), and also driven by the desire for objective and measurable criteria. As it turns out, much of Moody's alternative "physics" was no less subjective than the CDs themselves, since supposedly "physical" facts such as semantic interpretation are always dependent on the observer. The Physics of Notations has become far more widely cited than CDs, and certainly has greater appeal to quantitative PL researchers, but meta-analysis of the many studies citing this work show that they are just as lacking in scientific rigour and replicability as Moody felt that Green had been ((Linden and Hadar 2018)).

The desire for a physics-based set of design principles for PLs, or a laboratory controlled trial for choosing the objectively best features, is typical of first-wave HCI ((Bødker 2006)), with its emphasis on human factors as an optimisable component within an engineering system. Perhaps CDs might be considered a part of the turn toward context that was characteristic of second-wave HCI. The possibility of focusing on a more diverse variety of creative experiences, including artistic practices as well as utilitarian ones, is a focus more characteristic of third-wave HCI. Blackwell and Fincher's suggestion that CDs might be regarded as a pattern language describing Patterns of User Experience (PUX) ((A. F. Blackwell and Fincher 2010)), offers a third-wave alternative to the design orientation of the field ((A. Blackwell 2015)), following first-wave Physics of Notations and second-wave original CDs.

9. Conclusion: Lack of consensus on how to improve human centric programming

As the breadth of the discussion above shows, in the 50 years since Dijkstra's speculation on the socio-cognitive implications of the goto statement, no consensus has arisen as to where and how to go about improving programming language usability.

Existing communities lend varying degrees of support to a collection of different methods: PLATEAU supports empirical methods for discussing professional software engineers, ICLC focuses on practice-based research for supporting artist programmers, and PPIG accepts contributions in a wide variety of forms ranging from participant ethnographies, to design discussions, theoretical framework and philosophical speculations.

There is an ongoing tension within the community as to what the scientific status of the research is, and what techniques are appropriate to address these problems. Even more so, there are tensions between what are the right questions for the field to try and address.

This brief history of the study of human centric programming language design has provided the background to those tensions, to be used as a reference for the ongoing work in our group, looking at what kinds of questions programming language designers face, what techniques are currently available for answering them, and what is missing

10. Acknowledgements

The authors would like to thank the community for many years of interesting discussions

11. References

Anslow, Craig, Shane Markstrum, and Emerson Murphy-Hill Email: 2009. "Evaluation and Usability of Programming Languages and Tools (PLATEAU) PLATEAU." https://ecs.wgtn.ac.nz/foswiki/pub/Main/TechnicalReportSeries/ECSTR10-12.pdf.

Arawjo, Ian. 2020. "To Write Code: The Cultural Fabrication of Programming Notation and Practice." In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 1–15. CHI '20. New York, NY, USA: Association for Computing Machinery.

Blackwell, Alan. 2015. "Patterns of User Experience in Performance Programming." In . Zenodo.

https://doi.org/10.5281/zenodo.19315.
Blackwell, Alan F. 1996. "Metacognitive Theories of Visual Programming: What Do We Think We Are Doing?" In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, 240. VL '96. USA: IEEE Computer Society.
———. 2017. "6,000 Years of Programming Language Design: A Meditation on Eco's Perfect Language." In *Conversations Around Semiotic Engineering*, 31–39. Cham: Springer International Publishing.
———. 2022. "Wonders without Number: The Information Economy of Data and Its Subjects." *AI & Society*, January. https://doi.org/10.1007/s00146-021-01324-8.
Blackwell, Alan F., and Sally Fincher. 2010. "PUX: Patterns of User Experience." *Interactions* 17 (2): 27–31.
Blackwell, Alan F., Marian Petre, and Luke Church. 2019. "Fifty Years of the Psychology of Programming." *International Journal of Human-Computer Studies* 131 (November): 52–63.
Bødker, Susanne. 2006. "When Second Wave HCI Meets Third Wave Challenges." In *Proceedings of the 4th Nordic Conference on Human-Computer Interaction: Changing Roles*, 1–8. NordiCHI '06. New York, NY, USA: Association for Computing Machinery.
Brooks, Frederick P. 1975. *The Mythical Man-Month : Essays on Software Engineering*. Reading, Mass.: Addison-Wesley Pub. Co.
Brooks, Ruven E. 1980. "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology." *Communications of the ACM* 23 (4): 207–13.
Brown, Neil Christopher Charles, Michael Kölling, Davin McCall, and Ian Utting. 2014. "Blackbox: A Large Scale Repository of Novice Programmers' Activity." In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 223–28. SIGCSE '14. New York, NY, USA: ACM.
Church, L., N. Rothwell, and M. Downie. 2012. "Sketching by Programming in the Choreographic Language Agent." *Proceedings of the*. https://pdfs.semanticscholar.org/17c3/e3b7530bc25c9e45fa4f7430a3fc54ba3db4.pdf.
Craik, K. J. W. 1944. "Medical Research Council Unit for Applied Psychology." *Nature* 154: 476–77.
Curtis, Bill. 1986. "By the Way, Did Anyone Study Any Real Programmers?" In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, 256–62. Norwood, NJ, USA: Ablex Publishing Corp.
Dijkstra, Edsger W. 1968. "Letters to the Editor: Go to Statement Considered Harmful." *Communications of the ACM* 11 (3): 147–48.
Dijkstra, Edsger Wybe. 1997. *A Discipline of Programming*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR.
Endrikat, Stefan, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. "How Do API Documentation and Static Typing Affect API Usability?" In *Proceedings of the 36th International Conference on Software Engineering*, 632–42. ICSE 2014. New York, NY, USA: ACM.
Evershed, D. G., and G. E. Rippon. 1971. "High Level Languages for Low Level Users." *Computer Journal* 14 (1): 87–90.
Goldstine, H. H., and J. von Neumann. n.d. "Planning and Coding of Problems for an Electronic Computing Instrument. Part II, Vol." *The Institute for Advanced Study Princeton, New*.
Green, T. R. G. 1990. "Cognitive Dimensions of Notations. People and Computers V: Proc. British Computer Society HCI'89 Conference." Cambridge University Press.
Green, T. R. G., and M. Petre. 1996. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework." *Journal of Visual Languages & Computing* 7 (2): 131–74.
Hadhrawi, Mohammad, Alan F. Blackwell, and Luke Church. 2017. "A Systematic Literature Review of Cognitive Dimensions." In *Proceedings of the 28th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2017)*, 13.
Hanenberg, Stefan, and Andreas Stefik. 2015. "On the Need to Define Community Agreements for

Controlled Experiments with Human Subjects: A Discussion Paper." In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, 61–67. PLATEAU 2015. New York, NY, USA: ACM.

Hicks, Mar. 2017. *Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing*. MIT Press.

Kabáč, Milan, Nic Volanschi, and Charles Consel. 2015. "An Evaluation of the DiaSuite Toolset by Professional Developers: Learning Cost and Usability." In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, 9–16. PLATEAU 2015. New York, NY, USA: ACM.

Katz, Irvin R., Marian Petre, and Laura Leventhal. 2001. "Editorial: Empirical Studies of Programmers." *International Journal of Human-Computer Studies* 54 (2): 185–88.

Kernighan, Brian W., and P. J. Plauger. 1974. *Elements of Programming Style*. New York, NY, USA: McGraw-Hill, Inc.

Knuth, Donald E. 1971. "An Empirical Study of FORTRAN Programs." *Software: Practice & Experience* 1 (2): 105–33.

Ko, Andrew J. 2016. "What Is a Programming Language, Really?" In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, 32–33. ACM.

Kurtev, Svetomir, Tommy Aagaard Christensen, and Bent Thomsen. 2016. "Discount Method for Programming Language Evaluation." In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (plateau 2016)*. Association for Computing Machinery. http://people.cs.aau.dk/~bt/PLATEAU2016/Preprint-PLATEAU2016-KurtevChristensenThomsen.pdf.

Linden, D. Van Der, and I. Hadar. 2018. "A Systematic Literature Review of Applications of the Physics of Notation." *IEEE Transactions on Software Engineering*, 1–1.

MacLean, Allan, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. 1990. "User-Tailorable Systems: Pressing the Issues with Buttons." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 175–82. CHI '90. New York, NY, USA: Association for Computing Machinery.

Martin, C., and J. Hughes. 2011. "Robot Dance: Edutainment or Engaging Learning." *Proceedings of the 23rd Psychology of Programming*. http://www.ppig.org/papers/23/14%20Martin.pdf.

Mc Iver, Linda. n.d. "The Effect of Programming Language on Error Rates of Novice Programmers." https://pdfs.semanticscholar.org/ac30/ee4129122006bbe2c1af6a935d958c416eb4.pdf.

McLean, A., and G. Wiggins. 2010. "Bricolage Programming in the Creative Arts." *22nd Psychology of Programming Interest*. http://www.academia.edu/download/30255078/22nd-eup-2.pdf.

Menabrea, Luigi Federico, and Ada King Countess of Lovelace. 1843. *Sketch of the Analytical Engine Invented by Charles Babbage, Esq*. Richard and John E. Taylor.

Moody, Daniel. 2009. "The ``Physics'' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering." *IEEE Transactions on Software Engineering* 35 (6): 756–79.

Nardi, Bonnie A. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA, USA: MIT Press.

Newell, Allen, and Stuart K. Card. 1985. "The Prospects for Psychological Science in Human-Computer Interaction." *Hum. -Comput. Interact.* 1 (3): 209–42.

Onorato, Lisa A., and Roger W. Schvaneveldt. 1986. "Programmer/Nonprogrammer Differences in Specifying Procedures to People and Computers." In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, 128–37. Norwood, NJ, USA: Ablex Publishing Corp.

Pierce, Benjamin C. 2002. *Types and Programming Languages*. 1st ed. The MIT Press.

Pritchard, David. 2015. "Frequency Distribution of Error Messages." *arXiv [cs.SE]*. arXiv.

http://arxiv.org/abs/1509.07238.
Pritchard, David, and Troy Vasiga. 2012. "CS Circles: An In-Browser Python Course for Beginners." *arXiv [cs.CY]*. arXiv. http://arxiv.org/abs/1209.2166.
"RELATING INDIVIDUAL DIFFERENCES IN COMPUTER PROGRAMMING PERFORMANCE TO HUMAN INFORMATION PROCESSING ABILITIES." 1977. https://search.proquest.com/openview/c39f5fd5a80bb3296095014ef2d2b881/1?pq-origsite=gscholar&cbl=18750&diss=y.
Reynolds, L. A., and E. M. Tansey. 2003. *The MRC Applied Psychology Unit: V. 16*. Edited by L. A. Reynolds and E. M. Tansey. Wellcome Witnesses to Twentieth Century Medicine S. London, England: Wellcome Trust Centre for the History of Medicine at UCL.
Sadowski, Caitlin, and Jaeheon Yi. 2014. "How Developers Use Data Race Detection Tools." In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, 43–51. ACM.
Schoenfeld, Alan H. 2006. "What Doesn't Work: The Challenge and Failure of the What Works Clearinghouse to Conduct Meaningful Reviews of Studies of Mathematics Curricula." *Educational Researcher* 35 (2): 13–21.
Seaver, Nick. 2022. *Computing Taste: Algorithms and the Makers of Music Recommendation*. University of Chicago Press.
Sheil, B. A. 1981. "The Psychological Study of Programming." *ACM Comput. Surv.* 13 (1): 101–20.
Shneiderman, Ben. 1980. *Software Psychology: Human Factors in Computer and Information Systems (Winthrop Computer Systems Series)*. Winthrop Publishers.
———. 1986. "No Members, No Officers, No Dues: A Ten Year History of the Software Psychology Society." *SIGCHI Bull.* 18 (2): 14–16.
Sime, M. E., T. R. G. Green, and D. J. Guest. 1973. "Psychological Evaluation of Two Conditional Constructions Used in Computer Languages." *International Journal of Man-Machine Studies* 5 (1): 105–13.
———. 1977. "Scope Marking in Computer Conditionals—a Psychological Evaluation." *International Journal of Man-Machine Studies* 9 (1): 107–18.
Soloway, Elliot. 1986. "What to Do next: Meeting the Challenge of Programming-in-the-Large." In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, 263–68. Ablex Publishing Corp.
Soloway, Elliot, and S. Sitharama Iyengar. 1987. *Empirical Studies of Programmers (Human-Computer Interaction Series)*. Intellect.
Spohrer, James G., and Elliot Soloway. 1986. "Analyzing the High Frequency Bugs in Novice Programs." In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, 230–51. Norwood, NJ, USA: Ablex Publishing Corp.
Stefik, A., and S. Hanenberg. 2017. "Methodological Irregularities in Programming-Language Research." *Computer* 50 (8): 60–63.
Stefik, Andreas, and Stefan Hanenberg. 2014. "The Programming Language Wars: Questions and Responsibilities for the Programming Language Community." In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 283–99. Onward! 2014. New York, NY, USA: ACM.
Stefik, Andreas, Stefan Hanenberg, Mark McKenney, Anneliese Andrews, Srinivas Kalyan Yellanki, and Susanna Siebert. 2014. "What Is the Foundation of Evidence of Human Factors Decisions in Language Design? An Empirical Study on Programming Language Workshops." In *Proceedings of the 22Nd International Conference on Program Comprehension*, 223–31. ICPC 2014. New York, NY, USA: ACM.
Stefik, Andreas, and Richard Ladner. 2017. "The Quorum Programming Language (abstract Only)." In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM. https://doi.org/10.1145/3017680.3022377.
Stefik, Andreas, and Susanna Siebert. 2013. "An Empirical Investigation into Programming Language

Syntax." *Trans. Comput. Educ.* 13 (4): 19:1–19:40.
Uesbeck, Phillip Merlin, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. "An Empirical Study on the Impact of C++ Lambdas and Programmer Experience." In *Proceedings of the 38th International Conference on Software Engineering*, 760–71. ACM.
Waite, William M., and Gerhard Goos. 1984. *Compiler Construction*. Springer-Verlag New York.
Weissman, Larry, and University of Toronto. Computer Systems Research Group. 1973. *Psychological Complexity of Computer Programs: An Initial Experiment*. Computer Systems Research Group, University of Toronto.
Zuboff, Shoshana. 2019. *The Age of Surveillance Capitalism: The Fight for a Human Future at the New Frontier of Power: Barack Obama's Books of 2019*. Profile Books.

Overcoming the Mental Set Effect in Programming Problem Solving

Agnia Sergeyuk, Sergey Titov, Yaroslav Golubev, Timofey Bryksin
JetBrains Research
{agnia.sergeyuk, sergey.titov, yaroslav.golubev, timofey.bryksin}@jetbrains.com

Abstract
This paper adopts a cognitive psychology perspective to investigate the recurring mistakes in code resulting from the mental set (Einstellung) effect. The Einstellung effect is the tendency to approach problem-solving with a preconceived mindset, often overlooking better solutions that may be available. This effect can significantly impact creative thinking, as the development of patterns of thought can hinder the emergence of novel and creative ideas. Our study aims to test the Einstellung effect and the two mechanisms of its overcoming in the field of programming. The first intervention was the change of the color scheme of the code editor to the less habitual one. The second intervention was a combination of instruction to "forget the previous solutions and tasks" and the change in the color scheme. During the experiment, participants were given two sets of four programming tasks. Each task had two possible solutions: one using suboptimal code dictated by the mental set, and the other using a less familiar but more efficient and recommended methodology. Between the sets, participants either received no treatment or one of two interventions aimed at helping them overcome the mental set. The results of our experiment suggest that the tested techniques were insufficient to support overcoming the mental set, which we attribute to the specificity of the programming domain. The study contributes to the existing literature by providing insights into creativity support during problem-solving in software development and offering a framework for experimental research in this field.

1. INTRODUCTION

In recent years, there has been a growing trend of applying knowledge from cognitive psychology to address problems in software engineering (Lenberg et al., 2015; Bidlake et al., 2020; Mangalaraj et al., 2014). This approach has shown its effectiveness in various areas, such as fault detection in software inspections (Anu et al., 2016), understanding the role of the cognitive load in program comprehension and maintenance (Fakhoury et al., 2018), and exploring the neural correlates of programming tasks (Floyd et al., 2017), among others. In this paper, we adopt a cognitive psychology perspective to examine habitual mistakes in code, aiming to shed light on the role of cognitive processes in software development tasks. By leveraging insights from cognitive psychology, we seek to deepen our understanding of the underlying cognitive mechanisms that contribute to coding errors and explore potential interventions to improve coding performance and foster creativity in software development tasks.

Habitual mistakes, *i.e.,* mistakes driven by a fixation on previous knowledge, in the context of programming can result in maintenance issues and software bugs (Glass, 2003). Some of these recurring mistakes or error-prone points in code may be attributed to the mental set (Einstellung) effect — a cognitive rigidness that occurs during problem-solving when a habitual solution is applied without considering other potentially more optimal and suitable alternatives, particularly in familiar contexts (Luchins, 1942). The mental set effect can significantly impact creative thinking, as the development of patterns of thought can hinder the emergence of novel ideas that may be better suited for the task at hand (Davis, 1999; Wiley, 1998; Kilgour, 2006).

In the existing research, techniques for overcoming the Einstellung effect were studied in a laboratory on relatively abstract mathematical tasks in the form of water-jar problems, or anagrams, labyrinths, etc. (Chen & Mo, 2004; Lovett & Anderson, 1996; Louis Lee & Johnson-Laird, 2004). It was shown that one of the possible ways to weaken the effect is a change in the direction of awareness and cognitive control: its enhancement on the target task or refocusing on an additional task (Tukhtieva, 2016, 2014). Research also show that people who were instructed to intentionally forget the habituated solution of some problem relied less on it, *i.e.*, the Einstellung effect weakened for them (Tempel & Frings, 2019; Storm & Angello, 2010). Our aim was to test the Einstellung effect and the mechanisms of its overcoming in more real-world circumstances and in the field of programming.

Our study investigated the Einstellung effect in a sample of 129 Python developers. We conducted an online pre-experimental survey to identify participants who exhibited habitual mistakes (mental sets) that were relevant to our research. Among them, 57 individuals (with a mean experience in Python of 25 months) demonstrated at least one mental set. The final sample for our study consisted of 39 participants who successfully completed all experimental procedures.

For the experiment, we developed an online Flask[1] application with an embedded Ace9 code editor[2] serving as the programming environment. In this environment, participants were presented with two sets of tasks and one of three interventions between those sets. The tasks were carefully selected to specifically test the Einstellung effect, with each task having at least two solutions: one habitual (indirect) solution, and one "correct" (direct) solution that is recommended in style guides of the used programming language and more efficient in terms of code execution time. Participants were assigned to one of three groups: **Control** (no intervention), **Change** (task-irrelevant change in the color scheme of the environment), or **Change and Forget** (the same change in the color scheme plus an additional "forget cue" instruction). Following the experiment, participants were asked to rate the helpfulness and comfortability of the intervention they received (if any) on a scale from 0 to 5, and provide reasons for switching or not switching from the indirect solution to the direct one during the second set of tasks.

The results of our study shed light on the effectiveness of the tested interventions in overcoming mental sets among Python developers. Despite the participants' reported engagement in the process of rethinking their solutions (in 54% of cases), our findings indicate that the provided interventions were unhelpful (M = 0.97, SD = 1.18 on a scale from 0 to 5) and insufficient for the switch from habitual indirect solutions to the recommended direct solutions.

In summary, our study contributes to the existing literature by providing insights into creativity support during problem-solving in software development, offering a framework for experimental research, and sharing open-source code and data for future investigations in this field.[3]

2. BACKGROUND

2.1. The Einstellung Effect and Creativity

The Einstellung effect, also known as the mental set effect, was originally demonstrated by Luchins in 1942 (Luchins, 1942). In Luchins' experiments, participants were given a series of 11 problems that involved imagining using three water jars of different volumes to measure a certain amount of water. The tasks were divided into two sets. The first set had a fixed solution — *e.g.*, getting 100 liters with 21, 127, and 3 liters jars might be solved as $(127 - 21 - 2 * 3)$, while the second set could be solved using either the same approach or a more direct and simpler approach — *e.g.*, the task to get 18 liters with 15, 39, and 3 liters jars might be solved in two ways: a habituated one $(39 - 15 - 2 * 3)$ or a simple one $(15 + 3)$. The habituated approach, referred to as the **indirect** solution, required more time and effort, while the direct approach — **direct** solution — was quicker and less habitual but still known. The findings showed that direct approaches to solving the second set of tasks were less frequent compared to indirect approaches, indicating the presence of the mental set effect, which was operationalized as a difference in frequencies of indirect and direct solutions (Guetzkow, 1951).

Subsequent studies replicated Luchins' findings (McKelvie, 1990) and applied the concept of the mental set to different types of problems: some of them did not require specific prior knowledge or skill (Chen

[1]Flask framework: `https://flask.palletsprojects.com/en/2.3.x/`
[2]Ace9 code editor: `https://ace.c9.io/`
[3]All the code used for the experiment and the data analysis is publicly available on Zenodo: `https://doi.org/10.5281/zenodo.5893501`

& Mo, 2004; Lovett & Anderson, 1996; Louis Lee & Johnson-Laird, 2004), and in others, the mental set was assumed to be already formed based on the knowledge stored in the long-term memory (Bilalić et al., 2008; Wiley, 1998; Saariluoma, 1992).

The mental set was proven to inhibit creativity by leading individuals to approach new problems with preconceived notions or rigid thinking patterns based on past experiences or familiar solutions (Wiley, 1998; Kilgour, 2006). Therefore, overcoming the mental set may be necessary to unlock creative and innovative solutions.

In our current research, we investigate the Einstellung effect from the perspective of the Adaptive Control of Thought theory (ACT-R), which posits that the effect is a result of selection mechanisms in human cognition (Anderson, 1993; Ritter et al., 2019). ACT-R is a comprehensive theory of cognition that seeks to explain how humans acquire, represent, and process knowledge to perform various cognitive tasks, such as problem solving. According to this framework, the likelihood of selecting a solution for a given problem depends on the history-of-success and distance-to-goal rates that are internally attributed to the solution, and this likelihood changes as these rates change (Lovett & Anderson, 1996; Öllinger et al., 2008). Therefore, overcoming the mental set may be achieved by proving the solution unsuccessful in terms of the task or modifying the characteristics of the task's goal. Previous studies on overcoming the Einstellung effect have mainly been conducted in laboratory settings using abstract mathematical tasks, such as water-jar problems or anagrams. However, our goal is to investigate the effect and its mechanisms in more realistic contexts, specifically in the field of programming.

In the context of software engineering, the Einstellung effect can be demonstrated in coding practices and code style. Unlike bugs or logical errors, coding style issues may not have an immediate effect on the functionality of the code, which can lead to them becoming ingrained habits among developers. As a result, developers may struggle with adopting suggested code style guidelines, even when presented with feedback or recommendations. This can lead to error-prone code and reduced maintainability (Martin, 2009; Glass, 2003). To address this issue, several tools have been developed to detect and highlight coding style problems, and in some cases, even automatically fix them (Smirnov et al., 2021; Wiese et al., 2017; Birillo et al., 2022; Blau & Moss, 2015). These tools aim to provide direct feedback to developers to help them identify and correct coding style issues. However, these tools often do not take into account known cognitive mechanisms that underlie the mental set effect.

2.2. Overcoming of the Einstellung Effect

Our research aimed to test two approaches that have been proposed in prior works to overcome the mental set effect (Tukhtieva, 2011, 2014; Vallée-Tourangeau et al., 2011; Tempel & Frings, 2019; Storm & Angello, 2010). According to previous research, the Einstellung effect may arise due to the mechanization of a particular solution, where the problem solver becomes automatic in their approach. To overcome this effect, one needs to shift from automaticity to consciously controlled actions. This can be achieved by triggering a reevaluation of the solution's history-of-success and distance-to-goal rates (Lovett & Anderson, 1996), and by restructuring the situation in terms of connections and relationships between objects (Wertheimer & Wertheimer, 1959).

One way to de-automatize cognitive processes is to **diversify task conditions**, which can affect distance-to-goal rates (Lovett & Anderson, 1996), stimulate conscious activity, and facilitate generation and testing of different hypotheses (V. Allakhverdov, 2009). For instance, systematic changes in the irrelevant parameters of the task can complicate it, leading to the activation of conscious control (V. Allakhverdov, 2008). This implies that changing aspects of the task that are irrelevant to the solution could be a potential strategy to overcome mechanization or the mental set effect (Tukhtieva, 2014; Vallée-Tourangeau et al., 2011; Luchins & Luchins, 1950). In the context of the water-jar problem, the given volume of water is a relevant aspect, but the way the jars are presented, such as their color or whether they are physical or virtual, does not affect the solution (Luchins & Luchins, 1950). Tukhtieva (Tukhtieva, 2011, 2014, 2016) conducted detailed studies on the effect of task-irrelevant changes on overcoming the mental set and found that when water jars were presented as digital images with a dramatic change in the background of the task, such as from a neutral one-colored background to bright and colorful photographs, it reduced the effect of the mental set and helped individuals to overcome it.

Another approach to trigger reevaluation of solutions and overcome the mental set is through **intentional forgetting**, which involves giving direct instruction to "forget the tasks and solutions from before" (Tempel & Frings, 2019; Storm & Angello, 2010) and which might affect the internal history-of-success rating (Lovett & Anderson, 1996). This technique is adapted from research on intentional forgetting using the list method (Basden et al., 1993). In the list method, two lists of words, *e.g.*, furniture pieces, are sequentially presented. The first list is presented with an instruction to remember it, and after the participant memorized it, the list is marked with an instruction to forget it under some pretext, such as defining the list as needed only for warm-up. Next, the second list is presented with an instruction to remember it. After a certain period of time, the subject's ability to recall the material from both lists is tested. This method is designed to label information that should not be processed and, therefore, remembered, requiring the suppression of such information using cognitive control. This process, as suggested by previously mentioned research, may be effective in overcoming the mental set effect.

We took a unique approach by combining the existing experimental paradigm of the Einstellung effect testing with the known intervention techniques and applied them in the field of programming. This novel application allowed us to explore uncharted territory and draw conclusions about the domain specificity and toolification potential of cognitive phenomena. By bridging the gap between cognitive psychology and programming, we aimed to uncover new insights and contribute to the understanding of how cognitive processes influence creative problem-solving in programming tasks.

3. METHODOLOGY

In our study, we utilized Luchins' paradigm (Luchins, 1942) and adapted it for coding tasks in order to compare two approaches for overcoming the Einstellung effect in a sample of developers. The first approach involved **diversifying task conditions** by changing a task-irrelevant aspect, specifically the color scheme of the development environment. The second approach included an **intentional forgetting** cue along with the color scheme change.

To investigate the effectiveness of the interventions in overcoming the Einstellung effect, we formulated the following research questions:

- **RQ1.** What is the prevalence of the Einstellung effect among Python developers?
- **RQ2.** Do the provided interventions facilitate a shift in problem-solving strategies from indirect to direct, indicating the overcoming of the mental set?
- **RQ3.** Do the provided interventions facilitate solution de-automatization?
- **RQ4.** Do the reported helpfulness and comfortability scores, as measures of the usability of the interventions, correlate with each other and with the overcoming of the mental set?

In this section, we provide a detailed description of the sample, materials, interventions, data collection, and analysis procedures. First, we outline the sample characteristics. Next, we describe the study materials, which include the adapted programming tasks serving as a problem-solving proxy, as well as the interventions applied to overcome the Einstellung effect. We then detail the data collection process and the analysis procedures employed to address the formulated research questions.

3.1. Sample

To gather our sample, we administered a screening questionnaire to identify Python developers who consistently use less efficient (indirect) code, despite being aware of more direct and efficient alternatives, indicating a mental set for tasks relevant to our research. The questionnaire included informed consent and inquiries about participants' experience with Python in months. Participants then completed a programming task (see Figure 1). This task consisted of four steps, which were counterparts to tasks used

```
Provided: fun = "awesome"

 1. Check if the type of the given variable is str.

 2. If it is, iterate through the variable characters and
 add a "-" symbol to each one. Save the result as a list.
 (For example, "cat" -> ["c-", "a-", "t-"])

 3. Iterate through the objects of the resulting list and
 print the index of each object and its value.

 4. Make a list that includes all the received indexes,
 sum them up, and divide by the list's length.

Print the result.
```

Effective solution

```python
fun = "awesome"
```

Task 1

```python
if isinstance(fun, str):
```

Task 2

```python
    letters = list(map(lambda x: x + "-", fun))
```

Task 3

```python
for i, letter in enumerate(letters):
    print(str(i) + letters[i])
```

Task 4

```python
nums = range(len(letters))
print(sum(nums) / len(letters))
```

Less effective solution

```python
fun = "awesome"
```

Task 1

```python
if type(fun) == str:
```

Task 2

```python
    letters = []
    for i in fun:
        letters.append(i + "-")
```

Task 3

```python
for i in range(len(letters)):
    print(f"{i}{letters[i]}")
```

Task 4

```python
my_sum = 0
for i in range(len(letters)):
    my_sum += i
print(my_sum / len(letters))
```

Figure 1 – The screening task and its solutions.

in the subsequent experiment. They were also asked to select familiar Python functions from a provided list, including both target functions (considered indirect and direct in our research) and other functions not used in the study serving as noise. This allowed us to assess whether participants used the indirect function in the tasks, despite being familiar with the direct function based on the checklist.

We extended invitations to participate in the primary stages of the study to respondents who exhibited a mental set for the target Python functions. Specifically, we invited developers who satisfied both of the following criteria: (a) their responses to the given problem contained indirect solutions, as presented in Figure 1, and (b) they indicated familiarity with both indirect and direct functions in the checklist. This yielded an invitation to 57 out of 129 respondents who completed the questionnaire to participate in the primary experimental part of the research. However, not all the invitees could complete this step, resulting in a final sample size of 39 individuals.

3.2. Materials

The crucial part of our experiment was finding the appropriate programming problems that could serve as the Einstellung tasks. It was decided to focus on the mistakes in code style rather than bugs, as the former don't make the code un-executable and due to that might be easily habituated. We identified four Python problems from the *Wemake-python-styleguide*[4] that could be solved in two different ways, one of which was more efficient time-wise and recommended as a "correct" one in style guides:

- **Iterate through the list and return every item's index**. This task might be solved using either enumerate or range, the first option is preferred.
- **Compare types**. Two possible solutions for this task imply using the isinstance or type functions, the first option is preferred.
- **Sum things**. *Wemake-python-styleguide* indicates that for such tasks, using for loops with the += assigning operator inside indicates that one iteratively sums elements of the collection. However, this is what the built-in sum function does. Thus, the latter should be used.
- **Iterate through a string**. One could do this (1) with a loop, (2) using list comprehension, or (3) using the built-in lambda function. The latter is more straightforward. However, it is recommended to use more readable list comprehension. That is why we marked only the loop-based solution as an indirect one, and the other two were considered direct.

A more detailed discussion of the recommended variants and a comparison of their performance can be found in our online appendix available on Zenodo.[1]

[4]Wemake-python-styleguide: `https://wemake-python-stylegui.de/en/latest/`

Further in the text, we use the names of direct (in terms of our research) functions as labels of the corresponding mental set types (*e.g.*, enumerate as a label of the mental set for range).

To effectively assess the impact of experimental treatments on mental set overcoming, we designed two sets of four previously described programming tasks. Each task in both sets required participants to provide solutions that could be attained through either direct or indirect functions. The second set of tasks was intentionally structured to have the same underlying structure and require the same computational logic as the tasks in the first set, but the wording and input data were changed. This approach allowed us to make an apples-to-apples comparison of the solutions provided by participants across various treatments and reach definitive conclusions about their ability to overcome the mental set.

As a medium for our experiment, we developed an online Flask application using the Python language with an embedded Ace9 code editor as the programming environment.

3.3. Intervention

To ensure ecological validity (Orne & Holland, 1968), our experimental treatment aimed to replicate real-life situations. To achieve this, we selected task parameters and cues for the intervention that could be easily implemented in an Integrated Development Environment (IDE) without significant modifications, either through a simple plugin or existing environmental features. Previous research has used irrelevant changes, such as background color or representational form changes in various tasks, from simple math equations to anagram solving and memory tasks (Moroshkina & Gershkovich, 2008; Gershkovich, 2011; Tukhtieva, 2014; Vallée-Tourangeau et al., 2011; Luchins & Luchins, 1950). Drawing insights from this research, we adapted and applied similar principles in our study. As a result, we formed three intervention groups:

- The **Change** group was presented with only the task-irrelevant change. Specifically, we changed the color scheme of the coding environment from the habitual Light or Dark to the contrast one, using the two most common, standard, and most contrast themes of the embedded Ace9 code editor: *Eclipse* as the light one and *Monokai* as the dark one. This change was selected as it is available in most coding environments and is not dependent on the task or the editor, making it easily implementable. Additionally, such a change is similar to a real-life situation where people switch their gadget's theme from light to dark and vice versa.

- The **Change and Forget** group received the same change as the Change group, along with the forget cue. The forget cue was designed based on the classical list-method paradigm used in forgetting research and was adapted to our study's goals. Specifically, participants in this group were instructed to forget their previous solutions and tasks as they were part of the warm-up phase, and to proceed to the main stage.

- The **Control** group received no intervention except the instruction to continue.

3.4. Data Collection

The experimental procedure, depicted in Figure 2, is described in detail below:

- After a delay of one week from the day the screening questionnaire was filled out (to allow the memory trace of the questionnaire and its tasks to cool down), we sent an email to the selected participants containing a URL to access the experiment asynchronously.

- As the first step, participants were asked to choose their habitual color scheme (Light or Dark) for their programming environment. This was done to compare solutions created in an environment where the mental set had been repeatedly habituated and used, with solutions generated under relatively new circumstances.

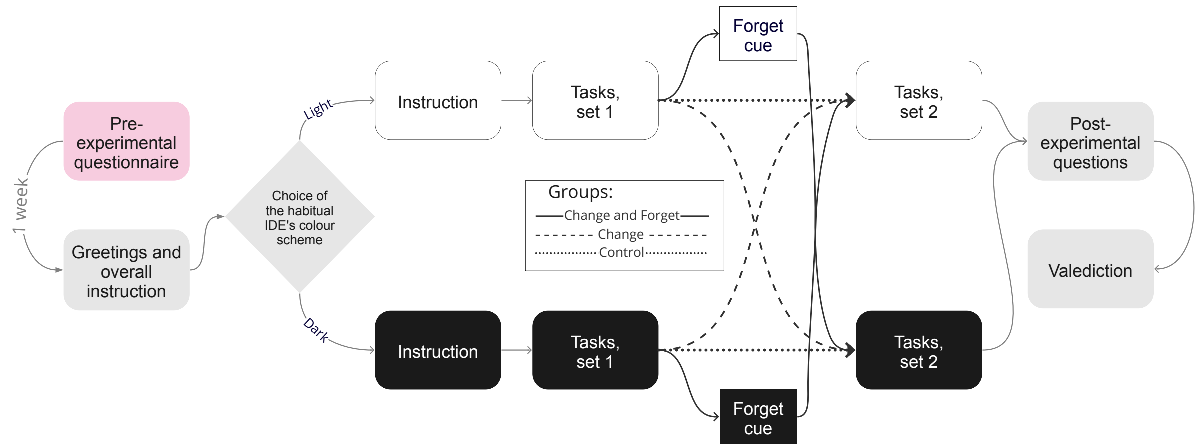


Figure 2 – The procedure of the experiment.

- Next, participants were presented with a general experiment description stating, "In the next 10 minutes, you will have to solve a set of problems in Python. We will not check the correctness of the code, but it is important to us that the solution to the problem is as algorithmically efficient as possible. Please use the most efficient code to solve the problem."

- Following this, a randomized sequence of four programming problems, as described in Section 3.2, was presented in the chosen color scheme.

- Following this step, the participants received an intervention according to the group to which they were assigned: Change, Change and Forget, or Control, as described earlier in the section.

- After that, the second set of programming problems was provided. The set consisted of four problems similar to those in the first set, but with the altered wording containing different input data to manipulate.

- Finally, an open-ended question was presented to determine the reasons for applying a particular type of solution for the second set of tasks. For the groups with the change of the task-irrelevant aspect, two 5-point Likert scales (Likert et al., 1934) were presented to mark the comfortability level of the change in the presented color scheme (if any), as well as the level of its helpfulness in terms of solving the second set of tasks.

During the experiment, we gathered various data such as the participants' preferred habitual color scheme, the solutions they generated for each task, and their response time for each task. A response time difference served as a proxy measure of the cognitive load, with a larger difference indicating higher cognitive processing demands (Rheem et al., 2018). Additionally, we assigned each participant a label indicating their group and the corresponding mental set effect they exhibited.

3.5. Data Analysis

All the obtained data was analyzed in Python. Statistical analysis was performed using Scipy (Virtanen et al., 2020), NumPy (Harris et al., 2020), Pandas (McKinney, 2010), and Pingouin (Vallat, 2018).

To correspond with the research questions, the following statistical hypotheses were formulated:

- **H1.** There is a statistically significant difference in the experience between individuals who exhibited the Einstellung effect and those who did not. An independent T-Test was conducted on the initial questionnaire data to test this hypothesis.

- **H2.** The provided interventions result in a significant shift in problem-solving strategies from indirect to direct, indicating the overcoming of the mental set. Data visualization was performed to examine if there is any solution change.

- **H3.** The provided interventions lead to a significant difference in response time between corresponding blocks of tasks. The Kruskal-Wallis H test was utilized to determine whether there is an effect of group on the response time difference.

- **H4.** The reported helpfulness and comfortability scores are significantly correlated with each other and with the overcoming of the mental set. To examine these correlations, we used Spearman's correlation coefficient.

4. RESULTS

4.1. Prevalence of the Mental Set (RQ1, H1)

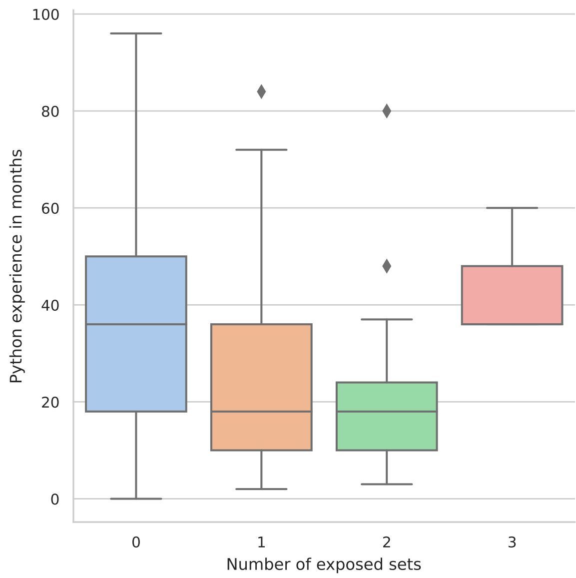


Figure 3 – Number of sets exposed by participants of different months of experience.

Our analysis showed that approximately 44% of the participants in our sample exhibited the mental set. Among the 129 individuals who took part in the screening questionnaire, 57 showed evidence of using less efficient, indirect functions to solve programming tasks despite being aware of more efficient, direct functions. Notably, we found a statistically significant difference in the number of months of Python experience between individuals who exhibited the mental set and those who did not, with those displaying the mental set having an average of 25 months of experience compared to an average of 37 months for those without the mental set ($T(126.998) = 2.806$, $p = .006$). This finding suggests that programmers with less experience are more prone to producing code with errors. This increased error rate can be attributed to their tendency to rely heavily on habituated knowledge, resulting in a fixation on familiar solutions without considering alternative approaches.

In terms of the types of mental set displayed, 41 participants exhibited only one type, 13 exhibited two types, and 3 exhibited three types, with none displaying all four types (see Figure 3). The mental set related to the isinstance function was the most common, observed in 26 cases, followed by the lambda mental set in 19 cases. The enumerate mental set appeared in 17 solutions, while the sum mental set was identified in 14.

4.2. Overcoming the Einstellung Effect (RQ2, H2)

The primary experimental phase of the study was completed by 39 participants. In the second set of tasks, two participants from the Control group demonstrated a shift from indirect to direct problem-solving strategies. As a result, we cannot definitively conclude that the intervention was effective in overcoming the mental set. The potential factors that may have contributed to these results will be discussed at length in Section 5.

4.3. Response Time Difference (RQ3, H3)

The mean response time plot in Figure 4 sheds light on the cognitive load experienced by participants during the experiment. Specifically, we examined the response time difference between corresponding tasks, which provided insights into the process of solution de-automatization in each group. Interestingly, our results indicate that the participants in the intervention groups (Change, Change and Forget) did not experience a significantly larger response time difference than those in the Control group ($H(2) = 1.403$, $p = .496$). Given that response time is widely used as a measure of cognitive load in experimental research (Huang et al., 2009; Zheng & Cook, 2012), our findings suggest that the intervention did not noticeably affect the de-automatization of habitual problem-solving strategies among the participants in the intervention groups.

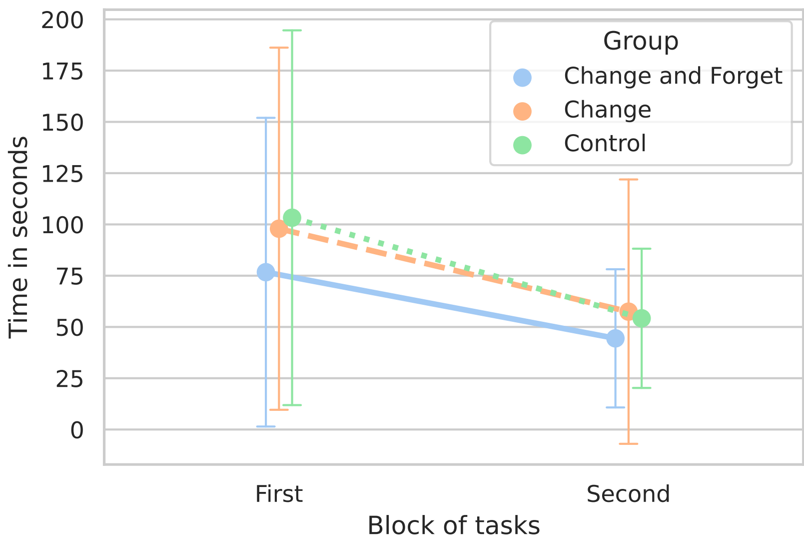


Figure 4 – Mean response time for each set of tasks by the group of participants.

4.4. Helpfulness and Comfortability of Intervention (RQ4, H4)

The plot shown in Figure 5 displays the participants' estimates of the helpfulness and comfortability of the intervention on a scale ranging from 0 to 5. Our analysis revealed that the participants perceived the color scheme change as unhelpful ($M = 0.97$, $SD = 1.18$). This is not surprising given that there was no significant change in the solutions. Additionally, we found that the comfortability level of the color scheme change varied among participants, as indicated by the relatively high standard deviation ($M = 2.55$, $SD = 2.01$). These results suggest that the interventions provided in our research may not be effective in overcoming the mental set and that in the future research it is important to consider individual differences which may play a role in how participants perceive the interventions.

With 29 participants in the intervention groups, we were unable to find a statistically significant correlation between the levels of helpfulness and comfortability ($r(29) = .3$, $p = .36$). However, we did find a weak but significant positive correlation between the helpfulness scores and the difference in response time ($r(116) = .272$, $p = .003$). On the other hand, we found no significant correlation between the comfortability and the difference in response time ($r(116) = .003$, $p = .972$). These results suggest that

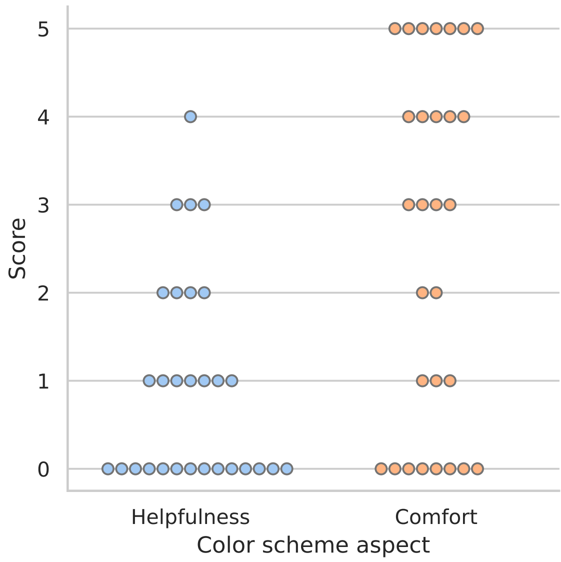


Figure 5 – Comfortability and helpfulness scores of the color scheme change.

there is a logical connection between how participants rated the helpfulness of the intervention and the observed change in solution speed after the treatment.

4.5. Insights from the Open-ended Question

In the final step of our study, we asked participants to explain their decision to change or maintain their problem-solving approach when presented with the problem the second time. Of the 39 participants, 22 made some changes to their solution, but the majority of these changes did not involve attempting to overcome the mental set. Rather, they were minor edits such as fixing typos or making minor code adjustments. Only 2 participants from the Control group demonstrated the successful mental set overcoming by recalling a more efficient function. Based on our qualitative analysis of the data, we conclude that the presented tasks may not have been challenging enough for some participants and that the mental set on such material could potentially be overcome without any intervention.

Here are some quotes from the gathered answers. One of the participants wrote:

> *I went through the tasks quickly and didn't even notice that tasks were repeated. It was less comfortable to think on a white background, it was more difficult to come up with a solution that I would like. Indeed, in some tasks, ideas of other solutions came, but in such simple tasks it was difficult to use a different solution. After all, there is a call to 1-2 functions of the standard library.*

We also got such feedback:

> *In one of the tasks of the second set, I decided that instead of taking up space with a "for" loop I can use a list expression to iterate through the elements, as well as using ".join". On the other hand, such code would be more difficult to understand when looking for errors.*

5. DISCUSSION

The primary objective of our study was to investigate the efficacy of two interventions, namely task-irrelevant change and intentional forgetting techniques, in aiding programmers in overcoming the mental set (Einstellung) effect.

Our study found that 44% of the participants exhibited the mental set with the prevalence among less experienced programmers, as those who exhibited the mental set had significantly less experience with Python than those who did not. This finding aligns with the research on the relationship between expertise and problem-solving, which has shown that less experienced individuals often rely more heavily on habituated knowledge and less on analytical reasoning than experts (Bilalić et al., 2008; Chi, 2006).

In terms of the types of mental set displayed, the isinstance mental set was the most common, followed by the lambda mental set. This finding suggests that these mental sets may be particularly ingrained in programmers' minds and could be difficult to overcome without explicit interventions. Future research could examine the effectiveness of different interventions in addressing these specific mental sets.

However, interventions provided in our research — a task-irrelevant change in the developing environment and an instruction to forget the previously provided information — were not efficient in terms of mental set overcoming. The main finding of our study is the fact that there was no switch between indirect and direct solutions in our experimental groups. Only two participants in the Control group exhibited a shift from indirect to direct problem-solving strategies in the second set of tasks.

Several factors may have contributed to these results.

It is probable that, despite the familiarity with both direct and indirect solutions, which was tested by our survey, the advantages of the direct solution were not explicit for all participants.

Previous studies on the impact of task-irrelevant change on the mental set have argued that the change disrupts the fixation by signaling the task as new, thereby prompting the generation of a fresh solution strategy (Tukhtieva, 2011, 2014). However, these findings were obtained in studies involving numerical and mathematical tasks. In our research, the tasks were presented in a natural language and required writing several lines of code in a specific programming language. It is possible that the difference in the form and the level of task abstraction, specifically the lack of items to manipulate, contributed to the absence of the Einstellung effect overcoming.

It is also possible that the intervention was not strong enough to facilitate significant changes in problem-solving strategies. In previous research, the change in task-irrelevant parameters was much more noticeable than in our study. The change involved more visual noise and a different modality during presentation (Tukhtieva, 2016; Luchins & Luchins, 1950). However, we intentionally avoided such a dramatic change to keep experimental conditions as close to real-life situations as possible. We do acknowledge that a change to a less common color scheme, as seen in Tukhtieva's work with bright and colorful photographs, could produce a more noticeable effect. However, implementing such a change in a code editor could harm its usability, as suggested by the low comfortability ratings in our study. One feasible solution to introduce a significant yet comfortable task-irrelevant change is to request users who are habituated to the solution, for example, in a Web-based Massive Open Online Courses, to switch from a Web-based coding environment to a standalone IDE.

Additionally, some studies have shown that the stability of a mental set can be attributed to the differences between the mental set formation environment and the intervention environment (Chen & Mo, 2004). As we did not include the step of mental set formation in our design, it is possible that due to that, the provided interventions did not affect the Einstellung effect.

Based on our results, it seems that the interventions did not have a substantial effect on the participants' cognitive load and conscious control, as evidenced by the response time difference between the corresponding tasks. Previous research has suggested that an increase in the cognitive load can be beneficial for conscious control, as it can help individuals pay more attention to the task at hand and solve it more efficiently, leading to better de-automatization and solution efficacy (M. Allakhverdov et al., 2019; Moroshkina & Gershkovich, 2008). However, the results of our study indicate that this may not have been the case in our research.

The results of our study suggest that the interventions we provided may not have been effective in breaking the participants' habitual problem-solving strategies. These findings emphasize the importance of further research aimed at exploring alternative approaches to promote de-automatization and facilitate creative problem-solving in the field of programming.

6. THREATS TO VALIDITY

To ensure the validity of our findings, we considered several threats to validity.

6.1. Construct Validity

The construct validity of this study refers to the extent to which our measurements accurately capture and represent the presence of the mental set and the effectiveness of the interventions employed. In other words, it assesses how well our chosen measurements align with the underlying concepts they are intended to measure, ensuring the reliability and accuracy of our findings.

The primary threat to construct validity of this study is the absence of a standardized measurement tool specifically designed for assessing the mental set phenomenon. Although we used a screening questionnaire and a programming task to identify participants with relevant mental sets, there is a possibility that our methods did not capture some participants with a mental set. However, it is important to note that we focused our attention on the four common mental sets in programming, which mitigates some concerns regarding our ability to capture all types of mental sets.

Another threat to the construct validity is the measurement of the interventions' effectiveness in overcoming the mental set. To address this, we used two complementary measures as a proxy for problem-solving de-automation: (a) the number of switches from indirect to direct solutions and (b) the response time difference between the sets of tasks.

It is also important to note that our sample size was relatively small and there is a possibility of the selection bias, as participants with a mental set may have been more likely to participate in the study to improve their problem-solving skills. This may limit the generalizability of our findings.

6.2. Internal Validity

Internal validity refers to the extent to which our experimental design accurately identifies the effect of the intervention on the mental set.

The primary threat to the internal validity of our study is the presence of alternative explanations for the observed results. We used a randomized controlled design to minimize the impact of confounding variables. To mitigate the risk that the improvement of participants' performance was due to task repetition, rather than the interventions, we used two different problems with different structures, rather than repeating the same task. Additionally, we carefully arranged and varied the sequence in which participants received the different interventions to control for order effects. Also, we attempted to control for potentially confounding variables such as age and education level, by selecting a diverse sample. However, it is possible that there were other unmeasured variables that could have impacted our results.

6.3. External Validity

External validity refers to the extent to which our findings can be generalized to other contexts. Our study was conducted on a sample of Python programmers who voluntarily participated in the study. Therefore, our results may not generalize to programmers who use different programming languages. Moreover, our sample included participants who are more likely to be motivated to learn and improve their programming skills, which may not be representative of all programmers. To increase the generalizability of our results, future research should replicate our study with participants from different programming communities or with different problem structures.

7. CONCLUSION

In this work, we explore the intersection of cognitive psychology and software engineering. Specifically, we address the issue of cognitive rigidness in programming.

We provide evidence that the mental set (Einstellung) effect, which refers to the tendency to stick to habitual solutions rather than exploring new ones during problem-solving, is prevalent among Python developers who have not exceeded their second year of familiarity with the language. Our study examines the effectiveness of two interventions — task-irrelevant change and intentional forgetting technique

— in overcoming the Einstellung effect. Although participants reported engagement in the rethinking process, our findings suggest that these interventions were insufficient in facilitating a switch from indirect to direct solutions. We suggest that a more dramatic and unhabitual change of the working environment, such as altering font size/type or switching from a web-based to a standalone IDE, may be helpful in overcoming cognitive rigidity.

Our study contributes to the existing literature on creativity support during problem-solving in software development and provides a framework for future experimental research. By sharing our open-source code and data, we offer resources for further investigations in this field. We believe that studies combining cognitive psychology and programming, like this one, can shed light on ways to help practitioners to be more efficient in the field of software engineering.

8. References

Allakhverdov, M., Scott, T., Chernaya, A., & Allakhverdov, V. (2019). Cognitive control of irrelevant stimulus changes. *Modern technologies in medicine*, *11*(1), 63–67.

Allakhverdov, V. (2008). Awareness as a result of choice. *Psychology in Russia: State of the Art*, *1*(1), 136–152.

Allakhverdov, V. (2009). *Reflections on the science of psychology with an exclamation point*. Faculty of Philology, St. Petersburg State University.

Anderson, J. R. (1993). Production systems and the ACT-R theory. In *Rules of the mind* (pp. 17–44). Lawrence Erlbaum Associates.

Anu, V., Walia, G., Hu, W., Carver, J. C., & Bradshaw, G. (2016). Using a cognitive psychology perspective on errors to improve requirements quality: An empirical investigation. In *Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 65–76).

Basden, B. H., Basden, D. R., & Gargano, G. J. (1993). Directed forgetting in implicit and explicit memory tests: A comparison of methods. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *19*(3), 603–616.

Bidlake, L., Aubanel, E., & Voyer, D. (2020). Systematic literature review of empirical studies on mental representations of programs. *Journal of Systems and Software*, *165*, 110565.

Bilalić, M., McLeod, P., & Gobet, F. (2008). Inflexibility of experts — Reality or myth? Quantifying the Einstellung effect in chess masters. *Cognitive psychology*, *56*(2), 73–102.

Birillo, A., Vlasov, I., Burylov, A., Selishchev, V., Goncharov, A., Tikhomirova, E., ... Bryksin, T. (2022). Hyperstyle: A tool for assessing the code quality of solutions to programming assignments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V.1* (pp. 307–313).

Blau, H., & Moss, J. E. B. (2015). Frenchpress gives students automated feedback on java program flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 15–20).

Chen, Z., & Mo, L. (2004). Schema induction in problem solving: a multidimensional analysis. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *30*(3), 583–600.

Chi, M. T. (2006). Laboratory methods for assessing experts' and novices' knowledge. In *The Cambridge handbook of expertise and expert performance* (pp. 167–184). Cambridge University Press.

Davis, G. A. (1999). Barriers to creativity and creative attitudes. In *Encyclopedia of creativity* (Vol. 1, pp. 165–174). Academic Press.

Fakhoury, S., Ma, Y., Arnaoudova, V., & Adesope, O. (2018). The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)* (pp. 286–296).

Floyd, B., Santander, T., & Weimer, W. (2017). Decoding the representation of code in the brain: An fmri study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (pp. 175–186).

Gershkovich, V. (2011). Explicit ignoring of information during the performance of memory tasks. *Journal of Russian & East European Psychology*, *49*(3), 68–83.

Glass, R. L. (2003). *Facts and fallacies of software engineering*. Addison-Wesley Professional.

Guetzkow, H. (1951). An analysis of the operation of set in problem-solving behavior. *The Journal of General Psychology*, *45*(2), 219–244.

Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... others (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362.

Huang, W., Eades, P., & Hong, S.-H. (2009). Measuring effectiveness of graph visualizations: A cognitive load perspective. *Information Visualization*, *8*(3), 139–152.

Kilgour, A. M. (2006). Improving the creative process: Analysis of the effects of divergent thinking techniques and domain specific knowledge on creativity. *International Journal of Business and Society*, *7*(2), 79–107.

Lenberg, P., Feldt, R., & Wallgren, L. G. (2015). Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and software*, *107*, 15–37.

Likert, R., Roslow, S., & Murphy, G. (1934). A simple and reliable method of scoring the thurstone attitude scales. *The Journal of Social Psychology*, *5*(2), 228–238.

Louis Lee, N., & Johnson-Laird, P. (2004). Creative strategies in problem solving. In *Proceedings of the Annual Meeting of the Cognitive Science Society* (Vol. 26, pp. 813–818).

Lovett, M. C., & Anderson, J. R. (1996). History of success and current context in problem solving: Combined influences on operator selection. *Cognitive psychology*, *31*(2), 168–217.

Luchins, A. S. (1942). Mechanization in problem solving: The effect of einstellung. *Psychological monographs*, *54*(6), 1–95.

Luchins, A. S., & Luchins, E. H. (1950). New experimental attempts at preventing mechanization in problem solving. *The Journal of General Psychology*, *42*(2), 279–297.

Mangalaraj, G., Nerur, S., Mahapatra, R., & Price, K. H. (2014). Distributed cognition in software design: An experimental investigation of the role of design patterns and collaboration. *MIS Quarterly*, *38*(1), 249–274.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

McKelvie, S. J. (1990). Einstellung: Luchins' effect lives on. *Journal of Social Behavior and Personality*, *5*(4), 105–121.

McKinney, W. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51–56).

Moroshkina, N., & Gershkovich, V. (2008). Conscious control in mnemonic and learning tasks. *Vestnik of Saint-Petersburg University. Sociology*, *12*(2), 91–100.

Öllinger, M., Jones, G., & Knoblich, G. (2008). Investigating the effect of mental set on insight problem solving. *Experimental psychology*, *55*(4), 269–282.

Orne, M. T., & Holland, C. H. (1968). On the ecological validity of laboratory deceptions. *International Journal of Psychiatry*, *6*(4), 282–293.

Rheem, H., Verma, V., & Becker, D. V. (2018). Use of mouse-tracking method to measure cognitive load. In *Proceedings of the human factors and ergonomics society annual meeting* (Vol. 62, pp. 1982–1986).

Ritter, F. E., Tehranchi, F., & Oury, J. D. (2019). ACT-R: A cognitive architecture for modeling cognition. *Wiley Interdisciplinary Reviews: Cognitive Science*, *10*(3), e1488.

Saariluoma, P. (1992). Error in chess: The apperception-restructuring view. *Psychological research*, *54*(1), 17–26.

Smirnov, O., Lobanov, A., Golubev, Y., Tikhomirova, E., & Bryksin, T. (2021). Revizor: a data-driven approach to automate frequent code changes based on graph matching. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1242–1246).

Storm, B. C., & Angello, G. (2010). Overcoming fixation: Creative problem solving and retrieval-induced forgetting. *Psychological Science*, *21*(9), 1263–1265.

Tempel, T., & Frings, C. (2019). Directed forgetting in problem solving. *Acta psychologica*, *201*, 102955.

Tukhtieva, N. (2011). Irrelevant parameters' change in luchins tasks as the condition for the choice of new ways of the decision. *Izvestia of the Russian State Pedagogical University*(129), 98–101.

Tukhtieva, N. (2014). Influence of the types of change of irrelevant task parameters on the installation effect. *Bulletin of St. Petersburg University. Sociology*(3), 41–48.

Tukhtieva, N. (2016). Influence of irrelevant information on the contradictions. In *Humanities, Social Sciences and Environment Conference* (pp. 164–171).

Vallat, R. (2018). Pingouin: statistics in python. *Journal of Open Source Software*, *3*(31), 1026.

Vallée-Tourangeau, F., Euden, G., & Hearn, V. (2011). Einstellung defused: Interactivity and mental set. *Quarterly Journal of Experimental Psychology*, *64*(10), 1889–1895.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... others (2020). Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, *17*(3), 261–272.

Wertheimer, M., & Wertheimer, M. (1959). *Productive thinking* (Vol. 1). Harper New York.

Wiese, E. S., Yen, M., Chen, A., Santos, L. A., & Fox, A. (2017). Teaching students to recognize and implement good coding style. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (pp. 41–50).

Wiley, J. (1998). Expertise as mental set: The effects of domain knowledge in creative problem solving. *Memory & cognition*, *26*(4), 716–730.

Zheng, R., & Cook, A. (2012). Solving complex problems: A convergent approach to cognitive load measurement. *British Journal of Educational Technology*, *43*(2), 233–246.

Automatic, Suggestive Feedback in Algorithm Visualisation Exercises

Artturi Tilanterä
Department of Computer Science
Aalto University
artturi.tilantera@aalto.fi

Abstract
This short paper discusses an ongoing doctoral research on the field of computing education research. The subject is undergraduate computing education at universities: teaching basic data structures and algorithms, meaning the principles of designing computer programs that solve tasks efficiently, without wasting computing time or memory. The research aims to deepen knowledge on what are students' difficulties in learning the subject, and how educational tools, such as visualisations and automatically assessed exercises, could help the student. Published and preliminary results display the technical and theoretical complexity of the subject.

1. Introduction
Computing has become ubiquitous: consider electric vehicles, mobile devices, balancing production and consumption of renewable energy, robotics, or Internet of Things. We need scientists and engineers capable of understanding computing regardless of whether their major is in computer science. University studies of computing typically begin with a basics of programming course on the idea of "How to instruct the computer to do new tasks?"

One typical course after the very basics introduces *data structures and algorithms* (DSA). The key aspect of this course is efficiency, as in "How to make the computer solve new tasks within reasonable processing time and memory?" Classic problems introduced on an DSA course are sorting a collection of data items according to some property, keeping the collection organized and rapidly searchable while adding and deleting items, and exploring alternative routes in a map or more abstract problem.

Teaching and learning programming is not a trivial psychological task. Since 1960s *computing education research* has revealed that for humans, it is hard to grasp how a computer runs a program, and how to formulate precise instructions for a computer. (Guzdial & du Boulay, 2019) Correspondingly, understanding data structures and algorithms is not trivial either, as it requires learning even more abstract concepts.

2. Background
One approach to improve the teaching of any STEM subject (Science, Technology, Engineering, and Mathematics) is to list the important concepts that students should learn. Moreover, the list can be refined into a set of multiple-choice questions which test students' knowledge across different courses in different universities. This carefully selected and validated set of questions is called a *concept inventory*. A concept inventory for data structures and algorithms has been built recently. (Porter et al., 2019)

Regardless of quality of instruction, students may struggle by learning an incorrect mental model while thinking they understood the subject; this is called a *misconception*. For example, a physics misconception is that summer is warmer than winter, because the Earth is closer to the sun on its orbit. Misconceptions can be seen to relate concept inventories in a way that they are popular way of students diverging from the correct mental model, even encountering confusion later in their studies. DSA misconceptions are known to exist for various subtopics, for example recursion (Götschi, Sanders, & Galpin, 2003) and heaps (Seppälä, Malmi, & Korhonen, 2006).

Data structures and algorithms are abstract, time-dependent concepts which are dynamic by nature: an algorithm processes data over time following certain instructions, and the data is converted into a result of computation. Therefore *algorithm visualisation* has been studied as a pedagogical tool: data structures can be represented as figures of one- or two-dimensional tables (arrays), chains (lists), trees, and networks (graphs). For example, Figure 1 shows an example of a visualisation of a graph: *nodes* labeled by letters A–P may represent road intersections, while the lines between the nodes, *edges*, can represent road segments, with numbers representing the length of the roads. The figure is a part of an image sequence depicting how the computer finds shortest paths from the starting node A (Dijkstra's algorithm). Essentially, the algorithm adds edges to the tree of shortest paths one by one. The brown color represents all possible road segments, while the yellow color represents choices already made by the algorithm.

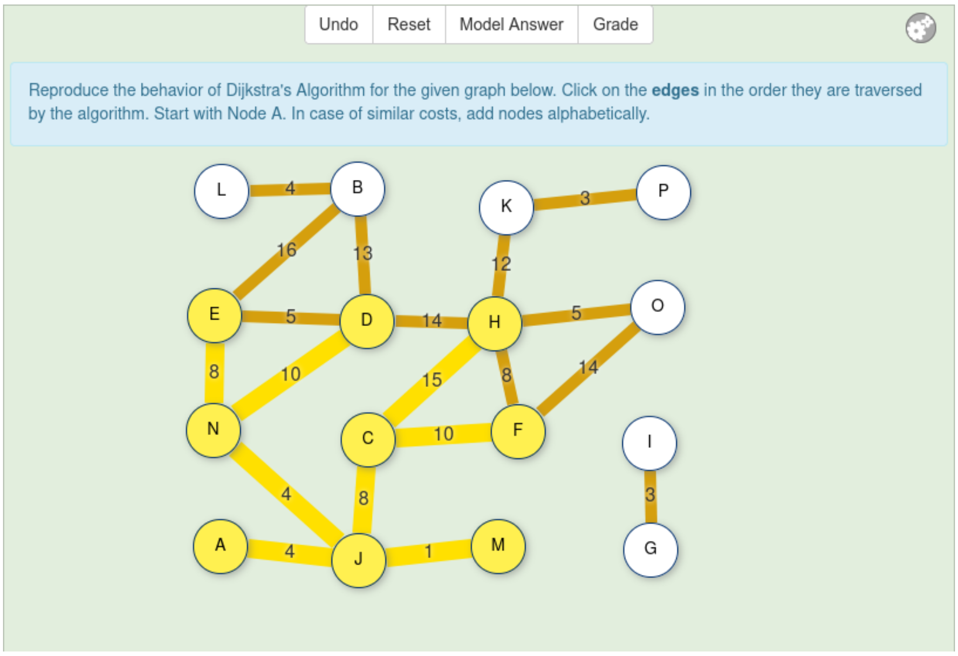


Figure 1 – A visual algorithm simulation exercise on Dijkstra's algorithm for single-source shortest paths.

Algorithm visualisation as a learning aid has been studied for decades. Many computing educators believe in the power of visualisation as a learning aid. However, the educational effectiveness of visualisation depends on what the student does with it, not the careful design of a visual presentation. A student can *engage* many ways with the visualisation. It seems that interaction (e.g. constructing, responding to questions) has significant learning effect compared to passively viewing e.g. an animation. (Naps et al., 2002) A meta-research of the algorithm visualisation (Shaffer et al., 2010) revealed that the visualisation tools are often low-quality, emphasizing easy topics, and hard to find for instructors. Therefore, to support instructors with a collection of algorithm visualisations, an open-source, community-based interactive textbook named OpenDSA was established (Shaffer, Karavirta, Korhonen, & Naps, 2011; *OpenDSA*, n.d.).

One particular type of engaging algorithm visualisation is *visual algorithm simulation* (VAS) exercises, where the student simulates the steps of an algorithm with certain input. For example, the whole Figure 1 is a screenshot of a VAS exercise about Dijkstra's algorithm. The input (graph) is given, and the student must click on the edges of the graph one by one in the same order as the algorithm adds them to the tree of shortest paths. The exercise is assessed automatically: when the student clicks the Grade button, they receive a grade depending on the number of correct steps in the simulation. These kind of interactive, visual exercises with automatic assessment are currently implemented with the JSAV software in OpenDSA (Karavirta & Shaffer, 2013, 2016), while Matrix / TRAKLA2 (Malmi et al., 2004) is an older software with the same principles. The exercise in Figure 1 has been implemented with JSAV.

3. Research problems and methods

The overarching research problem in this doctoral research is to give automated feedback for learners in VAS exercises at a Data Structures and Algorithms course in tertiary education. The feedback should be *suggestive*, meaning that it contains information on how to proceed. The following general research questions characterise the publications.

- RQ1. How data structures and algorithms should be taught?
- RQ2. What are students' key difficulties in DSA?
- RQ3. How can we give automatic, corrective feedback in DSA exercises, especially VAS?

An answer to RQ1 and RQ2 is sought by a systematic literature review on computing education research related to data structures and algorithms. It is expected that we will have an organised overview of this subtopic listing the key methods, problems, and misconceptions. For example, how has the field of algorithm visualisation evolved since of the report (Shaffer et al., 2010) ?

Another perspective to RQ2 begins with an existing study (Nelson et al., 2020), which includes a problem set to tests students' knowledge related to both programming and DSA. Next, a qualitative research will test the problem set empirically. What information does the problem set reveal in practice when students are asked to solve it?

VAS exercises can be used to study student's misconceptions of DSA (see, e.g. (Seppälä et al., 2006)), thus providing answers to RQ2. Quantitative analysis of students' solutions to VAS exercises combined with think-aloud interviews of students solving the same exercises is expected to reveal common misconceptions among students. Similarly, it would be interesting to see whether the same misconceptions appear in students' solutions to DSA-related programming exercises.

RQ3 seeks to improve the quality of automatically assessed DSA exercises. The JSAV-based VAS exercises only report the correctness of the answer to the student. The student might like to see where they diverged from the correct solution, or even an automatic hint on how to proceed. Correspondingly, the automatic assessment of programming exercises is relatively easy to implement by designing unit tests for the student's program and reporting to the student which tests did not pass. Answers to RQ1 and RQ2 might help constructing elaborate, suggestive automatic feedback. The technical challenges of producing such feedback is a research problem itself.

4. Publications

Publication I. Algorithm visualisation and the Elusive Modality Effect

I assisted a research related to the cognitive effects of multimedia instruction, a so-called *modality effect*. The research involved an experiment where students were exposed to different versions of learning videos of the Dijkstra's algorithm, while tested with multiple-choice questions and a VAS exercise Figure 1. The experiment failed to show the existence of the modality effect, which seems to imply that the phenomenon is more complex than expected. However, simultaneously, the piloted a software to record students' solutions to the VAS exercise on Dijkstra's algorithm in a real learning context: an electronic material of a DSA course provided by the A+ Learning Management System (Karavirta, Ihantola, & Koskinen, 2013).

Publication II. Towards a JSON-based Algorithm Animation Language

The software development experience related to the modality effect research (Zavgorodniaia et al., 2021) was discussed in another paper, (Tilanterä, Mariani, Korhonen, & Seppälä, 2021). The conclusion was that software and storage format which the authors developed for recording solutions to JSAV-based VAS exercises required further development. The paper presents new requirements for the software.

5. Emerging results

This section describes emerging results which are not yet published.

(Submitted) Empirical Evaluation of Differentiated Assessment of Data Structures: The Role of Prerequisite Skills. Marjahan Begum, Pontus Haglund, Ari Korhonen, Filip Strömbäck, Artturi Tilanterä.

The authors evaluated the DSA-related question set designed in (Nelson et al., 2020) with students on a DSA course in two institutions. We interviewed total 18 students on their written answers to the question set. Qualitative content analysis of the results revealed that some of the questions do indeed reveal students' fragile knowledge about basics of programming or DSA. However, the extent to which we can pinpoint student's lack of knowledge varies by question. Moreover, there seems to be skills which are not taught explicitly on an introductory programming course, but which help students on the following DSA course, such as program tracing and the reasoning about constraints of program execution; these are called *middle-ground skills*.

(Work in progress) Students' Misconceptions of Dijkstra's Algorithm in Visual Algorithm Simulation. Artturi Tilanterä, Ari Korhonen, Otto Seppälä, Juha Sorva.

The software described in Publication II has been developed further. There is now a formal specification of the storage format as JSON Schema. The recorder software has been updated according to the redesign of the storage format (Mariani, Sänger, & Tilanterä, n.d.). We have collected hundreds of students' solutions to the exercise in Figure 1 with the updated software. Moreover, we have conducted think-aloud interviews where some students solve the same exercise. This mixed-methods research aims to find hypotheses for misconceptions in the interviews and then strengthen the evidence by quantitative analysis of the automatically collected solution data.

(Work in progress) Computing Education Research on Data Structures and Algorithms. Artturi Tilanterä.

Related to RQ1, I have conducted a preliminary literature review as a part of my doctoral studies. It seeks answers to the following questions: (i) What literature exists on computing education research related to data structures and algorithms? (ii) How should the topics in the existing literature be categorized? The material is:

1. Jan Vahrenhold's publications (University of Münster, 2022).
2. Proceedings of SIGCSE from year 2000 onward (ACM, 2022b)
3. Proceedings of ITiCSE from year 2000 onward (ACM, 2022a)
4. The bibliography in Christopher Hundhausen's dissertation (Hundhausen, 1999)

I have first screened the publications by title and then by abstract. Rudimentary topic-based categorization reveals the following. Basic DSA (linear structures, trees, graphs, hashing, sorting, complexity) has been discussed in 204 publications. Teaching of advanced topics, such as algorithmic techniques (e.g. dynamic programming, approximation, heuristics), and applications of algorithms (e.g. data compression, computational geometry, networking, optimization) have been studied to a lesser extent (22 and 23 publications, respectively). Based on the preliminary results, there is a plan to conduct a more comprehensive literature review on the topic which could lead to a journal publication or a concept inventory.

6. Conclusion

This short paper summarized two years of doctoral research related to teaching data structures and algorithms. Besides empirical experiments, the research has involved developing suitable research software and conducting a literature review. The research problem is manyfold, involving misconceptions, middle-ground skills, visualisation, and automatic assessment and feedback.

7. References

ACM. (2022a). *Iticse conference - proceedings.* Retrieved 2022-08-11, from https://dl.acm.org/conference/iticse/proceedings

ACM. (2022b). *Sigcse conference - proceedings.* Retrieved 2022-08-11, from https://dl.acm.org/conference/sigcse/proceedings

Götschi, T., Sanders, I., & Galpin, V. (2003). Mental models of recursion. In *Proceedings of the 34th sigcse technical symposium on computer science education* (pp. 346–350). New York, NY, USA: ACM. doi: 10.1145/611892.612004

Guzdial, M., & du Boulay, B. (2019). The history of computing education research. In S. A. Fincher & A. V. Robins (Eds.), *The cambridge handbook of computing education research* (p. 11–39). Cambridge University Press. doi: 10.1017/9781108654555.002

Hundhausen, C. D. (1999). *Toward effective algorithm visualization artifacts: Designing for participation and communication in an undergraduate algorithms course* (Doctoral dissertation, Graduate School of the University of Oregon). Retrieved 2023-01-11, from https://eecs.wsu.edu/~veupl/proj/dis/Tech%20Report.pdf

Karavirta, V., Ihantola, P., & Koskinen, T. (2013, 7). Service-oriented approach to improve interoperability of e-learning systems. In *2013 ieee 13th international conference on advanced learning technologies* (p. 341-345). IEEE. doi: 10.1109/ICALT.2013.105

Karavirta, V., & Shaffer, C. A. (2013). Jsav: The javascript algorithm visualization library. In *Proceedings of the 18th acm conference on innovation and technology in computer science education* (pp. 159–164). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2462476.2462487 doi: 10.1145/2462476.2462487

Karavirta, V., & Shaffer, C. A. (2016, 4). Creating engaging online learning material with the jsav javascript algorithm visualization library. *IEEE Transactions on Learning Technologies*, *9*(2), 171-183. doi: 10.1109/TLT.2015.2490673

Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppälä, O., & Silvasti, P. (2004). Visual algorithm simulation exercise system with automatic assessment: Trakla2. *Informatics in Education*, *3*(2), 267–288. doi: 10.15388/infedu.2004.19

Mariani, G., Sänger, J., & Tilanterä, A. (n.d.). *Jsav exercise recorder.* Retrieved 2023-04-13, from https://github.com/Aalto-LeTech/jsav-exercise-recorder/tree/jaal2.0 (Software version 2.0)

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., … Velázquez-Iturbide, J. A. (2002). Exploring the role of visualization and engagement in computer science education. In *Working group reports from iticse on innovation and technology in computer science education* (p. 131–152). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/960568.782998 doi: 10.1145/960568.782998

Nelson, G. L., Strömbäck, F., Korhonen, A., Albluwi, I., Begum, M., Blamey, B., … Monga, M. (2020). Assessing how pre-requisite skills affect learning of advanced concepts. In *Proceedings of the 2020 acm conference on innovation and technology in computer science education* (p. 506–507). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3341525.3394990 doi: 10.1145/3341525.3394990

Opendsa. (n.d.). Retrieved 2023-04-13, from https://opendsa-server.cs.vt.edu/

Porter, L., Zingaro, D., Liao, S. N., Taylor, C., Webb, K. C., Lee, C., & Clancy, M. (2019). Bdsi: A validated concept inventory for basic data structures. In *Proceedings of the 2019 acm conference on international computing education research* (pp. 111–119). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3291279.3339404 doi: 10.1145/3291279.3339404

Seppälä, O., Malmi, L., & Korhonen, A. (2006). Observations on student misconceptions—a case study of the build – heap algorithm. *Computer Science Education*, *16*(3), 241 - 255. doi: 10.1080/08993400600913523

Shaffer, C. A., Cooper, M. L., Alon, A. J. D., Akbar, M., Stewart, M., Ponce, S., & Edwards, S. H. (2010, 8). Algorithm visualization: The state of the field. *ACM Trans. Comput. Educ.*, *10*(3). Retrieved from https://doi.org/10.1145/1821996.1821997 doi: 10.1145/1821996.1821997

Shaffer, C. A., Karavirta, V., Korhonen, A., & Naps, T. L. (2011). Opendsa: Beginning a community active-ebook project. In *Proceedings of the 11th koli calling international conference on computing education research* (p. 112–117). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2094131.2094154 doi: 10.1145/2094131.2094154

Tilanterä, A., Mariani, G., Korhonen, A., & Seppälä, O. (2021). Towards a json-based algorithm animation language. In *2021 working conference on software visualization (vissoft)* (p. 135-139). doi: 10.1109/VISSOFT52517.2021.00026

University of Münster. (2022). *Publications - jan vahrenhold.* Retrieved 2022-08-11, from https://www.uni-muenster.de/Informatik.AGVahrenhold/en/personen/prof.dr.janvahrenhold/publikationen.shtml

Zavgorodniaia, A., Tilanterä, A., Korhonen, A., Seppälä, O., Hellas, A., & Sorva, J. (2021). Algorithm visualization and the elusive modality effect. In *Proceedings of the 17th acm conference on international computing education research* (p. 368–378). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3446871.3469747 doi: 10.1145/3446871.3469747

Bots in Software Engineering and their impact on development

Linda Erlenhov
Chalmers University of Technology
linda.erlenhov@chalmers.se

Keynote abstract

Bots that support software development ("DevBots") are seen as a promising approach to deal with the ever-increasing complexity of modern software engineering and development. Existing DevBots are already able to relieve developers from routine tasks such as building project images or keeping dependencies up-to-date. In this talk I will outline what what DevBots are, how developers use and relate to bots, and will end with a few key takeaways:

- Not all tools that developers refer to as bots are utilising some type of AI/ML/NLP
- Software engineers expect to become “better” and “more productive” using DevBots
- Interactions need to be carefully thought out, especially due to the collaborative nature of DevBots

How A Data Structure's Linearity Affects Programming and Code Comprehension: The Case of Recursion vs. Iteration

Aviad Baron
Dept. Computer Science
The Hebrew University
aviad.baron@mail.huji.ac.il

Dror G. Feitelson
Dept. Computer Science
The Hebrew University
feit@cs.huji.ac.il

Abstract
Iteration and recursion are two ways to traverse data structures. But when is one of them preferable over the other? Some researchers argue that it is natural to use recursion to process recursively defined data structures, such as linked lists and trees. In order to test this, we conducted two experiments of writing and understanding code segments related to the processing of different data structures, with about 100 participants in each. The results showed that the way the structure is defined is not a significant factor affecting the nature of the processing. On the other hand, the main factor that influenced the selected approach was having a "linear mindset", either due to the linearity of the structure (where data items are arranged in a linear fashion, as in a list), or the linearity of the processing path (where data items are visited in a linear manner, without branching). When we process a linear structure like a linked list, the nature of its processing tends to be more iterative; a similar but weaker effect is obtained when we process a linear path in a tree, for example when looking for an item in a binary search tree. But when we need to understand code, recursion is problematic in the event that the processing is divided both before and after the recursive call, and especially when the processing after the call (the so-called "passive flow") is non-trivial. These results provide insights into the cognitive factors which affect how programmers interact with code. They can be explained by the observation that linear processing and simple recursion (with a trivial or non-existent passive flow) are more suitable for the serial thinking of humans.

1. Introduction
Data structures are used to store and organize data. This organization often also reflects the way the data is processed. But does it *dictate* the way they are processed? In this study we examine how the topology of the structure affects its processing, and specifically the choice of recursive versus iterative processing.

Recursion is a basic construct in computer programming, where a function calls itself. A possible alternative is to use iteration, where a set of operations is performed repeatedly within a loop. In both cases this continues until a specific stopping condition is met.

The concept of recursion is also used in data structures. For example, a linked list is commonly defined recursively, in that each element contains a pointer to the remainder which is also a list. Thus a linked list is either:

- The empty list, represented by `None`, or
- A node that contains some data and a reference to a linked list

Given this definition, it seems natural to process such a list using a recursive function:

- If the list is empty, handle this case, or
- Handle the data in the head of the list, and then make a recursive call to handle the rest of the list

This approach is advocated by many, e.g. (McCauley, Hanks, Fitzgerald, & Murphy, 2015; Bloch, 2003; Choi, 2021; Pattis, 2006; Elkner, Downey, & Meyers, 2012; Snyder, n.d.). For instance, Turing Award laureate Niklaus Wirth wrote (Wirth, 1976, p. 127):

> "Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms."

But from our personal experience we observe that we often actually process linked lists using loops rather than using recursion. We therefore designed two experiments to check the relative advantages of recursion and iteration. The first experiment deals with *writing* code, and checks whether recursively defined data structures are indeed natural to process recursively — and also identifies other factors that may influence this choice. The second deals with *understanding* given code, which is written using either recursion or iteration. These experiments had 97 and 101 participants respectively, most of them from industry. Both are controlled experiment that compare data structures with different topologies, in particular linked lists and trees.

The code development experiment included questions about basic processing tasks on these data structures. Importantly, these tasks employ different paths for traversing the data structures. What we found was that the recursive definition of a data structure does not have a statistically significant effect on the processing method: for example, linked lists are indeed most often processed using iteration. The more important factor appears to be having a "linear mindset". If it is appropriate to think about the processing as a linear progression, there is an increased tendency to use iteration. This happens when the topology is linear (as in a linked list) or when the processing path is linear (as in a binary search tree). The effect is even stronger if the path is known in advance and does not depend on the input.

In the code comprehension experiment we try to identify different factors that may influence the understanding. This experiment led to more mixed results: in one case the recursive version was more understandable, in another the iterative version was more understandable, and in two there was no major difference. It seems that recursion has an advantage when it simplifies and shortens the code. But iteration again can have an advantage when it presents the processing in a more linear manner. For example, this may happen when we can avoid saving state when making the recursive call and then performing non-trivial operations on it on the exit path.

Our main contributions in this work are

- We focus on how developers in industry think about recursion, as opposed to studies on students in an academic setting;
- We design experiments based on scenarios of processing data structures, instead of classic academic examples that do not necessarily reflect real situations in programming life;
- We show that linear processing is more often done using iteration, especially if it is input-oblivious, even when the underlying data structure is defined recursively;
- We find that recursion can be hard to understand when it includes non-trivial computation after the recursive call.

In terms of practical implications, our results can inform technical debates on software design. For example, rather than accepting the dogma that recursive structures should be processed recursively, developers may benefit from taking the linearity of the processing and the length of the code into account. Likewise, our results suggest additional factors that may inform the debate about learning materials. Educators can use them to support learning by choosing examples that are both natural and simple.

2. Related Works

Recursion has been addressed in several studies on code comprehension and debugging, development, and more (Benander, Benander, & Pu, 1996; Kessler & Anderson, 1986). Benander et al. (Benander et al., 1996) compared iterative and recursive versions of search and copy routines, using student subjects and Pascal. Their experimental participants understood the recursive version of search better than the iterative version, and this result was statistically significant. However, for copy they understood the iterative version better, but this result was not statistically significant. McCauley et al. (McCauley et al., 2015) replicated this study and found that the recursive version of search is not better than the iterative version. Their interpretation of the difference in results was the way the search code is written in Pascal vs. Java.

Several papers discuss cognitive models of understanding recursion. These first require some definitions. Recursion is defined as "a process that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones" (Kahney, 1983). George (George, 2000) then defines *active flow* to mean when control is passed forward to new instantiations, and *passive flow* when control flows back from terminated ones. The main model of how recursion is understood is the "Copies Model" which allows programmers to accurately and consistently represent the mechanics of recursion. Other models may lead to misconceptions because they capture only part of what happens in recursion (Kahney, 1983; Kiesler, 2022; Scholtz & Sanders, 2010). As an example of such misconceptions, several studies have identified base cases as a source of error (Dicheva & Close, 1996; Segal, 1995). They found that participants tend to look at the stop condition as stopping all processing, and ignore the processing after the recursive call (the passive flow).

Researchers have also examined the pedagogical aspect, in order to find methods that promote a correct perception of the recursive concept (Kahney, 1983; Sanders, Galpin, & Götschi, 2006; Scholtz & Sanders, 2010). Aqeel et al. use eye tracking to study how students read iterative vs. recursive code, but do not find any significant differences (Aqeel, Peitek, Apel, Mucke, & Siegmund, 2021). Some researchers claim that recursion is a hard concept to teach (Haberman & Averbuch, 2002; Kahney, 1983). On the other hand, other researchers have wondered if this is really a difficult concept to teach, or perhaps the difficulty is due to the chronological order of teaching recursion versus iteration (Mirolo, 2012). Wiedenbeck pointed out the importance in teaching recursion by providing a wide variety of examples (Wiedenbeck, 1989). She performed an experiment in which she compared two groups of students, a group that was given a variety of examples and another group that was given only a few abstract examples. Students who learned from a variety of concrete examples solved more questions correctly. Other researchers have also dealt with the question of which materials and examples should be used to teach recursion. For example, some have suggested teaching recursion using mathematical formulas (Segal, 1995; Greer, 1989). Another approach talks about introducing recursion in a "real life" connotation. They suggest building a game using recursive code as an example (Lee, V., Beth, & Lin, 2014).

3. Experiment 1: Development

3.1. Research Questions

We are interested in knowing when programmers tend to use a recursive approach and when they prefer the iterative alternative. More specifically, we would like to test whether processing recursively defined structures is necessarily recursive. Our hypothesis is that the reality is much more complex. We would therefore like to locate cases where the tendency is not so unequivocal, and identify which factors naturally lead to implementing a programming task in a recursive way, and alternatively which factors or properties make a programming task to be processed naturally in an iterative way.

The world of programming is wide and diverse, and in this experiment, we try to capture specific characteristics. Our concrete research questions were:

RQ1. Are recursive data structures, like linked lists and trees, usually processed in a recursive manner?

RQ2. Are linear data structures, like linked lists and arrays, usually processed iteratively?

RQ3. Are linear processing trajectories, like in a binary search tree, usually processed iteratively?

RQ4. Is there a difference between linear processing in which the trajectory is known in advance and cases where it is not known in advance?

Table 1 – Summary of programming problems used in the first experiment.

Name	*Data structure*	*Description*	*RQ*
ARRAY	array	check identity of two arrays	control
LIST	linked list	count positive elements	RQ1 RQ2
FACTORIAL	–	compute the number of possible permutations of different elements	RQ1 RQ2
TREE	binary tree	count odd elements	RQ1
SEARCH	binary search tree	check if a value exists	RQ3
MAX	binary search tree	find maximal value	RQ4

RQ5. If the choice between recursion and iteration dichotomous, or are there cases where both approaches are used?

3.2. Methodology

Our methodology is to conduct a controlled experiment to understand which approach (iterative or recursive) programmers take in different situations.

3.2.1. Experimental Materials

The experiment is based on data structures problems, that can be solved either iteratively or recursively. We use different data structures to see how their properties, and the specific programming problem, affect the choice of using iteration or recursion.

There were several methodological considerations behind the choice of problems. First, we wanted to avoid canonical problems, i.e., those that are typically taught in programming courses in academia or in other training courses. At the same time, we wanted the problems not to be too difficult. This had two reasons. First, we wanted participants to be able to answer the questions directly based on their knowledge, without resorting to Google. Second, we wanted the experiment to be reasonably easy, to keep it short and make it easier for us to recruit participants.

The programming problems we used in the experiment are outlined in Table 1. The considerations for choosing these problems are as follows.

1. ARRAY: Check the identity of two arrays. It is included as an example of a problem that is inherently iterative — the data structures are linear with no recursive structure, and it is natural to process them linearly using iteration.

2. LIST: Count the positive elements in a linked list. Like the previous problem this concerns a linear scan of a linear structure (the list), but in this case the data structure is *defined* recursively — each node in the list contains a reference to the remainder of the list.

3. FACTORIAL: Compute the number of possible permutations of an array with different elements. This is similar to the previous problem in that it deals with a linear expression that is naturally defined in a recursive manner. However, in this case it concerns a mathematical structure rather than a data structure.

4. TREE: Count the number of odd elements in a binary tree. This concerns a non-linear data structure which is defined recursively.

5. SEARCH: Check if a certain value exists in a binary search tree. This makes an interesting addition due to the fact that it combines a non-linear recursively defined structure with an algorithm that maps out a linear trajectory within this structure.

6. MAX: Find the maximal value in binary search tree. This is similar to the previous problem, but in this case we know in advanced the linear path in the tree — you always choose the right son.

All the problems together enable a controlled experiment, where the main factor that changes between treatments is the data structure. In ARRAY it is an array, which is a non-recursive linear structure. In LIST and FACTORIAL it is a recursive linear structure: either a linked list or a recursive linear computation. In TREE, SEARCH, and MAX it is a tree, which is a non-linear recursive structure. In TREE this is a general binary tree, and in SEARCH and MAX it is a search tree.

Together the problems are diverse and cover three dimensions. The first dimension is the recursivity of the structure. An array is not a recursive structure. But linked lists and trees are conventionally defined recursively, and have independent substructures which can be viewed as having the same features. The second dimension is the topology of the structure. Linked lists and arrays are linear structures, while trees have a non-linear structure. The third dimension is the predetermination of the traversal of the data structure. In linear structures there is only one option. And when searching for the maximum element in a tree we also know the direction of the route in advance. On the other hand, for finding a general element, different trees will require different trajectories, and this is only known at runtime.

Note that each problem can be solved either iteratively or recursively. All the problems are fundamentally similar and require a basic traversal of the structure. This reduces the number of confounding factors that may affect the results. At the same time, the differences between the problems mean that the same code or algorithm cannot be used for all of them.

The FACTORIAL problem is different from the others due to the fact that it does not deal with a data structure. We formulate the question in data structure terminology to be similar to other questions, and to obscure the fact that we focus on recursion.

The experimental task was to write code that solves these problems. Participants were not constrained which programming language to use, and indeed they used a diversity of languages (Python, C/C++, Java). As the code was typed into a Qualtrics questionnaire, there was no issue of compiling and testing.

3.2.2. Experiment Execution

The experiment was conducted online, using the Qualtrics platform. This platform supported all the features we needed: question selection randomization and questions order randomization. The description of the experiment was "an experiment on data structures". The purpose of this minimalistic description was to hide the true purpose of the experiment from the subjects. If we had called it "an experiment on recursion vs iteration" it would cause subjects to ponder this issue rather than doing what comes to them naturally.

At first, in order to estimate times, we conducted a small pilot study, and concluded that each subject should receive 3 problems. Each subject's problems were selected at random from the 6 problems listed above. Due to this randomization, the numbers of participants who received each problem were not identical. The order of the problems was also randomized, to neutralize the possibility that a problem on a particular topic that appeared previously will systematically affect the answer to a subsequent problem.

As far as the recruitment of participants is concerned, the general ambition was to locate a wide variety of subjects with different backgrounds. In many experiments on understanding and using code there is a tendency to use students, due to their accessibility and availability. Our ambition was to try to reach as wide an audience as possible, including both academics and professional developers. Specifically, we tried to avoid giving the questionnaire to first-year students as they would just do what they recently learned in class.

Participants were recruited in multiple ways. Some were recruited at the university, by approaching fellow students to advanced degrees and undergraduates in the programming labs. We also sent invitations to WhatsApp and Telegram groups of students. Others were recruited based on personal connections, including many that now work in industry. Participants were not compensated for their participation.

We received IRB approval to conduct an experiment involving humans. The experiment was preceded with an explanation and obtaining consent to participate. At the beginning of the questionnaire we gave

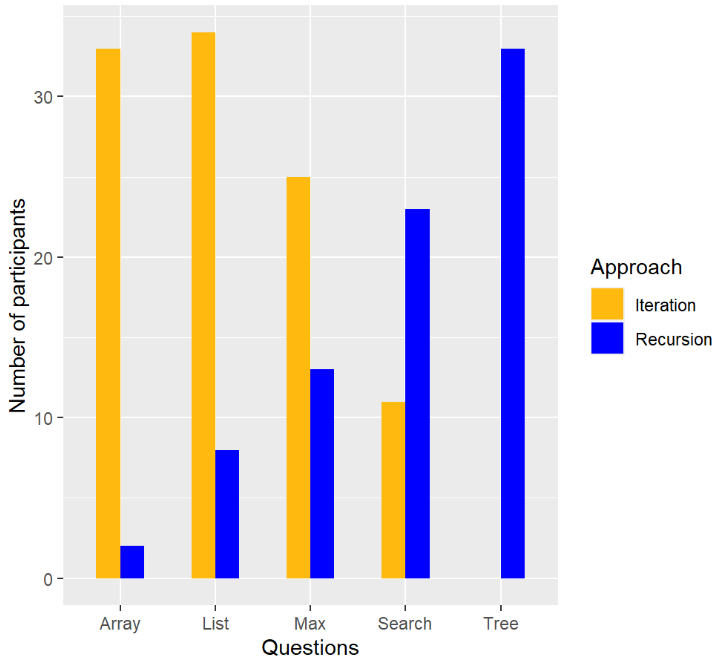


Figure 1 – Use of iteration and recursion in the different problems.

an introduction briefly recalling what the data structures are. In this introduction we tried to avoid an iterative or recursive description of the structure, but to give a neutral description as much as possible. At the end of the questionnaire we included demographic questions in which we collected data on years of experience, gender, etc.

3.3. Experimental Results

3.3.1. Participants and demographic background

The experiment was conducted from January to June 2022. All told, we had 97 participants who answered at least one problem. 85% of the respondents reported their gender: 82% of them were men and 18% women. In terms of education, 82% of the respondents reported their education: most of the participants had an academic background, but not all of them. 42 had a BSc degree or were studying for one, and 28 had an MSc or were MSc students. 7 were PhD students. We also asked about professional programming experience (excluding studies). 79% of them reported it. The median was 3 years. 28 participants had experience between 0–2 years and 49 had more than 3 years of experience.

3.3.2. Approaches for Traversing Data Structures

Basic Results

In total our participants provided 214 answers to the different problems. These answers were classified as using either iteration or recursion. Iteration was identified based on using `for`, `foreach`, or `while` loops. Recursion was identified based on calling the function from within the function itself.

Figure 1 shows the number of participants who elected to use iteration or recursion when solving the different problems. It is immediately obvious that the division between using iteration and recursion is not dichotomous: in most of the problems there was a mix of participants who used both approaches. This answers research question RQ5. However, one approach was typically preferred over the other.

The problems are listed in the graph in order of reduced use of iteration and increased use of recursion. For ARRAY the natural tendency by a wide margin is an iterative implementation: 33 participants used this approach, and only 2 used recursion. In LIST the common approach is still iterative processing, but the difference is somewhat less extreme, with 34 using iteration and 8 using recursion. At the other extreme, we see that in TREE the tendency for a recursive realization is universal: 33 participants used recursion and none used iteration.

The most interesting results occur in the remaining two problems, MAX and SEARCH. Both these problems involve searching in a binary search tree. And in both we observed relatively wide use of the two possible programming approaches. But the proportions were different. In MAX there is a somewhat greater tendency for iterative processing rather than recursive processing, with 25 participants choosing to use iteration and 13 using recursion. In SEARCH, on the other hand, the tendency is reversed: most of the participants, 23 of them, preferred a recursive implementation, while only 11 used iteration.

Statistical Analysis and Interpretation

Our experiments check the percentage of developers who use recursion in different programming situations. We would like to check whether the expected values of these percentages in different situations are unequal. For this we will use a between-subjects Z-test for proportions, where the null hypothesis is that they are equal, and the alternative hypothesis is that the expected values are different.

Research question RQ1 is whether a structure that is defined recursively is overwhelmingly natural to process recursively. In the experiment we checked this using a recursively-defined linked list (problem LIST). We then applied a statistical test to see whether the fraction of participants using recursion was above 50%. The results in Figure 1 indicate that this is not the case (only 8 of 42, which is 19.04%, used recursion), with an extremely high significance ($p < 0.0001$). This provides strong evidence against the claim that if something is defined recursively then it is natural to process it recursively.

To examine the degree of influence of the element of recursive definition on the way of processing we compared a data structure that is defined recursively (a linked list) and a data structure that is not defined that way (an array). We tested if there is a statistically significant difference in the expected values of the fraction of participants who use recursion and iteration in these two cases.

Formally, the independent variable is the way the data structure is defined (whether there is a tendency to define it recursively or not), and the dependent variable is the implementation mode (recursive or iterative). The null hypothesis is that expected value of the percentage of the participants who use recursion is equal in both cases. The alternative hypothesis is that the expected values of these percentages are different. The experimental results were that 5.7% used recursion in the ARRAY problem, and 19.04% in the LIST problem. According to the statistical test this difference is not significant, with a p-value = 0.1637. This means that the null hypothesis was not rejected. Again, we see that a recursive definition does not necessarily lead to recursive processing.

Therefore, the claim that a recursive definition is a significant factor influencing the processing, and specifically that it increases the tendency to process recursively, is not supported by our experiment. This is contrary to the claims made by some educators and researchers that it is natural to process a linked list recursively due to the tendency to define it recursively.

However, other recursive data structures are indeed processed recursively, as we saw for instance in the TREE problem. We suggest in RQ2 that this may be attributed to the topology of the structure: linked lists are inherently linear, while trees are branching. The statistical analysis shows that there is a statistically significant difference between the tendency to recursively process these two data structures. Formally, we define our independent variable to be the linearity of the data structure's topology, and the dependent variable to be the nature of the processing (recursive or iterative). The null hypothesis is that the expected value of the percentage of the participants who use recursion is equal in both cases. The alternative hypothesis is that the expected values of these percentages are different. The experimental results were that only 19.04% used recursion in LIST, and 100% used recursion in TREE. The results of the statistical test is that this is a very significant difference ($p < 0.0001$), and the effect size is 2.23 (large). This means that the null hypothesis was rejected, and we can conclude that a linear topology of the data structure definition is a significant factor.

However, we also have other experimental results on trees in which the use of recursion was much lower: problems SEARCH and MAX. These can be explained by distinguishing between the topology of the structure and the unfolding of the processing. Specifically, we suggest in RQ3 that the linearity alluded to above may be inherent in the processing even if the underlying structure is not linear. In SEARCH the linearity comes about because we trace a single specific path in the tree. In MAX this effect is even stronger, because we know in advance exactly which path we are going to trace: to find the maximal element we always continue to the right son (RQ4).

Formally, we support these suggestions with two statistical tests. In the first test we compare the TREE and SEARCH problems. The topology of the data structure is the same: both are binary trees. So the independent variable is not the topology but the linearity of the processing — in SEARCH we have a single linear path in the tree, while in TREE we backtrack to explore different branches. The dependent variable is as usual the percentage of the participants who use recursion. The null hypothesis is that expected value of the percentage of the participants who use recursion is equal in both cases. The results were that 100% used recursion in TREE, and 67.6% did so in SEARCH. The results of the statistical test is that this is highly significant, with a p-value = 0.001178 and effect size of 1.2 (large). Thus the null hypothesis was rejected.

In the second statistical test the independent variable is again categorical with two levels, whether we know the direction of the path in advance or not. The dependent variable remains the method of processing (recursive or iterative), and the null hypothesis is that the expected value of the percentage of the participants who use recursion is equal in both cases. The alternative hypothesis is that the expected values of these percentages are different. The results were that in SEARCH 67.6% used recursion, and in MAX 34.2% did so. The results of the statistical test is that this is also significant, with a p-value = 0.009411 and effect size of 0.68 (medium). Thus the null hypothesis was rejected.

To summarize, our results indicate that the most important factor affecting the choice of processing style is *the topology of the computation path*. The more linear and obvious the path, the higher the tendency to use iteration. This has important implications regarding how we interpret the results. For example, we observed a strong tendency to process linked lists iteratively. But we suggest that this is not because of the list's linear structure, but rather because of the liner path of the processing. The linear structure of a linked list is just one situation in which such a linear path arises.

Another important factor is being input-oblivious, in the sense that the direction (and perhaps also the length) of the path is predetermined. For example, this is the reason that MAX is more iterative than SEARCH, and nearly as iterative as LIST (and indeed the statistical test showed that the difference between MAX and LIST is not significant, with a p-value = 0.110). This also indicates that emphasizing the underlying structure is the wrong consideration to apply, because the concrete path used in each specific instance is more important.

Supportive Evidence

In parallel to the execution of our experiment we became aware of an exercise in the data structures course in a university unrelated to us, which was very similar. The students in that course were required to merge two sorted linked lists into a single linked list. This is similar to our LIST problem. There were 284 students who submitted the exercise, so the sample was much larger than in our experiment. We requested and received a statistical summary of the approach they used in their solutions. The results were that 16.2% used recursion, very close to the result of 19.5% in LIST (Figure 2). Interestingly, the solution provided by the teaching staff was recursive, and the TA told us she thought this was the natural approach in this case. But obviously the students thought otherwise.

Factorial

In addition to previous problems which were described in Figure 1, our experiment also included the FACTORIAL question. This question is different from the other questions because it does not deal with a data structure. However, there is a similarity to LIST in that it is a linear structure that is defined recursively. We then applied a statistical test to see whether the fraction of participants using recursion was above 50%. 31.25% gave a recursive solution (Figure 3), with a significance p-value = 0.02591.

This result is interesting because factorial — together with Fibonacci — is a canonical problem used to

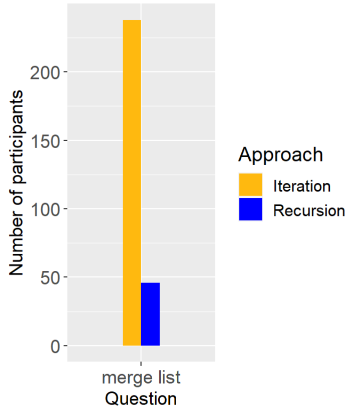


Figure 2 – Use of iteration and recursion in a data structures exercise.

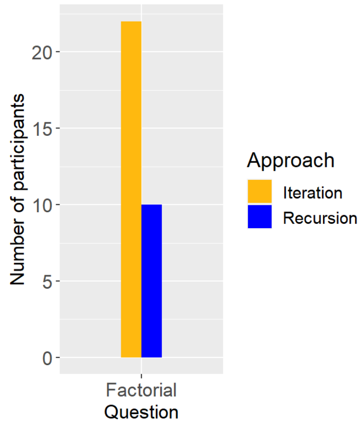


Figure 3 – Use of iteration and recursion in FACTORIAL.

teach recursion. This context may explain the higher use of recursion relative to LIST. But still, the vast majority did not use recursion.

4. Experiment 2: Comprehension

The second experiment complements the first experiment and examines the comprehension of iterative and recursive codes.

4.1. Research Questions

This experiment is limited to linear recursion, in the sense that only one recursive call is made each time. But we do want to distinguish between two types of such recursive calls. The first is tail recursion, in which the recursive call is the last statement that is executed by the function. It can therefore be followed in a similar way to iteration. The other is "classic" recursion, where some processing occurs after the recursive call. This may require a more general mental model, and is expected to be harder to follow. The research questions are:

RQ6. Is there a difference between the understanding of tail recursion and iteration?

RQ7. Is there a difference between the understanding of classic recursion and iteration?

RQ8. Can the code length, for example when recursive code is shorter, affect the relation of understanding recursive vs. iterative code?

4.2. Methodology

Our methodology is again to conduct a controlled experiment, this time to see which approach (iterative or recursive) programmers find easier to understand.

4.2.1. Experimental Materials

In this experiment we need problems that are not obviously iterative like traversing an array, and not obviously recursive like traversing a whole tree. All the problems are based on recursively-defined data structures, and linear processing. We saw in the first experiment that such problems are not necessarily natural to process using recursion, but perhaps it is easier to understand their recursive implementation?

The problems for the comprehension task are different from each other and have been chosen so that the functionality is realistic and not artificial. This is important because participants may waste time trying to understand what exactly the code is used for, when in fact it has no real use. Also, using contrived problems may increase the fraction of inaccurate and incomplete answers, leading to much variation in the content and length of the answers. All this is in contrast to a writing experiment, where made-up

Table 2 – Summary of comprehension problems used in the second experiment.

Name	*Data structure*	*Description*	*RQ*
LSEARCH	linked list	find a value	RQ6
MAX	binary search tree	find maximal value	RQ6
MERGE	linked list	merge two sorted lists	RQ6 RQ8
REVERSE	linked list	reverse a list	RQ7

problems like counting odd elements are useful. Ideally, it is desirable that both versions (recursive and iterative) have the same length, but this is not always possible.

The programming problems we used in this experiment are outlined in Table 2. The considerations for choosing these problems are as follows.

1. LSEARCH: Check if a certain value exists in a linked list. This problem was used by Benander et al. in their groundbreaking work comparing recursion and iteration, where they found that recursion is easier to understand (Benander et al., 1996). We therefore wanted to include a replication of their work in our experiment.

2. MAX: Find the maximal value in a binary search tree. This is the same problem as in the first experiment, but in this case we are interested in understanding the code. It is included as a representative of using a non-linear data structure.

3. MERGE: Given two sorted lists, merge them into one sorted list. There are several reasons for choosing this problem. First, it is an example of common cases where the recursive code is shorter. Also, it is an interesting case since the linear processing is over two lists at the same time.

4. REVERSE: Return a linked list in reverse order. This is interesting because during the processing we make a change to the structure itself. Importantly, the change is done after the recursive call, so this is a classic and not tail recursion.

Note that in the previous experiment the main difference between problems was in the underlying data structures, and the differences in processing were rather minor. Here, on the other hand, we focus on processing the same structure. The first problem, LSEARCH, is basically just a traversal of a linked list, similar in spirit to the problems in experiment 1. The second problem, MAX, represents the alternative platform of a search tree instead of a linked list. The last two problems involve more complex processing, which may lead to more complex code. In MERGE we traverse two lists in parallel, and in REVERSE we modify the list as we traverse it.

4.2.2. Experiment Execution

This experiment was also conducted online using the Qualtrics platform. We described the experiment as an experiment on code comprehension of recursive and iterative code.

Each participant was asked about all four problems. Initially we reminded them about binary search trees and asked them to understand the code of the MAX problem. Then we gave a short introduction on linked lists, and presented them with SEARCH, REVERSE, and MERGE. For each problem, one version — either recursive or iterative — was chosen at random. Participants never got both versions of the same problem.

In recruiting participants we again emphasized recruiting participants from industry. This has two reasons. One is to be more relevant to the "real life" of software development. In addition, we wanted participants who were farther removed from the programming patterns common to academic studies. Participants were not compensated for their participation.

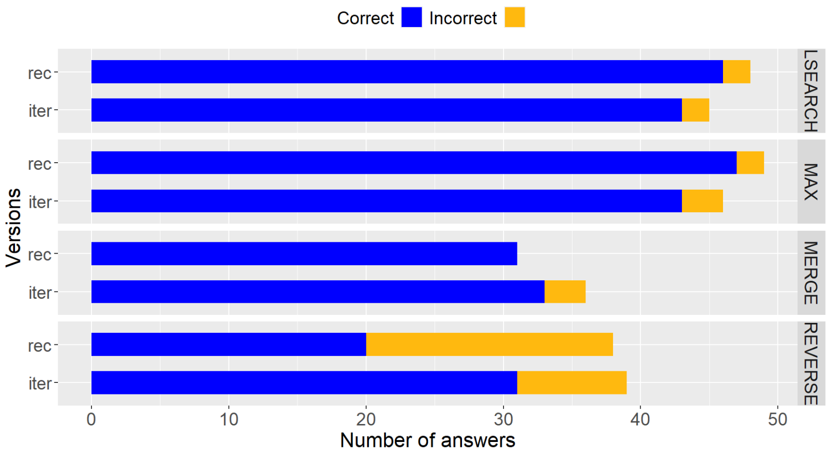


Figure 4 – Correctness results.

4.3. Results

4.3.1. Participants and demographic background

The experiment was conducted from August to November 2022. All told, we had 101 participants who answered at least one problem. 70% of the respondents reported their gender: 75% of them were men and 25% women. In terms of education, 73% of the respondents reported their education: most of the participants had an academic background, but not all of them. 40 had a BSc degree or were studying for one, and 26 had an MSc or were MSc students. 8 were PhD students. We also asked about professional programming experience (excluding studies). 71% of them reported it. 22 participants had experience between 0–2 years, 50 had more than 3 years of experience, and the median is 3.5.

4.3.2. Correctness Results

We measured two response variables in the experiment: the correctness of the answer and the time it took for the participants.

In judging correctness we want to emphasize the essence of the code. In other words we want to avoid penalizing participants for superficial mistakes, provided they did understand the basic functionality.

For example, the correct answer to the MAX problem is that it finds the maximal element in the search tree, which is also the rightmost element. A possible mistake is saying that this is the minimum element. But a likely interpretation for this mistake is that the programmer actually understood the code itself, and was just confused about which son is larger and which smaller — information that does not appear directly in the code. We therefore accepted his answer even though it is technically incorrect.

On the other hand, in the REVERSE question, a common answer was that the function returns the last element of the list. This answer is factually true: the function returns a pointer to the last element of the input list. But this answer misses the essential part of the code which reverses the list, and makes the last element first. Therefore we did not accept this answer even though technically it describes a correct fact.

Figure 4 presents the number of correct answers out of all the answers for each version of each question. We conducted a statistical test in each of the questions see whether there is statistically significant difference between the versions. The independent variable is the nature of the processing (recursion or iteration) and the dependent variable is the fraction of correct answers. The dependent variable is binomially distributed, so we used the Z-test.

It is easy to see that in the first 3 questions there is little difference between the versions, and the statistical test confirms that this is not statistically significant. So from the perspective of correctness, the answers to RQ6 and RQ8 is that there is no difference between iteration and recursion. The only problem where there is a statistical difference is REVERSE: in the iterative version of REVERSE, 31 of 39 gave a correct

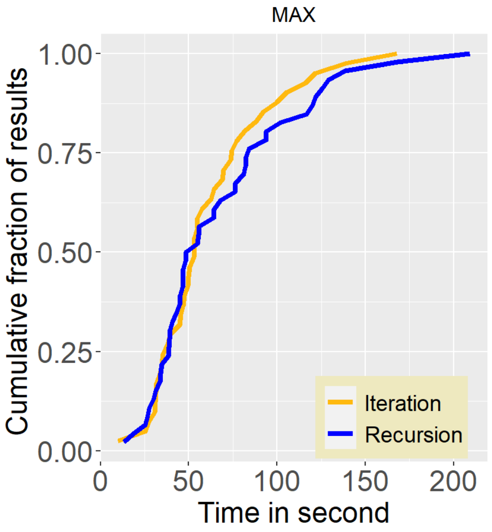


Figure 5 – Cumulative Distribution Functions (CDFs) of time to correct solution for MAX*.*

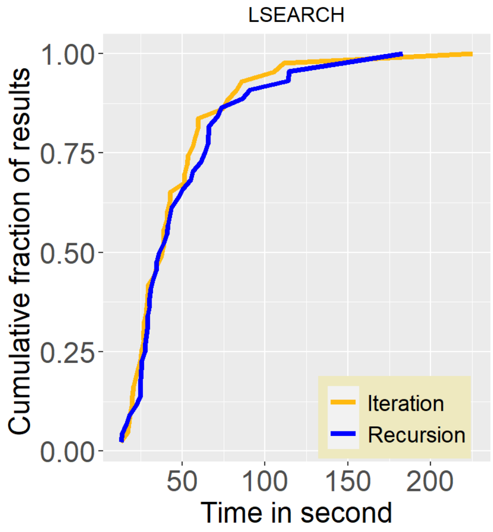


Figure 6 – CDFs of time to correct solution for LSEARCH*.*

answer. But in the recursive version only 20 gave a correct answer out of 38. The statistical test gave p-value = 0.0244 and effect-size 0.578 (Medium). This shows that for RQ7 there is a difference. We will discuss this after we analyze the time differences.

4.3.3. Time Results

Our experiments check the time for developers to understand each of the codes. We would like to check whether the expected times in different versions are equal. For this we will use a t-test between subjects, where the null hypothesis is that they are equal, and the alternative hypothesis is that the expected values are different. Only correct answers were considered in this analysis.

The difference between the recursive version and the iterative version in the MAX question is not significant, since the p-value = 0.4808, and Fig. 5 shows that indeed the distributions are very similar to each other. This finding, together with the statistical result for the percentage of correct answers, shows that there is no essential difference in terms of code comprehension as far as this question is concerned.

Similarly, for the question LSEARCH we got that p-value = 0.6802, that is, the differences are not statistically significant, and Fig. 6 shows that again the distributions are similar. The combination of statistical tests again indicates that there are no differences in the difficulty of understanding the two versions. This is contrary to the results of Benander et al., who received a statistically significant result in the LSEARCH problem (Benander et al., 1996), but consistent with the replication results of (McCauley et al., 2015). In our view, a possible explanation for the difference between the results is the difference in program lengths. In Benander et al.'s experiment the recursive version came out more readable, but this could be because its code was shorter. Another possible reason for the difference may be that we appealed to experienced programmers, while Benander et al. used students from a first data structures course. This is meaningful because recursion is a major topic in such a course, but is not very common in everyday work. So the students may have been more "into" recursion than the professional programmers.

Taken together it can be seen that for simple linear processing, where the code segments are approximately the same length, using iteration or recursion does not dramatically affect the code comprehension. This answers research question RQ6.

In MERGE we can see in Figure 7 that the distributions are separated, and understanding the iterative version takes a longer time (the CDF is shifted to the right). This is statistically significant with p-value = 0.02434 and effect-size (Cohen's d) of 0.581 (medium). While this problem is a classic linear

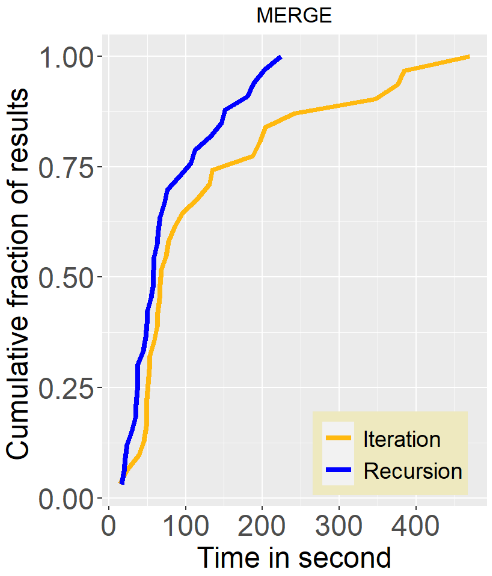


Figure 7 – Cumulative Distribution Functions (CDFs) of time to correct solution for MERGE.

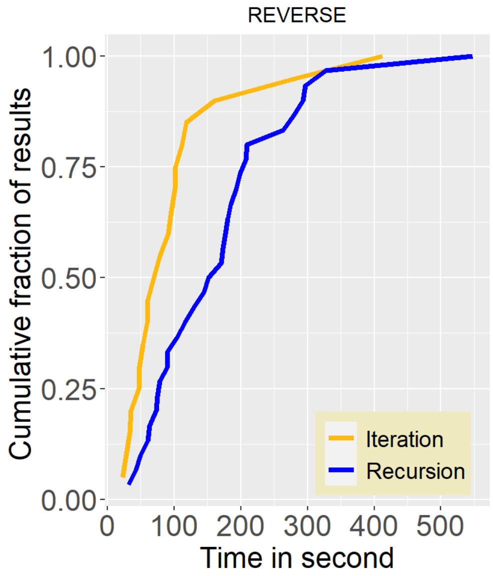


Figure 8 – CDFs of time to correct solution for REVERSE.

processing of lists, like the previous two question, in this one the recursive code is significantly shorter than the iterative alternative. This is similar to the above-quoted result by Benander et al., that the recursive code is more readable since it is shorter, and answers RQ8.

Finally, we turn to the last problem REVERSE. We can see in Figure 8 that the distributions are separated, but in contrast to the previous result, in this case understanding the recursive version takes a much longer time (the CDF is shifted to the right). The results are very significant with p-value = 0.002819 and the effect size (Cohen's d) is 0.95 (large). This answers RQ7.

In order to analyze the results of REVERSE we begin with the code. The recursive version is as follows:

```
1 def func(node):
2     if (node == None):
3         return node
4     if (node.next == None):
5         return node
6     node1 = func(node.next)
7     node.next.next = node
8     node.next = None
9     return node1
```

Our interpretation is that REVERSE is an example of a recursive function with non-trivial commands that have to be executed after the recursive call (on the "passive flow", lines 7-8). But when you read such a function and reach the recursive call, you do not know this yet. Therefore you cannot really know what the recursive call does, which makes the recursion harder to trace. This analysis is consistent with (Scholtz & Sanders, 2010; Sanders et al., 2006; Götschi, Sanders, & Galpin, 2003) and others, who claim that recursion of the above type requires a more accurate "mental model". The alternative iterative version is more transparent, due to the fact that we can see the mechanism change each element one after the other. This is likely to be easier to follow.

This analysis is bolstered by the nature of the errors that participants made when trying to understand the function. The most common error (7 from 18) was to state that it returns the last element in the list. We speculate that they reached this conclusion by tracing the code in the following way. First, they noted that the function returns `node1` (in line 9). Then they saw that `node1` is assigned its value from the recursive call (in line 6). As there are no other assignments to `node1`, this implies that its value percolates back through all the recursive calls. So they looked for the termination condition, and found that the last element (the one with `next == None`) is returned (lines 4-5). But in this tracing they skipped the two hard-to-understand lines that change the structure and cause the list to be reversed after the exit from the recursion (lines 7-8). This analysis is consistent with previous studies that found that a common misconception is that participants tend to think that a recursive method stops execution when the base case is reached (Dicheva & Close, 1996; Segal, 1995).

Additional work is needed to verify all this. We are planning followup experiments using other recursive versions, and specifically using tail recursion.

5. Threats to Validity

Construct validity Our experiment measures understanding according to the criteria of measuring times and checking the correctness of the answer. These are the commonly used metrics (Rajlich & Cowan, 1997). But time and correctness are only proxies for understanding. Time can be affected by many external factors that are not relevant to the understanding itself. Also, it can be that a correct description does not derive from a deep and thorough understanding of the code, but from an informed guess, prior knowledge, and more.

Internal validity Our conclusions may not follow from the experiments for several reasons. In comprehension experiments it is possible that the observed differences are not a result of the intended difference in the independent variable, namely iteration vs recursion, but maybe something else. For example, the names of variables in the iterative and recursive codes of the REVERSE problem are different. Due to the nature of the iterative code, there are more variables in it, and their names are slightly more informative, such as `current` and `prev`. Of course, these are still names that do not provide information about the functionality of the code, but they help a little in tracing it. However, this is probably not really a problem in this case because the differences were very significant in both time and correctness, so it is unlikely that this difference affects the results themselves.

External validity Our experiment included only a few pieces of code, and although we tried to reach a diverse audience in terms of programming experience, gender, and education, it is still likely that it is not fully representative. The study should be replicated to ensure its validity and map out its generalizability.

6. Conclusions

Our research consisted of two stages, concerning code development (experiment 1) and code comprehension (experiment 2). In the first experiment we identified several factors affecting the use of iteration vs. recursion. The first is the *topology* of the data structure, and specifically whether it is linear. The second is the trajectory of the *processing*, and again whether it is linear. The third is whether this is *oblivious* or input-dependent. In retrospect, it appears that all three factors reflect the degree to which the problem is amenable to a straightforward serial solution, or in other words, to a simple iterative solution. This is reflected in the results, where the more linear and oblivious problems were more often solved using iteration.

In the second experiment we observed an apparent contradiction between MERGE and REVERSE: in MERGE the recursive version was easier to understand, while in REVERSE it was dramatically harder. We believe that this difference indicates that a more nuanced classification is needed: the distinction should not be only between iteration and recursion, but also between different types of recursion. Specifically, MERGE (and also LSEARCH and MAX) are solved using a tail recursion which is similar to iteration: the processing is essentially done one element at a time, from beginning to end. But REVERSE, in contradistinction, requires a more classic form of recursion, in which part of the processing occurs on the exit path, after the recursive call. So each element is visited twice, in the call and in the return, and in the return it is not just passed back passively, but manipulated in a non-trivial manner. This is what makes the recursive version more complex. And in such a case, if there is a simple iterative alternative where the transition is more linear and serial, then it is likely that the iterative code will be more understandable.

Taken together, these two experiments suggest possible cognitive underpinnings of the choice between recursion and iteration. Algorithmic thinking is inherently serial—we define algorithms as a sequence of steps, and find it hard to think about things that happen in parallel. Likewise, it is easier to think about linear processing, where we handle elements one after the other, and do not need to return to those we have already visited (less linear). Such thinking is analogous to using iteration. Recursion is useful when such thinking is insufficient: when we need to save state for use in additional processing later (as in REVERSE), or when the basic structure is not linear (as in TREE).

The above observations have several implications for programming practice. One is to beware of dogmas. For example, it is often thought that using a recursive definition implies a preference for recursive processing. We believe that a recursive definition may be a necessary condition for using recursive processing, but it is not a sufficient condition. Other considerations also need to be applied.

7. Experimental Materials

The experimental materials are available in:
https://doi.org/10.5281/zenodo.8266266

8. Acknowledgements

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18).

9. References

Aqeel, A., Peitek, N., Apel, S., Mucke, J., & Siegmund, J. (2021). Understanding comprehension of iterative and recursive programs with remote eye tracking. In *32nd Workshop Psychology of Programming Interest Group.* Retrieved from `https://ppig.org/papers/2021-ppig-32nd-aqeel/`

Benander, A. C., Benander, B. A., & Pu, H. (1996). Recursion vs. iteration: An empirical study of comprehension. *J. Syst. Softw.*, *32*(1), 73–82. doi: 10.1016/0164-1212(95)00043-7

Bloch, S. (2003, May). Teaching linked lists and recursion without conditionals or null. *J. Comput. Sci. Coll.*, *18*(5), 96-108.

Choi, S. G. (2021). *United States Naval Academy, introduction to computer science.* `https://www.usna.edu/Users/cs/choi/si204/lec/l36/lec.html`. (Accessed: 2022-01-16)

Dicheva, D., & Close, J. (1996). Mental models of recursion. *J. Educational Computing Research*, *14*(1), 1-23. doi: 10.2190/AGG9-A5UD-DEK0-80EN

Elkner, J., Downey, A. B., & Meyers, C. (2012). *How to think like a computer scientist: Learning with python.* `https://www.cs.swarthmore.edu/courses/cs21book/build/ch18.html`. (Accessed: 2022-01-16)

George, C. E. (2000). EROSI - visualising recursion and discovering new errors. In L. B. Cassel, N. B. Dale, H. M. Walker, & S. M. Haller (Eds.), *Proc. 31st SIGCSE Technical Symp. Computer Science Education* (pp. 305–309). Retrieved from `https://doi.org/10.1145/330908.331875` doi: 10.1145/330908.331875

Götschi, T., Sanders, I. D., & Galpin, V. (2003). Mental models of recursion. In S. Grissom, D. Knox, D. T. Joyce, & W. P. Dann (Eds.), *Proc. 34th SIGCSE technical symp. Computer science education* (pp. 346–350). doi: 10.1145/611892.612004

Greer, J. E. (1989). A comparison of instructional treatments for introducing recursion. *Computer Science Education*, *1*(2), 111–128. doi: 10.1080/0899340890010204

Haberman, B., & Averbuch, H. (2002). The case of base cases: Why are they so difficult to recognize? *Proc. 7th conf. Innovation & Tech. in Comput. Sci. Education*, 84-88. doi: 10.1145/637610.544441

Kahney, H. (1983). What do novice programmers know about recursion. *Proc. SIGCHI Conf. Human Factors in Computing Systems*, 235–239. doi: 10.1145/800045.801618

Kessler, C. M., & Anderson, J. R. (1986). Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, *2*(2), 135–166. doi: 10.1207/s15327051hci0202_2

Kiesler, N. (2022). Mental models of recursion: A secondary analysis of novice learners' steps and errors in Java exercises. In S. Holland, M. Petre, L. Church, & M. Marasoiu (Eds.), *33rd Workshop Psychology of Programming Interest Group* (pp. 226–240). Retrieved from https://ppig.org/papers/2022-ppig-33rd-kiesler/

Lee, E., V., S., Beth, B., & Lin, C. (2014, July). A structured approach to teaching recursion using cargo-bot. In *10th Conf. International Computing Education Research* (p. 59–66). doi: 10.1145/2632320.2632356

McCauley, R. A., Hanks, B., Fitzgerald, S., & Murphy, L. (2015). Recursion vs. iteration: An empirical study of comprehension revisited. In *46th ACM Tech. Symp. Comput. Sci. Education* (pp. 350–355). Retrieved from https://doi.org/10.1145/2676723.2677227

Mirolo, C. (2012, Sept). Is iteration really easier to learn than recursion for CS1 students? In *Proc. 9th Intl. Conf. Comput. Education Research* (p. 99-104). doi: 10.1145/2361276.2361296

Pattis, R. E. (2006). *Recursion on linked lists, advanced programming/practicum.* https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/llrecursion/index.html. (Accessed: 2022-01-23)

Rajlich, V., & Cowan, G. S. (1997, Mar). Towards standard for experiments in program comprehension. In *5th International Workshop on Program Comprehension* (pp. 160–161). doi: 10.1109/WPC.1997.601284

Sanders, I. D., Galpin, V., & Götschi, T. (2006). Mental models of recursion revisited. In R. Davoli, M. Goldweber, & P. Salomoni (Eds.), *Innovation & Tech. in Comput. Sci. Education* (pp. 138–142). doi: 10.1145/1140124.1140162

Scholtz, T. L., & Sanders, I. D. (2010). Mental models of recursion: investigating students' understanding of recursion. In R. Ayfer, J. Impagliazzo, & C. Laxer (Eds.), *Innovation & Tech. in Comput. Sci. Education* (pp. 103–107). doi: 10.1145/1822090.1822120

Segal, J. (1995). Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science*, *22*, 385–411.

Snyder, W. (n.d.). *Recursion and linked lists.* https://www.cs.bu.edu/fac/snyder/cs112/CourseMaterials/LinkedListNotes.Recursion.html.

Wiedenbeck, S. (1989, January). Learning iteration and recursion from examples. *International Journal of Man-Machine Studies*, *30*(1), 1–22. doi: 10.1016/S0020-7373(89)80018-5

Wirth, N. (1976). *Algorithms + data structures = programs*. Englewood Cliffs, N.J.: Prentice-Hall.

Exploring cognitive waste and cognitive load in software development - a grounded theory

Daniel Helgesson
Dept. of Computer Science
Lund University
daniel.helgesson@cs.lth.se

Abstract
This paper provides theoretical account of a critical/grounded theory study in industrial software engineering/development settings. We extend our research into the 'critical ethnography' domain, drawing on Rosen, by revisiting previously collected data using social theory as a theoretical filter. We explore the relations and underlying tensions between alienation and *absence of* user centered design in regards to software development tools design using a critical lens. We further experiment with representation of qualitative data and analysis using commentary excerpt units. We present a grounded theory account of the consequences of absent user centred design activities in regards to digital tools in industrial software engineering. We conclude that not deploying *user centered design* when developing software development tools is really (stressing *really*) bad waste management policy for a multitude of reasons.

Research paradigm: design science

Epistemological position: pragmatist

Methodology: case study

Method: grounded theory/(critical) ethnography

Analysis: abductive/iterative

Data collection: interviews, semistructured

Theoretical underpinnings: distributed cognition, cognitive load theory, cognitive load drivers, perspectives, social theory, marxism

1. Intro
As we have previously stated (Helgesson, 2021) the meaning of 'software engineering' is dual. One meaning is literal, i.e. 'the process of engineering software', the other is a scientific engineering discipline aimed providing software development practitioners with scientific evaluations, suggestions and knowledge in regards to software development processes and tools. The actual phrase was coined at the first NATO conference on the matter in the late sixties (Naur & Randell, 1969) . Despite the fact that "personell factors" (Naur & Randell, 1969) was brought up as important factor in software engineering activities at the seminal conference, and that software engineering as such was identified as "socioculturally constituted phenomenon" in the mid nineties (Bertelsen, 1997) it has as a scientific discipline and academic field been dominated by positivism and quantitative methodologies largely focused on non-human factors (Lenberg, Feldt, & Wallgren, 2015) .

Further 'odd' and/or 'novel' research in terms of methodology within the community are often met with a 'so what' response from reviewers (Sharp, Dittrich, & de Souza, 2016) rather than an 'now thats interesting' (Davis, 1971). That being said being said, considerable efforts within the field have been made to draw on qualitative methodology from social sciences, e.g. (Runeson, Höst, Rainer, & Regnell, 2012), (Stol, Ralph, & Fitzgerald, 2016), (Sharp et al., 2016).

The core phenomena we study are *cause*, and subsequent *consequence*, of cognitive load (i.e. 'roughly mental effort' (Helgesson, 2021)) as a consequence of digital work environment and corresponding tools in the software industry. We have previously used grounded theory (Charmaz, 2014) (Bryant, 2017) approach in three studies resulting in a taxonomy of 'cognitive load drivers' (Helgesson, Engström, Runeson, & Bjarnason, 2019), an ethnographic study of cognitive load in a distributed cognitive setting (Helgesson, Appelquist, & Runeson, 2021), and a synthesis of previous research and extant literature (Helgesson & Runeson, 2021).

Purpose of paper – this paper, an industrial case study, is positioned as a halfway marker in (4th research cycle/paper) in an extended (five year) grounded theory project on cognitive load (Helgesson & Runeson, 2021), as a consequence of digital work environment in software industry. In this specific paper we explore 'critical traditions' especially 'critical ethnography'(Prasad, 2018) in software engineering by revisiting and re-analysing our aggregated dataset in regards to the sensitising concepts (Charmaz, 2014) of 'cognitive sustainability' and 'cognitive productivity' we noted in previous work (Helgesson & Runeson, 2021). Further we deploy concepts from 'social theory' (Luastsen, Larsen, Nielsen, Ravn, & Sörensen, 2017) (i.e. primarily 'alienation') as analytical filter/lens. Additionally, the analysis serves as a vehicle for open-ended exploration on how one can 'write' qualitative research in the software engineering research community using 'commentary excerpt units' (Rennstam & Wästefors, 2018).

2. Background

2.1. Cognitive load as phenomenon

Cognitive load (i.e. 'roughly mental effort' (Helgesson, 2021)) is per definition inherent in all forms of cognitive work. The limits of the human mind in terms of information processing and corresponding bandwidth has been well known for more than fifty years (Miller, 1956). Most, if not all, activities in software development are inherently cognitively loaded (Sedano, Ralph, & Péraire, 2017). We use the term 'cognitive load driver'(Helgesson et al., 2019) to describe the causal nature of cognitive load in regards to digital tools and work environment in the software industry. In the context of this paper we look at cognitive load from a more general cognitive work environment perspective as described by Gulliksen, Lantz, Walldius, Sandblad, and Åborg (2015). Kirsh (2000) describes cognitive overload[1] in work settings.

2.2. Cognitive load in software engineering

Cognitive load in software engineering has, to date, largely been investigated using quantitative methodology (Gonçales, Farias, & da Silva, 2021) (Helgesson, 2021). We let (Müller & Fritz, 2016) and (Fritz & Müller, 2016) serve as contemporary examples. In addition to our qualitative studies (Helgesson et al., 2019), (Helgesson et al., 2021), (Helgesson & Runeson, 2021), Sedano et al. (2017) used grounded theory to investigate different forms of 'waste' in software development, finding 'extraneous cognitive load' being one aspect.

2.3. Marxist/critical traditions and social theory in software engineering

In software engineering marxist traditions (Prasad, 2018) appear to have had very small impact as of yet (Melegati & Wang, 2021) . The authors discuss 'Critical theory' as a 'research paradigm' rather than 'a tradition'. The study mentions Hilderbrand et al. (2020) on 'gender'. We have further found (Vorvoreanu et al., 2019) and (Burnett, Peters, Hill, & Elarief, 2016) on the same issue.

Finally, marxist approaches to explore software related phenomena can be found in other fields of research - e.g. (Fuchs & Sevignani, 2013), (Pfeiffer, 2014), (Nygren & Gidlund, 2016) and (Krüger & Johanssen, 2014). See also: cyberspace Froomkin (2002) on 'cyberspace' and Söderberg (2011) on 'hacking', and (D'Ignazio & Klein, 2020) on 'data science' .

Use of social theory in relation to software engineering has been studied in (Ralph, Chiasson, & Kelley, 2016) and (Lorey, Ralph, & Felderer, 2022) respectively.

[1]See also Simon (1971): "In an information-rich world, the wealth of information means a dearth of something else: a scarcity of whatever it is that information consumes. What information consumes is rather obvious: it consumes the attention of its recipients. Hence a wealth of information creates a poverty of attention and a need to allocate that attention efficiently among the overabundance of information sources that might consume it."

3. Method

3.1. Grounded theory and epistemological position

In this paper, an industrial case study, we use grounded theory[2] largely as prescribed by Charmaz (2014) and Bryant (2017)[3][4].

We deploy an ethnographic approach (Charmaz & Mitchell, 2001) using distributed cognition (Hollan, Hutchins, & Kirsh, 2000) and concepts from 'social theory' (Luastsen et al., 2017) (primarily 'alienation') as theoretical observational filters. We venture further into the 'critical' ethnography tradition (Prasad, 2018) using extant literature [5].

The work reported in this paper is conducted within the research paradigm of 'design science' (Runeson, Engström, & Storey, 2020). We see grounded theory as a general purpose qualitative method for generating 'theory' (Abend, 2008) in order to describe a problem in practice, and we take a pragmatist (Rorty, 1979) epistemological grounded theory position (Bryant, 2017) [6].

3.2. Research goal and research questions

Given the exploratory nature of grounded theory it is common for studies informed by grounded theory to start of with "an open ended research goal" (Stol et al., 2016) or "initial research questions that evolve throughout the study" (Charmaz, 2014).

Our initial **research goal** was:

– To explore previously recorded data in regards to cognitive load in software engineering using 'grounded theory ethnography' (Charmaz & Mitchell, 2001) from a 'critical ethnography' (Prasad, 2018) angle using the sensitising concepts of 'cognitive sustainability', 'cognitive productivity' and 'cognitive sustainability' we noted in previous work in conjunction with 'social theory' (Luastsen et al.,

[2]Grounded theory is an exploratory (Stol et al., 2016) qualitative method, by some heralded as one of the (if not 'the') most important qualitative methods to appear in the scientific toolbox to date (Bryant, 2017). The aim of grounded theory is to provide a framework for generating 'theory' (Abend, 2008) from (largely) qualitative data (Charmaz, 2014). Core elements in most grounded theory method sections written by phd students is a lengthy (and somewhat pretentious) segment on the history of grounded theory (Bryant, 2017). Having written three of those ((Helgesson et al., 2021), (Helgesson & Runeson, 2021), (Helgesson, 2021)) we will not waste further space on this matter in this context. We see grounded theory as a general purpose iterative qualitative method/vehicle for generating theory from largely qualitative data abductively (or inductively). The guidelines provided by Charmaz (2014) helps us keep control of data and ensures methodological rigour (Gioia, Corley, & Hamilton, 2013) and transparency. No more. No less.

[3]In order to be very precise here: Bryant (2017) does not consider his methodological approach as a singular version of grounded theory in contradiction to that of Charmaz (2014), but rather as a complement. The main difference lies in epistemological position. In order to rule out the relativisation that a constructivist position allows for, and may result in, "'ultimate caricature of postmodernism", Bryant advocates for a 'pragmatist' (Rorty) epistemological position. So from a methodological perspective we rely on Charmaz, while framing our epistemological within the epistemological framework of Bryant since we want to close the door on relativisation.

[4]It should also be noted that Bryant (2017) highlights that the tension does not lie in a (false) dichotomy between 'qualitative' and 'quantitative' methodologies (and analysis and data). This is not really that hard to comprehend. How would it be possible to make sense of quantitative findings without qualitative analysis? Instead Bryant highlights that the main epistemological tension lies between 'objectivist' and 'constructivist' stand points. They further offers a solution to this tension by taking the middle ground of 'pragmatism', thereby taking a position immune to constructivist epistemological critique of objectivism as the core tenet in 'pragmatism' is that it, itself, is inherently 'fallible' and 'contingent'. With a background in 'sciences of the natural'(Simon, 1969) constructivism (and corresponding risks of relativisation) is is not an easy epistemological position to uphold and We thus lean toward pragmatism.

[5]Reading Glaser from a distance (i.e. not by using close reading, but reading explicitly): "...reading and use of literature is not forsaken in the beginning of a grounded theory project. It is vital to be reading and studying from the outset of the research, *but in unrelated fields*" (Glaser, 1992, p. 35). This is a clear indication that the use of literature and abduction has been relevant in traditional grounded theory for almost 30 years. From our perspective the role of abduction has been rhetorically underplayed and the matter of literature absence has equally been rhetorically overplayed in grounded theory discourse. See also (Martin, 2019) and Dey (1999), (1993).

[6]Given the many variants and interpretations on grounded theory (Bryant, 2019) guidelines on grounded theory often highlight the need to be very specific on describing what version of grounded theory and what epistemological position is being used in order to avoid 'method slurring' and avoiding criticism for deploying a "rhetorical sleight of hand", e.g. (Stol et al., 2016). We will, however, not venture further into this discussion at this point.

2017) in order to see what a ’critical’ ’perspective’ (Helgesson & Runeson, 2021) of cognitive load in software engineering might consist of.

This evolved into the following **research questions**:

1 – What problems (psychosocial and cognitive) can be observed, in regard to a software development tool (and its replacement) from a user perspective by means of social theory and distributed cognition?

2 – What are the underlying root causes of these problems, from a ’critical’ angle?

3 – Can we further our understanding of ’cognitive load drivers’ and ’cognitive waste’ by means of deploying a ’critical’ analysis?

3.3. Study design consideration

In this study we revisit and use qualitative data previously collected, and reason abductively (Martin, 2019)[7] in regards to these observations using extant literature from social sciences. So, the design is labelled as ’flexible explorative case study’ (Runeson et al., 2012) using ’grounded theory’(Charmaz, 2014). We take deploy a ’critical ethnography’ (Prasad, 2018) angle in this study, drawing on (Rosen, 2000).

In closing, in regards to the labelling of this research – we investigate tool use, and consequences thereof, within a culture in an exploratory fashion, so the research approach is, in our humble opinion the inherently ethnographic. With that said, we are perfectly fine with leaning on the Chicago school tradition root of grounded theory (Charmaz, 2014), (Bryant, 2017) and simply label the study ’an industrial case study, using grounded theory and an ethnographic approach/data set’.

3.4. Case description

We are operating within ’case study’-methodology (Runeson et al., 2012). The case in this paper is a large, 1000+ developers, international multi-site software development organisation. The object under study is cognitive load, as a consequence of digital work environment in software industry, as experienced by software developers. Over all we study the object from an individual as well as a distributed perspective. In this specific study the unit of analys is the individual engineer but in the distributed cognitive context (Hollan et al., 2000).

3.5. Data collection & data set construction

In this study we rely on 3 semi structured interviews (1 test engineer, 1 development engineer, 1 tools responsible). Interviews were recorded, and transcribed by first author who also translated them into English. Interviews were conducted in another language.

The first interview was conducted in the first field trip. We then approached the ’gatekeeper’/company contact point and were provided access to a person responsible for tool activities within the company in order to provide some background. This lead to an informal discussion lasting for about an hour during which notes were taken. It was not recorded on account on the person being uncomfortable in recording a session that they was unprepared for.. This was later recorded in a formal interview. The engineer was approached and asked if willing to volunteer for an interview to add a different perspective on the previous discussions for ’triangulation’ purposes (van Maanen, 1979). We also added extant literature from ’social sciences’ (Luastsen et al., 2017) (e.g. ’alienation’) to the data set in order to explain the phenomena we encountered after the open coding.

The interviews were fractured at the transcriptions, using emacs text line editor and .csv formatted spatially separating the interviewers and the interview subject. Time codes were added at key passages.

3.6. Analysis

Coding was executed in two stages (’open’ and ’focused’) as suggested by Charmaz (2014). ’Open coding’ was executed, in multi-pass fashion to allow for ’analytical bracketing’ (Rennstam & Wäste-

[7] See also (Bryant & Charmaz, 2019), (Bryant, 2017). Further, the definition offered by Alvesson and Sköldberg (2018, p. 4-8) is very succinct.

fors, 2018) [8], using the transcripts printed on paper and Pilot-V disposable reservoir pens in red and blue. Codes for relevant 'incidents'(Glaser, 1978) were written on the printed transcripts. Coding was executed 'chunk-by-chunk' rather than actual 'line-by-line'.

The codes were then transferred to Post-It stickers (set A) as prescribed by Bryant (2017) and grouped, thus forming 'categories'. The 'categories' were then named. Prior to 'open coding' we had used (Prasad, 2018)) in order to sensitise ourselves with the 'critical' tradition and to achieve some level of "theoretical sensitivity"(Glaser, 1978) for the phenomena under study. We then produced a few 'memos' (Charmaz, 2014) outlining the findings of the 'open coding' stage. Prior to this stage we read (D'Ignazio & Klein, 2020) in order to achieve some level of senzitation in regards to contemporary critical discourse in software settings. We then read, and coded using Post-It-stickers, 'Social theory' by Luastsen et al. (2017) to further explore our findings (Set B).

'Focused coding' was done in three stages: firstly we grouped the two sets (A and B) of Post-It stickers and based on this we produced another 'memo' into which fractured interview transcripts were added forming lengthy 'excerpt commentary units' (Rennstam & Wästefors, 2018). This process was also done in multi-pass fashion to allow for 'analytical bracketing'(Rennstam & Wästefors, 2018), but the purpose was to generate what van Maanen (1979) refers to as 'second order themes' (i.e. not only allowing the 'what' and 'how' but also the underlying ethnographically important 'why' (Sharp et al., 2016)) to emerge. Following this process we started writing the analysis using 'memoing'. Finally, these 'memos' were fused into one that largely makes up the analysis section of this paper.

The rendering of our theory (Charmaz, 2014) was done in the same fashion as the one we generated in previous work (Helgesson et al., 2021), using Post-It stickers and an A1 cardboard sheet. We have previously found this technique extremely useful as it allows for a level of tactile feedback and interaction impossible if one uses software. The theory was then transferred into digital format[9]. It has been constructed from the actual 'memos' and presented in a seminar.

3.7. Literature review

We did informal/preliminary literature surveys prior and post open coding. Following the focused coding and theory generation we extended this using a similar strategy as in (Helgesson & Runeson, 2021). We queried ACM from 2010 to 2013 with queries in regards to 'marxism/critical theory' in conjunction with 'alienation' [10] as well as 'social theory' [11]. In doing so we found 48 and 36 papers respectively. Having done a reading of the abstracts we found no papers directly relevant to this study. But, we conclude that the findings of Melegati and Wang (2021) are correct - the impact of the critical tradition in the area of software development/engineering has been small. We also similarly queried IEEE [12] [13] with similar queries finding 99 and 87 titles respectively. From an abstract reading we could find no text directly relevant for this study.

[8] See also: (Gearing, 2004), (Tufford & Newman, 2012) for in-depth discussions in regards 'bracketing'.

[9] While some reviewers have previously lamented on the absence of artefacts in regards to 'coding', 'memoing' and 'theory generation' we are in agreement of Bryant (2017) – these artefacts are private. The paper is the actual outcome, and artefact, of the study. With that said, we are, of course happy to allow for audit of our anonymised transcripts and field notes – please contact first author for details

[10] [[All: "marxism"] OR [All: "critical theory"] OR [[All: "marxism"] AND [All: "alienation"]]] AND [[All: "software development"] OR [All: "software engineering"]] AND [E-Publication Date: (01/01/2010 TO 12/31/2023)]

[11] [All: "social theory"] AND [[All: "software development"] OR [All: "software engineering"]] AND [E-Publication Date: (01/01/2010 TO 12/31/2023)]

[12] (("Full Text Metadata":"marxism" OR "Full Text Metadata":"critical theory") OR ("Full Text Metadata":"marxism" AND "Full Text Metadata":"alienation")) AND ("Full Text Metadata":"Software Development" OR "Full Text Metadata":"Software Engineering")

[13] ("Full Text Metadata":"social theory") AND ("Full Text Metadata":"Software Development" OR "Full Text Metadata":"Software Engineering")

4. Analysis

4.1. Theory and theorising

We draw on excerpt commentary units (Rennstam & Wästefors, 2018) here; ’Tell’, ’Show’, ’Explain’ but intentionally convoluted and meta-level. The two engineers describe similar phenomena and the tool responsible provides the underlying background. ’Categories’ are contained in [hard brackets], explanations in <sharp brackets>, narrative is denoted with ’–talking dash’ and dialogue is contained within “quotes”.[14] is work in progress. The elements we explore in this theory are shown in Figures 1.

We see our ’theory’ (Abend, 2008) as an abstract account of events that have taken place within a software development organisation that allows for ’theorisation’ (Sutton & Staw, 1995),(Weick, 1995) in regards to consequences of poor tooling within the software development industry.

4.2. Absent UCD, cognitive waste, cause and consequence for the individual digital worker

The first facet of the theory we generate from the data set explores absence of user centred design in relation to digital tools and cognitive waste from the perspective of the individual digital worker. The ’categories’ we choose to explore are: **absent ucd**, **missing functionality**, **cognitive waste**, **frustration**, **coping strategies** and **alienation**.

The first facet of the theory, the individual perspective, is visualised in Figure 1, and each ’category’ is explored and further explained in a corresponding subsection below.

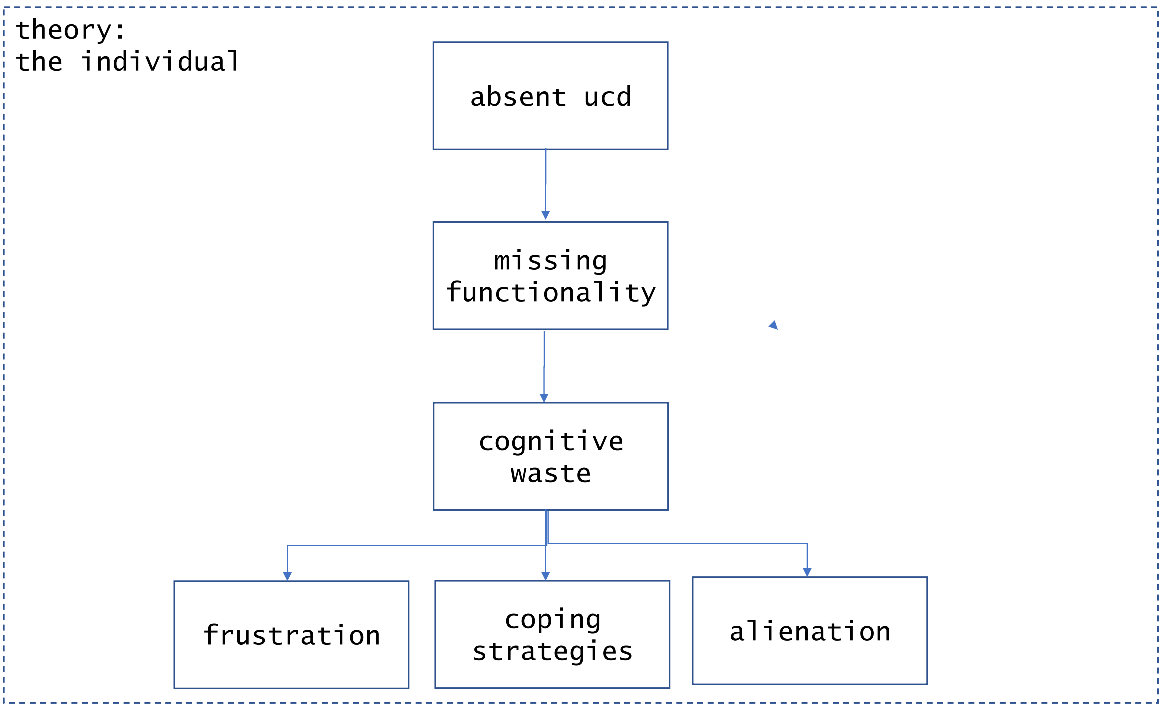


Figure 1 – Theory – consequence for the individual worker caused by absent UCD

4.2.1. cognitive waste

4.2.2. frustration

The following discussion between the interviewer and the informant, reveals that the developer is acutely aware of the problems associated with the tools with which they are provided by the corporation.

“So... what digital tools do you use?”

[14]We use ’categories’(Charmaz, 2014) rather than ’codes’ or ’themes’ as we have aggregated the ’codes’ into ’categories’ (of ’codes’) and we will be visualising ’categories’ in our substantive theory.

“Well – ’Tool 1’, ’Tool 2’ and.... (<trying to remember>)... well, actually I try to avoid all tools if possible!’ They are a pain in the butt to work with. I can’t come up with any good things to say about them[**alienation**]! Starting off with ’Tool 1’: If you are to file an issue, I have to fill in a lot of input fields. About twenty of them[**waste**]. For every field you have to think for one minute[**cog waste**], then you can figure out how *fun* <sarcasm>[**frustration**] writing issues is....”

We clearly see that the effort that the developer need to spend on filing ’issues’ is clearly identified as ’waste’ and that it causes frustration.

4.2.3. absent ucd

4.2.4. alienation

The following two comments from developers display a connection between the absence of user centered design activities and alienation of employees.

”I have a feeling that the software developers haven’t been asked what requirements they have[**no ucd**]. Instead the requirements come from, I don’t really know but I assume, project people and operator managers etc. And they live in a different world than we do[**alienation**].”

”At this company there is an affinity for adding stuff. But not for removing them...This company is extremely poor at that. If *they*[**alienation**] were to start *counting how much time we spend* on this[**waste**] they would probably get shocked.´´

The way the developers use ’*they*’ clearly indicates that individuals see themselves as distanced, and estranged from their employer.

4.2.5. missing functionality

This segment described the missing functionality in the tool discussed at detail. It focuses on two aspects, the plethora of unnecessary information and selections that needs to be supplied when reporting an ’issue’ (a defect report) and missing search functionality preventing the users to navigate the system efficiently. It has been shorted considerably to fit page limitation (currently it only discusses one issue, the interaction, not the missing search functionality).

The following passage in which the interviewer discusses the system with one of the engineers identifies the first source of waste, that the user has no support for recalling his/her most common selections. Instead they have to tediously selected from very long scroll lists in the user interface.

“So there is no *support* [**missing functionality**] that allows you to *see just your* common selections?”

“No, and that would be really good to have[**no support**]. ”

The need for the missing functionality is obvious from the perspective of the developer. Again we see the alienation and the frustration the poor tooling causes. “With one new field?” “No. Three new fields <sarcasm>. Why couldn’t they have removed the other fields[**no support**]? (<implying that redundant work is necessary>)”

“But you have to fill in every field? Or you can’t [**oppression**] file your issue?”

“No, exactly... <sarcasm>[**frustration**] – there is no point in looking at the ’old stuff’ (<the fields that are no longer used>), but you still need to fill them in. At this company there is an affinity for adding stuff. But not for removing them... and ’clean up’. This company is extremely poor at that. If *they*[**alienation**] were to start *counting how much time we spend* on this[**waste**] they would probably get shocked. But it is not as easy as simply filing an issue in ’Tool 1’, once you have done that you must find it again, and go into it and ’do modify’ as you must add what projects you want the issue tagged for. This can’t be done *while you actually file* the issue. So you *can’t even file all information is relevant in one go*. So you file, save, find it *again*[**waste**], select ’modify’, enter the additional info and save it *again*[**waste**]. “That sounds harsh.” “I get *paid*[**alienation**] for this. It is not as bad as it could be. But it could be handled a lot better [**frustration**]...”

Switching over to the other engineer we see that he/she is well aware of the same issues discussed in the previous segment.

“... one thing that strike me in the earlier interviews... All these fields in ’Tool 1’ that need to be entered, all variants that exist...”

“A lot of different tabs with a lot of different fields. And I am expected to fill out a subset of these, but it is hard to understand which fields that should be filled out. And what information is expected to be entered. So it appears as there has been requirements coming from a lot of different directions... We need this, and that, and that... And while I’m sure that a lot it is relevant for the projects, it is not relevant for me <as a developer> **[alienation]**”

“So you don’t really know what <information> you are expected to hand over <enter into ’Tool 1’>?

“Exactly. Sometimes it is just ridiculous **[frustration]** - a lot of these fields are designed under the premise that we are working with an error, but ’Tool 1’ is used for development as well, and to fill out ’in what version of the system the error was detected’ is really not applicable but still you have to fill out these fields.”

The developer here clearly indicates that the user does not know neither what information to supply, neither where nor why. This largely confirms what was discussed in the previous section.

“Something else that was indicated was that some of these fields were made obsolete/replaced by new fields, yet they remained.”

“Correct. Another thing that has occurred is that something new <kind of data> is invented and a field is reused for this purpose. There was ’Found in’ and ’Found during’ and in ’Found during’ you were supposed to fill out what team you were part of. Or something along those lines. Simply because a field was reused. These kind of problems are quite common - you don’t really know how to use the system, what information to enter.”

“That it is counter intuitive?”

“Counter intuitive, yes. It gives very little guidance of what you are supposed to do. And it is very easy to do the wrong thing. To enter a state that you don’t understand how to exit <refers to state of issue not state of program>).”

The consequences of absent user design is clearly visible to the developer. The tool, and corresponding tasks, have become ’counter intuitive’ and makes the user error prone,

4.2.6. coping strategies

“... and to search for information in ’Tool 1’ is a problem?”

“Yes... I have solved in this way[**coping strategy**]... I get a mail for every issue in ’Tool 1’ that is created for my team, and I save these. And I search in these mails because searching in ’Tool 1’ for something that is handled and closed is not something I have ever been able understand how to do. And this is actually something we struggle with in newer systems as well. That you get an error report and you think ’didn’t we have an issue like that six months ago?’ - then you would like to find out what caused it, and how it was solved. But once an issue has been closed... it does exist somewhere in the database... but it is really hard to locate, it is really difficult to search for.. Searching for everything that I have been involved in for instance, or... It might be possible to do if you are really good at creating queries, but it is quite hard for normal users <lit. ordinary mortals>. ”

4.3. Absent UCD, cognitive waste, cause and consequence for the cause and consequence for the corporation

The analysis originally also contained a theoretical account of the corporate perspective of this specific case/tool, based on an interview with the person responsible for development and maintenance of the system, and it successor. The person fully acknowledged the issues discussed by the users, and provided an explanation on why the situation had been allowed to deteriorate. For space reasons we only present a brief overview.

The root cause was absent organisational ownership, compounded by a confounding factor – namely company culture and cultural differences, as well as a managerial lack of understanding of user needs. As a consequence the tool was very inefficient, on account of missing functionality. In addition to the corresponding cognitive and temporal/fiscal waste stemming from the absent UCD, the person also noted that the data quality and reliability (of the data that could be extracted from the system) was very low.

While having observed the aforementioned problems, and the fact that the tool had been voted 'the worst tool within the organisation' by one of the development organisations the rationale of the business case used to replace the existing tool was simply licensing fees rather than lacklustre functionality and reliability. As a consequence the same mistakes were made in the customisation of the replacement – ultimately resulting in a similarly inefficient tool. Again...

4.4. Conclusion

There is a complete absence of understanding of what the user needs from the tool, since no user centered design steps are taken to ensure that it functions properly and serves the needs of the organisation and the users. Only the management perspective of the tool is taken into consideration when designing and implementing it. This is the *complete opposite* of core tenets of human factors engineering practice (Gulliksen et al., 2015) (Wickens, Hollands, Banbury, & Parasuraman, 2015).

In fact it can be noted that Wickens et al. (2015, p. 1) explicitly state that human factors engineering came about "...just after World War II when experimental psychologists were called in to help understand *why pilots were crashing perfectly good aircraft* (Fitts & Jones, 1947)..." It is well known within the company that 'Tool 1' is extremely unpopular among the users in software production, yet it is only changed for short term profitability reasons (lower licensing fees). The frustration it causes, the cognitive and temporal waste it causes is not considered in the business case, neither is the poor quality of data – that renders it useless as an analytical tool – which is one of the main reasons it is there in the first place.

5. Discussions in regard to RQ's

1 – What problems (psychosocial and cognitive) can be observed, in regard to a software development tool (and its replacement) from a user perspective by means of social theory, distributed cognition and "perspectives' respectively? – We focus on psychosocial rather than cognitive issues in this study. We see several aspects of such issues in connection to what we have read in social theory (e.g. 'alienation' and 'frustration',). These, to some extent correspond to what Gulliksen et al. (2015) describes. The "psychological distress" as described by Sedano et al. (2017) is clearly visible in our data set as well.

2 – What are the underlying root causes of these problems, from a 'critical' and cognitive perspectives, respectively? – The underlying root causes of these problems, from a 'critical' perspective can mainly be attributed to absence of user centred design when developing digital tools as a consequence of absent mandate/responsibility and organisational issues.. From a cognitive perspectives, it is mainly related to the presence of 'poorly functioning digital tools' (Håkansson & Bjarnason, 2020) in conjunction with the limits of the human mind (Wickens et al., 2015) (Miller, 1956). The underlying root cause is absence of user centered design (Wickens et al., 2015)[15].

3 – Can we further our understanding of 'cognitive sustainability', 'cognitive productivity' and 'cognitive waste' by means of deploying a 'critical perspective'? – Yes. Indeed we see that we further our understanding 'cognitive productivity' and 'cognitive waste' by means of a 'critical' perspective.

6. Validity/Generalisability

GT studies are commonly evaluated based on the following criteria (Charmaz, 2014) (Stol et al., 2016):

[15]The phenomena of poorly designed (from user centered design/interaction design perspective) software development tools is well known within the software engineering community – it is referred to as 'developers in their own dog-food'

Credibility: *Is there enough data to merit claims of the study?* – Yes. While the data set the theory is grounded in is limited, it still is indicative of a relevant problem.

Originality: *Does the results offer new insight?* – Hopefully, yes. It provides an inside account on the consequences of absence of User Centered Design in the design process of software development tools. It further shows that the there is no tension in relation between cognitive sustainability and efficiency respectively; rather that the tension lies between cognitive sustainability/efficiency and corporate lack of understanding of the need of user centred design.

Usefulness: *Is the theory generated relevant for practitioners?* – As of yet, we do not know. With that said, it is a novel way of explaining the intangible cognitive waste, drain and frustration that occur as a consequence of poorly functioning tools. So hopefully yes.

Resonance: *Does the theory resonate with practitioners?* – from the limited feedback received at the time of writing, yes.

Ethnography is tricky, van Maanen (1979) lists several ways in which the researcher can be ¨"misled" by informants [16] . In this case we don't see that as an issue. The data set triangulates our findings well What we witnessed largely agreed with what the informants let us in on. We thus consider ourselves "explicitly and extensively informed" (Rosen, 2000).

In regards to general criticism of single case generalisation we humbly point to Anzai and Simon (1979): "It may be objected that a general psychological theory cannot be supported by a single case. One swallow does not make a summer, but one swallow does prove the existence of swallows. And careful dissection of even one swallow may provide a great deal of reliable information about swallow anatomy.".

In the end of the introduction Rosen (2000) cites the dissertation of Kunda (unpublished), to make the point that: "...ethnography is the only human activity in the social sciences. As a method it is not divorce from the modes of experience that I consider human, that is, it is not divorced from my 'reality'.It is therefore one of the few ways of doing research that speaks the 'truth' as I understand it.". We couldn't agree more.

7. Emerging concepts for future studies

In addition to the generated theory, we made some additional observations that we will shortly present and theorise around.

7.1. learning/unlearning

When discussing systems in general with one user we noted that there might be something interesting at play when systems are changed/replaced by something smilar, yet different. The person stated that when a tool was replaced it became very difficult to adapt the mental model of how to operate the system.

Allowing for some theorisation here it is not farfetched to think that when a user has to make an active choice/recall on 'how' to perform an activity (depending on what tool is being used) this will result in unnecessary cognitive load, since what was previously an instinctive recall now has become an active choice. This would not be far fram what Rasmussen (1983) observed, that response to system stimuli is dependent on whether it is a 'skill', 'rule' or 'knowledge'.

7.2. cognitive work & labour

When discussing cognitive load it is interesting to note that it is, similarly to 'load' in physics, essentially momentary. In order to fully understand cognitive waste, it would make sense to reason in terms of cognitive work as a temporal aggregation of cognitive load (similarly to the relation between 'power'/'load' in physics and 'energy'/'work'). Similarly the process of exerting cognitive load for wage could/should be denoted 'cognitive labour'.

[16]See also: (Briggs, 1986) and (Agar, 2008)

8. Discussion

The problem with poorly functioning software development tools is rather well known within the industry , to the point that the term 'developers eat their own doogfood' was been coined decades ago. In this case, however, the issue is rather that 'engineers eat manager dogfood', in a sense. The issues with the tool in question are well known within the organisation, yet there is seemingly little interest in doing something about it. Here we might draw similarities with early industrialisation, where workers were provided tools by industrialists and User Centered Design and ergonomics lay a long time into the future.

When discussed in course seminars the 'mundanity' of the story were reflected on by students from varying disciplines. Given the high level of identification in the kafka-esque story displayed by audience at seminars, this might suggest a wider applicability – and a wider problem[17].

In light of the post conference publication of PPIG we will pend the discussion section post conference presentation.

9. Acknowledgements

We wish to thank librarian Andeas Karman for (once and again) helping out with database queries, and esteemed supervisor prof. Per Runeson for input and proof read (as well as putting up with the grounded theory shenanigans).

10. References

Abend, G. (2008). The Meaning of 'Theory'. *Sociological Theory*, *26*(2), 173–199.

Agar, M. H. (2008). *the Professional Stranger An informal Introduction to Ethnography* (2nd ed.). Bingley, UK: Emerald Group Publishing Ltd.

Alvesson, M., & Sköldberg, K. (2018). *Reflexive Methodology - New Vistas for Qualitative Research* (3rd ed.). London, UK: SAGE Publications.

Anzai, Y., & Simon, H. A. (1979). The Theory of Learning by Doing. *Psychological Review*, *86*(2), 124–140.

Bertelsen, O. W. (1997, November). Toward A Unified Field Of SE Research And Practice. *IEEE Software*, *14*(6), 87–88.

Briggs, C. L. (1986). *Learning how to ask - A sociolinguistic appraisal of the role of the interview in social science research*. Cambridge ; New York: Cambridge University Press.

Bryant, A. (2017). *Grounded Theory and Grounded Theorizing – Pragmatisism in Research Practice*. Oxford, UK: Oxford University Press.

Bryant, A. (2019). *The varieties of grounded theory* (1st ed.). London, UK: SAGE Publications Ltd.

Bryant, A., & Charmaz, K. (2019). *The SAGE Handbook of Current Developments in Grounded Theory*. London, UK: SAGE Publications.

Burnett, M., Peters, A., Hill, C., & Elarief, N. (2016, May). Finding Gender-Inclusiveness Software Issues with GenderMag: A Field Investigation. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (pp. 2586–2598). San Jose, California, USA: Association for Computing Machinery. doi: 10.1145/2858036.2858274

Charmaz, K. (2014). *Constructing Grounded Theory* (2nd ed.). London, UK: SAGE Publications.

Charmaz, K., & Mitchell, R. (2001). Grounded Theory in Ethnography. In *Handbook of Ethnography*. London, UK: SAGE Publications.

Davis, M. S. (1971, June). That's Interesting!: Towards a Phenomenology of Sociology and a Sociology of Phenomenology. *Philosophy of the Social Sciences*, *1*(2), 309–344.

Dey, I. (1993). *Qualitative Data Analysis: A user-friendly guide for social scientists*. London, UK: Routledge.

[17] At the time of writing (May 2023) there has been a discussion in media e.g. https://www.expressen.se/ledare/patrik-kronqvist/politikerna-skryter-men--sverige-ar-ett-u-land/ in regards to poorly functioning systems in the wake of the ongoing digitalisation of society which also supports that, albeit not applicable to software development tools and organisations, this is becoming a societal issue at a large scale.

Dey, I. (1999). *Grounding Grounded Theory - Guidelines for Qualitative Inqury*. San Diego, California, USA: Academic Press.

D'Ignazio, C., & Klein, L. F. (2020). *Data Feminism* (1st ed.). Cambridge, Mass. USA: MIT Press.

Fritz, T., & Müller, S. C. (2016, March). Leveraging Biometric Data to Boost Software Developer Productivity. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Vol. 5, pp. 66–77).

Froomkin, A. M. (2002). Habermas@Discourse.Net: Toward a Critical Theory of Cyberspace. *Harvard Law Review*, *116*(3), 749–873.

Fuchs, C., & Sevignani, S. (2013, June). What Is Digital Labour? What Is Digital Work? What's their Difference? And Why Do These Questions Matter for Understanding Social Media? *tripleC: Communication, Capitalism & Critique. Open Access Journal for a Global Sustainable Information Society*, *11*(2), 237–293. doi: 10.31269/triplec.v11i2.461

Gearing, R. E. (2004, December). Bracketing in Research: A Typology. *Qualitative Health Research*, *14*(10), 1429–1452. (Publisher: SAGE Publications Inc) doi: 10.1177/1049732304270394

Gioia, D. A., Corley, K. G., & Hamilton, A. L. (2013, January). Seeking Qualitative Rigor in Inductive Research: Notes on the Gioia Methodology. *Organizational Research Methods*, *16*(1), 15–31. Retrieved from `https://doi.org/10.1177/1094428112452151` (Publisher: SAGE Publications Inc) doi: 10.1177/1094428112452151

Glaser, B. G. (1978). *Theoretical Sensitivity*. CA, USA: Sociology Press.

Glaser, B. G. (1992). *Emergence vs Forcing - Basics of Grounded Theory Analysis*. CA, USA: Sociology Press.

Gonçales, L. J., Farias, K., & da Silva, B. C. (2021, August). Measuring the cognitive load of software developers: An extended Systematic Mapping Study. *Information and Software Technology*, *136*, 106563. doi: 10.1016/j.infsof.2021.106563

Gulliksen, J., Lantz, A., Walldius, Å., Sandblad, B., & Åborg, C. (2015). *Digital arbetsmiljö, en kartläggning (RAP 2015:17)* (Tech. Rep.). Retrieved from `https://www.av.se/arbetsmiljoarbete-och-inspektioner/kunskapssammanstallningar/digital-arbetsmiljo-kunskapssammanstallning/`

Helgesson, D. (2021). *Exploring grounded theory perspectives of cognitive load in software engineering* (Unpublished doctoral dissertation).

Helgesson, D., Appelquist, D., & Runeson, P. (2021). A grounded theory of cognitive load drivers in novice agile software development teams. In *Unpublished manuscript.* Retrieved from `http://arxiv.org/abs/2107.04254`

Helgesson, D., Engström, E., Runeson, P., & Bjarnason, E. (2019). Cognitive Load Drivers in Large Scale Software Development. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering* (pp. 91–94). Piscataway, NJ, USA: IEEE Press. doi: 10.1109/CHASE.2019.00030

Helgesson, D., & Runeson, P. (2021). Toward grounded Theory perspectives in Software Engineering. In *Ppig'21.*

Hilderbrand, C., Perdriau, C., Letaw, L., Emard, J., Steine-Hanson, Z., Burnett, M., & Sarma, A. (2020, June). Engineering gender-inclusivity into software: ten teams' tales from the trenches. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 433–444). Seoul, South Korea: Association for Computing Machinery. doi: 10.1145/3377811.3380371

Hollan, J., Hutchins, E., & Kirsh, D. (2000, June). Distributed Cognition: Toward a New Foundation for Human-computer Interaction Research. *ACM Trans. Comput.-Hum. Interact.*, *7*(2), 174–196.

Håkansson, E., & Bjarnason, E. (2020, August). Including Human Factors and Ergonomics in Requirements Engineering for Digital Work Environments. In *2020 IEEE First International Workshop on Requirements Engineering for Well-Being, Aging, and Health (REWBAH).* doi: 10.1109/REWBAH51211.2020.00013

Kirsh, D. (2000). A Few Thoughts on Cognitive Overload. *Intellectia*(30), 19–51.

Krüger, S., & Johanssen, J. (2014, September). Alienation and Digital Labour—A Depth-Hermeneutic Inquiry into Online Commodification and the Unconscious. *tripleC: Communication, Capitalism & Critique. Open Access Journal for a Global Sustainable Information Society*, *12*(2), 632–647. doi: 10.31269/triplec.v12i2.548

Lenberg, P., Feldt, R., & Wallgren, L. G. (2015, September). Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, *107*, 15–37.

Lorey, T., Ralph, P., & Felderer, M. (2022). Social science theories in software engineering research. In *Proceedings of the 44th international conference on software engineering* (p. 1994–2005). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3510003.3510076` doi: 10.1145/3510003.3510076

Luastsen, C. B., Larsen, L. T., Nielsen, M. W., Ravn, T., & Sörensen, M. P. (2017). *Social Theory - A Textbook* (1st ed.). New York, NY, USA: Routledge.

Martin, V. (2019). Using Popular and Academic Literature as Data for Formal Grounded Theory. In *The SAGE Handbook of Current Developments in Grounded Theory.* London, UK: SAGE Publications.

Melegati, J., & Wang, X. (2021, May). Surfacing Paradigms underneath Research on Human and Social Aspects of Software Engineering. In (pp. 41–50). IEEE Computer Society. doi: 10.1109/CHASE52884.2021.00013

Miller, G. A. (1956). The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychological review*, *63*(2), 81–97.

Müller, S. C., & Fritz, T. (2016, May). Using (Bio)Metrics to Predict Code Quality Online. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 452–463). (ISSN: 1558-1225) doi: 10.1145/2884781.2884803

Naur, P., & Randell, B. (1969, January). *Software engineering: Report on a conference sponsored by the nato science committee* (Tech. Rep.). Scientific Affairs Division, NATO.

Nygren, K. G., & Gidlund, K. L. (2016). *The Pastoral Power of Technology. Rethinking Alienation in Digital Culture*. Brill. (Pages: 396-412 Section: Marx in the Age of Digital Capitalism) doi: 10.1163/9789004291393_013

Pfeiffer, S. (2014, September). Digital Labour and the Use-value of Human Work. On the Importance of Labouring Capacity for understanding Digital Capitalism. *tripleC: Communication, Capitalism & Critique. Open Access Journal for a Global Sustainable Information Society*, *12*(2), 599–619. doi: 10.31269/triplec.v12i2.545

Prasad, P. (2018). *Crafting Qualitative Research: Beyond Positivist Traditions*. Routledge.

Ralph, P., Chiasson, M., & Kelley, H. (2016). Social theory for software engineering research. In *Proceedings of the 20th international conference on evaluation and assessment in software engineering.* New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2915970.2915998` doi: 10.1145/2915970.2915998

Rasmussen, J. (1983, May). Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-13*(3), 257–266. doi: 10.1109/TSMC.1983.6313160

Rennstam, J., & Wästefors, D. (2018). *Analyze! Crafting your Data in Qualitative Research* (1st ed.). Lund, Sweden: Studentlitteratur.

Rorty, R. (1979). *Philosophy and the mirror of nature* (1st ed.). Prinston, NJ, USA: Princeton University Press.

Rosen, M. (2000). *Turning Words Spinning Worlds* (Vol. 25). Harwod.

Runeson, P., Engström, E., & Storey, M.-A. (2020). The Design Science Paradigm as a Frame for Empirical Software Engineering. In M. Felderer & G. H. Travassos (Eds.), *Contemporary Empirical Methods in Software Engineering* (pp. 127–147). Cham: Springer International Publishing. doi: 10.1007/978-3-030-32489-6_5

Runeson, P., Höst, M., Rainer, A., & Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons.

Sedano, T., Ralph, P., & Péraire, C. (2017, May). Software Development Waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (pp. 130–140). doi: 10.1109/ICSE.2017.20

Sharp, H., Dittrich, Y., & de Souza, C. R. B. (2016, August). The Role of Ethnographic Studies in Empirical Software Engineering. *IEEE Transactions on Software Engineering*, *42*(8), 786–804.

Simon, H. A. (1969). *The sciences of the artificial* (2nd ed.). Cambridge, USA: MIT Press.

Simon, H. A. (1971). Designing organizations for an information-rich world. In M. Greenberger (Ed.), *Computers, communication, and the public interest* (pp. 40–41). Baltimore. MD: The Johns Hopkins Press.

Stol, K.-J., Ralph, P., & Fitzgerald, B. (2016, May). Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 120–131). doi: 10.1145/2884781.2884833

Sutton, R. I., & Staw, B. M. (1995). What Theory is Not. *Administrative Science Quarterly*, *40*(3), 371–384. (Publisher: [Sage Publications, Inc., Johnson Graduate School of Management, Cornell University]) doi: 10.2307/2393788

Söderberg, J. (2011). *Free software to open hardware: critical theory on the frontiers of hacking* (No. 17). Gothenburg: University of Gothenburg, Dept. of Sociology.

Tufford, L., & Newman, P. (2012, January). Bracketing in Qualitative Research. *Qualitative Social Work*, *11*(1), 80–96. (Publisher: SAGE Publications) doi: 10.1177/1473325010368316

van Maanen, J. (1979). The Fact of Fiction in Organizational Ethnography. , 13.

Vorvoreanu, M., Zhang, L., Huang, Y.-H., Hilderbrand, C., Steine-Hanson, Z., & Burnett, M. (2019, May). From Gender Biases to Gender-Inclusive Design: An Empirical Investigation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (pp. 1–14). New York, NY, USA: Association for Computing Machinery.

Weick, K. E. (1995). What Theory is Not, Theorizing Is. *Administrative Science Quarterly*, *40*(3), 385–390. (Publisher: [Sage Publications, Inc., Johnson Graduate School of Management, Cornell University]) doi: 10.2307/2393789

Wickens, C. D., Hollands, J. G., Banbury, S., & Parasuraman, R. (2015). *Engineering Psychology & Human Performance*. Psychology Press.

Directions in Computational Music

Ian Clester
Center for Music Technology
Georgia Institute of Technology
ijc@gatech.edu

Abstract
This doctoral consortium submission describes my work on computational music, which has primarily focused on three problems: representation, distribution, and environment. Among other topics, this work relates to programming language design, interactive development, live coding, and programming interfaces. I welcome feedback from the PPIG community, especially regarding methodology and future directions, which I anticipate will prove helpful at this juncture midway through my doctoral program.

1. Introduction
Computational music is music represented as a computer program rather than as fixed media (such as a score, DAW project, MIDI file, or audio file). Its representation may thus be far more compact than its output (as it can directly encode abstractions of the composer's choice), and it may encompass many possible outputs by depending on external input or state (e.g. using a different seed for a random number generator). This notion is closely related to generative music (Eno, 1996), aleatoric music (Cage, 2004), process music (Reich, 1965), and algorithmic composition (Essl, 2007).

The general aim of my work is to realize the possibilities of computational music, primarily through the analysis and development of languages, tools, and interfaces for its creation and use. I have been pursuing this aim along three lines of investigation. The first is representation: how can computational compositions be written down, and which abstractions are useful or necessary for a computer music language? The second is distribution: how can computational compositions be effectively distributed (i.e. with ease approximating that of distributing static, recorded music)? The third is environment: what is a 'good' interface for creating computational compositions, and how can the possibilities enabled by computational composition be extended to composers who do not consider themselves programmers?

These questions relate to several topics of interest in the PPIG community. The question of representation relates to programming language design and choice of abstraction, as embodied in the variety of computer music languages and libraries available today. The question of distribution relates to issues of software development, dependency management, containerization, and virtualization. And the question of environment relates to user interfaces, interactive development, HCI, and programming pedagogy.

2. Existing & Related Work
I have conducted initial work in each of the three lines of investigation described above. The question of representation is explored in Aleatora (Clester & Freeman, 2021), a composition framework that challenges concepts taken for granted in most computer music systems. It is developed as a library for an existing, general-purpose programming language (Python), and it eschews hard distinctions between score and orchestra or between synthesis and control. Aleatora is built around the unifying abstraction of *streams*, which support sequential, parallel, and function composition. Streams span the abstraction hierarchy from patterns all the way down to samples, enabling the composer to compose at any level. The generative composition itself is a first-class value within the language.

The challenge of distribution is addressed by Alternator (Clester & Freeman, 2022), a distribution system that bundles computational compositions (written in existing languages such as Csound, Pure Data, ChucK, Python, JavaScript, C, and Rust, among others) into self-contained packages for client-side execution via WebAssembly. This presents the listener with a familiar music player interface which supports common functions such as pausing and seeking, without requiring additional work on the part of the composer/programmer. In addition, it enables each listener to hear different pieces and generative variations without requiring a central server to run separate sandboxes for all concurrent listeners.

Finally, the question of environment is taken up by LambDAW (Clester & Freeman, 2023). LambDAW takes the digital audio workstation (DAW) as its starting point and brings computation into the timeline, introducing the concept of *expression items* that generate their contents by evaluating a Python expression. Like spreadsheet formula, these expressions can reference other items in the timeline and transform them, enabling the composer to freely mix code and data, combining the expressive power of programming with the direct manipulation of the DAW. By taking an existing DAW (REAPER) as its starting point, LambDAW augments the DAW's capabilities and offers an entry point into computation for composers and producers who already use the DAW in their creative workflows.

The full set of related work for these three projects is beyond the scope of this short submission; for a more complete overview, see the related work section of each project's paper. To briefly summarize, Aleatora is related to computer music languages such as SuperCollider (McCartney, 2002) and Nyquist (Dannenberg, 1997) and work on streams in general-purpose languages such as Scheme and Haskell. Alternator is related to artistic works such as Generative.fm (Bainter, 2019) and generative radio stations such as `rand()%`[1]and Streaaam (Hollerweger, 2021). And LambDAW is related to environments such as EarSketch (Magerko et al., 2016), Manhattan (Nash, 2014), and DeadCode (Beverley, 2020), as well as work on end-user programming more broadly.

3. Questions for the Community

3.1. Research Questions & Methodology

A key question for the community is how best to pose overarching research questions that relate to this agenda. Of the various frames offered by relevant disciplines (music, creative coding, HCI, Psychology of Programming), which provide the best fit for this kind of work? A related question is what research methodologies may be most effective in answering these questions. Thus far, evaluation has consisted largely of using these systems myself, demonstrating them to others, and discussing their design. What quantitative or qualitative methods (such as surveys, user studies, interviews, etc.) will aid in more rigorously evaluating them? To what extent is practice-led research appropriate for this area of inquiry?

3.2. Potential Directions

There are several potential directions to proceed from my current stage in my research, and I would appreciate community feedback on the merits of these directions and relevant related work. One possibility is "completing the loop." Aleatora already connects nicely to LambDAW, as it is a Python framework well-suited to writing concise expressions that generate audio or MIDI. Completing the loop by building a way to export generative bundles from LambDAW that can run in Alternator would demonstrate a complete approach to generative music, from composition to production to distribution.

Another direction is to enhance Aleatora ('everything as a stream') or Alternator ('everything in a box') by building out infrastructure for embedding more within them, such as SuperCollider and other systems which depend on it. This direction is particularly relevant for recording and replaying live coding performances *as code* (while preserving random elements of the code, which can be re-decided on playback).

A third direction, and perhaps the one most relevant to the PPIG community, is using LambDAW as a jumping-off point to explore hybrid compositional interfaces and programming environments, with the general aim of supporting bricolage by allowing composers to freely mix code and data, taking inspiration from work on interactive visual syntax (Andersen, Ballantyne, & Felleisen, 2020) and *livelits* (Omar et al., 2021). This direction further invites an exploration of how this work—which I have thus far discussed in terms of 'composition' and 'distribution'—relates to liveness, improvisation, and performance. Such an exploration connects to broader questions about the relationship between generative music and live coding (Blackwell & Collins, 2005) and may involve expanding the music/code composition environment beyond conventional computing interfaces such as screens, mice, and keyboards.

[1] `https://www.bbc.co.uk/radio3/cutandsplice/rand.shtml`

4. References

Andersen, L., Ballantyne, M., & Felleisen, M. (2020). Adding Interactive Visual Syntax to Textual Code. *Proc. ACM Program. Lang.*, *4*(OOPSLA). doi: 10.1145/3428290

Bainter, A. (2019). Generative.fm. In A. Xambó, S. R. Martín, & G. Roma (Eds.), *Proceedings of the International Web Audio Conference* (p. 148). Trondheim, Norway: NTNU.

Beverley, J. (2020). Liveness, Code, and DeadCode in Code Jockeying Practice. In *Proceedings of the 2020 International Conference on Live Coding (ICLC2020)* (p. 117-131). Limerick, Ireland: University of Limerick. doi: 10.5281/zenodo.3939222

Blackwell, A. F., & Collins, N. (2005). The Programming Language as a Musical Instrument. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)* (p. 120-130).

Cage, J. (2004). Composition as process: indeterminacy. *Christoph Cox, Daniel Warner, Audio Culture: Readings in Modern Music*, 176–187.

Clester, I., & Freeman, J. (2021). Composing the Network with Streams. In *Audio Mostly 2021* (p. 196–199). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3478384.3478416

Clester, I., & Freeman, J. (2022). Alternator: A General-Purpose Generative Music Player. In *Proceedings of the international web audio conference.* Cannes, France: UCA.

Clester, I., & Freeman, J. (2023). LambDAW: Towards a Generative Audio Workstation. In *Proceedings of the 7th International Conference on Live Coding (ICLC2023).* Utrecht, Netherlands. doi: 10.5281/zenodo.7842002

Dannenberg, R. B. (1997). Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. *Computer Music Journal*, *21*(3), 50–60. (Publisher: The MIT Press) doi: 10.2307/3681013

Eno, B. (1996). *A Year with Swollen Appendices: Brian Eno's Diary*. Faber and Faber.

Essl, K. (2007). Algorithmic composition. In N. Collins & J. d'Escrivan (Eds.), *The cambridge companion to electronic music* (p. 107–125). Cambridge University Press. doi: 10.1017/CCOL9780521868617.008

Hollerweger, F. (2021). Streaaam: A Fully Automated Experimental Audio Streaming Server. In *Audio mostly 2021* (p. 161–168). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3478384.3478426

Magerko, B., Freeman, J., Mcklin, T., Reilly, M., Livingston, E., Mccoid, S., & Crews-Brown, A. (2016). EarSketch: A STEAM-Based Approach for Underrepresented Populations in High School Computer Science Education. *ACM Trans. Comput. Educ.*, *16*(4). doi: 10.1145/2886418

McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, *26*, 61–68. doi: 10.1162/014892602320991383

Nash, C. (2014). Manhattan: End-User Programming for Music. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 221–226). London, United Kingdom: Goldsmiths, University of London. doi: 10.5281/zenodo.1178891

Omar, C., Moon, D., Blinn, A., Voysey, I., Collins, N., & Chugh, R. (2021). Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (p. 511–525). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3453483.3454059

Reich, S. (1965). Music as a gradual process. *Writings on music*, *2000*, 34–36.

Large Language Models and the Psychology of Programming

Clayton Lewis
University of Colorado Boulder
clayton.lewis@colorado.edu

Abstract

Large Language Models have emerged rapidly as powerful coding tools, in some cases showing the ability to create entire working programs, and (more commonly) providing help with the details of a great many APIs and frameworks. This emergence raises a number of questions for the PPIG community. Will these tools change what programmers do, in ways that affect "the psychology of programming"? Given the (apparent) command that these systems have of natural language, and of the semantics of a great many domains of activity, can they be used to enhance the kinds of interactions with software tools that PPIG researchers have studied? Noting that large language models exhibit analogical reasoning as an emergent capability, can they leverage insights from early research on programming by analogy? Does predictive modelling, as a key cognitive process that can be applied in many domains, suggest new ways to think about programming not based on text?

1. A new approach to programming

1.1. Domains and mapping

> [P]rogramming is often defined as a process of transforming a mental plan that is in familiar terms into one that is compatible with the computer ... many [bugs] and difficulties arise because the distance between these is too large. This concept is called closeness of mapping by Green & Petre ... The closer the programming world is to the problem world, the easier the problem-solving ought to be. Conventional textual languages are a long way from that goal.
>
> There are also many ambiguities in natural language that are resolved by humans through shared context and cooperative conversation .... Novices attempt to enter into a human-like discourse with the computer, but programming languages systematically violate human conversational maxims because the computer cannot infer from context or enter into a clarification dialog. (Pane, Myers, et al., 2001)

As programmers we have been used to mapping entities and structures in a *problem domain* into a *representational domain* that includes things like variables, lists, statements, and so on. The psychology of programming has been the psychology of this mapping process. Aspects of it have included how to learn about and understand the representational domain, how to form the mapping itself, and how to evaluate the effectiveness of the mapping, for example in debugging.

The mapping is multifaceted, with concerns in different facets often being to some extent separable. For example, we may separate the development of a "high level" view of a program, from the details of elaborating this view as a syntactically correct expression in a given programming language.

2. Large Language Models (LLMs) change this picture dramatically.

Figures 1, 2, and 3 show screenshots of programs created using the LLM GPT-4. Each figure also shows the initial input provided to GPT-4. Hereafter I will refer to the system simply as GPT.

The example in Figure 1 is based on an example developed originally in Boxer (diSessa & Abelson, 1986; Roschelle & Mason, 1995), a system created to support the development of a deep computational literacy in schools. As shown in the figure, the initial prompt for that example was:

I'd like the code for a Web page that shows a 3 x 3 grid of H's and T's, Each row and column should have a button that flips all the H's in the row or column to T's, and vice versa.

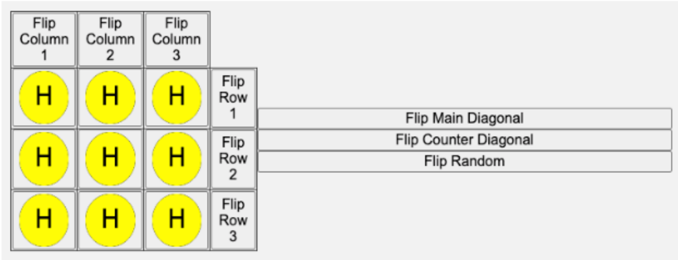


Figure 1 – Initial prompt– I'd like the code for a Web page that shows a 3 x 3 grid of H's and T's, Each row and column should have a button that flips all the H's in the row or column to T's, and vice versa.

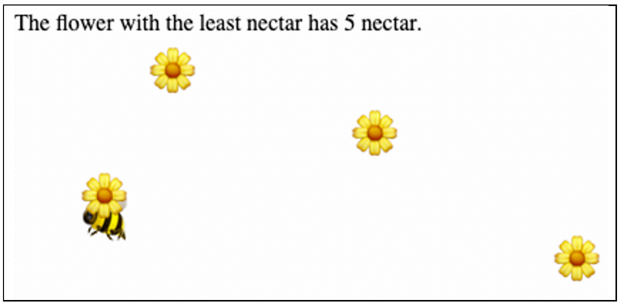


Figure 2 – Initial prompt– Make a Web page that shows a bee and several flowers. Each flower has a certain amount on nectar in it. Make the bee visit all the flowers, looking at how much nectar they have, and show which flower has the least nectar.

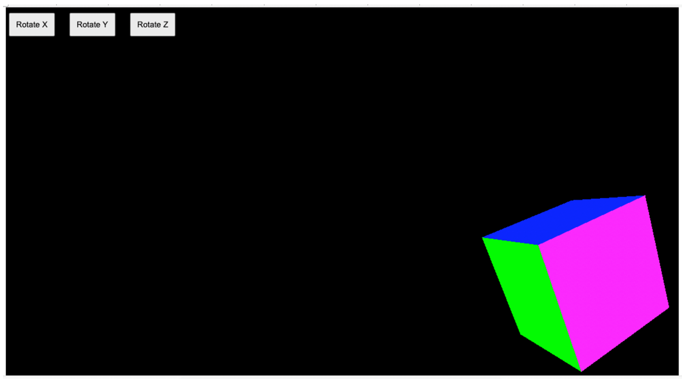


Figure 3 – Initial prompt– Can you give me a Web page, all in one file, that draws a wire frame cube, and has buttons to rotate the cube around the x,y, and z axes?

The program shown in Figure 1 was developed in stages, with the following additional inputs.

```
Can you put that all in one file for me? [The system had initially provided
    separate HTML, CSS, and Javascript code files.]

Actually, I do want those buttons for the rows and columns. Can you put those in?
    [The system had produced code for these buttons, but not the buttons
    themselves, and said that it had left the buttons out (no idea why, as it had
    no problem adding them).]

Can you add buttons to flip the two diagonals?
```

```
Can I have a button that will flip all the coins randomly?

Now, can you replace the H's and T's with pictures of coins?

That works fine, but I'd rather not have to use those img files. Can you change
    the program so that it draws the coins itself? [The system had explained where
     to use img files with the program, and that had worked fine, but finding
    suitable images had required some work.]
```

The versions of the program created at each stage worked correctly.

Examining the inputs that produced these results, we can see that the user (me) had no need to know anything about the syntax or semantics of HTML, CSS, or Javascript, or even to look at the code. I did need to know how to run the Web page from my browser, though had I not known that, I could have asked the system:

```
User:
If I have a Web page in my computer, how can I run it?

GPT:
To view a web page that is stored locally on your computer, you just need a web
    browser such as Google Chrome, Firefox, or Safari. Here's how you can do it:

Find your HTML file: This file should have a .html or .htm extension. This is the
    file that your browser can interpret and display as a web page.

Double-click the HTML file: In most operating systems, double-clicking the HTML
    file will automatically open it in your default web browser.

If double-clicking doesn't work for some reason, or if you want to open the file
    in a different web browser, you can do so manually:
```

Similarly, I knew enough about directories to be able to put the img files in the right place, when using those, but I could have asked the system about that, too.

The example in Figure 2 is based on the tutorial for John Pane's Hands system (Pane & Myers, 2006), which was designed to support natural programming. Pane and collaborators did a great deal of research to explore what features of a programming language would make it easy for nonprogrammers to use it.

In the example, a bee and some flowers are shown on the screen, with each flower having some amount of nectar. The bee visits each flower, and then displays the minimum amount of nectar found in any flower.

This example was more difficult for GPT than the coins example. The initial input,

```
Make a Web page that shows a bee and several flowers. Each flower has a certain
    amount of nectar in it. Make the bee visit all the flowers, looking at how
    much nectar they have, and show which flower has the least nectar.
```

produced a page with the flowers lined up in a column. Later versions corrected that, but failed to show the bee moving. It took a total of six inputs to get the working version. It's important to note, though, that none of the inputs required any knowledge of the code, or potential problems in it. Rather, typical inputs were

```
The flowers and text are now fine, but the bee does not seem to be moving to the
    flowers.
```

or

```
I'm still not seeing the bee move, unfortunately!
```

These inputs, and all the others, are framed in the problem domain, not the representation domain.

By contrast, the original Hands tutorial (Pane & Myers, 2006) on which the example is based, includes more than ten pages of conceptual material that users must understand, for example this:

```
The front of the card has spaces on it; this is where we can add information. Each
     row on the card is a property. It has a pair of boxes to hold information:
    the boxes on the left are called name boxes; the boxes on the right are called
     the value boxes. Soon you will see what these names and values are for.

Handy gives each new card three properties to start with. We can see the names of
    these properties by looking at the name boxes. Can you see the names of the
    three properties that we are starting with? The properties are named: cardname
    , x, and y. Now let's find out what these properties mean.

Look at this card's first property, cardname. The value box for the cardname
    property has the name of the card, Card-1. Since we haven't named our flower
    yet, Handy has given it this name to start with. Handy also shows the card's
    name at the very top of the card. Do you see it?
```

Besides cards and properties, the tutorial introduces lists, events, and rules, and the admonition, in bold text, that Handy will only understand rules written a certain way.

In its historical context, of course, Hands represented a notable advance. Hands is quite easy to understand, compared to

```
languages = ['Swift', 'Python', 'Go', 'JavaScript']
# run a loop for each item of the list
for language in languages:
   print(language)
```

or

```
   char c;
   bool isLowercaseVowel, isUppercaseVowel;
   cout << "Enter an alphabet: ";
   cin >> c;
   isLowercaseVowel = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
   isUppercaseVowel = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');
   if (!isalpha(c))
     printf("Error! Non-alphabetic character.");
   else if (isLowercaseVowel || isUppercaseVowel)
      cout << c << " is a vowel.";
   else
      cout << c << " is a consonant.";
```

let alone

```
global  _start

        section .text
_start: mov    rax, 1         ; system call for write
        mov    rdi, 1         ; file handle 1 is stdout
        mov    rsi, message ; address of string to output
        mov    rdx, 13           ; number of bytes
        syscall
```

The challenge and opportunity of LLMs is that people can now create programs without understanding the representational domain of computational concepts at all. No syntax, no semantics of computational entities, not even any higher level view of the program. In short, many of the the traditional concerns of the psychology of programming, as we have conceived it, or at least what we have seen as the central concerns, simply vanish, for tasks like these.

3. Analogy and programming

The concept of analogy appears often in the literature of the psychology of programming. Often it arises as an aspect of how people understand programs and programming, when an analogy is used to connect something in the domain of computation to something familiar. For example, Ahadi, Lister, and Teague (2014) report the use of "box" as an aid to understanding variables (and they note the misleading suggestion that, like boxes, variables ordinarily can contain multiple values.)

(Is this use of the "box" concept an analogy, or a metaphor? it could be either, and I won't take up the difference here. The more an explanation is offered of the relationship between two things, the more the relationship is an analogy (A. F. Blackwell, 1996).

Do such relationships play any role in how LLMs create programs? Superficially, it might appear so. GPT is certainly able to explain the box analogy, if asked to do so; see Box 1. Note that it mentions the difficulty with the box model that Ahadi et al. reported. It can also *propose* an analogy to explain variables, if asked to do so. However, in one interaction, it suggested different vegetables in a recipe, and the possibility of replacing one vegetable by another, as useful things to think about, in understanding variables. That seems quite odd, and indeed GPT itself suggests that a container might be better, at the end of the vegetable response.

Box 1

```
User: is a variable in a programming language like a box?
GPT: Yes, that's one way to understand it!

A variable in a programming language can be thought of as a box where you can
    store a value. Just like you might label a box with what's inside (like "
    holiday decorations"), variables are given names to indicate what data they're
     storing.

When you create a variable in a programming language, you're essentially telling
    the computer: "Hey, I need a box where I can keep this specific piece of
    information." And when you assign a value to that variable, you're putting
    something into that box.

For example, you might have a variable called age and assign it the value 25. In
    this case, age is the label on the box, and 25 is what's inside the box.

Later on, when you use that variable in your code, the computer will go to the "
    box" and see what's inside. If you change what's inside the box - that is, if
    you change the value of the variable - the computer will use the new value
    next time it checks.

It's worth noting that this box metaphor has its limitations. For example, unlike
    physical boxes, a variable can only hold one value at a time; assigning a new
    value to a variable replaces the old value. Moreover, different types of
    variables (like integers, strings, or arrays) can hold different types of data
    , whereas a box could hold a mix of different items.

And in languages that include reference types (like objects in JavaScript or
    classes in Python), the variable might hold a reference to the value rather
    than the value itself - more like a note saying where the box is, rather than
    the box itself. But despite these limitations, the box metaphor is a helpful
    way to start understanding what a variable is and how it works.
```

More generally, analogical reasoning is a strength of LLMs. Webb, Holyoak, and Lu (2023) show that GPT-3 performs well on a wide range of tasks used to assess or study analogical reasoning in humans.

Despite these facts, it's doubtful that connections between computational structures and things like boxes are involved when LLMs create programs. Rather, connections between programs and problem statements seem to be important.

An example reported by Lewis (2021) can illustrate this. An LLM has likely encountered in its corpus many programs that involve looking things up in collections, and producing an appropriate message, if an item is not found. But it is unlikely that it has ever seen a program that keeps a list of gifts that someone has given, and the recipients, as was called for in the example. Yet in the program the LLM created, the message "No gifts found for Fred" was used, if Fred was the recipient in the search. By some means, the LLM has adapted the messages seen in its corpus of example code to the particular task at hand.

That description, with its use of the word "adapted", should be taken only as saying *what* the LLM has done, and not *how* it has done it. The nature of its training, and its internal operation, mean that the LLM rarely has access to any stored representation of the examples it has seen, that it could "adapt". Rather, the LLM has created a model that relates requests for code to code. This model produces an appropriate message, as well as the rest of the code that the prompt calls for.

The examples presented earlier illustrate this same process. The LLM's model is able to map a wide range of requests for code to code that is responsive to the requests.

Past efforts at automatic or technically aided programming have sometimes focussed on analogical relationships between programs. Brand's Programming without Code proposal, at PPIG 2016 (Brand, 2016), built on Anderson and Thompson's PUPS model (Anderson & Thompson, 1989), in which analogical transformations of programs are supported by *annotations*.

Consider this code for factorial:

```
factorial(x){
      if (x==0)
            return 1;
      return x*factorial(x-1)
}
```

The analogy summorial : factorial :: + : * can be solved by replacing * by + in the code, after annotating the 1 in the base case by "identity for *" and noting that "identity for +" is 0.

Dershowitz (Dershowitz, 1986) shows how a whole family of rather diverse programs can be developed, by implementing analogical relations between members of the family. Starting with a program that uses binary search over an interval to approximate the quotient of two numbers, Dershowitz derives a program that uses binary search to approximate the cube root, and then one that finds an element in an ordered list. Each step of the process can be framed as solving an analogy: what program is to quotient, as this given program is to cube root?

GPT can reproduce a progression of programs of this kind, in response to these requests:

```
Here is some code for finding the quotient of two reals. Please modify it so it
    computes the cube root of a number, instead.
```

and

```
Based on that, can you give me a program that takes an array of numbers, in order,
     and a number that occurs in the array, and returns the index of that number
    in the array?
```

The main difficulty in this is that GPT does not think that binary search is a good way to compute quotients, so getting the starting program takes a little persistence. Later on, on the list problem, it strayed from the form of the example, but readily responded appropriately when asked

```
That works, but can you make the code closer to the example I gave?
```

While GPT can produce a progression of programs that share a common algorithmic theme, as Dershowitz envisioned, its operation is quite different from that described in Dershowitz's account. Dershowitz's code is heavy with annotations, like the following:

$$achieve\ |a^{\frac{1}{3}} - r| < e, varying\ r$$

or

$$assert\ A[z+s+1] \leq b < A[z+1]$$

These express the effect or purpose of the code, and developing a new version of a program, to serve a modified purpose, calls for close reasoning about how the annotations need to be transformed, when the problem is changed.

Notably, GPT needs none of this apparatus. Its model is able respond directly to a modified program requirement, accompanied by code to be adapted. to create the desired transformed program.

Arguably Dershowitz's annotations, while not needed by GPT to transform the code, could be useful in avoiding errors. When done correctly, reasoning about the annotations, and the accompanying code, could detect possible coding errors. How well GPT might be able to reason about program annotations is a subject for further investigation. Box 2 shows a very simple example.

Box 2

```
User: evaluate this pseudocode:
x=3
y=4
x=y
y=x
assert x=4
assert y=3

GPT: The pseudocode you've provided appears to have a mistake. Here is the line-by
    -line explanation of the pseudocode:

x is set to 3.
y is set to 4.
x is set to y, which means x is now 4.
y is set to x, which means y is now 4 as well, not 3.
So, both x and y are equal to 4, not 3. Therefore, the final two lines of the
    pseudocode, where you assert that x is 4 and y is 3, will fail for y because y
     is not 3, it's 4.

If you intend to swap the values of x and y, you need to use a temporary variable,
     as shown below... In this corrected version, x is indeed 4 and y is indeed 3,
     so the assertions will pass.
```

4. Conversation

A progressive strand in work at PPIG (and elsewhere (Van Brummelen, Weng, Lin, & Yeo, 2020)) is the exploration of conversational interactions with software tools. Church, Söderberg, and McCabe (2021) review many ways in which human conversation supports effective communication, and envision ways in which these benefits might be secured in (for example) exploring errors in code found by a compiler.

That GPT supports such interaction has already been suggested in the examples above. We've seen that the user can rely on GPT to maintain the context needed to interpret an input like "Can you add buttons to flip the two diagonals," in which "add to what?" and "what two diagonals" need not be spelled out.

Similarly, the input "I'm still not seeing the bee move, unfortunately!" makes no reference to the code at all, but is correctly interpreted as a request to examine the code, and find and fix in it whatever is responsible for the indicated shortcoming.

Box 3

```
User: Can you give me a Web page, all in one file, that draws a wire frame cube,
    and has buttons to rotate the cube around the x,y, and z axes?

GPT: Sure! You can use HTML to structure the page, CSS for styling, and JavaScript
     with the Three.js library to generate and control the 3D cube. Here's a basic
     example of how this could work:

Please note that JavaScript is disabled in this text interface, so you will need
    to copy and paste the code into an .html file and run it in your browser to
    see it in action.

[code given here]

In this example, we first include the Three.js library with a script tag. Then we
    set up a basic scene with a camera and a renderer. We also create a cube with
    a wireframe material and add it to the scene. Then we set up an animation loop
     that will render the scene.

We also set up three buttons that, when clicked, will rotate the cube around the X
    , Y, or Z axis respectively.

Finally, we set up an event listener that will resize the canvas if the window
    size changes.

This is a very basic example. Depending on your needs, you may want to add more
    functionality, such as different types of controls or other 3D objects.

User (after viewing the page): That's pretty good, but the axes change when I
    rotate the cube. I'd like the axes to be fixed when the cube rotates.

GPT: If you want the rotation buttons to always rotate the cube relative to the
    original axes, not the current orientation of the cube, then you would need to
     change how the rotations are performed.

Here is a modification of the previous script that implements this. It works by
    creating three different rotation matrices for the X, Y, and Z rotations, and
    then applying these to the cube's position:

[code given here]

In this version of the script, when a button is clicked, the corresponding
    rotation matrix is applied to the cube's transformation matrix, causing it to
    rotate relative to the world axes, rather than its own axes. This gives the
    effect of the axes remaining fixed as the cube rotates.

User: Could you color the faces of the cube with different colors?

GPT: Three.js does not natively support different colors for each face of a
    BoxGeometry directly in one Mesh. But you can create six different Meshes each
     with a PlaneGeometry, color them differently, and group them together to form
     a cube.
Here's an example of how you can achieve this:

[code given here]

This script creates six planes, colors them differently, positions and rotates
    them to form a cube, and then groups them together using a THREE.Group object.
     Then it applies the rotations to the group, so all the planes will rotate
    together.

[code works as desired.]
```

Box 3 shows the example for Figure 3, including not only the turns taken by the user, but also the responses from the system. One can see that GPT has no trouble placing the user's inputs in the right context, and understanding them, including the somewhat tricky matter of how the axes should be handled. One can also see that GPT assumes that the user wants to know quite a bit about how the code works. In a tutorial interaction, as has been common in this kind of situation, that assumption is reasonable. In fact, though, since I no longer need to understand the computational representation, I actually did not want these explanations! (I likely could have asked GPT to omit this material, but I haven't tried that. It's not impossible that suppressing the explanations would actually damage GPT's performance, since its outputs form part of the input when processing later requests.)

Another interaction, not presented here in detail, illustrates some additional conversational capabilities. I wanted to create a client for accessing the GPT models via their API. Starting with the input

```
I need a Web page that will allow me to get continuations from the OpenAI davinci
    model. Can you give me the page, and tell me how to set it up?
```

I obtained a working client (actually, a server-client pair). GPT handled some intermediate requests, like "Can you tell me how to set that up using node.js?" and "Great. Can you tell me more about how to get my API key?" Its explanations were adequate for me, whose knowledge of UNIX-style commands and practices is sketchy at best, to do what was needed.

After completing the setup as directed, there was trouble. GPT responded appropriately to the inputs:

```
After starting the server I get the message 'Cannot GET' at localhost:3000

That seems to be working, but I just see [object Object] on the page, instead of
    the completion,

that's giving me 'Cannot read properties of undefined (reading '0')'

now I am getting 'AxiosError: Request failed with status code 404' and no console
    output.
```

It provided guidance that I could act on, except in the last case, when it listed seven things to check, most of which I felt unable to pursue in an effective way.

That was the only situation, among all those discussed here, in which I looked at the code, other than to insert code snippets where GPT directed. I noticed that whereas I had asked for a client for a model called "davinci", the code mentioned "davinci-codex". That superficial difference triggered my next input:

```
User: Is it possible that "https://api.openai.com/v1/engines/davinci-codex/
    completions" in my server.js should have just "davinci" rather than "davinci-
    codex"?
GPT: Yes, it's definitely possible. ...
So, to answer your question, if you want to use the Davinci model, you should
    indeed change "davinci-codex" to "davinci" in your URL.
```

After dealing with one more error, for which GPT's guidance was easy to follow, the application worked as desired. Even including the one case of examining the code, just mentioned, GPT allowed me to accomplish this with no real understanding of the code concerned.

In the process, GPT fielded a mix of inputs, including ones that called for more explanation of what to do, so as to support a novice user, and ones that reported errors that needed to be corrected. The inputs could be framed in English in a natural way, with no attention to what GPT would understand or not understand, just as would be true with a knowledgeable human conversational partner. There was no need to mark the difference between inputs of different kinds.

GPT's guidance was usually provided in a clear and actionable way, except for the list of issues to check, mentioned near the end of the interaction. Code changes were described by providing new code, in complete form, together with a clear indication of where in the program the new code needed to be placed.

In a few cases, especially near the start of the interaction, GPT's guidance assumed knowledge I did not have. But requests for more guidance were simple to pose, and were responded to effectively.

5. Understanding the problem domain as well as the computational domain

In mapping problem statements or requests to code, GPT can draw not only on its knowledge of programming, but also on its knowledge of many aspects of the world and human affairs. Here the word "knowledge" perhaps should carry scare quotes, since the relationships among what GPT "knows" how it can be said to "know" it, and human knowledge and knowing, are unclear. But the role of this "knowledge" in the following examples will be apparent.

The first example begins with the following request:

```
User: I need a Web page that lets people enter reports of vandalism in their
    neighborhood, and sends an email with their report to vandalism@gmail.com.
```

This would just be silly as an input to a conventional programming tool. A compiler or interactive development environment can't be expected to know anything at all about vandalism. GPT provides code for a page with fields for name, address, and description. This further request

```
User: That's ok, but I thought maybe you could suggest (for example) some common
    kinds of incident, to make it easier for someone to make a report.
```

leads to a form with a drop down menu for five categories of incident, graffiti, property damage, littering, illegal dumping, and other, with working Javascript code that implements this simple system. In its responses one can see that GPT "knows" quite a bit about vandalism, and what would be involved in reporting it.

This is a very new situation in the realm of programming. Commonly people who understand programming don't understand a target domain in which they may need to work, and people who understand the target domain don't understand programming. Indeed, Curtis, Krasner, and Iscoe (1988) cited "the thin spread of application domain knowledge" among software developers as a leading cause of system development failures.

In a study of software design organized by Petre and Van Der Hoek (2013), skilled, professional designers were challenged to develop a system for simulating traffic flow, in a situation that included traffic signals with traffic sensors. None of the designers knew about, or sought out, representations that have been developed in this domain for problems of this kind, nor about common problems that arise, such as the *yellow trap*. The "yellow trap" is a specific issue in traffic engineering where a driver is "trapped" in the dilemma zone during the yellow phase of a signal. This usually occurs in a situation where left-turning drivers mistakenly believe that oncoming traffic is also facing a yellow light, which can lead to collisions if the oncoming traffic actually has a green signal. But GPT does know about the yellow trap; indeed the definition just given was provided by it.

Further, GPT knows about useful representations for such problems, such as

```
Signal Sequence Charts: This is a more detailed type of timing diagram that not
    only shows when each signal is red, yellow, or green, but also when each
    vehicle movement (straight, left, right) is permitted. This would clearly show
     that left turns are only allowed during the protected left turn phase.
```

That description is part of GPT's response to the request "I'm worried about 'yellow trap'. what representation might help with that?"

But can GPT coordinate this problem domain knowledge with its knowledge of the domain of computation, as it was able to do in the much simpler vandalism example? The answer is a qualified yes. The request

```
Could you give me the HTML and js for a Web page that would allow me to simulate a
     yellow trap, and a control scheme that avoids it?",
```

led eventually to a working simulator, shown in Figure 4.

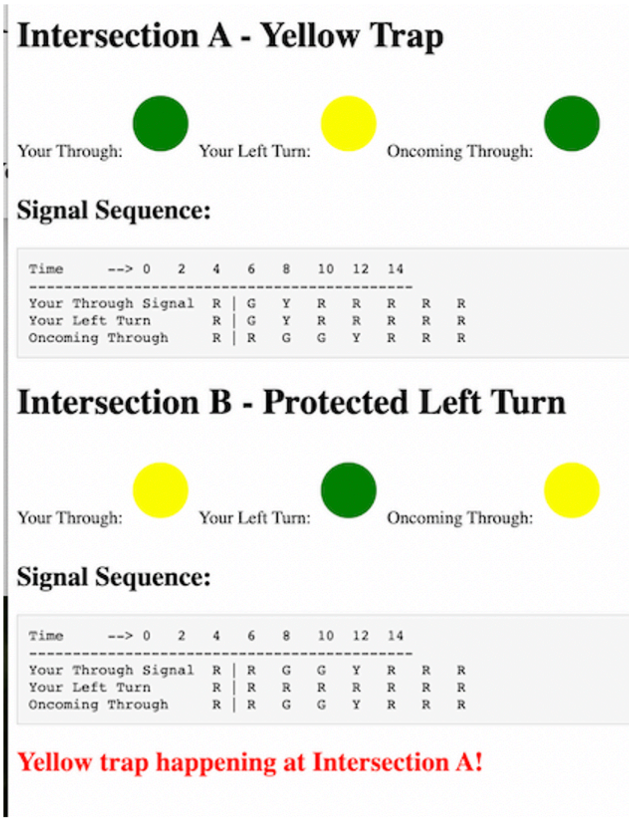


Figure 4 – Traffic signal simulator

This page is animated, with the traffic signals changing color so as to show that yellow trap occurs for Intersection A, with the "Your left turn" signal being yellow while the "Oncoming Through" signal is green. That does not happen in Intersection B.

The qualification of the positive assessment stems from two issues. First, the interaction required 12 follow-up requests to produce the result shown. Some of the follow-ups were refinements, such as slowing the simulation to make it easier to follow, or giving a specific alert on the screen when a yellow trap occurs. But seven of the follow-ups were needed to report that the simulation was not showing yellow traps at all, that is, that the code did not satisfy the requirements. GPT was successful in correcting the problems, with no guidance about the code from the user.

However, the value of GPT's problem domain knowledge is undercut by the occurrence of these failures. While one might hope that working with GPT would mean that a user could learn about yellow traps from the simulation that they asked GPT to create, the user needed to know what a yellow trap is, and to recognize that it was not happening, in order for GPT to create a working simulator. GPT's "knowledge" on its own was not sufficient.

Two of the follow-on requests elaborated on another aspect of the original request, that the system should simulate a "control scheme". Specifically, a request was issued that the system should use and show a "signal sequence chart", one of the representations it had itself suggested as appropriate for understanding the yellow trap.

Unfortunately, the representation it uses, as shown in the figure, is not actually a "signal control chart", as used by practitioners. Indeed, the representation in the figure does not fit the description GPT itself provided for these charts, quoted earlier. Such a chart "not only shows when each signal is red, yellow, or green, but also when each vehicle movement (straight, left, right) is permitted." The representation in the figure does not show that. Further, the representation does not reflect what the simulation actually does.

These limitations aside, we can see that GPT indeed "knows" a good deal about traffic signals, and is able to create code that usefully represents much of that "knowledge". For example, it "knows" that traffic signals have green, yellow, and red aspects, that these aspects cycle in the indicated order, and that there are signals for traffic in two directions, in situations where a yellow trap can occur, that there can be signals for left turns, and so on.

The examples in the sections above show that GPT offers ways to program that are quite different from familiar programming tools and environments. Users can describe their intentions, without being concerned with the code that will implement them. But the expression of intentions must be text.

There are many visual programming systems that use diagrammatic representations, rather than text. These are much discussed at PPIG and elsewhere (see among many, Good and Brna (1999); Rosian (2022)). But these provide different ways to express intentions in the *computational domain*, such as rules or data paths, not the *problem domain*. LLMs, or similar systems, might support different modes of expression.

6. Does the new approach to programming offer new opportunities for programming not based on text?

While GPT is trained on a corpus of text, related developments work with non textual material, such as movements or images (Brohan et al., 2022; Khan et al., 2022). Do these suggest that there may be ways to create programs that do not require textual expression of any kind?

It might seem that *no-code systems*, like Google Appsheet, already offer this. Such systems replace coding by selections in a family of dialog boxes (Appsheet requires writing some expressions, too.) These systems don't use textual expression, and they allow intentions to be expressed without concern for code (though as in visual programming, one's intentions have to be translated from the problem domain into a computational domain, in the case of Appsheet a domain of tables and views.) Unfortunately these systems are full of hidden dependencies, and hidden information of other kinds. The consequences of one's selections are invisible, and only become manifest when the application one is building runs or fails to run.

Could the capabilities shown in systems like GPT help? No-code systems seem weak as the basis for models like GPT, even if a corpus of actions (button clicks, form filling in the no-code tool) and effects (Web pages and interactions with them) could be gathered. The same hidden effects that make no-code systems challenging for human users might present problems for training systems like GPT, too. But this could be an interesting problem to explore. Very preliminary attempts, using manual coding of commands and effects, as used in Lewis (Lewis, 1988), show that GPT can learn to interpret commands from a few examples. These actions and effects are far simpler than those seen in Appsheet, though.

6.1. Multimodal systems based on predictive modeling might support new kinds of nontextual programming

The core of an LLM is *predictive modeling*: given a sequence of tokens in the corpus, what comes next? A complex neural network is trained to make accurate predictions in that setting. The resulting model can be used to make predictions that continue any sequence of tokens, including sequences that occur nowhere in the corpus. Remarkably, this predictive model, adapted by a limited amount of "fine tuning" using examples of human interaction, is able to perform the tasks we've been discussing.

In principle, the same approach could be used to make predictions for corpora of other kinds, for example corpora of gestures. As mentioned, there are systems that operate over domains of this general kind, like robot movements (Brohan et al., 2022). Perhaps more usefully for present purposes, corpora might mix gestures and code. Humans operate habitually in a mixed domain of this kind, that includes gestures and speech (Kendon, 1994). In ways we are in general unaware of, gestures help our words convey meaning.

Conjecturally, we function in this mixed domain by identifying *regularities*, that is, patterns with predictive value, that contain elements of both kinds, so that a sequence of words can predict a gesture, or a sequence of words and gestures can predict a word, and so on. Further, it appears that the learning process results in the establishment of analogies between regularities. For example, a relationship between a sequence of gestures and a sequence of words could be extended to a relationship between those gestures and a different, but related, sequence of words.

Analogies between regularities can be identified with some kinds of (often implicit) concepts. For example, the concept of Ohm's p-prim (phenomenological primitive), identified by DiSessa (1993), expresses the analogy between

```
more voltage and same resistance gives more current
```

and

```
more push and same friction gives more motion
```

and many other situations, in which magnitudes of various kinds are related as those in the examples are. The concept applies when there is an input of some kind, and an output, promoted by the input, and some intervening factor that modulates the effect of the input. Given this very general concept, a gesture that might accompany the assertion about electricity could also be used to help express the assertion about push and friction. Could similar relationships, between gestures and code, or between gestures and program behavior, be useful in programming?

The question is bafflingly broad. A narrower question, perhaps more approachable, is, could these relationships make programming more like sketching? A joint meeting of PPIG and the Art Workers Guild (London, September 5-7, 2018; see report at `https://ppigattheartworkersguild2018.wordpress.com/`), included testimonies from Guild members that cast programming, as repellent, in the literal sense, an activity that people would like to avoid, even if it is useful. "Sketching" in the question stands for another kind of activity, lacking these repellent qualities, and having the attractive qualities of enjoyable expression. As Charlie Gere asked at that meeting, can programming be like sketching?

Lewis (2019) presented some fragmentary suggestions about this, in which lines and ribbons have defined behaviors, that direct the movement of dynamic structures called vines. Figure 4 shows an example of a sketch that forms palindromes. Could a predictive model do anything to make this programming paradigm more useful, while retaining its sketch-like character?

This does seem barely possible. An idea that might be explored posits a model trained on encodings of the drawing actions that create the sketches, along with discourse about the resulting programs, and their behaviors. Could this model supply interpretations of drawing actions not included in the training corpus, in the same way GPT can continue sequences of tokens that do not occur in its corpus? These interpretations could then be supplied to an LLM, that would produce conventional code that implements them.

The chain of activities would be:

1. user enacts drawing actions
2. predictive model trained on actions and discourse, based on example sketches, generates discourse appropriate to the drawing
3. LLM generates code from the discourse

Given such a system, it might further be possible to intermix discourse about the intended purpose of a drawing, with the drawing actions. Doing this could make it more likely that the discourse generated in

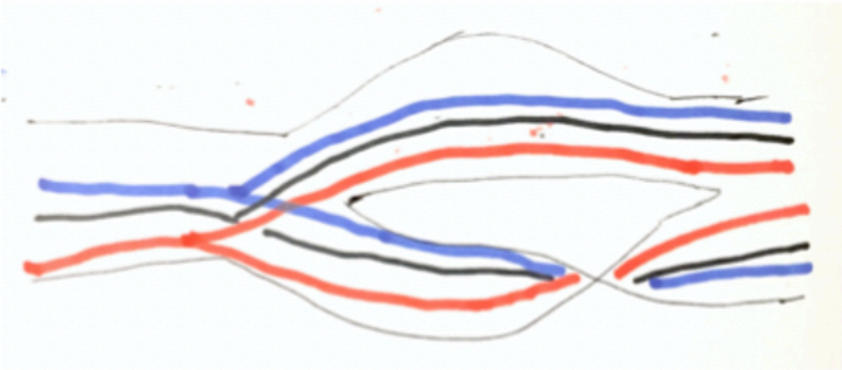


Figure 5 – A sketch that creates palindromes.

Step 2 would be more appropriate to the drawing, and its creator's intent. Thus the code generated in Step 3 could be more appropriate.

Such a system might address a key limitation of the proposal in Lewis (2019), that there is a fixed repertoire of drawing actions, with conventional interpretations. Not only does this impose definite constraints on what can be expressed, but also it imposes a substantial burden on the user, who must learn these actions and their interpretation. In the speculative system, users could draw whatever they like, and hope to get a satisfying effect, at least some of the time.

The system would support a wide range of sketch-likeness. At one extreme, users could draw freehand, and be satisfied with the resulting behavior. As they devoted more time to looking at example sketches and their effect, and as they said more about their intentions, the system would become less like sketching, and somewhat more like programming as we have known it. Even in this less sketch-like usage, however, users would not be concerned with the bit, bytes, lists, and variables that contribute at least some of the repulsion that the Art Workers feel.

7. What is ahead for the psychology of programming?

7.1. Artificial and natural semantics

What is different about using a predictive model like GPT to create programs, from programming as we have known it? Summarizing, users can express what they want in the problem domain, not the computational domain; they can describe their intentions iteratively, in conversation; the tool "knows" a good deal about many problem domains.

The conversational interaction that ties these differences together is supported by semantics for what users say that is quite different from the semantics of programming languages. The key feature of *artificial semantics*, as seen in programming languages, is that the meaning of expressions is determined by the expression, by applying specified rules of interpretation. In *natural semantics*, as seen in human language, and in interactions with LLMs, the form of an expression contributes something to the meaning assigned to it, but the influence of the form is often dominated by a complex context.

In artificial semantics, an expression means what the rules of the system say it means. The rules of Fortran, for example, say what a Fortran statement means.

In natural semantics, as Garfinkel (1969) observed, an utterance means what it has to mean, in order for it to have meaning, in the context in which it is used. For example, if I were to say, in September, 2023, "I really enjoyed giving my talk at PPIG in Stockholm," "Stockholm" means Lund. It would be understood to mean that by any interlocutor who knows the context.

The character of natural semantics isn't expressed only when repair is needed. The examples presented earlier, like human discourse, show many cases in which expressions aren't wrong, but can only be interpreted in the light of context. For example, as we saw, in "Can you add buttons to flip the two diagonals," the questions "add to what?" and "what two diagonals" did not have to be answered. They are taken to mean what they need to mean for the request to make sense, in its context.

Historically, efforts at "natural language programming" like REL English (Henisz-Dostert & Thompson, 1969), or natural programming, like Hands, have stayed on the artificial side of the boundary. While REL looked like English, its interpretation was governed by rules. Similarly, the extended Hands tutorial was needed to enable users to know enough about how Hands expressions are interpreted to accomplish their goals.

The ribbons and vines system in Lewis (Lewis, 2019) also uses artificial semantics. As discussed, users would have to learn the acceptable drawing forms and their conventional interpretation. The speculative extended drawing system uses natural semantics to assign meaning to drawings, even novel ones.

The ability to support natural semantics in an LLM like GPT comes from the nature of its predictive model, and how it is trained. The model can exploit any regularities in its corpus, that is, any patterns that have predictive value, and use these in making predictions in new situations. Because the training corpus intermingles discourse about all kinds of things, as well as a wide variety of code, predictive regularities that link problem domain material to code can be found, and are used.

One of the first interactions with an LLM that captured my attention illustrates this blending of considerations. I wanted to include descriptive text on a Web page much like that shown in Figure 1 (see Lewis (2022)). But the Codex LLM did not add the text, that was provided to it, until I changed the title of the requested Web page from "nums" to "nums with legend". In a system with artificial semantics, it just doesn't matter what title is assigned to something. But in natural semantics, it *does* matter. Indeed, it's the role of titles in the natural semantics of humans that explains why titles actually exist!

Further perspective on the distinction between natural and artificial semantics comes from reflecting on the intentional programming project of Simonyi, Christerson, and Clifford (2006). Recognizing the challenge of coordinating knowledge of problem domains with knowledge of coding, they say:

> For the creation of any software, two kinds of contributions need to be combined even though they are not at all similar: those of the domain providing the problem statement and those of software engineering providing the implementation. They need to be woven together to form the program.

The problem statements would be expressed in domain-specific languages:

> Were the domain chess, for example, we would need a def for the pieces, the colors, the board and its squares, and the various states: initial, check, checkmate, draw. It is not required that we define the game in the schema - presumably that will be a part of the program that we create - but we need to have the vocabulary complete enough to define the game.

Creating such languages would be a colossal enterprise. But the problems go well beyond the effort involved. Phil Agre observes that deployment of information technology in work settings imposes, implicitly or explicitly, a grammar of action:

> Grammars of action frequently oversimplify the activities they are intended to represent, if only because the people who articulate the grammars are only superficially acquainted with its actual complexities and the actual social forces that determine its form .... The ontology may fail to make enough distinctions, or else whole subcategories of "invisible" activity might go unrepresented. The grammar might impose overly restrictive ordering constraints

> on the unitary actions, it might neglect the interleaving of distinct forms of activity, or it might mistake prescribed procedures for an accurate descriptive account (or at least a practicable form) of the activity ... . As a result, the participants in the newly instrumented activity will find it necessary to evolve a system of "work-arounds" to keep things going. (Agre, 1994)

More broadly, and long ago, Harold Garfinkel argued that human activities, including communication, simply cannot be accounted for by the kinds of rules and structures that support artificial semantics.

> Garfinkel approaches the topic by stressing that understanding language is not to be regarded as a matter of 'cracking a code' which contains a set of pre-established descriptive terms combined, by the rules of grammar, to yield sentence meanings which express propositions about the world. ... An utterance is thus the starting point for a complicated process of interpretative inference rather than something which can be treated as self-subsistently intelligible. (Heritage, 2013, p 139)

We can suggest that the workings of LLMs are much closer to Garfinkel's idea of meaning than are the forms of artificial semantics seen in programming languages, including domain specific languages.

7.2. Cognitive dimensions and natural semantics

A very productive line of development in the psychology of programming has been cognitive dimensions analysis (Green, 1989; A. Blackwell & Green, 2003), more fully, analysis of the cognitive dimensions of notations. A cognitive dimension is an aspect of a notational system that is likely to cause problems for, or assist, a user who must create, interpret, or modify the notation needed in some situation. For example, *hidden dependency* describes a situation in which two or more aspects of a notation are linked, so that one aspect can't be changed without changing another, or can't be interpreted without understanding another, but the connection between the aspects is not apparent. One can anticipate that a hidden dependency will be a common source of errors and frustrations for users. More generally, *viscosity* names a complex of considerations that make it difficult to make changes in the notation for something. Even dependencies that are clearly apparent can create viscosity.

To the extent that computerized tools continue to use notations, cognitive dimensions analysis will still be important. But computerized tools that do not use representations, at least of the familiar sort, may now become widespread. (Not ubiquitous, though. Gould et al. (Gould, Lewis, & Becker, 1976) found that artificial notational systems were often preferred by users, even as a way to communicate with other people, for tasks like describing an assemblage of toy blocks.)

Compare "book me a room in Lund for August 21-25", and the ensuing conversational interaction, with the usual dialog process, in which one fills in forms and clicks buttons. In the usual dialog, there can certainly be a hidden dependency, or viscosity, if one can't see that information entered in one place has to match other information. For example, a very poorly designed booking system might ask for an email to be entered in two places, say in the user's account information, and then for a new booking, and be unable to determine which address to use, if these do not match.

Could something similar happen in a conversational interaction? Logically, these things could happen, but would be resolved in a natural way. That is, hidden dependencies cause problems because the system governed by artificial semantics usually can't work out what parts of an ill-formed expression to modify to make it meaningful. This is seen routinely when compilers find errors in source code. The programmer has to make the repair, and it is hard to do that if you can't see the dependency.

In a conversational interaction, the meaning of what is said is worked out as the conversation proceeds, and the system is responsible for resolving conflicts. In the booking example, a conversational system would detect the mismatch and ask which address to use.

Relatedly, viscosity happens when the programmer has to make needed changes in other parts of a program, when a given part is changed. In a system with natural semantics more adjustments can be made without user intervention.

On the other hand, breakdowns of the kind that are common in artificial semantics, when expressions can't be interpreted, can happen in natural semantics as well. The expression "the set of all sets that do not include themselves" is an example. But in natural semantics such problems are pushed out to the edges of common communication. Life can't proceed with discourse in which breakdowns are frequent. Ideas that lack expressions that can be interpreted as needed don't get talked about much.

Natural semantics works better the more comprehensive the system's semantic grasp is. As we've seen, an LLM has semantic grasp that includes aspects of a problem domain like vandalism, as well as of computational domains like HTML. This enables it to make many choices in code, based on a single user intention, either when an initial, high level request is made ("give me a Web page for reporting vandalism", or in response to a request for modification ("make it easier to file a report").

Viscosity in an artificial semantic system like a programming language is likely still relevant, even in a world of LLMs. The more complex the constraints in a language are, that is, the dependencies among parts of a program, the harder it may be for an LLM to obtain command of the relationships between user intentions and working code, when modeling a corpus of examples. But these difficulties are no longer the direct concern of users.

7.3. Psychology of Programming, Psychology of Natural Semantics

Since, until recently, programming has been limited to artificial semantics, the psychology of programming has been the psychology of artificial semantics. Key questions have been, are some forms of artificial semantics easier to work with than others? Are there psychological interpretations of the difficulties programmers face in working with artificial semantics, and can these suggest helpful interventions? How can people be assisted in learning a system of artificial semantics?

To the extent that some programming is no longer based on artificial semantics, these questions lose importance. In the short to medium term, it appears that the natural semantics of LLMs isn't up to the job of supporting the full range of programming, or software engineering, tasks, though the limits of their eventual or even immediate applicability remain quite unclear. For now, a lot of professional programming won't be eliminated.

But much work in the psychology of programming has focused on *end user programming*, including programming in school settings for which programming knowledge itself is not the focus. As natural semantics becomes more widely useful in programming, will the psychology of natural semantics become a central part of the psychology of programming? If so, there is a simple and natural direction such a possible shift can be accommodated: we'll study what actually happens when people try to use LLMs to create programs. No doubt there will be problems.

We can already tell what some of them are. LLMs produce code that looks correct, whether it is or not, and skill is often needed to tell that the code is not actually correct. Technical approaches are emerging that seem likely to help with this. In particular, it is possible to arrange for an LLM to generate its own test cases, and to arrange to have these run, and the results interpreted and responded to by the LLM.

Beyond identifying and responding to problems, we can turn more attention to two questions that we've not been able to give much focus to, as long as we needed to deal with the complex of problems created by artificial semantics. How should one decide what program, if any, to create, in a given situation? What intellectual value does knowing how to program have in itself, rather than as a means to an end?

Acknowledgements

I thank Tamer Amin, Antranig Basman, Alan Blackwell, Luke Church, Andrea diSessa, Jonathan Grudin, Katherine Hermann, Owen Lewis, and Jason White for many valuable conversations.

8. References

Agre, P. E. (1994). Surveillance and capture: Two models of privacy. *The information society*, *10*(2), 101–127.

Ahadi, A., Lister, R., & Teague, D. (2014). Falling behind early and staying behind when learning to program. In *Ppig* (Vol. 14).

Anderson, J. R., & Thompson, R. (1989). Use of analogy in a production system architecture.

Blackwell, A., & Green, T. (2003). Notational systems–the cognitive dimensions of notations framework. *HCI models, theories, and frameworks: toward an interdisciplinary science. Morgan Kaufmann*, *234*.

Blackwell, A. F. (1996). Metaphor or analogy: how should we see programming abstractions. In *Ppig* (p. 8).

Brand, C. L. (2016). Programming without code: A work in-progress paper. In *Ppig.*

Brohan, A., Brown, N., Carbajal, J., Chebotar, Y., Dabis, J., Finn, C., ... others (2022). Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint arXiv:2212.06817*.

Church, L., Söderberg, E., & McCabe, A. T. (2021). Breaking down and making up-a lens for conversing with compilers. In *Ppig.*

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, *31*(11), 1268–1287.

Dershowitz, N. (1986). Programming by analogy. *Machine learning-an artificial intelligence approach*, *2*, 393–422.

DiSessa, A. A. (1993). Toward an epistemology of physics. *Cognition and instruction*, *10*(2-3), 105–225.

diSessa, A. A., & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM*, *29*(9), 859–868.

Garfinkel, H. (1969). *Studies in ethnomethodology*. Prentice-Hall.

Good, J., & Brna, P. (1999). Getting a grip on the comprehension of data-flow visual programming languages. In *Ppig* (p. 7).

Gould, J. D., Lewis, C., & Becker, C. A. (1976). *Writing and following procedural, descriptive, and restricted syntax language instructions*. IBM Thomas J. Watson Research Division.

Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, 443–460.

Henisz-Dostert, B., & Thompson, F. (1969). A rapidly extensible language system: Rel english. In *1969 international conference on computational lingttistics.*

Heritage, J. (2013). *Garfinkel and ethnomethodology*. John Wiley & Sons.

Kendon, A. (1994). Do gestures communicate? a review. *Research on language and social interaction*, *27*(3), 175–200.

Khan, S., Naseer, M., Hayat, M., Zamir, S. W., Khan, F. S., & Shah, M. (2022). Transformers in vision: A survey. *ACM computing surveys (CSUR)*, *54*(10s), 1–41.

Lewis, C. (1988). Why and how to learn why: Analysis-based generalization of procedures. *Cognitive Science*, *12*(2), 211–256.

Lewis, C. (2019). Why can't programming be like sketching? In *Companion proceedings of the 3rd international conference on the art, science, and engineering of programming* (pp. 1–6).

Lewis, C. (2021). *Codex as a personal assistant?* (`https://docs.google.com/document/d/1b3LqmZa0dhk46WORy1kS7uxP_WjphCpbw-HdKgIWaXM/edit?usp=sharing`)

Lewis, C. (2022). Automatic programming and education. In *Companion proceedings of the 6th international conference on the art, science, and engineering of programming* (pp. 70–80).

Pane, J. F., & Myers, B. A. (2006). More natural programming languages and environments. In *End user development* (pp. 31–50). Springer.

Pane, J. F., Myers, B. A., et al. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, *54*(2), 237–264.

Petre, M., & Van Der Hoek, A. (2013). *Software designers in action: A human-centric look at design work*. CRC Press.

Roschelle, J., & Mason, J. (1995). Sketching a multidisciplinary microworld: A collaborative exploration in boxer. In *Computers and exploratory learning* (pp. 289–304). Springer.

Rosian, V. A. (2022). Intuition-enhancing gui for visual programming. In *Ppig* (pp. 39–46).

Simonyi, C., Christerson, M., & Clifford, S. (2006). Intentional software. In *Proceedings of the 21st annual acm sigplan conference on object-oriented programming systems, languages, and applications* (pp. 451–464).

Van Brummelen, J., Weng, K., Lin, P., & Yeo, C. (2020). Convo: What does conversational programming need? In *2020 ieee symposium on visual languages and human-centric computing (vl/hcc)* (pp. 1–5).

Webb, T., Holyoak, K. J., & Lu, H. (2023). Emergent analogical reasoning in large language models. *Nature Human Behaviour*, 1–16.

Interactive Narrative Visualization for Learning Markov Decision Process

B. Mbuu Mutua
Computer Laboratory
University of Cambridge
bmm51@cam.ac.uk

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

Abstract

This paper asks how we can use visualization tools to help learners develop intuition on probabilistic programming and how Markov Decision Processes (MDP) algorithm models problems in probabilistic planning. The project seeks to address this question by presenting a novel pedagogical technique that brings together visualization, use of interactive interfaces and a practical application of the semantic wave theory in a learning environment. We also present an evaluation study with university students in Kenya, focusing on use of narrative for teaching in that context.

1. Introduction

This paper adds to the existing literature on probabilistic planning and modeling using Markov Decision Process (Alagoz, Hsu, Schaefer, & Roberts, 2010), but on the perspective of computing and programming education. To fashion our motivation as a research question; how can we use visualization tools to help learners develop intuition for how Markov decision processes works and its use in probabilistic planning? This project seeks to address this question, by presenting a novel pedagogical technique that brings together visualization, use of interactive interfaces in computing and programming education and a practical application of the semantic wave theory (Maton, 2013) in a learning environment.

1.1. Research Objectives

The objectives of this paper can be stated in three points as follows:

1. To advance the acquisition of probabilistic reasoning and planning skills among undergraduate computer science students through an introduction of a novel technique of learning Markov Decision Process (MDP) as an important concept in Artificial Intelligence.

2. To further the work done by other researchers in the field on Computing and Programming Education especially on integration of digital aids/tools with learning;

3. To lay a foundation for the long term learning of probabilistic programming which underlies the idea of probabilistic planning and modeling. Numerous problems in AI and in real life necessitate that the agent operates with incomplete or uncertain information.

1.2. A brief review of Markov Decision Process (MDP)

For probabilistic planning, MDP uses a parametric probability distribution (Mausam & Kolobov, 2012) to model the various states that the agent or decision maker goes through while it transitions from the initial state to the final state of the planning environment. Through this distribution, we get a clearly defined mechanism of how an action taken at one state leads to another state; whereas a reward function determines the cumulative reward values the decision maker receives from the actions they have taken and the states they have transitioned through to get to the current state.

Since the determination of what action is to be taken is done at the current state, the whole MDP planning process is guided by the Markov Property, that is; For a stochastic process, the probability distribution of future states depends only on the present state (there are no dependencies on previous states or actions) (Khoshnevisan & Schilling, 2016).

In the interactive visual interface we have developed as an implementation of the ideas presented in this paper, we achieve that by presenting an interactive state chart to the learners as the first point of contact with the tool.

The state chart (which can be recreated to model different scenarios) introduces the learners to Markov Decision Process planning using well labeled states, actions, transitions and rewards, but more importantly, the concept of Markov Property where decisions are made at each state independent of prior actions or states.

1.3. Semantic Wave Theory

Semantic Wave Theory (Maton, 2013), a framework invented by Karl Malton as part of the Legitimation Code Theory project, describes a process where concepts are presented to learners with language that changes from technical to simple meanings and back (Curzon, 2019). Semantic wave theory has been used to structure learning across numerous disciplines since its invention, and it has been applied in computing and programming education with tremendous success.

A semantic wave illustrates the learning transition between the abstract and highly condensed technical explanation of a concept to the concrete, contextualised and simple language used to unpack the abstract concept and help learners understand it (Waite, Maton, Curzon, & Tuttiett, 2019).

Computing education, especially programming, is a subject that has benefited hugely from this theory since it has more than its fair share of technical terms (Curzon, Waite, Maton, & Donohue, 2020). These jargons form the basis of abstract concepts in Semantic Wave Theory. Learners are tasked with understanding these terms well while still applying them in programming exercises.

To illustrate the application of the Semantic Wave Theory in a pedagogical setting, when an educator walks into a class and introduces the concept of probabilistic planning using Markov Decision process (MDP), the learners are at the top of the semantic wave as the educator defines the technical terms such as uncertainty, probability distribution, transition function, state space, reward function and so on.

To help learners descend the semantic wave, the educator might give a day to day example of a student who wishes to plan their day that is full of demanding activities with the goal of saving as much time as possible to attend a movie show in the evening. Such student needs to come up with an optimal plan of how she will attend to both indoor and outdoor activities, factoring uncertainties such as the possibilities of rain falling any time of the day or the movie theatre management cancelling the show or a lecture taking longer than planned.

The educator must then go further to help the learners link this practical contextual scenario back to the abstract concept of probabilistic planning using Markov Decision Process (MDP); otherwise the wave will be incomplete. This will take the learners from the bottom of the wave where the language is contextual, back to the top of the wave where the educator can introduce more technical terms.

1.4. Computer Science Education in non-western settings

Use of visualization tools developed on digital interfaces is a practice that is gaining traction in education in general, but more so in computing education (Rosminah, 2013; Blackwell et al., 2019; Maloney et al., 2004). Computer science and programming education in Kenya, like in many other non-western settings, is done mainly using the traditional techniques of lectures and occasional lab sessions.

Research on the pedagogical approaches and resources invested towards this goal is still very limited; just like in many other sub Saharan African countries (Engineer Bainomugisha & Irungu, 2023), perhaps due to the sub-population disparities as outlined in the Capacity for, Access to, Participation in, and Experience of computer science education (CAPE) Framework (Fletcher & Warner, 2021).

In a pilot study carried out to understand computing education in four sub-Saharan African countries (Botswana, Kenya, Nigeria and Uganda) (Tshukudu et al., 2023), the authors collected data from teachers in these countries to investigate the resources available to them and the curriculum for computing, vis-a-vis teachers from developed countries.

The results showed that access to resources by teachers in these countries was alarmingly low as compared to teachers from other regions. Further, the authors argue that because of such reasons, there were very little efforts put by the teachers to teach algorithms and programming as compared to education settings in high-income countries (Tshukudu et al., 2023).

With the increasing penetration of digital devices among the student population in these countries, there is a very good cause for shifting the pedagogical approach towards use of digital interfaces, visualization and smart learning interactive screens (Morgado & Kahn, 2008). This approach has so far met tremendous success in the western setting where we find most high income countries, and in this project work we work towards the goal of creating one such learning tool tailored for a non-western education setting.

Our interactive tool for learning Probabilistic Planning using Markov Decision Process (MDP) received positive rating from the learners, and as we show in study results section of this paper, learners who used our interface to learn Probabilistic Planning and MDPs scored significantly higher points in a standardized test than those who depended on books and lectures.

2. Related work

2.1. Use of Visualization and Interactive Interfaces in Computing and Programming Education

In his work on the importance of Visualization in Education, Veřmiřovský (Veřmiřovský, 2013) argues that visual representation of concepts achieves very high level of clarity regardless of other factors such as the language used, difficult level and understanding ability of the learner. This has informed it's widespread use in all scientific fields including technology, construction, engineering, and architecture, just to mention a few (Veřmiřovský, 2013).

In relation to this work, one closely related project that we have drawn great inspiration from is by Erwig and Walkingshaw 2013 ("erwig2013Vis", n.d.). The authors of this work borrow the concept of explainable AI to present an explanation based and domain specific platform for visualizing probabilistic reasoning. On this platform, well understandable metaphors in form of stories that take the users of the tool through the steps involved in solving a probabilistic reasoning problem are presented. For interactivity, the in-coded programs in the platform can be edited to generate different versions of the same explanation.

In a different but closely related work on the use of digital interfaces in computing education, Blackwell et al (Blackwell et al., 2019), describe an approach to learning conditional probability in the context of Bayesian framework. This visual teaching tool (the spinner app) that is developed to be interactive, seeks to engage the the Ju|'hoan people who live in the Namibian region of the Kalahari, in probabilistic reasoning. The spinner app shows a number of spinners representing a binary choice with two sectors of different likelihood. The spinners flick on tapping the screen, and the app visually records the results of each 'tap' on a row of tally marks which accumulate over time, and a pie chart that shows the corresponding proportion of each outcome.

In (Zainab Attahiru, 2022), the authors build on the above work by Blackwell et. al. 2019 (Blackwell et al., 2019) to propose an interface for visualizing Bayesian probability in a school in Nigeria where the dynamics of learning and the pedagogical approaches differ from those found in a typical western education setting. The authors develop and present a web application that introduces users to the concept of Bayesian reasoning with the main goal being to nudge the users into the realization of importance of evidence in their beliefs.

2.2. Application of Semantic Wave Theory in Computing and Programming Education

Semantic wave Theory (Maton, 2013) has been successfully applied in education particularly in teaching computing and programming and has had revolutionary impact (Waite et al., 2019; Curzon et al., 2020). In a study published by the National centre for Computing Education on "Improving explanations and learning activities in computing using semantic waves" (Curzon, 2019), the authors argue that Semantic Wave Theory has had a far reaching impact in teaching programming through the use of unpacking / repacking techniques to decrease the complexity and precision of terminologies that have for long acted as hindrance to understanding of different programming paradigms by learners.

In “Teaching London Computing: Application of Semantic Waves” project, Curzon P, 2019 (Curzon, 2019) argues that programming, being a very technical subject with a lot of precise technical vocabulary to learn, requires a multifaceted approach in teaching, and making explanations and learning activities follow the semantic wave structure is a successful way of making the learning experience unforgettable.

Further, the author of this article cautions against flatlining when using semantic wave theory. Flatlining is a term given to a scenario where the teacher moves along same level of abstraction in the whole lesson. In a semantic profile, this can manifest as either high flatlining (Waite et al., 2019) – where explanations are done using completely technical language, as found in conversations involving experts, or low flatlining (Curzon et al., 2020) – where the educator makes everything overly simple using contextual language.

2.3. Use of narrative as a Tool in Learning

Narrative (also known as storytelling), is an implicit cultural practice in Africa and has been a medium of passing information from one generation to another for centuries (Osei-Tutu, 2022; Huber & Jonaitytė, 2020). Numerous studies conducted in the western culture also attest to the use of narrative as a tool of knowledge sharing not only in past centuries but also in modern era (Dimaculangan, Hadji Abas, & Quinto, 2022; Guillemin & Heggen, 2011; Hoggan & Strong, 1994; Ironside, 2006) So much is this practice prevalent that Butcher, S. 2006 in her work on “Narrative as a Teaching Strategy (Butcher, 2006)” explores narrative as a medium of teaching in modern educational setting, its effectiveness, and the role it plays in teacher and student connection, validation of student experience, and student perspective. The conclusion of this study is that narrative/storytelling is a practice so intertwined in education that it occupies two thirds of teacher student interactions in the form of examples, metaphors and explanations.

Similar observations have been made in several other studies (Forrest, Keener, & Harkins, 2010; Rogers, Marshall, & Tyson, 2006; Zander, 2007). The authors in (Dreon, Kerper, & Landis, 2011) also support Butcher’s conclusion on the influence narrative has in education, and goes ahead to argue that the shift from traditional storytelling to digital storytelling through YouTube as witnessed in recent times should offer teachers a platform to reach their students by creating educational content videos. A similar call is made by the authors of (Chung, 2007) and (Hung, Hwang, & Huang, 2012).

Scott Christian University Computer Science undergraduate students, who are the target users of our learning tool and where the evaluation study was done, come from different economic backgrounds ranging from agricultural practices, pastoralists communities, small business traders, transport sector and fishing. We targeted these backgrounds because a story related to any of them will work well in unpacking and explaining a technical concept.

For instance, the first scenario narrative that introduces the concepts of state space, action space and reward function in Markov Decision Process (MDP) tells a story of a cattle farmer who visits a business store to purchase milking jelly for his cows. The farmer has to satisfy constraints of time, sequentially chose an action at each step (in the face of multiple options) that take him to different states, while still maximizing his reward points for shopping in that particular store.

Whereas this story might look simplistic and slightly out of scope in a class that teaches probabilistic planning using Markov decision process, it is more relevant to a student who has grown up in agricultural practice environment than a story talking about robots and agents that might be of help to another student in a western education setting.

3. Implementation

3.1. Policy Graphs generation

The end product of a probabilistic planning process is a policy graph that outlines the sequential order of states and actions a decision maker need to take from the initial state to the goal state. With probabilistic

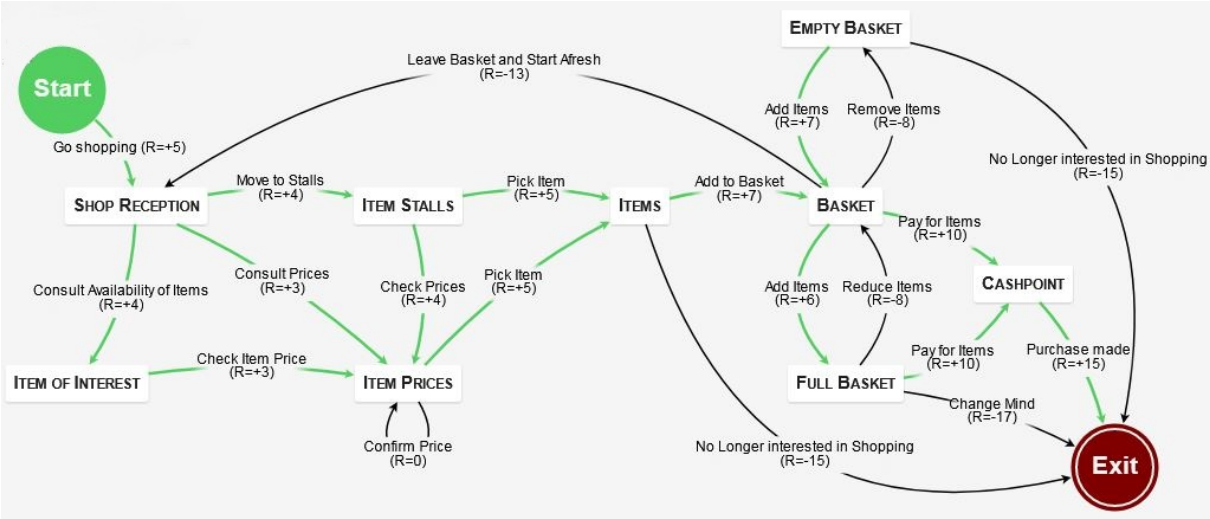


Figure 1 – Snippet of the MDP state Chart on our interface

planning using Markov Decision process, this policy graph presents the optimal set of states with their associated actions, all determined from the probability distribution that maps state space to action space, and the associated accumulative reward calculated using the reward function. Our interface is integrated with a policy graph generator algorithm that is developed using the RStan Probabilistic Programming Language (Guo, Gabry, Goodrich, & Weber, 2020).

3.2. Narrative

As explained in the background section of this report, semantic wave theory is based on two key components: abstract concept and concrete context. The section went further to justify our choices of Probabilistic Planning using Markov Decision Process as abstract concept and narrative as our concrete context presentation. Narrative is one of the core features implemented on our interactive interface to augment the learning experience of our educational tool. It also distinguishes our learning tool from hundreds of other tools used in computing and programming education.

3.3. User Interface Development

The user interface of our tool has mainly three sections: a Markov Decision Process state chart with well labeled actions, states and rewards, a JavaScript Object Notation script linked to the dimensional specifications of the state chart, and a narrative section that breaks down the abstract concepts of the MDP and which is connected to a probabilistic programming language policy graph generator algorithm.

The MDP state chart story-boards a typical shopping experience in a Kenyan Supermarket. The green transitions are for progress and they have positive reward values, while black transitions are for regress and they have zero or negative reward values. This chart introduces the learner to the concepts of state space, action space and reward function, where the learner can visually see the goal of probabilistic planning which is optimization of rewards or minimization of penalties/costs associated with each action. The business environment that the above chart is based on was chosen to align with the target user's daily encounters. Business education is a common subject taught to students in all programs in Scott Christian University in Kenya, and therefore it was prudent for the learning of probabilistic planning using MDPs to be based on a typical trip to a shopping mall by the students. That way the concepts would be easily relatable to the learners.

This state chart is developed to be interactive such that users can use drag and drop method to reorder the states and transition links and recreate a whole new scenario. This is handled by the controller program.

3.4. Optimal Policy Graphs Generation Algorithm

The outcome of a probabilistic planning process is an optimal policy. Optimal policies can be shown using an equation, a matrix or graph. For the sake of visualization, we chose to illustrate the optimal policies in our interface using state graphs. That justifies our integration of the optimal policy graph generator with the narrative section of our interface. Users can view a scenario and by a click of a button display the optimal policy graph generated from that particular scenario.

For this project, we sought a probabilistic programming environment where we could encode our state space, action space, probability distribution and reward function. We found RStan's Partially Observable Markov Decision Process (POMDP) library a perfect fit for our project. RStan is a combination of R and Stan, or simply put, the R interface to Stan (Guo et al., 2020). Stan is a probabilistic programming language extended from BUGs, while R is a free software environment for statistical computing and design.

4. Experimental Study Design

4.1. Target Users of our interface

Our interface was developed with the following target group in mind: Undergraduate Computer Science students in an institution based in a non-western learning environment; in particular, a university based in Kenya, where the cultural practices of the people are significantly different from those found in western countries.

We chose Kenya because there is no recorded use of interactive interfaces in computing and programming education, and therefore evaluation of our interface in the country's learning institutions is likely to give us an unbiased feedback on the efficacy of our tool in facilitating understanding of concepts.

4.2. The host institution

To achieve the goals of this project, Scott Christian University in Kenya was chosen as the study venue. The reason behind this choice is that the first author of this paper was a lecturer of computer science at the institution between 2019 and 2022 (just before joining the University of Cambridge), and therefore understands the teaching practices at the institution, and in Kenyan universities in general. This host institution was also settled at in order to reduce the bureaucracy common in Kenyan institutions of higher learning when requesting for approval to conduct these kinds of studies. This study was carried out on 12th April 2023 inside the Scott Christian University Computer Lab 1, between 10 AM and 1 PM.

4.3. Participants Recruitment

Probabilistic planning and Markov Decision Process being sub topics in Artificial Intelligence, we saw it prudent to recruit undergraduate computer science students taking Artificial Intelligence subject in the January to April 2023 semester at Scott Christian University. AI in the university's computer science department is taught when the students are in their third year of study (computer science program takes 4 academic years at the institution).

It is worth noting that apart from the undergraduate computer science program, the CS department also offers another related undergraduate program namely Bachelor of Business Information Technology (BBIT). These two programs have several common subjects, one of them being Artificial Intelligence, taught to students in both groups in their third year. For this study, I recruited both groups as participants because they were all taking Artificial Intelligence in the January to April 2023 semester. The Bachelor of Science in Computer Science group has 30 students, while the BBIT group has 17 students. All students in these two groups were present for the study, making the total number of participants in the study 47.

4.4. Set up of the study

For easy access the interface was hosted on the internet and the link and login details given to the participants. The venue where this study was conducted (Scott Christian University Computer Lab 1) has 53 desktop computers all connected with internet, and this made access to our interface challenge-free. However, some students requested to be allowed to access the interface using their tablets, for comfort.

Since the purpose of this study was to understand the impact of a digital interactive interface in facilitating understanding of computing concepts vis-a-vis the conventional method of teaching, the two groups of participants in this study were taught probabilistic planning and Markov Decision Process using the two different techniques. This was done on the material day of study (12th April 2023) between 10 AM and 11 AM. I used our interface to take the first group of participants through the concepts. This group comprised of students taking BSc Computer Science (30 students). The second group made of students taking BBIT (17 students) were taken through the topic by the means of a lecture by Dr. Ratemo, a senior lecturer in the computer science department of Scott Christian University. Both groups were then combined to tackle tasks testing their understanding of the concepts.

4.5. The Study Tasks

There were two set of tasks tackled by the participants. The first set consisted of the tasks on the interface, and required participants to perform the following:

1. Interact and recreate (model) a planning situation on the interface
2. Describe and illustrate the probabilistic planning process given a scenario in a narrative on the interface
3. Trace the optimal set of states and actions that take the decision maker from the initial / start state to the final state earning maximum rewards
4. Given a scenario in form of a narrative, identify the action space, state space, probability distribution of actions over the states, and the reward function
5. Analyse given scenarios and plot optimal policy graphs clearly showing the states, actions, related rewards and probability distributions

Participants who completed the tasks within the set time were allocated points according to the level of accuracy attained. For the tasks that required the participants to describe / illustrate a probabilistic planning process by plotting an optimal MDP policy graph, sheets of paper were distributed to the participants for writing the responses and collected at the end of the study for marking.

For the second set of tasks in the study, the participants who used our interface to learn (those in group one) were issued with evaluation survey forms with items that they were required to rate between 1 (Strongly Agree) to 5 (Strongly Disagree). These evaluation items were set to cover major usability areas of the interface:

1. Its success in facilitating the understanding of probabilistic planning and MDP concepts (without the learner needing to have studied the concepts elsewhere)
2. The ease of using the interface, for instance, was it easy to navigate across the sections of the interface?, likability (color, interface design etc), and user comfort while using it
3. Relevance of the text presented on the narratives, for instance, were the scenarios relatable?, was there excess/unnecessary text on the interface?
4. Accessibility, for instance, did the interface load on each device used?, were there any significant delays in loading (apart from those caused by a device's processing capacity)?
5. Lastly, the open comment section. This asked the participants for their opinions on the interface. For instance, there were questions such as: What did you like most about the interface? Which part interested you the least? Please suggest ways we can improve this tool

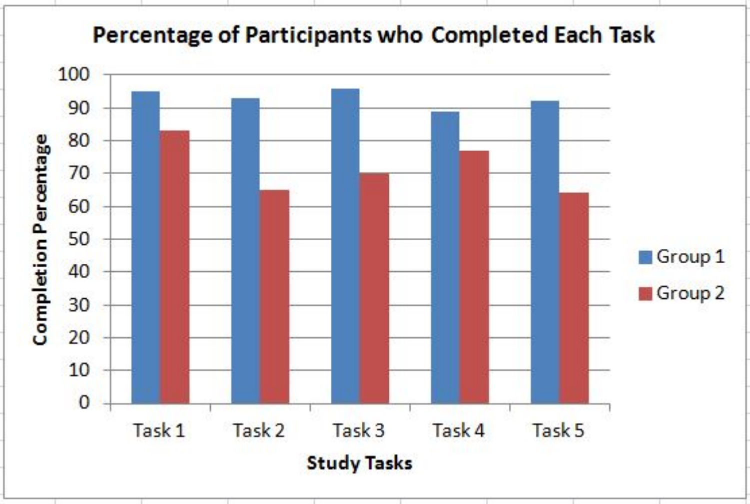


Figure 2 – A Bar Chart showing the percentage of participants who completed each task, from both groups

5. Study Results

The two groups of participants were taken through the concepts using two different techniques (one group learnt using our interface while the other learnt through the conventional method of a class lecture). Therefore, administering a standardized test to both groups was a sure way of finding out which group understood the concepts better and what the impact of our tool in learning is.

The results of the first section of the study that engaged participants in solving probabilistic planning tasks using Markov Decision Process concepts were analyzed based on a number of parameters. Such parameters included the number of students who were able to complete the tasks on time in each of the two groups, the response accuracy achieved by participants in each of the groups, and the average scores attained by participants in the two groups. Since both moderators of this study (Dr. Ratemo and I) have been examiners in the department for years, setting time for each task was not a challenge, and was done in a way that does not limit the participants' ability to tackle the study tasks, but at the same time encourages competence and quick thinking.

The bar chart in Figure 2 plots the percentages of participants from each group who completed the specific tasks within the set time. The results have been presented using percentage in order to avoid bias that might arise due to the difference between the numbers of participants in the two groups.

Still on the completion rates, we also sought to compare the two groups on the total tasks completion rate. That is, of all the participants who completed the five tasks within the set time, how many were part of group one and how many were part of group two. In total, 29 participants completed all the tasks in the study within the set time. 20 were from group one, whereas 9 were from group two. Since the higher number of total participants in group one influenced this composition, our interpretation of these results will be based on the ratio of total participants in group one to the total participants in group two. That way the bias attributed to the different numbers of participants will be eliminated. These completion rates translates to 62 percent and 38 percent for groups 1 and 2 respectively.

Lastly and equally important are the scores of the participants on each of the tasks in the first part of the study. Again to avoid bias, the results have been converted into average scores of the participants from each group and in each of the five tasks. The total scores of all the group members on a certain task are put together and then divided by the number of members; 30 for group one and 17 for group two. These scores are presented in the plot in Figure 4.

To ascertain that the differences between the task completion rates of group one and group two resulted from the effectiveness of our learning interface and not from a random variation, we conducted a chi

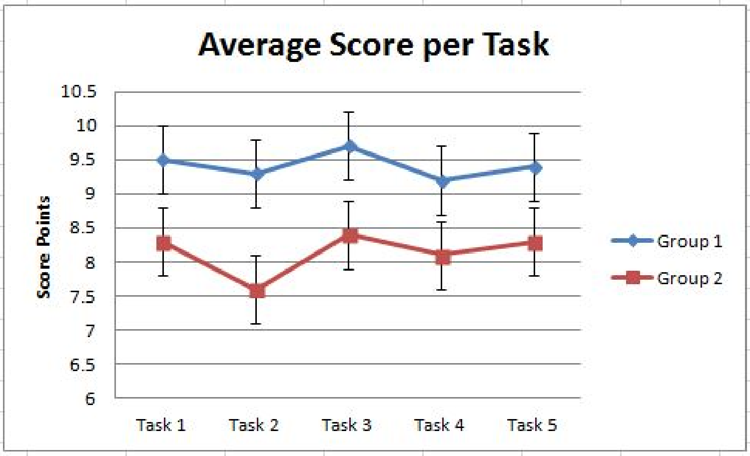


Figure 3 – A Plot of the average scores attained by participants of both groups in each task, with Error Bars of 1 Standard Deviation

square test of independence and a T-Test on the percentages of completion attained by both groups for each task. We conducted the test with 0.05 as the set level of significance (alpha value) and 4 Degrees of Freedom. The chi-square test gave us a P - value of 0.020728, while from the t-test we obtained a p-value of 0.023, both of which are below our set significance value of 0.05.

For the interface usability evaluation which was the second part of the study, 29.3 participants (on average) out of 30 strongly agreed that our interface facilitated their understanding of the presented concepts better than their current way of learning. On the ease of using the interface, 28.7 participants on average strongly agreed that the tool is easy to use, and transitioning from the basic concepts to the intermediary understanding to the more complex details was seamless.

The relevance of the material on the interface including the narratives scored equally high among the participants, with averagely 29.5 participants agreeing that the materials are highly relevant. All participants in this second part of the study were able to access the learning tool from their devices; therefore our interface recorded an accessibility score of 30 (out of 30). Lastly, the most common comments on the open comments section of the survey form were: "create an interface for other difficult concepts", and "digital learning interfaces are more effective than books".

We conclude:

- That our learning interface is more effective in facilitating understanding of probabilistic Planning and Markov Decision Process concepts as compared to the teaching method of lectures and textbooks.
- That the majority of participants (who are current undergraduate students) are more inclined towards learning using interactive interfaces as compared to the traditional and conventional method of books and lectures.

6. Conclusion and Future Work

6.1. Conclusion

This project work has built a case for the use of Markov Decision process (MDP) to model probabilistic planning problems. The project has gone ahead to fill the gap created by the absence of any existing work that creates a visualization of this concept (Probabilistic Planning using Markov decision Process). An interactive learning interface developed as an implementation of ideas in this report has also been presented.

This interface, as it has been proven in the paper, is an effective tool for teaching the concepts in an Artificial Intelligence class. This has been shown through results of an evaluation study conducted with participants recruited from an undergraduate computer science program in Scott Christian University in Kenya. The results show that participants who used the interface to learn the concepts scored three times higher in a standardized test than those who learnt the concepts through lectures and books. This attests to its efficacy.

6.2. Future Work

Subsequent work on the use of visualization, interactive interfaces and narratives in computing education will need to address the pertinent issue of scope. Whereas a text book takes learners from basic understanding to the complex concepts linking the ideas in between with everything else that the learner needs to understand, most interfaces used in education are too focused on the concept they seek to teach. Although this focus is their beauty, most often they leave the learners with knowledge gaps that have to be filled for the tool to be useful.

7. References

(n.d.).

Alagoz, O., Hsu, H., Schaefer, A. J., & Roberts, M. S. (2010). Markov decision processes: A tool for sequential decision making under uncertainty. *Medical Decision Making*, *30*(4), 474-483. Retrieved from `https://doi.org/10.1177/0272989X09353194` (PMID: 20044582) doi: 10.1177/0272989X09353194

Blackwell, A. F., Bidwell, N. J., Arnold, H. L., Nqeisji, C., Kunta, K., & Ujakpa, M. M. (2019). Visualizing bayesian probability in the kalahari.

Butcher, S. (2006, 01). Narrative as a teaching strategy. *Journal of Correctional Education*, *57*.

Chung, S. (2007, 01). Art education technology: digital storytelling. *Art Education*, *60*, 17-22.

Curzon, P. (2019). Teaching london computing: Semantic waves. *Linguistics and Education*. Retrieved from `https://teachinglondoncomputing.org/2019/06/29/semantic-waves/` (Cumulative knowledge-building in secondary schooling)

Curzon, P., Waite, J., Maton, K., & Donohue, J. (2020, 10). Using semantic waves to analyse the effectiveness of unplugged computing activities. In (p. 1-10). doi: 10.1145/3421590.3421606

Dimaculangan, K. A., Hadji Abas, H., & Quinto, C. S. (2022, Apr.). Narrative study of teaching strategies and challenges encountered by teachers in synchronous online classes. , *2*, 201–216. Retrieved from `https://ijsl.pubmedia.id/index.php/ijsl/article/view/113` doi: 10.47134/ijsl.v2i2.113

Dreon, O., Kerper, R., & Landis, J. (2011, 05). Digital storytelling: A tool for teaching and learning in the youtube generation. *Middle School Journal*, *42*, 4-10. doi: 10.1080/00940771.2011.11461777

Engineer Bainomugisha, M. M. U. J. N.-N. L. N. N. M. P. K., Karen Bradshaw, & Irungu, A. (2023). Computer science education in selected countries from sub-saharan africa.

Fletcher, C. L., & Warner, J. R. (2021, jan). Cape: A framework for assessing equity throughout the computer science education ecosystem. *Commun. ACM*, *64*(2), 23–25. Retrieved from `https://doi.org/10.1145/3442373` doi: 10.1145/3442373

Forrest, M., Keener, T., & Harkins, M. (2010, 05). Understanding narrative relations in teacher education. *Asia-Pacific Journal of Teacher Education*, *38*, 87-101. doi: 10.1080/13598661003677606

Guillemin, M., & Heggen, K. (2011, 12). The narrative approach as a learning strategy in the formation of novice researchers. *Qualitative health research*, *22*, 700-7. doi: 10.1177/1049732311431443

Guo, J., Gabry, J., Goodrich, B., & Weber, S. (2020). *RStan: the R interface to Stan.* Retrieved from `http://mc-stan.org/` (R package version 2.21.2)

Hoggan, K., & Strong, C. (1994, 04). The magic of "once upon a time". *Language Speech and Hearing Services in Schools*, *25*, 76. doi: 10.1044/0161-1461.2502.76

Huber, L., & Jonaitytė, E. (2020, 04). Oral narrative genres as communicative dialogic resources and their correlation to african short fiction. *Respectus Philologicus*, 137-146. doi: 10.15388/RESPECTUS.2020.37.42.45

Hung, C.-M., Hwang, G.-J., & Huang, I. (2012, 01). A project-based digital storytelling approach for improving students' learning motivation, problem-solving competence and learning achievement. *Educational Technology and Society*, *15*, 368-379.

Ironside, P. (2006, 09). Using narrative pedagogy: Learning and practising interpretive thinking. *Journal of advanced nursing*, *55*, 478-86. doi: 10.1111/j.1365-2648.2006.03938.x

Khoshnevisan, D., & Schilling, R. (2016, 12). On the markov property. In (p. 27-33). doi: 10.1007/978-3-319-34120-0_4

Maloney, J. H., Burd, L., Kafai, Y. B., Rusk, N., Silverman, B., & Resnick, M. (2004). Scratch: a sneak preview [education]. *Proceedings. Second International Conference on Creating, Connecting and Collaborating through Computing, 2004.*, 104-109.

Maton, K. (2013). Making semantic waves: A key to cumulative knowledge-building. *Linguistics and Education*, *24*(1), 8-22. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0898589812000678` (Cumulative knowledge-building in secondary schooling) doi: https://doi.org/10.1016/j.linged.2012.11.005

Mausam, & Kolobov, A. (2012, June). Planning with markov decision processes: An ai perspective. In *Synthesis lectures on artificial intelligence and machine learning* (p. 1-210). Morgan Claypool Publishers. Retrieved from `https://www.microsoft.com/en-us/research/publication/planning-markov-decision-processes-ai-perspective/`

Morgado, L., & Kahn, K. (2008, 10). Towards a specification of the toontalk language. *Journal of Visual Languages Computing*, *19*, 574-597. doi: 10.1016/j.jvlc.2007.10.002

Osei-Tutu, A. (2022, 05). Developing african oral traditional storytelling as a framework for studying with african peoples. *Qualitative Research*, 146879412210822. doi: 10.1177/14687941221082263

Rogers, T., Marshall, E., & Tyson, C. (2006, 04). Dialogic narratives of literacy, teaching, and schooling: Preparing literacy teachers for diverse settings. *Reading Research Quarterly - READ RES QUART*, *41*, 202-224. doi: 10.1598/RRQ.41.2.3

Rosminah, S. (2013). Integration of visualization techniques and active learning strategy in learning computer programming : A proposed framework..

Tshukudu, E., Sentance, S., Adelakun-Adeyemo, O., Nyaringita, B., Quille, K., & Zhong, Z. (2023, jan). Investigating k-12 computing education in four african countries (botswana, kenya, nigeria, and uganda). *ACM Trans. Comput. Educ.*, *23*(1). Retrieved from `https://doi.org/10.1145/3554924` doi: 10.1145/3554924

Veřmiřovský, J. (2013). The importance of visualisation in education..

Waite, J., Maton, K., Curzon, P., & Tuttiett, L. (2019, 09). Unplugged computing and semantic waves: Analysing crazy characters.. doi: 10.1145/3351287.3351291

Zainab Attahiru, A. F. B., Rowan Hall Maudslay. (2022). Interactive bayesian probability for learning in diverse populations.

Zander, M. J. (2007, 01). Tell me a story: The power of narrative in the practice of teaching art. *Studies in Art Education*, *48*, 189-203. doi: 10.1080/00393541.2007.11650099

Participatory prompting: a user-centric research method for eliciting AI assistance opportunities in knowledge workflows

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

Ian Drosos
Microsoft Research
t-iandrosos@microsoft.com

Rob Deline
Microsoft Research
rob.deline@microsoft.com

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

Carina Negreanu
Microsoft Research
cnegreanu@microsoft.com

Sean Rintel
Microsoft Research
serintel@microsoft.com

Jack Williams
Microsoft Research
jack.williams@microsoft.com

Ben Zorn
Microsoft Research
ben.zorn@microsoft.com

Abstract

Generative AI, such as image generation models and large language models, stands to provide tremendous value to end-user programmers in creative and knowledge workflows. Current research methods struggle to engage end-users in a realistic conversation that balances the actually existing capabilities of generative AI with the open-ended nature of user workflows and the many opportunities for the application of this technology. In this work-in-progress paper, we introduce participatory prompting, a method for eliciting opportunities for generative AI in end-user workflows. The participatory prompting method combines a contextual inquiry and a researcher-mediated interaction with a generative model, which helps study participants interact with a generative model without having to develop prompting strategies of their own. We discuss the ongoing development of a study whose aim will be to identify end-user programming opportunities for generative AI in data analysis workflows.

1. Introduction and motivation

Generative AI presents many opportunities for assistance and automation for end-users and end-user programmers. Our research team is interested in exploring how Large Language Model (LLM) assistance can be used in data-driven sensemaking (Russell, Stefik, Pirolli, & Card, 1993; Pirolli & Card, 2005) in spreadsheets, to identify key areas of strength and weakness in LLM assistance and identify opportunities for LLM assistance that address specific parts of the overall workflow. End-user data analysis workflows are complex and range over many steps, including problem conceptualization, identifying relevant datasets, data cleaning and structuring, developing an analysis strategy, learning how to use relevant features, open-ended exploration, and presenting results.

The effect of generative AI on knowledge work has been described as a shift *"from material production to critical integration"* (Sarkar, 2023). Critical integration consists of *"deciding where in the workflow to use the productive power of AI, how to program it correctly [...], and how to process its output in order to incorporate it"*. Sarkar builds on the theory of double-loop learning in organizations (Argyris, 1977), observing that there is both an inner-loop aspect to applying AI in knowledge workflows (incorporating AI assistance in various steps of existing workflows) as well as an outer-loop aspect (reconfiguring knowledge workflows to take better advantage of AI, and developing new ones which are only possible with AI).

For example, in data-driven sensemaking, critical integration in the inner loop might consist of finding applications for AI in data visualization, or data cleaning. Critical integration in the outer loop might consist of applying AI towards identifying a suitable analysis strategy or automating large portions of the sensemaking workflow (e.g., in the spirit of the "automatic statistician" (Steinruecken, Smith, Janz, Lloyd, & Ghahramani, 2019)) and developing new tools for human overseers focusing on auditing and quality control.

A key question for researchers and designers at this point in time is how to study the needs of users involved in such workflows. We are concerned with the first phase of the design "double diamond"; we first need to design the right thing, and only later can we attend to getting the design of the thing right (Buxton, 2010). As generative AI technology is new and continuously evolving, its use in society is limited and uneven. It may not be possible, for example, to simply observe participants working with generative AI, or interview them about their work practices with generative AI, if generative AI is not widely adopted within their workflows (which is the case for the vast majority of knowledge work at the time of writing). For example, code completion in code editors for professional software developers has been an early commercialization of generative AI, which gives researchers a wide pool of experienced users with mature behaviours to study (Sarkar et al., 2022). Other work has discussed code assistants for data analysis within computational notebooks (Mcnutt, Wang, Deline, & Drucker, 2023). Unfortunately, for our end-user scenario of interest (data analysis workflows in spreadsheets), this is not yet the case.

Furthermore, it is not ideal for researchers to develop high-fidelity experiences for generative AI as a way of testing its applicability to different workflow. It is time-consuming and expensive. Moreover, it is also limiting; out of the wide variety of potential interventions at the inner and outer loops, only a very small number can be feasibly explored using a functional prototype.

The traditional solution to this has been to use lower-fidelity methods such as Wizard-of-Oz (Gould, Conti, & Hovanyecz, 1983; Landauer, 1986), paper prototyping (Snyder, 2003) and champagne prototyping (Blackwell, Burnett, & Jones, 2004), which allow researchers to rapidly simulate a wide variety of user experiences with significantly lower engineering costs, while also enabling interaction with experiences that may be extremely challenging or impossible to build due to technical limitations. However, these methods have limitations as well; for a Wizard-of-Oz study to have direct implications for design, the Wizard protocol must correspond to the actually existing capabilities of the system(s) that are eventually built.

In particular, the mythologizing of AI's capabilities by the media, academia, and industry has led to a warped public conception of what AI can do and how it works (Sarkar, 2022). Thus Siddharth et al. (Siddarth et al., 2021) urge us to focus not on this collective mirage of what AI might be, but on "actually existing AI (AEAI)". There is a real risk that participant responses in low-fidelity studies will draw from their own biased and inflated expectations of AI capabilities to fill in the "gaps" left by the incomplete nature of the prototype. A poorly designed Wizard protocol, which allows too much improvisational deviation from a script, can exacerbate this. This problem is even greater in generic "need-finding" interviews where no prototypes are used.

There is thus a need for a research method that combines the advantages of low-fidelity methods such as Wizard-of-Oz, and rapidly exploring a wide range of potential interactions at both the inner and outer loops of a knowledge workflow, while still grounding conversations with participants in the capabilities of actually existing AI. In response, we have been developing a method called **participatory prompting**.

The participatory prompting method takes the form of a researcher-mediated interaction between a study participant and a working generative AI system. During the session, the researcher guides the participant through a workflow, seeking to test the potential applications for AI at each step. The researcher plays multiple roles in facilitating this interaction. Most importantly, they restructure user requests according to pre-identified prompting strategies, and help the user continue the interaction and recover from errors. The semi-structured interview is grounded in a specific real problem of interest to the user, drawing on principles of contextual inquiry (Raven & Flanders, 1996). The name of the method is inspired by participatory design (Spinuzzi, 2005), and we hope that in the spirit of participatory design, the method of participatory prompting contributes to the design of AI systems that empower and enfranchise users with their involvement from the outset. The next section describes the method in detail.

2. The participatory prompting method

2.1. Materials required

Choice of system. The participatory prompting method uses a real, functional generative AI system as representative of the functionality of generative AI in general. It is therefore important to choose the system carefully and consider multiple alternatives for their suitability to the particular study.

We compared the following four systems for their suitability for use in our study: OpenAI playground, OpenAI ChatGPT, Google Bard, and Microsoft Bing Chat. We compared them by entering some example queries that a user might have into each system, and attempting to elicit guidance and multiple stages of the data analysis process, as we were intending to do during the study. We then discussed and evaluated the comparative quality of the responses and how a participant might react to each response, with a view to choosing the system which would help produce the most insightful interactions during the study.

It is worth noting that of the four systems we tested, the latter three (ChatGPT, Bard, and Bing Chat) are consumer-facing products: they are built upon one or more large language models and consist of UI elements and other modules and heuristics which come together to create a coherent experience for the non-expert consumer. They can be considered to be significantly "opinionated" in a number of ways. One obvious user-facing manifestation of this is in the so-called "guardrails" which kick in whenever the conversation topic approaches an area deemed inappropriate by the system designers (such as violent or sexual content). Another example of opinionation is the turn limits imposed by Bing Chat: at the time of writing, a conversation with Bing Chat cannot exceed 15 turns (after 15 turns, the conversation is erased, and a new conversation is started).

In contrast, the OpenAI playground is intended for developers to interactively test different models, and as such allows for the choice between multiple individual LLMs, and control over parameters such as temperature (most of our testing on the OpenAI playground was using the default temperature of `0.7` and the `text-davinci-003` model, a GPT-3 model which was the state of the art at the time of testing). While there are still some heuristics and guardrails in place, using the OpenAI playground is much closer to getting the "raw" output of a language model.

For many studies, using a highly opinionated experience may not be ideal; the heuristics and modules used in these systems are proprietary, and researcher control and visibility into parameters such as temperature is poor. For our purposes, however, this was not a dealbreaker.

For our study we have chosen to use Bing Chat for the primary reason that it is the only system (at the time of writing) that is designed to seek and include information from the Web as part of its responses. In our testing, we found that for many steps of the data analysis journey (ideating potential analysis paths, identifying relevant datasets, learning to use relevant features), the ability to report information from the Web resulted in much better and more actionable suggestions for users.

Due to the complexity of deploying these systems at scale, during our comparative evaluation we noticed many outages, where the system was overloaded and did not respond to queries and/or displayed an error message. For a user study to go smoothly, a system that is stable and consistent is key. While we did not quantify the outages we experienced, our informal assessments of a particular system's reliability and uptime did influence our final decision.

Prompt strategies. Identifying performant and consistent strategies for prompting LLMs is a well-documented challenge. In consumer-facing products, the user query is rarely sent directly to an LLM; instead it is processed and augmented with additional instructions and prompts that have been determined by the system developers.

Thus, when non-experts directly interact with a "raw" LLM (e.g., via tools such as OpenAI playground), or with a generic chat application that is not tuned towards particular knowledge workflows, they may not be able to develop suitable prompting strategies to elicit good performance from the model. This is a key reason that our method involves researcher-mediated interaction, and why we do not simply study how end-users interact directly with the model.

A key strength of the participatory prompting method is that researchers familiar with the design of prompting strategies can prepare these ahead of time.

For our study, a group of researchers collaboratively experimented with different prompting strategies with Bing Chat over a period of several weeks, documenting screenshots of their interactions with Bing Chat and successful prompts in a shared document. A provisional list of prompting strategies which we developed through this process is given in Appendix B. For example, through this process we identified that Bing Chat:

- did not consistently use data sources from the web even if they were available, and we could bias it towards doing so by including a phrase in the prompt such as "use an online data source", "based on publicly available information", "with data from the web", and "use information from the web".
- did not consistently offer citations for sources, but could be biased to do so by including a phrase in the prompt such as "prove your sources are real" or "cite your sources".
- often provided multiple suggestions for types of data analysis the user could conduct, but the answers did not support an end-user's decision for what to do next. In this case we found that adding "justify your answer" or "justify your criterion" improved the actionability of the model's responses.
- is capable of rendering tables inline within the chat, which is very helpful for exploring ideas related to spreadsheet-based data analysis, but it does not consistently do so. We found that we could bias it towards generating tables by specifying "with an example", "make an example spreadsheet", or "make an example table".

Our method for identifying prompts is largely a pragmatic craft practice, based on trial-and-error and the intuitions of researchers. Due to the many sources of variability in LLM output, as well as variability between researchers' experience and the working examples they choose for testing different prompting strategies, our resulting prompts are subjective and difficult to reproduce. Another team, or the same team choosing different working examples, or a different model, may well have developed a different set of prompts, which will have significant downstream effects on the user study. Improving the consistency and systematicity of this step is a major challenge for user research with generative AI, as many libraries, toolkits, and even prompt marketplaces have been created to assist in this endeavour.

Demographics and generative AI experience. Participants will complete a standard demographics questionnaire which includes questions about spreadsheet experience, formula experience, and programming experience (Sarkar et al., 2020). In future participatory prompting studies, this can be replaced with another demographics questionnaire that gathers information relevant to those studies instead.

Based on the model of other questions in that questionnaire, we also developed a simple questionnaire item for assessing prior experience with generative AI, as follows: *"Which of the following BEST describes your experience with generative AI tools such as ChatGPT, DALL-E, Stable Diffusion, Midjourney, Google Bard, Bing Chat?"*

In response, participants choose from the following options:

1. Never heard of them
2. Heard of them but haven't tried any
3. Casually tried one or more
4. Occasionally use one or more
5. Regularly use one or more

As with other studies which use the aforementioned spreadsheet experience questionnaire, this item can be used in one of two ways: first, it can be used as part of the qualitative interpretation of participant interview data, to add context to their responses. Second, it can be used to group participants into rough categories of high and low prior experience (e.g., response levels 1-3 can be considered "low" experience and 4-5 can be considered "high" experience) for studying quantitative interactions between experience and any dependent variables gathered during the study (e.g., cognitive load (Hart & Staveland, 1988)).

Unlike spreadsheet experience or programming experience, the landscape of end-user experience with generative AI is shifting rapidly. The specific wording of this question and its response categories are thus likely to require periodic revision and updates.

2.2. Main interview activity

The main phase of the participatory prompting study takes the form of a semi-structured interview run concurrently with a researcher-mediated "conversation" between the participant and the model.

The interview consists of a number of "turns" consisting of 5 steps. (1) A turn begins by the participant expressing a query (e.g., asking for assistance, posing a question, asking for clarification). (2) Next, the researcher takes the user query, modifies and augments it according to the previously identified prompting strategies and sends it to the model. (3) The participant reads the model's response. (4) Next, the researcher asks the participant to reflect on the response. (5) Finally, the researcher guides the participant in continuing the conversation and choosing the next query.

We wish to explore the possibility for LLM assistance in the following scenarios:

- Problem conceptualization, decomposition, identifying parts of the problem that could be tackled in a spreadsheet
- Identifying relevant datasets
- Figuring out how to clean and structure data
- Developing an analytical strategy, involving applying multiple features in sequence
- Learning how to use relevant features
- Exploration of alternative analyses
- Presenting and communicating results

The problem chosen is ideally seeded by the participant's own problem domain. This can be elicited using a question such as: *"Can you share an example of a decision you had to make recently? The decision should be reasonably complex, requiring an evaluation of multiple criteria or sources."*

If elicited ahead of the study (e.g., in a pre-study communication, or as part of the initial demographics questionnaire), researchers could prepare a spreadsheet and problem that is familiar to the participant's own experience, or we can have the participant bring a shareable spreadsheet within their domain to work on. Alternatively, a suitable problem can be determined at the start of the interview. In practice we have found it is better to ask participants to think of such problems ahead of time, so that more time can be spent on the interactive portion of the interview.

The problems users bring can further be divided into the following types:

1. A well-established spreadsheet workflow where the user is already using spreadsheets.
2. An open-ended problem where the user has not tried to apply spreadsheets before.

From a research perspective, both types of seed problem have advantages, as they correspond respectively to the inner and outer loop of the double loop of AI assistance opportunities. In our study context, we are not interested in one or the other type in particular, or in comparing between the two, so we will not aim to control the distribution of types. However, future studies may be interested mainly in inner loop opportunities, or outer loop opportunities, or a direct comparison between them. In such cases care must be taken to ensure the seed problems used with participants are either predominantly of the type of concern, or roughly evenly distributed between the types to facilitate comparison.

With a suitable seed problem, we walk through the participant's problem step by step, entering their requests into the LLM system (using pre-identified prompt strategies) and relaying their response back to the user. We find that it is useful for participants themselves to also view the screen on which the LLM interaction is taking place, as the study can progress faster when participants can read the output directly themselves.

The user is then asked questions at each step such as:

- Is this useful? Why or why not?
- Are you confused or surprised? Why or why not?
- Does it contain anything that is factually incorrect or misleading?

To advance the conversation to the next turn, the experimenter may prompt the participant with a question such as

- Does this give you any further ideas?
- What would you like to know next, to continue your analysis?
- What information is missing?

Participants may also leverage suggested follow-up questions provided by the model as inspiration. However, since these suggestions may not adhere to experimenter prompting strategies, the suggestions may need intervention by the experimenter to align them.

2.2.1. Post-activity interview

After the turn-taking phase of the study, the participant is interviewed and asked to reflect on the experience. For instance, they could be asked how such a tool would fit into their workflow, what features they feel would improve the experience, and what were the strengths and weaknesses of the new modes of working enabled with generative AI.

This phase may additionally involve eliciting participants' responses to mock-ups of potential interface designs in a design probe. Importantly, because the participant has just had the experience of interacting with an actually existing AI system, they are more likely to have an accurate understanding of what a different interface design might actually achieve for their workflow in terms of usability. The participant's grounding in actual AI capabilities improves the validity of research insights over simply interviewing participants about their response to mock-ups.

If the participant reported that they had experience with generative AI, experimenters could elicit the participant to compare their previous workflows with what they experienced with participatory prompting. This might include the differences in solving similar or the same problems they saw in the study, or even understanding how the participant might change their own prompting strategies after the study.

The structure of and questions asked during the post-activity interview depends on the aims of the participatory prompting study. In our situation, we are interested in how generative AI can help non-expert end-users in data analysis workflows, particularly within spreadsheets, and so we selected our questions accordingly. Our full final script (incorporating revisions made after a pilot study detailed in Section 2.3) is given in Appendix A.

2.3. Pilot

The first version of this study protocol was piloted on a convenience sample of two participants who are familiar with spreadsheets and who use spreadsheets for their work. Each pilot took approximately 1 hour, as intended.

The pilots resulted in the following observations and adaptations:

It can be difficult for participants to settle on a suitable seed problem that is complex enough that it requires generative AI (as opposed to a traditional web search) to solve, but simple enough that the required context can be described to the system using a few sentences or a paragraph at most. We introduced more questions in the problem elicitation phase of the study that the experimenter could use to help the participant (e.g., *"Can you share an example of a problem that required you to develop a new workflow?"*). However, our recommendation is that if possible, the participant should be asked to think of a suitable problem ahead of the scheduled study session, to maximize the time available to engage with the problem in the turn taking phase. We also noticed that participants were not familiar with some jargon in our questions (e.g., "data-driven decision-making") and we modified our questions to elaborate and clarify these terms.

We noticed that the participants were able to go through 5-6 turns in the time allotted. This may seem like a small number of turns, but it nonetheless produced a wide range of qualitative insights. The turn-taking phase can be elongated in future studies if this is felt to be necessary, study duration targets and participant fatigue notwithstanding.

The most time-consuming aspect of each turn is in the reflection step, where the participant is asked to reflect on the system's response, and the advancement step, where the researcher guides the participant to decide what to do next. This observation helped us decide on setting Bing Chat to "creative" mode for the study. Bing Chat has a single user-facing setting. The user can choose between creative mode (described by the Bing Chat UI as "original and imaginative"), balanced mode ("informative and friendly"), and precise mode ("concise and straightforward"). We initially used "precise" mode because we believed that it would be the least likely to hallucinate misinformation, and because generating short responses would allow the user to read through them more quickly and therefore enable more turns overall. Since the number of turns is largely dominated by the time spent on the reflection and advancement phases, the small time advantage gained in precise mode by having to read less text did not accumulate to allow an increased number of turns overall. Moreover, in practice we observed that "creative" mode was no more likely to generate hallucinations, and since it was far more verbose, often emitting several paragraphs in response, it improved participants' reflections (by giving them more to reflect on) and the ease with which a suitable next query was selected.

We noticed that if the model's response is completely generic or not useful, especially at an early stage of the conversation, our pilot participants were not motivated to continue the interaction. In response to this, we introduced a number of different advancement-oriented questions the researcher could use to help suggest a way forward, such as: *"What would you change in your query to make this more useful? Would you ask this a different way?"*.

We noticed participants' preconceived notions about the system's capabilities were heavily influenced by their prior experience with search engines, and initially thought to use short queries of the kind used with search engines. This is unsurprising given Bing Chat's positioning within a more traditional search interface. Previous studies have also noted that participants' use of generative AI systems is influenced by their experience with search engines (Liu et al., 2023; Sarkar et al., 2022). However, such short queries cannot adequately capture the user's context and intent. We introduced a guidance statement in the protocol whereby the experimenter explains that the generative AI system permits longer and more conversational interaction.

We also introduced a couple of strategies for the experimenter to gently suggest a way to continue the conversation, if the participant was having difficulties ideating a next step. These include the experimenter directly suggesting an action (e.g. *"Let's see what happens if we try <some query>"*) but also suggesting an action as a baseline to help the participant conceive a contrastive alternative (e.g., *"I propose to continue by <some query>, but what would you have done instead?"*). However, it is important that the experimenter's suggestions do not bias or significantly change the course of the interaction, and serve mainly to unblock the participant. Much as with regular interviewing, the ability to elicit rich responses from the participant without introducing bias depends on the skill of the interviewer. To help guard against such bias, we recommend in the protocol that these experimenter-led strategies should not be employed in consecutive turns (i.e., if the experimenter led the query in the previous turn, they should not do so in the current turn).

There were issues with understanding participant expectations of model output, where even after working with the experimenter to craft a prompt, the participant did not know they would need to specifically request images. This led to needing to re-prompt the model to obtain the desired result. To prevent needing to do multiple prompts to obtain desired output, which can take up study time, experimenters should inquire on the expected results, including data types, from the participant to close this gap. Understanding what types of data could be useful for the participant, and explicitly requesting them in the prompt, is a necessary strategy.

Participants noticed that content the model had previously shown in the output could be missing in successive outputs. When the motivation of the participant was to build upon previous results, they wanted to make sure the data was consistent throughout the conversation with Bing Chat. Therefore, prompts crafted for the study need to include or refer to previous output in an attempt to have the model consider this data for further prompts.

One participant was suspicious of the data the model produced as a column in a table and wanted to verify this data by going to the websites the model referenced. This is a third workflow separate from prompting and spreadsheeting that requires a tangent into navigating to the source and verifying the data. This is a realistic strategy for users of chat based LLMs, but it is removed from prompting interactions. While this is not explicitly part of the protocol, we will allow participants the freedom to explore and verify the results returned by the model if desired.

In the post-activity interview, we noticed participants speculating on multiple occasions that *"if it could do <some action>, that would be helpful"*. Since these types of questions can actually be put to the system to test whether it can do it, we amended the protocol to permit the researcher to spot-test such participant speculations and get feedback from the participant. We also introduced the following question to specifically elicit perceived barriers to sensemaking with AI assistance: *"What barriers or frustrations did you have with this experience that prevented you from exploring the question to your satisfaction?"*

Our full revised script after the pilot is given in Appendix A.

2.4. Effectiveness of protocol during pilot

While we do not claim these are usable findings due to currently running a pilot of n=2 and the protocol was adapted live during these pilot runs, we believe there is evidence that this protocol was effective at revealing valuable insights from participants. These include:

- After a few prompts, a participant noted they would start a spreadsheet to maintain the data they were receiving from the model, but thought it would be difficult to switch between the spreadsheet and further prompting. Getting the model to generate a table to help the participant visualize a future spreadsheet was helpful in this situation.
- A participant was unsure the model considered the entire context of the prompt it was given, even when this context was in the prompt, and felt there was no way to verify this with the model.
- Upon noticing a result of potentially summarized or hallucinated data was given by the model, a participant noted that if they could not trust the results and had to manually search to verify the data in the table. They said they felt it severely limited the benefits of Bing Chat.

We believe this protocol is an improved adaptation of the traditional Wizard-of-Oz approach for studies of generative AI, since participants interact with a real AI model, but the implementation costs were extremely low.

3. Discussion and limitations

The participatory prompting approach detailed in this paper raises the question of whether some activities carried out by the human experimenter could be supported with AI. One question that remains to be answered is what affordances human prompt strategies have over an AI that is focused in helping the participant best interact with the generative AI.

However, such a protocol might increase the turn time or number of turns taken as the participant has to interact with this new AI prompting assistant while also performing their sensemaking task. Human-driven participatory prompting also allows the experimenter to ask user experience questions and inquire on participant motivations, which can provide valuable insights for researchers but may not be valuable for the actual prompting and might not be asked by an AI assistant.

One clear extension of this protocol is for the experimenter to also draw upon the library of existing AI plugins and recommend useful experiences that help the participant solve their problem. This could be directly compared to the effectiveness of the existing plugin experience where the model chooses which plugin to use, assuming the user has the installed plugins.

One limitation of this protocol is that because the experimenters will take a turn at helping craft prompts with the participant, results following this protocol may not give a clear understanding of where and when a user's unsupported prompting would have had issues. We attempt to address this limitation by having the participant reflect on how they would have modified a prompt (see Appendix A).

Similarly, because we are interested in how users might perform sensemaking in spreadsheets by leveraging AI, we have created an environment and crafted prompts that emphasize the use of spreadsheets and organized data. This might mean that the choice to move from prompting to spreadsheeting may not be as organic as it would if the user interacted with the model without experimenter assistance. There is a multitude of data analysis experiences that might also be useful for users (e.g., OpenAI's Code Interpreter, which performs data analysis tasks with Python code (OpenAI, 2023)). The freedom to choose from available interactions would provide useful insights for user needs for end-user data analysis and sensemaking tasks.

Some interactions were limited by the inability to re-trigger generation of a response with respect to a specific query within the conversation in Bing Chat (re-generation and editing a query is possible for instance, within ChatGPT and the OpenAI playground; this enables a kind of flexibility and fluidity that is akin to being able to independently edit and run different code cells out-of-order in a Jupyter notebook). For instance, if the participant changed their mind about a query, or if the system stopped generating text partway through a response, which worried one participant about continuing to prompt the model. In Bing Chat the only option is to submit a follow-up query within the same conversation, but which will include the undesired or incomplete previous queries and responses as part of the "context". The alternative is to start a fresh conversation and then laboriously "replay" the conversation, building up the same conversational state (or more likely, a *similar* state, since the system's responses are nondeterministic) through the same series of prompts until you arrive at the point in the conversation at which you wish to try a different query. Neither of these options is practical or predictable in a time-limited study.

4. Conclusion

In this work-in-progress paper we have presented the ongoing development of **participatory prompting**: a lightweight user research method for eliciting opportunities for AI assistance in knowledge workflows. It uses a real, functional generative AI system, thus improving on traditional Wizard-of-Oz or paper prototyping, where the user interaction can become unmoored from the technical reality of these systems. On the other hand, it allows researchers to use an "off-the-shelf" AI model with no additional engineering costs for fine-tuning, customization, or UI development, enabling rapid and broad-ranging testing of user experiences.

We reported a pilot study (n=2) in which we tested the participatory prompting protocol. The pilots have resulted in improvements to the protocol, changes to the script, and reflections on how to get the most insight out of a participatory prompting session. The pilots have validated the feasibility of the protocol as a method for understanding the user experience of generative AI in knowledge workflows. In future work, we are planning to run a full-scale participatory prompting study to elicit opportunities for AI assistance in the data analysis workflows of end-user programmers in spreadsheets.

5. References

Argyris, C. (1977). Double loop learning in organizations. *Harvard business review*, *55*(5), 115–125.

Blackwell, A. F., Burnett, M. M., & Jones, S. P. (2004). Champagne prototyping: A research technique for early evaluation of complex end-user programming systems. In *2004 ieee symposium on visual languages-human centric computing* (pp. 47–54).

Buxton, B. (2010). *Sketching user experiences: getting the design right and the right design*. Morgan kaufmann.

Gould, J. D., Conti, J., & Hovanyecz, T. (1983). Composing letters with a simulated listening typewriter. *Communications of the ACM*, *26*(4), 295–308.

Hart, S. G., & Staveland, L. E. (1988). Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology* (Vol. 52, pp. 139–183). Elsevier.

Landauer, T. K. (1986). Psychology as a mother of invention. *ACM SIGCHI Bulletin*, *18*(4), 333–335.

Liu, M. X., Sarkar, A., Negreanu, C., Zorn, B., Williams, J., Toronto, N., & Gordon, A. D. (2023). "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (pp. 1–31).

Mcnutt, A. M., Wang, C., Deline, R. A., & Drucker, S. M. (2023). On the design of ai-powered code assistants for notebooks. In *Proceedings of the 2023 chi conference on human factors in computing systems.* New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3544548.3580940` doi: 10.1145/3544548.3580940

OpenAI. (2023). *Chatgpt plugins: Code interpreter.* Retrieved from `https://openai.com/blog/chatgpt-plugins#code-interpreter`

Pirolli, P., & Card, S. (2005). The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of international conference on intelligence analysis* (Vol. 5, pp. 2–4).

Raven, M. E., & Flanders, A. (1996). Using contextual inquiry to learn about your audiences. *ACM SIGDOC Asterisk Journal of Computer Documentation*, *20*(1), 1–13.

Russell, D. M., Stefik, M. J., Pirolli, P., & Card, S. K. (1993). The cost structure of sensemaking. In *Proceedings of the interact'93 and chi'93 conference on human factors in computing systems* (pp. 269–276).

Sarkar, A. (2022, March). Is explainable AI a race against model complexity? In *Workshop on Transparency and Explanations in Smart Systems (TeXSS), in conjunction with ACM Intelligent User Interfaces (IUI 2022)* (pp. 192–199). Retrieved from `http://ceur-ws.org/Vol-3124/paper22.pdf`

Sarkar, A. (2023). Exploring Perspectives on the Impact of Artificial Intelligence on the Creativity of Knowledge Work: Beyond Mechanised Plagiarism and Stochastic Parrots. In *2023 Symposium on Human-Computer Interaction for Work (CHIWORK 2023)* (pp. 1–11).

Sarkar, A., Borghouts, J. W., Iyer, A., Khullar, S., Canton, C., Hermans, F., … Williams, J. (2020). Spreadsheet use and programming experience: An exploratory survey. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (pp. 1–9).

Sarkar, A., Gordon, A. D., Negreanu, C., Poelitz, C., Srinivasa Ragavan, S., & Zorn, B. (2022, September). What is it like to program with artificial intelligence? In *Proceedings of the 33rd Annual Conference of the Psychology of Programming Interest Group (PPIG 2022).*

Siddarth, D., Acemoglu, D., Allen, D., Crawford, K., Evans, J., Jordan, M., & Weyl, E. (2021). How AI fails us. *arXiv preprint arXiv:2201.04200*.

Snyder, C. (2003). *Paper prototyping: The fast and easy way to design and refine user interfaces*. Morgan Kaufmann.

Spinuzzi, C. (2005). The methodology of participatory design. *Technical communication*, *52*(2), 163–174.

Steinruecken, C., Smith, E., Janz, D., Lloyd, J., & Ghahramani, Z. (2019). The automatic statistician. *Automated machine learning: Methods, systems, challenges*, 161–173.

A. Study script

A.1. Materials/activities pre-interview

Ask to prepare spreadsheet / reflect on data-driven workflows.

A.2. [5 minutes] Opening

Introductions and pleasantries, consent form, demographics form.

A.3. [10 minutes] Discussion of current data decision practices.

- Can you briefly describe your role?
- Can you describe, with examples, what kinds of data-driven decision making you do as part of your role?
- Can you describe, with examples, what tools you use?
- Can you describe, with examples, how you approach an unfamiliar data-driven decision making problem? An unfamiliar problem where you had to make a decision based on some data. This could be tabular data, lists, or text, etc.
- Can you describe an unfamiliar data decision problem, potentially fictional, you may encounter in the future?

If this produces a satisfactory scenario, proceed to turn taking, else ask: Can you share an example of a problem that required you to develop a new workflow?

A.4. [30 minutes] Participatory prompting, turn taking

Per turn:

- Is this useful? Why or why not?
- Are you confused, surprised, or indirectly inspired?

To progress, choose one or more of:

- What would you like to know next? What else would you need to know to follow these suggestions?
- What would you change in your query to make this more useful? Would you ask this a different way?
- (Experimenter driven, at most once in a row) Let's see what happens if we try (X). Alternatively: I propose to continue by X, what would you have done (e.g., continued by Y, different task, abandon tool)?
- (If issue with result) I see that there is an issue here with (X), if we do this (new prompt) we can get that data back for you (if participant wants this, continue, else 1st question).
- (If participant is stuck, only thinking in terms of "classical" search engines) Imagine you're talking to a colleague, and bouncing ideas off them.

A.5. [15 minutes] Post activity interview

How would a tool like this fit or not fit into your workflow? If the participant says something like "If it could do X that would be helpful", try it out, get feedback from the participant, but keep time in mind.

1. What benefits does this hybrid spreadsheet-chat workflow provide over your existing workflow?
2. When you were surprised/inspired by X (from turn taking), what features/capabilities would be useful in exploring this inspiration further?
3. What features do you believe would increase the frequency and effectiveness of these inspiring results/moments (e.g., visualizations, videos, suggested prompts)?
4. How do you audit data/decisions now and how would that change with these AI-powered features?
5. How would your decision making workflow change with a tool like this?
6. What barriers or frustrations did you have with this experience that prevented you from exploring the question to your satisfaction?
7. What are the advantages or disadvantages of using a chat-based interface?

B. Pre-identified prompts

This section lists prompts that we have determined through trial and error for use during the study.

1. Problem conceptualization, decomposition, identifying parts of the problem that could be tackled in a spreadsheet
   (a) <Description of user problem>. Explain how to use a spreadsheet for this with an example.
   (b) Explain a different way to use a spreadsheet for this with an example.
   (c) I am trying to make a data-driven decision about <X>. Is this a good problem to use data tools such as spreadsheets to solve? Explain why or why not. What sub-problems or related problems are good candidates for spreadsheet solutions? Justify your answer.
2. Identifying relevant datasets
   (a) What data is relevant to this problem. List sources.
   (b) Use an online data source to add a useful column to this table. Prove your sources are real.
   (c) Add a column to the table containing a score representing <X>. Invent a criterion for this score based on publicly available information. Justify your criterion.
   (d) Add more rows and columns to the table based on information you consider relevant to the decision of <X>.
   (e) Add columns to the table with data from the web such as <X>. Cite your sources.
   (f) Use information from the web to populate the spreadsheet with more accurate figures. Cite your sources.
   (g) Make an example spreadsheet according to your suggestions above. Use information from the web to populate the spreadsheet with accurate information. Cite your sources.
3. Figuring out how to clean and structure data
   (a) Explain how to put this data in a spreadsheet with an example.
4. Developing an analytical strategy, involving application of multiple features in multiple steps
   (a) Explain how to <user problem> in Excel with steps.
   (b) Explain another way to <user problem> in Excel with steps.
   (c) It is not possible to <suggestion>. Explain an alternative method with steps.
   (d) What spreadsheet features can I use, such as charts, formulas, conditional formatting, pivot tables, etc. Show examples.
   (e) <Model suggestion> Explain how to do this with an example.
5. Learning how to use relevant features
   (a) Explain how to use <feature> to solve this problem in Excel with an example.
6. Exploration of alternative analyses
7. Presenting and communicating results

Others (non-categorised)

- Make a spreadsheet example
- <Model mistake> is not correct. Provide an alternative answer and prove that your answer is correct.
- Sometimes Bing Chat will refuse to make a spreadsheet. Try asking for a ‘table’ instead. Or ask repeatedly.

Prompt Programming for Large Language Models via Mixed Initiative Interaction in a GUI

Tanya A. Morris
Computer Laboratory
University of Cambridge
tam71@cantab.ac.uk

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

Abstract
Large Language Models (LLMs) now demonstrate many surprising capabilities that previously required special algorithms, for example interactive correction of syntax errors in structured text. However, the problem of how to systematically and reliably access these capabilities of LLMs has led to a new genre of "prompt programming" or "prompt engineering". This paper presents a design case study in which we apply OpenAI's Codex to an interface requiring syntax-constrained textual input, an email client. Via a mixed-initiative interface design, the system provides appropriate suggestions based on the output of the LLM. A user study was undertaken, finding that the incorporation of a LLM resulted in a decrease in perceived workload as well as a 62.5% reduction in errors. This work demonstrates how mixed-initiative interface design can better support attention investment in the use of LLMs, by delivering capabilities that might otherwise require prompt programming in a chat dialogue, but via a relatively conventional GUI.

1. Introduction
This paper describes a design case study exploring a new approach to the integration of predictive Large Language Models (LLMs) into functional user interfaces. At the time of writing, the release of the ChatGPT product, based on the GPT-n family of language models from OpenAI, has resulted in a step-change in popularity and public awareness of LLMs. In previous releases of the GPT-n series, and also in popular generative image models such as DALL-E and Stable Diffusion, substantial attention had been paid to the usability challenges of prompt engineering, or prompt programming. In ChatGPT and other dialog-based LLM deployments, the role of the generative prompt is combined with that of a chatbot dialog system. This obfuscates the distinction between user control of the generative output and Turing Test-style illusion of first person dialog to structure the interaction with the system.

Commentators on human-centric AI such as Ben Shneiderman (Shneiderman, 2020) have suggested that the first-person illusion is unhelpful or even unethical, and also that some people might find it easier to access LLM functionality if a constrained set of generative options were offered via a conventional user interface rather than a dialog prompt (Shneiderman, 2023). If the considerable power of LLMs was previously accessed via prompt *programming*, the substitution of more usable front ends for construction of those prompts can be considered as a new application of end-user programming. That is our motivation for presenting this work to the Psychology of Programming Interest Group.

Our case study is a classic domain for end-user programming research, in which non-programmers must construct and manipulate formally structured text syntax. Our illustrative example is the entry of syntactically-correct email addresses, when done by users who have impairments such that they may be unaware of the formal requirements of correct syntax, or be unable to generate valid examples due to cognitive, perceptual or motor impairments. We use the *Codex* variant of GPT-3 as the underlying language model. We apply principles of *mixed-initiative interaction*, an optimisation approach for design of intelligent user interfaces where AI agents collaborate with users to help them achieve their goals (Eric Horvitz, 1999).

2. Mixed-initiative interaction for user impairments
This application is designed for use by those who are facing cognitive, perceptual or motor impairments, including elderly users (Iancu & Iancu, 2017; Lindberg & De Troyer, 2021). (However, please note

that as an undergraduate student project, there is limited potential to undertake research with vulnerable populations - the evaluation study that is reported later was carried out with other students, as we will explain). For our intended population of users with impairments, in the case of entering email addresses, we address the following issues:

- *Difficulty remembering syntax* - suggestions to complete the email address will be offered to the user if they are paused, and corrections will be suggested if the syntactical rules of email addresses have not been followed.
- *Difficulty concentrating* - suggestions to change the input or move to a more supportive interface will appear if the user is paused for an extended period of time.
- *Difficulty using the keyboard* - suggestions or corrections will be offered if the user makes an error when inputting on the keyboard, and a user will be able to accept automated changes from the system.

Our system design applies key principles of mixed-initiative interaction (Eric Horvitz, 1999) as follows:

1. **Significant added value** where the user is unable to enter a correct email address.
2. **Consider uncertainty about the user's goal** by calculating the probability that the address needs to be corrected.
3. **Considering the status of a user's attention** by only interrupting the user if they are 'paused'.
4. **Inferring ideal action** by calculating the utility of each possible action.
5. **Employing dialog to resolve uncertainties** by offering a prompt with alternative actions.
6. **Allowing efficient direct invocation and termination:** If users want a higher or lower level of support, they can manually move to a more/less supportive interface via a task bar.
7. **Minimising costs of poor guesses about action and timing:** The dialog will timeout if the user does not respond. If the user continues typing, it will disappear.
8. **Providing mechanisms for efficient agent-user collaboration:** The user can accept an email address suggestion, and then edit it themselves.
9. **Employing socially appropriate behaviours for agent-user interaction:** The agent suggestions are polite and helpful.
10. **Maintaining working memory of recent transactions and continue to learn by observing:** The system continues to update the user model based on interaction with the system.

3. Implementation

Implementation is split into three main parts - the error model (including creation of an error model and appropriate Codex prompt), the user model (including the creation of the user model and its integration with the Codex prompt), and the GUI (including the creation of the graphical user interface and its integration with the user model). The back-end access to the Codex API was coded in Python and implemented as a Flask server. The front-end user model and GUI were coded in Javascript React.

3.1. Error Model

A dataset of pairs formatted as [erroneous email address, correct email address] was required to test various Codex prompts, and whether they could correct the erroneous email address. No existing dataset of this format existed, so we applied an error model to the Enron dataset[1] to create these pairs.

To form an appropriate error model, a corpus of spelling errors made by Wikipedia editors was used, which contains both the intended word (correct spelling) as well as the corresponding misspelt version[2]. The Levenshtein distance is used to calculate the 'difference' between strings, finding that the average edit distance of a misspelt word is 1.378 and the probability of an error at any given character is 0.158. The relative likelihood of that error being a substitution is 0.278, of a deletion is 0.428, and of an insertion is 0.294. Users are more likely to miss out a letter, as opposed to inserting an incorrect letter or substituting a letter for a different one.

A corpus of 150 email addresses was extracted from 'To:' fields in the Enron dataset, to which the error model was applied with probabilities of 0.01, 0.03 and 0.05, to explore the extent to which Codex can correct errors in email addresses.

3.2. User Model

The user model continually updates estimates of the attention of the user, estimating whether they are focused on their current task, and the skill level of the user, considering whether the difference between the text they have entered, and the most likely alternative suggested by Codex, implies they need more support. These values were used to form utility estimates that offered assistance when utility reached an appropriate threshold.

3.3. Codex Prompt

As is becoming increasingly well known, LLM behaviour in response to unconstrained text prompts can be unpredictable (Weidinger et al., 2022; Rae et al., 2022). Our approach, inspired by end-user programming, is to use a pre-tested structured prompt, with the input email address inserted into the prompt. This is inspired by Shneiderman's suggestion that a more structured approach can provide greater predictability for users (Shneiderman, 2023).

A variety of prompts to Codex were created, and evaluated using two Codex models: 'Cushman,' and 'Davinci' which is more capable but slower[3]. Other Codex parameters were `Temperature = 0`, to make the model more deterministic; `Stop = "\n"` to ensure that only one line of predicted text was returned; and `Best_of = true` to return only the most probable option.

OpenAI's guidance on prompt design (*Best practices for prompt engineering with OpenAI API | OpenAI Help Center*, n.d.) was followed, with all instructions at the beginning of the prompt. The 'context,' in this case the email address which requires correction, was separated from the instructions using triple quotation marks. Three prompts were tested as follows.

Prompt 1, no examples provided: `Correct the email address below. Incorrect company names or domain names should be corrected. Names which may have been spelt incorrectly should be corrected. Remove invalid symbols. The output should be an email address which is likely what the inputted email was intended to be. Email: """" + errorEmail + """" Correct Email:`

Prompt 2, the email address 'john.doe@hotmail.co.uk' is used to show the effect of individual errors (substitution, insertion, deletion) and how these should be corrected: `Correct the email address below. Incorrect company names or domain names should be corrected. Names which may have been spelt incorrectly should be corrected. Remove invalid symbols. The output should`

[1] https://www.cs.cmu.edu/ enron/
[2] https://www.dcs.bbk.ac.uk/ roger/wikipedia.dat
[3] https://platform.openai.com

be an email address which is likely what the inputted email was intended to be. Email: johbn.doe@hotmail.co.uk Correct Email: john.doe@hotmail.co.uk Email: johb.doe@hotmail.co.uk Correct Email: john.doe@hotmail.co.uk Email: jhn.doe@hotmail.co.uk Correct Email: john.doe@hotmail.co.uk Email: """" + errorEmail + """" Correct Email:

Prompt 3, a variety of email addresses were used, each showing how to correct multiple errors in one email address: Correct the email address below. Incorrect company names or domain names should be corrected. Names which may have been spelt incorrectly should be corrected. Remove invalid symbols. The output should be an email address which is likely what the inputted email was intended to be. Email: jpohn.doe@hotail.co.uk Correct Email: john.doe@hotmail.co.uk Email: alicesmith10o00@gmai1.com Correct Email: alicesmith1000@gmail.com Email: bjohnsonicloud,com Correct Email: bjohnson@icloud.com Email: """" + errorEmail + """" Correct Email:

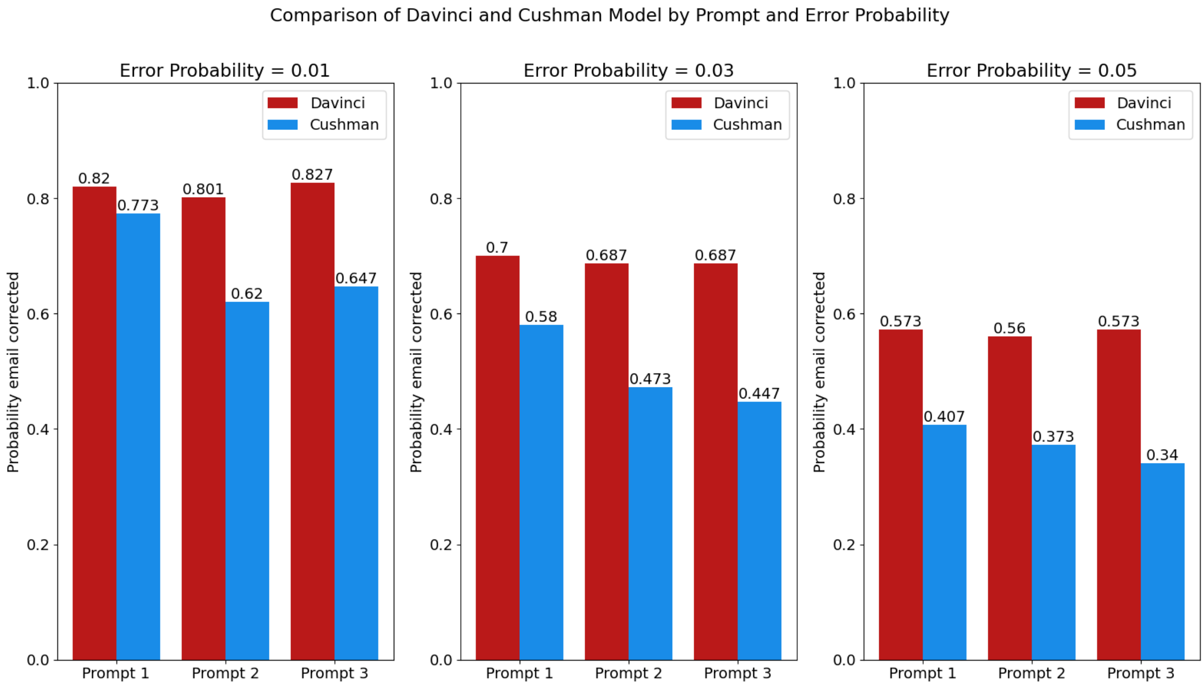


Figure 1 – Probability of an email address being successfully corrected by Codex, for each error probability, model, and prompt.

The results of the prompt evaluation are displayed in Figure 1. The Davinci model behaved similarly for each prompt, whereas the Cushman model performed better using Prompt 1. Davinci outperformed the Cushman model, with on average 17% more email addresses successfully corrected. Since the prompts had similar performance for the Davinci model, Prompt 1 was chosen because it requires fewer tokens to be sent.

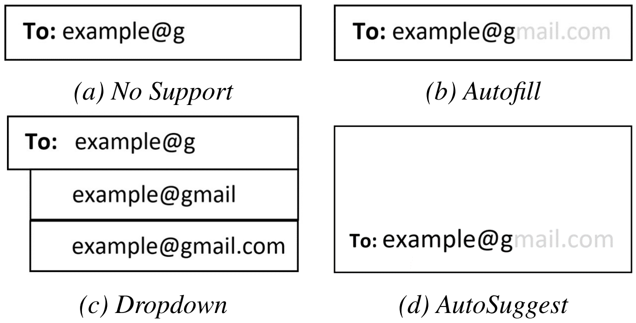
The Codex rate limit had to be considered when calling the API. Various techniques were employed to decrease the load on the API, and to avoid the user waiting while the system appears idle. On 23rd March, OpenAI discontinued support for Codex, including both the Davinci and Cushman models. Fortunately, we had already created a system for user testing which could use the edit distance to find the most similar cached response.

3.4. Interface overview

Four alternative input methods were implemented, offering increasing levels of support based on the Codex suggestions:

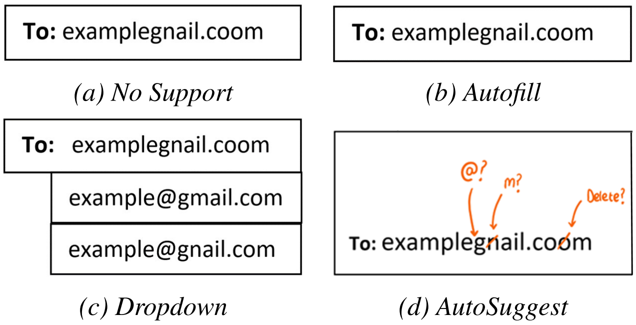
1. No support - a plain text input box.
2. Autofill - suggestions to complete the email address appear in the box, in a light grey shade.
3. Dropdown - suggestions to complete or correct the email address appear as a dropdown menu.
4. AutoSuggest - suggestions to correct the email appear above the text, with boxes suggesting insertion, deletion or substitution.

Mockups were created to understand how each of these options would behave when an email address is incomplete (Figure 2) compared to when an email address is erroneous (Figure 3). Note that the AutoSuggest option behaves in the same way as Autofill when there are no immediate errors to correct.



(a) No Support *(b) Autofill* *(c) Dropdown* *(d) AutoSuggest*

Figure 2 – How each text input option should behave when an email address is not completed.



(a) No Support *(b) Autofill* *(c) Dropdown* *(d) AutoSuggest*

Figure 3 – How each text input option should behave when an email address is erroneous.

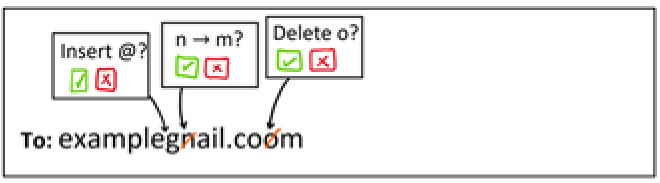


Figure 4 – Mockup including the scribble over errors.

Material UI 'MUI Core' React components were used to create the GUI, offering an open source library[4] to implement interactive elements of the project. The 'react-autocomplete-hint' component was employed to create the ghost text box[5].

[4]https://mui.com/pricing/
[5]https://www.npmjs.com/package/react-autocomplete-hint

3.5. AutoSuggest Interface

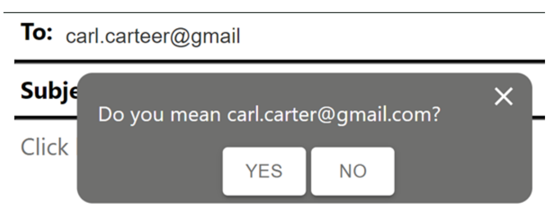
Typically, spelling errors are represented with a red line beneath the word. This does not tell the user where the error is or how to correct it, which is important to support an improved understanding of the syntax. Hence the most supportive interface option will point out specific errors (substitution, deletion, insertion), with the user able to accept or reject the system's suggestions.

An initial mockup showed a scribble over the text to be removed or replaced (Figure 4), but hiding the underlying text would make the text challenging to read, particularly for the target user who may already have reduced visual acuity. Instead, we reverted to the typical approach of a line underneath the text.

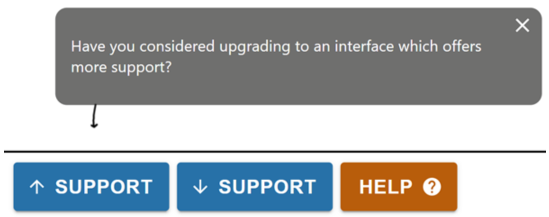
Figure 5 – Example of a suggestion being highlighted.

To support user control, it is important to respect when they reject suggestions by clicking the close button (Wen & Imamizu, 2022). A list of 'rejected' suggestions was formed to ensure the same suggestion was not made again.

Each page has a 'send' button, used in testing to submit the email and clear the input box. There are also 'increase support' and 'decrease support' buttons on each page, which enable a user to move between different interfaces. If the system deems a user needs help, but has no suggestions to change their current input (shown in Figure 6a), they may be prompted to move to a more supportive interface (shown in Figure 6b). The pages are summarised in Figure 7.



(a) A pop-up prompting a user to consider changing their input.



(b) A pop-up prompting a user to consider moving to a more supportive interface.

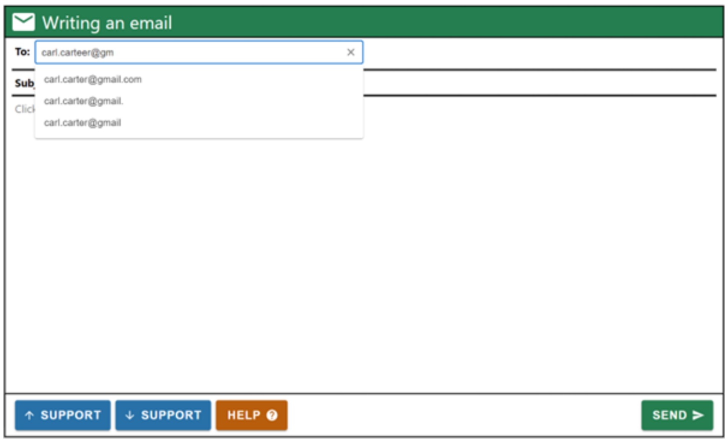
Figure 6 – Two types of pop-ups used in the system.

The previous example showed an email address with a suspected error, but each interface behaves differently if the email address is correct but incomplete (Figure 8).

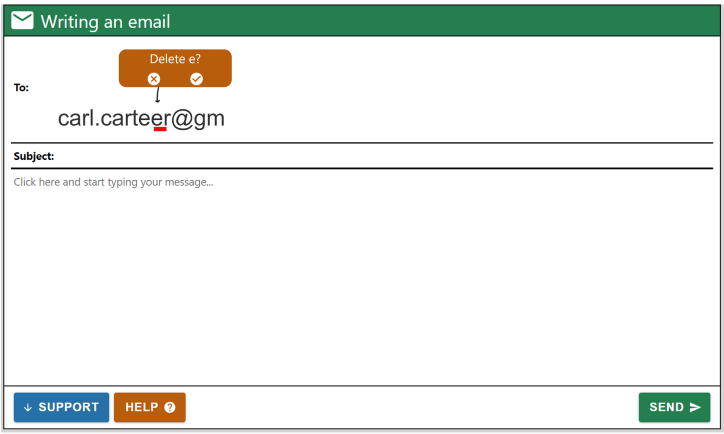
4. User Study

Because this student project did not offer scope to work with the target special population of older users with significant impairments, actual participants were Cambridge undergraduates, with errors forced into their typing stream to simulate the experience of the target users. A typical error rate for typing is 1.17%, with a standard deviation of 1.43% (Dhakal, Feit, Kristensson, & Oulasvirta, 2018). To approximate the skill level of someone who struggles to use a keyboard, we forced errors at 2 s.d. from the mean 4%.

A trade-off between the expected time required to complete the study and the volume of data expected to be retrieved was considered. While asking a user to input a higher number of email addresses into the system would provide more data to analyse, it would also increase the time required by the user, decreasing the likelihood of a participant agreeing to the study. 15 email addresses were randomly selected from the Enron dataset, with the study having an expected completion time of 15 minutes per

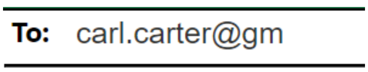


(a) Dropdown input box, where suggestions to correct and complete the email address appear in the dropdown menu, with the most specific option at the top.

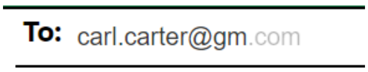


(b) AutoSuggest input box, where errors are pointed out individually.

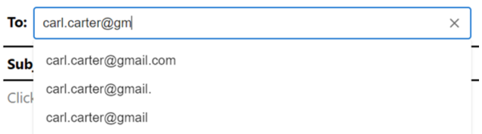
Figure 7 – Each page of the interface.



(a) The interface with no help.



(b) The Autofill interface displays 'ghost text' of Codex's prediction for the rest of the email.



(c) The Dropdown interface displays possible completions.

(d) The AutoSuggest interface also displays 'ghost text'.

Figure 8 – Each text input option interacting with a partially complete email address.

user. Having email addresses ending only with ‘enron.com’ could introduce bias and is not representative of the intended use, so each email address was given a 50% chance of being altered to take a different domain name (from the top domain names in the UK[6]).

The study has 5 sections:

1. Control: Navigation disabled, use only the interface with no help (including no help pop-ups).
2. Navigation disabled, use only the interface with autofill (with help pop-ups).
3. Navigation disabled, use only the interface with dropdown (with help pop-ups).
4. Navigation disabled, use only the interface with the auto-suggestions (with help pop-ups).
5. Intended use of system: Navigation enabled, maneuver between interface options if desired (with help pop-ups).

Users were warned that errors would be forced into their typing stream, but were asked to treat these in the same way as if they had made the error naturally.

The order of sections 1-4 was randomised for each participant, but section 5 was always last. The order of the email addresses was also randomised. In each section, users were asked to input 3 email addresses, which were shown in large font on a piece of paper. Once the user was ready, the email address was removed from sight, they typed it from memory, then clicked submit. Following each section, users completed the NASA TLX questionnaire (Hart, 1986).

At the end of the study, participants were asked to provide pairwise comparisons of the TLX factors, to be used in constructing a weighted ranking. The were then asked open questions about the support offered by the system.

A pilot study found that lack of errors led the system to believe the user did not need help. To increase the likelihood of participants needing help, we increased the likelihood of forced errors to 6%, and encouraged users to defer error correction until they had stopped typing.

14 participants were recruited, who were all Cambridge undergraduates from a variety of subjects. The study was approved by the Ethics Committee of the Cambridge Computer Lab. It took each participant approximately 17 minutes to complete.

5. Results

Written responses to the open questions were grouped, with the main themes summarised in Table 1. A majority of participants agreed that the system was easy to use, and that they would use a similar system again (Figure 9). Crucially, 71% of participants believed that the suggestions from the LLM added value to the interface.

[6]https://email-verify.my-addr.com/list-of-most-popular-email-domains.php

Sentiment	Theme	Times Mentioned	
Positive	Dropdown was best	4	14
	Helpful	7	
	Easy to use	3	
Negative	Would like a shortcut to select dropdown/buttons	3	11
	Help appeared too late	2	
	Hindrance or too busy	5	
	Suggestions inaccurate	1	

Table 1 – Common themes in response to ‘opinion of the interface’.

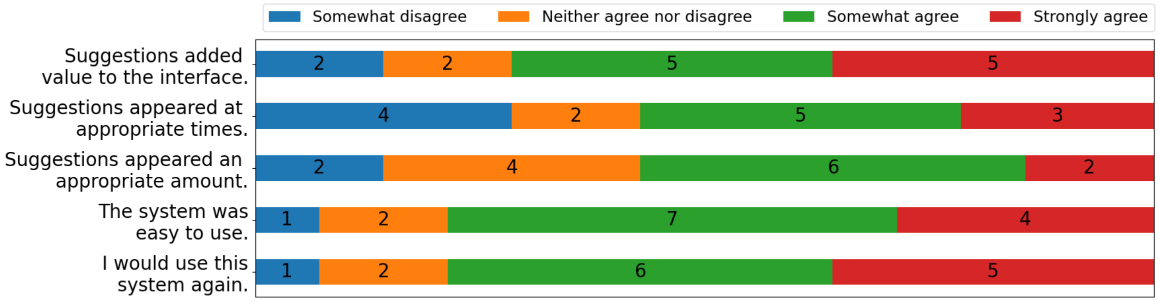
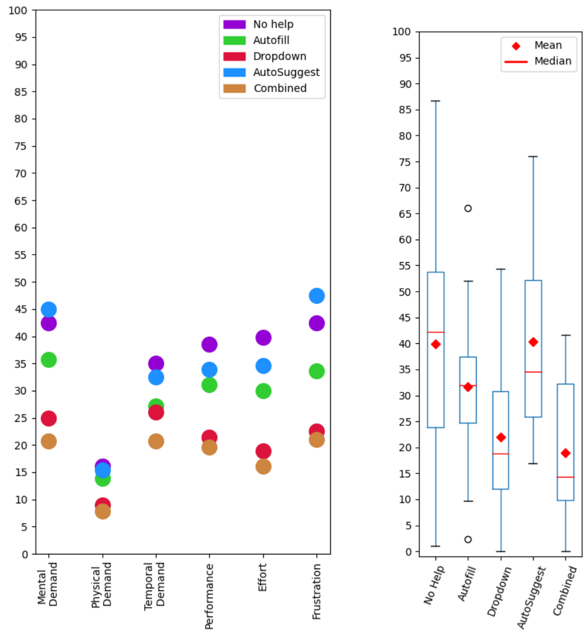


Figure 9 – Summary of final evaluation.

The NASA TLX responses for each section of the study, as in Figure 10a, shows that the novel AutoSuggest interface required the most mental demand and caused the most frustration. As expected, the interface which offered no help provided the worst performance rating yet required the highest effort. The overall system, where users could navigate freely, had the lowest demand in all categories, coupled with the best performance.

Users were asked to rank which factors have more impact on workload, and these ranks were used to create a weighted overall score (Figure 10b). The interface with no help has the greatest variability, suggesting some users appreciate extra support while others prefer to correct manually. The interface with autofill, which is most similar to existing products, had the smallest range.



(a) Comparisons of average perceived demand for each interface, including 'combined' (whole system).

(b) The mean, median and outliers for overall TLX of each interface option.

Figure 10 – Analysis of raw (a) and weighted (b) TLX scores for each interface.

To evaluate task performance, the average edit distance of entered email addresses from the correct address is displayed in Figure 11. The Autofill and Dropdown interfaces result in improved error rate. The AutoSuggest interface had more uncorrected errors than the interface with no help at all. It was unfamilar to users, with feedback suggesting that there were too many pop-ups to focus. The combined system, where users were free to manoeuvre between interfaces and customise their level of support, provided the best results.

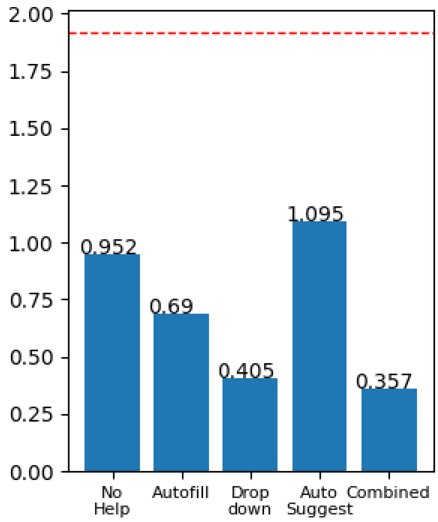


Figure 11 – The average edit distance of inputted emails for each interface, against the edit distance if no errors were corrected.

6. Conclusions

This project points to the positive impact that incorporating LLMs into 'everyday' interfaces can have. The adaptive combination of these supportive interfaces, allowing user choice and guided by a mixed-initiative utility model, showed not only an improvement in performance, but a decrease in perceived workload.

Despite the potential benefit of LLMs in a supportive interface, further issues must be considered, including risk of data leaks due to the training data of the LLM and risk of misinformation where the LLM may not be correct (Weidinger et al., 2022, 2021). These risks can be mitigated via careful prompt engineering. There is a further risk of intrinsic bias in the models (Bommasani et al., 2022; Weidinger et al., 2021), and it would be important to train a model with diverse email addresses, including names from a wide variety of cultures.

If LLMs are to be incorporated into supportive systems designed for people with impairments or elderly users, members of those populations should be included in discourse surrounding AI (Stypinska, 2022). Older people often self-stereotype, with lower self-efficacy leading them to doubt their ability to benefit from technology (Stypinska, 2022; Kottl, Gallistl, Rohner, & Ayalon, 2021). If AI is used to provide support in a mixed-initiative setting, such users should be able to make informed decisions about whether this support could benefit them (Stypinska, 2022). Instead of viewing AI as a tool which older users do not want to interact with, future research could turn to designing AI interfaces which are accessible for all.

7. Future Work

Further exploration, where LLMs are incorporated into other types of interfaces, could show whether these results are replicated in other contexts. In particular, the defined syntax required to type URLs into a browser would be applicable to a similar project. Other examples of defined syntax which could be explored include academic citations, phone numbers or credit card numbers.

Some participants in our limited user study did not want to interact with the help offered by the system, reporting that they do not in general enjoy 'helper' systems, because they find it quicker to correct errors manually themselves. Future work could explore whether these attitudes that we observed among students at the University of Cambridge are also reported by some members of the target design population of older and impaired users.

8. References

Best practices for prompt engineering with OpenAI API | OpenAI Help Center. (n.d.). Retrieved from https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., ... Liang, P. (2022, July). *On the Opportunities and Risks of Foundation Models.* arXiv. Retrieved 2023-04-04, from http://arxiv.org/abs/2108.07258 (arXiv:2108.07258 [cs])

Dhakal, V., Feit, A. M., Kristensson, P. O., & Oulasvirta, A. (2018). *Observations on Typing from 136 Million Keystrokes.* Retrieved 2023-01-20, from https://dl.acm.org/doi/epdf/10.1145/3173574.3174220 doi: 10.1145/3173574.3174220

Eric Horvitz. (1999). *Principles of mixed-initiative user interfaces.* Retrieved 2022-09-03, from https://dl.acm.org/doi/epdf/10.1145/302979.303030 doi: 10.1145/302979.303030

Hart, S. G. (1986, January). *NASA Task Load Index (TLX).* Retrieved 2023-03-03, from https://ntrs.nasa.gov/citations/20000021487 (NTRS Author Affiliations: NASA Ames Research Center NTRS Document ID: 20000021487 NTRS Research Center: Ames Research Center (ARC))

Iancu, I., & Iancu, B. (2017, September). Elderly in the Digital Era. Theoretical Perspectives on Assistive Technologies. *Technologies*, *5*(3), 60. Retrieved 2023-04-13, from https://www.mdpi.com/2227-7080/5/3/60 doi: 10.3390/technologies5030060

Kottl, H., Gallistl, V., Rohner, R., & Ayalon, L. (2021, September). *â[U+0080][U+009C]But at the age of 85? Forget it!â[U+0080][U+009D]: Internalized ageism, a barrier to technology use | Elsevier Enhanced Reader.*

Lindberg, R. S. N., & De Troyer, O. (2021, November). Towards an Up to Date list of Design Guidelines for Elderly Users. In *CHI Greece 2021: 1st International Conference of the ACM Greek SIGCHI Chapter* (pp. 1–7). Online (Athens, Greece) Greece: ACM. Retrieved 2023-02-01, from https://dl.acm.org/doi/10.1145/3489410.3489418 doi: 10.1145/3489410.3489418

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., ... Irving, G. (2022, January). *Scaling Language Models: Methods, Analysis & Insights from Training Gopher.* arXiv. Retrieved 2023-04-17, from http://arxiv.org/abs/2112.11446 (arXiv:2112.11446 [cs])

Shneiderman, B. (2020, April). Human-Centered Artificial Intelligence: Reliable, Safe & Trustworthy. *International Journal of Humanâ[U+0080][U+0093]Computer Interaction*, *36*(6), 495–504. Retrieved 2023-04-23, from https://www.tandfonline.com/doi/full/10.1080/10447318.2020.1741118 doi: 10.1080/10447318.2020.1741118

Shneiderman, B. (2023, June). *97th NOTE on Human-Centered AI.*

Stypinska, J. (2022, October). AI ageism: a critical roadmap for studying age discrimination and exclusion in digitalized societies. *AI & SOCIETY*. Retrieved 2023-04-28, from https://doi.org/10.1007/s00146-022-01553-5 doi: 10.1007/s00146-022-01553-5

Weidinger, L., Mellor, J., Rauh, M., Griffin, C., Uesato, J., Huang, P.-S., ... Gabriel, I. (2021, December). *Ethical and social risks of harm from Language Models.* arXiv. Retrieved 2023-04-07, from http://arxiv.org/abs/2112.04359

Weidinger, L., Uesato, J., Rauh, M., Griffin, C., Huang, P.-S., Mellor, J., ... Gabriel, I. (2022, June). Taxonomy of Risks posed by Language Models. In *2022 ACM Conference on Fairness, Accountability, and Transparency* (pp. 214–229). New York, NY, USA: Association for Computing Machinery. Retrieved 2023-04-14, from https://dl.acm.org/doi/10.1145/3531146.3533088 doi: 10.1145/3531146.3533088

Wen, W., & Imamizu, H. (2022, April). The sense of agency in perception, behaviour and humanâ[U+0080][U+0093]machine interactions. *Nature Reviews Psychology*, *1*(4), 211–222. Retrieved 2023-04-25, from https://www.nature.com/articles/s44159-022-00030-6 doi: 10.1038/s44159-022-00030-6

Back to the future: What do historical perspectives on programming environments tell us about LLMs?

Tao Dong
Google LLC

taodong@google.com

Luke Church
Computer Laboratory
Cambridge University
luke@church.name

Abstract

Many people, regardless of whether they consider themselves programmers, are interested in the ability of large language models (LLMs) to assist with programming tasks. The rapid rise of LLMs is transforming the socio-technical constraints and incentives under which programming environments are designed and adapted. This change is affecting languages, APIs, tools, and documentation. For instance, the practice of example-centric programming is being enhanced by LLMs, resulting in a shift of the primary programming activity from creating to curating. It may seem like this is all very new, but we see a resurgence of age-old themes in the psychology of programming and developer tools design.

Re-examining these historical themes in light of the growing popularity of LLM-assisted programming leads to open questions about how future programming environments will be designed. In particular, we argue that the implications of LLMs will go far beyond adding features to code editors. Design choices in programming languages and frameworks, which have so far been largely unaffected by LLMs, will be made differently due to changing programmer behaviors and preferences enabled and amplified by LLMs.

1. Introduction

Every few years a technology comes along that gives the feeling of changing everything, creating as venture capitalists might like to call it a ‘disruption’ to society. Looking back over recent years, these have included the invention of the Internet, development of search engines, the adoption of smartphones, all of which have had a profound impact on programming. Other changes seem to have a less enduring effect or a mixed outlook, such as mashups, the semantic web, cryptocurrencies, and the metaverse.

Inside the moment of any of the revolutions it can be difficult to know which way is up. They are often accompanied with a great deal of noise from techno-evangelists proclaiming a glorious future, moral panics of people warning about threats to the end of society and human morality, and techno-apathists saying that there will be no observable effect, we’ve been here before. All three of these groups have proven to be systematically incorrect. Amongst all of this noise, there is a practical need: To understand how these technical advances can be integrated into the almost mundane day to day work of software engineering, like plumbers, unglamorous but very necessary to current society.

Large Language Models are the latest of such technologies, simultaneously hailed as the future and the end of our future. In this paper we’ll leave these grand claims to others and focus on their likely impact on the making of programming environments, including programming language, APIs, tools, and documentation.

So far, the impact of LLMs on programming has been predominantly felt and embraced by IDE developers through the integration of multi-token code completion[1] and AI chat bots[2] in code editors. Their enthusiasm towards LLMs is, however, in stark contrast to the relatively muted reaction to LLMs from technical communities which design and build the foundational layers of a programming environment, including languages, frameworks, compilers, language servers, etc. The IDE might be the primary site of application for LLMs, but will their impact on the design of programming environments stop there?

A study of the history of programming language design teaches us that new capabilities in IDEs and the user behavior and expectations cultivated by such capabilities could have profound implications for the design of the rest of the programming environment. For example, static typing gained widespread acceptance partially because many programmers have developed an appreciation of, if not dependence on, accurate and fast code completion. In the past decade, we've seen static type systems get retrofitted to previously dynamic languages such as JavaScript (through TypeScript[3]) and Python (through Mojo[4]) at least partially for this purpose.

Given the co-evolutive nature of programming environments and programer behavior, will the proliferation of LLM use among programmers lead to a profound impact on the design of languages, frameworks, and other foundational building blocks of programming? How will these impacts affect design choices and decision making? We explore those questions by first revisiting age-old themes of programer behavior and experiences. We then speculate on how those phenomena could be amplified or changed by the adoption of LLMs. Finally, we discuss the implications of changing behaviors for the design of programming environments. It's too early in our understanding of this technology to really have answers; this paper is more an invitation to a conversation than a proposed way forward.

2. Recurring themes about programming in the age of LLMs

In this section, we revisit some long-running themes of human factors in programming and discuss how LLMs shed a new light on them. These are not in any particular order, or have any claim to completeness but rather act as prompts for our readers to think of other pre-existing themes and the new relevance they might have.

2.1. End-user development

End-User Development (EUD) refers to the phenomenon of users, often professionals with domain knowledge, writing programs for themselves to aid their work or everyday tasks (Nardi, 1993). This is usually in contrast to traditional Software Engineering where the programs are written for others, often as products. There have been many computational and notational paradigms EUD including:

- Spreadsheets
- Block-based programming (e.g., Scratch, VEXcode)
- Interface builders
- Diagram-based programming (e.g., Yahoo Pipes)
- Programming by examples
- Scripting (e.g., VBA, AppleScript, SQL queries, etc)

[1] https://github.com/features/copilot
[2] https://developer.android.com/studio/preview/studio-bot
[3] https://www.typescriptlang.org/
[4] https://www.modular.com/mojo

The psychological approach that many of these approaches take to make programming more accessible is turning recall tasks to recognition tasks. For example, block-based programming visualizes grammatical rules through different shapes of "sockets" and "plugs." And the success of interface builders owes to their ability to remove the need to recall code corresponding to visual layout and interactive elements.

Though EUD tools lowered the barriers to programming, they're often considered having low ceilings—easy to get started with but hard to achieve advanced results or fine-grained control. As Myers et al. (2000) commented,

> *"The most successful current systems seem to be either low threshold and low ceiling, or high threshold and high ceiling."*

LLMs offer the potential of enabling the psychological design maneuver of turning recognition into recall without the attendant limits to the ceiling of what can be achieved. This might be done through their surprising capability to generate working code based on natural language descriptions. The primary task in programming is shifting towards recognizing whether a piece of LLM-generated code can satisfy the requirements and identifying chunks of code that require further customization and refinement.

The implications for EUD can be profound. First, the threshold of previously high-threshold, high-ceiling tools are coming down, making low-threshold, low-ceiling tools on the market less attractive. A lowered threshold can help more end-users gain the confidence and desire to write programs, motivating more software systems to offer EUD support, either via a natural language style interaction, a general purpose language whose interaction is intermediated through an LLM, or a mixture of the two.

Second, one of the perceived utility of visual programming tools might need to be re-evaluated, since a key benefit they provide—turning recall tasks into recognition tasks—may be obtainable through conversing with LLMs in a natural language. Even before the introduction of LLMs this trend was already emerging, for example, with declarative UI programming techniques making UI builders less popular. The adoption of LLMs might further decrease the utility of UI builders in their current form. With the introduction of multimodal LLMs with capabilities to generate code based on visual input (e.g., UI sketches), the design of UI builders must evolve to complement users' multimodal interactions with LLMs.

2.2. Example-centric programming and opportunistic programming

Example-centric programming (ECP) describes a pragmatic approach towards writing code by retrieving and modifying examples found on the Internet (Brandt et al., 2010). The approach was popularized by the widespread use of online Q&A forums for programming such as StackOverflow and open-source code hosting platforms such as SourceForge and eventually Github. This approach is closely related to opportunistic programming (Brandt et al., 2009), a term used to describe how online information foraging, just-in-time learning, and sometimes coding without understanding are practiced in the production of computer programs.

As widespread as ECP has become, researchers have noted that doing so is not without its challenges. Systems have been designed and implemented to address two common challenges when practicing ECP: 1) constructing effective queries are difficult, especially when specific attributes of user's programming environments need to be considered; and 2) sifting through search results for relevant code is tedious and slow. In response, some ECP systems embed a custom search facility in the IDE to help the user construct search queries and clean up results (Brandt et al., 2010).

LLMs significantly lowered these two barriers to practicing ECP through code completion and generation. Prompts—equivalent to queries in previous systems—are constructed with natural language and snippets of code the user has written and additional context supplied by the IDE. In some cases an entire codebase can be included as contextual information in the prompt.[5] The output of those LLMs are well-formatted code that can be iteratively adjusted through tweaking the prompt. Early research shows that using LLM-based assistance could boost productivity by as much as 55.8 percent in select tasks (Peng et al., 2023). As a result, ECP is likely to become even more popular among programmers, especially when learning a new programming language or using an unfamiliar library. We can expect more code to be written opportunistically based on examples suggested by an LLM.

The full understanding of the task being performed of selecting the relevant material out of the example, the equivalent of information foraging within an LLM, is not well characterized. It is also not currently clear how other problems with ECP, such as the associated security challenges, might carry across to code generated via LLMs, and whether the tools used for managing emergent properties of statistical tools such as 'de-biasing' could be effectively applied to address them.

2.3. Live programming

Traditional programming systems require an explicit interaction between changing the content of a program and observing the effects of its execution. Technical steps such as compilation often take place after the user requests it. This adds up to creating an episodic user experience where the user makes a change and then waits to see what the effect of that change is.

Live programming provides an alternative to this. Initially characterized by Tanimoto into 4 levels (Tanimoto, 1990), at the higher levels it describes an experience where parts of the interface and associated notations react rapidly after the user has made a change to the program.

Live programming interaction styles are especially beneficial in some interaction domains. A genre of music has emerged around live coding, where the execution of the program creates the sound effect, and the program is often modified live on stage, often with the audience able to see the changes whilst they dance in an event known as an algorave (Blackwell & Cocker, 2022).

Interactive creation of music is not the only application domain in which liveness is a benefit. Liveness has been applied to game design (Kato & Shimakage, 2020), spreadsheets, data analysis (Church, Marasoiu, et al., 2016), and user interface engineering in general purpose programming languages[6][7] where it sometimes goes by the name 'hot reload'.

There are many challenges with using live systems in production environments, these include the relative instability of the behavior of the systems being created, so as you make small changes the program flickers, to problems that most modern programming languages try to exclude partial or incomplete programs, but during development most programs are 'illegal' most of the time. What the right behavior in these intermediary stages is is a domain specific problem, hence we have not seen many general purpose live programming environments, but rather separate ones for music, for data and for UIs.

[5] https://github.com/mpoon/gpt-repository-loader
[6] https://docs.flutter.dev/tools/hot-reload
[7] https://gaearon.github.io/react-hot-loader/getstarted/

In previous work (Church, Söderberg, et al., 2016) we speculated on how some of the interaction problems with Liveness could be addressed using program synthesis techniques, both having the programming language infrastructure 'fix up' the program to the most probable outcome that the user is 'likely to have meant', this could be extended to large areas of functional synthesis where significant components are automatically generated.

The primary effect that LLMs have here is to make the synthesis computationally possible and perhaps even plausible. As this means that the task of creating the code to run becomes easier, the focus will shift on to interacting with and understanding the behavior of the program, including how to provide safe environments where the behavior of speculative programs can be understood without damaging side effects.

2.4. Path of least resistance

Path of least resistance is a notion introduced by Myers et al. (2000) in their paper *Past, Present, and Future of User Interface Software Tools*. The main insight is that properties of tools can influence the kinds of programs more likely to be written through making it easier to achieve the desirable outcomes. In UI programming, one of the major advancements in the past decade is the rise of design systems (Churchill, 2019), which make well-designed UI components readily available through UI frameworks and ensure UI designers and developers communicate with the same UI primitives. Design systems made it much easier to create UIs with adequate usability and aesthetics without direct support from designers.

Nonetheless, it remains a challenge to encourage programmers to adapt standard UI components to accommodate users with special needs, international audiences, and multiple types of devices. Model-based UI development was proposed as a potential solution to this challenge. The general concept is that the programmer specifies the UI as an abstract model of user tasks and then has the system create concrete user interfaces based on attributes of the user and the computing environment (Gajos & Weld, 2004; Miñón et al., 2016). However, this model-based approach has yet to gain traction due to the difficulty of constructing models in the first place.

LLM provides a new possibility to offer paths of least resistance in programming through its power to transform code, both newly generated examples and the user's existing code, through prompt engineering.

2.5. Student and Professional Programmers

As we are suggesting above, the adoption of LLMs within programming is likely to have an effect in a number of different ways depending on what activity is being performed and for what purpose. Programmers are in general not homogeneous, so we wouldn't expect them all to be affected in the same way. We have already suggested above that end-user developers may have a different experience of LLM-mediated programming to professionals. Similarly the effect of LLMs may be different to students than it is to professionals, reflecting their differing motivations and practices.

For example, we've already suggested a number of times above that as program synthesis becomes easier more of the work of programming will become rapid comprehension and curation as opposed to authoring. However, this presents a challenge to the underlying educational philosophy of software engineering which is often dominated by constructionist perspectives—that the act of building something, especially out of logical components such as program statements, provides a strong form of knowledge. Instead, LLMs could shift the intellectual approach in software engineering education to one better characterized as "reverse engineering," "discovery," and "evaluation," often more associated with scientific disciplines than engineering ones. Perhaps this brings the working practice of the student, closer to the pragmatic approaches often adopted by industry professionals[8].

LLMs may also affect the aspirations and motivations of both individual students and corporate actors. If there is little to be gained from a systematic study of programming, rather than an opportunistic one, will students still be interested in studying computer science? And if LLMs increase the cross-language mobility of code, will the corporate sponsors of programming languages still be interested in getting their languages into classrooms?

3. Implications for building programming environments

As has happened with previous "disruptions," programming environments will respond to new evolutionary pressures created by the adoption of LLMs as a programming aid. At first this is likely to occur with local adaptations in the places that are easiest to evolve—often the code editors—but we predict these pressures will start to shape the rest of the technology stack a programmer needs to interact with as well.

3.1. Notational Evolution

Previous work in the study of the evolution of notations might give us some pointers as to how these changes might take place. Green and Fetais (2016) and others (Green and Church, forthcoming) have pointed out that notations change along predictable patterns. LLMs within programming can currently be seen as very large "helper devices" (Green & Petre, 1996)—that is external components that sit off to the side of the main notation peripherally interacting with it.

There are a number of other such tools in the evolution of programming environments, some, such as Stack Overflow, have remained largely independent of the editing context, but even so have had a significant influence on development practice and on the design of an API or an IDE. Others such as static analysis tools have tended to be folded into the environment directly, and are now an expected part of programming support.

In both these cases the tools and resources then co-evolve with the programming language and the APIs. The most notable example of this is the introduction of code completion into Visual Studio, where the designers of the C# language explicitly stated that it was designed to be written in IDEs with this feature. This supports the long running argument of Cognitive Dimensions that notations and the environments where notations are used need to be considered as a joint design exercise.

Whilst it's difficult to predict the course of evolution that LLMs will take, it seems likely that it will subsequently influence the design of future tooling and notations, either via integration or independent evolution. In this section, we consider some concrete examples of how this evolution might take place.

3.1. Evolving a programming language or framework

Introducing a new syntax to a programming language or new APIs to a framework often carries risks of increased complexity, insufficient adoption, and higher maintenance cost. Such risks are even higher when the new syntaxes and APIs are not backwards compatible. *Will the benefit of the new feature outweigh its risks and costs?* That's a question programming language and API designers often need to consider. For every feature that sees the light of the day, many others are buried after tradeoffs are evaluated.

[8] https://blog.codinghorror.com/mort-elvis-einstein-and-you/

How will LLMs influence the outcome of evaluating those tradeoffs? One potential consequence is that it could become harder to introduce programmers to the latest language or API changes, once they form a habit of asking LLMs rather than a search engine for example code. As mentioned earlier, LLMs enable example-centric programming at an unprecedented level of efficiency. However, LLMs are biased against, if not completely ignorant of, new changes to a programming language or framework. Without mitigations, users' growing reliance on LLMs in coding can slow down the adoption of new features and prolong the life of deprecated APIs. This will in turn demotivate new language and API changes, especially when the changes in question are to improve user experience rather than filling a mission-critical gap such as fixing a security vulnerability.

The provider of LLM-based coding assistance can potentially mitigate the problem through prompt engineering and retrieval augmented generation.[9] One possibility is injecting information about new features into the user's prompt behind the scene, once the user's original prompt matches the programming language or API in question. For example, a library author reported that by including short usage examples for newly introduced features as part of a prompt, ChatGPT was able to generate code utilizing those new features correctly.[10] Nonetheless, both how effective and how reliable this technique can be remain open questions.

Another possibility is to instruct the LLM to acknowledge what information it doesn't have access to and direct the user to alternate methods of programming assistance. This requires the LLM to disclose its data cutoff date and have the ability to estimate how likely the user's prompt requires more recent information.

It's worth pointing out both types of mitigations require coordination between LLM assistance providers and vendors of programming languages and frameworks. Thus, LLM service providers could become an intermediary between programmers and the languages and APIs they use in a way more prominent than web searches have been. Maintaining a good relationship with major LLM assistance providers might become vital to a programming language's ability to evolve its features without disrupting its users.

3.2. Encouraging good programming practices

Programming language and framework designers want their users to use their products in the right way. Some common concerns include writing tests, supporting accessibility, avoiding security pitfalls, making code readable and well-documented. Making those tasks easier to carry out is a powerful way to get programmers to do them more often, because it's natural for people to gravitate towards "the path of least resistance." As Myers et al. (2000) pointed out, *"Our tools have a profound effect on how we think about problems. It will remain imperative that our tools make 'doing the right thing' easy (in preference to 'doing the wrong thing')."*

Before LLMs became part of programming support, lowering such resistance against good practices was often achieved through scaffolding, verification, and automation. First, scaffolding includes resources such as templates, example code, and saved snippets. They provide reminders and partially implemented solutions for the programmers to follow and build on. Second, verification involves tools such as lints, accessibility checkers, and vulnerability scanners. They ensure flaws are discovered sooner rather than later. Last, automation is often applied to routine tasks such as running tests, generating API docs from source code and comments, and upgrading dependencies.

[9] https://www.pinecone.io/learn/retrieval-augmented-generation/
[10] https://www.reddit.com/r/ChatGPT/comments/12xzcpj/teach_new_apis_to_chatgpt_in_one_prompt/

LLMs have the potential to augment all these three approaches. To begin with, LLMs can personalize templates and examples used to scaffold good programming practices based on the project context either provided by the user explicitly or the user's IDE implicitly. Next, LLMs have shown an ability to check whether a program follows specific requirements. For example, it's been documented that ChatGPT can detect accessibility issues in HTML code and make suggestions to fix them.[11] The interactions between traditional lints and LLM-driven checks will be especially interesting to explore, as LLMs are being more deeply integrated into code editors. Finally, LLMs can potentially automate more tasks such as generating inline comments[12] and enumerating test cases[13].

Moreover, LLMs could incentivize better programming practices, because they generate more useful responses when the user includes clearly-written and well-documented code in the prompts. For example, Jones (2023) demonstrated that using descriptive variable and function names and including inline comments improved ChatGPT's ability to reason about the results of executing a program.

3.3. Designing end-user development support

LLMs are likely to fuel the growth of end-user development tools both as standalone software and as extension mechanisms for existing systems. As mentioned earlier, this growth will be driven by two factors. First, LLMs can lower the barriers for end-users to leveraging existing programming interfaces through code generation from natural languages, and hence making such tools no longer reserved to power users. Second, as more users taste the power of customizing their software, they will demand and pay for EUD support in more types of software systems than what's available today. This creates economic incentives for software developers to invest in EUD support.

When designing an EUD tool, one of the central questions is: *how much code will the user have to deal with?* Based on the answer to this question, EUD tools are often classified as no-code or low-code tools. LLMs are likely to make the no-code approach less appealing, as more users become accustomed to consuming LLM-generated code.

LLMs can influence the evolution of low-code tools as well, in particular, how they leverage visual programming techniques. On the one hand, LLMs are likely to make visual programming less necessary for end-users to create programs. On the other hand, LLMs could make it more important to visualize a program—both its structure and its output—so the end-user developer can quickly verify the program's behavior. Related, liveness of the EUD tool will be highly valuable due to the shift of cognitive work from program authoring to program verification.

Another fundamental question LLMs could influence is whether the EUD tool should provide a domain-specific language (DSL) or adopt a popular general-purpose language. A niche DSL, while often simpler in its syntax and higher in its abstraction, can make it harder to leverage LLMs' code generation capabilities than general-purpose languages in wide-spread usage. The tradeoff could gradually tip in favor of the latter as LLMs improve.

[11] https://intopia.digital/articles/using-chatgpt-to-make-chatgpts-experience-more-accessible/
[12] https://tecadmin.net/add-comments-and-refactor-your-code-with-chatgpt/
[13] https://research.aimultiple.com/chatgpt-test-automation/

3.4. Introduction of new programming languages

Up to this point we've mainly discussed the trade-off space around how LLMs, used as a helper device, influence existing ecosystems and working practices. However, new languages will be designed after the introduction of LLMs, and just as C# and the .NET libraries were influenced by the presence of code completion; we can expect any such new language to be influenced by the existence of LLMs.

Experiments in language production are rare and expensive, as are LLMs themselves. It will take some time before the way in which LLMs influence language design becomes clear, but we can speculate around how such a technology might affect when it makes sense to invent a new programming language or to adapt existing ones.

On the one hand, LLMs could make it more costly to create new programming languages in the future. LLMs rely on large corpuses of existing code to be available to train the models, the methods by which to make such training data available for a new language is currently unclear, as is the needed quantity and shape of the training data. The implication of this "data" requirement may amplify the extent to which successful languages are only built by very wealthy organizations.

On the other hand, once the amount of code written in a new programming language reaches a critical mass, which is unknown at this time, LLMs can lower the cost of its continued development and accelerate its adoption. This is because a significant fraction of the cost of a language is associated with supporting developers performing tasks that LLMs already do well, such as curating examples of how to open files, or translating large existing bodies of code from one language to another. The existence of powerful automated support tools for performing these tasks is likely to significantly decrease the switching cost for developers from one language to another.

Related, programmers might feel less "attached" to an existing language and consider that as part of their professional identity. Current languages are designed mostly with the process of creating code in mind, but as we have pointed out above, with LLMs this may become a less significant component of the overall cost of creating software, and instead the focus may be on consuming, curating and verifying the behavior of code generated by LLMs.

If the production and interchange between languages becomes very low cost, there is the possibility of using different languages for different purposes. For example, if it were easier to implement an algorithm in Haskell, but a web component in Javascript, and a security audit in Rust/WASM, perhaps you might write the initial code in Haskell and ask an LLM-based tool to translate it to Javascript, integrate it into a Web API, and then ask the LLM to convert it into Rust to be read. This kind of practice is equivalent to the experience of people who speak multiple languages fluently occasionally wanting to express a subtle concept in one language rather than another.

If this, perhaps far fetched, idea becomes plausible it may be that it is the logical conclusion of the research strand discussed in the Match Mismatch hypothesis (Green et al., 1991), that the choices that we have previously seen as fundamental such as OOP vs functional, or static vs dynamic are only tradeoffs for a given purpose (reading, algorithm development, testing), rather than fundamental tradeoffs in themselves. It is perhaps slightly ironic that a tool that comes close to passing the Turing test, demonstrates for the first time the practical utility of the Church-Turing thesis, which suggests that any computation that can be performed by a Turing machine can also be programmed in any programming language.

A further productive avenue of research in LLM-enabled programming languages would be understanding how the data that can be captured about the dynamic execution of a program can be combined with its static description. This is a core task that developers perform during production debugging of systems, and it rests on the ability to manage and manipulate very large volumes of data, but we are not aware of any work currently exploring this direction.

3.5. Supporting live programming and dynamic execution of code

As we have discussed, the introduction of the ability to synthesize significant volumes of code via LLMs may facilitate a change from expressing the program logic to reviewing created programs and their behaviors. This would make it significantly beneficial to be able to execute programs in order to see what effect they have. However in the general case this is risky - what if the program does something you don't want, a say a script has learned `rm -rf *`? You wouldn't want to execute this to "try and see if it's the right program" without the ability to go back.

This has always been the case with live programming environments, and perhaps may be one of the reasons that they're most commonly used in contexts where mistakes might even be welcomed such as online code playgrounds.

4. Anti-Conclusion

LLMs are clearly a significant technical advance and will affect to a greater or lesser extent the way in which programmers go about doing their work. As some members of the PPIG community have been pointing out (Blackwell, 2022), we need to develop a response to this change.

It can sometimes feel that this is rather irrelevant, that the technology contains its own intrinsic logic, and that any opposition to this logic is futile - a position sometimes known as "techno-determinism." This would place the community solely in the position of adapting to the changes as best it could.

However historically this doesn't tend to have been a good description of invention—instead there's a middle ground that we've been proposing in this paper. If we as a community think that the creation of programming languages and their associated ecosystems of technology, *now including LLMs*, are valuable contributions to society, what can we do?

In this discussion, we've seen some of the benefits that LLM might bring the creation of new languages (e.g., easier translation, easier learning, lower switching costs), some of the challenges (e.g., the need for large training sets), and some of the potential opportunities (e.g., better use of live programming technology, a replacement of drudgery with more interesting work, and empowerment of end-user developers).

It's much too early in the life cycle of these technologies to propose a conclusion - instead we'd like to engage in a discussion with programming language and tool designers, researchers, and educators. To do this, we suggest three broad questions for consideration, and encourage those that are interested to get in touch with more:

1. How has the rising adoption of LLMs by programmers changed the way you think about your craft, and motivated or demotivated plans in your institution?
2. If we think that the design, creation, and evaluation of programming languages and tooling is of fundamental benefit to society, how should we react to the shifts associated with the invention of LLMs?

3. What are the benefits and costs of LLMs, and what are some strategies that we might use to leverage and mitigate them?

4. References

Blackwell, A.F. (1999) How to format PPIG submissions. International Journal of PPIG Administrivia, 1(1), 1-3.

Blackwell, A. F. (2022). Coding or AI? Tools for Control, Surprise and Creativity. *PPIG*, 57–66.

Blackwell, A. F., & Cocker, E. (2022). *Live Coding: A User's Manual (Software Studies)*. MIT Press.

Brandt, J., Dontcheva, M., Weskamp, M., & Klemmer, S. R. (2010). Example-centric programming: integrating web search into the development environment. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 513–522.

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1589–1598.

Churchill, E. F. (2019). Scaling UX with design systems. *Interactions*, *26*(5), 22–23.

Church, L., Marasoiu, M., & Blackwell, A. F. (2016, July). Sintr: Experimenting with liveness at scale. *2nd Workshop of Live Programming Systems (LIVE 2016)*. 2nd Workshop of Live Programming Systems, Rome, Italy.

Church, L., Söderberg, E., Bracha, G., & Tanimoto, S. (2016). Liveness becomes Entelechy-a scheme for L6. *Second International Conference on Live Coding (ICLC 2016)*. Second International Conference on Live Coding (ICLC 2016), McMaster University, Canada.

Gajos, K., & Weld, D. S. (2004). SUPPLE: automatically generating user interfaces. *Proceedings of the 9th International Conference on Intelligent User Interfaces*, 93–100.

Green, T. R. G., & Fetais, N. (2016). How Notations Are Developed: A Proposed Notational Lifecycle. *Product Lifecycle Management in the Era of Internet of Things*, 659–671.

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.

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). *Comprehensibility of visual and textual programs: A test of superlativism against the "match-mismatch" conjecture*. http://dx.doi.org/

Jones, S. (2023, January 20). *Coding Standards for ChatGPT*. Medium. https://blog.metamirror.io/coding-standards-for-chatgpt-ef25f0d4f6d8

Kato, J., & Shimakage, K. (2020). Rethinking programming “environment”: technical and social environment design toward convivial computing. *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*, 149–157.

Miñón, R., Paternò, F., Arrue, M., & Abascal, J. (2016). Integrating adaptation rules for people with special needs in model-based UI development process. *Universal Access in the Information Society*, *15*(1), 153–168.

Myers, B., Hudson, S. E., & Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, *7*(1), 3–28.

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press.

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. In *arXiv [cs.SE]*. arXiv. http://arxiv.org/abs/2302.06590

Tanimoto, S. L. (1990). VIVA: A Visual Language for Image Processing. *J. Vis. Lang. Comput.*, *1*(2), 127–139.

How Developers Extract Functions: An Experiment

Alexey Braver
Dept. Computer Science
The Hebrew University
alexey.braver@mail.huji.ac.il

Dror G. Feitelson
Dept. Computer Science
The Hebrew University
feit@cs.huji.ac.il

Abstract

Creating functions is at the center of writing computer programs. But there has been little empirical research on how this is done and what are the considerations that developers use. We design an experiment in which we can compare the decisions made by multiple developers under exactly the same conditions. The experiment is based on taking existing production code, "flattening" it into a single monolithic function, and then charging developers with the task of refactoring it to achieve a better design by extracting functions. The results indicate that developers tend to extract functions based on structural cues, such as `if` or `try` blocks. The extracted functions tend to be short, but not necessarily due to an explicit desire to create short functions, but rather because of other considerations (like keeping logic cohesive or making functions easily testable). Finally, while there are significant correlations between the refactorings performed by different developers, there are also significant differences in the magnitude of refactoring done. For example, the number of functions that were extracted was between 3 and 10, and the amount of code extracted into functions ranged from 37% to 95%.

1. Introduction

It is generally accepted that programming should be done subject to a software design. The design can have a significant impact on the continued development and maintenance of the software. A good design can save a lot of effort for the developers, and as a result decrease the development time of the product and facilitate faster response to evolving needs. It is common to divide software design into two main parts:

- *High-Level Design:* The overall design of the product showing how the business requirements are satisfied. This is created by the software architect, and consists of dividing the functionality into modules and defining interfaces between them.
- *Low-Level Design:* The detailed design of the code based on refining the high-level design. Mostly, this is created on the fly by the system's developers.

A core activity in low-level design is to define functions. Functions are the basic elements for structuring code. Creating functions is one of the first things that one learns when learning to program. Famous coding guides like Bob Martin's *Clean Code* (Martin, 2009) and Steve McConnell's *Code Complete* (McConnell, 2004) devote full chapters to writing functions. Martin, for example, opines that functions should be short, but also notes that this is based on experience and that there is no research to support it (Martin, 2009, pg. 34). And indeed, there has been relatively little research about functions and how they are defined. In particular, we have found no controlled experiments on this topic. A possible reason is that developers enjoy considerable freedom during low level design. Due to this freedom, it is impractical to conduct an experiment just by giving a coding task to the participants, and asking them to implement it, because it can be very difficult to compare the results.

Our solution to this problem is to base the experiment on refactoring a given code-base, rather than asking participants to produce new code. And while we do not answer Martin's explicit question concerning the best length for functions, we do take a first step towards showing what sizes developers prefer and how function creation can be studied experimentally. Our main contributions in this work are:

- To devise a methodology for studying function creation by different developers, at least in the context of refactoring.
- To demonstrate that code structure provides cues for function extraction.
- To find that developers tend to use short functions even if this is not their explicit goal.

2. Background and Related Work

As noted above, Martin in *Clean Code* claims that functions should be as small as possible (Martin, 2009). A survey of practitioners indicates that there is wide acceptance of this suggestion (Ljung & Gonzalez-Huerta, 2022). Martin goes on to claim that the block of code within if statements, else statements, while statements, and so on should be one line long — and probably this line should be a function call with a descriptive name. This approach emphasizes explanation and documentation of the code as the reasons for creating functions. A similar stand is taken by Clausen, who uses the rule that functions should be no more than 5 lines of code as the title of his book on refactoring (Clausen, 2021).

Conversely, McConnell writes that functions which implement complex algorithms may be allowed to grow organically up to 100–200 lines, and that this is not expected to cause an increase in defects (McConnell, 2004).

The only study we know of which specifies a concrete optimal function size is Banker et al. (Banker, Datar, Kemerer, & Zweig, 1993). This 30 year old study analyzed the maintenance of large Cobol projects, and found the optimal procedure size to be 44 statements. McCabe suggested that functions with a cyclomatic complexity above 10 should be refactored, as otherwise they may be too complex to test effectively (McCabe, 1976). Fenton and Neil argue against the "Goldilocks conjecture", that there is an ideal size that is not too small and not too large, but this was about module size, not individual function sizes (Fenton & Neil, 1999).

Several studies have found that code complexity is indeed related to scope (Banker et al., 1993), and some have claimed that this is such a strong correlation that length is the only metric that is needed (Herraiz & Hassan, 2011; Gil & Lalouche, 2017). Landman et al., in contradistinction, claim that the correlation between code complexity and lines of code is not as high as previously thought (Landman, Serebrenik, Bouwers, & Vinju, 2016). Their view is that high correlations are the result of aggregation, and if individual functions are considered then complexity metrics provide additional information to length. This echoes earlier work by Gill and Kemerer, who propose to normalize complexity metrics by length (Gill & Kemerer, 1991). Similar disagreements have been voiced about the size of modules and classes (Fenton & Neil, 1999). El Emam et al. also argue against the claim that small components are necessarily beneficial, and fail to find evidence for a threshold beyond which large classes cause more defects (El Emam et al., 2002).

Refactoring is used to improve the structure of code without changing its functionality (Fowler, 2019), and reflects a perception that the code is "dirty" and its quality can be improved (Allman, 2012; Tom, Aurum, & Vidgen, 2013; Zabardast, Gonzalez-Huerta, & Šmite, 2020). One of the most basic actions performed during refactoring is function extraction, that is turning a block of code into a separate function (Murphy-Hill, Parnin, & Black, 2012; Tsantalis & Chatzigeorgiou, 2011; Tsantalis, Chaikalis, & Chatzigeorgiou, 2018; Hora & Robbes, 2020; Liu & Liu, 2016). There has been extensive research on tools to perform function extraction semi-automatically (Maruyama, 2001; Komondoor & Horwitz, 2000, 2003; Ettinger, 2007; Haas & Hummel, 2016). These are usually based on the identification of *program slices* — subprograms responsible for the computation of the value of some variable (Weiser, 1984). Data collected on using refactoring tools indicates that function extraction is one of the most common types of refactoring. Half of all refactoring done with JDeodorant were function extractions (Tsantalis et al., 2018). An analysis of more than 400 thousand refactorings by Hora and Robbes found that 17% were method extraction (Hora & Robbes, 2020). But we know of no research about how developers actually perform function extraction, especially when not using tools. Our experiment is designed to explore this issue.

3. Research Questions

We are motivated by very basic questions in programming, and specifically by the question of function length. For example, how does function length interact with testing, with the propensity for having defects, and with understanding the whole system? These questions are interesting and important, but complicated, because myriad factors and considerations may affect the design of code. They are also hard to study experimentally, because if we give developers a non-trivial programming assignment, which includes enough logic to get different designs, they can come up with very different designs that are hard to compare (and in fact, multiple version are also possible with very simple assignments (van der Meulen & Revilla, 2008)). In addition, the answer may be ill-defined, as it may depend on the specific developers involved in writing and maintaining the code, who could be different from each other and from other developers.

To make some progress we therefore start with much more limited and concrete questions. We focus on function extraction rather than the whole issue of writing functions. Within this context we ask

1. How do developers extract functions?
2. Do developers agree regarding how to refactor code by extracting functions?
3. What are the considerations that developers apply when deciding about functions?
4. Do developers conform with the recommendation in *Clean Code* to use very short functions?

To answer these questions we start from a given code-base, and conduct an experiment in which developers are asked to refactor this code. As they all start from the same base, it is then straight-forward to compare their results. We then also ask them to answer a short survey about their considerations.

4. Methodology

Our experiment was based on real production code. This code was modified to make it monolithic by inlining all the functions. The experiment participants then refactored it, and we analyzed the resulting codes.

4.1. Code Selection

To make the experiment as realistic and relevant as possible we decided to use real production code as the basis. The main consideration was that it be a general utility that can be understood with reasonable effort. It also had to be long enough, and have enough logic, so that it would be reasonable to expect some variations in the results. In other words, we needed to avoid code that could be arranged in only one reasonable way.

The code we chose is from file adapters.py in the Requests http Python library[1]. This module allows to send HTTP/1.1 requests easily. The code was originally 534 lines long including comments and blank lines. Excluding blank lines leaves 434 lines, and also excluding comments leaves only 286 lines of actual code. After we found a suitable code, we performed a pilot and sent the experiment to 4 developers. The results we received indicated that the experiment works.

4.2. Code Flattening

We call the procedure of converting the code from many functions to one function "flattening". The code was flattened manually in the following way:

1. One function was chosen to be the main function (send). This function was the biggest function in the original code as well. It included the main logic and calls all the smaller functions.

[1] https://github.com/psf/requests/blob/master/requests/adapters.py

2. Every function which was only defined, but not called internally, was removed. These were functions which were only called from other classes. We decided to remove them since it would be unreasonable to include unused code in our monolithic function.
3. The remaining functions were inlined by replacing calls to them with the function's code. The names of each function's parameters were replaced with the names of the arguments the function was called with. In one case we also needed to change a function name that was the same as that of a function from a different module (changed `send` to `send_the_request`).
4. All unrelated classes and imports were removed. There were some additional classes which were only defined but not used. We remove them to prevent the code from being too long and cluttered.
5. All the comments in the original code which were added by its developers to describe some flow in the code, and the main function definition, were left as they are. We wanted the participants to experience the code as close to the original as possible. But the header comments of the inlined functions had to be removed, since there was no suitable place for them.
6. All calls to third-party functions remained as they are. We wanted the participants to refactor the code in a specific context, and not in such large context as all the Python language. So we inlined only the project's functions, and left calls to functions from other sources.

All these changes reduced the total length of the code to 298 lines including blank lines and comments, and 208 lines of actual code. This is the code the participants received to refactor.

4.3. Experiment Execution

The experiment was created with Google Forms. However, Google Forms does not allow to upload files anonymously (it requires to login to upload, and the email of the uploader is included in the results). As we wanted to avoid collecting any personally identifying information, we used the services of Formfacade, which integrates with Google Forms and allows to add various functionalities including anonymous file upload. The form contained an introduction and 3 sections as follows:

1. Initially we informed the participants that the experiment and the questions are not mandatory, they can quit the experiment whenever they want, and that no personal information is collected — the experiment is completely anonymous. The introduction page also informed them that by moving to the next page they agree to participate under these conditions.
2. After the introduction participants were asked questions related to their background, such as age, programming education, development experience, etc.
3. For the experiment itself they got our flattened code as a link to GitHub, and were requested to download it. They could work on extracting the functions in their preferred IDE and work environment, with no time limit. After they finished refactoring it, they needed to make a zip file of their code and upload it.
4. Finally, the participants were asked to complete a short survey with questions related to the code assignment they were requested to do. One question in the survey was about what they thought was the ideal length for functions. In addition we asked them to rate different considerations that may apply to function extraction refactoring.

The link to the website we created was sent to personal contacts with familiarity with programming. In addition, the experiment was published in two Reddit forums, r/SoftwareEngineering and r/Code. We wanted the participants to be as varied as possible.

We received a total of 34 responses to the experiment. Of these only 23 submitted the refactored code, which is the most important part; the 11 others were removed from the analysis. The background of the

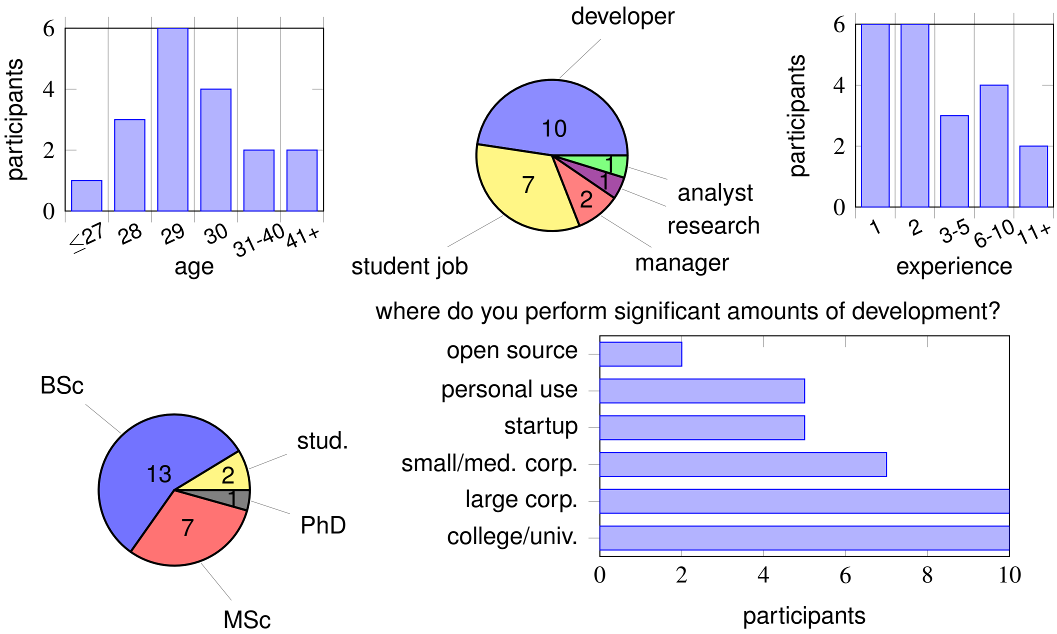


Figure 1 – Participants demographics.

participants was as follows (Figure 1). Their ages were rather focused, with 72% being between 28 and 30 years old; the oldest was 49. 51% had a BSc/BA in a computer-related field, and another 30% had an MSc; only 9% were students who had not completed their first degree yet. 91% reported that they were employed. The range of development experience in industry, that is excluding studies, was from 1 to 29 years, but for 57% it was up to 2 years. 48% were employed as developers, 33% had student jobs, and 10% held management positions. Most of the participants perform significant amounts of development in industry, in large corporations, small/ medium companies, or startups (in decreasing order). Many also code in academic institutions or for personal use. The most common programming language they felt comfortable with was Python (64%), followed by Java (50%) and C (36%). Finally, 44% reported that they use agile development practices.

4.4. Code Analysis

As noted above the code comprised 208 lines of actual code. After refactoring it was a bit longer, due to the added function headers and function calls. Analyzing this amount of code is hard and error prone. We therefore wrote a set of scripts to analyze it. Most of the analysis was done with regex matchers.

Code Stripping The first script strips the results from the irrelevant code parts such as whitespaces and blank lines. The purpose of this script is to align all the results and make them comparable by changing only the way the code was presented, but not the content. The script works in the following way:

1. Remove all the import statements.
2. Remove all the comment lines, and comment blocks demarcated with """ on the first and last line.
3. Remove all blank lines.
4. If a statement was divided into multiple lines, they were joined into one line. Specifically, if there is an opening bracket, ‘(’, but no closing one, ‘)’, all the following lines were concatenated to the initial line until the line with the closing bracket.

5. Remove all the white space at the beginning and at the end of each line. Since Python is a language where indentation is part of the syntax, this step must be done last. Its purpose is to make the comparison between the lines independent of the exact indentation used.

Lines Identification The code submitted by the experiment participants needs to be compared to the original code. This comparison was based on comparing individual lines. For instance, to identify which part of the participant's code was originally in a function, we needed to compare each line in the participant's code to the original code. At first, we compared the lines just by using the Python "==" operator after we cleaned the white spaces, but it was not good enough. For example, where were many false negatives where the participants changed a variable name, changed the argument name the function receives, or just added an underscore.

Because of these problems, we switched to comparing with SequenceMatcher from the difflib Python library. This provides a similarity ratio between 0 and 1. We could then choose a threshold to identify lines with small changes as equal, and lines with more changes as different. The threshold we used for identifying lines extracted into functions was 0.95. The threshold for noting lines that were in the original code but were not found in the results and were probably changed was 0.75 (these are the lines denoted by black dots in Figure 2 below).

Functions Identification To compare code refactoring by extracting functions we need to identify the functions in the results. This was done by a script which counts the indentation and looks for the def keyword which defines the start of a function. When a def is found, the string after it up to the character '(' is taken as the name of the function. The body of the function is all following lines with a larger indentation.

Once all the lines in a function had been identified, we retrieve from the original code the range of lines that the function was extracted from. First the line which starts the function is checked for its line number in the original code. If the line is not found in the original code, the next line is checked. This process continues until a line appears in the original code. After that, the line which ends the function is checked for the line number in the original code using the same process. If the line is not found in the original code, the previous line is checked until a line appears in the original code. The range is then determined by these lines, and we verified that it matches the length of the function that was extracted. Overall there were a total of 61 lines that were skipped in this way because they were not found in the original code. Most of these (57) were the return lines which ended the extracted functions. Obviously, these lines did not appear in the flattened code, because the flattened code contained only one main function. The remaining 4 lines were not identified due to extreme changes that were made by the participants, or were completely new lines they decided to add.

We also checked if the lines' order was changed relative to the original code. This is done by running over all the functions and for each function checking if there are 2 lines which appear in different order than they appear in the original code. This is done by looking at every pair of successive lines, retrieving their lines numbers in the original code, and validating that those numbers are also in the same order relation as the lines in the function. In all the results we had only one case where the order of 2 lines was changed.

Manual Analysis A manual scanning of the results was also done. We did this for two main purposes:

1. Verify the scripts: we wanted to do some manual testing to make sure the scripts are accurate.
2. Looking for patterns that can't be seen in the scripts: We tried to retrieve more information about the function extraction by understanding what were the participants' main considerations for the refactoring.

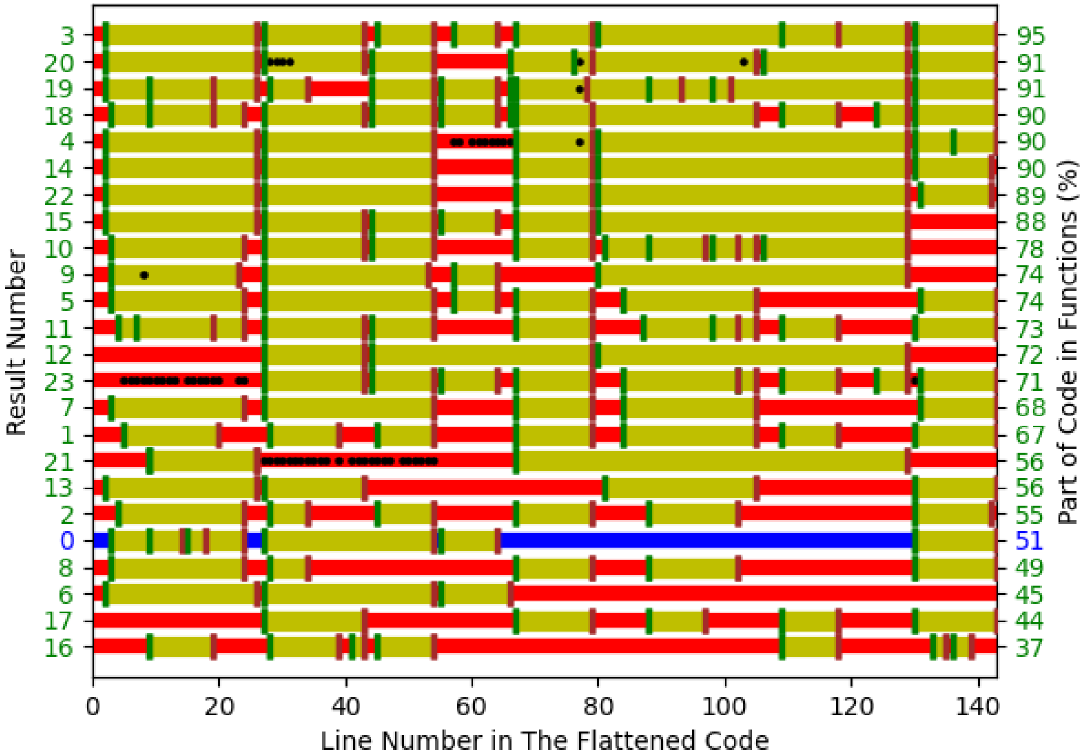


Figure 2 – Visualization of extracted functions locations relative to the flattened code. The horizontal dimension represents location in the flattened code, with the yellow-olive segments showing lines of code that were extracted to functions. Green and brown lines indicate function start and end, respectively. Each horizontal bar represents the refactoring performed by one participant. Result 0 (in blue) is the original code before flattening.

5. Results and Discussion

5.1. Function Extraction Results

The participants in the study were requested to provide us with two inputs: the refactored code, and answers to a set of questions. We begin with the code analysis.

We analyzed the coding task results both manually and automatically with scripts as described above. Figure 2 shows a bird's-eye view of all the function extractions by all the participants, compared with the original code. The horizontal dimension represents the lines of code in the original flattened code, after excluding all the blank and comment lines. The range is from the first line at the left to the last line, line 143, at the right. This is less than the original 208 lines of actual code because in the original layout arguments to functions were spread over multiple lines, one each, for readability. As we noted above, in the analysis we unified such argument lists in a single line.

Each horizontal bar represents one experimental subject. The red segments are lines of code that were left in the main function. The yellow-olive parts are ranges that were extracted to functions. The numbers on the left are the participants' serial numbers in the results data. They are sorted by the fraction of the code that they extracted into functions, as noted on the right-hand side. The blue line, with serial number 0, is the original code before it was flattened. The black dots represent lines in the original flattened code that did not appear in the results, meaning these lines were deleted, replaced by new lines, or changed too drastically to allow the comparator to find them.

Looking at the figure, we find that *no two participants produced exactly the same refactoring*. However, it is immediately obvious that function extraction is often correlated across many participants. Starting to analyze this data, we find that all the participants extracted between 3 and 10 functions, with just

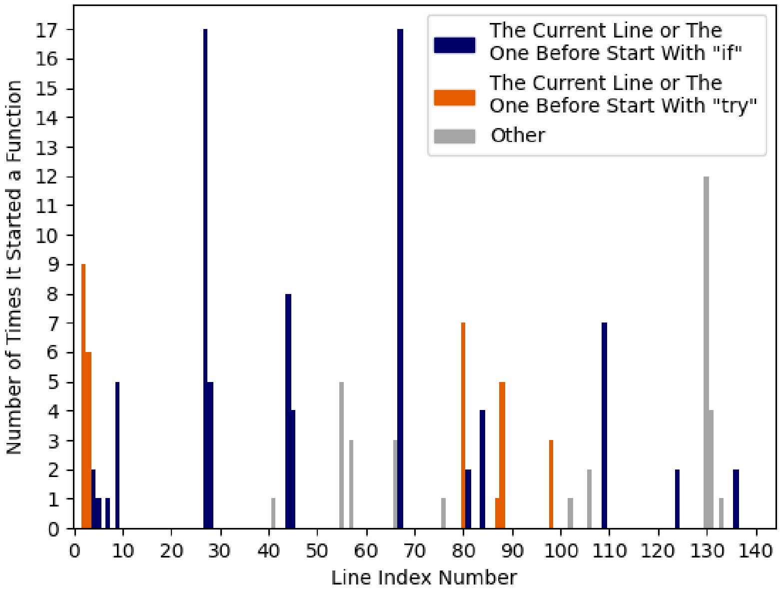


Figure 3 – Lines that start functions by type.

over half extracting 5–7 functions. (The original code had 6 functions.) In many cases the end of one function is closely followed by the beginning of another. But in many other cases functions are separated by blocks of code that remained in the main function. Notably the line at the beginning of the whole code was always left out, which is expected: it is the def of the main function.

The lines that start functions are shown in Figure 3, colored by the code construct which appears in that line (or adjacent to it): if in dark blue, try in orange, and other in gray. As we can see the vast majority of functions start on an if (77 of the 142 functions defined by the experiment participants) or try (31 functions). In addition, near the end of the code there is a block where the response object is created, and many of the participants extracted this block of code to a function. These are the functions that start on line 130 (including the creation of the object itself) or line 131 (only the population of the object's data members).

Our conclusion is that the main triggers for extracting functions in our experiment are coding constructs. This could be related to the Python syntax, which requires such constructs to be indented. The start of a new block of indented code may then be taken as an indication to extract a function. Notably, our results resonate with the recommendations made by Martin in *Clean Code* (Martin, 2009): that the body of an if should be one line long and specifically a function call, and that try-catch blocks be extracted to separate functionality from error handling.

Figure 4 shows the lengths of the functions the participants chose to extract from the flattened code (all the functions extracted by all the participants, but not the main function). As we can see in the figure, 80% of the extracted functions had 5–30 lines of code, and 55% of the functions had a length of 10–14 lines. This result is discussed below in connection to the survey question about the ideal function length.

Returning to Figure 2, we can see the majority of the participants extracted at least 70% of the code to functions. The reason may be a general understanding that code should be divided into functions and not be structured as one big main function. But on the other hand there might be a bias due to the definition of the experiment. The experiment instructions were to extract functions, and this might cause the participants to perform more function extractions than they would otherwise. In addition, if we compare the results to the original code, the original code had only 51% of the code in functions, placing it fifth from the end in this parameter. This reinforces the conjecture that there may be a bias because of the experiment definition.

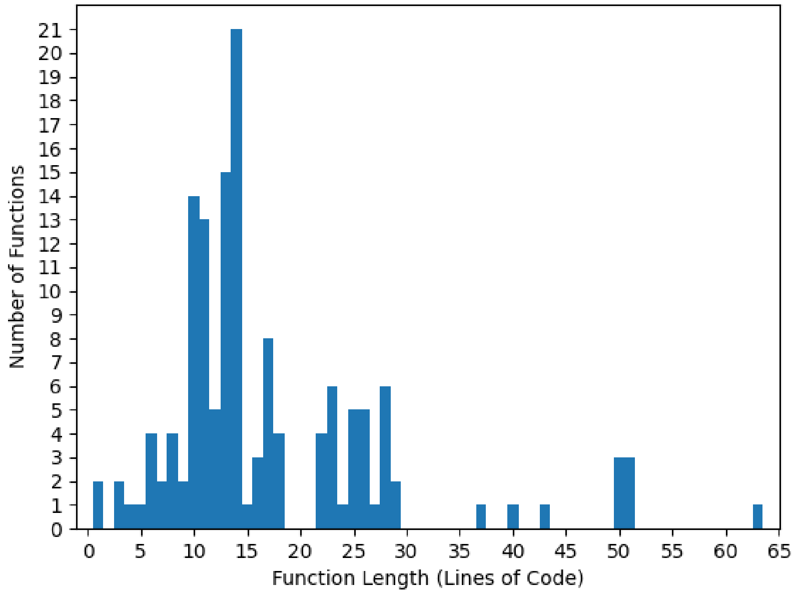


Figure 4 – Histogram of function lengths.

A very important observation is the diversity in the fraction of code extracted to functions. This ranges from a low value of extracting only 37% of the code lines, leaving nearly two thirds of the code in the main function, to a high mark of extracting no less than 95% of the code, leaving nearly nothing in the main function. And between these extremes we see a wide distribution of values. This testifies to differences of opinion between the experiment participants regarding what parts of the code should be extracted to functions.

To get a better picture of these differences, we identify 6 blocks of code that were the main candidates for extraction:

1. Lines 3 to 26, extracted (at least partially) by 20 participants and in the original code. This is a big try block surrounding the creation of the connection for sending the request.

2. Lines 27 to 54, extracted (at least partially) by 22 participants and in the original code. This verifies the SSL certificate, and contains two big if blocks, which were extracted together or separately.

3. Lines 55 to 64, extracted by only 9 participants, but also in the original code. This requests the URL and contains a smaller if.

4. Lines 65 to 79, extracted by 19 participants, but not in the original code. This is a large if block that checks whether a timeout is needed. It was nearly always extracted in exactly the same way.

5. Lines 80 to 129, extracted (at least partially) by 22 participants but not in the original code. This large block of code is where the actual sending is done and the response received. But these operations may fail for myriad reasons, so they are surrounded by three nested try-except blocks and error handling code. There was especially high variability concerning what parts of this to extract, reflecting different opinions regarding whether to extract the full largest try-except block as one large function, or to extract only some internal parts from it.

6. Lines 130 to 143, extracted by 17 participants and in the original code. This is code that creates the object to return, and it was nearly always extracted in exactly the same way.

Interestingly, three of these code blocks (numbers 1, 2, and 6) correspond to extraction types identified by Hora and Robbes as the most common, which are for creation, validation, and setup (Hora & Robbes, 2020).

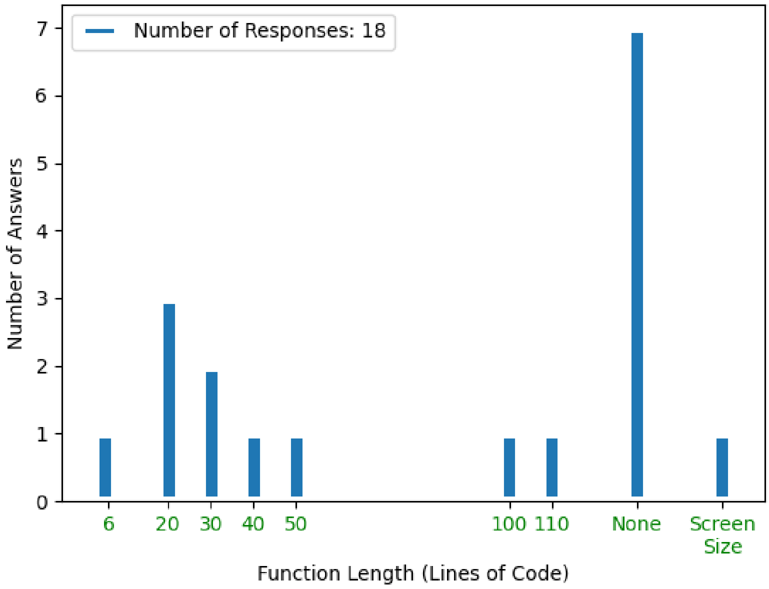


Figure 5 – Responses to the question concerning the ideal length of functions.

To summarize, the variability in the amount of code extracted to functions stems from different opinions on whether certain blocks should be extracted, whether to include or exclude nested constructs, and whether to include or exclude a surrounding construct (e.g. include the if itself or only its body).

Note that part of the diversity concerns try-except blocks and error handling code. This reflects the fact that indeed there are different ways and designs to handle errors. Some believe that error handling logic should be located in conjunction with the functions which suffered the errors, while others believe that it should be separated such that all errors are handled together. It is worthwhile to mention that in the original code this error handling code was not extracted to a dedicated function.

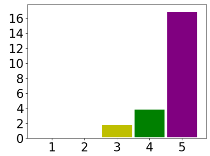
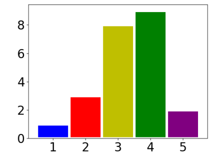
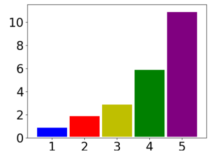
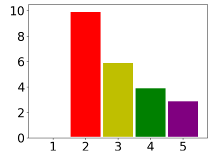
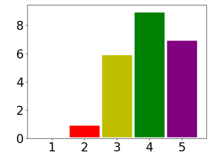
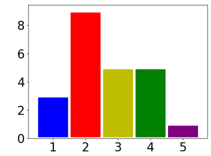
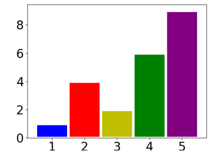
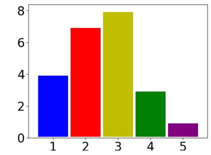
5.2. Survey Results

After the code assignment the participants were asked questions about the considerations involved in function extraction. We started with asking them about the "clean code" approach to code development. 61% of the participants indicated that they were familiar with this approach. And of those that were familiar, 93% said they agreed with the principles of clean code and strive to act on it (answers of 4 or 5 on a scale of 1 to 5).

The next question concerned the ideal function length. The responses we received are shown in Figure 5. As we can see the most common answer was that there is no ideal length of function, but from those who did provide a number most of the responses were 20–30 lines of code. Comparing this to the results of the actual functions they extracted, shown above in Figure 4, we find that the actual functions were even shorter than the ideal length the participants stated they believe in. Interestingly, the actual results agree with the recommendation that functions should usually have no more than 20 lines, suggested by Martin in *Clean Code* (Martin, 2009, p. 34).

We then asked about the participants' main considerations when extracting new functions. The distribution of answers for the different considerations we suggested are shown in Table 1. The scale was from 1 = Not important to 5 = Very important. As we can see, the most popular consideration was "Making each function's logic cohesive" with an average score of 4.62/5. The second most popular was "Making each function do only one thing", with an average score of 4.10/5. The consideration that corresponds to our observation in Figure 3 is the third most popular consideration — "Separating code control blocks", with an average score of 3.93/5. We can't really measure the actual difference in the importance between these considerations by looking at the function extraction results, because "Making each function's logic

Table 1 – Main considerations for function extraction

Consideration	*Mean*	*Histogram*	*Consideration*	*Mean*	*Histogram*
Making each function's logic cohesive	4.62		Following design patterns	3.41	
Making each function do only one thing	4.10		Not needing to pass too many arguments	3.10	
Separating code control blocks	3.93		Making functions as short as possible	2.72	
Making functions easily testable	3.89		Making each function close to the ideal length	2.65	

cohesive" and "Making each function do only one thing" are quite an abstract considerations, and incomparable to the "Separating code control blocks" consideration which can be measured by counting the lines that started a function on a new block of code. Interestingly, the two considerations concerning function length, "Making functions as short as possible" and "Making each function close to the ideal length", were ranked as having lower importance than all other considerations.

As this questionnaire appeared at the end of the experiment, after the participants had submitted their refactored code, we also asked whether seeing the survey questions might have changed the way they refactored the code. 48% answered yes, possibly implying that they did not think about what they had done deeply enough. The main consideration cited in this context was to make the extracted functions more testable; this was mentioned by 4 participants. Several issues were mentioned by two each: using design patterns, the number of arguments, separation into classes or files, and the length of the functions.

5.3. Discussion

One of our main results is that participants in the experiment tend to extract functions based on the blocks of code defined by existing coding constructs, such as if and try. They also gave the consideration of "Separating code control blocks" relatively high scores in the survey. Using blocks of code as cues for function extraction is reasonable because such blocks indeed encapsulate separate sections of the computation. However, we note that extracting functions by looking at blocks of code is also the easiest way to extract. It doesn't even require the participants to understand the code's logic.

Another interesting observation is the difference between the approaches taken by tools and by human developers. As we noted above in the related work section, tools are often based on extracting program slices (e.g. (Maruyama, 2001; Tsantalis & Chatzigeorgiou, 2011)). Program slices are defined by the data flow in the program: they include instructions that may affect the value of a certain variable. This is useful for code analysis, and to ensure that the extracted code is self-contained and independent, so the program will retain exactly the same behavior after it is extracted. But for humans, data flow is not easy to follow in programs written in conventional procedural languages (C, Java, Python). Therefore the participants in our experiment preferred to extract functions based on easily-observable coding constructs, which reflect control flow. It may be interesting to contrast these approaches in terms of the similarities and differences between the refactored codes they produce. We leave such a study to future work. In the meanwhile, we note that our results resonate with those of Pennington from 25 years ago, who found that professional programmers base their mental representations of programs on control flow, and not on functionality (Pennington, 1987).

An interesting issue is the question of ideal function length. As we noted, Martin famously makes the case that functions should be as short as possible (Martin, 2009, p. 34). Given that 61% of our participants acknowledged that they are familiar with the "clean code" discipline, this might be the reason that 80% of the functions were 10-30 lines long, and 55% had 10-14 lines. However, in the survey we saw that "Making functions as short as possible" and "Making each function close to the ideal length" were the two least important considerations for function extraction (Table 1). Thus the participants believe that small functions and ideal functions length are not as important as at least 6 other considerations we suggested to them. But in fact they do extract small functions. This is quite an interesting result, because it means the participants extracted small functions without explicitly thinking about then in this way. We might carefully conjecture that nowadays developers extract small function as part of their coding skills. Another possibility is that other considerations, like keeping the logic cohesive and doing just one thing, naturally lead to shorter functions. So short functions are not an end in themselves, but they happen to be the solution to other goals.

6. Threats to Validity

As in any experiment, we had some difficulties and threats to the validity of our results. The following threats are the most significant remaining ones.

6.1. Methodological Difficulty

The problem of studying function creation is a difficult one. Ideally we would like to observe and compare developers creating functions when programming to the same specification. However, the diversity of programs that can be expected is huge, including in how exactly the problem is solved and how the solution is divided into functions. Nevertheless, analyzing such datasets (e.g. (van der Meulen & Revilla, 2008)) is an interesting direction for future work.

Another possible option is to start with "born-flat" code, that was written without being divided into functions, and ask that it be refactored. This may be extremely hard, because in such code the logic for different functionalities may be intermixed, and global variables may be used to communicate state. In such a situation splitting the code into functions is not representative of writing new functions. We used flattened code that was derived from production code including functions. This suffers from the drawback that the signature of the original functions may be observable in the code, and may have contributed to the correlation between the refactorings performed by different participants. However, we note that there was also significant agreement where the original code had *not* been divided into functions (lines 65–129), supporting our conclusion that structural cues may be at work.

6.2. Confounding Factors

Our experiment was explicitly about extracting functions. This may have affected the participants, who may have tried to "live up to expectations" and extracted more than they would have under real work conditions. Such behavior, if it exists, would lead to a bias in the results.

Another possible effect of an experiment environment is that it is perceived as less serious than "real" work, leading participants to invest less effort. This was noted in a few comments we received to the postings inviting participants on reddit. One wrote "Living organisms are lazy (including humans) and will usually choose the cheapest way to get out of a problem. So your experiment might tell you what's the cheapest way to reformat code". Another wrote "this is a 150 lines function, there were occasions where reformatting something like that was a task for the whole day in my job". It is also supported by the fact that nearly half the participants indicated they might have changed the way they refactored the code after seeing the survey questions. Finally, the three cases of stretches of lines that were not identified (sequences of black dots for participants 4, 21, and 23 in Figure 2) were actually lines that were deleted with the probable intention of turning them into a function, but this was then not done — which again might indicate lack of seriousness.

It is not clear how these problems could be avoided in an experiment. A possible approach is to conduct a much larger experiment, in both scope and cost, in which developers are actually hired to generally refactor a large body of code (Sjøberg et al., 2002). Such an experiment can be attempted based on our experiment's demonstration that the methodology of using flattened production code looks promising.

6.3. Limited Generalizability

Being an initial experiment in a new direction, and given the need to develop a new methodology, our experiment was rather limited. We used only one function, which raises the question of how representative this is of other codes. For example, one of the main reasons for creating functions is to reuse code, so if the code contains clones they may be prime targets for extracting. However, our code did not contain any clones, so this was not checked. In addition, our results are limited to the context of refactoring, and may not generalize to function creation in the context of writing new code.

We also had only 23 participants who submitted the refactored code. Given the need to invest around half an hour in this non-trivial task this is not a bad response. However, it is not enough to enable an analysis of the behavior of different demographic groups, as each one would have too few members. Another issue is whether our participants are representative of developers in general. For example, in our experiment only 61% of the participants indicated they know clean code, while in a survey by Ljung et al. the vast majority of the participants had heard of clean code, and tended to agree with its principles (Ljung & Gonzalez-Huerta, 2022). Again, based on our initial experiment, a larger one can be attempted.

7. Conclusions

7.1. Summary of Results

In this study we tried to make initial observations of how developers create functions, including issues like how much they agree with each other and how long are the functions they create. Our approach was to design an experiment, where we took real production code, "flattened" it, and then asked the experiment participants to refactor it by extracting functions. In addition, we asked them to answer some questions related to their considerations when extracting the functions.

Analyzing the results we found that most of the participants extracted small functions, as suggested in the "Clean Code" approach, and that the main cues for extracting functions were related to the code structure. Interestingly the actual functions were somewhat shorted than what the participants said would be ideal. Also, they didn't explicitly consider length to be a major factor, citing logical cohesion and doing only one thing as the most important considerations.

We also found that in most cases the fraction of the code that was extracted to functions was bigger than it had been in the original code. This was probably due to the fact we asked participants explicitly to extract functions. But it also implies that developers can be influenced in how they go about designing functions. This can be studied in future experiments.

We believe that in addition to these concrete results, we produced some promising methods to study such complicated issues like software design. Since this kind of experiments are almost non-existent, we hope our study and our methods will encourage additional experiments in this field. In particular, a desirable follow-up experiment is to replicate our experiment at a larger scale, and then study the quality of the refactored code produced by the experiment participants (in terms of readability, testability, etc.). It would be especially interesting to see if there is any correlation between the quality of the produced code and the considerations or approaches employed in the refactoring.

7.2. Implications

Our work has several possible implications for practitioners. The most immediate is to highlight a simple and accessible approach to function extraction. Extracting functions goes hand in hand with the teachings of procedural programming, which perhaps explains why it is one of the most commonly used types of refactoring. And by focusing on code blocks defined by constructs like loops, conditionals, or exception handling one can perform function extraction in a straightforward manner.

Extracting such blocks also provides a simple methodology to control the desired granularity, as one can continue to extract nested blocks down to the most basic blocks, or alternatively stop at a larger scope. For example, when extracting an if construct to a separate function, one can then decide whether to also extract the "then" and "else" blocks or not. If one desires to create very short functions, extracting blocks is an easy way to achieve this.

Using code blocks for function extraction is also related to program comprehension. When faced with unknown code, understanding what it does requires a bottom-up approach (Shneiderman & Mayer, 1979). This is greatly simplified if each function is short, and composed mainly of calls to other functions which have good descriptive names, as such a structure allows the readers to perceive the code at a higher level of abstraction rather than forcing them to contend with the basic constructs. Extracting blocks achieves exactly this effect.

8. Experimental Materials

Experimental materials are available at

`https://doi.org/10.5281/zenodo.7044101`

9. Acknowledgments

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18).

10. References

Allman, E. (2012, May). Managing technical debt. *Comm. ACM*, *55*(5), 50–55. doi: 10.1145/2160718.2160733

Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993, Nov). Software complexity and maintenance costs. *Comm. ACM*, *36*(11), 81–94. doi: 10.1145/163359.163375

Clausen, C. (2021). *Five lines of code*. Manning Publications.

El Emam, K., Benlarbi, S., Goel, N., Melo, W., Lounis, H., & Rai, S. N. (2002, May). The optimal class size for object-oriented software. *IEEE Trans. Softw. Eng.*, *28*(5), 494–509. doi: 10.1109/TSE.2002.1000452

Ettinger, R. (2007, Oct). Refactoring via program slicing and sliding. In *Intl. Conf. Softw. Maintenance* (pp. 505–506). doi: 10.1109/ICSM.2007.4362672

Fenton, N. E., & Neil, M. (1999, Sep/Oct). A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, *25*(5), 675–689. doi: 10.1109/32.815326

Fowler, M. (2019). *Refactoring: Improving the design of existing code* (2nd ed.). Pearson Education, Inc.

Gil, Y., & Lalouche, G. (2017, Oct). On the correlation between size and metric validity. *Empirical Softw. Eng.*, *22*(5), 2585–2611. doi: 10.1007/s10664-017-9513-5

Gill, G. K., & Kemerer, C. F. (1991, Dec). Cyclomatic complexity density and software maintenance productivity. *IEEE Trans. Softw. Eng.*, *17*(12), 1284–1288. doi: 10.1109/32.106988

Haas, R., & Hummel, B. (2016, Jan). Deriving extract method refactoring suggestions for long methods. In *Intl. conf. softw. quality* (pp. 144–155). doi: 10.1007/978-3-319-27033-3_10

Herraiz, I., & Hassan, A. E. (2011). Beyond lines of code: Do we need more complexity metrics? In A. Oram & G. Wilson (Eds.), *Making software: What really works, and why we believe it* (pp. 125–141). O'Reilly Media Inc.

Hora, A., & Robbes, R. (2020, May). Characteristics of method extraction in Java: A large scale empirical study. *Empirical Softw. Eng.*, *25*(3), 1798-1833. doi: 10.1007/s10664-020-09809-8

Komondoor, R., & Horwitz, S. (2000, Jan). Semantic-preserving procedure extraction. In *Ann. Symp. Principles of Programming Languages* (pp. 155–169). doi: 10.1145/325694.325713

Komondoor, R., & Horwitz, S. (2003, May). Effective, automatic procedure extraction. In *IEEE Intl. Workshop Program Comprehension.* doi: 10.1109/WPC.2003.1199187

Landman, D., Serebrenik, A., Bouwers, E., & Vinju, J. J. (2016, Jul). Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *J. Softw.: Evolution & Process*, *28*(7), 589–618. doi: 10.1002/smr.1760

Liu, W., & Liu, H. (2016, Aug). Major motivations for extract method refactorings: Analysis based on interviews and change histories. *Frontiers Comput. Sci.*, *10*(4), 644–656. doi: 10.1007/s11704-016-5131-4

Ljung, K., & Gonzalez-Huerta, J. (2022, Nov). "to clean-code or not to clean-code" a survey among practitioners. In *Intl. conf. product-focused softw. process improvement* (pp. 298–315). (Lect. Notes Comput. Sci. vol. 13709) doi: 10.1007/978-3-031-21388-5_21

Martin, R. C. (2009). *Clean code: A handbook of agile software craftmanship*. Prentice Hall.

Maruyama, K. (2001, May). Automated method-extraction refactoring by using block-based slicing. In *Symp. softw. reusability* (pp. 31–40). doi: 10.1145/375212.375233

McCabe, T. (1976, Dec). A complexity measure. *IEEE Trans. Softw. Eng.*, *SE-2*(4), 308–320. doi: 10.1109/TSE.1976.233837

McConnell, S. (2004). *Code complete* (2nd ed.). Microsoft Press.

Murphy-Hill, E., Parnin, C., & Black, A. P. (2012, Jan/Feb). How we refactor, and how we know it. *IEEE Trans. Softw. Eng.*, *38*(1), 5–18. doi: 10.1109/TSE.2011.41

Pennington, N. (1987, Jul). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*(3), 295–341. doi: 10.1016/0010-0285(87)90007-7

Shneiderman, B., & Mayer, R. (1979, Jun). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *Intl. J. Comput. & Inf. Syst.*, *8*(3), 219–238. doi: 10.1007/BF00977789

Sjøberg, D. I. K., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., … Vokác, M. (2002, Oct). Conducting realistic experiments in software engineering. In *Intl. symp. empirical softw. eng.* (pp. 17–26). doi: 10.1109/ISESE.2002.1166921

Tom, E., Aurum, A., & Vidgen, R. (2013, Jun). An exploration of technical debt. *J. Syst. & Softw.*, *86*(6), 1498–1516. doi: 10.1016/j.jss.2012.12.052

Tsantalis, N., Chaikalis, T., & Chatzigeorgiou, A. (2018, Mar). Ten years of JDeodorant: Lessons learned from the hunt for smells. In *Intl. conf. softw. analysis, evolution, & reengineering* (pp. 4–14). doi: 10.1109/SANER.2018.8330192

Tsantalis, N., & Chatzigeorgiou, A. (2011, Oct). Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. & Softw.*, *84*(10), 1757–1782. doi: 10.1016/j.jss.2011.05.016

van der Meulen, M. J. P., & Revilla, M. A. (2008, Nov/Dec). The effectiveness of software diversity in a large population of programs. *IEEE Trans. Softw. Eng.*, *34*(6), 753–764. doi: 10.1109/TSE.2008.70

Weiser, M. (1984, Jul). Program slicing. *IEEE Trans. Softw. Eng.*, *SE-10*(4), 352–357. doi: 10.1109/TSE.1984.5010248

Zabardast, E., Gonzalez-Huerta, J., & Šmite, D. (2020, Aug). Refactoring, bug fixing, and new development effect on technical debt: An industrial case study. In *Euromicro conf. softw. eng. & advanced apps.* (pp. 376–384). doi: 10.1109/SEAA51224.2020.00068

Pronto: Prototyping a Prototyping Tool for Game Mechanic Prototyping

Eva Krebs
Hasso Plattner Institute
University of Potsdam
eva.krebs@hpi.de

Tom Beckmann
Hasso Plattner Institute
University of Potsdam
tom.beckmann@hpi.de

Leonard Geier
Hasso Plattner Institute
University of Potsdam
leonard.geier@hpi.de

Stefan Ramson
Hasso Plattner Institute
University of Potsdam
stefan.ramson@hpi.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
robert.hirschfeld@uni-potsdam.de

Abstract

The development of video games revolves to a large extent around the *feel* of an idea. From the very beginning, developers need to be able to quickly create and try out as many ideas as possible, as assessing the feel of an idea, and thus its viability as a game, is best done through experiencing it. An approach to prototyping is essential in this regard, as it allows developers to identify promising ideas without committing too much time or resources.

To support developers in prototyping game mechanics, we created the *Pronto* framework for the Godot game engine. This framework focuses on fast, throw-away prototypes for specific game mechanics created through visual interactions mixed with code.

Pronto consists of a set common, modular concerns in games, such as moving or colliding, called Behaviors. Developers place Behaviors visually in the game scene and connect them through code also placed in the game scene to achieve their desired effect. The first version of Pronto was itself a prototype, only suitable for a very narrow range of games. To approach a still minimal, yet flexible set of behaviors that allow developers to create any kind of game, we designed a university course where students alternate between working with the framework and extending it. In the process, they iteratively identified and addressed shortcomings and potentials of Pronto.

In this paper, we present Pronto as a tool for game developers to quickly validate game mechanics ideas, as well as the process and results of the students in the seminar to improve it.

1. Introduction

Game development benefits from fast iterations (Schell, 2014, p. 94). Developers will need to tweak and play test all parts of a game many times to ensure that they both work correctly and are fun to play (Murphy-Hill, Zimmermann, & Nagappan, 2014). This experimentation is important before main development even starts; there are many ideas for potential games, but only some of those are actually feasible or fun to play. To find these promising ideas, developers seek ways to quickly and easily try out ideas to find the few they want to polish to a full game (Kasurinen, Strandén, & Smolander, 2013).

Games are a visual medium. Developers have introduced ways to effectively prototype and obtain immediate feedback for many of the visual aspects of games via direct manipulation. For example, creating a game level is often done by means of dragging and placing assets directly at the position they will later appear at in the game.

In contrast, defining behavior is usually achieved through code. To leverage the conveniences of modern development suites, the development even tends to occur in a separate application than the game engine. This separates how developers edit concrete, visible game elements from how they edit the abstract code behind the behavior of those elements. As a result, developers have to go through multiple steps to feel how changes to the code will impact the behavior of the game. For effective prototyping, however, developers benefit from experiencing the effect of their changes as quickly as possible. Games are about the experience players have while playing them; for instance, it might be important for a mechanic that allows players to bounce around the game world to feel fun, natural, and non-frustrating. In another situation a game might want to include a difficult chase sequence that partly feels stressful and frustrating on purpose. Game developers need access to these human experiences as often as possible during development.

To address the gap between authoring and experiencing, we designed a prototyping framework called Pronto, aimed at throw-away prototypes to validate ideas for game mechanics. For example, we want developers to be able to quickly try out whether a car racing game where the cars have a grappling hook for special movement is feasible and/or fun. Or whether a platformer where the player may invert gravity in certain parts of the game world is interesting, and so on. Pronto currently does not aim to be a tool for non-throw-away prototypes like vertical slices that are usually the basis for a later complete game after developers have already gained insights from a lot of smaller prototypes and ideas.

Pronto's main idea is to move code that is currently in source files, removed from the game, to the game itself, by scattering snippets of code into the visual representation of the game's scene, close to where they will manifest. Scattered code is anchored by means of composable Pronto *Behaviors*. These Behaviors are visual representations of a concern, such as collisions with other objects, moving an object around, or listening to player input, that are placed visibly in the game's scene and display relationships to the objects in the scene that are involved in their function.

Pronto's main benefit for prototyping is dependent on finding a comprehensive but not overwhelming set of composable Behaviors. Given too many specific Behaviors, developers might no longer be able to locate the desired Behaviors and their function may not be sufficiently flexible to allow for useful combinations with other Behaviors. Given too few, developers will need to resort back to writing code in source files to achieve their goals. A sweet spot would yield a limited few but powerful and expressive Behaviors that can be combined to achieve anything the game engine is capable of. To approach this sweet spot, we designed a seminar where students alternate between prototyping games with Pronto and extending or changing the framework to better suit the needs uncovered during their own work with the framework.

In this paper we give an overview of game prototyping in general as well as the game engine we used as a basis for our framework in section 2. We introduce the Pronto prototyping framework and its concepts in section 3. To demonstrate how and for what our prototyping framework can be used, we provide a walkthrough in section 4 followed by a description of and results from our prototyping seminar in section 5. We conclude this paper with a summary and discussion of future work in section 7.

2. Iterating Games

Game development usually relies on fast iterations. Developers need to tweak and try ideas to see if they work as intended and are fun to play. Particularly, developers often try many ideas as prototypes to find the few promising ideas that they want to commit to. Since resources for development are usually limited, it is important to determine quickly which ideas are feasible before starting a complete development process (Schell, 2014; Kultima, 2015).

Game developers often use game engines. Game engines are development systems that streamline game creation by providing tools and solutions for tasks that often appear during game development, thus making main parts of game development reusable. Since many games are visual, game engines usually have dedicated tools for creating and editing the visual game scene. Game engines also usually include a code editor and might provide direct support for patterns that are often needed for games, such as a game loop (Nystrom, 2014). Since game engines are a core part of game development, we integrated our prototyping framework into an existing game engine.

2.1. Godot - An Open-source Game Engine

Pronto was created for the Godot game engine. We chose Godot because it is a light-weight, open-source game engine with excellent support for custom extensions(Juan Linietsky & contributors, n.d.). Godot supports both the visual creation of game scenes as well as code editing, see Figure 1.

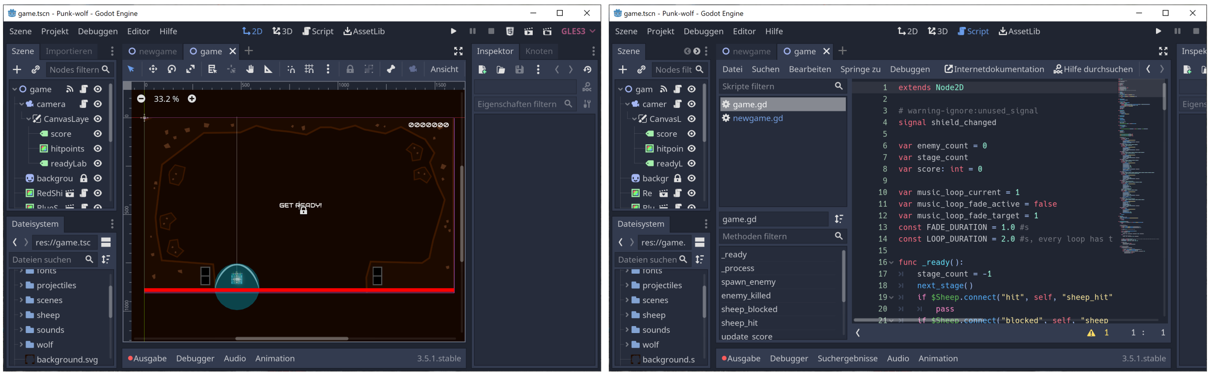


Figure 1 – The game scene editor of the Godot game engine for a 2D game can be seen on the left. The right shows the code editor of the Godot game engine for a script of the same game.

Godot includes visual scene editors for both 2D and 3D graphics. Developers can edit these scenes in an interactive, visual editor via drag and drop as well as by using specialized sidebars that provide additional editing options (e.g. setting the color of a node via a color picker). A scene consists of elements ordered in a scene tree. These tree elements are called nodes. While users can create their own nodes, Godot provides a variety of pre-defined nodes, e.g. for buttons and other visual elements. Scenes themselves can also be used as nodes. Since the scenes contain all visual information about the game, they look and feel very close to the game when it is actually run. However, behavior is defined as code in so called scripts. Scripts are attached to a specific node.

To edit a script, developers need to switch from the visual scene editor to Godot's code editor. Here, behavior of a node in the game world is defined. This editor supports many common code editing features such as syntax highlighting and autocompletion. The main language used for Godot projects is gd script, a programming language similar to Python.

Godot is extensible via an extension system. Extension developers can define custom types of node, modify editing widgets of the engine, or add additional visual cues into the game world preview. Extensions in Godot may use the same default language as scripts written for the game, allowing game developers to become tool developers with a low barrier.

3. The Pronto Prototyping Framework

The Pronto prototyping frameworks provides a way to visually and quickly create throw-away game mechanic prototypes. The framework is not designed for creating entire games; it is meant for quickly trying new, small ideas, like a rough sketch of a game mechanic.

The core idea behind Pronto is to bring code written for a game's behavior closer to the visual representation of the game. By having developers interact with visual elements in the game scene, we hope to reduce the need for context switches to a code-only view. Pronto aims to make behavior that is separate from the game scene, or "invisible", concrete and tangible directly in the scene itself. This should make it easy to create and modify parameters of game behavior.

The core concepts Pronto provides are *Behaviors* and *Connections*. Behaviors are custom Godot nodes that are a visual representation and encapsulation of a game element or aspect. Like all nodes, they can be placed and edited in the visual Godot scene editor. Even usually hidden concepts, e.g. the representation of player control keys, are now placed visually, and spatially, in the scene editor. To facilitate interaction between Behaviors, Connections can be added between them, e.g. a Connection between a *Timer* and *Spawner* could be used to make the Spawner spawn new items regularly. These Connections are also visually represented, see Figure 2.

Types of Behaviors include but aren't limited to: Behaviors that trigger events (e.g. based on time or a collision), Behavior that cause certain actions when triggered (e.g. *Move* that moves its parent node or

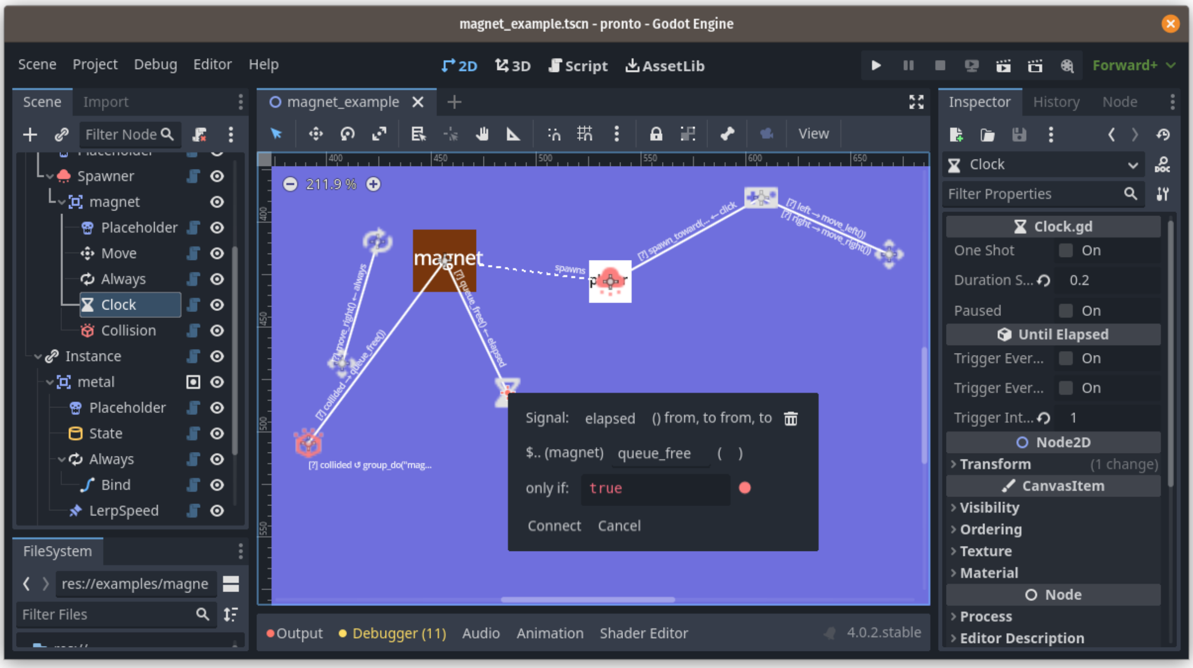


Figure 2 – The game scene editor of the Godot game engine for a 2D game including the Pronto prototyping framework. Pronto Connections are visible between game elements.

a Spawner that spawns other nodes), behaviors that contain or visualize state (e.g. Value, which can be altered via a Slider or Placeholder, a label), and Godot nodes (e.g. Area2D).

Pronto is compatible with Godot and most of its existing features. Importantly, Pronto is not designed a visual programming tool to make game programming friendlier for beginners. Instead, Connections are merely an offer to game developers to expose behavior in a visual manner, as opposed to hiding it in source code files. In particular, Connections are just standard Godot function calls, reacting to events emitted by other Nodes. Developers write code inside the Connection editor dialog that may for example transform arguments prior to the function call, as shown in Figure 2.

Of special note is that Pronto is compatible with most liveness features of Godot itself. If values are changed in the sidebar of the scene editor while the game is run, the running game will receive the updated value accordingly. Additionally, Pronto visualizes when Connections are triggered in the running game by lighting up the Connection lines in the editor to help developers isolate issues in their Connection setup quickly.

While Pronto aims to make the need for code obsolete, the compability with Godot makes possible to switch to and use the code editor if neccessary. Thus, developers are not limited in what prototypes they can create by our framework.

4. Prototyping with Pronto

The following walkthrough details how Pronto can be used to build a small game mechanic. The example we want to build is part of a racing game. The player needs to be able to control a car and use it to collect items. The finished game can be seen in Figure 4. While not detailed as part of this walkthrough, there are two labels in the game for displaying the current score and fps during testing.

We will first prepare the items that we later want to collect. An item is represented as an *Area2D*, a Godot node that reports collisions. To this main item node we add a *Placeholder*, a Pronto Behavior node that displays a rectangle with a label in the game, in this case "pickup" (top-right of Figure 4).

To spawn items, we add *Spawner* and a *Clock* Behavior to the scene. We add the item is as a child of the

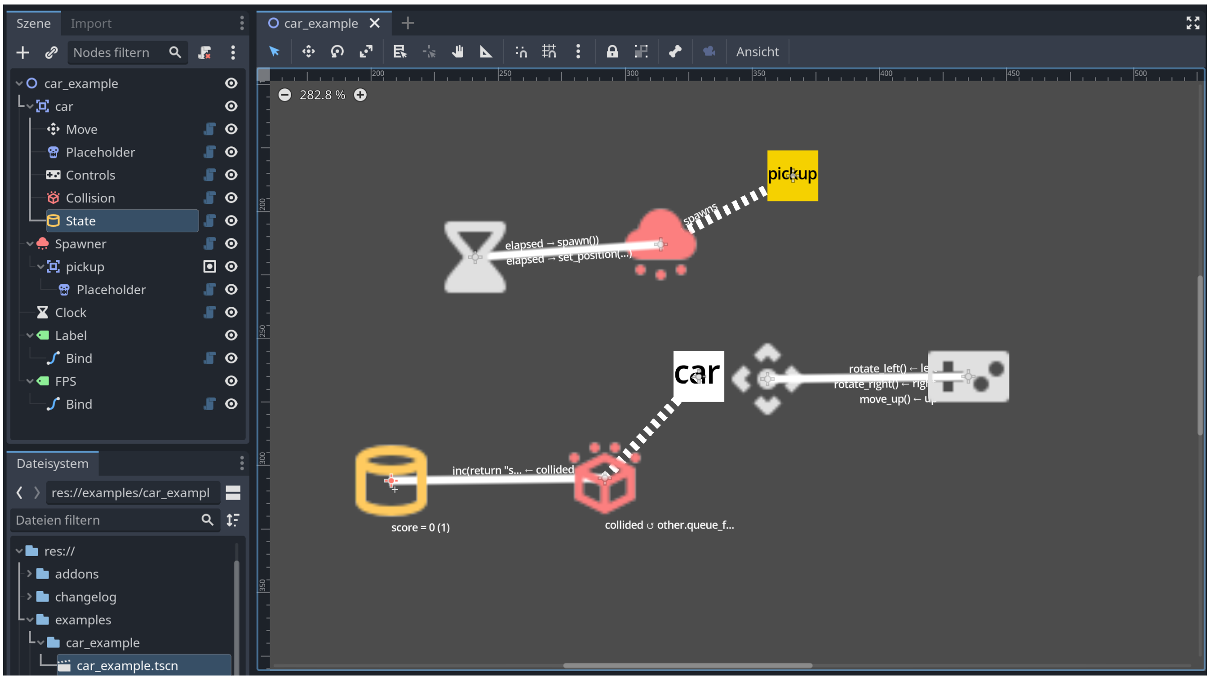


Figure 3 – The example game opened in the scene editor. The Clock *timer is connected to the* Spawner *which in turn is connected to the "pickup" item it spawns. The car node is connected to a* Collision*, while the* Move *child node of the car is connected to the* Controls.

Spawner in the node tree, which instructs the *Spawner* to hide the item on game start and create a visible duplicate of it when called. We then add two connections between the *Clock*, acting as a timer, and the *Spawner*. The first connections triggers the *Spawner* and the second moves it to a random position on the screen. We configure the *Clock* to trigger every five seconds.

The second game element we need is the car itself. The car is another *Area2D*. We again add a *Placeholder*, displaying the label "car", as a child (bottom-center of 4). We then add several more children; Move, Controls, Collision, and State.

To make our car controllable by the player, we add connections between Move and the Controls for the *up*, *left*, and *right* triggers. The *Move* behavior moves its parent node when triggered.

The car and *Collision* are then connected. When triggered, we remove the collidee, which in this case is the item. We also connect the *Collision* node with our Pronto *State* node that in this case saves the score of our player. Whenever a collision is triggered, the score is incremented (bottom-left of Figure 4).

Now that the basic setup for a simple racing game is complete, developers may start experimenting with parameters of their setup. For example, they may change the frequency in which coins spawn or the speed of the car.

5. Prototyping Pronto

The following example use cases are from a prototyping seminar we designed with Pronto in mind. The seminar is aimed at graduate computer science students and teaches both game mechanics prototyping as well as working and modifying the tools and frameworks that participants use for development. With Pronto as its subject, the students in the course iteratively and collaboratively extended the scope of games that could be created.

Students worked in small teams; two students per team were encouraged, but they could work on their own if preferred. Nine graduate students participated in the seminar, in five teams. In order for students to create many prototypes, we structured the seminar into several short iterations. Each iterations has

Figure 4 – The example game while running. As some time has passed, some "pickup" items have been spawned. The player can steer the car with their keyboard.

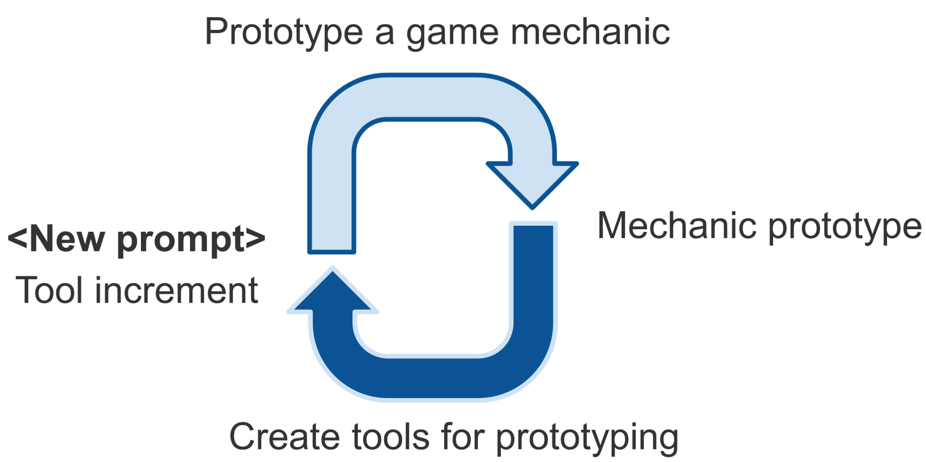


Figure 5 – The two phases of the seminar. In the first phase, students build a game mechanic based on the prompt. In the second phase, students work on Pronto itself.

two phases, each a week long, which is visualized in Figure 5. In the first week, students prototype a new game mechanic. In the second week, students modify Pronto based on insights gained in the previous week. After each phase, students hand in a changelog containing a short video and text detailing what they build and their insights, which is also used to inform the other teams of changes in the use of the framework.

The first phase starts with the students receiving a prompt, e.g. "Build a mechanic for a Terraria-style game"[1] during the weekly meeting. Teams then get to discuss briefly what mechanic they would find interesting. In particular, we encouraged students to consider whether their chosen mechanic would work well in a paper prototype. If so, they are encouraged to look for a different idea that would rely more strongly on the possibilities provided by Pronto. Once chosen, the team presents their plan and the group discusses its applicability and further directions.

Each team then implements their own game mechanic. In this phase each team works on separate branches of Pronto. Students may deliberately break functionality of Pronto or extend it in non-idiomatic ways in this phase. The only goal is to build a prototype of a mechanic as quickly as possible and to obtain feedback on it. The changelog for this phase includes a demo of the mechanic prototype, which parameters or variations were tweaked and considered to be especially interesting during testing, and changes to the Pronto framework that they had to hack in or would like to see to better support their workflow in the previous week.

The goals for the second phase are based on the feedback and change requests to the framework from the previous phase. During a meeting with all teams and the course instructors, changes are discussed. Desired changes are prioritized and feedback from all teams concerning its rough design is gathered. Each team then picks framework modifications from the discussed list and implements the modifications directly on Pronto's main branch. Teams communicate breaking changes through the seminar's group text channel. The changelog at the end of this phase contains a demonstration of how to use the new framework modification as well a textual description of API changes.

The seminar took place during the summer of 2023.

5.1. Phase 1: Building Grappling Hooks with Pronto

For the prompt "Build a mechanic for a Terraria-style game", one team implemented a grappling hook mechanic. The player shoots out a grappling hook that collides with walls and allows the player to move along the hook's line once attached. The team tested three variants of grappling hooks.

The first grappling hook variant is a teleportation-based grappling hook. The player can teleport directly to the hook after it landed, which can be seen in Figure 6. The second variant was considered a "static" grappling, the player can move on a straight line between their original position and the hook. The third and final variant can be considered a "classic" grappling hook; the player can swing around as if connected by rope to the hook. The team experimented with velocity and reach of these variants.

Based on their implementation, the team proposed several changes or new features for Pronto. Two proposals were especially positively received by other teams: the first would extend the list of triggers supported by the *Controls* behavior for mouse drag inputs. Teams had encountered multiple times that they wanted to react to the mouse cursor moving only while a mouse button was held. The proposal would allow to do achieve this without having to keep track of mouse button states manually.

The second proposed change would introduce a new Behavior to be added to Pronto to let developers render a line between two elements in the game, which otherwise required referencing two nodes in code and continuously repositioning and rotating a thin rectangle according to the nodes' positions. The new Behavior instead introduced a new basic component that would realign automatically based on other nodes' positions, for example to easily visualize a grappling hook, laser, or the next target of an enemy during testing. The implementation proposed by the team would only fulfill the basic needs so

[1] `hhttps://terraria.org/`, accessed: 2023-05-31

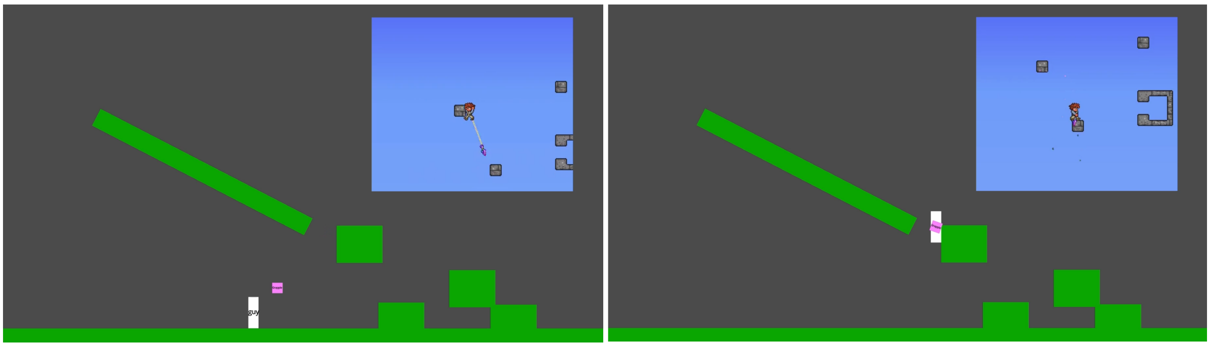


Figure 6 – Phase 1: The grappling hook mechanic that was built by students, in this case the teleportation grappling hook. The left side shows the player while still standing on the ground and shooting the grappling hook. On the right side, the hook has landed and the player has teleported to the hook.

far uncovered by the teams and could be extended further once further potential has become clear.

5.2. Phase 2: Adding Mouse Dragging and Line Visualizations to Pronto

The student team worked on both an extension for mouse dragging and the new line rendering Behavior.

The new *VisualLine* Behavior must be configured to take references to two Nodes in the game scene. When the game is run, a line will be drawn between these two node and update according to their position. This enables building grappling hook lines, laser beams, and other mechanics without leaving the scene editor to reach for code.

To support better mouse handling, the students extended the *Controls* Behavior. Connections to and from *Controls* can now also use a drag trigger, which is fired when a mouse button is held and provides the mouse cursor's position as a parameter. Additionally, the mouse-up event now also provides a duration as a paramater than Connections may choose to use. Both changes can be seen in Figure 7.

5.3. Breaking Changes and Throw-Away Prototypes

Initially, there were issues as we asked students to work in parallel on the same, relatively small Pronto code base without creating separate branches during the framework extension phase. Breaking changes could mean that code written by other teams may no longer work from one commit to the next.

To mitigate the issue, we asked students to consider two factors. First, as mentioned before, breaking changes are to be announced in the seminar's text channel. Second, the prototypes created during the mechanics phase should be kept small and fast to re-create. Not only does this emphasize their nature as throw-away prototypes, it also means that teams do not have to be extra careful when performing more fundamental changes to the framework. If a team needs to explore further directions from a prior prototype, they should be able to re-create the prototype within minutes. As we asked teams to create branches during the prototype phase, there was also no risk of prototypes that teams were still working on to spontaneously break due to other teams' changes.

5.4. After the Seminar

At the end of the seminar, the student teams had successfully created prototypes for six different prompts and made additions as well as changes to Pronto. During the in seminar discussions, students were both able to discuss their prototypes and what the experienced during playtests as well as what they want from and how they want interact with Pronto based on working with it.

Prompts focused on one game aspect, or type of game, that mechanic prototypes should be created for. To give students a direction for their own prototypes, prompts included at least one already existing from that pool as an example together with a list of fitting mechanics from that game. To test Pronto's capabilities, the prompts aimed at covering a variety of possible game mechanics. The final prompt was testing even the pupose of Pronto itself: Should it stay focused on individual mechanics prototypes or

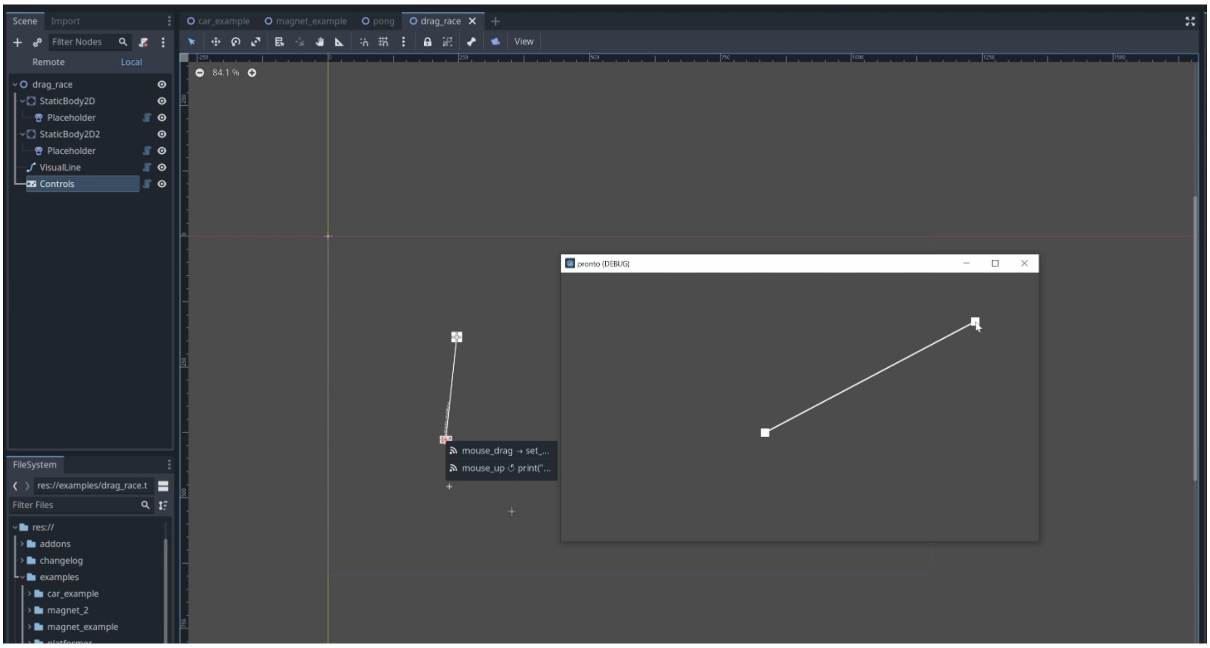


Figure 7 – Phase 2: A screenshot of two Pronto framework modifications that were built by students, visible in both scene editor and the running game. The new Behavior can render lines between two nodes. The Controls *node has extended support for mouse interactions.*

could it even be used to make an "entire" game? Since this prompt was given the same iteration time as all other prompts, one week, this tested of course the feasability for very small games, not larger projects,

The following six prompts, and example games, were used during the seminar:

- Mechanics for a top-down car racing game (e.g. Mario Kart)
- Mechanics that are triggered by player actions in a platformer game (e.g. Terraria)
- Mechanics that focus on the game environment in a platformer game (e.g. Celeste, Mario)
- Mechanics in simulation games (e.g. SimCity)
- Mechanics in turn-based games (e.g. Pokémon)
- An entire game (e.g. Asteroids, Doodle Jump)

While discussing with the students and looking at their created prototypes, it became apparent that Pronto is currently not equally well suited for all kind of mechanics. For mechanics with one or more clear player characters (e.g. the top-down racing game mechanics and the platformer mechanics), students usually reported Pronto as well suitable. For the other prompts (simulation game mechanics, turn-based game mechanics, and "an entire game"), students reported some trouble; most often creating all neccesary connections created a lot of visual clutter that made it difficult to create new connections, edit existing connections, and to what is influencing what at a glance.

6. Related Work

Most modern game engines feature low-code programming interfaces, either aimed at designers who do not have professional programming training, or beginners. As an example, Unreal Engine features a Blueprints system (Epic Games, 2012) that allows developers to use node-and-wires visual programming to define game behavior. Targeted at beginners, Construct 3 (Scirra, n.d.) offers a visual interface akin to a table that defines connections between triggers and actions. Its mental model is thus similar to Pronto as presented here; however, the table is removed from the game's visuals and developers are led through a wizard with multiple forms to set the connections up, whereas in Pronto script code is used what would be specified in forms in Construct 3.

Game engines tend to advertise prototyping primarily when it comes to level design. For example, most engines feature tools to quickly "block out" levels, by combining primitive shapes that will later be replaced by specifically designed assets. Another common practice to try out novel ideas is to "mod" existing games: a shooter such as Half-Life 2 is taken and parts of its mechanics are modified. This may result in a new objective for the game, potentially not even involving elements integral to the original game such as shooting. Starting from a finished game allows developers to benefit from a well-defined set of basic functionalities, such as character movement, artificial agents, or menus, on which to add their own spin.

Visual and Low-Code Programming Visual programming environments for graphical programs, such as Scratch (Resnick et al., 2009) or Snap (Harvey & Mönig, 2015), are also commonly used to develop prototypical or even fully realized games. Their strengths for prototyping typically stems from a highly domain-specific set of primitive programming units, such as character movement, and great feedback loop.

While there are visual programming environments that utilize connection-based interactions and metaphors, they are usually designed for more general purposes than game prototyping. For instance, Lively Kernel features the lightweight Lively Connections (Lincke, Krahn, Ingalls, & Hirschfeld, 2009) as well as the full dataflow system Fabrik (Ingalls et al., 2016). Both systems allow users to connect general UI elements, which makes it possibly to create dataflow-based programs in a visual manner.

Video Games and Education There are many ways in which video games are used in educational context. As video games are a large and influential medium, there are also many students that play video games, making them a promosing and potentially enganging part of education (Squire, 2003). For instance, games may be used as part of edutainment that uses games to teach general topics such as mathematics or biology through a video game.

As video games themselves are programs and part of computer science, there are also used in computer science education. Several programming environments intend to introduce programming by allowing students to program games or solve game-like puzzles. For example, the aforementioned environment Scratch was conceived for use in educational context. Another example is the programming language Logo, and other languages based on or inspired by it, that was meant to be easily learnt by programming beginners and can be used to solve game-like puzzles(Abelson, Goodman, & Rudolph, 2004).

Additionally, there are also courses on general programming aspects that may use video games as examples in a lecture or as a topic for a lecture accompanying project. Some of these courses teach or study also aspects of game development itself, with results being published at venues such as the Game Developers Conference (more focused on industry participation) or the Conference on Games (more focused on participation from scientific communities).

7. Conclusion and Future Work

In this paper we introduced Pronto, a visual and interactive prototyping framework for game mechanics in the Godot game engine. Pronto focuses on quickly creating prototypes by letting programmers define behavior directly between the visual elements of their game. To demonstrate how our framework can be used, we described both a walkthrough as well as a seminar we designed and held.

There are several ways that the Pronto framework could be extended in the future. Both small quality of life improvements as well as bigger extension, e.g. new behavior nodes or even completely new interactions, are possible.

Currently, Pronto was only used in one seminar. To further evaluate its uses, it would be possible to both use it in future seminars as well as other venues. For example, it would be possible to explore how useful Pronto is at prototyping ideas if used in a game jam. Studies like these would provide insights on Pronto's usefullness in a complete, non-restricted prototyping phase.

Additionally, further studies on using Pronto in an educational context seem promosing. This could evaluate how the topic of game development influences motivation and results of students as well as what students learn about tool and framework creation and modification.

It would also be interesting if ideas of Pronto could not only be used for throw-away prototypes, but integrated into a finished game. The inherent nature of prototypes as quick, throw-away may make this impossible, but maybe some of the concepts behind Pronto might also be of use outside of prototyping.

8. References

Abelson, H., Goodman, N., & Rudolph, L. (2004, 10). Logo manual.

Epic Games. (2012). *Unreal engine blueprint.* Retrieved from `https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/`

Harvey, B., & Mönig, J. (2015, October). Lambda in blocks languages: Lessons learned. In *2015 IEEE blocks and beyond workshop (blocks and beyond)* (p. 35-38). USA: IEEE. doi: 10.1109/BLOCKS.2015.7368997

Ingalls, D., Felgentreff, T., Hirschfeld, R., Krahn, R., Lincke, J., Röder, M., ... Mikkonen, T. (2016, October). *A world of active objects for work and play - the first ten years of lively.* ACM. Retrieved from `https://doi.org/10.1145/2986012` doi: 10.1145/2986012

Juan Linietsky, A. M., & contributors. (n.d.). *Godot engine.* Retrieved 2021-07-08, from `https://godotengine.org/`

Kasurinen, J., Strandén, J.-P., & Smolander, K. (2013). What do game developers expect from development and design tools? In *Proceedings of the 17th international conference on evaluation and assessment in software engineering* (p. 36–41). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2460999.2461004` doi: 10.1145/2460999.2461004

Kultima, A. (2015). Developers' perspectives on iteration in game development. In *Proceedings of the 19th international academic mindtrek conference* (p. 26–32). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2818187.2818298` doi: 10.1145/2818187.2818298

Lincke, J., Krahn, R., Ingalls, D., & Hirschfeld, R. (2009, January). Lively fabrik a web-based end-user programming environment. In *2009 seventh international conference on creating, connecting and collaborating through computing.* IEEE. Retrieved from `https://doi.org/10.1109/c5.2009.8` doi: 10.1109/c5.2009.8

Murphy-Hill, E. R., Zimmermann, T., & Nagappan, N. (2014). Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *36th international conference on software engineering, ICSE '14, hyderabad, india - may 31 - june 07, 2014* (pp. 1–11). Retrieved from `https://doi.org/10.1145/2568225.2568226` doi: 10.1145/2568225.2568226

Nystrom, R. (2014). *Game programming patterns.* Genever Benning.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... Kafai, Y. B. (2009). Scratch: Programming for all. *Commun. ACM*, *52*(11), 60–67. Retrieved from `https://doi.org/10.1145/1592761.1592779` doi: 10.1145/1592761.1592779

Schell, J. (2014). *The art of game design - a book of lenses, second edition.* Boca Raton, Fla: CRC Press.

Scirra. (n.d.). *Construct 3.* Retrieved 2021-07-08, from `https://www.construct.net`

Squire, K. (2003, 10). Video games in education. *International Journal of Intelligent Simulations and*

Gaming, *2*, 49-62. doi: 10.1145/950566.950583

Integrating Traditional CS Class Activities with Computing for Social Good, Ethics, and Communication and Leadership Skills

Renato Cortinovis
Freelance Researcher
Italy
rmcortinovis@gmail.com

Devender Goyal
Raytheon Technologies
USA
dg1988@gmail.com

Luiz Fernando Capretz
Western University
Canada
lcapretz@uwo.ca

Abstract

Software and information technologies are becoming increasingly integrated and pervasive in human society and range from automated decision making and social media and entertainment, to running critical social and physical infrastructures like government programs, utilities, and financial institutions. As a result, there is a growing awareness of the need to develop professionals who will harness these technologies in fair and inclusive ways and use them to address global issues like health, water management, poverty, and human rights. In this regard, many academic researchers have expressed the need to complement traditional teaching of CS technical skills with computer and information ethics (computing for social good), as well as communication and leadership skills. In this paper, we describe our goals and some possible class activities we have developed and refined over the past few years with encouraging results, to help CS students understand the potential uses of computing for social good. In these carefully planned project assignments, we seamlessly integrate traditional approaches to develop technical skills with broader professional responsibility and soft skills. We then discuss the lessons learned from these activities and briefly outline future plans.

1. Introduction

In the realm of computer science, a new frontier is emerging – one that transcends algorithms and syntax. It is a call from academic researchers, leaders of society, and many public and private institutions like IEEE and ACM, to research and teach computing for social good, ethics, trust, transparency, and building software systems that are appropriate, safe, and reliable, and which can improve the lives of all people (Burton et al., 2018; Charette, 2021; Goldweber et al., 2013; Gotterbarn et al., 2018; Taherdoost et al., 2011). Many academic researchers and industry leaders have also emphasized the need to develop better soft skills in CS students, such as written and verbal communication, collaboration across disciplines, ability to give and take constructive feedback, and empathy, with the aim of better managing conflicts (Capretz, 2014; Capretz et al., 2017; Carter, 2011; Brown et al., 2009; Hazzin & Har-Shai, 2013; Winters & Manshreck, 2020).

We previously described several methods for engaging students and encouraging them to develop these soft skills, highlighting the need to teach computational sustainability, ethics, trust, fairness, and other values in computing (Goyal & Capretz, 2021; Cortinovis, 2021). In this paper, the main goal is to describe class projects and activities dedicated to helping CS students learn concepts of computing for social good, that is, how computer and information technologies can be used to address social issues ranging from health, water resources, poverty, and climate change, to human rights, mainly following the CSG_ED approach described by Goldweber et al. (2013). In order to further improve students' abilities to do good as well as helping to build their professional and personal success, we also integrate some approaches to helping them develop their communication and professional leadership skills.

In the sections below, we first describe Responsible Software Engineering concepts and present a framework for learning computing for social good, computational sustainability and ethical values as important professional skills. We then describe some pilot class activities we carried out, to help students learn these concepts and skills, followed by a discussion of the lessons learned so far. Finally, we hint at the intended follow-up activities.

2. Responsible leadership framework

Many researchers are advocating the adoption of social and ethical responsibility by businesses and professionals so that information technologies are used to address social issues in fair, ethical, and inclusive ways. Schieferdecker (2020) has stressed the need to adopt a broader set of values concerns in Software Engineering, which he has entitled ‘Responsible Software Engineering,’ described as follows:

1. Sustainability by design: In addition to the promotion of privacy, safety and security, and software quality, sustainability concerns like ecological sensitivity for energy and resource efficiency and value sensitivity in data collection and algorithms should be part of software engineering.
2. Techno-social Responsibility: Understanding how digital models could affect the society and shaping the digital business models and solutions according to agreed upon societal principles.
3. Responsible Technology development: Promoting research and development technology that is aligned with the UN sustainable development goals.
4. State-of-the-art Software Engineering: Promoting a sense of societal responsibility in the appropriate use of state-of-the-art software engineering methods and tools that fit the level of software criticality.

Based on our professional experience and academic research, in this paper we are also proposing that a broader set of professional values, responsibilities, and leadership qualities be considered. We propose the following framework and principles for computer science students and professionals to consider:

1. Consider research and development in the areas of computing for social good as well as sustainable computing. That is, harness computational and information technologies to achieve environmental, economic, and societal goals for sustainable development and global good. Many of the global issues that can be addressed by computing professionals relate to health, poverty, climate, legislation, and justice, as well as human rights. Some computational sustainability related areas are sustainable supply-chain systems, disaster management and resilience, energy savings and efficiency, sustainable agriculture, biodiversity and species conservation, and ecosystem informatics (Chatterjee & Rao, 2020).
2. Consider ethics and fairness when designing and developing software and information systems and consider non-functional qualities of software and information systems, such as reliability, safety, and security. Teachers should encourage critical inquiry, reflection, and discussion among participants to help students better prepare for the future (Burton et al., 2018). It is important to help students recognize that multiple stakeholders are impacted by the increasing use of information technologies such as automation of processes, connected systems, social networks, and Artificial Intelligence. In this regard, teachers can also stress the importance of expanding business processes such as risk assessment and risk management which are needed to protect the interests of an organization and its various stakeholders from unethical, and many times unintended, uses of technologies (Clarke, 2019).
3. Be willing to take time to consider wider and long-term impacts of their research and development projects. In this regard, students can be exposed to various frameworks like ethical decision making (Barger, 2008) and RRI (Responsible Research and Innovation) processes of governance (Jirotka et al., 2017), which aim to ensure that the processes as well as the outcomes of the research are aligned with social values by encouraging more inclusive and democratic decision making among various stakeholders.
4. Be professional in their behavior and help each other to learn and grow. As more companies are adopting Agile and Scrum methodologies and becoming aware of servant leadership concepts like empathy, stewardship, healing, and building community (Layton & Ostermiller, 2017), it is imperative that students internalize and practice these concepts. It would be helpful for students to learn ‘Virtuous Advocate’ values of concern for others and use moral means for social good (Akers et al., 2004).

3. Computing for social good

Goldweber et al. (2013), in particular, advocated for the seamless integration of computing educational activities for the social good (CSG_ED) in the traditional CS core activities. Their approach addresses two interrelated issues: the need for students to have skills and attributes that help them contribute to social good outcomes, and motivational issues in students who avoid taking computer science classes as a result of the inaccurate perception that computing careers do not make a difference or contribute to society. Their main objective was indeed to increase the interest in computing by women and other traditionally underrepresented groups, who typically wish to orient their studies in areas where they feel they can contribute to the common good and to the betterment of society in concrete ways. Hence, their approach aims at broadening the view of CS students beyond classical abstract technical domains such as faster algorithms for eigenvalue calculations or sorting numbers, to more socially relevant domains (Schneiderman, 1971).

The key idea is therefore to propose programming assignments that are fully integrated into the core activities typically found in the traditional curriculum but framed in socially relevant ways. They claim that existing research shows that integrating these activities all along the traditional curriculum pathway is more effective than segregating them into separate activities. Additionally, this approach is less intrusive in the sense that it does not force CS educators to reduce the number of technical aspects addressed in the curriculum. Over the years, Goldweber et al. (2019) further refined their model, proposing a taxonomy with four levels of CSG_ED involvement, ranging from redefining an existing problem using a socially-oriented framework, to tackling real world problems with tangible real-world benefits. The main difficulty of this approach seems to be finding a way to avoid increasing significantly the complexity of the assignments for the students.

4. Toastmasters approach to developing leadership and communication skills

Goyal et al. (2022) proposed modelling some of the class activities on Toastmasters International, which shows promising results in improving the communication and leadership skills of its members. Toastmasters supports its members' leadership and communication skills in the following ways:

1. Participation in club meetings and competitions provides a safe and encouraging environment to help members practice their communication and leadership skills.
2. Every member supports the growth of other Toastmasters members by way of encouragement and mentoring. The biggest reason for the success of Toastmasters is their encouraging and cooperative environment which helps the members overcome their fear of speaking in front of a group.
3. Members take various leadership roles in running the club successfully and contribute to the growth of Toastmasters clubs. The Toastmasters program provides training sessions to members to prepare for these leadership roles.
4. Providing documentation and mentoring to its members for various roles and club activities.
5. Helping members build relationships and cultivate a sense of community.

The activities present in Toastmasters meetings help its members to develop their communication and leadership skills by giving several opportunities to practice these skills, as follows:

1. Prepared speeches: Each member is provided opportunities to prepare and present speeches in front of other members in a cooperative and encouraging environment. As members gradually become immersed in the encouraging and dynamic environment and listen and learn from other people making speeches, they overcome their fear of speaking in front of a group. Giving a prepared speech also helps a member improve his/her planning, organization, and time management skills.
2. Evaluation and feedback: Giving and receiving feedback is a very important skill. It requires excellent listening skills as well as empathy. Toastmasters training material provides helpful guidelines to its members for providing feedback by first pointing out what the speaker did well and then how he/she can be challenged to improve. The speaker is evaluated on the structure and contents of the speech, as well as delivery skills like the use of effective gestures, body language, vocal variety, and eye contact.

3. Impromptu speaking: Impromptu speaking is called “Table Topics” in Toastmasters. One member prepares the questions and then randomly selects other members to answer the questions by ideally speaking between 1 to 2.5 minutes. Besides improving public speaking skills, it helps members to know more about one another, build relationships, and have some fun time.

5. Activities undertaken

We started about three years ago by experimenting with the Toastmasters approach with specific pathways on code of conduct and sustainability (Goyal et al., 2022). Students were instructed to read relevant documentation about these subjects, to prepare a personalized presentation of the main issues, and to discuss it in a Toastmasters like setting. The Toastmasters approach worked well, in the sense that the students engaged actively in the activity, but a preliminary evaluation showed that many students perceived these activities as not sufficiently relevant, and that they preferred to focus on more traditional technical aspects of computer science. This was likely influenced by the specific target population of a continuing education school for adults specialized in computing. These students were mainly young working adults looking for the quickest possible way to acquire the skills necessary to move into a new career in computer science. A few of them were already certified or had even graduated in different fields, but were looking to acquire more easily employable technical skills. A minority had higher objectives such as deepening their studies in CS at the university level, but again these students were usually quite focused on technical skills.

A possible solution to the negative perception reported by some students comes from the integration of the CSG_ED approach. While one of the main aims and rationales of Goldweber et al. (2013) was to enhance the appeal of computer science for potential students intrigued by social welfare, our aim was, on the contrary, to leverage the already existing strong motivation to study CS, in order to expand their perspectives and engagement with social issues, and acquire a broader understanding and appreciation of ethics and values in CS. Based on this strategy, we refocused the activities by seamlessly integrating CSG_ED project assignments into the computer programming activities planned in the traditional curriculum, followed by a scaled down version of the Toastmasters approach to share and discuss the outcomes. The project assignments proposed were as follows:

1. An adapted version of a RoboCat chasing radioactive mice: a simulation of a robot seeking out and catching contaminated mice in a nuclear energy facility following radioactive leakages (Goldweber et al., 2013). The students of an introductory first year Computer Science class first developed a solution to the basic problem of a cat with a single mouse. In doing so, they applied what they had learned about bi-dimensional arrays, and about functional decomposition, working in Java with simple – though creative in some cases – text-based interfaces (Fig. 1). The most advanced students in the class then considered more complex situations with more stationary mice, and experimented with different strategies such as brute force, greedy techniques, or Brownian movements.

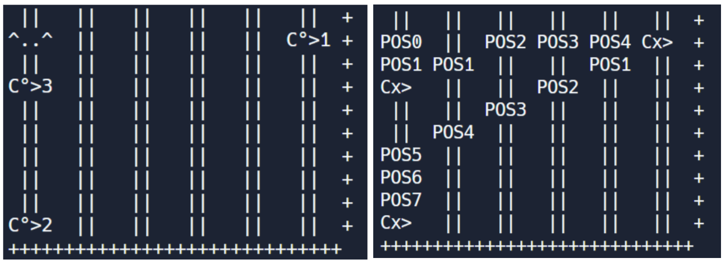


Figure 1: RoboCat with 3 radioactive mice, greedy strategy (text-based version)
Credit: K. D. Faye

2. As an important spin-off activity, a more advanced student in a final class started developing a graphical interface for the RoboCat world, to be used as a framework so that the students of the first class could concentrate on writing the strategy algorithm but would see the outcomes of their work through a more appealing and motivating interface. This framework will be

supported by a few HTML pages clarifying the rationale for the exercise (sustainability, nuclear energy) and some background information. The framework will be published in the open as an OER (Open Educational Resource), with the hope of making it a building block of an OER-enabled open pedagogy activity, where other students will further refine and extend the framework in a self-sustainable cycle – as one of us has already successfully demonstrated in similar contexts (Cortinovis, 2022).

3. Another CSG_ED style project assignment, this time for a more advanced class, involved working on a program to compute the (carbon dioxide and water) ecological footprints. Students were asked to provide a definition of ecological footprint, document how to compute it, and implement an interactive Java application to actually carry out its computation. Here they applied fundamental object-oriented technical concepts such as inheritance, polymorphism, use of abstract classes and interfaces, advanced static versus dynamic data structures of different complexity – both pre-existing in the library as well as fully developed from scratch, graphical user interfaces with absolute and relative positioning, up to the use of the iterator and composite design patterns (Fig. 2).

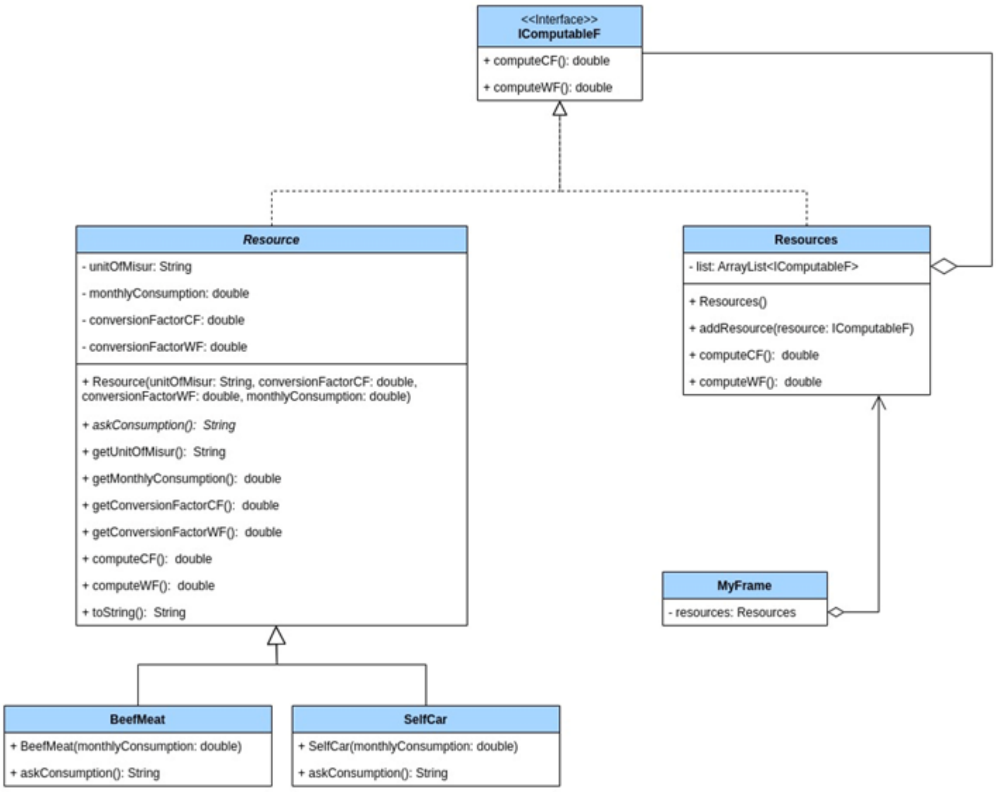


Figure 2: UML class diagram of the ecological footprint with composite design pattern
Credit: N. Invernizzi

4. Another group of students, studying database design, was asked to analyze the indicators used to track the progress of the Sustainable Development Goals 2030, in order to design a supporting database. Students carried out the analysis and produced an Entity Relationship diagram with the corresponding relational model, implemented it on a relational database management system, and performed a few SQL queries.

Crucially, in order to proactively stimulate critical thinking and concerns about values and social good, all the activities were complemented with a – this time – less preponderant subset of the Toastmasters approach previously piloted in the same classes, mainly including the Table Topics. Some sample questions for Table Topics were:

1. How do you compare the two attempted approaches: reading relevant seed papers, producing a presentation, and discussing it, versus developing a program in a socially relevant domain and briefly discussing it?
2. Do you feel motivated by these assignments in socially relevant contexts, or do you prefer traditional purely technical CS assignments?
3. How does the inclusion of social good concepts affect your perceptions of computer science?
4. How do these activities help you to realize that software can be harnessed to help people in society, improve ecology, and the environment?

In our current pilots, so far, we have done some informational assessment of students' perceptions of the proposed project assignments and teaching approach. Students have definitely found these projects helpful in increasing their motivation and awareness. Yet most students lamented an excess of complexity in the assignments, which – on the contrary – was seen as an interesting challenge by a few more skilled students. The main problem that emerged, in agreement with the literature, is therefore the need to carefully control the complexity of the proposed assignments, which should be personalized according to the students' competencies. The project assignment concerning the computation of the ecological footprint, for example, was carefully refined over a couple of years, to provide plenty of alternative options. These included varying levels of complexity and pedagogical guidance, and optional activities, to improve inclusion by catering to different abilities or motivations of the students. We also found it important to foresee suitable synchronization points, that is, pre-prepared partial baseline solutions, such as a skeleton application or just an UML diagram (Fig. 2), to make sure that nobody got stuck and was unable to progress towards the project goal.

6. Discussion – lessons learned

In this section, we summarize the main lessons that we learned from our experience in the last few years, that we feel are worthwhile sharing even at this preliminary stage.

1. We learned in our very first activities that students do not like to address socially oriented concepts or soft skills separately from the activities where they learn technical skills, as they feel they are drifting away from their core objectives – technical orientated for the majority of the students. In contrast, students definitely appreciate the integration of human and social aspects in their standard technical activities, under the form of project assignments with strong technical content complemented with suitable real world and social orientation. This is perfectly in line with the experience and suggestions from the literature (Goldweber et al., 2013).
2. We found that it is necessary to carefully plan the activities with plenty of embedded flexibility so that tackling real world-oriented problems does not excessively increase the difficulty of the proposed assignments. This is again in line with the recommendations from the literature (Goldweber et al., 2013). To help solve this problem in our later activities, we found it convenient to carefully pre-plan different levels of guided support as well as plenty of alternative and optional paths, to allow for educators to quickly react to the potential frustration or disengagement of the students. This proved instrumental to improving the level of inclusion, allowing students with different abilities to feel fully integrated and to fully participate in the activities.
3. We consider it convenient to foresee pre-defined synchronization points, that is, pre-prepared intermediate/partial solutions to periodically guarantee that everyone is on the same page and can progress towards the project goal. These synchronization points need to be frequent enough, so that they can be used to timely bring back on track those individual students who are drifting too far away from the intended path. A timely and personalized intervention in these cases avoids the risk that those students feel frustrated, if they are left drifting away for too long, having to set aside their work later on.
4. To keep the proposed activities at the desired level of complexity, even when tackling real world-oriented problems which may require technical skills beyond the level possessed by the students, we found it useful to develop suitable supporting frameworks, for example implementing complex graphic visualizations. This helps to simplify the proposed assignments

while retaining their realism, and therefore to maximize students' satisfaction and understanding of the whole picture.

5. In our experience the best approach to developing these frameworks is to propose them to students of higher-level classes as non-disposable assignments, motivating them to contribute to the common good by developing software as OER, that will facilitate and improve the learning and understanding of their peers. As reported by Cortinovis (2021), students deeply appreciate the possibility to personally contribute to the common good through this Open Pedagogy approach: engaging in these activities, indeed, is worthwhile even if just considering their exceptional motivational impact. Moreover, this method ensures sustainability by allowing the resulting OER, originating from what may seem like one-time tasks, to continuously evolve and expand through ongoing Open Pedagogy initiatives.
6. We found that to maximize the impact of these supporting frameworks (OER) and students' motivation, it is important that their effort is shared beyond their local environment, making every possible effort to involve students from other countries and different cultures. This has additional advantages in terms of intercultural understanding and motivation.

7. Conclusions

In closing, we advocate for an integrated approach that instills in students the duality of their roles as skilled professionals and conscientious global citizens. We attempted to illuminate a promising avenue for preparing the next generation of computer scientists, armed not only with coding skills but also with the power to wield technology for the betterment of society.

More specifically, we advocated for teaching human and social values and responsibilities as well as soft skills to CS students, and presented some class activities that we attempted and refined over a few years with encouraging results. Suggested class activities include CSG_ED projects related to computing for social good, suitably integrated with Open Pedagogy activities, and further activities modelled after Toastmasters International to help develop student's communication and leadership skills in an encouraging, dynamic, and cooperative environment. In our experience, even short exposures to understanding values and responsibilities and developing soft skills are beneficial and motivating for students, hence we have summarized the main lessons we learned from these activities in the last few years. While we do acknowledge that this is not yet a comprehensive, rigorous research study, we felt it was worthwhile sharing our preliminary results with potentially interested communities of researchers and practitioners.

In the near future, we plan to further study the impact of these activities by further improving, extending, and offering them in collaboration with more faculty members in different contexts, and collecting data for a more rigorous scientific analysis. Additionally, we would like to explore the possibility of establishing a mechanism to crowdsource successful CSG_ED projects as OER, to make it possible to reuse and continuously improve them, ideally in the context of OER-enabled activities (Wiley & Hilton, 2018). If anybody is interested in collaborating, please contact us.

8. Acknowledgements

The authors would like to thank S. Camarda for his contribution to the Ecological Footprint application, and all the students who keenly engaged in, and contributed to, the activities described.

9. References

Aker, M., Eaton, T.V., & Giacomino, D.E. (2004). Measuring and Changing the Values of Accounting Students. Journal of College Teaching & Learning, 1(4), 63-70. DOI: https://doi.org/10.19030/tlc.v1i4.1937.

Barger, R.N. (2008). Computer ethics: A case-based approach. Cambridge University Press, Cambridge, U.K.

Brown, Q., Lee, F., & Alejandre, S. (2009). Emphasizing Soft Skills and Team Development in an Educational Digital Game Design Course. Proceedings of the 4th International Conference on Foundations of Digital Games, 240-0247. DOI: https://doi.org/10.1145/1536513.1536557.

Burton, E., Goldsmith, J., & Mattei, N. (2018). How to Teach Computer Ethics through Science Fiction. Communications of the ACM, 61(8), 54-64. DOI: https://doi.org/10.1145/3154485.

Capretz, L.F. (2014). Bringing the Human Factors to Software Engineering. IEEE Software, 31(2), 104-106. DOI: https://doi.org/10.1109/MS.2014.30.

Capretz, L.F., Ahmed F., & Silva, F.Q.B. (2017). Soft Sides of Software. Information and Software Technology, 92(2017), 92-94. DOI: https://doi.org/10.1016/j.infsof.2017.07.011.

Carter, L. (2011). Ideas for Adding Soft Skills Education to Service Learning and Capstone Courses for Computer Science Students. Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, 517-522. DOI: https://doi.org/10.1145/1953163.1953312.

Charette, R. (2021). Demanding Fair and Ethically Aligned IT for the Future. IT Professional, 23(3), 46-52. DOI: https://doi.org/10.1109/MITP.2021.3070986.

Chatterjee, D., & Rao, S. (2020). Computational Sustainability: A Socio-Technical Perspective. ACM Computing Surveys, 53(5), 1-29. DOI: https://doi.org/10.1145/3409797.

Clarke, R. (2019). Principles and Business Processes for Responsible AI. Computer Law and Security Review, 35(4), 410-422.

Cortinovis, R. (2021). An educational CPU Visual Simulator. Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG).

Cortinovis, R. (2022). Evaluating and improving the Educational CPU Visual Simulator: a sustainable Open Pedagogy approach. Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group (PPIG).

Goldweber, M., Barr, J., Clear, T., Davoli, R., Mann, S., Patitsas, E., & Portnoff, S. (2013). A Framework for Enhancing the Social Good in Computing Education: A Values Approach. ACM Inroads, 4(1), 58-79.

Goldweber, M., Kaczmarczyk, L., & Blumenthal, R. (2019). Computing for the Social Good in Education. ACM Inroads, 10(4), 24-29.

Gotterbarn, D., Brinkman, B., Flick, C., Kirkpatrick, M.S., Miller, K., Vazansky, K., & Wolf, M.J. (2018). ACM Code of Ethics and Professional Conduct. Association of Computing Machinery.

Goyal, D., & Capretz, L.F. (2021). Promoting and Teaching Responsible Leadership in Software Engineering. Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG).

Goyal, D., Cortinovis, R., & Capretz, L.F. (2022). A Framework for Class Activities to Cultivate Responsible Leadership in Software Engineering Students. 15th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE 2022), Pittsburgh, PA, USA.

Hazzin O., & Har-Shai, G. (2013). Teaching Computer Science Soft Skills as Soft Concepts. Proceedings of the 44th ACM Technical Symposium on Computer Science Education, 59-64. DOI: https://doi.org/10.1145/2445196.2445219.

Jirotka, M., Grimpe, B., Stahl, B., Eden, G., & Hartswood, M. (2017). Responsible Research and Innovation in the Digital Age. Communications of the ACM, 60(5), 62-68. DOI: https://doi.org/10.1145/3064940.

Layton, M.C., & Ostermiller S.J. (2017). Agile Project Management for Dummies. John Wiley & Sons, New York, NY, USA.

Schieferdecker, I. (2020). Responsible Software Engineering. Future of Software Quality Assurance, 137-146. DOI: https://doi.org/10.1007/978-3-030-29509-7_11.

Shneiderman, B. (1971). Computer Science Education and Social Relevance. SIGCSE Bulletin, 3(1), 21-24.

Taherdoost, H., Sahibuddin, S., Namayandeh, M., & Jalaliyoon, N. (2011). Propose an Educational Plan for Computer Ethics and Information Security. Procedia-Social and Behavioral Sciences, 28, 815-819. DOI: https://doi.org/10.1016/j.sbspro.2011.11.149.

Wiley, D., & Hilton, J. (2018). Defining OER-enabled Pedagogy. International Review of Research in Open and Distance Learning, 19(4).

Winters, T., & Manshreck, T. (2020). Software Engineering at Google: Lessons Learned from Programming Over Time. O'Reilly Media, Boston, MA, USA.

Yu-Chih, S. (2008). The Toastmasters Approach: An Innovative Way to Teach Public Speaking to EFL Learners in Taiwan? RELC Journal, 39(1), 113-130. DOI: https://doi.org/10.1177/0033688208091143.

User-Centric Study and Enhancement of Python Static Code Analysers

Steven Chen
Lund University
UPC - ETSEIB
ohangyu@outlook.com

Emma Söderberg
Lund University
emma.soderberg@cs.lth.se

Alan T. McCabe
Lund University
alan.mccabe@cs.lth.se

Abstract

Despite the growing integration of code analysis tools into developer workflows, usability challenges persist in many aspects. Previous research, primarily focused on static languages and professional developers, has largely overlooked the needs of novice developers and non-static languages like Python. In this paper, we investigate the experiences of novice Python programmers with static code analysis tools. We aim to understand how these novices interact with and perceive these tools, with a focus on identifying usability pain points. To this end, we conducted initial user research with a survey followed by interviews. The insights derived from these studies were used to develop an enhanced version of the Pylint extension to Visual Studio Code, incorporating additional quick-fixes to improve the user experience connected to configuration of static analysis tools. The developed prototype extension was finally evaluated in a user study. The results from the interviews and surveys suggest that false positives, otherwise reported as a dominant cause of usability issues with code analysis, may not be as dominant for novice users who may focus on other aspects and not challenge the code analysis results. In addition, the results from the evaluation give early input on one possible direction for an enhanced Python code analyser interaction focused on novices.

1. Introduction

Code analysers, tools that inspect source code for potential problems like coding standard violations, are often integrated into the development process in different ways (Nachtigall, Nguyen Quang Do, & Bodden, 2019). Some analysers are plugged into integrated development environments (*IDEs*) to provide developers with immediate feedback as the code is written down. Others may need to be executed separately in the command-line interface (*CLI*). One of the main advantages of static code analysis is that it helps to identify issues early in the development process, at a time when it can be easier and less expensive to fix. For example, identifying a security vulnerability in the source code during development can be much less costly than finding it after the code has been deployed and is in use.

However, while static code analysers can be a powerful tool for developers, they also have limitations. Empirical research over the past decade has looked into programmers' experiences to understand why they need or use static code analysers (Christakis & Bird, 2016; Do, Wright, & Ali, 2022). Johnson et al. (Johnson, Song, Murphy-Hill, & Bowdidge, 2013) found the issue of false positives to be the most impactful reason why professionals stopped using code analysers. They also emphasized the importance of integrating the analysis tool into the coding environment. In a later study by Christakis and Bird (Christakis & Bird, 2016), they also found false positives to be significant barrier and suggested not enabling all code analyser rules by default. Moreover, some tools may require a significant amount of configuration and setup, or may produce messages that are difficult to understand (Nachtigall et al., 2019; Nachtigall, Schlichtig, & Bodden, 2022). These issues can make it challenging for developers to integrate static code analysers into their development process and get benefits from them, especially for novice users.

In our work, we focus on the experiences of novice Python users. As Python has rapidly grown within the computer science industry, particularly in the fields of machine learning and data analysis, there is a notably higher percentage of non-programmers who are less familiar with code analysis tools. This target group differs from those of previous studies, as most beginners are early in their careers or students. They may not be concerned about false positives, and some might not even be aware of code analysis despite encountering it within their coding environments, such as *IDEs* or notebooks. With the perspective that a good user experience with code analysers is critical to fully utilise the benefits of these tools (Do et al., 2022), we set out to increase our understanding of novice Python developers' experience of code analysers in order to gain insights into how the design may be adapted to meet their needs.

2. Method

The overall objective of this work is to investigate Python static code analysers and to design and implement improved features that can enhance the efficiency and fluency of the novice programmers' workflow. With this in mind, we focused on the following two research questions:

RQ_1 What experience do novice Python developers have with code analysers?

RQ_2 How can the interaction with Python code analysers be improved for novices?

The work presented in this paper was conducted as part of a M.Sc. thesis project at Lund University (Chen, 2023).

2.1. Data gathering and analysis

To address **RQ_1**, we first gathered empirical data by circulating a questionnaire among novice Python users. This covered topics such as familiarity with code analysers, perceived usefulness, and challenges faced. The questionnaire had three sections: basic information, users' experience with code analysers, and prior knowledge of static code analysers; and included a four screenshots featuring several Python code analysis results as examples (Fig. 4 - Fig. 7). These screenshots were included because we suspected that most users would have previous experience with code analysers, though they may not have had explicit knowledge of the concept and technology. Before distribution, we conducted a pilot survey for feedback, which was used to improve the questionnaire. We collected 39 survey responses from both offline and online sources.

After the survey, we followed up with a semi-structured interview to gather greater insight. The participants for the interviews were recruited from the survey respondents via a final question in the survey. In total, six interviews were conducted and recorded with informed consent, after which they were analysed for relevant content. An overview of participants in the survey and interviews is included in Table 1.

2.2. Prototype development and evaluation

To answer **RQ_2**, we analysed the gathered empirical data for issues that may be addressed with an intervention that could be implemented within the scope of a Master's thesis project. We elected to focus on the Python code analyser Pylint (Microsoft, n.d.; Pylint Contributors, n.d.-b) and the invalid-name (Pylint Contributors, n.d.-a) issue. Over the course of the project, we modified an existing Pylint extension to the Visual Studio Code editor[1], to provide an alternative interaction connected to the invalid name issue. We described the enhanced extension in more detail in Section 3.3.

To evaluate the enhanced Pylint extension, we designed two simple Python exercises containing errors detectable by Pylint. Both Python test files were similar, each including an introduction explaining the objectives to be achieved, namely the elimination of all linted and underlined code. In addition to this requirement, the users were instructed to name every term (constants, variable, class...) using a specific format. One exercise required terms to be in upper-case letters (UPPER_CASE standard), while the other required naming everything in lower-case letters (snake_case standard).

The exercises were presented in a random order, one using the existing Pylint extension from the marketplace, and the other using our enhanced Pylint. The exercise was supervised by the first author, and participants were encouraged to think aloud, allowing us to guide them if they deviated from the purpose of the exercise.

The aim of this experiment was to observe user interaction with the Pylint analyser and determine if

[1] `https://code.visualstudio.com/`

they reacted differently to Pylint's new features. By enforcing a specific naming standard, we simulated a situation where users intended to name variables in a non-standard way. This approach allowed us to observe, in real time, user interactions with the Pylint analyser, providing valuable insights into potential usability issues.

After completing the exercises, we asked users about their experience with Pylint. As each user's experience was unique, we remained flexible in our guidance, providing explanations about the exercise as necessary. During the interview, we made sure to explain how Pylint typically handles naming code issues, the conflict we sought to test, and the enhanced features we developed. Once confident that the interviewee understood the topic, we sought their preference between the two solutions our enhanced tool offered: solving the conflict by adding annotations to their source code, or adjusting the configuration file of the Pylint analyser.

Finally, we openly asked for suggestions and feedback to gain insights for future studies and improvements. An overview of participants in the evaluation is included in Table 1 and the prototype extension is available as open-source[2].

Table 1 – Overview of study participants and their background

<table>
<tr><th>Phase</th><th>Participant ID</th><th>Participant's Background</th></tr>
<tr><td rowspan="4">Survey</td><td>S1</td><td rowspan="4">The experience with Python ranged from less than 6 months (31%), 6m-2y (36%), 2-5y (26%), >5y (8%). Python had been encountered in academic courses (95%), personal projects (46%), and work (26%).</td></tr>
<tr><td>S2</td></tr>
<tr><td>⋮</td></tr>
<tr><td>S39</td></tr>
<tr><td rowspan="6">Interview</td><td>P1 (also S38)</td><td>Industrial/Informatics Engineering</td></tr>
<tr><td>P2 (also S11)</td><td>Informatics Engineering</td></tr>
<tr><td>P3 (also S35)</td><td>Computer Science</td></tr>
<tr><td>P4 (also S39)</td><td>Industrial Engineering</td></tr>
<tr><td>P5 (also S37)</td><td>Computer Science</td></tr>
<tr><td>P6 (also S36)</td><td>Computer Science</td></tr>
<tr><td rowspan="4">Prototype Evaluation</td><td>C1</td><td>Civil Engineering</td></tr>
<tr><td>C2</td><td>Molecular Biology</td></tr>
<tr><td>C3 (also P3)</td><td>Computer Science</td></tr>
<tr><td>C4</td><td>Computer Science</td></tr>
</table>

2.3. Threats to Validity

The **sample size** in this study, especially in the prototype evaluation, is relatively small. An investigation with a larger sample size could potentially enhance the internal validity of our findings and the representation of the target population. For gathering of empirical data and recruitment of participants, there may a bias in the **sample selection**. Specifically, for the interviews we used convenient sampling techniques, which resulted in some interviewees having prior relationships with the author. This could introduce bias, as these individuals may have tendencies to express positive comments, potentially losing objectivity.

Additionally, it is important to consider the potential influence of experimenter bias on our findings. Participants may have been aware of their involvement in an experiment focused on code analysis. This awareness could have influenced their behavior and responses, leading to experimenter bias. While we cannot confirm this hypothesis definitively, it is a factor worth considering when evaluating the outcomes of our study.

In conducting the survey, we employed both online and offline procedures to collect data, introducing **inconsistent procedure and conditions** as the degree of interaction varied between online and offline

[2]`https://gitlab.com/lund-university/vscode-pylint-invalid-name`

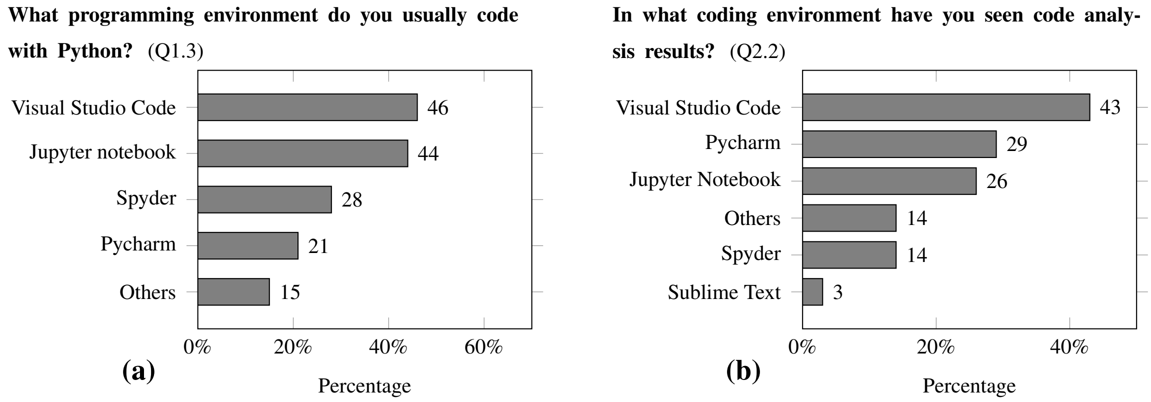


Figure 1 – Summary of the programming environment experience of the participants in the survey.

respondents. In the semi-structured interviews, the majority were conducted virtually via Zoom, while a single session was performed in person. However, the final evaluation took place in person. It is important to note that the location and environmental conditions differed for each interview, which could have also introduced variables affecting the result. Given that the respondents' backgrounds and expertise varied, the questions posed during the semi-structured interviews and evaluations were not consistent. This inconsistency could have introduced bias. Furthermore, the protocols for the surveys, interviews, and evaluations were generally flexible and open to adjustments, which could also have introduced variability into the results.

3. Results

In this section, we present the results of the study split into results from different activities; survey, interviews, prototype development, and prototype evaluation.

3.1. Novice Experience with Python Code Analysers: Survey

We found that the majority of users (85%) had seen code analysis results for Python before, while a minority (33%) had seen code analysis results for other programming languages. When considering the interaction with code analysis results, a majority of participants (59%) usually check the code analysis results and the majority (69%) typically checks warnings as they appear during coding, as opposed to after finishing the script (36%).

We also asked participants about their usual Python programming environment and found VS Code to be the most commonly used platform (46%), followed by Jupyter Notebook (44%) (see Fig. 1 for details). In connection to the environment, we asked participants if they had ever manually installed code analysers in their coding environment and found that the majority had not done so (77%). We further found that the vast majority of participants (97%) had never configured a code analyser.

While a majority of the participants (69%) viewed code analysis as beneficial, believing that it has a positive effect on code quality, almost half of the participants (47%) expressed frustration upon seeing their code highlighted (see Fig. 2 (b)). Two reported reasons for ignoring warnings, as opposed to investigating highlighted code, were that the warning looks useless, relating to the concept of false positives, and because of indifference to warnings or messages (Fig. 2 (c)). Despite these frustrations, a high percentage (92%) of users who interact with code analysis agreed that the analyser positively influences their coding process by detecting errors and bugs early (Fig. 2 (a)). When it comes to the reason for ignoring warning messages *after* reading them, 62% of users felt the warnings didn't apply (false positives), and 47% considered the problem difficult to fix (Fig. 2 (d)). In some cases, users mentioned that the problem looked hard to fix, meaning that the tool did not provide an immediate or adequate solution.

For cases where a participant might encounter a code analysis result they do not understand, the majority

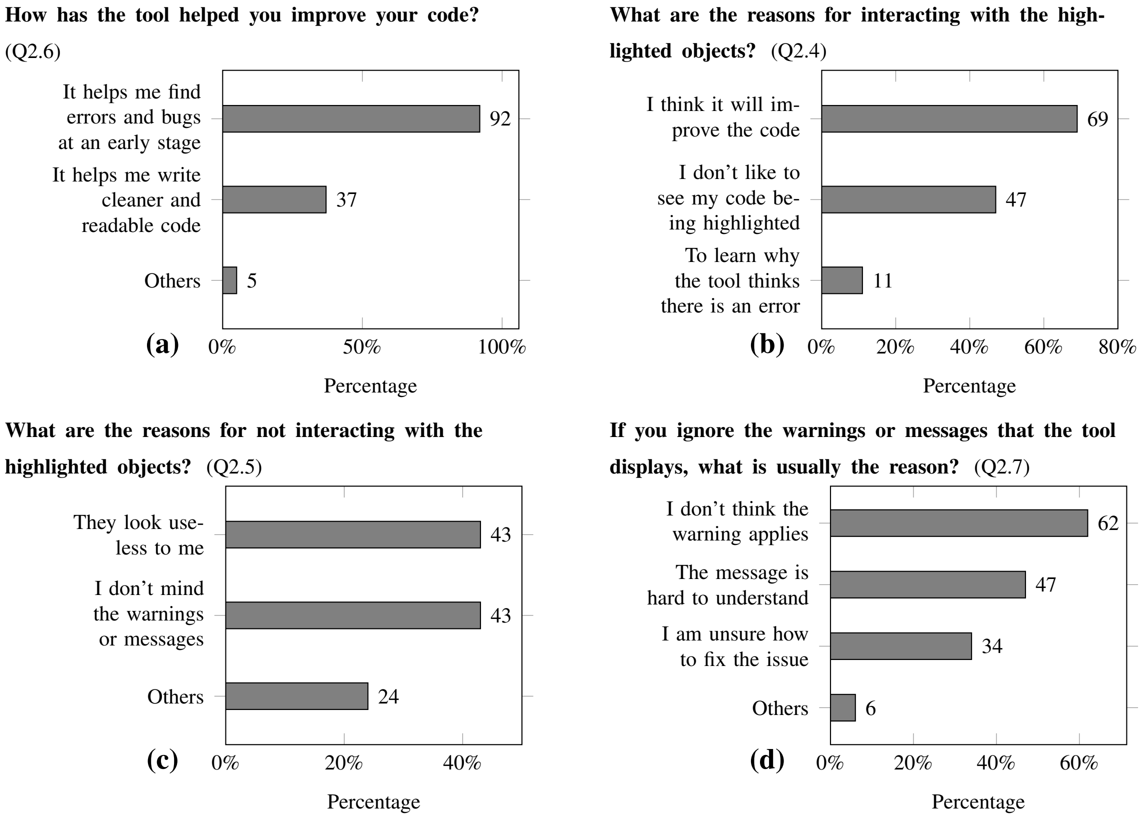


Figure 2 – Summary of survey participants' reported interaction with code analysis.

look to the internet or other programmers (74%) for assistance, while others attempt to fix the issue directly (49%), dig deeper into the code analysis tool (31%), and a fifth of the participants ignore the results altogether (21%).

The full text of the questionnaire can be found in appendix A, and a detailed breakdown of the results can be found in appendix B.

3.2. Novice Experience with Python Code Analysers: Interviews

All the participants in the interview part of the study expressed a positive attitude towards code analysers in general. They conveyed the belief that they are powerful tools that improve their coding by detecting errors at an early stage. (e.g., *"Analysers are incredibly helpful because they save a lot of time. Instead of having to run the code, wait for it to break, and then look at the interpreter to figure out what went wrong, you can catch the mistake ahead of time. It's especially helpful when you miss something like a semicolon or a mismatched parentheses. If I don't catch an error like that before running the code, and it breaks, it can be frustrating."* – P3).

After hearing about the positive aspects of code analysers, we asked the respondents to share their dislikes about the technology. We specifically asked about any pain points or frustrations they had experienced while using code analysers in their work or studies. This part of the survey was an important source of feedback for identifying areas of improvement, which can inform the design of features to address issues (e.g., *"Sometimes it tells me that a variable should not be named a certain way, but I feel like I can name my variable however I want."* – P1, *"Many times you just accidentally click on something and change your code, like maybe you press enter and it automatically changes things. It is really annoying. I tried to get rid of it but I couldn't."* – P4, and *"I really dislike when the analyser suggests a possible error of code. Because I wrote it for a reason, I probably wanna do it. I don't like when it informs me of something that may not be an error."* – P6).

We asked the respondents if they felt whether the warning messages given by analysis tools were clear enough, and what they typically do if they don't understand them. Based on their responses, we found that messages are usually understandable when the issue is simple, but when it becomes more complex the messages can become difficult to understand and do not provide enough options for clarification. Most users tended to turn to Google to find a solution (e.g., *"I think I rarely read what it says. It's just like, oh, it marks this place. And I quickly realize what is wrong. Almost all the time I understand the message, but when I don't understand it, I really don't understand it. If it marks something and I don't understand, reading it probably won't help me."* – P3, and *"I would try to understand the message. If it has a little help icon I probably would use it, it's easier to press up than to go to Google."* – P4).

False positives are a major usability problem with code analysers. To better understand the frequency of this issue among students and novice Python developers, we asked the respondents about their experience with false positives (e.g., *"I think it happens with typos, for example when you misspell a variable, but it's actually something you wanted to call it in that way on purpose."* – P2, *"I don't think there have ever been false positives to me that I can remember. If it suggests something, it's probably going to be very technically correct."* – P3, and *"I don't think that's ever happened. My code doesn't usually get too complicated."* – P4).

We asked the interview participants about a specific feature that we believed would be useful in code analysers: providing suggestions to help users solve the problems identified by the tool. We asked for their thoughts on this feature and on the quality of the solutions suggested by the tool, and the attitude towards suggestions was mixed (e.g., *"Quick fixes for simple things are good, you just click it and change it and it is really nice, I use a shortcut in VS all the time. I don't think it's ever happened that it marks something and the suggestion is bad. Most of the time there is a suggestion that is probably good."* – P3, and *"I've almost never seen a quick fix that works. The only time I've seen it work is it's like, oh you haven't imported, Numpy or other packages. But I, if it's more complicated than that, I've never seen that actually function in the tools I've used."* – P6).

Finally, we asked the respondents to suggest areas of improvement for the code analyser tool. We encouraged them to provide suggestions for enhancements in various areas, including user interface design and other functionalities that they felt were lacking (e.g., *"It would be nice to dismiss some specific warnings. I don't care about this thing on this line, but it usually will recheck every time. Like just dismiss this specific error in this specific situation. So in general, I want the check, but for this time, ignore it."* – P3, *"I think a lot of code analysis doesn't provide quick fixes or don't for a lot of cases, and I feel often there's pretty easy solutions to things. So getting some more quick fixes would be helpful."* – P5, and *" I think a feature that is quite lacking in a lot of these built in code analysers is they give you a very superficial description of the problem, but there's no option to have them explain more, or dig deeper. Then you have to go to an external source to understand what that means. You only get the superficial explanation from the tool most of the time."* – P6).

3.3. Prototype Development: Invalid Name Config Support

When analysing the interview results, one specific issue that caught our attention was the problem of naming terms that do not follow certain style standards. This type of warning occurs when a user names a variable, for example, with a term that does not follow Python's official PEP8 (Guido van Rossum and Barry Warsaw and Nick Coghlan, 2001) style standard, though it is not related to the functionality of the code. In some cases, users may want to name a term or variable in a specific way, regardless of formatting, capitalization, or case. The fact that the tool provide neither a solution nor a fast and intuitive way to ignore the warning can be frustrating for users and may discourage them from using the tool. We found this to be an interesting problem to study and address.

We opted to tackle the invalid-name issue in the context of VS Code, as it was the most used programming environment among survey respondents. We found that Pylint, a well known Python code analyser available as an extension to VS Code, offers a limited set of quick-fixes that address 14 code problems which do not include the invalid-name issue. Hence, we set forth to explore a solution for this issue in a variant of the Pylint extension for the Visual Studio Code editor[3].

In our implementation, we developed four quick-fix options that enable users to address the invalid-name issue at different levels: on a single line, from a specific line, within the current block, or at the document level by adding annotations. Additionally, we designed two quick-fix options that handle a pylintrc configuration file, allowing users to either disable the rule entirely in the working directory or add the treated naming variable as an always accepted term. We also considered adding support for a quick fix that would adjust the name to the form suggested by the issue, but unfortunately we did not have enough time in the project to also develop this feature. The final view of our implementation is illustrated in Fig. 3.

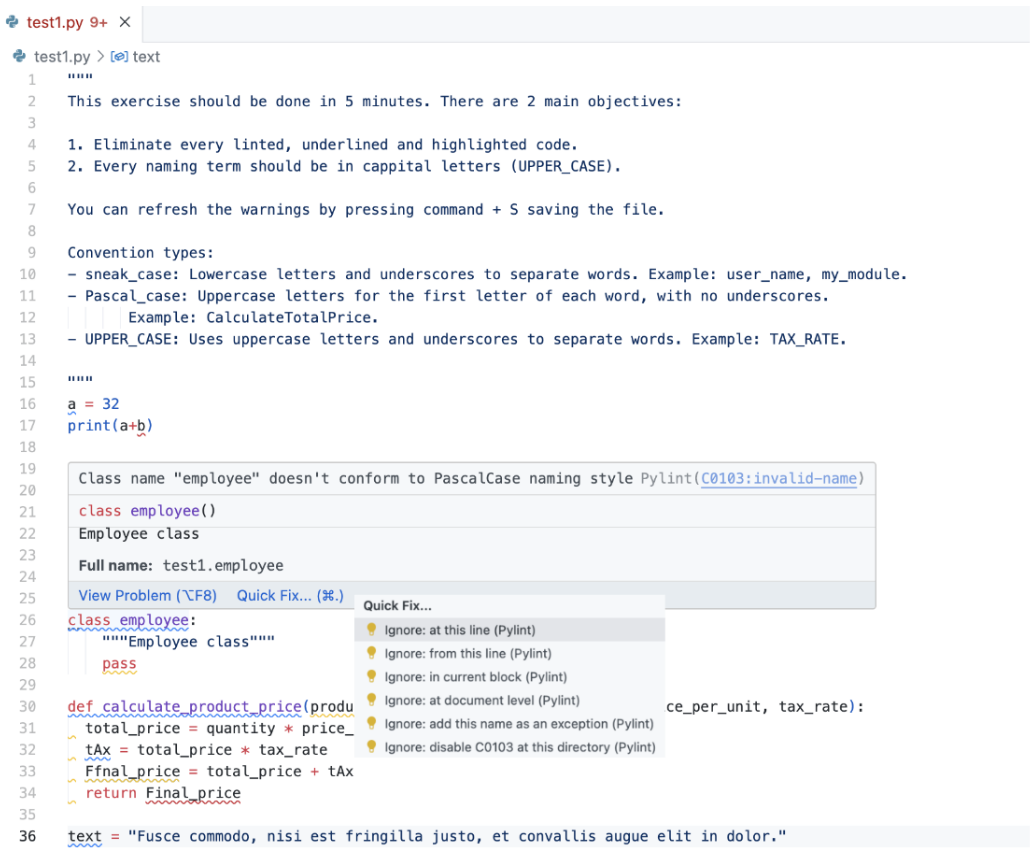


Figure 3 – Implemented quick-fixes.

3.4. Prototype Evaluation: Invalid Name Config Support

The two types of quick-fix options developed in the prototype extension address the invalid name issue in different ways, each presenting unique advantages and disadvantages. For instance, adding annotations directly into the user's source code might be a flexible and visual way of controlling and customising message rules. This can be an advantage, as users are always aware of which rules are enabled or disabled at different points in their code, and can quickly modify the rules when necessary. However, users might be annoyed by the tool adding extra information and code into their work.

On the other hand, using a configuration file, such as the pylintrc file, could be a more discreet solution. The drawback is that it's not as flexible as annotations, as users cannot specify which lines, blocks, or Python files they intend to customise rules for. Although the configuration file might be more desirable because it remains hidden, users might not be as aware of the available rules as they would be with annotations. Moreover, if users later want to edit or change the settings, they need to locate the pylintrc

[3]The extension with more implementation details is available at `https://gitlab.com/lund-university/vscode-pylint-invalid-name`.

file and modify its content, which might not be an intuitive process for developers.

The purpose of this final evaluation was to assess users' experiences with Pylint, and the newly developed features, as well as their thoughts on these two distinct methods for addressing code issues. We asked the participants to edit two simple Python programs via instructions given in comments in the beginning of each file. The instructions gave the participants the task of naming terms according to a specific naming scheme, edits designed to trigger naming issues generated by Pylint. The participants carried out one of the tasks with the new Pylint extension developed in the project and one task with the default Pylint extension. We aimed to determine which option is more suitable, if there is one, as well as to collect valuable suggestions and constructive feedback that could guide future work on similar topics.

All of the volunteers interviewed in the evaluation had some level of Python knowledge, but their expertise varied. Some were computer science students with substantial experience with programming languages and coding environments, while others primarily used Python as a tool to complete tasks in specific courses. Given these differences, we will first discuss their performance during the exercise, followed by the topics discussed during the interview. We will refer to participants as C1-C4.

3.4.1. Experience of the Exercise

The participants' responses to underlined codes were diverse. For instance, C2 consistently clicked on the "View problem" option for more information and clicked the light bulb whenever possible. However, the participant quickly lost interest in the message as it wasn't immediately clear and intuitive. On the other hand, C3 quickly understood the error and manually resolved some of the warnings. These differences can be attributed to varying levels of expertise with the Python language and familiarity with the code editor.

Despite these differences, all candidates reacted confused to the naming conflict presented during the experiments. However, they expressed varying levels of annoyance. Some proceeded with the exercise without giving it much thought, while others spent a few minutes trying to find a resolution.

A key observation was that only C1 noticed the difference between the two exercises, where the user could apply the developed ignore options. For example, C2 tried to find similar options by right-clicking on the linted variables. C3, on the other hand, assumed that the quick-fix option would enforce the standard that the linter was expecting.

3.4.2. Adding Annotation

We asked the candidates how annoyed they would feel if the tool provided an option to ignore issues by adding an annotation to their source code. Candidates C2 and C4 expressed a low level of annoyance, provided the number of annotations wasn't excessive. In contrast, candidates C1 and C3 expressed a high level of annoyance. C1 stated that having annotations would affect the code's readability, while C3 raised concerns about sharing annotated code with others, as it could be less readable and confusing. This latter opinion contrasts with that of C4: they expressed a preference for including comments in the code rather than in the pylintrc file, arguing that this approach clarifies what is enabled and disabled in the code when it's shared with other collaborators.

3.4.3. Dealing with Pylintrc

We also asked the interviewees how they felt about adding a pylintrc file to their directory and ignoring the rule within the configuration. All interviewees were open to having a configuration file that manages different Pylint code rules. However, after discussing the advantages and disadvantages of this method versus adding annotations to the source code, only C3 insisted that addressing the problem with pylintrc would be more suitable. C4 expressed a preference for annotations to enhance collaboration, and C1 suggested that having the annotation at the top of the document would be the best solution. Participant C2 mentioned that the choice between methods could depend on the severity of the rule. They suggested that annotations could be useful for important rules or warnings to consistently remind the user of their status, while for convention problems, the pylintrc file might be preferable.

4. Discussion and Conclusions

Interestingly, the mainly novice developers among our participants did not appear to be significantly affected by false positives. Although a high percentage of respondents in Fig. 2 answered that they ignore warning messages because they believe the warning does not apply, none of the users we interviewed could provide an example of frustration with false positives.

Instead, in the suggestion section of the interviews, our participants talked about improving the warning messages, increasing the number and variety of quick-fix features, and improving the user interface and other functional options.

One possible explanation for this contrasting perspective lies in the expertise level of our participants. Given that most of them had limited experience with the Python language, they may have a higher level of trust in the code analyser compared to their own understanding. In contrast, professional developers, with their extensive knowledge and expertise, may exhibit a higher level of confidence in assessing the correctness of the analysis tool. Novices, therefore, might be more inclined to rely on the analyser's results, even when confronted with warning messages that they believe do not apply directly to their code.

Another contributing factor could be the purpose for which novices typically use the Python language. Since many novice users are likely to work on smaller-scale projects or specific tasks, their code is generally less complex compared to those of professional developers. As a result, they may encounter fewer instances of false positives, as the simplicity of their code reduces the probability of provoking such cases.

Many of the issues that later emerged during the prototype evaluation interviews were related to the general usability of the tool, suggesting that there is a need for further design iteration. For instance, one of the most significant issues was that 3 of the 4 candidates did not recognise the difference between the original and enhanced Pylint tool. The reason for this varied depending on the user: candidate C2 sought functionality by right-clicking the variables, C3 misinterpreted the quick-fix functionality, and C4 didn't notice the option as it was situated at the bottom of the hover message. Furthermore, some problems echoed the survey findings, such as warning messages being difficult to understand.

Specifically for the invalid-name issue, C2 mentioned that the naming standard terms like "PascalCase" might be confusing for a user without prior knowledge of the topic, reflecting the warning messages' inadequate explanation of the correct standard or examples. On the other hand, she mentioned that the option to disable the rule might be confusing as it is referred to as error code "C0103", not the name of the issue itself.

Based on what we found in this study we see a need for further study of novice interaction with code analysis, both in Python and also other programming languages. As such, we have identified several directions for future work:

- Conduct a more thorough investigation of the usability challenges faced by novice Python developers. This should involve a larger and more varied sample size to capture a wider range of experiences and challenges.
- Expand the quick-fix options available in the Pylint tool for VS Code or other editor platforms. This could involve developing new features or refining existing ones based on user feedback.
- Consolidate the code implemented in this project and contribute to the wider Pylint community by submitting a merge request to the official GitHub repository. This could provide benefits to a large number of users.
- Refine the warning messages in Pylint to make them more comprehensible, particularly for novice programmers. This could involve rewriting the messages or providing additional context or examples to help users understand the issues identified.

5. Acknowledgements

The authors would like to thank all the participants in the study. This work has been partially supported by the Swedish Foundation for Strategic Research (grant no. FFL18-0231), the Swedish Research Council (grant no. 2019-05658), ELLIIT - the Swedish Strategic Research Area in IT and Mobile Communications, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

6. References

Chen, S. (2023). *User-centric study and enhancement of python static code analysers* (LU-CS-EX 2023-31). Lund University.

Christakis, M., & Bird, C. (2016). What developers want and need from program analysis: An empirical study. In *2016 31st ieee/acm international conference on automated software engineering (ase)* (p. 332-343).

Do, L. N. Q., Wright, J. R., & Ali, K. (2022). Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, *48*(3), 835-847. doi: 10.1109/TSE.2020.3004525

Guido van Rossum and Barry Warsaw and Nick Coghlan. (2001). *PEP 8 – Style Guide for Python Code.* Python Enhancement Proposal. Retrieved from `https://pep8.org/` (Status: Active. Created: 5-Jul-2001)

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *2013 35th international conference on software engineering (icse)* (p. 672-681). doi: 10.1109/ICSE.2013.6606613

Microsoft. (n.d.). *Vscode-pylint.* (`https://github.com/microsoft/vscode-pylint`)

Nachtigall, M., Nguyen Quang Do, L., & Bodden, E. (2019). Explaining static analysis - a perspective. In *2019 34th ieee/acm international conference on automated software engineering workshop (asew)* (p. 29-32). doi: 10.1109/ASEW.2019.00023

Nachtigall, M., Schlichtig, M., & Bodden, E. (2022). A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st acm sigsoft international symposium on software testing and analysis* (pp. 532–543). doi: 10.1145/3533767.3534374

Pylint Contributors. (n.d.-a). *Invalid-name / c0103.* (`https://pylint.readthedocs.io/en/latest/user_guide/messages/convention/invalid-name.html`)

Pylint Contributors. (n.d.-b). *Pylint.* (`https://pylint.readthedocs.io/en/latest/index.html`)

A. Questionnaire

Please see below for questions included in the questionnaire. The screenshots included in the questionnaire are included in Fig. 4, Fig. 5, Fig. 6, and Fig. 7.

Section 1

1.1. How much experience do you have working actively with Python? (Select one.)

- ☐ Less than 6 months
- ☐ Between 6 months and 2 year
- ☐ Between 2 and 5 years
- ☐ More than 5 years

1.2. What experience do you have with Python? (Select all that apply.)

- ☐ University or academy courses
- ☐ Personal projects
- ☐ Work
- ☐ Other (please specify):

1.3. What programming environment do you usually code with Python? (Select all that apply.)

- ☐ Pycharm
- ☐ Visual Studio
- ☐ Sublime Text
- ☐ Jupyter Notebook
- ☐ Others (please specify):

Section 2

2.1. Figure 1 is an example of static code analysis in Python. Have you ever seen similar highlighted code or warnings before? (Select one.)

- ☐ Yes
- ☐ No

2.2. In which coding environments have you seen these warnings? (Select all that apply.)

- ☐ Pycharm
- ☐ Visual Studio
- ☐ Sublime Text
- ☐ Jupyter Notebook
- ☐ Others (please specify):

2.3. Would you typically interact with the highlighted objects? (Select one.)

- ☐ Yes
- ☐ No
- ☐ Sometimes

2.4. If you interact with the highlighted objects, what are the reasons why? (Select all that apply.)

- ☐ I don't like to see my code being highlighted.
- ☐ I think it will improve the code.
- ☐ Other reasons, please specify why:

2.5. If you don't interact with the highlighted objects, what are the reasons why? (Select all that apply.)

- ☐ I don't mind the warnings or messages.
- ☐ They look useless to me.
- ☐ Other reasons, please specify why:

2.6. Figures 2-4 show more examples of Python code analysis results. From your own experience, how has the tool helped you improve your code? (Select all that apply.)

- ☐ It helps me write cleaner, more readable code.
- ☐ It helps me find errors and bugs at an early stage.
- ☐ Others, please specify:

2.7. If you ignore the warnings or messages that the tool displays, what is usually the reason? (Select all that apply.)

- ☐ I don't think the warning applies.
- ☐ The message is hard to understand.
- ☐ I am unsure how to fix the issue.
- ☐ Others, please specify why:

2.8. If you encounter a message that you don't understand, what do you typically do? (Select all that apply.)

- ☐ Ignore them
- ☐ Try to check the issue by myself.
- ☐ Look for external help (e.g., other people, the internet).
- ☐ Try to understand the message by digging deeper into the tool.
- ☐ Other, please specify:

2.9. When do you typically check the warnings? (Select all that apply.)

- ☐ When the warning appears as I code.
- ☐ After finishing the script.
- ☐ Before committing to Git or similar.
- ☐ Other, please specify:

Section 3

3.1. Have you ever used code analysers in other programming languages? (Select one.)

- ☐ Yes, please specify:
- ☐ I have not used code analysers in other programming languages.

3.2. Have you ever manually installed external code analysers into your coding environment to improve your code quality? (Select one.)

- ☐ Yes, please specify:
- ☐ No

3.3. Have you ever customized or configured a code analyser? (Select one.)

- ☐ Yes, please specify:
- ☐ No

Would you be willing to participate in an interview to provide further insight? (Select one.)

- ☐ *Yes, please leave us a contact email:*
- ☐ *No*

Disclaimer*: The information you provide in this survey will be used for research purposes only. All responses will be kept anonymous and confidential.* ***Consent statement****: By proceeding with this survey, you agree to allow us to use the information you provide for research purposes. You understand that all responses will be kept anonymous and confidential. If you choose to withdraw from the survey at any time, you may do so without penalty.*

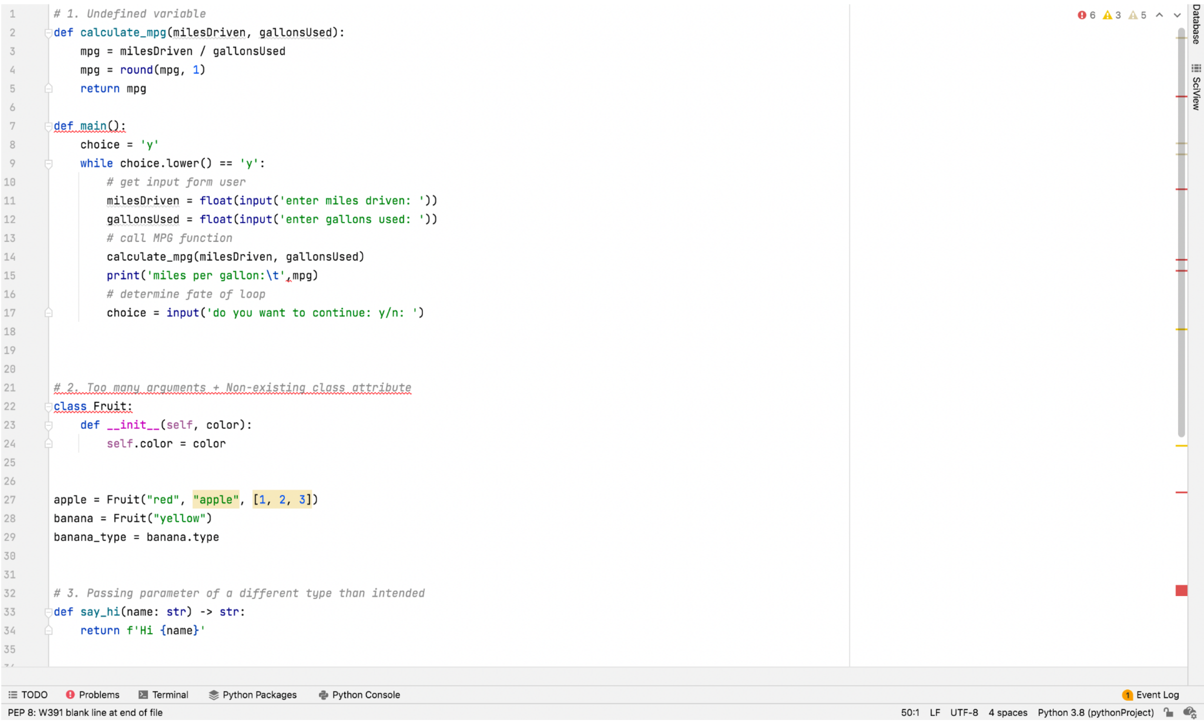


Figure 4 – Piece of code highlighted by a code analyser.

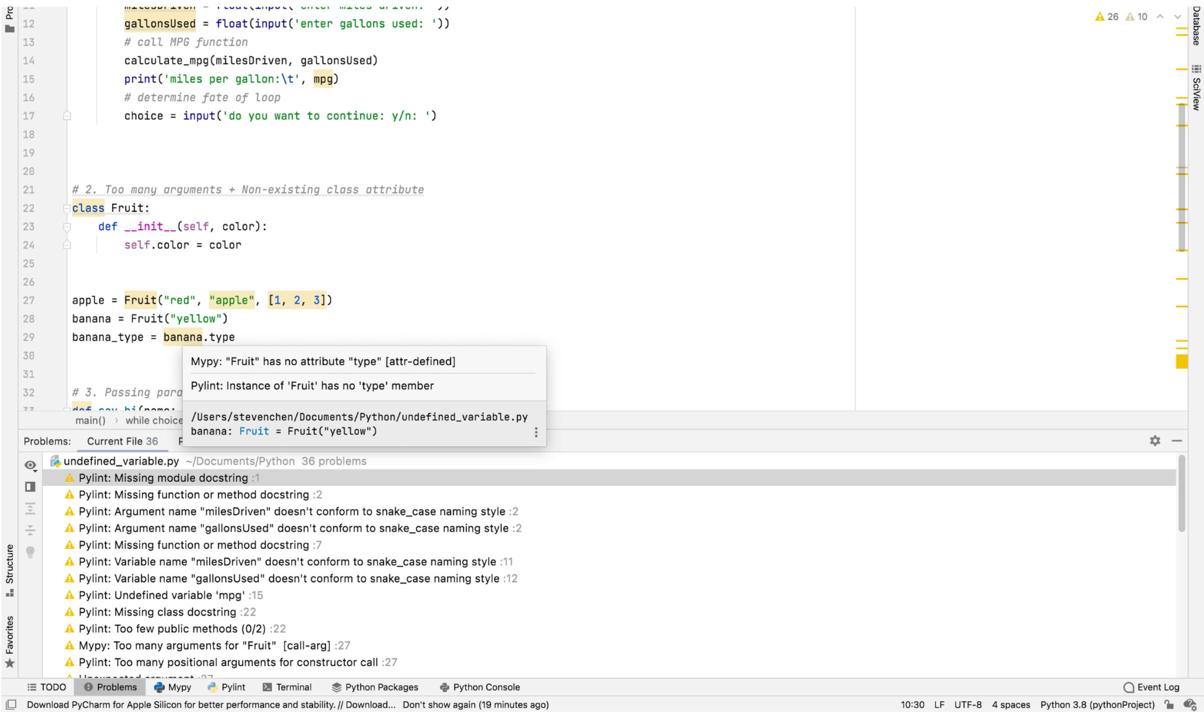


Figure 5 – Code analysis results by Pylint and Mypy in Pycharm.

B. Survey results

Please see Fig. 8 for additional survey results complementing the details included in Section 2.

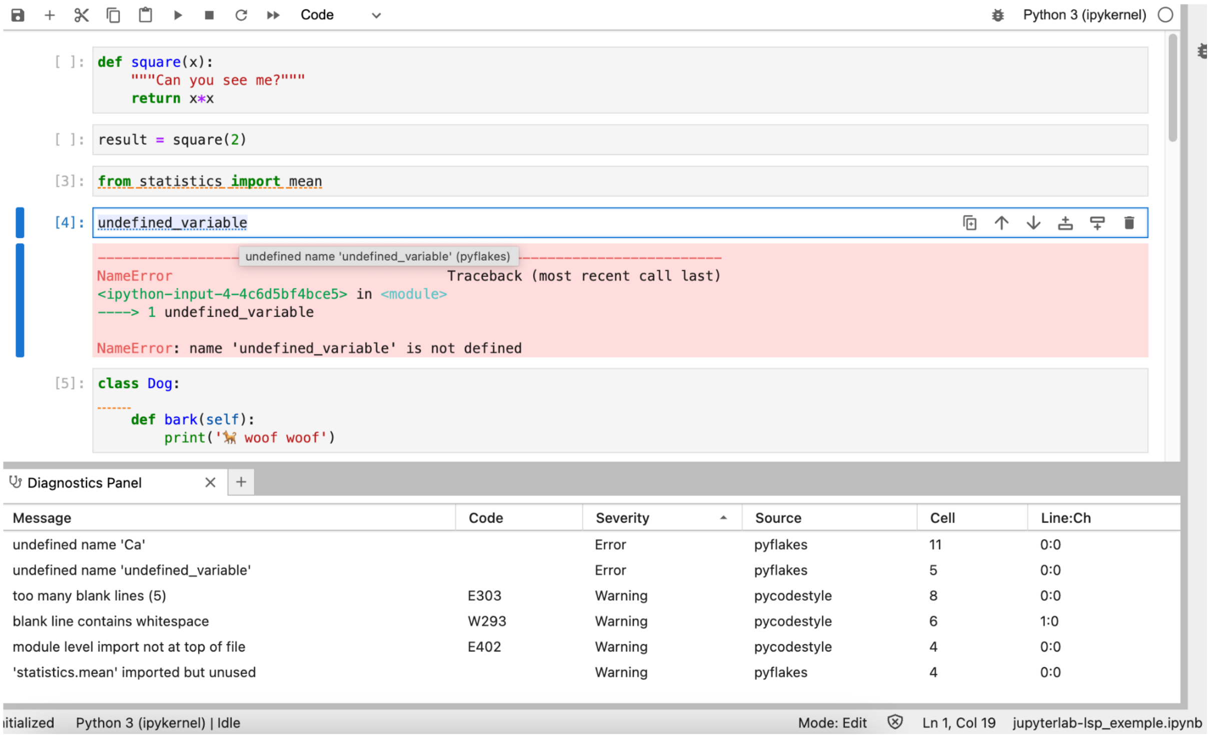


Figure 6 – Code analysis results by Jupyter-lsp in Jupyter Notebook.

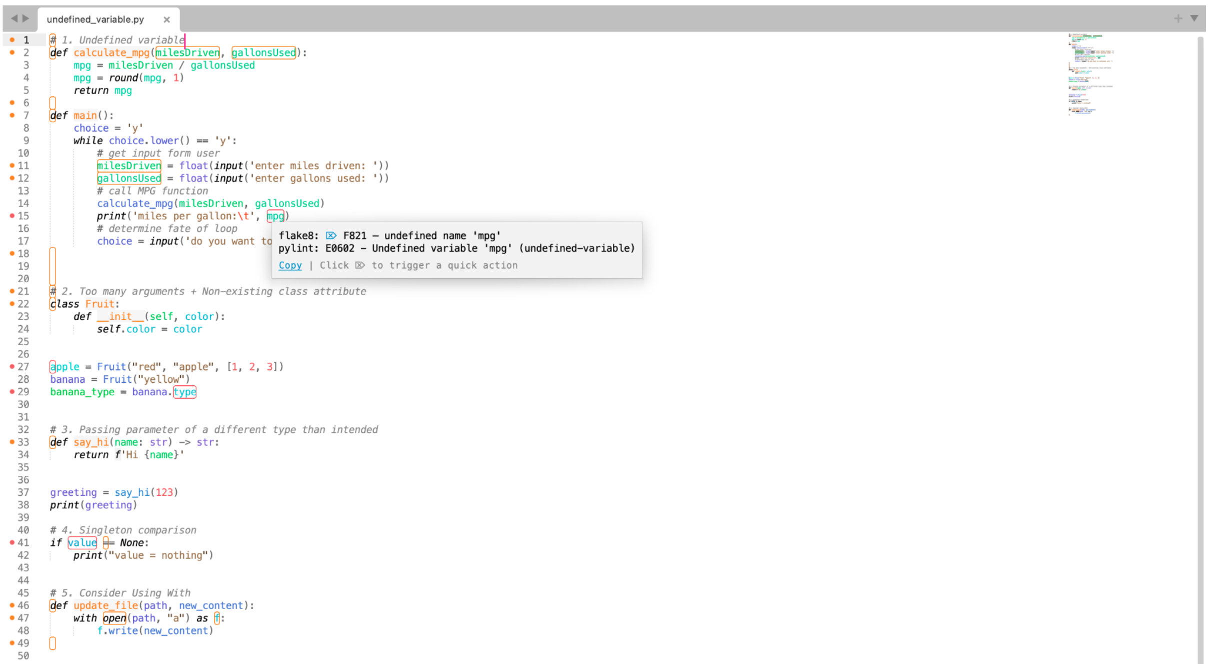


Figure 7 – Code analysis results by Pylint/Flake8 in SublimeText.

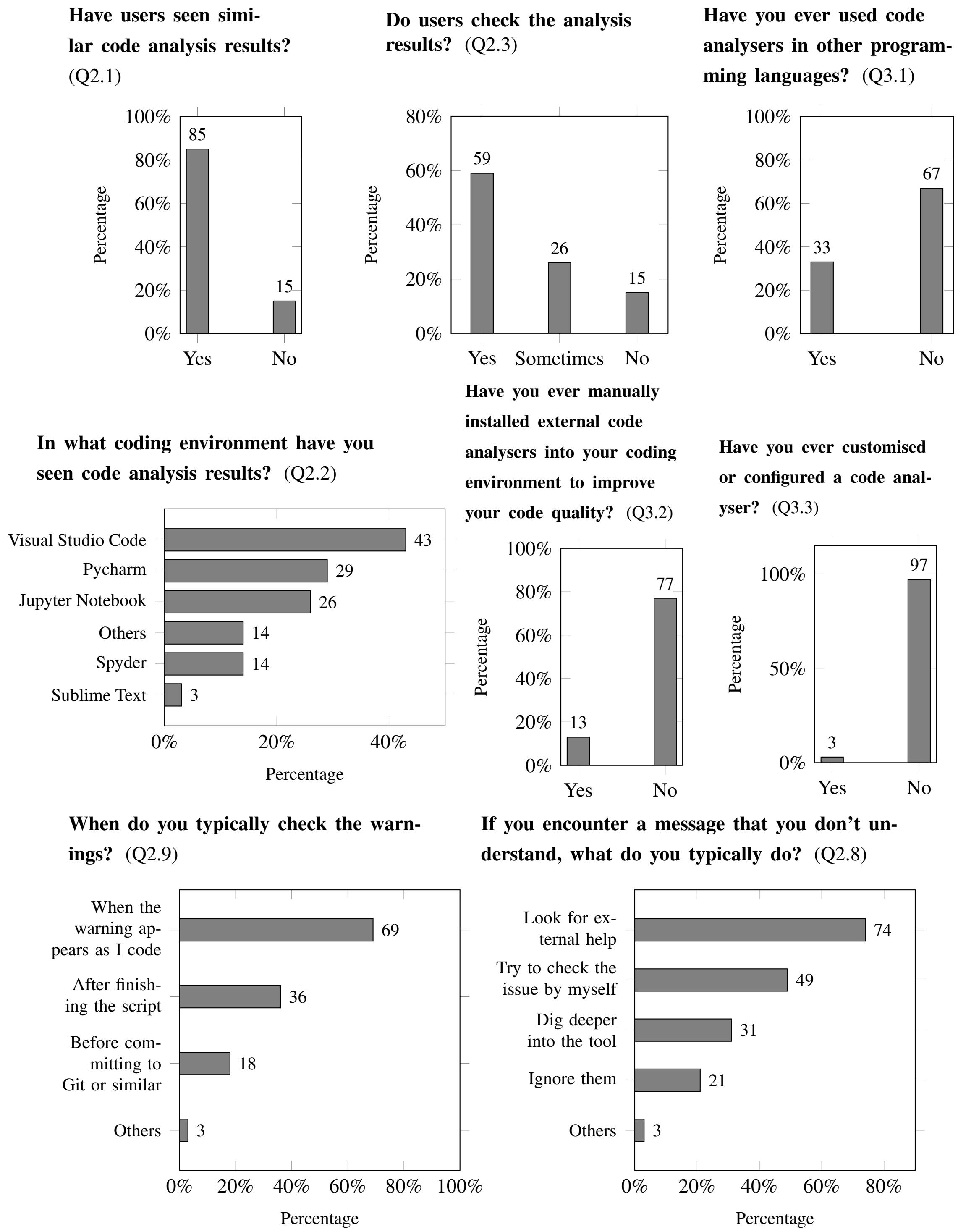


Figure 8 – Summary of survey results.

Parallel Program Comprehension: A Mental Model Approach

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

Empirical research on mental model representations formed by programmers during parallel program comprehension is important for informing the development of effective tools and instructional practices including model based learning and visualizations. This work builds upon our initial pilot study in expanding the research on mental models and program comprehension to include parallel programmers. The goals of the study were to validate the stimulus set, consisting of programs written in C using OpenMP directives, and to determine the type of information included in expert parallel programmers' mental models formed during the comprehension process. The task used to stimulate the comprehension process was determining the presence of data races. Participants' responses to the data race task and the level of confidence in their responses were analyzed to determine the validity of the stimuli. Responses to questions about the programs were analyzed to determine the type of information that was included in participants' mental models and the type of models (situation and execution) participants may have formed. The results of the experiment indicate that the level of difficulty of the stimuli (accuracy rate of 80.88%) was appropriate and that participants were from our target population of experts. The results also provide insight into the type of information included in expert parallel programmers' mental models and suggest that the data structures aspect of the situation model (the identification of data structures) was not present, however there is evidence that the data structures aspect of the execution model (the behaviour of data structures) was present. Further investigation into this topic is needed, and this study provides a stimulus set that would be useful for those wanting to expand the research on mental model representations to include the parallel programming paradigm.

1. Introduction

Programmers are frequently tasked with modifying, enhancing, extending, and debugging applications. To perform these tasks, programmers must understand existing code by forming mental representations. The understanding of programmers' mental representations formed during parallel program comprehension is important for developing instructional practices as well as tools and visualizations that are effective in assisting programmers with tasks that are uniquely challenging in parallel programming such as debugging and verifying the correctness of parallel code. For example, data races are a type of bug that can occur only in parallel programming. 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, requiring close consideration of the code by programmers as they will not appear every time that the program is executed. Since there are no applications that can do this work reliably, programmers need to be taught strategies for identifying data races and they need tools that help them better understand the parallel execution of code to assist them with this task.

Empirical research on mental representations formed by programmers during program comprehension has been predominately conducted using sequential code (Bidlake, Aubanel, & Voyer, 2022). Because of the considerable differences between parallel and sequential programming, it is impossible to determine if the findings of the empirical research using sequential code would resemble the mental representations formed during program comprehension using parallel code. The comprehension of parallel code requires programmers to mentally execute multiple timelines that are occurring in parallel at the machine level. Therefore, parallel program comprehension may require additional dimensions to construct a mental representation (Aubanel, 2020). As a first step to uncover the mental models formed during program comprehension we conducted a pilot study (Bidlake et al., 2022) followed by the main study presented here, that examines the components of shared-memory parallel programs that are critical to programmers' understanding of the execution of the code to detect data races. This study also investigates the types of mental models formed during program comprehension of parallel code when detecting a data race. Another goal of this study was to validate the programs we developed that were written in C and used OpenMP directives and for parallelization, referred to here as stimuli or stimulus set. OpenMP is a collection of compiler directives, library routines, and environment variables for shared-memory parallelism in C, C++ and Fortran programs in multi-core architectures (OpenMP-ARB, 2013). Given the continued popularity of OpenMP (Gonçalves, Amaris, Okada, Bruel, & Goldman, 2016), the stimulus set would be relevant for replication studies and incremental research that builds on previous work in the psychology of programming field, as well as expanding research on mental representations to include the parallel programming paradigm.

- RQ1: Does the stimulus set produce a reasonable level of difficulty?
- RQ2: What type of information is included in programmers' mental representations of parallel programs?

2. Background

Research in program comprehension encompasses both the study of the cognitive processes used by programmers to understand code and how programming languages and tools support these cognitive processes (Storey, 2006). During the comprehension process, programmers form mental representations of the code they are working with (Détienne, 2001). The mental model approach to program comprehension using sequential code involves the construction of the program model and the situation model (Détienne, 2001). The program model is formed by programmers when applying structural knowledge to the code resulting in a surface level representation (Pennington, 1987). The situation model, also referred to as the domain model, is formed during program comprehension when prior knowledge is used to form an understanding of the main goals, meaning, or real world situation represented by the program (Pennington, 1987). The definition of the situation model has also been extended by Aubanel (2020) to include the identification of data structures since they are used to infer the program's meaning. In addition to the program and situation model, an execution model has been proposed that includes the behaviour of the program in terms of data flow and data structures (Aubanel, 2020). The behaviour of data structures is considered an important part of parallel program comprehension. Specifically, how the data structures are accessed and changed by one or more threads and the relationship between data structures, is particularly important for determining if a data race exists.

3. Method

This study was preregistered with Open Science Framework (OSF) prior to data collection and analysis, and the stimuli, data, and R code are publicly available (Bidlake, 2022).

3.1. Participants

Participants had to have experience programming in C and using OpenMP 4.0 directives to implement parallelization. To recruit participants an advertisement with a link to the study was placed in the March 2022 OpenMP newsletter, emailed to the OpenMP Language Committee mailing list, shared during the International Workshop on OpenMP 2022, posted in the Psychology of Programming Interest Group discussion board, and posted on social media. In the end, a final sample of 20 participants completed the experiment online. Participants could choose to receive a $10 e-gift card as an incentive. Participation was voluntary and the protocol was reviewed by the research ethics board at the University of New Brunswick. Five participants were excluded from analysis because their average response rates were less than three seconds which is not a reasonable amount of time to examine the code, leaving a final sample of 15. Eleven participants were professionals and four participants were students. The mean age of participants was 39 years. Their mean amount of programming experience was 20 years and their mean amount of work experience was 12.53 years. The mean self-perceived level of expertise of participants, from 0 (novice) to 100 (expert), was 82.8 for programming expertise and 80.4 for parallel programming expertise. The mean self-perceived level of expertise of participants compared to their peers, from 0 (much worse) to 100 (much better), was 66.73 for programming and 70.07 for parallel programming.

3.2. Materials

The stimulus set used in this study was taken from our pilot study (Bidlake et al., 2022). They are programs written in C using OpenMP 4.0 directives and clauses with no comments or documentation. The selection of OpenMP directives and clauses for the stimuli was based largely on the set of directives and clauses referred to as The Common Core (Mattson, 2019).

In the pilot study (Bidlake et al., 2022), participants were asked what cues or program components they used to determine whether or not there was a data race for 20 of the stimuli. The responses varied greatly ranging in level of detail and also in the number responses they provided making it challenging to analyze these data. We speculate that the reason for the variation in responses is due to the open-ended nature of this question. In the main study we used an approach similar to Burkhardt, Détienne, and Wiedenbeck (2002) and included questions about specific components of the code to determine the type of information that is included in the mental representations formed by participants. In adding questions for the main study, we also considered that, in addition to the program and situation model, an execution model has been proposed that includes the behaviour of data structures (see section 2). The importance of data structures was also evident by the responses in the pilot study (Bidlake et al., 2022). Therefore, in the main study, 12 of these 20 stimuli, six with a data race and six without, were followed by more specific questions regarding the data structures in the programs (see Table 2). For eight of the stimuli, four with a data race and four without, participants were asked what cues or program components they used to determine whether or not there was a data race. The stimuli were presented to participants in a random order, therefore randomizing the order of the questions. The stimuli was not available for participants to look at while answering the questions and for each question participants were provided a text box to type their answer.

3.3. Procedure

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

The link to the online experiment was included in the advertisement for the study. Participants were instructed to determine as quickly and accurately as possible if each program contained a data race and respond by pressing the 'y' or 'n' key on their keyboard. The 76 experimental stimuli were presented to the participants in random order. For each stimulus, participants were given a time limit of 60 seconds to view the stimulus and respond to the data race task. If participants exceeded the time limit, exposure to the stimulus ended and they were asked to decide if the stimulus contained a data race or not. Participants were asked after each data race question to rate their level of confidence in their answer using a visual analogue scale that ranged from "Not Confident" to "Very Confident". Twenty of the stimuli were followed by a question about the code (see section 3.2). There was no time limit for answering the questions about the code. After completing the experiment portion, participants were redirected to the questionnaire documenting their level of education, age, programming experience, and their perceived level of programming expertise (Feigenspan, Kastner, Liebig, Apel, & Hanenberg, 2012). Participants were then asked if they would like to receive the e-gift card, and lastly, were redirected to the debriefing.

4. Results

For each trial, the accuracy (1 = correct, 0 = incorrect), level of confidence (0 = not confident to 100 = very confident), response time, and whether each trial had a race condition (y) or no race condition (n) were recorded and analyzed for 15 participants using the statistical software program R (R Core Team, 2021). The responses to questions following select stimuli were also collected and analyzed for 15 participants. The results of the experiment were analyzed in two contexts: identification of data races and responses to questions about the code.

4.1. Identifying Data Races

The mean accuracy for correctly determining whether or not a program contained a data race was 80.88% (SD = 39.34), the mean level of confidence in their responses was 83.09 (SD = 21.04), and the mean response time was 28.54 seconds (SD = 15.97). A one sample t-test was performed to test the null hypothesis that the sample accuracy was equal to chance (mu = .50) with a 95% confidence interval. The results indicated that the accuracy of participants was significantly higher than chance, $t(14) = 12.52$, $p < .001$.

Spearman rank correlations were computed to examine the relationship between accuracy and confidence, and the measures of self-perceived expertise, to determine if participants were from our target population. The strength of the correlations was determined using the scale for r values: r = .1 is weak, r = .3 is moderate, r $\geq$ .5 is strong (Cohen, 1988). The correlation coefficients are listed in Table 1. There were moderate positive correlations between accuracy and self-estimation of expertise in programming compared to their peers and between accuracy and self-estimation of expertise in parallel programming compared to their peers. When correlating their years of experience, a common measure of expertise, and accuracy there was no significant correlation.

	Accuracy	Confidence
	r	r
Number of years of programming experience	-.11	.61
Number of years of work experience	-.09	.47
Self-estimation of expertise in programming compared to their peers	.37	.12
Self-estimation of expertise in parallel programming compared to their peers	.40	.58
Self-estimation of level of expertise in programming (novice to expert)	-.20	.63
Self-estimation of level of expertise in parallel programming (novice to expert)	.03	.78

Table 1 – Spearman correlation coefficients.

The data were also analyzed using a mixed linear model that was fitted with the **lme4** package (Bates, Mächler, Bolker, & Walker, 2015) in R. The design used confidence as a continuous predictor, race condition as a repeated measures factor, and accuracy as the dependent variable. Using generalized linear mixed model with the **glmer** procedure from the **lme4** package, we first determined the best fitting model. Likelihood ratio values were then obtained with the **Anova** procedure from the **car** package (Fox & Weisberg, 2019), using the best fitting model with accuracy as the dependent variable for confidence and race condition. Results showed a significant effect of confidence, $LR = 38.55$, $p < .001$ and race condition, $LR = 30.63$, $p < .001$. The main effect of confidence in the data race task was significant, indicating that as confidence increased, accuracy on the data race task also increased (slope = .019, SE = .01, z = 3.80, $p < .001$).

4.2. Responses to Questions

The responses to the specific questions about the stimuli were also analyzed to determine the types of mental models formed during program comprehension (see Table 2). Questions that participants did not provide a response to were marked as incorrect since there was no time limit. Question 7, that asked "What is the value of 'z' at the end of the program?", was omitted from the analysis as some participants stated that the value was undefined since there was a data race whereas some participants answered according to what the value would be if there was no data race.

The 11 questions included in the analysis were six questions pertaining to the behaviour or mutation of data structures by one or more threads (execution model) and five questions related to the contents and size of the data structures (situation model). For five of the six questions pertaining to the relationship between data structures and how data structures are accessed and changed by one or more threads, participants were able to answer correctly with accuracy significantly greater than zero ($t(72) = 4.33$, $p < .001$) (see Table 2). However, no participants were able to correctly answer any of the five questions pertaining to the specific contents of the data structures and the size of data structures. There was no significant correlation between the accuracy of the specific questions related to the execution model and the accuracy of detecting a data race (r = -0.06, $p = 0.463$).

			Question Accuracy		Data Race Accuracy	
Question	Model	Data Race	Mean	SD	Mean	SD
1. How is array ‘a’ being updated?	E	y	40.00	50.71	80.00	41.40
2. How are array ‘a’ and array ‘b’ related?	E	y	33.33	48.80	73.33	45.78
3. How are array ‘a’ and array ‘b’ related?	E	y	13.33	35.19	73.33	45.78
4. How are the elements for array ‘a’ determined?	E	n	6.67	25.82	60.00	50.71
5. How is array ‘a’ updated after values were initially assigned?	E	n	6.67	25.82	93.33	25.82
6. What value is the last element in each row assigned?	E	n	0	–	80.00	41.40
8. What is the value of a[0]?	S	y	0	–	80.00	41.40
9. What are the dimensions of matrix ‘a’?	S	y	0	–	73.33	45.77
10. What is the length of array ‘a’?	S	n	0	–	100.00	0
11. What are the contents of array ‘a’?	S	n	0	–	80.00	41.40
12. What are the contents in the first column of matrix ‘a’?	S	n	0	–	80.00	41.40

Table 2 – Mean accuracy (percentage of correctness), when answering specific questions and determining the presence of a data race, as a function of specific question, model (E = execution, S = situation), and data race condition (y = data race, n = no data race).

Question 1, “How is array ‘a’ being updated?”, followed a stimulus where the data race occurred due to how array ‘a’ was updated. This may have been why the most participants were able to answer this question correctly. For this question, six participants accurately recalled how the array was updated with varying levels of detail. Two participants recalled the exact assignment statement (a[i+1] = a[i];). Another participant recalled the exact if condition that the update was made under (if a[i+1] > a[i]) although they only recalled the assignment statement in general terms as “…assigned to another value in the array”. Similarly, another participant recalled it was updated “according to valor (value) of other iteration”. This implies that observing that the update to the array was from another part of the array or used a different index value than the current position was important in determining the presence of the data race.

For question 2, “How are array ‘a’ and array ‘b’ related?”, followed a stimulus where the data race only involved array ‘a’. Five participants were able to give accurate responses describing the relationship between the arrays. Four participants responded directly that they were unrelated and one participant stated that they were both updated in a nowait clause and thus could happen concurrently, which was also correct. Another participant stated they could not recall if they were related but did state correctly that the data race was with array ‘a’ and therefore they did not look any further into the code.

Question 3, “How are array ‘a’ and array ‘b’ related?”, was repeated following a stimulus that also contained a data race, however, only two participants answered correctly. The data race in this stimulus did not involve array ‘a’ or ‘b’ which may explain why so few participants were able to answer correctly. Two participants who were not able to recall the answer did include in their responses that the race involved a scalar variable or the value‘ z’ which was correct. Another participant responded that they were both used in loops which was accurate but did not answer the question posed.

For question 4, “How are the elements for array ‘a’ determined?”, the stimulus it followed did not contain a data race. Only one participant was able to answer this question correctly. However, another participant’s answer was correct for the stimulus they saw before this one which did contain a data race.

Question 5, “How is array ‘a’ updated after values were initially assigned?”, followed a stimulus that did not contain a data race. Only one participant was able to correctly answer this question. Another participant did correctly identify that the update occurred within a nowait clause, however, their response was marked as incorrect because they said that it was updated by another array, when in fact it was updated using the original values from array ‘a’.

The responses to the specific questions were also analyzed to determine if the presence of a race condition affected the accuracy of the specific questions about the behaviour of data structures in the code (questions 1-6). The design used accuracy as the dependent variable and race condition as predictor. Using linear mixed model with the **lmer** procedure from the **lme4** package, we first determined the best fitting model was the fixed slope model given that the random slope model had a singular fit. Likelihood ratios were then obtained with the **Anova** procedure using the best fitting model with accuracy as the dependent variable for race condition. Results showed the effect of race condition on accuracy of the specific questions was significant, $LR = 11.01$, $p < .001$. The mean accuracy of the specific questions about the behaviour of data structures for stimuli without a data race was 4.44% (SD = 20.84) and the mean accuracy of the specific questions for stimuli with a data race was 28.89% (SD = 45.84).

The responses to the question asking what cues or program components participants used to determine whether or not there was a data race were also analyzed. There were 116 responses to this question collected. Based on the responses, categories were created that correspond to the components of the code that were included (see Fig. 1). If responses included more than one component of the code they were counted under each of the categories referred to in the response. The most common component of the code that was used to determine the presence of data races was the reading from and/or writing to memory with 64 responses referring to reading, writing, modification, accessing, or updating of variables or memory.

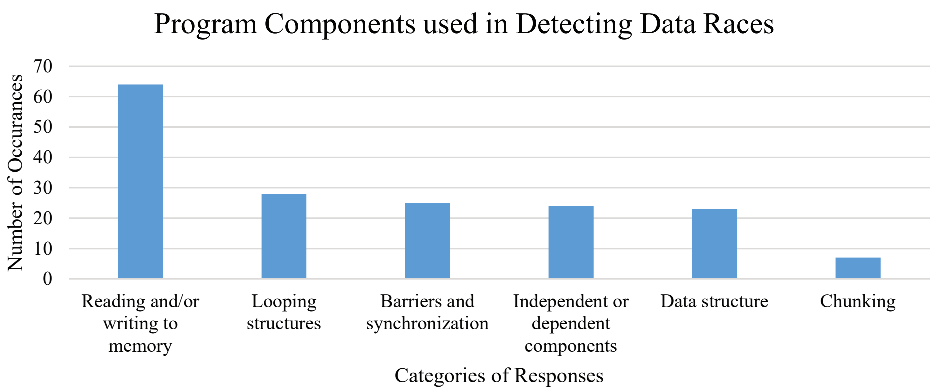


Figure 1 – Responses to question asking what cues or program components were used to determine whether or not there was a data race.

Twenty-eight responses referred to looping structures or iterations of a loop. These responses varied in detail with some referring generally to the "iteration space" and "access pattern of the loops", others specified they were looking for "overlap between loop bounds" and "independent loop iterations". Other responses used the presence of "barrier(s) between the loops" to verify the correctness of the code.

Barriers, both implicit and explicit, and synchronization, were referred to in 25 of the responses. In some responses where the barrier was implicit, participants made statements that implied they visually saw a barrier:

- "I saw that there was a barier between the loops and that in the second loop a thread didn't work with adjacent i indices."
- "I looked for the barrier between writing x and reading x in the second loop. Since that barrier was there, I determined there was no race."

Participants indicated in 24 responses that they were examining the code to determine if there were dependent or independent components (e.g.: tasks, sections, portions of arrays, data, loops, iterations). Responses that used dependencies to describe how they determined whether or not there was a data race tended to be less descriptive and included fewer details. For example:

- "Backwards dependency on a loop"
- "dependencies between sections"
- "Independent loop iterations"

There were 23 responses that referred to the data structure or array in the code. These included general references such as "array", referring to the indices of the array, and specifically using the variable name. In 19 responses, the indices of the data structure were used when determining the presence of a data race. The responses again varied in detail with some generally referring to the "index" or "indices" while others gave examples using variables for the indices: "out-of-bounds access to array a[i][j-1] which causes actual access at a[i-1][...]" and "the unshared j direction rather than i". In other responses, participants recalled specific index values that were used in verifying the correctness of the code:

- "...the first assignment (a[0]) was not synchronized..."
- "The last task reads a(0) while the first one writes the same data."
- "The barrier ensured that a element 0 was set before the loop started. Thus a0 was set before interation i started accessing element 0."

Seven of the responses described breaking the code into chunks described as blocks, regions, or sections[1]. Participants indicated the use of barriers, parallel regions, or loops when chunking. Experiments conducted using sequential code have also found that programmers use chunking to develop a mental representation of a program (Shneiderman, 1976; Barfield, 1986, 1997). In program comprehension, chunking is a cognitive process where groups of related lines of code are recalled together as a unit rather than each line individually (Barfield, 1997).

- "Break programm into implicitly 'barriered' blocks, then find Read/Write dependecy"
- "... Only check inside each "region" determined by barriers/synchronization."
- "Identify parallel blocks, look for reads overlapping with writes between blocks"
- "Divide programm into to blocks, analyze parallel loop iterations"
- "Checking if follow-up sections have to wait in a barrier for the previous one to finish."

[1]The sections directive was not used in the stimuli these responses were given for.

5. Discussion

This section discusses the results of our study with respect to our two research questions presented in the introduction (see section 1).

5.1. Level of Difficulty

Participants' self-estimation of expertise in programming and parallel programming compared to their peers had the strongest correlations with their accuracy on the data race task (see Table 1). This implies that their self-estimation of expertise was more accurate when comparing themselves to their peers as a point of reference. Given the mean level of expertise of participants compared to their peers for parallel programming was 70.07 (0 = much worse, 100 = much better), we believe that our participants can be classified as expert parallel programmers. We suspect, given the weak correlations between self-estimations of expertise and accuracy, participants may have found it difficult to measure their expertise (novice to expert) without a point of reference, i.e.: who is a novice and who is an expert.

The result of the t-test showing that the accuracy of participants is significantly higher than chance suggests that the level of difficulty of the stimuli is appropriate. The mean response times for all participants were within the 60 second exposure limit implying that the time exposure limit was appropriate. The results of the study show that as confidence increased, accuracy on the data race task also increased. Confidence based assessment which combines accuracy and confidence levels provides four regions of knowledge: uninformed (wrong answer with low confidence), doubt (correct answer with low confidence), misinformed (wrong answer with high confidence), and, mastery (correct answer with high confidence) (Maqsood & Ceravolo, 2018). In addition to the finding that the accuracy of participants was significantly higher than chance, the significant effect of confidence in the data race task indicates that higher performing participants had greater confidence in their answers and therefore had higher levels of mastery of the programming language and parallelization directives, both suggesting that it is unlikely participants were guessing.

5.2. Mental Representations

The ability of participants to correctly answer specific questions pertaining to the behaviour of the data structures, suggests the presence of an execution model. However, since no participants were able to answer any questions related to the contents and size of the data structures, this implies that the data structure aspect of the situation model was not formed during the data race task. The results also showed that when the specific questions pertaining to the behaviour of data structures were about stimuli that had a data race, participants performed significantly better than when the stimuli did not have a data race. One possible explanation is because to conclude that a data race exists programmers would need to devise an ordering of thread execution that would result in a race and then be able to convince themselves that ordering is possible. By developing this execution ordering and testing it to confirm the race is possible, programmers may have developed a stronger execution model and therefore were able to more accurately answer specific questions about the behaviour of the data structures when a data race was present, especially when the data race involved the data structure.

The responses to the question of what cues or program components participants used in determining the presence of a data race can be used to inform instructional practices in parallel programming education. Reading from and writing to memory was the most common consideration when detecting data races. Although this may seem obvious, to detect all of these instances and then determine if at least one write may occur at the same time as a read is not trivial. Reading and writing may occur in obvious places in code such as assignment statements but they can also occur within condition statements and function calls, for example. Even though assignment statements are easily identified, it has been shown that novice programmers often possess non-viable mental models of assignment statements (Ma, Ferguson, Roper, & Wood, 2007). Consequently, learning to identify data races may be even more challenging if programmers do not completely understand what is occurring in memory during a basic assignment statement. Information pertaining to looping structures including the iteration space, overlap between loop bounds, and access patterns of the loops were also important in determining the presence of data races. Being able to observe dependencies in the code either between or within looping structures or between different sections of the code was also essential to data race detection. Given the importance of barriers in identifying data races, identifying implicit barriers in particular is critical. In OpenMP, for example, programmers must be able to recall for each directive if an implicit barrier follows or not. OpenMP uses braces for implicit barriers and in cases where there is only a single line of code in a region braces are not required. The braces used for implicit barriers can easily be missed and also confused with braces used for other structures such as loops. Data structures, in this case arrays, were also frequently mentioned in the responses as program components used to identify data races. Unfortunately, arrays are among the topics that novice programmers feel they have the most difficulty with (Piteira & Costa, 2012), potentially compounding the challenges of teaching programmers how to identify data races.

One interesting observation came from looking at the stimuli that participants had the most difficulty in finding the data race. The stimulus with the lowest accuracy (mean = 26.67%, SD = 45.77) that used only the most basic OpenMP directives and clauses was one that involved shared memory and two-dimensional arrays (see Fig. 2). This stimulus was followed by the question asking what program components or cues were used in determining whether or not there was a data race. Nine participants that responded no to the data race task stated there were no loop carried dependencies. These participants may not have looked closely enough at the code; they would have been correct if the inner for loop had started at one instead of zero in line 18 of Fig. 2. Only one participant correctly indicated there was a data race and identified that it was from a loop carried dependency. Four participants stated in their answer that the program was incorrect and three of them specified it was incorrect due to an out of bounds error (see line 19 of Fig. 2). In programming languages such as Java this statement would be correct, however, this is not the case in C. The typical visualization of a two-dimensional array is a matrix in the form of rows and columns where the elements in the left and rightmost columns have no adjacent elements to their left or right (Leopold & Ambler, 1996). This visualization matches the limitations in programming languages like Java where indices are constrained but does not accurately reflect how these structures are stored in memory. In memory the rows are contiguous, meaning that the last element of the first row and the first element of the second row are adjacent. Despite the confusion the matrix visualization of two-dimensional arrays may cause as demonstrated here, these visualizations are commonly used in the context of the C programming language (Mattson, 2019; Srivastava, 2020). Given that the matrix visualization of two-dimensional arrays does not easily transfer from one programming language to another, and in fact may contribute to misunderstandings, we should also be questioning how likely it is that our visualizations and models of sequential programming will transfer to parallel programming.

6. Threats to Validity

The sitmulus set used in this study was taken from our pilot study (Bidlake et al., 2022), and although precautions were taken to reduce bias in the selection of OpenMP directives, the types of errors that were introduced to create data races may have been biased towards the background knowledge and experiences of the authors'. Bias that may have been introduced by participants because of their prior experiences with data races that may have caused them to look for mistakes they commonly make and having more familiarity with some directives than others. Another threat to validity is our lack of control over the experiment environment. Because the study was conducted online, participants may have been in a distracting environment.

Another bias that may have been introduced was the specific questions about the code and the selection of which stimuli they would be asked for. The level of difficulty of the questions may not have been equitable depending on what data structures the question referred to and if there was a data race. For example, if the question asked about the data structures that were directly involved in the data race, this may have been an easier question to answer than if the data structure was not involved directly with the data race or if there was no data race.

The small sample size we ended up with was likely the biggest limitation of our work as it greatly reduced statistical power. Prior to conducting our main study, we performed a power analysis on the pilot study (Bidlake et al., 2022) and also considered the minimum recommendation proposed by Brysbaert

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

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

    int n = 5, m = 10;
    int a[n][m], i, j;

    #pragma omp parallel
    {
        #pragma omp for
        for(j = 0; j < m; j++){
            a[0][j] = 2 + j;
        }

        #pragma omp for private(j)
        for (i = 1; i < n; i++){
            for (j = 0; j < m; j++){
                a[i][j] = a[i][j-1] * 2;
            }
        }
    }

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

    return 0;
}
```

Figure 2 – Stimulus containing a data race.

and Stevens (2018) of 1600 observations per condition. The target for the main study was to recruit 60 participants (2280 observations) to ensure adequate power. Despite our efforts to recruit participants, we were not able to meet our target. In the end we had 570 observations (38 observations per condition x 15 participants). We suspect this was due to the specific programming language and API knowledge required to complete the study.

7. Conclusion

Parallel architectures, such as multi/many-core processors, are now widely available, however, to exploit this processing power we need more programmers capable of working with parallel applications and applying parallel computing models (Gonçalves et al., 2016). The lack of programmers with this specialization may be due to its reputation of being more challenging than sequential programming (McKenney, 2017). Despite having found no empirical evidence that parallel programming is harder to learn, one possible explanation as to why may it may be perceived as such could be that the teaching strategies, models, and visualization techniques that worked for sequential programming are not transferable to the parallel programming paradigm.

The results of this study lead us to believe that expert parallel programmers' mental models formed during program comprehension given a data race task consist of the data structure components of the execution model. Therefore, visualizations that allow programmers to better understand the relationships between data structures how they are accessed and changed by one or more threads of execution could assist them in detecting data races. Visualizations of data structures should also accurately match the indexing capabilities of the programming language.

Further research is needed on the mental representations formed by programmers when interacting with parallel code to determine how best to model this for learners and to inform the development of visualizations and tools that are effective in assisting with tasks such as detecting data races. We have developed a stimuli set and experimental design that produces a reasonable level of difficulty and would be relevant for replication studies and for expanding the psychology of programming research to include parallel programmers.

8. References

Aubanel, E. (2020). Parallel program comprehension. In *31st Annual Workshop of the Psychology of Programming Interest Group (PPIG 2020)* (pp. 8–16).

Barfield, W. (1986, 1). Expert–novice differences for software: Implications for problem-solving and knowledge acquisition. *Behaviour & Information Technology*, *5*(1), 15–29. doi: 10.1080/01449298608914495

Barfield, W. (1997, 12). Skilled performance on software as a function of domain expertise and program organization. *Perceptual and Motor Skills*, *85*(3, Pt 2), 1471–1480. doi: 10.2466/pms.1997.85.3f.1471

Bates, D., Mächler, M., Bolker, B., & Walker, S. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, *67*(1), 1–48. doi: 10.18637/jss.v067.i01

Bidlake, L. (2022, Feb). *Validation of stimuli for studying mental representations formed by parallel programmers during parallel program comprehension.* OSF. Retrieved from `https://osf.io/fcnyx/?view_only=d8b98b812f164de28dd12eb29d35fd14` doi: 10.17605/OSF.IO/FCNYX

Bidlake, L., Aubanel, E., & Voyer, D. (2022). Pilot study: Validation of stimuli for studying mental representations formed by parallel programmers during parallel program comprehension. In *33rd Annual Workshop of the Psychology of Programming Interest Group (PPIG 2022)* (pp. 47–56).

Brysbaert, M., & Stevens, M. (2018). Power analysis and effect size in mixed effects models: A tutorial. *Journal of Cognition*, *1*(1). doi: 10.5334/joc.10

Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, *7*, 115–156. doi: 10.1023/A:1015297914742

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed ed.). Hillsdale, N.J.: L. Erlbaum Associates. Retrieved from `http://www.gbv.de/dms/bowker/toc/9780805802832.pdf`

Détienne, F. (2001). *Software design-cognitive aspect.* Springer Science & Business Media.

Feigenspan, J., Kastner, C., Liebig, J., Apel, S., & Hanenberg, S. (2012, Jun). Measuring programming experience. In *20th IEEE International Conference on Program Comprehension (ICPC)* (pp. 73–82). IEEE. doi: 10.1109/ICPC.2012.6240511

Fox, J., & Weisberg, S. (2019). *An R companion to applied regression* (Third ed.). Thousand Oaks CA: Sage. Retrieved from `https://socialsciences.mcmaster.ca/jfox/Books/Companion/`

Gonçalves, R., Amaris, M., Okada, T., Bruel, P., & Goldman, A. (2016, Jan). OpenMP is not as easy as it appears. In *2016 49th Hawaii International Conference on System Sciences (HICSS)* (p. 5742–5751). doi: 10.1109/HICSS.2016.710

Leopold, J., & Ambler, A. (1996, Sep). A user interface for the visualization and manipulation of arrays. In *Proceedings 1996 IEEE Symposium on Visual Languages* (p. 54–55). doi: 10.1109/VL.1996.545267

Liao, C., Lin, P.-H., Asplund, J., Schordan, M., & Karlin, I. (2017). Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 11:1–11:14). ACM. doi: 10.1145/3126908.3126958

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007, Mar). Investigating the viability of mental models held by novice programmers. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (p. 499–503). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1227310.1227481` doi: 10.1145/1227310.1227481

Maqsood, R., & Ceravolo, P. (2018, Jul). Modeling behavioral dynamics in confidence-based assessment. In *2018 IEEE 18th International Conference on Advanced Learning Technologies (ICALT)* (p. 452—454). doi: 10.1109/ICALT.2018.00112

Mattson, T. (2019). *The OpenMP Common Core: A hands on exploration.* Argonne Training Program on Extreme-Scale Computing (ATPESC).

McKenney, P. E. (2017). Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a). *CoRR*, *abs/1701.00854*. Retrieved from http://arxiv.org/abs/1701.00854

OpenMP-ARB. (2013). *OpenMP Application Program Interface Version 4.0* (Tech. Rep.). OpenMP Architecture Review Board (ARB). Retrieved from http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Peirce, J., Gray, J. R., Simpson, S., MacAskill, M., Höchenberger, R., Sogo, H., ... Lindeløv, J. K. (2019, Feb). Psychopy2: Experiments in behavior made easy. *Behavior Research Methods*, *51*(1), 195–203. doi: 10.3758/s13428-018-01193-y

Pennington, N. (1987, July). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*(3), 295–341. doi: 10.1016/0010-0285(87)90007-7

Piteira, M., & Costa, C. (2012, Jun). Computer programming and novice programmers. In *Proceedings of the Workshop on Information Systems and Design of Communication* (p. 51–53). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2311917.2311927 doi: 10.1145/2311917.2311927

R Core Team. (2021). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from https://www.R-project.org/

Shneiderman, B. (1976, 6). Exploratory experiments in programmer behavior. *International Journal of Computer & Information Sciences*, *5*(2), 123–143. doi: 10.1007/BF00975629

Srivastava, A. K. (2020). *A practical approach to data structure and algorithm with programming in c*. Oakville, ON: Arcler Press.

Storey, M.-A. (2006, 9 01). Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, *14*(3), 187–208. Retrieved from https://doi.org/10.1007/s11219-006-9216-4 doi: 10.1007/s11219-006-9216-4

Influencing Attention In Code Reading: An Eye-Tracking Study

Alan T. McCabe
Lund University
alan.mccabe@cs.lth.se

Diederick C. Niehorster
Lund University
diederick_c.niehorster@humlab.lu.se

Emma Söderberg
Lund University
emma.soderberg@cs.lth.se

Abstract

When interacting with other humans, we attempt to develop a shared understanding using various means. One such method is through our eyes: when someone is looking at something, we understand that their attention is focused on that object. In this work, we present the results of an eye-tracking study built upon the Progger tool, in which we used additional code highlighting in an attempt to influence the gaze behaviour of a human programmer, thereby focusing their attention. We found that though it is possible to draw attention towards areas of particular interest to the compiler, this has no apparent effect upon performance when confronted with a bug-finding code comprehension task. We conclude that although this strategy may be of use in the future when attempting to humanise the process of programming, further research is required to establish the efficacy of such interventions.

1. Introduction

In a series of recent research papers (McCabe, Söderberg, & Church, 2021; Church, Söderberg, & McCabe, 2021; McCabe, Söderberg, & Church, 2022), we have explored the idea of viewing the activity of programming as a "conversation" between two participants, namely the developer and their development environment. This "conversational lens" (Church et al., 2021), used as a "tool for thinking with"[1], led to the development of a prototype tool, named Progger (McCabe et al., 2021).

With Progger, we have explored the consequences of making visible the sequence of actions taken by a Java compiler in the lead up to a compiler error. This was an attempt to give the user a means of delving into the "thought process" of the compiler: much as in a human-to-human conversation a misunderstanding could be resolved by asking one participant to clearly and thoroughly explain what they are thinking about. This approach for humanising the interaction between developer and development environment centred around the usage of a linguistic-based strategy. Sections of code were highlighted and explicitly linked with descriptive text in the sidebar, which served as a summary of the computations made using attributes assigned to the code during compilation.

However, in our initial framing paper (Church et al., 2021), we also noted the use of so-called *side-channels* in human interactions. These side-channels are non-verbal cues which can greatly alter the meaning of speech, and can take on a variety of forms: body language, tone of voice, facial expressions, where a person is looking, and so forth.

Having built the Progger prototype, we conducted a pilot study (McCabe et al., 2022) with the aim of validating the design choices made in the development of the tool. Two interesting conclusions were drawn from this study: the additional descriptive text – the linguisic component – was found to be too esoteric for the target demographic of relative programming novices; and that the highlighting – the non-linguistic component – was of significant interest, drawing praise when it worked well and ire when it did not. In other words, Progger users were less interested in the somewhat abstract and complex thought process of the compiler, and much more interested in simply *where the compiler was looking.*

[1] As coined at the industry panel during PPIG 2016 in Cambridge by Steven Clarke.

Being interested in where someone or something is looking is an innate characteristic in humans. The following of another person's gaze is a behaviour that has been found to start developing at a very young age (Frischen, Bayliss, & Tipper, 2007), and is of great significance in the development of social cognition. When two humans align their gaze, it signifies a "joint attention", indicating that both participants in the interaction are focused on the same thing. By aligning the "gaze" of a compiler with that of the developer, it may be possible to encourage the development of joint attention, with a focus on the most relevant areas of code. With this new insight an updated version of Progger was produced (Progger 2.0), stripping out extraneous text information and focusing solely on highlighting as a means of conveying information.

In this paper, we present the results of an eye-tracking study conducted using Progger 2.0, with the aim of understanding how showing where a program analysis tool is "looking" can influence the gaze and code reading behaviour of a human partner in the programming conversation, and help foster joint attention.

2. Background

Most visual tasks performed by humans are bottlenecked by the foveated nature of their visual system. Because the area of highest visual acuity of the human retina only spans a few degrees, visual tasks requiring resolving fine spatial detail often also require eye movements. As such, the study of eye movements has been fruitfully used to study the moment-to-moment unfolding of cognitive processes such as reading (Rayner, 1998) and visual search (Hooge & Erkelens, 1996; Rao, Zelinsky, Hayhoe, & Ballard, 2002; Niehorster, Cornelissen, Holmqvist, & Hooge, 2019).

When viewing static visual scenes, such as pages of text or displays of programming code, humans predominantly exhibit two types of eye movements: fixations (periods during which the eye is still so that a relatively constant area of the visual scene is projected to the fovea to allow for fine visual processing) and saccades (rapid eye movements to bring gaze to the next area of interest in the scene) (Hessels, Niehorster, Nyström, Andersson, & Hooge, 2018). Eye trackers, devices that measure what a person looks at, are used to study such gaze behaviors (Holmqvist et al., 2011). In this study, like many before us (Sharafi, Soh, & Guéhéneuc, 2015; Obaidellah, Al Haek, & Cheng, 2018; Kuang, Söderberg, Niehorster, & Höst, 2023), we make use of eye trackers to study how participants read programming code.

The act of program comprehension, when a person reads and attempts to understand unfamiliar code, has been the focus of multiple studies stretching across decades of research (Crosby & Stelovsky, 1990; Feitelson, 2019). For instance, a recent study (Busjahn et al., 2015) found that novices, as opposed to experts, have a tendency to read through code in a linear fashion, like how we would read a natural language text, and exhibited short average saccade length due to their eyes moving through the code from one line to the next. By comparison, experts were found to have a greater average saccade length and lower element coverage of the code, meaning that they focused on fewer lines of code and made larger jumps between lines as they read the code in a non-linear fashion.

By highlighting multiple non-consecutive lines of code, we hypothesised that analysis tools such as Progger 2.0 may have an affect on this phenomenon. Specifically, we speculate that by visualising the non-linear compiler gaze, we may encourage the user to spend a a greater amount of time dwelling on highlighted lines, leading to the adoption of a similar gaze pattern and a more closely aligned focus of attention.

3. Method

With the aim of increasing our understanding of how showing the "attention" of a program analysis tool can influence human gaze and performance, we conducted an eye-tracking experiment. We broke down our objective into the following research questions:

RQ$_1$ Does the addition of compiler heatmap highlighting affect bug finding performance?

RQ$_2$ Does the addition of compiler heatmap highlighting affect gaze behavior when reading code?

3.1. Participants

In order to maintain consistency with the previous Progger study, as well as to reduce the number of independent variables, we decided to target novice programmers for the experiment. The participants were recruited from the pool of undergraduate computer science students at Lund University. An advertisement was initially made to students taking a course on agile software development, however the scope of the recruitment was later extended to include general advertising to the student cohort using Facebook groups. The only requirement for participation was the completion of at least one programming related class, and a total number of 15 students ultimately took part in the experiment. Of these participants, all were studying at an undergraduate level and none had any industrial experience outside of a summer internship. Participants were compensated for their participation in the form of a gift card for a cinema chain.

3.2. Stimuli

A set of eight stimuli was presented using Tobii Pro Lab, consisting of screenshots of small (8 to 17 lines) Java programs rendered within Progger, each containing at least one compiler error. A single error was identified with a red outline in the code, and the related error-message displayed in the side-bar.

Each stimulus contained either only a simple code snippet without highlighting and with only the error location indicated, or additionally contained highlighting provided by Progger indicating different lines of code which the compiler considered during computation of the error. An example stimulus showing both the non-highlighted and highlighted conditions is shown in Figure 1. Two sets of eight stimuli were created, sets A and B, each of which contained four non-highlighted code snippets and four with additional highlighting. The non-highlighted/highlighted screenshots were swapped between sets A and B. Of the 15 participants, 7 were shown stimulus set A and 8 were shown stimulus set B. The stimuli were presented in random order for each participant. It should be noted that extra whitespace was added between lines of code in order to achieve greater accuracy when determining exactly which line of code a fixation falls on.

3.3. Apparatus and Experimental Procedure

Figure 2 – The experimental setup. The apparatus is contained within a booth, with the eye-tracker visible below the screen and a chin- and forehead rest in the foreground.

The experiment was conducted on-site at the Lund University Humanities Lab, using a Tobii Pro Spectrum that recorded gaze at 600 Hz. Each participant was seated at a booth which constricted their peripheral view, and viewed the stimuli from a viewing distance of approximately 63 cm on a 52.8 x 29.7 cm (47° x 27°) computer screen (EIZO FlexScan EV2451, resolution 1920 x 1080 pixels) attached to the eye tracker while their head was placed on a chin- and forehead rest that was attached to the desk. The headrest and desk height were adjusted until the participant was comfortable. A five-point calibration procedure was executed followed by a four-point validation (mean accuracy 0.501 deg). An example of the experimental setup can be seen in Figure 2.

The eight code stimuli were presented during eight separate trials using Tobii Pro Lab. Each trial, the participants were asked to attempt to comprehend the code and determine both why the error had occurred, and how it might be fixed. Once they felt they had a good understanding of these questions, they were instructed to press any key in order to advance to a text-input screen where they were asked to summarise in their own thoughts why they believed the error had occurred, and how to resolve

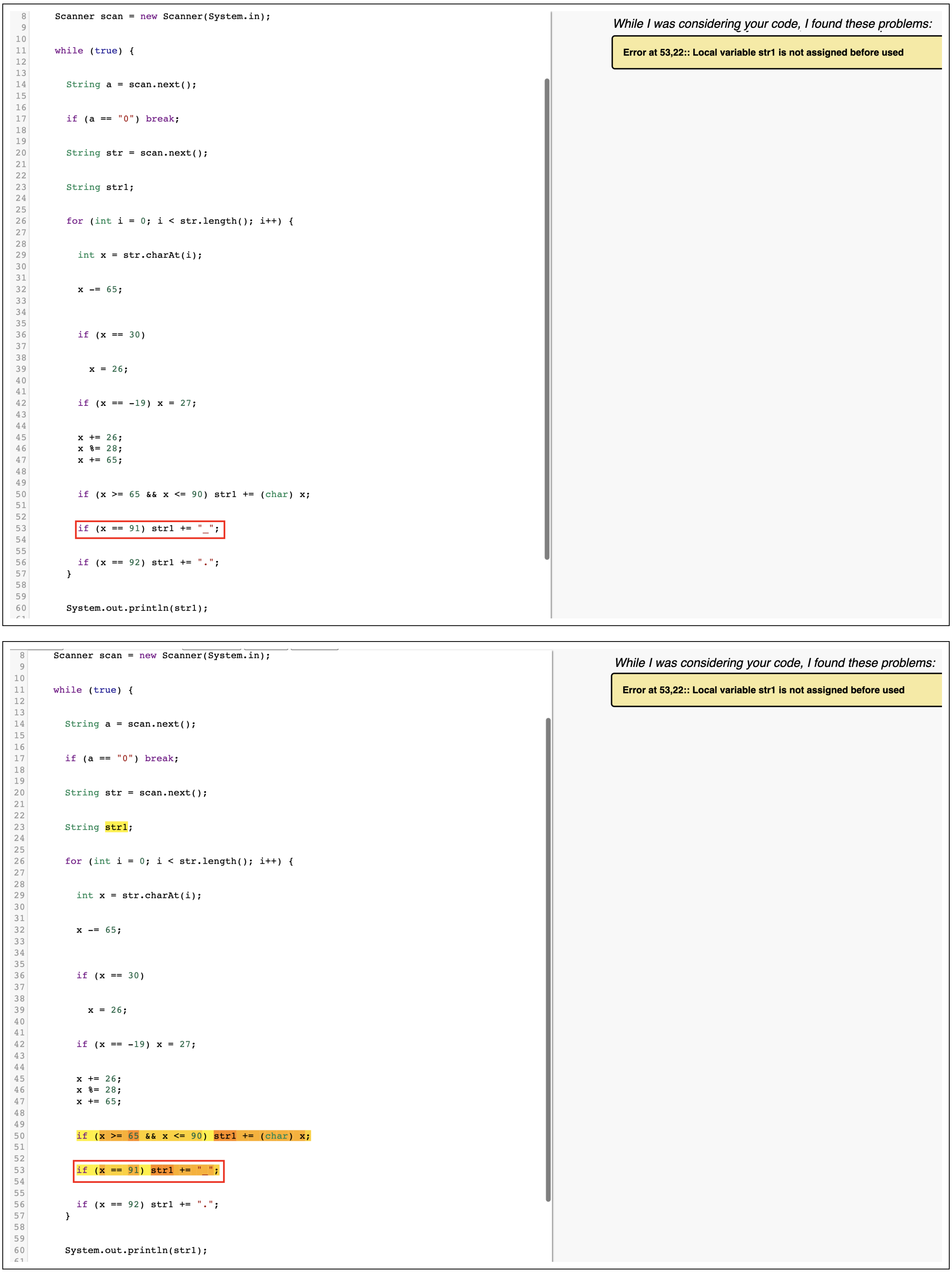


Figure 1 – An example of non-highlighted (top) versus highlighted (bottom) versions of a stimulus. In the stimulus, a compiler error has been thrown stating that a variable has not been assigned before use due to the initial assignment being dependent upon the results of a scanned input, which is indeterminate at compile time.

it. Once completed, they were able to move on to the next stimulus by pressing a key combination. In order to cater to the target demographic of novices, no advanced language constructs or external libraries were used within the code samples. The whole experiment took approximately 15 to 50 minutes to complete, depending on participant.

3.4. Data Analysis

The experiment contained one independent variable: the state of the highlighting, either turned on or off for a given stimulus. Dependent variables included areas-of-interest (AOIs) analyses. To perform these analyses, for each of the experiment stimuli, areas-of-interest (AOIs) were created for each line of code and the error message. In the event of inconsequential lines of code (for example, a trailing "}" to close a block), these lines were grouped with the AOI defined for the preceding line. An example of a stimulus with AOIs defined is provided in Figure 3.

We used Tobii Pro Lab software (Tobii AB, 2023) to classify gaze into fixations using the default fixation filter with default settings, and then to annotate for each fixation whether it was in an AOI or not.

The output from Pro Lab was analyzed using a custom Java program. The program first reads in the data for a single stimulus and from this constructs a "timeline" of AOI hits, corresponding to a list of consecutive AOI fixations. Some of the analyses listed below were performed using this timeline.

In total, six dependent variables are analyzed:

- **Time to completion:** how long a participant takes to solve a task, in seconds.
- **Correctness:** Whether the participant successfully solved a task, with a binary grading of either correct or incorrect. To compute correctness values, the documents where participants recorded their proposed solutions for each problem were analysed by the first author. Each solution was marked using a traffic light system, with red signifying an incorrect solution, yellow an incomplete solution or one where the participant demonstrated understanding but was unable to solve the issue, and green indicating a complete and correct solution. The third author then checked the proposed solution grading for correctness, and any disagreement was discussed until there was a consensus.. For statistical analysis purposes, the possible grades were then recoded to a binary format by coding incorrect or incomplete solutions as unsuccessful task solutions.
- **Hit-rate:** The percentage of AOI fixations that fell on lines that *would have been highlighted by Progger* in a given stimulus. This means that the hit-rate was calculated for the same lines regardless of whether the highlighted or non-highlighted version of a stimulus was presented. For example, in the stimulus shown in Figure 1 on line 23, the variable name is highlighted by Progger in the declaration statement: `String str1`. This AOI was therefore marked as an area of interest for both versions of the stimulus (regardless of the stimulus treatment).
- **Dwell duration:** The average amount of time the participant looked at an area of interest before moving their gaze to another part of the screen, in milliseconds. Like for hit-rate, for the dwell duration calculation only highlighted lines or lines that would have been highlighted by Progger are considered. Dwell times were computed by summing together the duration of consecutive fixations that fell in the same AOI.
- **Saccade length:** The average distance between gaze fixations while reading the code, in degrees. Specifically, the saccade classification output provided by Tobii Pro Lab was used, and the distance between the two fixations that are adjacent to each saccade computed.
- **Linearity:** The percentage of gaze movements corresponding to a linear forward change. Specifically, we counted consecutive AOI fixation pairs that represented a *single-step forward-progression* in corresponding lines. For example, consecutive AOI fixations of lines 5 and 6 would constitute a linear change and thus contribute to this metric. Examples of pairs that would not contribute are fixations on lines 5 and 7 (multiple-step) or lines 6 and 5 (backwards-progression).

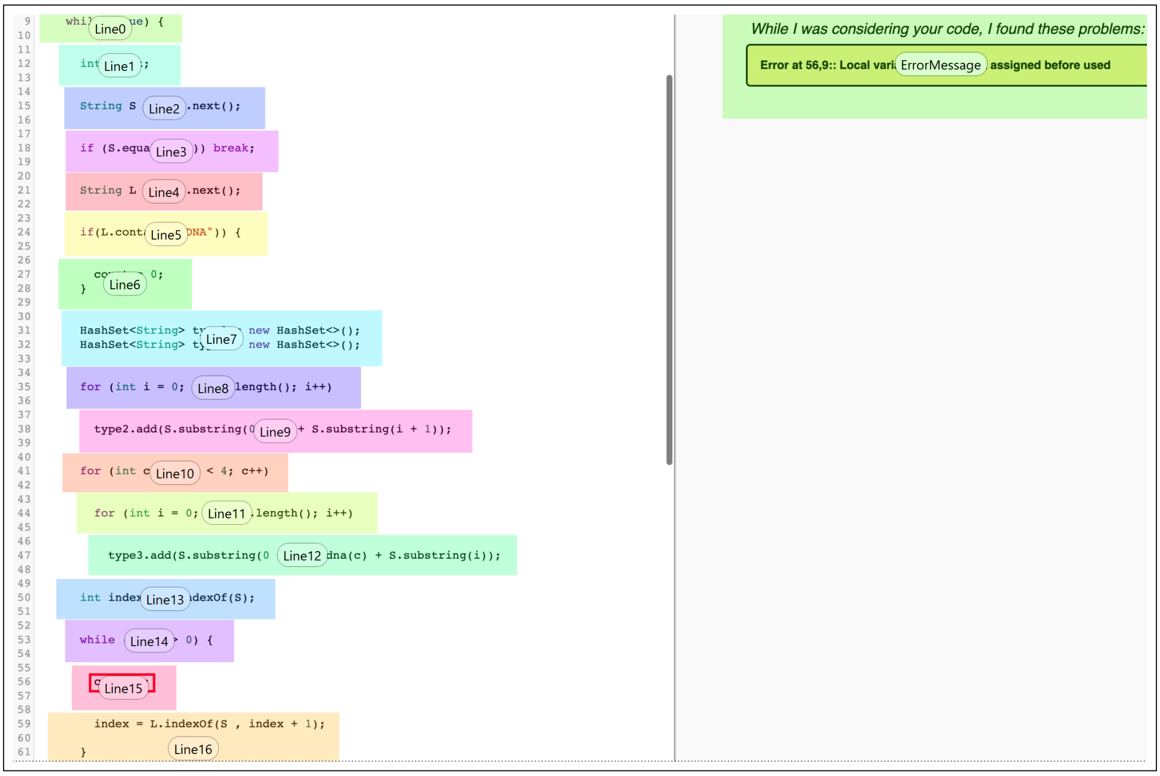


Figure 3 – The areas-of-interest for a stimulus, as defined in Tobii Pro Lab.

The data were analyzed in Jamovi version 2.3 (the jamovi project, 2022; R Core Team, 2021) with the GAMLj module (Galluci, M., 2019). Linear mixed effect modelling was performed with highlighting as a fixed effect and participant and stimulus as random effects for five out of the six dependent variables. Correctness was instead analyzed using a generalized mixed model to perform a logistic regression using a logit link function. As for the other analyses, highlighting was specified as a fixed effect and participant and stimulus as random effects. Data plots were also created using Jamovi and show data for individual participants along with the mean across participants. Error bars denote standard errors of the mean.

3.5. Threats to validity

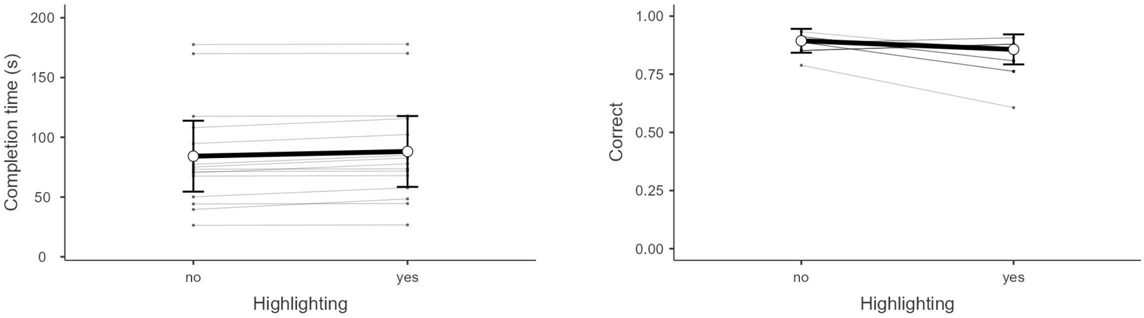
The primary threats to the validity of this study are the low **sample sizes**. To ensure the greatest accuracy of the eye-tracking equipment, we found it necessary to invite participants on-site to the Lund University Humanities Lab, which may have been a deterrent for some people as they were impelled to go out of their way to contribute. We attempted to offset this by offering an incentive, however the total number of participants was ultimately quite low at 15. To improve the internal validity of the study, it would be desirable to recruit a larger pool of participants.

Similarly, the relatively low **stimuli number** could have had an effect on the data. This led to several participants completing the tasks very quickly, however some participants also used the entire 1 hour allotted to them. For future studies, it may be desirable to increase the number of stimuli, although with the caveat that this could impact participation numbers.

Sample selection may also have affected the internal validity of the study, as only novices were recruited for participation. This led to some difficulties for specific tasks, and may have impacted the validity of the correctness metric in particular.

4. Results

Here we present the results broken down into performance and gaze metrics on AOIs.



(a) Mean **completion times** per participant of non-highlighted and highlighted stimuli.

(b) Mean **correctness** score per participant for non-highlighted and highlighted stimuli.

Figure 4 – Charts detailing the completion time and correctness metrics.

4.1. Time to complete tasks

In order to answer RQ_1, the time taken to solve each trial was analysed. As can be seen in Figure 4 (a), highlighting had no significant effect on the completion time ($F(1, 96.5) = 0.258, p = 0.613$).

4.2. Correctness

Across all participants, 101 of 120 trials (84%) yielded a correct solution. Statistical analysis was again performed to investigate whether the addition of highlighting had an effect, as can be seen in Figure 4 (b). The influence of highlighting on mean correctness was found to be insignificant ($\chi^2(1.00) = 0.401, p = 0.527$).

4.3. Highlighted line hit-rate

The mean hit-rates across all participants were then calculated and analysed, as seen in Figure 5 (a). Hit-rate was larger for highlighted stimuli than non-highlighted stimuli ($F(1, 95.8) = 10.6, p = 0.002$), indicating that highlighted lines of code are looked at more.

4.4. Dwell duration

The results of dwell time analysis can be seen in Figure 5 (b), and, consistent with the hit rate metric, exhibit an increase of 80 ms in average dwell duration between non-highlighted and highlighted versions of the stimuli ($F(1, 96.2) = 4.53, p = 0.036$). This indicates that participants spent more time looking at lines of code that the compiler considered to be of interest when the highlighted versions of the stimuli were presented to them.

4.5. Saccade length

No significant differences in saccade length were found between highlighted and non-highlighted stimuli ($F(1, 97.7) = 0.184, p = 0.669$), see Figure 5 (c).

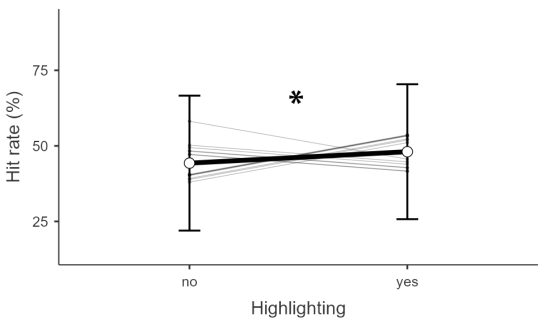
4.6. Reading linearity

As can be seen from Figure 5 (d), there was no significant difference in linearity scores between the non-highlighted and highlighted conditions ($F(1, 96.3) = 0.490, p = 0.486$). Together with the saccade length metric, this finding suggest that the way in which participants scanned the stimuli did not differ depending on availability of highlighting.

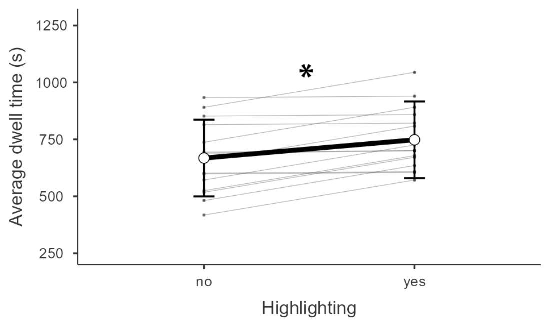
5. Discussion

Of the six dependent variables which were analysed, we used the first two (completion time and correctness) to answer RQ_1 (compiler heatmap highlighting effect on bug finding performance). We found that the addition of compiler heatmap highlighting did not have an effect on both the completion time ($p = 0.613$) and the correctness ($p = 0.527$).

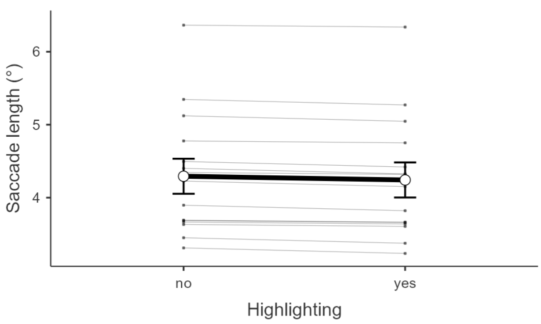
We used the analysis of dependent variables three to six (hit-rate, dwell duration, saccade length, linearity) to answer RQ_2 (compiler heatmap highlighting effect on gaze behaviour when reading code).



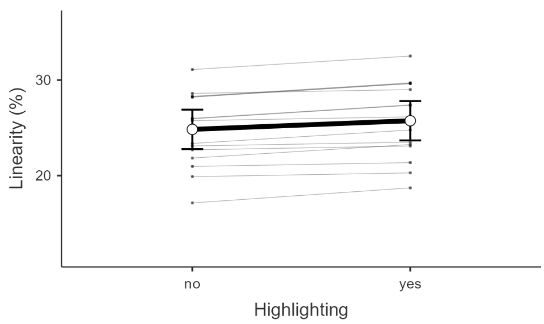
(a) Mean AOI **hit-rate** per participant of non-highlighted and highlighted stimuli.



(b) Mean AOI **dwell time** per participant of non-highlighted and highlighted stimuli.



(c) Mean **saccade length** per participant of non-highlighted and highlighted stimuli.



(d) Mean **linearity score** per participant of non-highlighted and highlighted stimuli.

Figure 5 – Charts detailing the collected eye-tracking data. An asterisk (*) indicates a statistically significant effect of highlighting change.

Of these variables, the closely related metrics of saccade length and linearity displayed no significant difference between the two highlighting conditions. This may suggest that, in the act of program comprehension, novice programmers exhibit similar patterns of gaze movements when reading unfamiliar code, regardless of additional compiler heatmap highlighting.

By contrast, the metrics relating to where (AOI hits) and for how long (dwell duration) the participants looked show a marked difference. From the data, we see that by adding highlighting to a line, novice programmers tend to look more often, and for longer periods of time, at that line. Despite this, as related in the discussion on RQ_1, this difference in gaze behaviour has no significant effect on either time to complete a task or correctness of the solution when presented with a code comprehension problem.

The effect of higher hit-rate and dwell duration ultimately warrants further investigation, as there were several confounding factors which may have influenced the results. It may be that the experimental setup being focused on comprehension was not conducive to the highlighting aiding the participants: a desire to understand the code accurately may have led participants to reading in a very methodical way. We believe that in the future it would be worth studying the effect of compiler heatmap highlighting in different contexts, such as when examining control flow, or when investigating a bug in an already familiar code base. The relatively low number of participants may also have affected our ability to detect effects of highlighting, and it may be worth attempting to re-run the study with a larger number of subjects.

The low level of experience across all participants may also have fed into the results due to their lack of familiarity with certain features of the Java language. For instance, the stimulus which received most incorrect solutions (6 out of 15 responses, or 40%) was centred around the use of the `final` keyword. Some participants were unsure how this would affect the mutability of the relevant variable, and thus offered incorrect solutions.

Considering these potentially confounding factors, it may be of worth to run a follow-up study with steps taken to mitigate their effects. A larger pool of participants, selected from both novices and experts, and a lesser focus on code comprehension within the study may lead to further interesting findings.

Despite these factors, the experiment led to interesting findings when considering the concept of joint attention. The method in which the heatmap highlighting is computed (as described in past papers, (McCabe et al., 2021, 2022) is based on the lines which are considered by the compiler in the code analysis resulting in a found error. When no highlighting exists on a line, the compiler did not consider it to be of importance. In contrast, the darkness of the highlighting on a given line or term is related to the number of times the compiler "gaze" passed over this area of code. The fact that participants looked at highlighted sections more often, and for longer periods, shows that this visualisation of compiler attention had a notable effect on user attention.

One of the main motivations in developing the conversational lens for analysing interactions with a programming environment was to draw upon human characteristics to make the process more natural. We believe that in moving towards more natural, instinctive methods of communication, many of the abstractions of human-computer interaction, for example hiding a complex compilation process behind a simple error message, can be made more clear. In this study we have shown that, although no effects were found on performance in this experimental setup, it is indeed possible to use interaction design to encourage the very human phenomenon of joint attention with a computer.

6. Acknowledgements

The authors gratefully acknowledge Lund University Humanities Lab for providing the facilities for the experiment, and would like to thank Prof. Martin Höst for early feedback on the experiment design. This work has been partially supported by the Swedish Foundation for Strategic Research (grant no. FFL18-0231), the Swedish Research Council (grant no. 2019-05658), ELLIIT - the Swedish Strategic Research Area in IT and Mobile Communications, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

7. References

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., … Tamm, S. (2015). Eye movements in code reading: Relaxing the linear order. In *2015 ieee 23rd international conference on program comprehension* (p. 255-265). doi: 10.1109/ICPC.2015.36

Church, L., Söderberg, E., & McCabe, A. (2021). Breaking down and making up-a lens for conversing with compilers. In *Psychology of programming interest group annual workshop (ppig).*

Crosby, M., & Stelovsky, J. (1990). How do we read algorithms? a case study. *Computer*, *23*(1), 25-35. doi: 10.1109/2.48797

Feitelson, D. G. (2019). Eye tracking and program comprehension. In *2019 ieee/acm 6th international workshop on eye movements in programming (emip)* (p. 1-1). doi: 10.1109/EMIP.2019.00008

Frischen, A., Bayliss, A. P., & Tipper, S. P. (2007). Gaze cueing of attention: visual attention, social cognition, and individual differences. *Psychological bulletin*, *133*(4), 694-724.

Galluci, M. (2019). *Gamlj: General analyses for linear models.* Retrieved from `https://gamlj.github.io/`

Hessels, R. S., Niehorster, D. C., Nyström, M., Andersson, R., & Hooge, I. T. C. (2018). Is the eye-movement field confused about fixations and saccades? a survey among 124 researchers. *Royal Society Open Science*, *5*. doi: http://doi.org/10.1098/rsos.180502

Hooge, I. T. C., & Erkelens, C. J. (1996). Control of fixation duration in a simple search task. *Perception & Psychophysics*, *58*(7), 969-976. doi: 10.3758/BF03206825

Kuang, P., Söderberg, E., Niehorster, D. C., & Höst, M. (2023). Towards gaxe-assisted developer tools. In *International conference on software engineering: New ideas and emerging results (icse-nier).*

McCabe, A. T., Söderberg, E., & Church, L. (2021). Progger: Programming by errors (work in progress). In *Psychology of programming interest group annual workshop (ppig).*

McCabe, A. T., Söderberg, E., & Church, L. (2022). Visual cues in compiler conversations. In *Psychology of programming interest group annual workshop (ppig).*

Niehorster, D. C., Cornelissen, T., Holmqvist, K., & Hooge, I. (2019). Searching with and against each other: Spatiotemporal coordination of visual search behavior in collaborative and competitive settings. *Atten Percept Psychophys*, *81*, 666-683. doi: https://doi.org/10.3758/s13414-018-01640-0

Obaidellah, U., Al Haek, M., & Cheng, P. C.-H. (2018). A survey on the usage of eye-tracking in computer programming. *ACM Comput. Surv.*, *51*(1).

R Core Team. (2021). *R: A language and environment for statistical computing.* Retrieved from `https://cran.r-project.org`

Rao, R. P., Zelinsky, G. J., Hayhoe, M. M., & Ballard, D. H. (2002). Eye movements in iconic visual search. *Vision Research*, *42*(11), 1447-1463. doi: https://doi.org/10.1016/S0042-6989(02)00040-8

Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin*, *124*(3), 372-422. doi: 10.1037/0033-2909.124.3.372

Sharafi, Z., Soh, Z., & Guéhéneuc, Y.-G. (2015). A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology*, *67*, 79-107.

the jamovi project. (2022). *jamovi.* Retrieved from `https://www.jamovi.org`

Tobii AB. (2023). *Tobii pro lab.* Retrieved from `http://www.tobii.com/`

Towards a definition of the concept of logic of an algorithm

Federico Gómez
Instituto de Computación -
Facultad de Ingeniería
Universidad de la República
fgfrois@fing.edu.uy

Sylvia da Rosa
Instituto de Computación
Facultad de Ingeniería
Universidad de la República
darosa@fing.edu.uy

Abstract

Within the framework of an empirical study carried out with students of an introductory programming course of a Computer Engineering Under-graduate Program, the students were asked to design an algorithm to solve the linear search problem and implement the solution, using two imperative programming languages (Pascal and C++) in two different groups of students. It was found that the solutions proposed by students followed different strategies (different "logics") to solve the problem. This fact led to the need to propose a definition of the concept of logic of the algorithm, as far as we know, not found in the literature, as well as to explore the reasons that lead each student to follow one strategy or another.

This article describes the activities carried out during the study and includes selected excerpts of students' responses. The results are analyzed in the framework of the theory of the investigation (Genetic Epistemology of Jean Piaget) with the focus on students solutions according to the proposed definition of logic of the algorithm. Conclusions and some future work that may eventually lead to a more exact definition are included.

1. Introduction

In 2022, we carried out an empirical study that investigates the construction of knowledge about the linear search problem, its algorithmic solution, and the implementation and execution of a program that solves it on a computer (da Rosa & Gómez, 2022). The study had as the main goal to investigate the process of students' thinking from solving a concrete instance of the problem to the writing of a program that implemented the search for a given value in an array of integers. The students were asked to explain how and why the program worked when executed on a computer, both at the level of its source code (the textual part of the program) and its execution on a machine (the executable part of the program). The dual nature of a program (textual and executable) was disscused in (da Rosa,S. & Chmiel,A. & Gómez, F., 2016). As for other empirical studies in which we investigate knowledge construction of several computing concepts, the study is based on principles of Jean Piaget's Genetic Epistemology theory (Piaget, 1977; Piaget & Garcia, 1980). In (da Rosa & Aguirre, 2018) the general law of cognition that Piaget describes to explain the construction of algorithmic knowledge (Piaget, 1964), is extended to encompass the construction of computational knowledge (da Rosa, 2018). The extended law of cognition is used in detail in (da Rosa & Gómez, 2022) to explain the process of students' thinking from solving the concrete instance of the problem (instrumental knowledge), to explaining their algorithms (conceptual knowledge) and writing the general program (formal knowledge). Besides the expected results, we observed an additional fact that we had not initially foreseen in the study design: when writing down the steps of the algorithm that solved the problem, some students used a certain strategy to implement the search, while others employed a different strategy. In order to easily identify them in the context of the study, we decided to name them the *a priori* strategy and the *a posteriori* strategy (see Section 3. Each strategy follows its own "logic", in the sense that it executes the steps that solve the problem in a different order than the other strategy. This made us formulate two investigation questions:

- First, what are the reasons that led each student to propose one or the other when implementing their solution, given that both the instructions and the activities proposed in the study are the same for all students.
- Second, whether there is a formal definition of the concept of "logic" of an algorithm, as the

term is often used informally to refer to the behavior given by the steps that define its algorithmic strategy and distinguish it from other strategies (other "logics").

The rest of this paper is organised as follows: Section 2 presents a summary of the empirical study on the linear search. Section 3 analyses possible responses to the questions above. Finally, in section 4, some conclusions are presented, as well as some lines of future work that could lead to a more precise definition of the concept of logic of an algorithm. Finally, references are included.

2. Summary of the empirical study

The first part of the study focuses on the process of students' thinking from solving a concrete instance of the problem of linear search to the writing of an explanation about how they did and why their methods solves the problem. According to Piaget's general law of cognition (Piaget, 1964), the success of students in doing that is evidence of the construction of conceptual knowledge about the algorithm.

The second part of the study examines the process of students writing a program from their explanations and executing it on a computer, which according to the extension of the aforementioned law is evidence of the construction of formal knowledge (da Rosa & Aguirre, 2018).

Thirteen students from an introductory programming course participated in the study: seven students working with C++ and six students working with Pascal. The topics covered in the courses were identical for both groups, the only difference was the programming language used. All activities were carried out individually by the students.

The concrete instance of the linear search problem is presented to the students in the form of searching for ID numbers in a row of numbered cards representing door numbers of houses on a street, as shown in Figure 1. Each card has a second card underneath, representing the ID number of the person living in that house, as shown in Figure 2. It is assumed that only one person lives in each house. The ID numbers are not ordered and hidden from the student's view. The student is asked to search for a number that is in the row and for another one that is not.

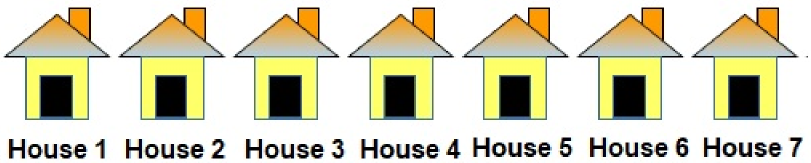


Figure 1 – Simulation with houses

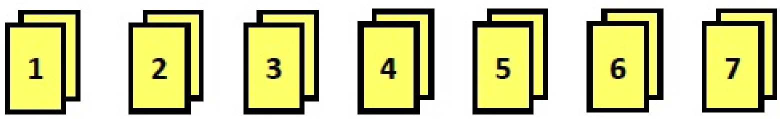


Figure 2 – Simulation with cards

Students successfully solve the problem working with the row of cards and answer questions aimed at obtaining accurate descriptions of the method used and why they were successful, providing a written description in natural language. As an example, the description in natural language given by one of the students is shown below. *Assuming that you want to find the person with ID number x, you go through the doors in order, opening them and asking the person behind them for their ID number. If it's the one we were looking for, the search ends, otherwise, we go to the next door. If the doors are visited and the person with ID number x is not found, it is deduced that he/she does not reside behind any of the doors.*

The student expresses all necessary actions for solving the problem, comparison (*asking the person behind them for their ID number. If it's the one we were looking for*), advance (*we go to the next* *door*) and repetition (implicitly in the description of actions referring to the doors, in plural). He also describes the changes imposed on objects, he stops both when he finds the ID number (*If it's the one we were looking for, the search ends*) and when there are no more doors to visit (*it is deduced that he/she does not reside behind any of the doors*) (da Rosa & Gómez, 2022).

Once all students write similar descriptions, they are asked to write a version of the algorithm using pseudocode, based on her/his description in natural language. Pseudocode allows an external agent (in this case, an imaginary robot played by the interviewer) to execute it, which helps the student visualise the behaviour of the algorithm and correct errors, leaving for later aspects related to machine execution. This tool is called **automation** and induces the student to reflect on errors that may be detected and correct them. The student is asked to write multiple progressive versions of the algorithm until finally arriving at a correct one.

An example of a process with several versions is shown below.

```
while (not find id number)
    ask for id number in door
    if it's the one I look for
        stop
    else
        end search
    end
end
```

In this version, the student expresses two of the three actions: comparison (if it's the one I look for) and repetition (while), while omitting the action of advancing to the next door (he writes "end search" after else). Regarding the condition for the while loop, the student intends for the external agent to stop searching upon finding the desired number, but does not consider the case where there are no more doors remaining. The student's primary focus lies on achieving the desired result. He has conceptualised the algorithm after executing it by himself but needs to further conceptualise the action to be taken when the number is not found behind the current door. In this case a second condition for stopping the iteration when there are no more doors left has to be included. This student needs to write three more versions before finally coming up with a correct one. Between each version and the next, automation was used to detect and fix various errors. The final (and correct) version of this student's pseudocode is shown below.

```
while (not find id number) and (there are more doors)
    ask for id number in door
    if it's the one I look for
        stop
    else
        go to the next door
    end
end
```

Given that the goal is to write and execute on a computer a solution to the more general problem of searching for a value in an array with N cells, each student first writes some code snippets to gain familiarity with the syntax and semantic rules associated with arrays in her/his respective programming language (C++ or Pascal) before continuing with the writing of the program.

In the final activity, each student writes, compiles and executes a program that searches for a given value in an array of N integers, based on their previous pseudocode. The focus is on the correspondence between the steps of the pseudocode and the program instructions (knowledge about the *textual* part of the program) and on aspects specific to machine execution (knowledge about the *executable* part of the program) (see Section 1).

Automation is used to detect and correct errors, using a paper-based array to visualise the execution, guiding the student to consider machine execution issues, such as an infinite loop or an out-of-range index error.

As with the pseudocode, each student needs to write more than one version of the program prior to arriving at a correct one. As an example, the final C++ version of the same student included above is shown below.

```
boolean found = FALSE;
int i = 0;
while ((found == FALSE) && (i <= N-1)) {
    if (arre[i] == number) {
        found = TRUE;
    } else {
        i = i+1;
    }
}
```

The student establishes a suitable correspondence between the steps in pseudocode and the instructions in the program. At the end of the study, all the students (both those who worked with C++ and those who worked with Pascal) successfully wrote a correct program, compiled and executed it on a computer.

3. Different logics within the implemented algorithms

Analysing students' algorithms using pseudocode, we found that the final version written by each student follows one (and only one) of the following search strategies, that we call *a priori* and *a posteriori*.

- A priori: the student consults the ID number at the first door before starting the iteration. If the searched ID number is not in the row, she/he ends up positioned at the last door at the end of the search.
- A posteriori: the student consults the document at the first door after starting the iteration (i.e., within the iteration itself). If the searched document is not in the row, she/he advances once more after consulting the last door and only then finishes the search.

Five students follow the *a priori* strategy in their final version, while the remaining eight follow the *a posteriori* strategy. In some cases, the employed strategy is already noticeable in the initial version, while in others, it becomes evident as their versions progress. When designing the study, no specific consideration is given to either of the two strategies. The arrival of each student at one of them is observed during the implementation. The students who follow the *a priori* strategy never position themselves beyond the last door. Whether the searched document is in the last door or it doesn't exist in the row, they are equally positioned at the last door, with the iteration stopping based on either one of the conditions of the while loop depending on whether the document is found or not. For example, the following is the implementation of this strategy by student 1:

```
Check ID at Door 1
While (there are more Doors) AND NOT (it is the ID I'm looking for)
    Check ID at next Door
End
```

The students who follow the *a posteriori* strategy advance once more after visiting the last door when the searched document is not in the row. However, they appropriately control the termination, avoiding checking for the document at a door that doesn't exist. For example, student 2 follows this strategy in his final version:

```
While (there are doors in the block) AND (haven't found the person)
    Knock on the door
    Ask for the person's ID
    If (it is the person I'm looking for)
        Finish the search
    Else
        Go to the next door
    End
End
```

Informally, we refer to the algorithm "logic" as the order of execution of the steps expressed in pseudocode, without yet considering aspects of their subsequent implementation in a programming language. This involves, for example, the choice of a data structure to be used in the program. In the study, linear search on an array is chosen for implementation, but it could also be done on a linked list. In both implementations, the data structure varies, but the algorithm logic remains the same. What determines the algorithm logic is the behaviour produced when the external agent executes the pseudocode steps in the defined order. However, the observation of both strategies is something that drew our attention, as all students were given the same instructions and the study design was never intended to induce one strategy or the other.

In the program writing phase, each student maintains the logic expressed in their pseudocode algorithm, as shown below.

- Student 1 (Pascal language, *a priori* strategy, array indices ranging from 1 to N)

  ```
  door := 1;
  WHILE (Door <= N) AND NOT (id = arre[Door]) DO
      door := door + 1;
  ```

- Student 2 (C++ language, *a posteriori* strategy, array indices ranging from 0 to N-1).

  ```
  boolean found = FALSE;
  int i = 0;
  while ((i ≤ N-1) && (!found))
  {
      if (arre[i] == ID)
          found = TRUE;
      else
          i++;
  }
  ```

It can be observed that the *a priori* programs do not use selection (if/else) or boolean variables, whereas the *a posteriori* programs do use both elements.

A remarkable fact is that almost all students working with Pascal follow the *a priori* strategy (only one follows the a posteriori strategy) and all the students who used C++ follow the *a posteriori* strategy. However, the definition of the search strategy emerged when specifying the logic of the algorithm during the **pseudocode** writing phase where the thirteen students used the exact same pseudocode syntax rules, **prior** to the writing of the code in the programming language. Even more, both strategies can be implemented in both languages, which allow working with the same elements (integer and boolean variables, if/else and while structures).

This led us to wonder if prior knowledge of the formal language could influence the student's thinking when constructing knowledge about the logic of a new algorithm before its formalisation. At the beginning of the study, each student only knew the language used in their group, and none of them had worked with arrays before. They had all worked with the same elements (variables, basic data types, expressions and simple instructions, selection structures, and iteration structures).

To find an explanation for the observations, we reviewed the curricula of both groups and found a single difference that may be relevant. In the Pascal course much emphasis is placed on the difference between short-circuit and complete-circuit evaluation of boolean expressions, because part of the bibliography is a book describing standard Pascal (Konvalina & Wileman, 1987), that evaluates boolean expressions with complete-circuit, while the compiler used in the course is the Free Pascal compiler that evaluates boolean expressions using short-circuit. Because of that, the students are encouraged to evaluate using short-circuit, ignoring that part of the course book. The C++ group works with Code::Blocks and although the difference between complete and short circuit as well as the fact that Code::Blocks evaluates by default using short-circuit were known, no especial emphasis is placed on the topic. Therefore, students in C++ may unconsciously assume complete-circuit, tending to use boolean variables and if/else statements while Pascal students may have a greater awareness of short-circuit, avoiding the use of boolean variables and if/else statements. These observations are an example of the impact that prior knowledge constructed in different domains has into mental schemas and how it becomes integrated in the process of construction of new concepts (Cellerién, 1987).

In the context of the study, although students define the algorithm logic in the pseudocode stage, they had become acquainted with short-circuit and complete-circuit evaluation. When defining the logic in pseudocode the Pascal group integrated the short-circuit into their mental schema of the new concept, due to the emphasis mentioned above. Anyway, these are primary explanations that deserve to be deepened through future studies, in the same way that the expression *logic of the algorithm* deserves a formal definition, as posed as the investigation questions in Section 1.

4. Conclusions and further work

Within the framework of the study, we propose defining the concept of *logic of an algorithm* as the order of execution of the steps expressed in pseudocode, before considering aspects of its implementation in a programming language. We consider that this definition is insufficient because it does not express the concept with the desired accuracy. The logic of an algorithm concerns not only the order of the execution of the steps, but there is something underlying that leads a person to propose one order over another when devising the steps of the algorithm. We found an explanation related to what was observed in this study with the *a priori* and *a posteriori* search strategies.

However, we believe that it is necessary to further investigate how individuals construct the logic of an algorithm prior to its formalization in general, which could be linked to the logic of actions and significations developed by Piaget and García in (Piaget & García., 1987). These investigations would complete a cycle of our work that focuses on studying the construction of knowledge about algorithms, data structures, and programs. In this context, understanding the construction of the logic of an algorithm is fundamental since it permeates the entire construction process, from its genesis at an instrumental level, through the conceptual level, and reaching the formal level. In all the studies conducted, it was observed that this construction is complex. The evidence collected shows that previously constructed schemas influence this construction (Cellerién, 1987), including formal knowledge, as observed in the study with the evaluation of boolean expressions using short-circuit or complete-circuit.

During this work, we reviewed the academic literature for any definition of the concept (even under other denominations that do not involve the word "logic") that aligned with the notion proposed in Section 3, but we did not find any. The closest we found is Kowalski's proposal in (Kowalski, 1979): "An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used." However, this proposal does not reflect the same notion that we use, because it associates the idea of logic with specification rather than behaviour, associating the latter with what he calls control. Furthermore, in the rest of the work, Kowalski links both notions (logic and control) with the data structures used in the implementation, deviating from the idea that the logic of an algorithm does not depend on the structure on which it is subsequently implemented in a programming language as mentioned in Section 3) for the case of the algorithm that solves the linear search, that has the same logic whether implemented on an array or a linked list.

The contrast between the difficulty in finding a definition of the concept of logic of an algorithm and its underlying presence in all the studies conducted during the elaboration of the model for investigating the construction of knowledge of programs (da Rosa & Gómez, 2019) leads us to think that obtaining empirical data may eventually lead to the emergence of a more precise definition. In our opinion, this is closely linked to the absence of a more precise definition of the concept of computational thinking. Since Wing popularized a characterization of it involving solving problems, designing systems, and understanding human behaviour, by drawing on the concepts fundamental to computer science (Wing, 2008), the literature has proposed multiple attempts to define the computational thinking, all of which are incomplete, as mentioned by da Rosa in (da Rosa, 2018). In the same work, da Rosa suggests that an adequate definition of the concept could arise from an extension of the general law of cognition defined in the same paper to explain the construction of formal knowledge. It is believed that delving deeper into this line of work could provide more precise and complete definitions for both concepts, while also allowing for a deeper explanation of how individuals construct knowledge about algorithms, data structures, and programs.

5. References

Cellerièr, G. (1987). *Structures and Functions in Piaget today.* Lawrence Erlbaum Associates Publishers.

da Rosa,S. & Chmiel,A. & Gómez, F. (2016). Philosophy of Computer Science and its Effect on Education - Towards the Construction of an Interdisciplinary Group. *Special edition of the CLEI Electronic Journal (see http://www.clei.cl/cleiej/), Volume 19 : Number 1 : Paper 5.*

da Rosa, S. (2018). Piaget and Computational Thinking. *CSERC '18: Proceedings of the 7th Computer Science Education Research Conference*, 44–50. `https://doi.org/10.1145/3289406.3289412`.

da Rosa, S., & Aguirre, A. (2018). Students teach a computer how to play a game. *LNCS of The 11th International Conference on Informatics in Schools ISSEP 2018* .

da Rosa, S., & Gómez, F. (2019). Towards a research model in programming didactics. *Proceedings of 2019 XLV Latin American Computing Conference (CLEI)*, 1–8. doi: 10.1109/CLEI47609.2019

da Rosa, S., & Gómez, F. (2022). The construction of knowledge about programs. *Proceedings of PPIG 2022 - 33rd Annual Workshop*, 1–8.

Konvalina, J., & Wileman, S. (1987). *Programming with Pascal.* McGraw-Hill Book Company, ISBN 0-07-035224-0.

Kowalski, R. (1979). Algorithm = Logic + Control. *Communications of the ACM, vol 22, nº 7*, 424–436.

Piaget, J. (1964). *La prise de conscience.* Presses Universitaires de France.

Piaget, J. (1977). Genetic Epistemology, a series of lectures delivered by Piaget at Columbia University, translated by Eleanor Duckworth. *Columbia University Press.*

Piaget, J., & Garcia, R. (1980). *Psychogenesis and the History of Sciences.* Columbia University Press, New York.

Piaget, J., & García., R. (1987). *Hacia una lógica de significaciones.* Gedisa, ISBN 9784743266640.

Wing, J. (2008). Computational thinking and thinking about computing. *Philosophical transitions of the Royal Society*, *Phil. Trans. R. Soc. A 366,*, 3717–3725.