

PPIG 2024

Proceedings of the 35th Annual Workshop of the Psychology of Programming Interest Group

organised in conjunction with VL/HCC 2024

2 - 6 September 2024

Liverpool University, UK & online



Edited by
Mariana Marasoiu, Luke Church

Editors' preface

Thank you to all who participated at this year's workshop. It was PPIG's 35th annual meeting, and what better way to celebrate this than to get together with VL/HCC and gather everyone for a week of exciting events. The University of Liverpool generously hosted us all, and we enjoyed Liverpool's fare and sights.

We continued the precedent set in the past few years of enabling remote attendance for the paper presentations parts of the programme, which has enabled authors who otherwise would not have been able to join us to participate.

Many thanks go to the VL/HCC 2024 organising committee, in particular to Andrew Fish, Dulaji Hidellaarachchi, Mattia De Rosa, Anita Sarma and John Grundy for generously integrating VL/HCC's programming and publicity with PPIG. Kind thanks to Alexei Lisitsa, Safa'a Fallatah and Nouf Aljuaid for incredibly helpful local organising.

Thank you to Michael Coblenz and Margaret Burnett for helping us organise a wonderful joint Graduate Consortium, which exceeded expectations in numbers of submissions and needed us to get creative in supporting many promising doctoral students, both in-person on the first day of the conference week (Monday 2nd September) and online a few days before the conference with a few students who couldn't travel (Friday 30th August).

The paper presentation sessions were graciously chaired by Alan Blackwell, Dulaji Hidellaarachchi, Caitlin Kelleher, Judith Good, Tom Beckmann, Andreas Bexell and Michael Lee - thank you for enabling a convivial atmosphere. Special thanks to Tom Beckmann and Andreas Bexell for stellar tech and hybrid support.

And finally, thanks to all the authors, attendees, and contributors who collectively brought PPIG 2024 to life. We hope to see you, and diverse newcomers at PPIG 2025!

Mariana Marasoiu & Luke Church
October 2024

PPIG 2024

Programme & Proceedings Index

Monday, 2nd September

8:45 - 17:15 VL/HCC + PPIG Joint Graduate Consortium (invite-only meeting)

Tuesday, 3rd September

9:00 - 17:00 VL/HCC research presentations

Wednesday, 4th September

9:00 - 17:00 VL/HCC research presentations

Thursday, 5th September

8:30 - 9:30 *Coffee and registration*

9:00 - 10:30 *Joint PPIG / VL/HCC Keynote - hybrid*

PPIG 2024 Welcome

Luke Church

Keynote: **Computational Ekphrasis - Reflections on generative modes of cultural production** 7

Daniel Chávez Heras

10:30 - 11:00 *Coffee break*

11:00 - 12:30 *Presentations session - hybrid*

Ghost in The Paper: Player Reflex Testing with Computational Paper Prototypes 8

Tom Beckmann, Eva Krebs, Leonard Geier, Lukas Böhme, Stefan Ramson, Robert Hirschfeld

VL/HCC **RULER: Prebugging with Proxy-Based Programming**

Alexander Repenning, Ashok Basawapatna

VL/HCC **Where Are We and Where Can We Go on the Road to Reliance-Aware Explainable User Interfaces?**

José Cezar de Souza Filho, Rafik Belloum, Kathia Oliveira

Designing A Multi-modal IDE with Developers: An Exploratory Study on Next-generation Programming Tool Assistance 20

Peng Kuang, Emma Söderberg, Martin Höst

12:30 - 14:00 *Lunch*

14:00 - 15:30 *Presentations session - hybrid*

VL/HCC **Age-Inclusive Integrated Development Environments for End-Users**

Katharine Kerr, Reid Holmes

VL/HCC	ScrapeViz: Hierarchical Representations for Web Scraping Macros Rebecca Krosnick, Steve Oney	
	Assessing Consensus on Developers' Views on Code Readability Agnia Sergeyuk, Olga Lvova, Sergey Titov, Anastasiia Serova, Farid Bagirov, Timofey Bryksin	37
VL/HCC	Unfold: Enabling Live Programming for Debugging GUI Applications Ruanqianqian (Lisa) Huang, Philip Guo, Sorin Lerner	
15:30 - 16:00	<i>Coffee break</i>	
16:00 - 17:30	<i>Presentations session - hybrid</i>	
VL/HCC	Knotation: Supporting Exploration in Macrame Textile Crafting Through Parametric Motif Design Yanchen Lu, Tobias Höllerer, Jennifer Jacobs	
	For Modeling Programmers as Readers with Cognitive Literary Science Rijul Jain	45
VL/HCC	What Makes a Great Example Gallery? Junran Yang, Andrew McNutt, Leilani Battle	
VL/HCC	VL/HCC 2024 Closing Andrew Fish, Anita Sarma, John Grundy	
18:30 onwards	<i>In-person</i> PPIG Conference dinner @ The Philharmonic Dining Rooms	

Friday, 6th September

8:30 - 9:30	<i>Coffee and registration</i>	
9:00 - 10:30	<i>Presentations session - hybrid</i>	
	Predictability of identifier naming with Copilot: A case study for mixed-initiative programming tools Michael Lee, Advait Sarkar, Alan Blackwell	47
	Further Evaluations of a Didactic CPU Visual Simulator (CPUVSIM) Renato Cortinovis, Tamer Mohamed Abdellatif, Devender Goyal, Luiz Fernando Capretz	69
	Exploring Teachers' Perspectives on Navigating Recursion Pedagogies Jude Nzemeke, Marjahan Begum, Jo Wood	77
10:30 - 11:00	<i>Coffee break</i>	
11:00 - 12:30	<i>Presentations session - hybrid</i>	
	Understanding APIs and the software that provides them - Analysis of programmers' API mental models used in programming tasks Ava Heinonen	90
	Analysing Open Source Software to Better Understand Long Term Memory Structures in the Human Brain Thomas Mullen	102

	Designing a didactic model for programs and data structures	112
	Federico Gómez, Sylvia da Rosa	
	Craft Ethics - Aiming for Virtue in Programming with Generative AI	123
	Martin Jonsson, Jakob Tholander	
12:30 - 14:00	<i>Lunch</i>	
14:00 - 15:30	<i>Presentations session - hybrid</i>	
	Educational Tools for Probabilistic Machine Learning Curriculum in Schools	134
	Josephine Rey, Alan Blackwell, Xinyue Li, Gemma Penson, Hong Ge, Helen Arnold	
	Automatic Bias Detection in Source Code Review	144
	Yoseph Berhanu Alebachew, Chris Brown	
	Ethical Integration in Computer Science Education: Leveraging Open Educational Resources and Generative Artificial Intelligence for Enhanced Learning	152
	Ranjidha Rajan, Renato Cortinovis	
	Intention is All You Need	160
	Advait Sarkar	
15:30 - 16:00	<i>Coffee break</i>	
16:00 - 17:30	<i>Presentations session - hybrid</i>	
	How Do Developers Approach Their First Bug in an Unfamiliar Code Base? An Exploratory Study of Large Program Comprehension	174
	Andreas Bexell, Emma Söderberg, Christofer Rydenfält, Sigrid Eldh	
	PUX Explorer: An Interactive Critique and Ideation Tool for Notation Designers	186
	Justas Brazauskas, Alan Blackwell	
	Boxer Sunrise Development Update and Demos	204
	Steven Githens	
	PPIG 2024 Prizes and Closing	
	Mariana Marasoiu, Luke Church	

Computational Ekphrasis Reflections on generative modes of cultural production

Daniel Chávez Heras
King's College London
daniel.chavez@kcl.ac.uk

Keynote abstract

Contemporary generative AI systems enable a mode of visual production through verbal interaction. Multi-modal models that take text and/or images interchangeably as inputs and reproduce them as outputs are being used by millions of people around the world and are already having a significant impact on contemporary visual culture.

In philosophy, there is a term to describe this type of inter-semiotic correspondence between words and images: *Ekphrasis* —from the Greek *ek* meaning “out” and *phrasis* meaning “speak.” Current generative systems allow new forms of correspondence and permutation between visual and linguistic registers through calculation, I call this: **computational ekphrasis**, a process by which language can be frozen to be seen all at once, as if it was an image, and images can be sequentially unrolled as if they were read and written.

In this presentation I elaborate on this notion of computational ekphrasis. Through a series of examples, including from my own practice in the computational modelling of moving images, I explore ways to mobilise aesthetic frameworks to think critically about programming and about generative modes of cultural production, before and after generative AI.

Ghost in The Paper: Player Reflex Testing with Computational Paper Prototypes

Tom Beckmann

Hasso Plattner Institute
University of Potsdam
tom.beckmann@*

Eva Krebs

Hasso Plattner Institute
University of Potsdam
eva.krebs@*

Leonard Geier

Hasso Plattner Institute
University of Potsdam
leonard.geier@*

Lukas Böhme

Hasso Plattner Institute
University of Potsdam
lukas.boehme@*

Stefan Ramson

Hasso Plattner Institute
University of Potsdam
stefan.ramson@*

Robert Hirschfeld

Hasso Plattner Institute
University of Potsdam
robert.hirschfeld@*

Abstract

Paper prototyping is an effective strategy for digital game designers to explore a design space. However, a large area of the design space is out of reach for paper: prototypes that need to react to player reflexes are challenging to realize. Many game genres use player reflexes as an integral part of their concept.

Through a set of examples, we analyze factors that contribute to the effectiveness and spontaneous spirit of paper prototyping. We then propose digital tool support designed to broaden the scope of paper prototyping, by introducing computation to paper arrangement, while maintaining its effectiveness and spontaneous spirit. We present and discuss the design of an explorative study to evaluate the concept.

1. Introduction

Game designers are faced with the challenge of obtaining feedback on their concepts as quickly as possible (Schell, 2019). As they typically work to create the elusive quality of "fun", it is difficult for them to gain insights through any other means than observing players of their game or playing it themselves.

Consequently, game designers frequently create prototypes that reveal an insight about their concept with as little time invested as possible. To this end, paper prototyping, or more generally low-fidelity prototyping, is a popular method that allows designers to quickly materialize a playable concept, even improvising more elements or rules on the fly. Contrarily, in most digital game prototyping methods a designer first needs to laboriously formalize the rules of the game in a way that a computer can execute them. While a designer may prepare a set of parameters to tweak quickly in their digital game, free improvisation is turned nigh impossible, even in settings with fast edit-compile-run cycles like live programming systems.

However, there are certain questions about a concept that are difficult to answer through a paper prototype. Games appeal through a variety of factors, such as engaging puzzles, skill challenges, or social communities. Concepts that seek to exercise the players' reflexes will likely benefit from a digital prototype, as a computer can respond to a player's actions within milliseconds, whereas an interpretation of another human directing a paper prototype will likely take seconds and be subject to significant imprecision.

In this paper, we seek to find a way to bring the benefits of paper prototyping to concepts that want to test player reflexes. To this end, we constrained a digital drawing tool to closely resemble paper prototyping. We then added the possibility for a designer to express rules within that tool that are automatically simulated by the computer. With this setup, we present insights from an exploratory user study pilot that seeks to answer two research questions:

- How does our digital counterpart compare to analog paper prototyping in terms of the designer's and the player's experience?

*hpi.uni-potsdam.de

- How well does our digital paper prototyping tool allow for prototyping game concepts that rely on player reflexes?

In the following, we will present insights from several paper prototyping sessions we observed and drew conclusions from for our digital tool (Section 2). We will then briefly describe how we mapped those insights to a digital tool (Section 3). Next, we describe our setup and insights gained from the pilot user study (Section 4) and discuss those (Section 5), before considering related work (Section 6) and concluding the paper.

2. Paper Prototyping

To guide the design of our digital tool for paper prototyping, we first observed two colleagues while they collaboratively designed a paper prototype. In the session, the designers conceptualized and started drafting a single-player card game where the player draws cards from a stack and maintains a hitpoint and defense counter. We interrupted their prototyping session after about one hour. We took notes focused on the activities and actions they performed during prototyping. While activities describe the operation on a high-level, actions describe how the designers physically perform activities.

2.1. Activities

During the prototyping process, multiple activities freely interleave. Designers may spontaneously interrupt one activity, perform another one, and return within seconds. Based on the actions we observed designers perform, we derive three activities: discussion, asset creation, and playtesting.

Discussion At the very start, whenever the next step appeared unclear, or when a new insight surfaced, the designers started brainstorming and discussing ideas. These could involve rules for the game, the look and feel of game elements, or balancing decisions. Often, the discussion served to align ideas or conclusions between the designers.

Asset Creation Here, the designers created pieces for their playable prototype. In the observed instance, this involved paper cut-outs, blank cards, as well as tokens from a board game that the designers found in the room.

Playtesting Once enough assets were available, the designers began playtesting the prototype. Notably, the designers would frequently interrupt playtesting and return to discussion or asset creation. This occurred when a rule appeared to be missing or ambiguous but also when a designer believed to have identified potential for an even better rule. Often, one designer would be playing while the other was observing or creating new assets to quickly inject into the playing session.

2.2. Operationalization of Activities

During the three activities, the designers performed a variety of actions.

During discussions, the designers made use of a whiteboard. They wrote text on it, drew shapes, circled drawn elements, or connected them with arrows. The relative placement of text and shapes was used to signify relationships.

The designers verbally added context to ad-hoc syntax, such as dotted lines to signify an implicit effect between two elements, or arrows to demonstrate movement. They proposed rules verbally, named challenges, and frequently referenced mechanics from other games as a shortcut for explanations. When discussions interrupted playtesting, the designers pointed at paper elements, gathered relevant elements for a better overview, or asked for opinions from their colleague.

The actions that occur during asset creation and playtesting are of special importance to our goal of creating a digital paper prototyping tool. From the actions the designers performed, we derive three categories: **Arranging**, **Describing**, and **Editing**.

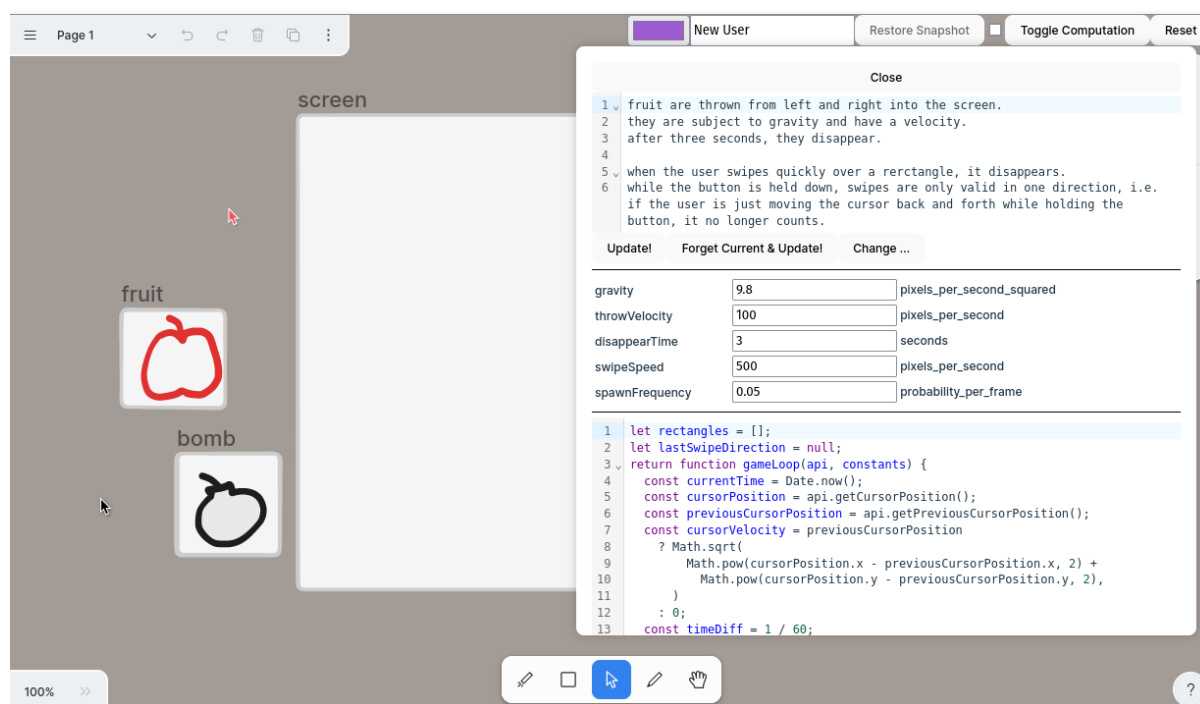


Figure 1 – Screenshot of our digital prototyping tool. Three pieces of paper are on the canvas that the user has named. The computations panel is currently open, showing first the users' instructions, then extracted constants for the generated code, and finally, the code itself in case it needs to be debugged. The second user's cursor is shown in red.

To create game elements, they took paper or cards and drew on them. While drawing, they sometimes also tried to exactly copy previously drawn shapes to another piece of paper. When the designers deemed a change necessary, they simply overwrote a previous shape on the paper, for example increasing a number.

Designers were rotating, stacking, and placing tokens and cards. They shuffled stacks of cards or drew cards from a stack.

Once game elements had been created, designers pointed at elements and named them verbally, for example proclaiming that this is the "draw pile" or "a token worth 5 points". Occasionally, they would also refer by name or point at an element and explain an associated rule or the meaning of an icon they drew. Throughout asset creation, designers also stated rules or revised previously stated ones.

During playtesting, the same Arranging actions came into play as the designers rearranged elements to signify changes to the game state.

3. Digital "Paper" Prototyping

The list in Subsection 2.2 provides us with an initial vocabulary of actions that a digital paper prototyping tool needs to support. Fortunately, most of these actions are already supported by digital whiteboards or drawing tools. We chose as a basis tldraw*, an open-source, web-based canvas component. A screenshot of our tool is shown in Figure 1.

In particular, our goal is to create an experience in the digital tool that is as close as possible to analog paper with a single change: designers can make elements move automatically based on a set of rules. As a consequence, we remove functionality that is commonly available in digital tools, such as undo. While there are still significant differences, notably the use of a digital interface as opposed to tangible paper, we hope to thus constrain the effects of other conveniences that the digital medium brings to a minimum, to focus on the single, desired intervention of automatic computation.

*<https://github.com/tldraw/tldraw>

3.1. Adaptations to tldraw

To make tldraw suitable for our needs, we included a remote collaboration feature, such that two users could work and play on the same canvas using two computers.

In terms of interactions, we removed all built-in tools besides those that we derived from our observations on paper prototyping. To ensure that the desired interactions feel natural, we added a pen/touch hybrid tool:

- By dragging the pen across the canvas, the user **creates a piece of paper** of variable size.
- By dragging the pen across a piece of paper, the user **draws** on it.
- By dragging a piece of paper with their finger, the user **moves** that piece. This way, they can also **stack** pieces.
- By dragging the canvas with their finger, the user pans the viewport.

We left the default tools for creating rectangles (pieces of paper), drawing, and selecting available, in case users wanted to use a mouse instead of a pen.

Finally, we added two special tools: one tool for **pointing at** an element and to **name** or **explain** it using a text, and a second tool that ignores all inputs but still provides mouse inputs to the computation system, as the interactions during play may overlap with those for editing.

3.2. The Ghost in The Paper

The reason for recreating paper prototyping in the digital space, as described in Section 1, is to allow elements to move automatically, reacting immediately to player input. We ideally want designers to be spontaneous and have quick feedback loops akin to prototyping with paper – ruling out the option of writing code manually. For this purpose, we added a text field that allows users to **state rules** in natural language, as seen on the right of Figure 1.

As guide for designers, we envision a scenario where they describe to a third, absent designer how elements should behave. That third designer would take the role of computer and execute these steps. As the third designer is not present during creation of the prototype, the rules need to be unambiguous and complete. Notably, not all rules of the prototype need to be spelled out, only those that require the computer to be involved.

Given this semi-formalized description of rules, we had a generative artificial intelligence produce a formal, executable version of the rules in JavaScript. Once JavaScript code has been generated, the user can toggle a play mode, which toggles continuous execution of the generated code for every frame in a game loop. When the play mode is activated, we persist a snapshot of the canvas state and allow designers to revert to that state before any modifications by the computer took place. To avoid any issues from networked play, we constrain the play mode to a single client. The other clients still see the scene evolving but the computational rules, such as checks for mouse position, are not run on their instances.

LLM Configuration We used the *gpt-4o* model in our experiments, with its JSON mode enabled and a temperature of 1. The prompt consisted of multiple system messages:

- Informing the AI that it is meant to create rules for a 2D game.
- The layout of the designers' scene, including the **names** and **explanations** that designers have attached to elements.
- The API available to it, auto-generated from a TypeScript file, resembling the **Arranging** actions of our vocabulary.

- The boilerplate for the gameLoop, shown in Listing 1.
- A request to extract and return any constants.
- A set of instructions aimed at preventing misuse of the API we have observed.
- Optionally: the previous code, instructions, and constants the LLM had generated, with the request to match it closely if possible.
- As a user message, the rules they specified.
- The layout of the JSON object to be returned: `{constants: {name: string, default: number, unit: string}[], code: string, instructions: string}`.

We display two buttons for updating the rules: one that includes and one that excludes the previously generated code. In practice including or excluding the previous code did not appear to make a difference, as the newly generated code often deviated significantly from the previously generated one, even for small change requests.

```
// any state you may want to store
// let a = ...
return function gameLoop(api, constants) {
  ...
}
```

Listing 1 – Boilerplate for the Game Loop

3.3. Exemplary Workflow

As an example, let us consider a simple jump-and-run to which we want to introduce a new mechanic: a ghost should follow the player while they are not looking at it. The scenario is drawn from an early experiment we did ourselves.

We begin by drawing pieces of paper for platforms onto the canvas. We then create two pieces of paper on which we draw a player icon and a ghost icon.

One designer is moving the player, the other is moving the ghost, each on their respective device. We observed that it is difficult to realize when the ghost should stop moving, i.e. the player is looking in its direction. An attempt to have the player call out the direction they are looking to turned out to be too unreliable. We also do not want to fully automate the ghost's movement, as we are still unsure about its ideal movement pattern and want to maintain the option to improvise its movement.

Instead, we add a computation rule that displays a piece of paper either on the left or on the right of the screen, depending on the direction the player last moved in. This suffices as a guide for the designer moving the ghost to react quickly.

4. Pilot User Study

We design a user study that answers (1) how the experience of using our digital paper prototyping tool compares to analog paper prototyping, and (2) how well our tool allows prototyping concepts that rely on player reflexes. Here, we are presenting the design of and discussing insights from a pilot study.

4.1. Setup

We recruit participants who state that they have prior experience with paper prototyping. To answer the two questions, we give participants two prototyping prompts: one that we believe would work well for analog paper prototyping and one that we believe would be challenging to prototype. Participants work in teams of two. They work on both prototype prompts once with analog paper and once with our digital tool, for a total of four prototyping sessions.



Figure 2 – Materials we provided during the prototyping session for analog prototyping.

For the analog prototypes, we provided a set of materials as shown in figure Figure 2. In addition, we allowed participants to ask for any additional materials, that the instructions would then procure or state that those cannot be used.

Before the first use of our digital tool, participants receive a brief introduction and can try out the tool. We also explain the framing of specifying rules for computations in the setting of recording instructions for a third, currently absent, designer who will later be performing the instructions. In addition, we prime them to consider involvement of computations a "last resort", when it is clear that manual execution will falsify the prototyping insights.

4.2. Prototyping Prompts

For the prompt that should work well with paper, we ask participants to extend the rules of the puzzle video game *Baba Is You*. In *Baba Is You*, the player moves the eponymous Baba on a top-down, rectangular grid. In the grid cells are objects or words. Pushing the words with Baba to form sentences changes the game rules. For instance, building the sentence "Wall is Push" means that Baba can now push wall objects around the grid.

We ask participants to add a new "invert" word to the game that should invert whatever words it is combined with. Participants should investigate two questions through their prototype:

- With what other words of the base game does the "invert" word work well?
- Does the "invert" word allow for interesting puzzles?

For the second prompt, we ask participants to modify the 2D game *Fruit Ninja* that is typically played on a touchscreen. In *Fruit Ninja*, fruits are thrown across the screen and the player tries to slice through these to gain points by swiping across them. Among the fruits are bombs that the player has to avoid.

We ask participants to investigate how the game is best played with a mouse through the question:

- How to best interact with *Fruit Ninja* using a mouse?

4.3. Method

The participants are given the prototyping prompt with a time limit of 30 minutes. While the participants are working on the prompt, we record their interactions and conversations for later analysis. Once the time limit has elapsed, we perform a semi-structured interview with the participants. In particular, we ask the following questions to facilitate answering our research questions concerning the comparison between the mediums and the effectiveness of computations for testing reflexes:

1. What was the most frustrating part about working with the given materials?
2. What worked best in with the given materials?
3. For the digital tool only: What actions you know from analog paper did you want to perform that were not possible?

During early experimentation, we realized that the quality of LLM-generated code was unreliable. Consequently, we opted for a "hybrid" Wizard-of-Oz (Green & Wei-Haas, 1985) mode: participants first generate code, execute it once, and if an error occurs or the behavior is not obvious within the first couple of seconds, we interrupt the session and an instructor repairs the code. We discuss the effect of this approach in Subsection 5.2.

4.4. Results from Pilot Run

As a first pilot for the described study, we executed the setup with two of the co-authors as participants. We briefly describe their approach to answering the prototyping prompt, challenges they encountered, and their answers to the interview.

Baba Is You: Analog The participants first brainstormed interpretations of the "invert" word in combination with other words from the original game. They used a single sheet of paper to document ideas and refer to them later again. To create assets, they made use of a set of multi-colored chips and placed them on a grid that they drew on a sheet of paper, see Figure 3. They drew icons or words on the chips to differentiate them.

During play, the participants quickly realized obvious solutions to their puzzles, rolled back a couple of steps, changed an aspect of the physical level or verbal rules, and tried again. The participants struggled with a consistent interpretation of the "invert" rule throughout the session, as most often they fell back on an interpretation as "not", for which they did not find interesting puzzles.

In the interview, the participants indicated that they kept struggling with the permanence of decisions and that they had to remind themselves that they can always overwrite icons. They found it challenging to quickly move arrangements of objects in their entirety, for instance, a prepared level, without it being damaged in the process. At times, space on the grid they had prepared appeared tight; however, they also noted that they appreciated the constraint to keep levels simple. In parts, it was difficult for them to verify if they were complying with the verbally agreed-upon rules, especially when they were quickly iterating different arrangements of levels.

On the other hand, they also appreciated that the material was not giving any hard constraints – if it was obvious that a sequence of moves was valid, they could move Baba directly to a destination. They expressed that the chips were useful for this prototyping prompt, as they were just heavy enough not to fly around when interacting with the grid and adding new objects by labeling the chips was so quick that they would often do so mid-play.

Baba Is You: Digital After the interview, the participants continued their prototyping session in the digital space. They again started by taking notes, this time on a digital piece of "paper" in our tool. Simultaneously, the other person already started quickly recreating the same tokens by drawing square rectangles and drawing or writing on them.

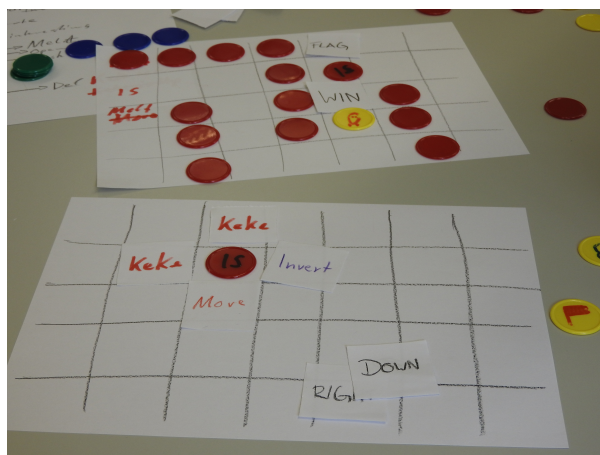


Figure 3 – One analog prototype created for *Baba is You*.

The mode of asset creation and play were largely the same: participants tried out ideas they deemed promising, adapted aspects mid-play, and discussed insights. When one participant was presenting a new idea, the other participant often tended to look at the presenter's laptop as opposed to their own, even though the partner's cursor positions were shown on both devices.

In the interview, the participants indicated that the pen/touch-mode initially led to one or two wrong inputs but that they quickly adapted. Reliably pointing at small elements using the finger, however, was sometimes challenging. Both participants indicated that there was slightly more friction to creating elements: written text ended up larger and so space was running out more quickly or giving precise inputs required more concentration. To make matters worse, one of the laptops used during the experiment was running a screen recording software, which caused the framerate on that device to be very low. The participants noted two fundamental issues: first, the limited viewports of the two laptops compared to the physical table often made it difficult to see what the other person was working on or even what they were looking at when talking about something specific, making communication more challenging. In particular, shifting the viewport required interactions with the hand on the screen, whereas with analog paper shifting the head or the focal point of the eyes was enough. Second, the participants described an "uncanny valley" of interactions, where the limited set of interactions we constrained our tool to caused some frustration, as they are used to operations like resizing or effortless duplication being available in digital tools. To them, the medium did not evoke the impression of paper enough to remove these expectations.

On the flip side, they praised the effortless creation of paper sheets in the digital tool, as well as the quick selection of the pen color, both of which typically involve reaching across the table or using scissors. The pen/touch-mode also worked very well, after the initial hurdle and aligned well with their expectations from the physical world. In this way, arranging objects, which was the bulk of the session, worked well, too.

In terms of actions they were missing from analog paper, they listed moving multiple things – which was not supported using the pen/touch-mode. They also noted that resizing does work to an extent in the analog world, simply by cutting off parts of a piece of paper. Lastly, rotation was also not well supported in our tool.

Fruit Ninja: Analog The participants again started by sketching ideas on a piece of paper, this time for four different interactions for collecting fruit. They quickly identified as their main challenge to convey a similar feeling to the indirection of using a mouse: an attempt to quickly draw a line between fruit previously drawn on a sheet was deemed too dissimilar; another attempt to tap fruit that the other participant was moving around on the table led to arms getting in each other's way; a third attempt where

another person was asked to close their eyes, sit down at the table, and move their finger to collect fruit while another player was "drawing" directions on their back, to simulate the indirection of the mouse, resulted in too high latency and inaccuracy. In a fourth attempt, the participants placed small sheets of paper on the table and two players competed against one another to collect the most, which ended up not feeling comparable to Fruit Ninja. Finally, the closest approximation was putting a table at an incline and throwing chips at it, such that they sled back down while the player had to slice through them; this created a similar feeling but was a better simulation of a touch screen than a mouse.

Consequently, the participants were dissatisfied with their progress, stating that they learned about the prototyping medium but nothing about the prototyping prompt. The major challenges concerned creating continuous, predictable movement, moving enough elements at once to present a challenge to the player, and to move elements without getting into the player's way. However, they also noted that they managed to prototype five distinct ways to approximate the mouse-based interface in the half-hour time span. The chips worked well again here, as they had the right size and weight to be thrown around.

Fruit Ninja: Digital In the final condition, the participants took two digital sheets of paper to sketch their ideas and created a fruit and a bomb asset. They then added computational rules that started out very simple, e.g., "move a shape from left to right on the screen" and gradually increased in complexity, e.g., "slice a rectangle at high speed to remove it". After every generation, the participants made heavy use of the constants to tweak intervals and velocities of elements to find the right feeling. Throughout, the LLM generation failed, meaning that the session had to be interrupted five times for the instructor to fix the code, before the session could resume, which took between 2 and 5 minutes each time. In the end, they prototyped a tap-to-remove interaction and had started tweaking the rules of how a slice-to-remove interaction should work.

In the interview, participants noted that they had wished for copying existing paper or coloring paper. They also noted that the interruptions due to generation failures were jarring and it was difficult for them to pick up where they had left. As the current version of our tool allows only one participant to interact with computations, they had to effectively share one small laptop and one set of controls, even though they saw great opportunity in tweaking constants while the other participant was playing. One aspect that they were missing was the possibility to demonstrate the velocity or the arc of a movement through motion, instead having to express it in text or tweak constants to achieve it.

The participants praised the automatic movement and the quick means to achieve it, allowing them to now have arbitrary numbers of elements moving, without interference from another set of limbs, and with precise and consistent control over movement. As a changed constant instantly made its effect visible, they felt in control when it came to tweaking the behavior.

5. Discussion and Future Work

Here, we briefly discuss insights from the pilot on our study design and on our tool design.

5.1. Study Design

Overall, the prototyping prompts elicited the challenges we had anticipated well: the Baba Is You prompt was quickly realized in both the analog and digital versions and the Fruit Ninja prompt posed a major challenge in analog but was realizable in digital.

As our participants noted the challenge of making sense of the "invert" rule for Baba Is You we are considering instead moving to simply asking participants to design an interesting Baba Is You level with a predetermined set of words. As this was the main aspect that exercised the prototyping medium and that worked well for the participants, we would reduce frustration while keeping relevant insights.

It is still unclear how to best address interruptions when LLM generation fails in the digital medium. One method may be to remove the LLM and instead have a set of higher-level functions prepared that the instructor can invoke. If the set of functions is well adapted to the prompt, it may allow faster cycles.

5.2. Digital Paper

While the digital paper lacked tactile properties and thus led to occasional erroneous inputs, we believe that our approximation of paper prototyping was already mapped well for the prompts we tried. As participants showed frustration that they did not have the functionality available they expected from digital tools, we might consider allowing their use. In particular, they requested scaling and duplicating shapes, which we had removed to better approximate what is possible in the real world. It remains to be seen if the addition of the tools might also change the types of prototypes designers are building in the tool. Whereas they might be inclined to go for a minimal set of elements without a duplicate function, duplication might, perhaps unnecessarily, push them to consider more complex arrangements.

Concerning the computation element, participants were positive in terms of its intention but its execution was challenging as part of the prototyping session. Interestingly, the participants slowed their own progress significantly and introduced further interruptions by incrementally adding rules to supposedly test the system's limits. In practice, the generated code varied so wildly in quality that a previously stable state could not be reliably built upon or even reproduced.

Errors included incorrect syntax (missing delimiters), incorrect use of language constructs (assignment to const variables), hallucinated API methods, writing to internal properties that had been explicitly forbidden by the prompt, or logical errors (modifying a collection while iterating over it, not storing the cursor position from the previous frame to calculate movement). The instructor addressed the issues either by adding further hidden constraints to the prompt and regenerating the code, fixing them manually, and sometimes even asking the participants not to generate code but implementing a change manually when the change was small. In addition, the use of the passed in constants tended to change between code versions, so, even though we kept values and units the same between generated code versions, the behavior changed drastically and participants had to tweak the behavior anew.

An interesting avenue for future work is to explore opting for a full Wizard-of-Oz setup. To realize this, we may consider preparing a set of high-level routines that will likely be of use during the prototyping session. In addition, AI-assisted coding that merely autocompletes as opposed to generating an entire program would give the instructor more control. It remains to be seen if changes can be realized quickly enough this way to keep the participants' attention. Alternatively, prior work has demonstrated the possibility of crowd-sourcing logic for a sketch in real-time (Lasecki et al., 2015). Other work points to the possibility of pre-processing the user sketches to extract structure, which could support the translation process (Li, Cao, Everitt, Dixon, & Landay, 2010).

6. Related Work

Low-fidelity prototypes such as paper prototyping are an essential part of the design process and an essential aid in discovering good designs (Nielsen, 1995). In game design, where according to "the rule of the loop" we want as many iterations as possible to improve our game, paper prototypes are especially relevant for their fast iterations (Schell, 2019). Aside from traditional paper prototypes, other low-fidelity prototypes with a similar essence are also used for game design. This includes physical prototypes, sometimes called "bodystorming", where humans and their interactions are the main focus (Macklin & Sharp, 2016).

Other approaches have combined aspects, or concepts of, paper prototyping with digital or mixed-media tools. For instance, there are several tools such as Miro boards (Chan, Ho, & Tom, 2023) that allow multiple people to collaborate digitally in a way that is similar to using paper and post-it notes. One previous project prepared paper prototypes and manually converted them to digital prototypes in Power Point (Uceta, Dixon, & Resnick, 1998). Even though the general style of the prototype stayed the same and only links between slides were added, the feedback was more concise and focused compared to using traditional paper prototypes because it was easier for testers to suspend their disbelief and not get distracted by the prototype medium.

As our pilot test runs already revealed problems with the limited viewport of a 2D screen, a possibility

would be to consider moving to virtual or augmented reality. Previous work used AR to prototype physical or spatial interfaces, e.g. the appearance of elements in cars (Porter, Marner, Smith, Zucco, & Thomas, 2010). There is also a multitude of modeling or drawing tools for virtual and augmented reality, some of which are also targeted at prototyping in particular or rapidly building complex systems in general (Gasques, Johnson, Sharkey, & Weibel, 2019; Kang et al., 2019). The inverse also exists: There are applications that can convert a basis made out of paper to simple VR applications for prototyping purposes (Nebeling & Madier, 2019). In fields like VR where the viewport of the playtester can be controlled, there are also approaches where the user interacts with the virtual environment while a human support team instead of a purely automated environment moves physical objects accordingly (Cheng et al., 2015).

A number of tools and systems facilitate programming interactive experiences like games. Pronto (Krebs, Beckmann, Geier, Ramson, & Hirschfeld, n.d.) and Unreal Engine blueprints (Valcasara, 2015) are both visual approaches to formulating game logic, both of which may speed up iteration time depending on the scenario. Block-based editors such as Scratch (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010) or Snap (Harvey & Mönig, 2015) provide an environment focused on creating interactive visual experiences that react instantly to changes in programming logic.

Finally, live programming methods are designed to bring faster feedback to programmers (Rein, Ramson, Lincke, Hirschfeld, & Pape, 2019). Some game engines, such as the Godot game engine, make it possible for some changes to be applied live to the running game. In live coding (Blackwell & Collins, 2005), where code is used as a performance art, it is even specifically required to be able to improvise changes over the course of seconds.

7. Conclusion

In this paper, we extracted what may approximate an essence of paper prototyping and translated it to a digital equivalent. In the digital realm, we augmented this baseline with the ability to specify computations to allow designers to create prototypes that react immediately to player reflexes.

In a pilot user study, we found that the digital tool was working similarly to analog paper for the prototyping prompts we tried, but in addition enabled prototyping concepts involving player reflexes. Our attempt to maintain the spontaneous flow of a prototyping session by letting designers formulate natural language and having an LLM generate code turned out to require further work, as the quality of code was highly unreliable.

8. References

- 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, brighton, uk, june 29 - july 1, 2005* (p. 11). Psychology of Programming Interest Group. Retrieved from <https://ppig.org/papers/2005-ppig-17th-blackwell/>
- Chan, T. A. C. H., Ho, J. M.-B., & Tom, M. (2023, March). Miro: Promoting collaboration through online whiteboard interaction. *RELC Journal*, 003368822311650. Retrieved from <http://dx.doi.org/10.1177/00336882231165061> doi: 10.1177/00336882231165061
- Cheng, L.-P., Roumen, T., Rantzsch, H., Köhler, S., Schmidt, P., Kovacs, R., ... Baudisch, P. (2015). Turkdeck: Physical virtual reality based on people. In *Proceedings of the 28th annual acm symposium on user interface software & technology* (p. 417–426). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2807442.2807463> doi: 10.1145/2807442.2807463
- Gasques, D., Johnson, J. G., Sharkey, T., & Weibel, N. (2019). What you sketch is what you get: Quick and easy augmented reality prototyping with pintar. In *Extended abstracts of the 2019 chi conference on human factors in computing systems* (p. 1–6). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3290607.3312847> doi: 10.1145/3290607.3312847

- Green, P., & Wei-Haas, L. (1985, October). The rapid development of user interfaces: Experience with the wizard of oz method. *Proceedings of the Human Factors Society Annual Meeting*, 29(5), 470–474. Retrieved from <http://dx.doi.org/10.1177/154193128502900515> doi: 10.1177/154193128502900515
- 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
- Kang, S., Norooz, L., Bonsignore, E., Byrne, V., Clegg, T., & Froehlich, J. E. (2019). Prototpar: Prototyping and simulating complex systems with paper craft and augmented reality. In *Proceedings of the 18th acm international conference on interaction design and children* (p. 253–266). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3311927.3323135> doi: 10.1145/3311927.3323135
- Krebs, E., Beckmann, T., Geier, L., Ramson, S., & Hirschfeld, R. (n.d.). Pronto: Prototyping a prototyping tool for game mechanic prototyping. , *34th Annual Workshop*, 157–168. Retrieved from <https://www.ppig.org/files/2023-PPIG-34th--proceedings.pdf>
- Lasecki, W. S., Kim, J., Rafter, N., Sen, O., Bigham, J. P., & Bernstein, M. S. (2015). Apparition: Crowdsourced user interfaces that come to life as you sketch them. In *Proceedings of the 33rd annual acm conference on human factors in computing systems* (p. 1925–1934). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2702123.2702565> doi: 10.1145/2702123.2702565
- Li, Y., Cao, X., Everitt, K., Dixon, M., & Landay, J. A. (2010). Framewire: a tool for automatically extracting interaction logic from paper prototyping tests. In *Proceedings of the sigchi conference on human factors in computing systems* (p. 503–512). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/1753326.1753401> doi: 10.1145/1753326.1753401
- Macklin, C., & Sharp, J. (2016). *Games, design and play*. Boston, MA: Addison-Wesley Educational.
- Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010, nov). The scratch programming language and environment. *ACM Trans. Comput. Educ.*, 10(4). Retrieved from <https://doi.org/10.1145/1868358.1868363> doi: 10.1145/1868358.1868363
- Nebeling, M., & Madier, K. (2019). 360proto: Making interactive virtual reality & augmented reality prototypes from paper. In *Proceedings of the 2019 chi conference on human factors in computing systems* (p. 1–13). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3290605.3300826> doi: 10.1145/3290605.3300826
- Nielsen, J. (1995). Using paper prototypes in home-page design. *IEEE Software*, 12(4), 88-89. doi: 10.1109/52.391840
- Porter, S. R., Marnier, M. R., Smith, R. T., Zucco, J. E., & Thomas, B. H. (2010). Validating spatial augmented reality for interactive rapid prototyping. In *2010 ieee international symposium on mixed and augmented reality* (p. 265-266). doi: 10.1109/ISMAR.2010.5643599
- Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., & Pape, T. (2019). Exploratory and live, programming and coding - A literature study comparing perspectives on liveness. *Art Sci. Eng. Program.*, 3(1), 1. Retrieved from <https://doi.org/10.22152/programming-journal.org/2019/3/1> doi: 10.22152/PROGRAMMING-JOURNAL.ORG/2019/3/1
- Schell, J. (2019). *The art of game design* (3rd ed.). London, England: CRC Press.
- Uceta, F. A., Dixon, M. A., & Resnick, M. L. (1998, October). Adding interactivity to paper prototypes. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 42(5), 506–510. Retrieved from <http://dx.doi.org/10.1177/154193129804200513> doi: 10.1177/154193129804200513
- Valcasara, N. (2015). *Unreal engine game development blueprints*. Packt Publishing Ltd.

Designing A Multi-modal IDE with Developers: An Exploratory Study on Next-generation Programming Tool Assistance

Peng Kuang
Lund University
peng.kuang@cs.lth.se

Emma Söderberg
Lund University
emma.soderberg@cs.lth.se

Martin Höst
Malmö University
martin.host@mau.se

Abstract

Researchers have envisioned and pioneered data-driven programming assistance for developers based on their interaction with the tools via multiple sensors such as eye trackers, microphones, and AI. However, these new sensors gather sensitive data from programmers, to what extent users can accept them and in what form they may work well are largely unclear. Meanwhile, developer tools such as static analyzers have been criticized for poor usability and not involving end users enough during their development. To make data-driven programming assistance design work, toolmakers need to partner with programmers.

In this paper, we adopt a design process under the guidance of Participatory Design (PD) which aims to empower end users. Our design pipeline builds on a survey of 68 professional developers. The next stage is a design workshop with five participants sketching out ideas for alleviating the pain points reported from the survey. Based on these inputs, we then developed two types of tentative design representations consecutively – three conceptual designs and one low-fidelity prototype¹. Lastly, we tested the interactive digital prototype with five experienced programmers.

From the user test, we learned that developers have more interest in gaze-driven assistance than voice-based assistance for reading and understanding code in an integrated development environment. Further, we report our hands-on experience with involving developers from the beginning to the end of this design process. This informs future work on using PD to study the development of developer tools.

1. Introduction

Data-driven techniques are making their way into developer tools. In the past decade, eye tracking technology has matured significantly and eye trackers are now cheaper and more available. Laptop manufacturers have started to embed eye trackers into their high-performance models for gamers (Tobii, 2023). Apple has introduced a three-dimensional interface with hands, eyes, and voice, alongside a bundle of productivity tools on its VisionOS platform (Apple, 2023), which makes reading natural language texts on a virtual screen foreseeable. We are also seeing early explorations of gaze-driven (Saranpää et al., 2023; Cheng, Wang, Shen, Chen, & Dey, 2022; Santos, 2021; Shakil, Lutteroth, & Weber, 2019; Radevski, Hata, & Matsumoto, 2016; Glücker, Raab, Echtler, & Wolff, 2014) and voice-based (Paudyal, Creed, Frutos-Pascual, & Williams, 2020; Talon, 2024) assistance in development tools. Meanwhile, we are in the middle of a shift toward AI programming (Liang, Yang, & Myers, 2024) or Autonomous Software Engineering (Zhang, Ruan, Fan, & Roychoudhury, 2024; Yang et al., 2024; AI, 2024), where Large Language Models (LLMs) trained on huge amounts of data are assisting developers with development. These data-driven technologies, gathering data from sensors or code bases, have the potential to enrich users' experience and improve their productivity.

We see an opportunity to incorporate these technologies into building the next generation data-driven developer tools, as some researchers have envisioned (Kuang, Söderberg, Niehorster, & Höst, 2023; McCabe, Söderberg, Church, & Kuang, 2022) and pioneered (Saranpää et al., 2023; Cheng et al., 2022; Hijazi et al., 2021). However, incorporating these new sensors implies gathering significantly more and possibly sensitive data from programmers. Previous work demonstrates that programmers may have concerns with such data collection (Kuang, Söderberg, & Höst, 2024). Further, how to encapsulate these fancy new technologies into a good design that can be accepted by end users can be a challenge.

¹A replication package is available at: <https://doi.org/10.5281/zenodo.13853909>

Taking AI as an example, it did not gain wide user acceptance until OpenAI captured it into the design of ChatGPT. Exploratory gaze-based developer tools have primarily used eye-tracking for code navigation on the interaction level and information generation about developers' collaborative work such as code review. As for voices, it has so far mainly been explored for accessibility improvement, for instance, as an alternative input channel for developers with visual impairment (Paudyal et al., 2020). It seems the exploration in this field is still in its infancy. There are possibly other user scenarios and developer cohorts that may benefit from the integration of these emerging modalities as well. To make it work, we believe developer tools need to be designed carefully, and the design process needs to partner with programmers.

How do we involve developers in the design process? As an example, the community around Open Source Software (OSS) is driven by programmers and there are many examples of successful projects (e.g., Linux, Apache, and Mozilla Firefox). However, many OSS projects have been criticized for poor usability or not being end-user friendly (Hellman, Cheng, & Guo, 2021). It is believed that developers tend to adopt a code-centric mindset and that this may create a gap between a human-centric designer role and a developer role, along with the different training and skills associated with these two roles (Maudet, Leiva, Beaudouin-Lafon, & Mackay, 2017). Even though developers who work with developer tools seem to embody the user, the designer, and the developer, there are proven records that the tools they make can still be poor to use because of this gap. To mitigate such a risk, we believe it is necessary to involve users other than the toolmaker developers themselves in the design process. We need a design process that reduces this gap.

Participatory Design (PD) emerged from Scandinavia to empower the workers to balance or counter the insights of management in the workplace (Bannon, Bardzell, & Bødker, 2018; Schuler & Namioka, 1993; Ehn & Sandberg, 1979). PD roots in a strong commitment to democracy (Bødker, Dindler, & Iversen, 2022), which can be translated into the paramount care for end users. It further pronounces that the designer needs to partner with users in the design process to discover possibilities or alternatives for a technology being imposed on them (Bødker et al., 2022). As an outcome of such a design process, "*mutual learning*" for both sides is curated. What distinguishes PD from other design methods is that it takes both a moral position and a pragmatic position (Carroll & Rosson, 2007) and that it emphasizes "*democratic empowerment*" but not just "*functional empowerment*" (Clement, 1996). We think PD may be a good fit for designing next generation data-driven programming tools with programmers.

In this paper, we explore participatory design through a series of design activities with professional developers to rapidly prototype and test out data-driven programming tools powered by AI and other assistive sensory technologies, e.g., eye tracking, that are novel to the current typical software development environments. This work is in line with what Kuang et al. (Kuang et al., 2023) have proposed as a method to study gaze-driven assistance. We focus on investigating the RQ: *What do developers want the next generation data-driven programming assistance to look like?* Through the design pipeline, we gradually capture developers' needs and wants with the next-generation programming tool assistance via several intermediate design representations and eventually actualize it into a tangible multi-modal Integrated Development Environment (IDE). The IDE contains four features: two gaze-driven, one voice-based, and one AI-enabled. We have evaluated this interactive artifact with experienced programmers and learned that there is greater interest from developers in gaze-driven assistance than voice-based assistance in the case of reading and understanding code in an IDE.

This exploratory study reports our hands-on experience on how to practice participatory design in partnership with developers for developing new types of programming tool assistance. Our findings, through the "*mutual learning*" between the designer and developers, provide insights into what modality may work and not work well with developers for designing the next-generation programming assistance. We hope that our reported experience, of end-to-end involvement of programmers throughout the PD process, will aid other toolmakers interested in exploring this direction.

2. Related Work

In this section, we give a brief overview of related work in multi-modal developer tools and the use of participatory design in the development of developer tools.

2.1. Multi-modal Developer Tools

A multi-modal interface usually supports the interaction means of gaze and voice, apart from the conventional input means of keystroke, mouse, and touch through one's hands (Benoit, Martin, Pelachaud, Schomaker, & Suhm, 2000). Owing to the naturalness that it resembles human-human communication and the expressiveness and adaptability that it provides (Oviatt & Cohen, 2000), multi-modal interface has attracted significant interest from HCI researchers in recent decades. Especially in the current era of AI, people are even more passionately envisioning and actualizing intelligent systems, also for developer tools.

Since reading code constitutes a significant part of developers' work, some researchers investigated the feasibility of utilizing gaze for several software development tasks that were undermined by code reading. Several studies have examined its suitability for code navigation (Santos, 2021; Shakil et al., 2019; Glücker et al., 2014), while several others experimented with it for code review (Saranpää et al., 2023; Cheng et al., 2022; Hijazi et al., 2021). The overall gaze behavior during software development has also been inspected (Clark & Sharif, 2017). The modality of voice has been less studied but has been piloted in the context of empowering developers with disabilities (e.g., visual (Paudyal et al., 2020) or hand impairment (Talon, 2024)).

The above-mentioned studies share a human-factor motivation and usually contain a component of a user study. However, they primarily focus on the introduction of implemented techniques, and the user studies are almost all done post-implementation for testing the usability of the tools. As software development is a high-cost activity, we think it is worth involving users from an early stage of the design process to mitigate the risk of having (if not completely avoiding) implementations that do not fit into users' needs. Further, the multi-modal features presented in our design also cater to user scenarios different from the ones investigated by the previous studies.

2.2. Participatory Design and Developer Tools

Participatory design is a well-known design method to software engineering researchers and gained significant interest throughout the 1990s to 2010s. It is usually brought up along with user-centered design when researchers discuss research questions pronouncing user involvement or end-user software engineering.

Common developer tools emerge from the industry, the open-source community, and the scientific community. These three sources are not mutually exclusive as both companies and scientists can be members of the OSS community. To provide a structure for reporting the use of PD in developing developer tools, we group the related works into these three categories.

In the industrial context, some researchers (B. Johnson, Song, Murphy-Hill, & Bowdidge, 2013) have adopted PD for eliciting non-verbal inputs from target users to improve the design of static analysis tools. It seems that PD is used at surface level in this case or a conventional user study alike.

In the area of OSS, we did not find a study in the case of developing a developer tool explicitly saying that they have used PD (this does not imply our search is exhaustive) although some well-known developer tools such as Apache Maven are OSS. However, there are literature reviews (e.g., (Hull, 2021)) and many case studies (e.g., (Hellman et al., 2021; Wubishet, Bygstad, & Tsiavos, 2013; Iivari, 2009; Gumm, Janneck, & Finck, 2006)) on user participation in other types of computer systems that can inform and perhaps extend to developer tools. These studies found that there are both technical and social barriers (e.g., technical capability and social credibility (Mockus, Fielding, & Herbsleb, 2002), documentation (Hull, 2021), subculture literacy (Iivari, 2009)) for developer users (who are usually the first batch of users of OSS) to participate in and contribute to OSS development.

In scientific programming, a case study (Letondal & Mackay, 2004) shows that because the resources

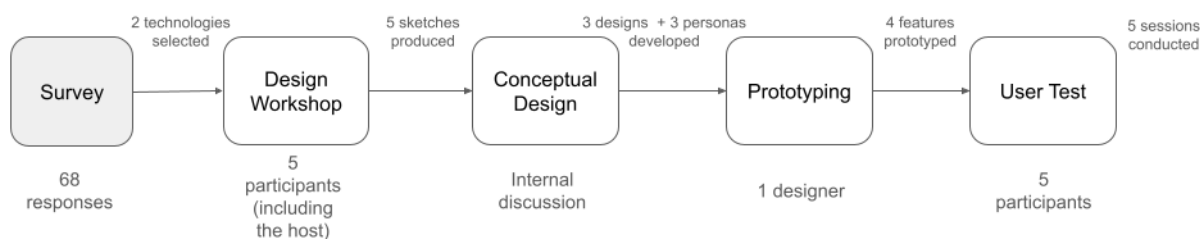


Figure 1 – Overview of the method. The greyed box means the component is not directly reported in this paper. It has been published separately but was done as a part of this study to inform the remaining components of the design pipeline.

for new hiring are limited, developers working with scientific teams employed PD as a design method to develop some intermediary tools to enable end-user scientists to perform some frequent but not-so-difficult programming tasks. However, such tools are not the typical developer tools used by professional developers for software development but are similar to business information systems.

To sum up, HCI methods (e.g., rapid prototyping, usability evaluation) are valued by researchers and have been adopted as a practice in the industry (Myers, Ko, LaToza, & Yoon, 2016) for developing programming tools. However, the explicit use of PD for such purposes seems to be rare, either in the industry, the OSS community, or the scientific programming community.

3. Method

The design process consists of a survey with professional developers, a design workshop, conceptual design, prototyping, and the user test (Figure 1). We conducted a survey targeting professional developers and received valid responses from 68 participants. In the workshop, we presented the pain points collected from the survey for participants to design solutions for. To catalyze brainstorming and sketching, we demonstrated a gaze-driven code review tool (Saranpää et al., 2023) as a data-driven programming assistance example. We then used the outputs curated from the previous modules to develop conceptual design ideas and create personas representing different types of prospective users. After that, we translated these design ideas into four features of a low-fidelity prototype. Lastly, we tested the interactive prototype with five prospective users who are experienced programmers.

3.1. Survey

The survey received responses from 68 professional developers located in 12 countries. In the survey, we asked developers about their experiences with current programming tool assistance and their perceptions of programming tool assistance in the future. The survey (Kuang et al., 2024) has been published separately and its preliminary results were leveraged in planning the design workshop.

3.2. Design Workshop

In the survey, there was a checkbox question for participants to indicate their interest in participating in the workshop. 12 participants ticked the checkbox and provided their email address. We reached out to these twelve potential participants who expressed their interest via email to poll their availability in the upcoming weeks. Four of these potential participants accepted the invitation. Because there was no single time slot that could accommodate all the participants, we chose the one that seemed to suit the majority. Since some of these participants were slow with the response, we actively recruited some other participants in person through our social circle to reduce the risk of no turn-out. Eventually, only one from the online participant pool managed to attend the workshop due to their challenging availability or large time difference. The rest were from the convenient recruitment in-person.

We organized a hybrid design workshop session, with one participant online and others onsite. Two participants were Ph.D. students in Sweden but in different cities. One was in the field of Computer

Science and was familiar with specialized developer tools such as program analysis. The other had a background in Electrical Engineering and worked with wearable devices and machine learning (ML). Two other participants were research assistants who have been working as developers for two years in the Department of Computer Science at Lund University. The host (a.k.a. the first author) also joined and was counted as the fifth participant since we adopted a co-design formality. All participants were given access to a digital collaborative tool called Box Canvas. The participants onsite were also provided with markers and white papers. Participants were encouraged to choose whichever way they felt most comfortable to output their design ideas.

The workshop followed the procedure of introduction, a demo of a screen-based eye-tracker, as well as a code review tool with integrated gaze-analysis (Saranpää et al., 2023), brainstorming, and debriefing. We included these two demos to lower the barrier for participants (as we are informed from the survey that many developers do not know what a contemporary eye tracker looks like and how eye-tracking has been experimented with software development) and to elicit ideas around it. However, since the participants were new to this form of collaboration, we did not prescribe that the focus had to be eye-tracking to get the best out of the workshop. Instead, we presented a list of pain points (e.g., code comprehension, tool setup, dependency management) and we asked participants to select one or two pain points as the problem of their interest. We asked them to brainstorm ideas to address the selected pain points using but not limited to machine learning or eye-tracking.

The pain points we presented were those reported by the developers from the previously mentioned survey. We suggested machine learning and eye-tracking as the potential underlying technologies for the designs because the developers in the survey were positive toward the former and neutral toward the latter. Other candidate technologies such as gamification toward which the developers from our survey were negative were dropped.

3.3. Conceptual Design

Following up on the design workshop, the first author developed three conceptual designs in the form of wireframes (Guilizzoni, n.d.). According to Johnson and Henderson, a conceptual design is a high-level model that describes the major design metaphor or analogy (J. Johnson & Henderson, 2002). We also developed personas (Rogers, Sharp, & Preece, 2011) to characterize prospective users. As Pruitt stated (Pruitt & Grudin, 2003), persona suits early-stage PD very well as it can unearth sociopolitical issues and is a great complement to other scenario-based usability methods. Our designs focus on leveraging eye-tracking, speech/voice, and machine learning as a kind of support to alleviate the key pain point – code comprehension that has been brought up during the design workshop for developers.

3.4. Prototyping

After internally discussing the designs within the author group, and with some colleagues with expertise in interaction design, the first author created a low-fidelity prototype using the popular design tool Figma (Figma, n.d.). According to Rogers et al. (Rogers et al., 2011), prototyping is a cheap way for designers to actualize design ideas and conduct user tests to select from design options. The first and second authors approached a few target users through their networks, some of these target users were the same persons as those who had expressed an interest in attending the earlier design workshop but missed it due to availability issues at the time.

3.5. User Test

The first author conducted user tests with five participants (referred to as PT1-PT5 later) via Zoom. Three are PhD students from the same university as the first author, all with significant industrial programming experience, and two external PhD students from a different university in Germany, with an academic background in researching Human-Computer Interaction of programming tools. None of the participants had participated in the earlier design workshop. We chose Zoom to conduct the user test even with the participants who work at the same university as the first author because they may not be physically in the same space on the same day and to capture their screens and the interviews. The first author video-recorded the test sessions with the users' consent.

The user tests followed the process of introduction, think-aloud testing, a semi-structured short interview, and debriefing. The five users conducted the user tests separately with the first author. The sessions started with the background of the study and instructions on how to adjust the size of the interface of the online Figma prototype for later use, the remaining steps were carried out as outlined below. After the user tests the recorded material was analyzed, also described below.

3.5.1. Think-aloud Testing

After that, the users were asked to explore the prototype think-aloud with the first author observing. The first two participants were not given a specific task, because the first author was interested in learning about how they would discover the entries of the features. This was to partly mimic what would happen after a new icon, for a new functionality, was introduced into a complex IDE. However, this caused some disorientation for these two participants, so the first author prescribed a task of reading and understanding the code for the later participants.

3.5.2. Semi-structured Interview

Next, the users were interviewed by the first author with structured questions as follows:

1. How would you describe your overall experience of using this prototype?
2. What did you like and dislike about the prototype?
3. Are there any parts confusing to you?
4. What parts would you change or remove?
5. Do you have any comments, particularly for any part of the prototype or the session?

The interviewer probed with spontaneous questions wherever there was an interest in further investigation or clarification.

3.5.3. Debriefing

Finally, the first author debriefed the user test session and prototype design to the participants. This was especially needed for some of the participants who were new to such concepts or study methods and artifacts. This also aligned with the "*mutual learning*" between the users and the designer that participatory design promotes. The sessions lasted for 25 to 45 minutes.

3.5.4. Data Analysis

For the interviews during the user tests, we used Microsoft Word's premium feature of auto-transcribing to generate the transcripts. The first author, who was also the interviewer with the users, skimmed each of them and marked the lines that were related to each of the features of the prototype. Based on this, the first author summarized the points for each feature. And when there was anything unclear, the first author re-consulted the videos.

3.6. Threats to Validity

We identify the following threats in our study:

Construct validity: For the design exploration, our approach may not authentically comply with how it is prescribed to be done in the textbook. This is partly because participatory design is a rich theory and overlaps with many neighboring design methods such as user-centered design, collaborative design, and interaction design. It is a learning process for us to better understand it by practicing it. There are also some components we adapted for the sake of participants or the process, e.g., we deliberately used "co-design" instead of "participatory design" to communicate with our participants. In our view, co-design is a simpler term and more self-explanatory. It conveys better the partnership and the action required to our participants who do not necessarily know the design theories. As co-design is commonly viewed as a branch of or an interchangeable, modern term for participatory design (Wikipedia, 2024), we deem this impact minimal. Further, as the original contributors of participatory design stated, there is no uniform way to practice this method (Schuler & Namioka, 1993).

Internal validity: The user test data was auto-transcribed so a complete coverage of the conversations was ensured. Although the second and third authors did not read the transcripts themselves, the first

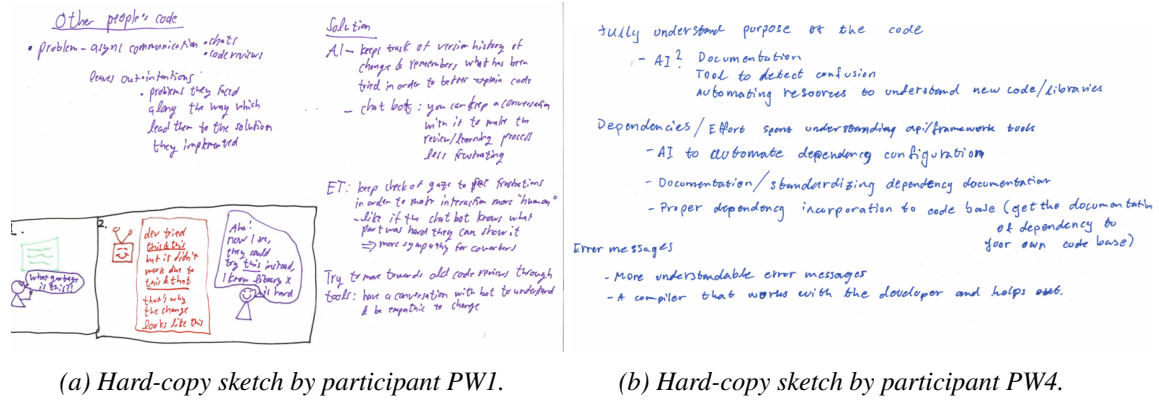


Figure 2 – Design workshop output: Sketches from PW1 and PW4.

author located the corresponding paragraphs through keyword-matching and marked them to allow the other two authors to cross-check when writing. Also, because the first author was the one who conducted the interviews with the participants during the user test sessions, the hands-on experience helps reduce the possibility of misinterpretation of the data.

External validity: For the design workshop and user test, our participants lean toward an academic profile. This is because of the challenging availability of industrial practitioners and the timing of being close to big holidays. The homogeneity of this aspect may bias the design outputs and findings from the user test. Hence, their generalizability shall be interpreted with caution.

4. Results

We present the results from the design workshop, conceptual design, prototyping and user test. We refer to participants in the workshop with the prefix "PW" and participants in the user test with the prefix "PT".

4.1. Design Workshop

Four out of the five participants picked a pain point related to code comprehension, especially understanding the code written by others. As shown in Figure 2a, participant PW1 mentioned the scenario of reviewing others' code, proposing an AI-enabled, interactive chatbot that allows consecutive conversations with the programmer to better explain code with richer context information, e.g., the history of changes and what has been attempted. Participant PW1 also touched upon tracking programmers' gaze to pinpoint frustrations so as to introduce more "human" support. Participant PW4 (Figure 2b) also selected the problem of "fully understand purpose & the code". The participant considered AI as a viable solution to "detect confusion" of programmers and to "automating resources" to facilitate the understanding of new code or libraries.

According to Figure 3, we can see participant PW2 also mentioned utilizing Large Language Models to generate summaries or explanations for a code base or variables of interest and to retain context within a time frame for debugging queries. PW3 wished for better support for finding bugs caused by unknown knowledge which therefore is very difficult to fix. He verbally elaborated on an example that was associated with changed dimensions and values of matrices during his composition of machine learning models. The host PW5 also picked a problem related to code comprehension and suggested several ways of providing gaze- and voice-based support to improve it. Participant PW3 reacted very positively to the voice-based proposal.

4.2. Conceptual Design

Consolidating the ideas and discussion inputs from the design workshop, the first author translated a selection of elements from the design workshop into three main conceptual designs, while taking into account the feasibility of prototyping and implementation as well as finding a balance between truly good

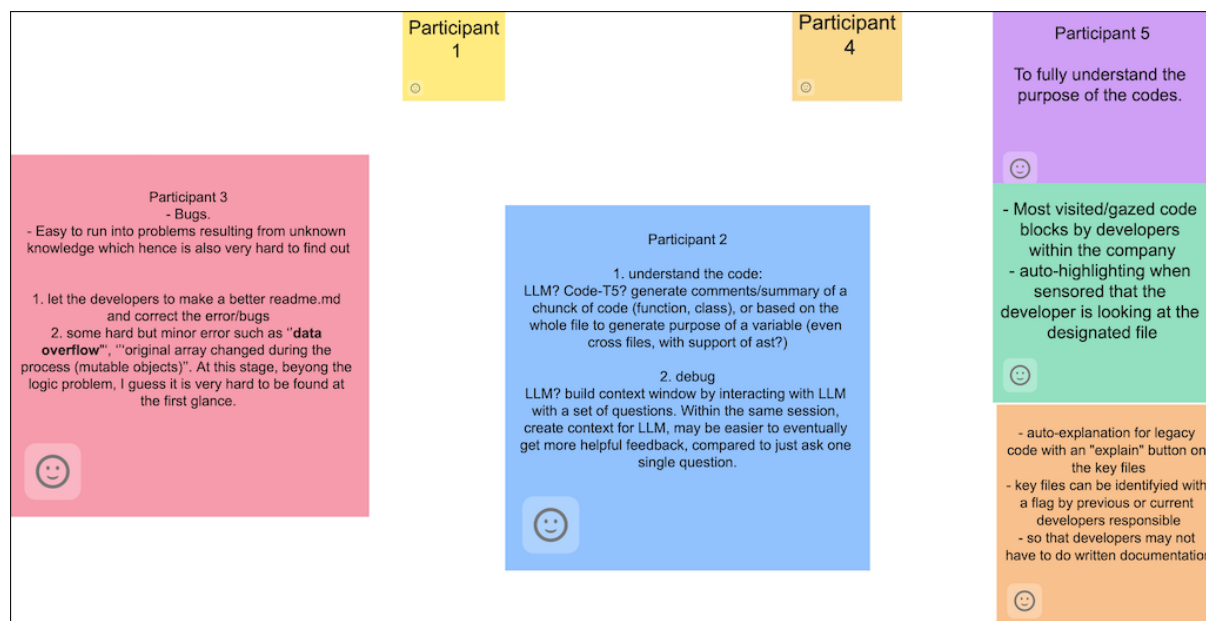


Figure 3 – Design workshop output: Digital post-it notes on Box Canvas by participants PW2, PW3, and the host PW5.

ideas and the research focus. The three conceptual designs are an individual view of the developer’s gaze with personal work notes, a collective view of a developer team’s gaze, and a conversational code explainer empowered by ML.

- The individual view (Figure 4a) mimics a text marker, which auto-highlights the code fragments for a developer. These code fragments will be based on the developer’s gaze metrics, e.g., the top 3 code fragments with the longest dwell time. We also added a component of work notes to this personalized space.
- The collective view (Figure 4b) metaphorizes a mirror that reflects or synthesizes a developer team’s gaze behavior on a high level. For instance, for each file of a huge program, it will show the top 5 code fragments that have been fixated by the team collectively for the longest time.
- The code explainer (Figure 4c) is conceptualized as similar to a tour guide working in a museum, who is knowledgeable about all the items, e.g., historical artworks, to give as much information as needed to the queries from visitors a.k.a. developers.

Lastly, we also developed personas (Figure 5) to capture the prominent profiles among the developers we observed from the survey and design workshop. The three personas represent seasoned professional developers, scientific programmers, and early-career junior developers, respectively.

4.3. User Test

The conceptual designs were translated into a low-fidelity prototype with four corresponding features. Two features are related to gaze, one to voice, and one to AI. The individual and collective views were converted into the gaze marker and gaze mirror. The voice notes resemble the work notes. The code explainer becomes the conversational AI assistant. The concept of work notes was actualized primarily in the voice notes feature but also partially as the filtered history of commands used in the AI assistant feature. The latter design echoes the finding from a survey study that developers often use ChatGPT for syntax recall (Liang et al., 2024). A demo of the prototype can be viewed here: <https://youtu.be/J9cGrK4oZ5U>

We recruited five experienced programmers to participate in the user test. Each of them tested the prototype in a think-aloud manner separately with the first author observing.

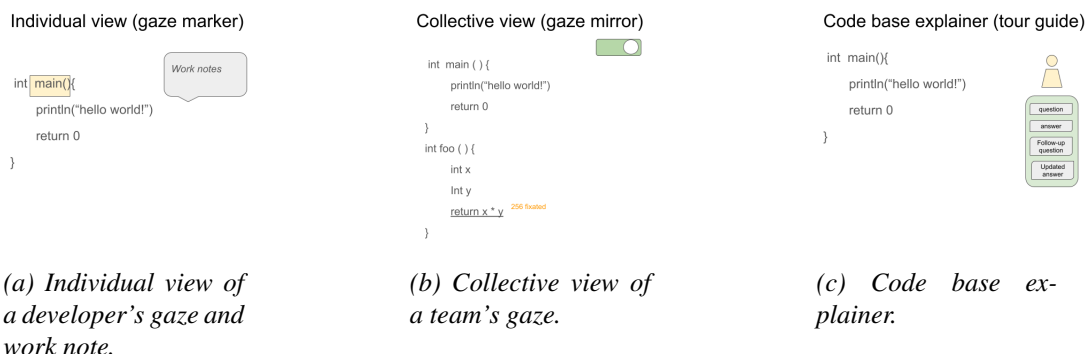


Figure 4 – Conceptual designs.

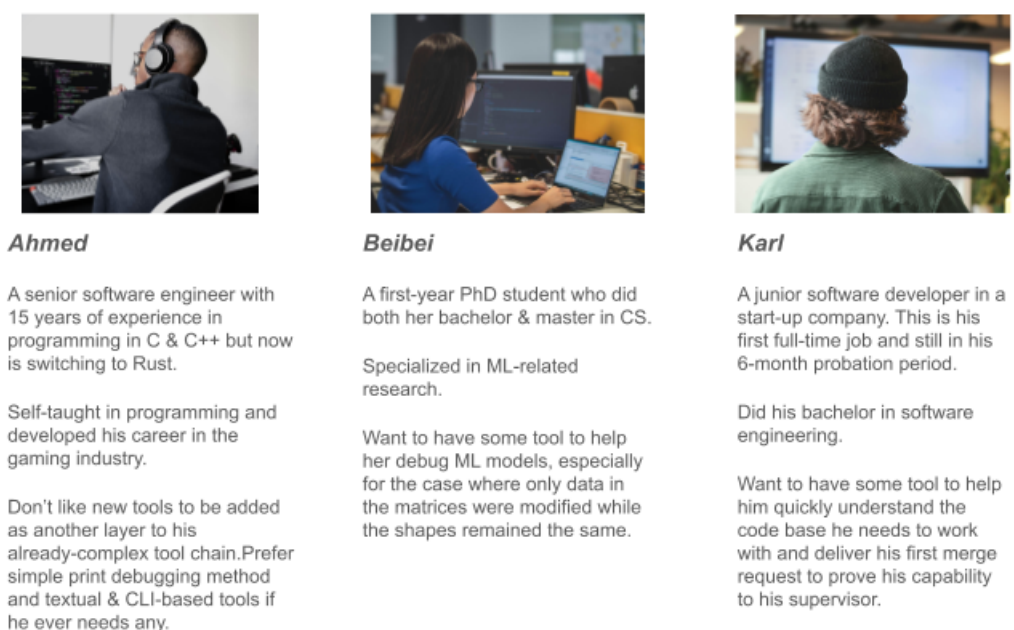


Figure 5 – Conceptual design output: Personas.

4.3.1. Individual View and Collective View

For the two gaze-related features (Figure 6a and Figure 6b), one out of the five users showed an evident interest in them, while three indicated moderate interest in seeing alternatives. Although the participants may not agree with the current visualization and/or the selection of the gaze data, they demonstrated some extent of acceptance of the leverage of gaze data in a programming environment or certain software development scenarios. Participant PT1 and PT2 somewhat agreed with its potential of being useful in certain cases specifically tailored to them, while participant PT3 disagreed with the usefulness of leveraging gaze data in the design. They explained:

Participant PT1: “I think I would be interested to see anyone’s trace just to get an understanding of what the data looks like right so.”

Participant PT2: “...but maybe if you’re like in a big project, there’s like some code that never gets any love because nobody’s looking at, maybe that would make sense trying to find stuff that nobody’s looking at.”

Participant PT3: “I don’t think it’s that useful, because yes, I probably looked at this line

of code the most for like three seconds or five seconds, but do I actually have to care about that? Why should I care?"

In summary, participants perceived these two features as somewhat interesting in the way they reveal programmers' gaze behavior but had different opinions on what way it should reveal the gaze and to what extent the gaze should be revealed to be recognized as useful. Participant PT1 was very interested in seeing others' individual gaze traces and defining their gaze traces for didactic purposes. Participant PT2 thought that locating the least gazed code of a file would be more informative. Participant PT4 suggested it would be more interesting to mark the code that their co-workers looked but they missed out. Participant PT5 was also interested in seeing others' gaze traces but only of those of their interest.

4.3.2. Voice-based Work Notes

For the feature of voice notes (Figure 6c), it was received worse although some participants praised this idea in the workshop and on another occasion. When it was presented as a tangible feature in a prototype, it was less favored. Some factors were attributed to the limitations of the mock-up design itself such as limited interactivity and lack of accuracy. But primarily it was because the participants were not convinced of the value it was proposing - to be an alternative for text comments or written documentation. Participants had a very critical and practical eye when examining it as a concrete feature of the prototype.

Participant PT3: *"But then regarding the voice note, I think theoretically it provides an alternative of not looking at the code by just hearing some voice explanation or summary of the code. But I mean there will be some obstacles of really using this feature, because first of all, why are you looking at the code? You do not bother the other colleagues or team members in the same office, but if you want to play voice notes then you probably need to plug in your headset."*

This kind of issue was escalated to how the voice notes get updated if the developer has changed the code, what will happen if the code is pushed to a remote repository, and so on. These all are rather reasonable and practical considerations. Nonetheless, participants PT2 and PT5 still saw its suitability for some rarer cases. Participant PT2 stated the potential of using it to record "meta-commentary" for the students instead of inserting "a giant comment in the middle of code". Participant PT5 pointed out that this kind of feature may be an enabler for "dyslexic" users or non-native speakers who prefer listening to audio over reading texts for learning. We also proposed the user scenarios: when developers reach fatigue from reading code, they can switch to this alternative; when their co-workers, e.g., who is the original author of a critical method, are unavailable for consulting, this can be of help.

4.3.3. Conversational AI Code Assistant

Lastly, for the conversational AI-enabled code assistant (Figure 6d), the fact that the choice of example conversation lacked a connection with the code presented in the editor led to some users' strong dissatisfaction with this feature.

Participant PT4: *"I mean, this seems completely disconnected. It shows terminal commands, it has nothing to do with the code. It says it's a code assistant but it's like my bash history excerpts."*

The same perception of lack of relevance and thus low-value or valueless was shared by participants PT3 and PT5. Participant PT5 stated,

Participant PT5: *"I don't really see how the assistant fits into the other features and I think the product as such would be leaner without that."*

While participant PT3 held the same opinion as PT5 in the beginning, he shifted his attitude when the interviewer probed with improvement or alternative ideas and he believed in the potential usefulness of those propositions.

The participant's view pivoted when a new design was proposed, for instance, participant PT3 reflected as follows:

Participant PT3: *“The last feature I mean, if it's just general code assistant, they're giving some hints like, you know, when you usually when we usually start an idea like IntelliJ or Eclipse. It will prompt up. It will pop up with some general...programming hints or some short shortcuts that you might commonly use, but this kind of assistance [is] very general. It has nothing to do with the code.”*

When asked if an added action that would analyze the current code snippet or summarize the current file, would be interesting, participant PT3 responded:

Participant PT3: *“Yeah, definitely... I mean, that is something that I always looking at ... For example, if you review any code pretty much you don't directly get into the code, you will read the comment to the Java doc...the documentation for it first, try to understand the what is the intention of this method. Or what they say, [the] intention of this class. So regarding the particular method, what...the input value [and] output value in which format? So I mean you have to get some kind of general sense what this class or this method is for and if that information is provided by this code assistant, I would consider it's useful.”*

Participants PT1 and PT2 were more open to this feature in part merely because of the concept of having an interactive code assistant. This might be influenced by their academic background and research interest in human-computer interaction.

In addition, the filtered history of commands used also triggered feedback from some participants. PT1 and PT2 acknowledged the usefulness of the most frequently used commands. PT1 further commented that rarely used commands might be more useful, as users tend to forget them due to their infrequent use. On the other hand, PT4 did not think it would be very useful and suggested this problem could be resolved by searching one's command records, for example, on the Linux command line interface. This part of the AI assistant feature did not receive any particular comments from the remaining two participants.

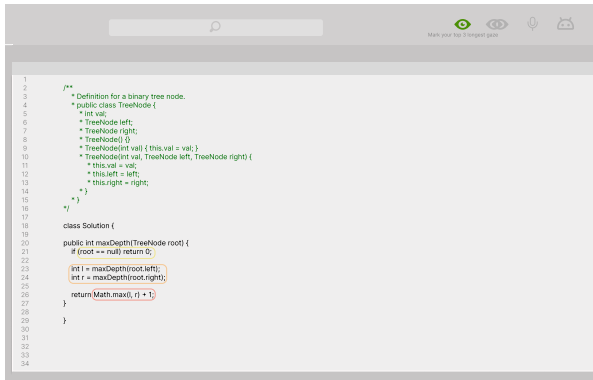
Summary

The design process comprised a pipeline of different components each with its low-level focus and staged goal. However, they curated learning together for the designers to better understand the user needs and to construct or modify the design representations. On the other hand, the users also became more aware of what the proposed and designed assistance would look like through the designers' idea pitches and their hands-on experiences interacting with the tangible artifacts.

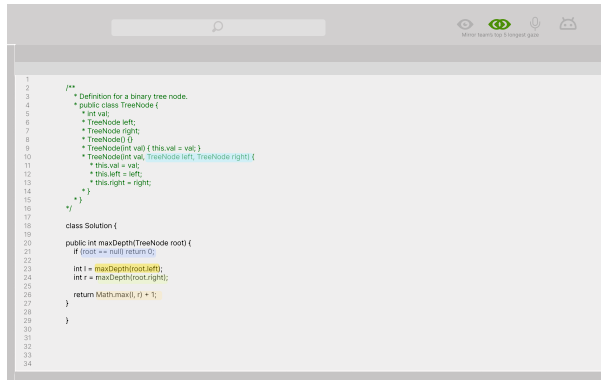
Specifically, from the user test, we found participants are interested in the gaze-related features because of the novelty of the underlying technology, the interest in inspecting self's and others' gaze and their deviations, and the potential they see that it can be applied to other scenarios such as pedagogical and accessibility use cases. The voice notes and AI code assistant received less interest because some participants were concerned with the practical use (the former) or distracted by the surface-level usability weaknesses (the latter) of the feature.

5. Discussion

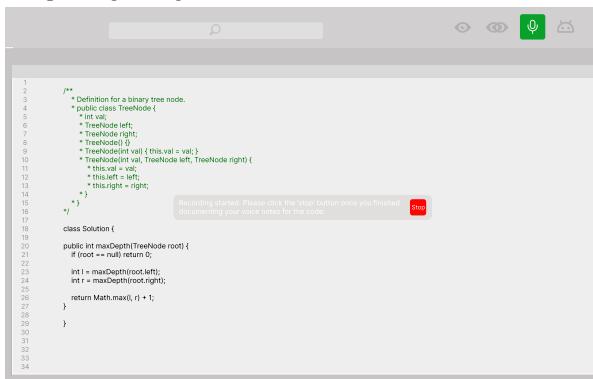
In this section, we draw on the high-level results to revisit the RQ. We examine the connection between our study and related work. We also reflect on the method and share the future work that we contemplated.



(a) Prototype feature 1: individual view of developer’s gaze (gaze marker).



(b) Prototype feature 2: collective view of a developer team’s gaze (gaze mirror).



(c) Prototype feature 3: voice explanation (tour guide).



(d) Prototype feature 4: AI code assistant (tour guide & personalized work notes).

Figure 6 – Prototyping output: Screenshots of the four features of the prototype.

The answer to the RQ “What do developers want the next-gen programming assistance to look like?” is multi-layered. The layers correspond to the components of the design pipeline. From the survey, we learned that developers are positive toward AI and unsure about eye-tracking as the enabling technology for future programming tool assistance. For the novel sensor eye-tracking, they did express privacy concerns. During the design workshop, participants exhibited varying degrees of motivation or interest in utilizing AI/ML, gaze, and voice to sketch solutions for the pain points that the current programming tools seemingly failed to address. These selections were later conceptualized as different functionalities alleviating a common pain point – code comprehension. Incorporating these functionalities gave birth to a prototype of a multi-modal IDE which reflects the designers’ interpretation of what the developers desired.

From the user tests with this prototype, we learned that experienced programmers want different programming tool assistance, but share a common perception that the design must be of practical use in helping them understand code more efficiently or effectively. Although the designs of the features that we prototyped are different from what we saw in related works, they did not gain particular favor from experienced programmers. We believe this implies that the underlying technology may have the potential to support developers, but finding the best shape of design encapsulating it may be challenging. However, compared with the voice-based feature, the gaze-based features appeared to have attracted more interest from the participants. This may be due to the efficiency of producing and consuming voice content. For instance, the effort needed to design the content for a useful voice note and to organize it well verbally (e.g., in terms of fluency, clarity, and tone variation) may surpass the effort needed for writing the documentation in some cases. Furthermore, although the question about privacy was not directly asked during the design workshop and user test, the participants never brought up the concerns

either. There seems to be a discrepancy between the concerns written in the survey and the ones expressed in the hands-on and face-to-face sessions. This seems to resemble what researchers said about consumers on using AI (Siau & Wang, 2020). That is, users may “pay lip service” to privacy issues but are pragmatic in their behavior.

5.1. Reflection on the Method in the Study

The design workshop is one of the most challenging parts of this study. Recruitment of human participants is usually expensive. We assume this is partly attributed to the limited availability of practitioners during work days. While filling out a survey takes 10 to 15 minutes, attending a workshop with active involvement is more demanding and usually requires 45 to 60 minutes. Further, we speculate that participation in a research study during a workday might be perceived as unethical by some practitioners who work in the industry if not in the context that the researchers are collaborating with their employers. This explains that the participants who expressed interest in attending and made it to attend our workshop were mostly from academia.

For the user tests, our participants have an academic background or a background of having recently shifted their career from industry to academia. With capable experienced programmers, we advise researchers to use as realistic code as possible and with reasonable complexity and cognitive load. Educational code snippets such as data structure manipulation and algorithms may still be perceived as simple by some practitioners. In addition, we recommend giving participants a concrete task that they are used to or can relate to such as finding errors/bugs in the code even though the main goal is to test out the usefulness and usability of the design. This together elevates the realism of the user scenario and immersion of the task which in turn helps elicit more realistic reactions from participants and thus more useful data. Researchers also need to take care of the link between the features designed and the code presented. Participants tend to expect some coherency between them rather than treating them separately. Letting participants use the features with a connection to the code presented can avoid distracting them from the main task.

5.2. Directions for Future Research

For future work, we plan to select gaze-driven assistance as the primary feature for iterations and refinements as it receives the most interest and positive reactions or beliefs from participants in the user test sessions. We will revise the design of the gaze feature with the inputs and implement a high-fidelity prototype, either in the form of a custom IDE/code editor or a plugin published in one of the mainstream IDEs.

Reflecting on the design process, we learned that the junior developer persona (which shares characteristics with novice programmers to a large extent) may be the most beneficial programmer cohort for us to work with. First, we think experienced programmers or experts tend to have well-established programming tool preferences, e.g., a specific tool or debugging method such as the simple but effective print statement. Some of such programmers involved in our design process demonstrated a more critical and skeptical attitude toward the new enabling technologies that we proposed (it is even more prominent in the survey component/study (Kuang et al., 2024)). This implies that perhaps they are less enthusiastic about or open to novel programming tool assistance. Additionally, they are deemed to be rather resourceful and capable of helping themselves. Hence, there is less room for this type of design to be useful to them.

Second, with the scientific programmer persona, we observe the problems that they deal with are inclined to be data- and ML-centric. This demands domain-specific tool support that embeds deep knowledge of such problems. We see our design as a less fit for this goal as we envisioned it to be independent of the actual functionality of the code, that is, to be at the presentation layer but not the logic layer of the code. Lastly, we believe there are scenarios where early-career junior developers and prospective professional developers such as novice programmers are outstandingly goal-driven to deliver the results, for instance, to push the first pull request in their new job or get the group assignment done on time. They are potentially more open to diverse forms of support, especially given the fact that how they

learn programming is drastically different from early generations of programmers, e.g., via ChatGPT or Co-pilot nowadays. In particular, we want to assist them with reading and understanding a code base for the first time. Studies (Green et al., 2023) report that it can take new software engineers 3 to 5 months to familiarize themselves with a new code base to be productive. Similar challenges may surface for prospective professional developers when collaborating on pre-scaffolded group assignments and contributing to open-source projects.

We further want to explicitly evaluate the PD method in a systematic way as per the recommendations from a survey on the use of PD (Bossen, Dindler, & Iversen, 2016) and a comprehensive review of this method in a related field (Spinuzzi, 2005). We wish to derive some representative quantitative criteria with developers to triangulate the benefits and gains of the use of PD, together with the qualitative data that we have partly reported in this paper. Lastly, we will also keep an eye on whether there is room for adopting AI/ML to enhance the gaze-driven assistance that we are going to realize.

6. Conclusions

In conclusion, we followed a design process that involved developers from the beginning to the end under the guidance of Participatory Design. We first surveyed professional developers about the pain points in their work and their attitude toward new technologies. Next, we organized a design workshop with five participants (including the host) to brainstorm and sketch out what programming tool assistance they wanted with the enabling technologies AI/ML and eye-tracking (toward which the developers from our survey have indicated positive or neutral attitudes). We then translated these inputs into three conceptual designs and developed three personas to capture the profiles that emerged from the previous modules. We further prototyped these designs as four features of a low-fidelity, multi-modal IDE. Finally, we tested this digital, interactive prototype with five experienced programmers.

From the discussions and interviews with these prospective users, we found that developers are open to new types of assistance powered by new and novel technologies. However, for them to be useful, their assistance must increase developers' efficiency or productivity. More specifically, developers from the user test recognize greater potential in gaze-driven assistance than in voice-based assistance facilitating code comprehension in an IDE.

7. Acknowledgements

We thank all the participants who took part in any component of the design process. We thank them for their curiosity, enthusiasm, creativity, and insights.

We would further like to thank the following funders who partly funded this work: the Swedish strategic research environment ELLIIT, the Swedish Foundation for Strategic Research (grant nbr. FFL18-0231), the Swedish Research Council (grant nbr. 2019-05658), and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

8. References

- AI, C. (2024, 3). *SWE-bench technical report*. (Retrieved 4 Apr, 2024 from <https://www.cognition-labs.com/post/swe-bench-technical-report>)
- Apple. (2023, 10). *The Home View on Apple Vision Pro*. Retrieved from <https://www.apple.com/newsroom/2023/06/introducing-apple-vision-pro/>
- Bannon, L., Bardzell, J., & Bødker, S. (2018, feb). Introduction: Reimagining participatory design—emerging voices. *ACM Trans. Comput.-Hum. Interact.*, 25(1). Retrieved from <https://doi.org/10.1145/3177794> doi: 10.1145/3177794
- Benoit, C., Martin, J.-C., Pelachaud, C., Schomaker, L., & Suhm, B. (2000). Audio-visual and multimodal speech systems. *Handbook of Standards and Resources for Spoken Language Systems-Supplement*, 500, 1–95.
- Bossen, C., Dindler, C., & Iversen, O. S. (2016). Evaluation in participatory design: a literature survey. In *Proceedings of the 14th participatory design conference: Full papers - volume 1*

- (p. 151–160). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2940299.2940303> doi: 10.1145/2940299.2940303
- Bødker, S., Dindler, C., & Iversen, O. S. (2022). *Participatory design*. Springer Nature.
- Carroll, J. M., & Rosson, M. B. (2007). Participatory design in community informatics. *Design studies*, 28(3), 243–261.
- Cheng, S., Wang, J., Shen, X., Chen, Y., & Dey, A. (2022, 06). Collaborative eye tracking based code review through real-time shared gaze visualization. *Frontiers of Computer Science*, 16. doi: 10.1007/s11704-020-0422-1
- Clark, B., & Sharif, B. (2017). itracevis: Visualizing eye movement data within eclipse. In *2017 IEEE working conference on software visualization (vissoft)* (p. 22–32). doi: 10.1109/VISSOFT.2017.30
- Clement, A. (1996). Computing at work: empowering action by “low-level users”. *Computerization and controversy: value conflicts and social choices*, 383.
- Ehn, P., & Sandberg, Å. (1979). *Företagsstyrning och löntagarmakt: planering, datorer, organisation och fackligt utredningsarbete*. Prisma i samarbete med Arbetslivscentrum.
- Figma. (n.d.). *Figma: The collaborative interface design tool*. Retrieved from <https://www.figma.com/>
- Glücker, H., Raab, F., Echtler, F., & Wolff, C. (2014). Eyede: gaze-enhanced software development environments. In *Chi'14 extended abstracts on human factors in computing systems* (pp. 1555–1560).
- Green, C., Jaspán, C., Hodges, M., He, L., Shen, D., & Zhang, N. (2023). Developer productivity for humans, part 5: Onboarding and ramp-up. *IEEE Software*, 40(5), 13–19. doi: 10.1109/MS.2023.3291158
- Guilizzoni, P. (n.d.). *What are Wireframes?*. Retrieved from <https://balsamiq.com/learn/articles/what-are-wireframes/>
- Gumm, D. C., Janneck, M., & Finck, M. (2006). Distributed participatory design—a case study. In *Proceedings of the dpd workshop at nordichi* (Vol. 2).
- Hellman, J., Cheng, J., & Guo, J. L. (2021). Facilitating asynchronous participatory design of open source software: Bringing end users into the loop. In *Extended abstracts of the 2021 chi conference on human factors in computing systems* (pp. 1–7).
- Hijazi, H., Cruz, J., Castelhana, J., Couceiro, R., Castelo-Branco, M., de Carvalho, P., & Madeira, H. (2021). ireview: an intelligent code review evaluation tool using biofeedback. In *2021 IEEE 32nd international symposium on software reliability engineering (issre)* (p. 476–485). doi: 10.1109/ISSRE52982.2021.00056
- Hull, M. F. (2021). The role of technical communicators in open-source software: A systematic review.
- Iivari, N. (2009). “constructing the users” in open source software development: An interpretive case study of user participation. *Information Technology & People*, 22(2), 132–156.
- 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
- Johnson, J., & Henderson, A. (2002, jan). Conceptual models: begin by designing what to design. *Interactions*, 9(1), 25–32. Retrieved from <https://doi.org/10.1145/503355.503366> doi: 10.1145/503355.503366
- Kuang, P., Söderberg, E., & Höst, M. (2024). Developers' perspective on today's and tomorrow's programming tool assistance: A survey. In *10th edition of the programming experience workshop, px/24*.
- Kuang, P., Söderberg, E., Niehorster, D. C., & Höst, M. (2023). Toward gaze-assisted developer tools. In *2023 IEEE/ACM 45th international conference on software engineering: New ideas and emerging results (icse-nier)* (p. 49–54). doi: 10.1109/ICSE-NIER58687.2023.00015
- Letondal, C., & Mackay, W. E. (2004). Participatory programming and the scope of mutual responsibility: balancing scientific, design and software commitment. In (p. 31–41). New York, NY,

- USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/1011870.1011875> doi: 10.1145/1011870.1011875
- Liang, J. T., Yang, C., & Myers, B. A. (2024, apr). A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *2024 ieee/acm 46th international conference on software engineering (icse)* (p. 605-617). Los Alamitos, CA, USA: IEEE Computer Society. Retrieved from <https://doi.ieeecomputersociety.org/>
- Maudet, N., Leiva, G., Beaudouin-Lafon, M., & Mackay, W. (2017). Design breakdowns: Designer-developer gaps in representing and interpreting interactive systems. In *Proceedings of the 2017 acm conference on computer supported cooperative work and social computing* (p. 630-641). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2998181.2998190> doi: 10.1145/2998181.2998190
- McCabe, A. T., Söderberg, E., Church, L., & Kuang, P. (2022). Visual cues in compiler conversations. In S. Holland, M. Petre, L. Church, & M. Marasoiu (Eds.), *Proceedings of the 33rd annual workshop of the psychology of programming interest group, PPIG 2022, the open university, milton keynes, UK & online, september 5-9, 2022* (pp. 25-38). Psychology of Programming Interest Group. Retrieved from <https://ppig.org/papers/2022-ppig-33rd-mccabe/>
- Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3), 309-346.
- Myers, B. A., Ko, A. J., LaToza, T. D., & Yoon, Y. (2016). Programmers are users too: Human-centered methods for improving programming tools. *Computer*, 49(7), 44-52. doi: 10.1109/MC.2016.200
- Oviatt, S., & Cohen, P. (2000). Perceptual user interfaces: multimodal interfaces that process what comes naturally. *Communications of the ACM*, 43(3), 45-53.
- Paudyal, B., Creed, C., Frutos-Pascual, M., & Williams, I. (2020). Voiceye: A multimodal inclusive development environment. In *Proceedings of the 2020 acm designing interactive systems conference* (pp. 21-33).
- Pruitt, J., & Grudin, J. (2003). Personas: practice and theory. In (p. 1-15). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/997078.997089> doi: 10.1145/997078.997089
- Radevski, S., Hata, H., & Matsumoto, K. (2016). Eynav: Gaze-based code navigation. In *Proceedings of the 9th nordic conference on human-computer interaction*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2971485.2996724> doi: 10.1145/2971485.2996724
- Rogers, Y., Sharp, H., & Preece, J. (2011). *Interaction Design: Beyond Human-Computer Interaction*. Retrieved from <http://discovery.ucl.ac.uk/1326236/>
- Santos, A. L. (2021). Javardeye: Gaze input for cursor control in a structured editor. In *Companion proceedings of the 5th international conference on the art, science, and engineering of programming* (p. 31-35). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3464432.3464435> doi: 10.1145/3464432.3464435
- Saranpää, W., Apell Skjutar, F., Heander, J., Söderberg, E., Niehorster, D. C., Mattsson, O., ... Church, L. (2023). Gander: a platform for exploration of gaze-driven assistance in code review. In *Proceedings of the 2023 symposium on eye tracking research and applications*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3588015.3589191> doi: 10.1145/3588015.3589191
- Schuler, D., & Namioka, A. (1993). *Participatory design*.
- Shakil, A., Lutteroth, C., & Weber, G. (2019). Codegazer: Making code navigation easy and natural with gaze input. In *Proceedings of the 2019 chi conference on human factors in computing systems* (p. 1-12). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3290605.3300306> doi: 10.1145/3290605.3300306
- Siau, K., & Wang, W. (2020). Artificial intelligence (ai) ethics: ethics of ai and ethical ai. *Journal of Database Management (JDM)*, 31(2), 74-87.

- Spinuzzi, C. (2005). The methodology of participatory design. *Technical communication*, 52(2), 163–174.
- Talon. (2024). *Talon: Powerful Hands-free Input*. (Retrieved 3 Apr, 2024 from <https://talonvoice.com/>)
- Tobii. (2023). *Eye tracking fully integrated and baked right into the very latest high performance gaming devices from alienware, acer and msi*. (<https://gaming.tobii.com/products/laptops/> (Feb 15, 2023))
- Wikipedia. (2024, 5). *Participatory design*. Retrieved from https://en.wikipedia.org/wiki/Participatory_design
- Wubishet, Z. S., Bygstad, B., & Tsiavos, P. (2013). A participation paradox: Seeking the missing link between free/open source software and participatory design. *Journal of Advances in Information Technology*, 4(4), 181–193.
- Yang, J., Jimenez, C. E., Wettig, A., Yao, S., Narasimhan, K., & Press, O. (2024). *Swe-agent: Agent computer interfaces enable software engineering language models*.
- Zhang, Y., Ruan, H., Fan, Z., & Roychoudhury, A. (2024). *Autocoderover: Autonomous program improvement*.

Assessing Consensus of Developers' Views on Code Readability

**Agnia Sergeyuk, Olga Lvova, Sergey Titov,
Anastasiia Serova, Farid Bagirov, Timofey Bryksin**
JetBrains Research

{agnia.sergeyuk, olga.lvova, sergey.titov}@jetbrains.com
{anastasiia.serova, farid.bagirov, timofey.bryksin}@jetbrains.com

Abstract

The rapid rise of Large Language Models (LLMs) has changed software development, with tools like Copilot, JetBrains AI Assistant, and others boosting developers' productivity. However, developers now spend more time reviewing code than writing it, highlighting the importance of Code Readability for code comprehension. Our previous research found that existing Code Readability models were inaccurate in representing developers' notions and revealed a low consensus among developers, highlighting a need for further investigations in this field.

Building on this, we surveyed 10 Java developers with similar coding experience to evaluate their consensus on Code Readability assessments and related aspects. We found significant agreement among developers on Code Readability evaluations and identified specific code aspects strongly correlated with Code Readability. Overall, our study sheds light on Code Readability within LLM contexts, offering insights into how these models can align with developers' perceptions of Code Readability, enhancing software development in the AI era.

1. INTRODUCTION

Large Language Models (LLMs) have seen rapid advancement, particularly in software development applications, where they serve as coding assistants and power tools like Copilot¹, JetBrains AI Assistant², Codeium³, and others.

The evolution of AI-supported programming tools is reshaping software development practices — while AI enhances productivity, developers spend more time reviewing code than writing it (Mozannar, Bansal, Fourny, & Horvitz, 2023). Given that code comprehension time is generally related to Code Readability—the easier the code is to read, the less time it takes for the developer to comprehend it—optimizing the programmer's workflow involves providing suggestions from an LLM that align with developers' understanding of Code Readability.

In academia, Code Readability is defined as a subjective, mostly implicit human judgment of how easy the code is to understand (Posnett, Hindle, & Devanbu, 2011; Buse & Weimer, 2008; Scalabrino, Linares-Vasquez, Poshyvanyk, & Oliveto, 2016). However, aligning LLMs with developers' understanding of Code Readability necessitates an explication of developers' notion of what is readable code.

In our previous research (Sergeyuk et al., 2024), we studied if existing predictive models of Code Readability (Posnett et al., 2011; Scalabrino, Linares-Vásquez, Oliveto, & Poshyvanyk, 2018; Dorn, 2012; Mi, Hao, Ou, & Ma, 2022) may be a proxy of developers' Code Readability notion. This study, in addition to defining 12 Code Readability-related aspects obtained via the Repertory Grid Technique (Edwards, McDonald, & Michelle Young, 2009), revealed a weak correlation between current Code Readability models and developer evaluations, pointing to a significant gap in these models' ability to reflect developers' perspectives on Code Readability. It underscored the need for developing more accurate Code Readability metrics and models. We also found that developers' evaluations of Code Readability were not always consistently aligned with one another. We hypothesize that these results are dictated by the subjectivity of Code Readability and its aspects, along with other confounding variables.

¹Copilot <https://github.com/features/copilot>

²JetBrains AI Assistant <https://plugins.jetbrains.com/plugin/22282-jetbrains-ai-assistant>

³Codeium <https://codeium.com/>

Therefore, we present a work that builds on top of the previous study, presenting the results of a survey we executed to delve deeper into developers' agreement level on Code Readability and its aspects.

We conducted a survey involving 10 Java developers from the same company, all with similar coding experience. Our aim was to assess whether a group of developers with similar backgrounds would reach a consensus on Code Readability assessments and related aspects, and which of those aspects are correlated the most with Code Readability. Each developer evaluated the same set of 30 Java code snippets using a 5-point Likert scale, rating code across 13 Code Readability-related dimensions.

The results of the study indicate a statistically significant intraclass correlation on Code Readability and several related metrics. This suggests that there is a degree of agreement among developers regarding what constitutes Code Readability. We also found a significant correlation of 12 Code Readability-related aspects evaluations with an assessment of Code Readability itself. It implies that LLMs could be tailored to Code Readability notion by adjusting metrics that are stable among developers and strongly related to Code Readability.

Overall, our work represents an approach to a deeper and at the time more explicit understanding of Code Readability concept among developers, which is instrumental in the present rapidly evolving AI-centered world of software development.

2. BACKGROUND

The development of code-fluent LLMs as coding assistants has fundamentally transformed the coding experience. Several research studies were conducted to examine how humans and AI interact in-depth and to understand how LLMs influence programmer behavior during coding activities, *e.g.*, (Mozannar et al., 2023; Liang, Yang, & Myers, 2023; Vaithilingam, Zhang, & Glassman, 2022; Barke, James, & Polikarpova, 2023).

A comprehensive study conducted by researchers from Cambridge and Microsoft informed the understanding of how developers interact with AI tools and how to improve this experience (Mozannar et al., 2023). Mozannar and colleagues studied the impact of GitHub Copilot on programmers' behavior during coding sessions. As a result, they identified 12 common programmer activities related to AI code completion systems. The researchers found that developers spend more time reviewing code than writing it. Indeed, approximately 50% of a programmer's coding time involved interactions with the model, with 35% dedicated to double-checking suggestions.

The investigation by Carnegie Mellon University researchers underscored the challenges encountered by developers while working with AI coding assistants (Liang et al., 2023). Their survey results suggest that developers mainly use these tools to save time, reduce keystrokes, and recall syntax. However, according to the same survey's results, the generated code is limited in meeting both functional and non-functional requirements. Additionally, developers struggle to comprehend the outputs of LLM due to the code being too long to read quickly.

Previous studies highlighted the importance of aligning Code Readability models with human notions, reducing coders' time and mental effort to comprehend AI coding assistants' suggestions. Developers need to quickly comprehend the code proposed by an AI coding assistant before integrating it into a project and implementing any changes. A critical aspect of this process is what is commonly referred to as Code Readability — the ease with which developers can read and understand code. In this notion, Code Readability forms a perceived barrier to comprehension that developers must overcome to efficiently work with code (Posnett et al., 2011; Buse & Weimer, 2008; Scalabrino et al., 2018).

Addressing developers' expectations regarding the readability of model-suggested code may involve various fine-tuning methods. Specifically, in addition to the fine-tuning process itself, when the developer modifies the model's weights and parameters, contrastive (Le-Khac, Healy, & Smeaton, 2020) and reinforcement (Lambert, Castricato, von Werra, & Havrilla, 2022) learning are valuable tools for this purpose. Implementing these methods encompasses the definition of a learning objective — informa-

tion about what output is “desirable” and what is not. Frequently, this objective is formed by annotating the models’ outputs or by formulating rules that indicate users’ satisfaction with the produced code.

Our previous research (Sergeyuk et al., 2024) explored the potential of existing state-of-the-art Code Readability models (Posnett et al., 2011; Dorn, 2012; Scalabrino et al., 2018, 2016; Mi, Keung, Xiao, Mensah, & Gao, 2018; Mi et al., 2022) to be learning objectives to guide the process of fine-tuning.

Posnett et al.’s Model. Posnett, Hindle, and Devanbu introduced a Simpler Model of Code Readability based on three features: Halstead volume, token entropy, and line count, surpassing the performance of Buse and Weimer’s earlier model (Posnett et al., 2011). They employed forward stepwise refinement for feature selection, manually incorporating features driven by intuition and familiarity with Halstead’s software metrics.

Dorn’s Model. Dorn developed a General Software Readability Model, expanding beyond Java to include multiple programming languages (Dorn, 2012). Dorn’s approach extended beyond syntactic analysis to include structural patterns, visual perception, alignment, and natural language elements, transformed into numerical vectors. This model also outperformed the retrained Buse and Weimer’s model, emphasizing the value of using a wider range of code characteristics in readability assessments.

Scalabrino et al.’s Model. Scalabrino, Linares-Vasquez, and Oliveto proposed a Comprehensive Model integrating syntactic, visual, structural, and textual elements of code (Scalabrino et al., 2018). Their binary Code Readability classifier with 104 features surpassed previous models, emphasizing the benefit of textual alongside structural and syntactic features.

Mi et al.’s Model. Mi, Hao, Ou, and Ma introduced a deep-learning-based Code Readability model leveraging visual, semantic, and structural code representations (Mi et al., 2022). This model outperformed traditional machine learning models on a combined dataset, showcasing the potential of deep learning in automating Code Readability evaluation.

In our previous study, we utilized the Repertory Grid technique (Kelly, 2003) to establish a user-centric understanding of aspects related to Code Readability. This understanding served as a proxy for a unified perception of Code Readability among the developers who participated in the consequent survey. During this survey, they assessed code snippets on various readability-related aspects and provided an overall judgment on whether the presented snippet was readable or not. The data from the survey was then used to assess the agreement between the models described above and human evaluations of Code Readability. Overall, we found 12 readability-related bi-polar code aspects presented in Table 1.

Our research uncovered discrepancies in the correlation between existing Code Readability models and its human evaluations. While Scalabrino’s model (Scalabrino et al., 2018) showed a moderate correlation with human assessments, other models like Posnett et al.’s (Posnett et al., 2011), Dorn’s (Dorn, 2012), and Mi et al.’s (Mi et al., 2022) demonstrated weaker correlations. This variation suggests that using these Code Readability models as learning objectives to fine-tune code-fluent LLMs for improved readability might not be optimal. A more precise model is needed to guide this adjustment process.

Additionally, we found that developers assess Code Readability inconsistently, highlighting the need for more standardized and validated definitions of Code Readability.

To address these findings, our current study aims to investigate if confounding variables contributed to the previously observed inconsistency in Code Readability assessments by developers and if agreement on Code Readability is achievable. We also seek to identify stable aspects of Code Readability that could serve as a foundation for defining Code Readability and forming learning objectives for future LLM adjustments. Specifically, our research questions are as follows:

RQ 1. Do Java developers with similar backgrounds consistently assess Code Readability and its related aspects?

RQ 2. Do previously elicited code aspects represent Code Readability?

Readable pole	Unreadable pole
Code is concise	Code is too long
Code reads well from top to bottom	While reading, the eyes jump from top to bottom and back up again
*Code is not sufficiently explained and needs additional info to understand what it does	Code is overexplained
The goal of the code is clear	The goal of the code is not clear
Code uses basic, known code patterns	Code looks unfamiliar, non-standard
Functionality is separated logically	Code needs refactoring
Code is flat and linear	Code is overly nested
There is one action per line of code	There are multiple actions on one line
Code uses named constants	Code uses “magic numbers”
Naming clarifies code functionality	Naming is confusing
Code conforms to style guides	Code is poorly formatted
There is balance in the color blocks	There are huge chunks of color blocks that stand out in a distracting way

*This characteristic forms a continuum, being “Readable” in the middle and “Unreadable” at the extremes.

Table 1 – Code Readability Aspects

3. METHODOLOGY

3.1. Sample

The sample was gathered by sending the survey link to the internal channels of JetBrains with the invitation to Java programmers to participate in the study on Code Readability.

Based on preliminary power analysis, the sample was designed to consist of 10 Java programmers with varying experience levels. Despite their different experience levels, we assume that they share the same understanding of functional and non-functional requirements, as they have actively participated in developing a shared codebase.

All participants in our study are proficient Java developers. We assessed their experience using both subjective and objective measures. Six participants self-assessed as “Advanced”, indicating *extensive* experience and high proficiency in Java programming. Four participants self-assessed as “Intermediate”, signifying a *strong* understanding and ability to work on complex projects. The distribution of objective experience measures is presented in Table 2.

Experience	Frequency
More than 10 years	4
9–10 years	2
5–6 years	1
3–4 years	1
1–2 years	2

Table 2 – Distribution of Years of Experience

3.2. Materials

In the current survey, we employed materials gathered in the previous study (Sergeyuk et al., 2024) — a set of 30 AI-generated Java code snippets and a rating list of 12 Code Readability aspects. We justified the reuse of these materials to retest our previous approach and investigate if our data collection and analysis methodology might have influenced the earlier results on the agreement of code readability assessment by humans. Therefore, we maintained consistency by using the same approach and materials but with greater attention to confounding variables and data analysis.

The code snippets represented the readability of outputs generated by LLMs, which is important for us

due to the fact that the overarching goal of this research is to enhance the Human-AI Experience. To create snippets, we selected tasks from the Code Golf game⁴ as prompts for ChatGPT 3.5 Turbo (due to the timing of the study) to generate Java language solutions for these tasks. Subsequently, we ensured that the snippets were meaningful and executable, adhering to a length limit of 50 lines, as defined by previous Code Readability models examined in prior research.

The primary aim of the list of Code Readability-related aspects (see Table 1), which we formulated from in-depth interviews using the Repertory Grid technique, was to offer consistent guidance to respondents during the evaluation of Code Readability. This aimed to ensure that developers assessed code uniformly and focused on key aspects related to readability.

3.3. Data Collection

On the greeting page of the survey, participants gave their consent and professional background information. After that, they were presented with a random sequence of the same set of 30 Java code snippets. Participants evaluated Code Readability of the snippet using the list of 12 bipolar characteristics and Code Readability itself with a five-point Likert scale measuring how much the code leans to the readable or unreadable pole. Participants could take breaks while completing the task, leading to completion times ranging from 40 minutes to 5 hours. Therefore, we believe that the random presentation of tasks and the flexible break schedule mitigated the effects of fatigue on the evaluations.

3.4. Data Analysis

To answer **RQ 1**, we calculated the intraclass correlation coefficient (ICC) to assess the agreement level of developers evaluating Code Readability and its aspects (Liljequist, Elfving, & Skavberg, 2019).

Additionally, to answer **RQ 2**, we calculated the Pearson's correlation coefficient (Cohen et al., 2009) of Code Readability-related aspects evaluations with overall Code Readability scores to see what metrics are related to Code Readability.

4. FINDINGS

RQ 1. Do Java developers with similar backgrounds consistently assess Code Readability and its related aspects?

The agreement level on assessments of Code Readability and related code aspects was found to be mostly from moderate to good (Koo & Li, 2016). The numerical values of ICC with corresponding Medians are presented in Table 3. The results of our study support the idea that developers of similar backgrounds would agree on evaluations of Code Readability and its related aspects.

Prior Code Readability studies show that human annotators exhibited imperfect agreement, with a correlation around .5 with the mean readability score (Buse & Weimer, 2008; Dorn, 2012). In our previous study (Sergeyuk et al., 2024), we did not find even this level of agreement. This discrepancy with the current results might be accounted for by the developers' shared backgrounds. In the current study, developers had similar backgrounds, all having experience consistently contributing to a specific code-base. Therefore, their views on some programming conventions are closed. In contrast, the developers in our previous study had a wide range of years of experience and worked at vastly different companies. Additionally, it might be the case that using Krippendorff's alpha with many missing values affected our previous findings, and that effect was mitigated by our data gathering this time. Namely, we avoid missing values in our study by presenting a fixed set of the same snippets to a fixed number of nonrandom raters and calculating ICC on that data.

Findings from the current study support the possibility of aligning LLMs' outputs with users' notions of readability. However, such alignment may be uniquely achievable within a specific company or among a group of developers with close views on various coding practices.

It is also noteworthy that not all Code Readability-related aspects have received a significant level of

⁴Code Golf game <https://code.golf/>

Code Aspect	Poles (if represented numerically — from 2 to -2)	ICC	Median
Readability	Readable / Unreadable	0.78	1
Code Structure	Functionality is separated logically / Code needs refactoring	0.81	1
Nesting	Code is flat and linear / Code is overly nested	0.80	2
Understandable Goal	The goal of the code is clear / The goal of the code is not clear	0.79	2
Code Length	Code is concise / Code is too long	0.78	2
Inline Actions	There is one action per line of code / There are multiple actions on one line	0.76	2
Reading Flow	Code reads well from top to bottom / While reading, the eyes jump from top to bottom and back up again	0.75	2
Sufficient Contextual Info	Code is not sufficiently explained and needs additional info to understand what it does / Code is overexplained	0.74	0
Code Style	Code conforms to style guides / Code is poorly formatted	0.70	1
Magic Numbers	Code uses named constants / Code uses "magic numbers"	0.69	0
Naming	Naming clarifies code functionality / Naming is confusing	0.67	1
Code Patterns	Code uses basic, known code patterns / Code looks unfamiliar, non-standard	0.53	2
Visual Organization	There is balance in the color blocks / There are huge chunks of color blocks that stand out in a distracting way	-0.03	0

Significant ICC values ($p < 0.05$) are highlighted in bold.

Table 3 – Agreement on Code Readability

agreement between developers. *Visual Organization* scale, which represents the balance between color blocks in the code snippets, *e.g.*, several lines of comments or big arrays, has a nonsignificant level of agreement. Having nonformal feedback from participants, we hypothesize that the wording and concept of this scale were unclear for developers and should be refined in future studies.

RQ 2. Do previously elicited code aspects represent Code Readability?

The results indicate that aspects of Code Readability correlate moderately to strongly with the Code Readability itself. We present a heatmap of statistically significant correlations in Figure 1.

Code aspects from our study, which we identified through in-depth interviews using the Repertory Grids Technique with developers, align with prior research and models of Code Readability. This alignment, along with the way these aspects were elicited, provides some grounds to hypothesize that they are indeed connected with Code Readability. Characteristics from our study resemble the combination of structural characteristics with visual, textual, and linguistic features as proposed by later Code Readability models (Dorn, 2012; Scalabrino et al., 2018; Mi et al., 2022). Moreover, Fakhoury et al. (Fakhoury, Roy, Hassan, & Arnaoudova, 2019) investigated commits that were explicitly aimed at Code Readability-enhancement and observed notable changes in *Complexity*, *Documentation*, and *Size* metrics that resemble *Code Structure*, *Nesting*, *Sufficient Contextual Info*, and *Code Length* metrics from our list. In the study of Fakhoury et al., it was also noted that *Code Style* and *Magic Numbers Usage* are the aspects where improvements in Code Readability-related commits are prominent. In another study, Peitek et al. (Peitek, Apel, Parnin, Brechmann, & Siegmund, 2021) examined 41 complexity metrics and their influence on program comprehension, discovering that factors such as *Textual Length* and *Vocabulary Size* increase cognitive load and working memory demand for programmers.

Further evidence supporting the idea that the code aspects we elicited in our previous study represent developers' notion of Code Readability is the statistically significant correlation between the entire list of 12 Code Readability-related aspects and evaluations of Code Readability itself. However, there are some differences in the strength of these correlations. The strongest correlation of Code Readability evaluation is with Naming, Code Length, Understandable Goal, and Reading Flow metrics. Combined

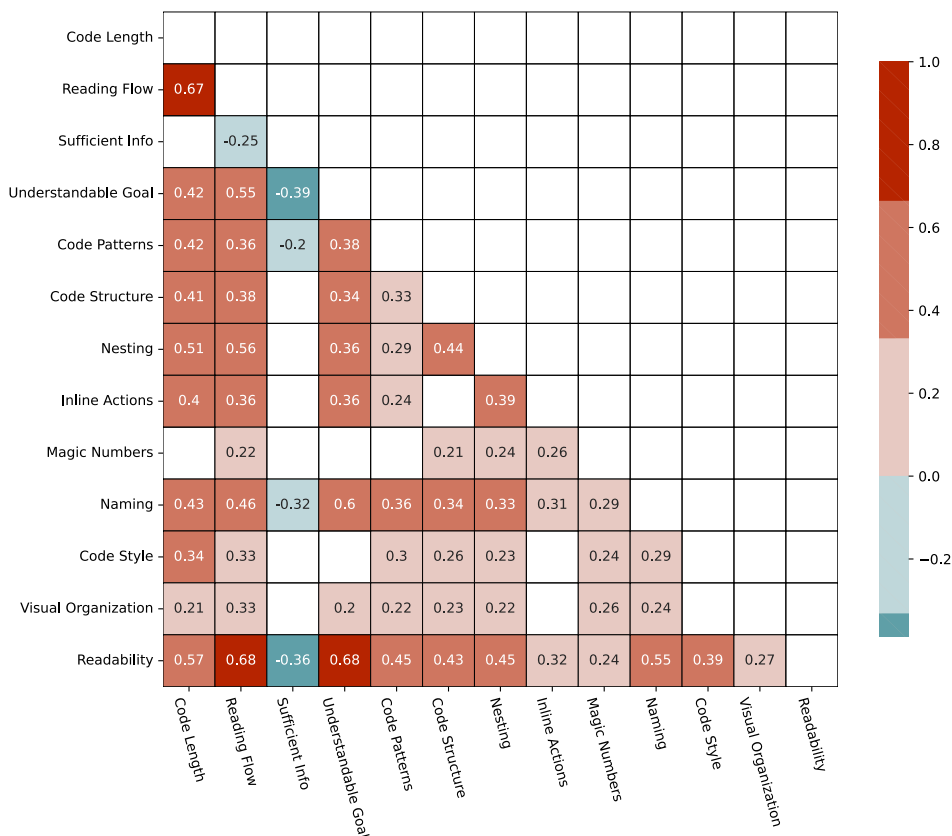


Figure 1 – Correlations of Code Readability-related aspects

with the fact that Code Length and Understandable Goal are also metrics that gained a good level of agreement among developers who assessed snippets, we can hypothesize that these two code aspects are most representative of Code Readability and could be used as guidance for LLMs alignment.

5. CONCLUSION AND FUTURE WORK

This study explored the possibility of agreement among developers on Code Readability evaluations, with the aim of potentially utilizing Code Readability as a learning objective for LLMs. Our findings indicate that developers with similar professional backgrounds tend to exhibit a good level of agreement in Code Readability evaluations. Additionally, certain code aspects related to Code Readability, *i.e.*, *Code Length* and *Understandable Goal*, demonstrate promising potential as representatives of the key scales influencing Code Readability.

With this supporting evidence in hand, our future endeavors will focus on further exploration of Code Readability aspects and their potential representations for LLM adjustment with the overarching objective of enhancing user experience with AI assistants in programming.

6. References

Barke, S., James, M. B., & Polikarpova, N. (2023, apr). Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1).

Buse, R. P., & Weimer, W. R. (2008). A metric for software readability. In *Proceedings of the 2008 international symposium on software testing and analysis* (p. 121–130). Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/1390630.1390647> doi: 10.1145/1390630.1390647

Cohen, I., Huang, Y., Chen, J., Benesty, J., Benesty, J., Chen, J., ... Cohen, I. (2009). Pearson correlation coefficient. *Noise reduction in speech processing*, 1–4.

- Dorn, J. (2012). *A general software readability model*. Retrieved from <http://www.cs.virginia.edu/weimer/students/dorn-mcs-paper.pdf>
- Edwards, H. M., McDonald, S., & Michelle Young, S. (2009). The repertory grid technique: Its place in empirical software engineering research. *Information and Software Technology*, 51(4), 785-798. Retrieved from <https://www.sciencedirect.com/science/article/pii/S0950584908001298> doi: <https://doi.org/10.1016/j.infsof.2008.08.008>
- Fakhoury, S., Roy, D., Hassan, A., & Arnaoudova, V. (2019). Improving source code readability: Theory and practice. In *Proceedings of the 27th international conference on program comprehension* (p. 2-12). IEEE. Retrieved from <https://doi.org/10.1109/ICPC.2019.00014> doi: 10.1109/ICPC.2019.00014
- Kelly, G. (2003). *The psychology of personal constructs: Volume two: Clinical diagnosis and psychotherapy*. Routledge.
- Koo, T. K., & Li, M. Y. (2016). A guideline of selecting and reporting intraclass correlation coefficients for reliability research. *Journal of chiropractic medicine*, 15(2), 155–163.
- Lambert, N., Castricato, L., von Werra, L., & Havrilla, A. (2022). Illustrating reinforcement learning from human feedback (rlhf). *Hugging Face Blog*. (<https://huggingface.co/blog/rlhf>)
- Le-Khac, P. H., Healy, G., & Smeaton, A. F. (2020). Contrastive representation learning: A framework and review. *IEEE Access*, 8, 193907-193934. doi: 10.1109/ACCESS.2020.3031549
- Liang, J. T., Yang, C., & Myers, B. A. (2023). *Understanding the usability of ai programming assistants*.
- Liljequist, D., Elfving, B., & Skavberg, K. (2019). Intraclass correlation—a discussion and demonstration of basic features. *PLoS one*, 14(7), e0219854.
- Mi, Q., Hao, Y., Ou, L., & Ma, W. (2022). Towards using visual, semantic and structural features to improve code readability classification. *Journal of Systems and Software*, 193(C). Retrieved from <https://doi.org/10.1016/j.jss.2022.111454> doi: 10.1016/j.jss.2022.111454
- Mi, Q., Keung, J., Xiao, Y., Mensah, S., & Gao, Y. (2018). Improving code readability classification using convolutional neural networks. *Information and Software Technology*, 104, 60-71. Retrieved from <https://www.sciencedirect.com/science/article/pii/S0950584918301496> doi: <https://doi.org/10.1016/j.infsof.2018.07.006>
- Mozannar, H., Bansal, G., Fournay, A., & Horvitz, E. (2023). *Reading between the lines: Modeling user behavior and costs in ai-assisted programming*.
- Peitek, N., Apel, S., Parnin, C., Brechmann, A., & Siegmund, J. (2021). Program comprehension and code complexity metrics: An fmri study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (p. 524-536). doi: 10.1109/ICSE43902.2021.00056
- Posnett, D., Hindle, A., & Devanbu, P. (2011). A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories* (p. 73–82). Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/1985441.1985454> doi: 10.1145/1985441.1985454
- Scalabrino, S., Linares-Vásquez, M., Oliveto, R., & Poshyvanyk, D. (2018). A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6), e1958. Retrieved from <https://doi.org/10.1002/smr.1958> doi: 10.1002/smr.1958
- Scalabrino, S., Linares-Vasquez, M., Poshyvanyk, D., & Oliveto, R. (2016). Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)* (p. 1-10). IEEE. doi: 10.1109/ICPC.2016.7503707
- Sergeyuk, A., Lvova, O., Titov, S., Serova, A., Bagirov, F., Kirillova, E., & Bryksin, T. (2024). *Re-assessing java code readability models with a human-centered approach*.
- Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (p. 1-7). Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3491101.3519665> doi: 10.1145/3491101.3519665

For Modeling Programmers as Readers with Cognitive Literary Science

Rijul Jain
Williams College
rijul.jain@williams.edu

Abstract

The prevalent text-based form of human-generative AI interactions has blurred the lines between code and prose. I argue that understanding the cognitive effects of these interactions by modeling users not only as programmers, but as readers, will inform the design of better tools to bolster human agency in generative AI interactions. I hope to begin a conversation around the uses of cognitive literary science for the study of the psychology of programming.

Reflections

The analogy between the “complexity of a large and thoughtful literary novel” and that of “a large computer program” holds because both literature and computer programs deal in “sophisticated information representations” (Blackwell, 2023). The forms of these representations change—especially now, as using natural language to interact with generative AI in increasingly formalized ways (or, prompt programming) has created a new way to conceive the forms that programming can take. Whether they have ever written code or not, users of generative AI interfaces are therefore often programming in some respect—but they are simultaneously put in the position of readers, reacting to and being acted upon by the varied, sometimes surprising outputs given by, for example, ChatGPT, while also directing the series of interactions to write the narrative of the exchange.

Questions of improving human agency with respect to new programming notations and generative AI have therefore never been more keenly related to developing and learning from cognitive models not only of programmers, but of readers working upon and being worked upon by texts. At PPIG 2023, Lewis (2023) indirectly took up these concerns by inquiring whether the “psychology of natural semantics” would “become a central part of the psychology of programming,” with a renewed focus on studying end-user programming with LLMs. Yet these issues bear on expert programmers, too—Floyd et al. (2017) find in an fMRI study of programmers’ brains that experts “treat code and prose more similarly at a neural activation level” than non-experts.

Hermans et al. (2017) frame programming and writing as closely related. To understand generative AI users as programmers even more comprehensively, it will also prove fruitful to model them as readers by attending to cognitive literary science—research using cognitive psychology and neuroscience to explicate the mental processes and the corner cases of reading. Gerrig et al. (2003) lay out readers’ processes of continually updating mental representations of narrative experiences—their work shows literature to be a fertile ground for throwing into relief cognitive processes relevant to the psychology of programming. In particular, Bergs’ (2017) research on coercion, or “the resolution of formal mismatch,” uses examples from literature to illustrate and complement neuroscience work on a fundamental cognitive phenomenon—one crucial to understanding how users react to unassimilably and unrectifiably anomalous output from generative AI.

More generally, modeling programmers as readers will advance research directions aimed at designing for diverse programming notations and bolstering human agency. Crichton et al. (2021), for example, draw from cognitive psychology—evaluating the load on programmers’ working memory during program tracing—to inform conclusions about programming tool design. Following this approach with cognitive literary science to understand the cognitive effects of programmers’ higher-level, often narrativized generative AI interactions may similarly reveal patterns and pitfalls to inform better design for generative AI tools. Doing so even more broadly will also improve computational creativity—e.g. Chandra et al., (2023) who create a new framework for animation in part by modeling their audience with cognitive-science-informed narrative theory. Efforts at computational creativity going forward must be attuned at a cognitive level to the ways people experience the arts.

Current forms of text-based interaction with generative AI systems have blurred the lines between code and prose. As their forms of representation tend toward convergence, and if indeed “AI is a branch of literature,” (Blackwell, 2023) understanding and evaluating how programmers-as-readers interact with speech in this generative AI context will propel the development of more reliable and usable technical notation than natural language alone. Insights from cognitive literary science may then yield a fuller picture of the possibilities of enabling human agency with respect to programming.

References

- Bergs, A. (2017). Under Pressure: Norms, Rules, and Coercion in Linguistic Analyses and Literary Readings. In Burke, M., and Troscianko, M. T. (eds), *Cognitive Literary Science: Dialogues between Literature and Cognition*, (Oxford: Oxford University Press).
- Blackwell, A.F. (2023). Chapter 14: Re-imagining AI to invent more Moral Codes. Retrieved from: <https://moralcodes.pubpub.org/pub/chapter-12/release/4>
- Chandra, K., Li, T., Tenenbaum, J. & Ragan-Kelley, J. (2023). Acting as Inverse Inverse Planning. In ACM SIGGRAPH 2023 Conference Proceedings (SIGGRAPH '23), 7, 1–12. <https://doi.org/10.1145/3588432.3591510>
- Crichton, W., Agrawala, M., & Hanrahan, P. (2021). The Role of Working Memory in Program Tracing. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21), 56, 1–13. <https://doi.org/10.1145/3411764.3445257>
- Floyd, B., Santander, T., & Weimer, W. (2017). Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), (175-186). doi: 10.1109/ICSE.2017.24.
- Gerrig, R. J., & Egidi, G. (2003). Cognitive psychological foundations of narrative experiences. In D. Herman (Ed.), *Narrative theory and the cognitive sciences*, 33–55. Center for the Study of Language and Information.
- Hermans, F. & Aldewereld, M. (2017). Programming is Writing is Programming. In Companion Proceedings of the 1st International Conference on the Art, Science, and Engineering of Programming (Programming '17), 33, 1–8. <https://doi.org/10.1145/3079368.3079413>
- Lewis, C. (2023). Large Language Models and the Psychology of Programming. In Proceedings of the 34th Annual Conference of the Psychology of Programming Interest Group (PPIG 2023), 77-95.

Predictability of identifier naming with Copilot: A case study for mixed-initiative programming tools

Michael Jing Long Lee
Computer Laboratory
University of Cambridge
mjl12@cam.ac.uk

Advait Sarkar
Microsoft Research
advait@microsoft.com

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

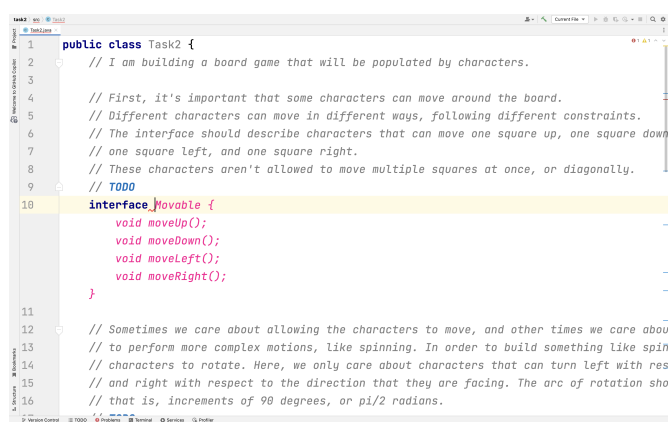
Abstract

Studies show that predictive text entry systems make writing faster, but written content more predictable. We consider if these trade-offs extend to code synthesis tools such as GitHub Copilot. While Copilot can make developers produce code faster, it may also affect how they choose identifiers for methods and classes. This may have non-trivial effects on the activity of programming, because identifier names are a primary semantic signal in code, and play important roles in authoring, debugging, and developer communication. In a controlled, within-subjects experiment ($n=12$), we compared identifiers chosen in the presence and absence of Copilot suggestions. We find that identifiers chosen in the presence of Copilot suggestions were significantly more predictable (have lower mean entropy), even when suggestions were only visible and could not be automatically accepted. These results imply that mixed-initiative systems can take an active role in shaping programmer intentions and potentially impact their sense of agency. We consider whether an increased convergence towards predictable names is an asset or a liability for the practice of programming, and suggest design opportunities for surfacing surprising identifiers and conceptual refactoring tools.

1. Introduction

Code synthesis tools based on generative Large Language Models (LLMs), such as GitHub Copilot (hence **Copilot**), have been widely adopted by developers and firms (Dohmke, 2023). In February 2023, Copilot was estimated to produce 46% of code across all programming languages, a percentage that had doubled over the previous year (Zhao, 2023). Unlike traditional code-completion tools, Copilot has the ability to suggest multiple lines of code at once, and to recommend potential identifiers (Ziegler et al., 2022). It has been found to increase productivity – both actual (Mozannar et al., 2023), and perceived (Peng et al., 2023; Ziegler et al., 2022).

We consider these programmer aids from the perspective of *mixed-initiative interaction*, in which either the user or the tool might take the next action. This means that the system must make judgments on how



```

1 public class Task2 {
2     // I am building a board game that will be populated by characters.
3
4     // First, it's important that some characters can move around the board.
5     // Different characters can move in different ways, following different constraints.
6     // The interface should describe characters that can move one square up, one square down
7     // one square left, and one square right.
8     // These characters aren't allowed to move multiple squares at once, or diagonally.
9     // TODO
10    interface Movable {
11        void moveUp();
12        void moveDown();
13        void moveLeft();
14        void moveRight();
15    }
16
17    // Sometimes we care about allowing the characters to move, and other times we care about
18    // to perform more complex motions, like spinning. In order to build something like spin
19    // characters to rotate. Here, we only care about characters that can turn left with res
20    // and right with respect to the direction that they are facing. The arc of rotation sho
21    // that is, increments of 90 degrees, or pi/2 radians.

```

Figure 1 – An example study task as seen by participants, with a suggestion by Copilot highlighted in pink. Participants were more likely to accept identifier suggestions (in this case, *Movable*), than to generate original names (e.g., *TakesSingleStep* or *Move*).

well it understands the user’s goals, and on when it might be appropriate to interrupt the user with an offer of assistance (Horvitz, 1999).

In the case of programming tools such as Copilot, the utility function for mixed initiative interaction is less easily calculated, because the only explicit “goal” of a working programmer is the system specification, itself often ambiguous or incomplete. More tractably, the programmer’s goal from moment to moment is simply to refine their understanding of the problem domain and of the required execution behaviour (Naur, 1985), expressing that developing model with conceptual clarity and economy, for example by well-chosen identifier names.

In this paper, we specifically consider the choice of identifier names as a valuable case study through which to understand the nature of interaction with Copilot from a mixed initiative perspective. Choosing good names for identifiers is a key skill of the working programmer (McConnell, 1993). For example, the name of an abstract type often reflects basic concepts in an application domain, a function name should succinctly describe the operation that will be performed, and a field in a database schema might express an important feature of a customer relationship. For programmers developing reusable frameworks, libraries and APIs, the name of each element is critical to the usability of the whole (Furnas et al., 1987).

As a result, defining, reviewing and updating identifier names is an essential conceptual element of programming work, in which the programmer both refines and communicates their understanding of the software engineering problem in a way that will be understandable by other programmers (Schankin et al., 2018).

Mixed-initiative interaction when choosing identifier names can be considered as a trade-off in attention investment (Blackwell, 2002). Mixed-initiative interaction and the attention investment model are both fundamentally about the cost-benefit tradeoff of automation (Williams et al., 2020). In this case, Copilot might suggest a conventional identifier name at relatively low attentional cost, but the programmer could alternatively invest attention in making the name more informative and specific to a distinctive context they are working in (Blackwell, 2022).

We relate this to prior work, showing that in certain cases, intelligent user interfaces that streamline *processes* to be more efficient can also make *content* more generically predictable (Arnold et al., 2020), and therefore less informative (Shannon, 1948). A common situation in machine learning-based code assistants is that the system may propose a conventional name based on its training corpus. This works well in highly standardised programming tasks such as student coding exercises, and also in very routine or conventional programming work where a single correct name might be highly predictable. The challenge that we address here comes in situations where a programming task is not standardised, perhaps in a new domain or involving an original approach. In those cases where the best name cannot be straightforwardly predicted from prior code, reuse of conventional identifiers could easily misrepresent the programmer’s intention and be subtly incorrect. Our investigation therefore focuses on this tradeoff between originality and predictability, recognising that each has its place in good quality code.

We adapt the experimental paradigm developed by Arnold et al. (2020) for study of predictive text, applying their approach in the context of source code identifiers. The authors of that study take care to note that their conclusions did not necessarily extend to tasks involving conceptual exposition. In contrast, the creation of identifiers, which describe the properties of abstract objects, is exactly the task of conceptual exposition. Our controlled, within-subjects experiment (n=12) compared identifier naming with and without Copilot support, including a condition where suggestions were only visible, but not available as automated actions.

The main contribution of this work is to demonstrate a statistically reliable effect, that the visible presence of Copilot suggestions results in more predictable identifiers, which may sometimes be desirable, but not in more novel domains or coding tasks. We consider the consequences of this phenomenon in relation to attention investment for mixed-initiative programming tools, and suggest design strategies that might mitigate the problems that can result where predictable or conventional code is not a primary

quality objective.

2. Related Work

2.1. Predictability of AI-assisted work and critical integration

The theory of *critical integration* (Sarkar, 2023b) is a general account of the nature of generative AI-assisted knowledge workflows. According to this, as the work of material production (e.g., the physical typing of text or code, or creation of images) is increasingly delegated to AI, the role of the user is to critically evaluate and integrate AI output into their broader workflow. However, the workflow itself and the user's objectives can be affected through interaction with AI output.

For example, models of interaction with predictive text systems (Bhat et al., 2023) have identified specific cognitive processes (Hayes, 2012) that are influenced by suggestions. Notably, the writer's respect for the system affects the degree to which suggestions are accepted. Additionally, suggestions shape Working Memory State. Therefore, they impact not only syntactic choices, but sentence structure and semantic content. Suggestions have even been found to influence authors' topic choices and opinions (Jakesch et al., 2023; Poddar et al., 2023).

Arnold et al. (2020)'s work is theoretically grounded in Rational Speech Act (RSA), a goal-oriented model of communication. Under RSA, speakers choose phrases by balancing utility and cost (Goodman & Frank, 2016). If words are chosen whilst writing (MacArthur et al., 2016), reducing the cost of a predictable word can prompt users to choose less informative phrases.

Buschek et al. (2021) and Singh et al. (2023) show how such findings may be operationalised, by designing predictive text interfaces that leverage cognitive impacts to aid ideation. Proposals generally involve encouraging users to *critically* integrate suggestions. They include surfacing multiple suggestions at once, and forcing explicit integration of suggestions rather than automatic acceptance.

2.2. Attention Investment

Good identifier names involve an attention investment (Blackwell, 2003): by *investing* immediate attention, programmers may choose a distinctive name that accurately summarises hundreds of lines of original code. In doing so, there is a *pay-off*: future attentional cost savings, since they or others may efficiently surmise the nature of the abstract object by its name. However, there is also a *risk* that no pay-off accrues, which varies depending on the nature of the task.

The attention investment problem is especially acute in settings where identifier names are hard to conceptualise but have the potential to be very informative (cf: Section 4.1). As identified by Blackwell (2022), such settings include

1. Naming concepts that are frequently *reused*, potentially in different settings,
2. Naming concepts that are *related*, for example, in API design, where related methods chain to form a language, and
3. Naming *refactored* concepts, where updated requirements or semantics are reflected in subtle changes to names (Blackwell, 2023).

2.3. Studies of GitHub Copilot

A systematic review of research on GitHub Copilot identified developer productivity, code quality, code security, and education as primary themes (Ani et al., 2023). Evaluation of Copilot as a mixed-initiative system tends to define utility in terms of productivity impact (Mozannar et al., 2023; Peng et al., 2023). While the effect of Copilot on identifier choice has not been considered, results show that Copilot increases productivity. However, as Buschek et al. (2021) found, ideation and efficiency are often in tension, so greater volume of code production may be associated with more conventional or homogeneous names.

Multiple evaluations of Copilot have found that while it is useful in some situations, it requires the programmer to still exercise algorithmic thinking, program comprehension, debugging and communication skills, and can prove a liability for non-expert programmers (Dakhel et al., 2023; Fajkovic & Rundberg, 2023; Imai, 2022; Zhang et al., 2023b). These have led researchers to caution against indiscriminate use of AI assistance in programming education settings (Puryear & Sprint, 2022; Wermelinger, 2023). Moreover, while the complexity and readability of Copilot-generated code is comparable to that written by humans, eye-tracking data suggests that programmers pay less visual attention to AI-generated code (Al Madi, 2022), corresponding to studies of agency in mixed-initiative interaction where users are less critical of automated suggestions when they perceive the machine as having greater agency (Yu et al., 2021).

Benchmark tests show that performance of GitHub Copilot, OpenAI ChatGPT, and Amazon CodeWhisperer can approach human level, but varies depending on the target language (Nguyen & Nadi, 2022; Yetistiren et al., 2022; Yetiştirin et al., 2024). Inappropriate sensitivity to the prompting language is also a challenge; in one study Copilot generated different code results for semantically equivalent natural language prompts in approximately 46% of the test cases (Mastroaolo et al., 2023). Moreover, while Copilot can be prompted in multiple natural languages, it is not equally performant, with one study finding that performance with Chinese language prompts was significantly worse than with English (Koyanagi et al., 2024).

Studies on developers' subjective experience (Kalliamvakou, 2023; Sarkar et al., 2022; Vaithilingam et al., 2022; Vasconcelos et al., 2023; Zhang et al., 2023a; Zhou et al., 2023) and mental models (Mozannar et al., 2022) have additionally found that Copilot reduces perceived mental effort and that users often accept suggestions without verification, which they defer to some future point. Such deferrals, as well as the introduction of suboptimal solutions or unaddressed issues which can interfere with future software development, can contribute to technical debt (O'Brien et al., 2024). Tools such as Copilot can be used to facilitate the authoring of code when programmer intent is clear, but also to aid exploration and discovery (Barke et al., 2023; Sarkar et al., 2022). While Copilot can improve efficiency, it can come at the cost of code comprehension and autonomy or control (Bird et al., 2022). An analysis of a corpus of software developers' tweets about GitHub Copilot found that programmers' negative emotions can become more positive when the capabilities of the AI tools are linked to their identity work (Eshraghian et al., 2023). When considered in the framework of attention investment, these both hint at less attention being invested into identifier names.

3. Research Questions

We aim to understand how developers are influenced by the identifiers suggested by Copilot. If developers tend to *accept* Copilot's suggestions, this may result in more predictable identifier names (the assumption being that Copilot produces more predictable names, an assumption which we discuss in Section 6). We also consider whether making it more effortful to accept suggestions, by disabling keyboard shortcuts for easy acceptance, can affect the influence of Copilot on identifier names (and thereby programmer agency). Our research questions are:

RQ1: To what extent are identifiers more predictably named in the presence of Copilot suggestions?

RQ2: To what extent do results differ if keyboard shortcuts for accepting suggestions are disabled?

4. Study Design

To evaluate the effect of Copilot suggestions on identifier choice, we conducted a within-subjects experiment in which participants completed short Java programming tasks (Section 4.1) under different levels of access to Copilot suggestions (Section 4.2). The 12 participants were computer science undergraduates at our institution, recruited via convenience sampling. All participants had prior knowledge of Java interfaces and experience in practical Java programming through undergraduate-level coursework.

Expression	Definition	Interpretation
C	A set of common concepts named by participants	NA
$\text{names}(c, t)$	The names given to concept c under treatment t	NA
$H_{\text{names}}(c, t)$	The entropy of $\text{names}(c, t)$	The <i>unpredictability</i> of the names given to a specific c under t
$\{H_{\text{names}}(c, t) \mid c \in C\}$	The set of all $H_{\text{names}}(c, t)$ in a given treatment t	Assuming $H_{\text{names}}(c, t)$ is independent of c , this estimates the <i>distribution</i> of $H_{\text{names}}(t)$
$\langle H_{\text{names}}(t) \rangle$	The mean of $\{H_{\text{names}}(c, t) \mid c \in C\}$	The predictability of the names given to an arbitrary c under t

Table 1 – Collated Definitions

4.1. Tasks

Using IntelliJ IDEA, participants defined Java interfaces based on natural language prompts. Three tasks were developed, each with the aim of reflecting a context where distinctive original names are useful, but hard to conceptualise.

Task 1 involved defining interfaces that form a pipeline for working with data. Participants had to consider the *relationships* between interfaces, and methods that could be *reused* in a variety of contexts. For example, a method for `checking` data could be called by a process writing to, or reading from, a database. The checks could differ in the two cases.

Task 2 involved defining interfaces for a game, where characters could move around a grid, and rotate in-place. Careful naming was required to capture the *relationships* between interfaces, for example, methods to move `right` and to rotate `right` might clarify if the motion is *relative* or *absolute*.

Task 3 involved participants developing a structure for managing custom user settings. Participants were asked to imagine that this was originally a command line tool, that was being replaced by a GUI. This *refactoring* task encouraged participants to consider how the changing context updates requirements, and how these updates may be reflected in changes to existing names.

The prompts were designed to avoid priming participants to pick certain identifier names adopting the method described in Liu and Sarkar, et al. (Liu et al., 2023). Tasks were described in verbose and indirect ways, encouraging participants to make new identifier choices rather than reuse vocabulary from the task descriptions.

In the original study of predictive text by Arnold et al. (2020), the experimental task involved writing image captions. The predictive text system was allowed to consider the image prompt as part of the context when generating suggestions. By analogy to that study, we included the prompt stimulus text in comment blocks so that it was visible to Copilot. Copilot may also consider content in other open files. To ensure all participants saw the same initial suggestions, the set of open files was controlled.

A full listing of our experimental tasks and prompts is given in Appendix A.

4.2. Treatments

We manipulate the visibility of suggestions and the mechanism for accepting suggestions, resulting in three conditions:

1. ON: Copilot is enabled, with keyboard shortcuts for accepting suggestions.
2. VIEW: Copilot is enabled, but keyboard shortcuts were disabled. Users could view the suggestion, but could only incorporate it in their code by manually typing it out.

3. OFF: Copilot is disabled, and no suggestions were shown.

Many IDEs, including IntelliJ IDEA, have native (non-AI based) code completion tools that are widely used in practice. These tools offer autocomplete so that programmers can repeatedly reference identifiers already present in the codebase. Because the autocomplete functionality acts as a confounding factor in this setting and interfere with programmer’s attention towards Copilot suggestions, code completion was disabled for all three treatments to preserve internal validity. This comes at a slight cost to external validity, but as IntelliJ IDEA’s native code completion tool does not suggest potential new identifiers, and our tasks did not require participants to reference the same identifier multiple times, its absence is unlikely to have been detrimental.

4.3. Protocol

The study was carried out in-person. All 12 participants declared that they were familiar with programming in Java, and read and signed a statement of informed consent. Participation in the study was voluntary and participants were not directly compensated. Our study protocol was approved by our institution’s ethics committee.

Participants were first asked to read a description of the study, which explained that they would be asked to define interfaces under three different treatments, and that the experiment was a study of Copilot. We did not explicitly draw attention to identifiers, but asked participants to consider the readability and maintainability of their code.

Before attempting any of the tasks, participants were first given a tutorial, where they familiarised themselves with defining interfaces and working with Copilot. Participants then completed each of the three tasks in turn. The assignments of tasks to conditions was counterbalanced, so that each task was completed by 4 participants each in the ON, OFF, and VIEW conditions respectively. The study sessions lasted between 45-60 minutes.

ds : An interface for reading and writing data			
	$t \in T$		
	$t = \text{ON}$	$t = \text{VIEW}$	$t = \text{OFF}$
$\text{names}(\text{ds}, t)$	DataSource, DataSource, DataSource, DataSource	DataSource, DataSource, DataSource, DataSource	Datum, GetAndSet, Manipulator, QueryData
$H(\cdot)$	0	0	2

Table 2 – Example Computation of $H_{\text{names}}(\text{ds}, t)$. $H_{\text{names}}(\text{ds}, t)$ is the entropy of the distribution of names (each column) given to ds under treatment $t \in T$.

4.4. Measures

Since all participants were given the same three task descriptions, all participants were creating names relating to the described set of concepts C .¹ As a running example, consider one such concept described in Task 1 — ds : an interface that reads and writes to some source of data. As shown in Table. 2, we consider $\text{names}(\text{ds}, t)$: the bag of names given by participants to ds under treatment $t \in \{\text{ON}, \text{VIEW}, \text{OFF}\}$.

To measure predictability, we employ Shannon Entropy, an information theoretic model for quantifying the average amount of information communicated by a source (Shannon, 1948). By measuring average surprisal, entropy quantifies unpredictability.

We ask: “What did programmer X name concept ds under treatment t ?”. $H_{\text{names}}(\text{ds}, t)$ quantifies the *uncertainty* of the answer. Mathematically, it is the entropy of the empirical distribution of the bag. If

¹Some concepts were not named by all participants. In particular, participants disagreed on how to encode inputs to functions, with some choosing not to specify them at all. These concepts were excluded.

the bag has only one unique element, the name can be predicted with certainty; the entropy is 0. In general, an entropy of n can be interpreted as the uncertainty associated with predicting the name from one of 2^n equiprobable candidates. This increases as predictability decreases.

To generalise from a single concept ds to the effect of a treatment t on an *arbitrary* concept c , we make the simplifying assumption that $H_{names}(c, t)$ is independent of c (not generally the case, but reflecting our experimental tasks). Hence, each $H_{names}(c, t)$ is an observation of the same random variable, $H_{names}(t)$, and $\{H_{names}(c, t) \mid c \in C\}$ is a sample from the underlying distribution. Let $\langle H_{names}(t) \rangle$ denote the sample mean. This is the *expected* unpredictability of an identifier under t , regardless of the concept it names. These definitions are collated in Table 1.

This analysis was repeated at the *word* level: $H_{words}(ds, t)$. This was to investigate whether Copilot encourages multiword identifiers that reshuffle words drawn from a smaller vocabulary. Finally, we noted cases where participants changed their first choice, effectively renaming the identifier, since revisiting a previous decision represents additional investment of attention by the namer.

We analysed the effect of each treatment on predictability. Two levels of granularity were considered: a fine-grained comparison of entropy *distributions* was reinforced by a coarse-grained comparison of *means*. 95% confidence intervals were estimated by bootstrap re-sampling with replacement (1000 iterations).

5. Results

5.1. Predictability

Fig. 2 plots histograms of the sample $\{H(c, t) \mid c \in C\}$ for each treatment t , as an estimate of the underlying distribution of $H(t)$.

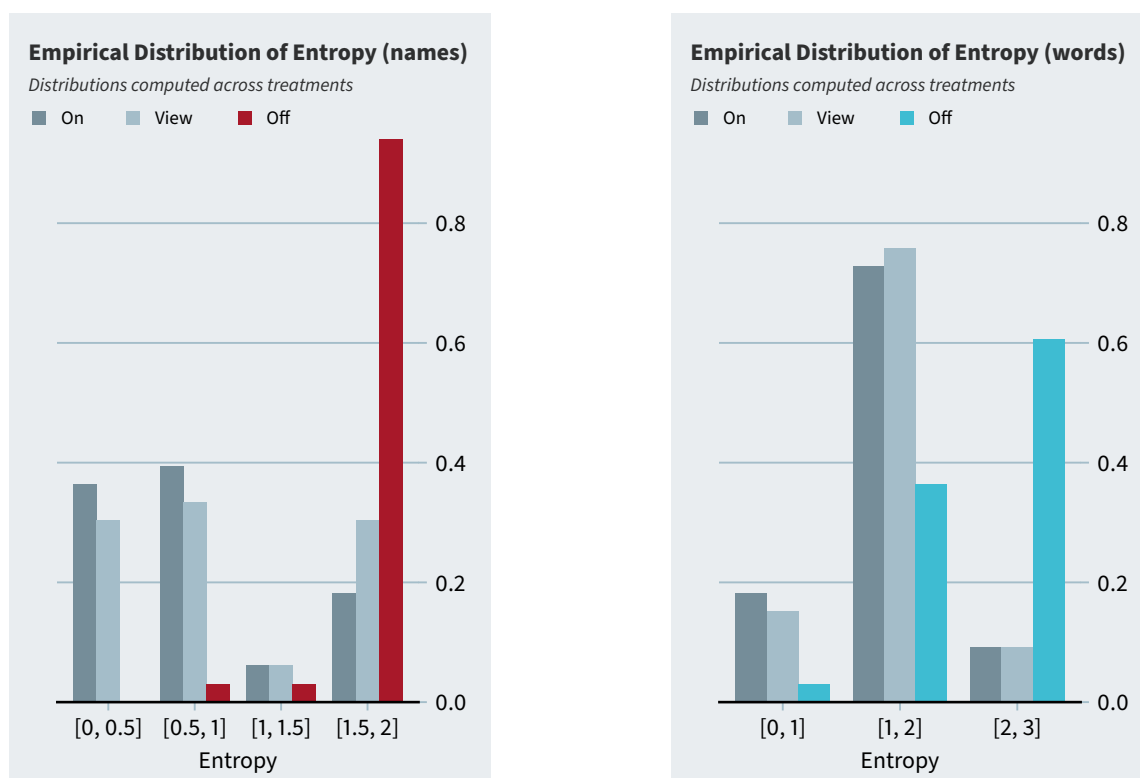


Figure 2 – Treating each $H(c, t)$ as an observation of $H(t)$, these histograms estimate the underlying distribution of $H(t)$ for each treatment t . Left: $H_{names}(t)$ and Right: $H_{words}(t)$ ²

²Bin sizes were chosen for interpretability.

Fig. 3 illustrates the sample mean $\langle H(t) \rangle$ for each treatment $t \in T$.³ Fig. 4 illustrates, for pairs of treatments $(t_1, t_2) \in T \times T$, the pairwise difference $\langle H(t_1) \rangle - \langle H(t_2) \rangle$.

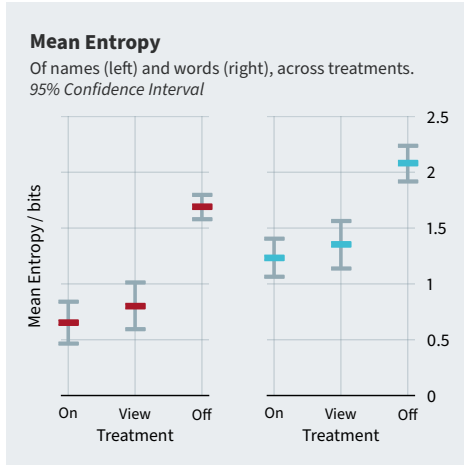


Figure 3 – Mean entropy $\langle H(t) \rangle$ for each treatment t . This is the mean of $H(c, t)$ (Table 2) over all concepts c under the same treatment t . Left: $\langle H_{names}(t) \rangle$. Right: $\langle H_{words}(t) \rangle$.

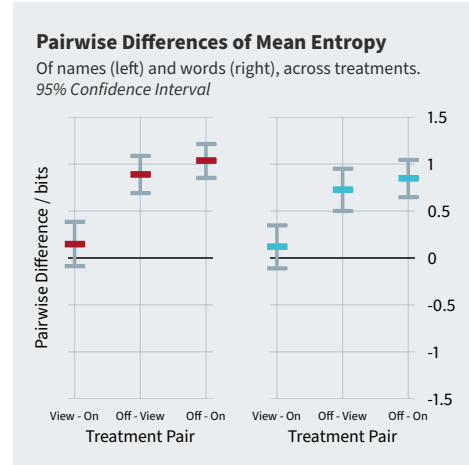


Figure 4 – Difference in mean entropy $\langle H(t_1) \rangle - \langle H(t_2) \rangle$ for pairs of treatments (t_1, t_2) . Left: *name* level. Right: *word* level.

At the name level, the mean entropy was 1.04 bits higher when Copilot was OFF compared to the ON treatment (CI: [0.81, 1.19]), and 0.90 bits higher than the VIEW treatment (CI: [0.69, 1.09]).

Fisher’s Exact Test confirmed this to be a statistically significant difference. Under the null hypothesis

$$\mathcal{H}_0 : \langle H_{names}(OFF) \rangle = \langle H_{names}(VIEW) \rangle$$

\mathcal{H}_0 was rejected at the $p = 0.01$ level ($p = 0.001$).

At the name level, the mean entropy was 0.15 bits higher under the VIEW treatment as compared to the ON treatment (CI: [−0.13, 0.36]). This is not statistically significant at the 0.01 level ($p = 0.18$).

Analysis at the word level revealed similar results. The mean entropy of words was 0.85 bits higher when Copilot was OFF compared to the ON treatment, (CI: [0.65, 1.05]), and 0.73 bits higher than the VIEW treatment. (CI: [0.51, 0.96]). The mean entropy under the VIEW treatment was 0.12 bits higher than under the OFF treatment. Symmetric to the name level results, this difference between ON and OFF was significant at the $p = 0.01$ level ($p = 0.001$) but the difference between VIEW and OFF was not ($p = 0.26$).

Table 3 presents the same data in terms of predictability rather than entropy. Given an arbitrary concept c , Fig. 3 tabulates the probability that c is more unpredictably named under the OFF treatment than the ON and VIEW treatments. More precisely, this is the probability that for some fixed concept c , $H_{names}(c, OFF) > H_{names}(c, t)$, $t \in \{ON, VIEW\}$. If given a random concept c , it is likely that c is more unpredictably named under the OFF treatment than under the ON ($p = 0.91$, CI: [0.82, 1.00]) or VIEW treatments ($p = 0.85$, CI: [0.73, 0.97]).

5.1.1. Probability of Renaming

When Copilot was OFF, participants renamed their “initial” identifier – defined as the first identifier they typed – with probability 0.26. Under the VIEW and OFF treatments, where the “initial identifier” is defined as Copilot’s suggestion, the probability of renaming dropped to 0.20 and 0.11 respectively (Table 4). While the difference in probability between the OFF and ON treatments is significant at the $\alpha = 0.05$ level ($p = 0.04$), the difference between OFF and VIEW treatments is not ($p = 0.06$).

³Mean entropy is higher for **words** than **names** because each name is counted as multiple words. Hence, an outlier name can be counted as three or four outlier words.

t	$P(H(c, \text{OFF}) > H(c, t))$	95% CI
ON	0.909	[0.818, 1.000]
VIEW	0.848	[0.727, 0.970]

Table 3 – Probability that a randomly drawn concept is more predictably named under the ON and VIEW treatment than under the OFF treatment, with 95% CI. (Unpredictability quantified by entropy of the empirical set of names).

t	$P(\text{renamed} t)$	95% CI
ON	0.106	[0.061, 0.160]
VIEW	0.197	[0.129, 0.267]
OFF	0.258	[0.182, 0.333]

Table 4 – Probability that a participant re-named the “initial” name for a concept under treatment t . If $t \in \{\text{ON}, \text{VIEW}\}$, the “initial” name is Copilot’s suggestion. If $t = \text{OFF}$, the “initial” name is the first name typed by participants.

5.1.2. Interfaces vs. Methods

Our experimental tasks required participants to name two distinct types of concepts: interfaces and methods. Fig. 5 compares the distribution of entropy for interfaces, $H_{\text{names}}(i, t)$, with the distribution of entropy for methods, $H_{\text{names}}(m, t)$. Under the ON treatment, the mean entropy of interface names is 0.39 bits higher than for method names. However, this is not statistically significant ($p = 0.06$). Under the VIEW treatment, the difference between the entropy of interfaces and methods is -0.01 bits, and under the OFF treatment, the difference between the entropy of interfaces and methods is 0.08 bits.

6. Discussion

We find that, regardless of the mechanism for accepting suggestions (**RQ2**), names are significantly more predictable in the presence of Copilot suggestions (**RQ1**). Under the ON and VIEW treatments, for more than 67% of concepts, less than 1 bit of information was needed to determine the chosen name. This means that concepts were *more* predictably named than if all participants were given the same two names, and asked to pick one at random. Under the OFF treatment, for more than 90% of concepts, $H_{\text{names}}(c, \text{OFF})$ was greater than 1.5 bits. As only four participants named each concept under each treatment, the maximum possible entropy is 2 bits (4 unique names), i.e., the empirically observed diversity in the OFF condition approaches the theoretical limit.

Our quantitative and qualitative data support the interpretation that participants experienced an attention investment trade-off in identifier naming with Copilot. Three participants indicated that they felt they could have improved on Copilot’s suggestions, but “*it just wasn’t worth the effort*” (P2, P6, P10). This suggests that participants felt that the marginal attentional cost of improving on Copilot’s suggestion was higher than that of improving on one’s own candidate name. This is corroborated by the experimental data, which showed that participants were more than twice as likely to re-name their initially chosen identifiers when Copilot was OFF than the suggested identifier when Copilot was ON.

However, this is not a complete explanation, as the difference in the probability of re-naming under the OFF and VIEW treatments was not significant. This could be attributed to two factors. First, the process of typing out Copilot’s suggestion reduced the marginal cost of thinking up a better name. Second, we underestimated the frequency of renaming under the OFF treatment, by assuming that the first identifier written down by participants was the first candidate name considered. Hence, if participants considered several names before typing out an identifier, which they did not later edit, this was *not* counted as a re-naming. In contrast, under the VIEW and OFF treatments, the frequency of re-naming could be measured much more accurately.

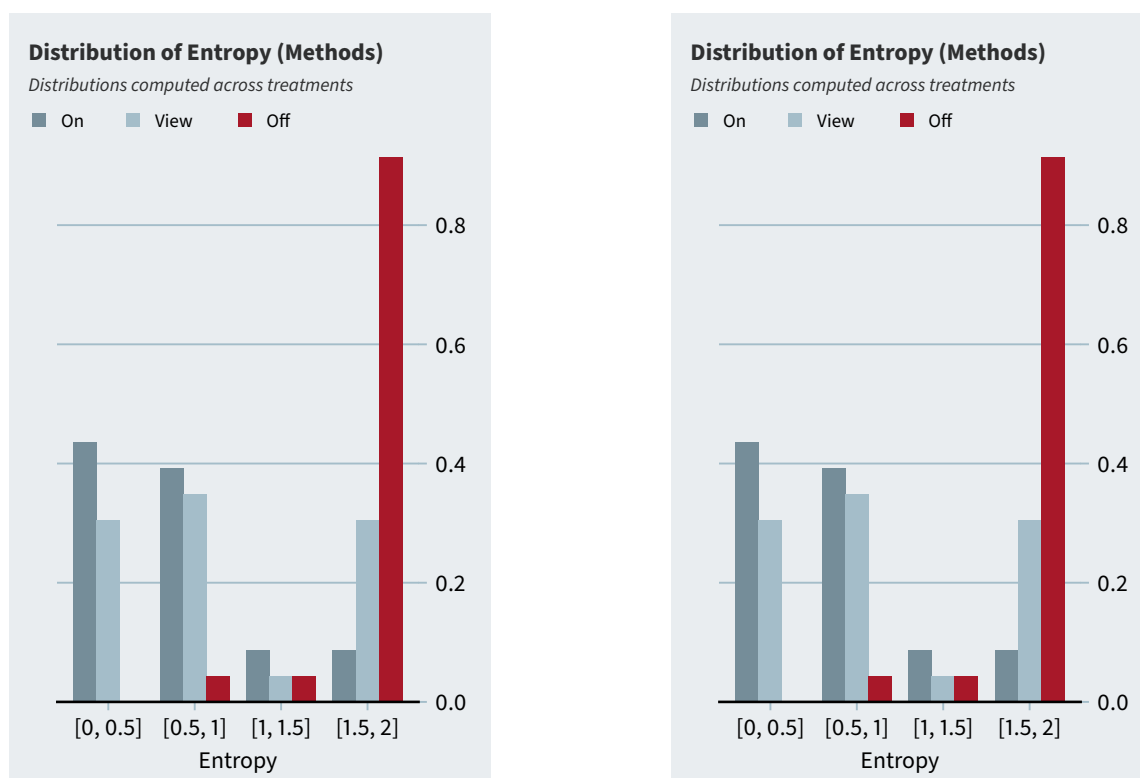


Figure 5 – Left: Distributions of entropy that only consider interfaces and ignore methods: $H_{names}(i,t)$. Right: Distributions of entropy that only consider methods and ignore interfaces: $H_{names}(m,t)$.

6.1. Mixed-initiative systems, agency, and mechanised convergence

These results have implications for our understanding of how contemporary mixed-initiative programming tools can introduce much broader concerns than the traditional narrow focus on task completion. In particular, our findings suggest that there may be implications for mixed-initiative interaction on agency as well as the convergence (homogeneity) of output.

For example, one participant who stated that they “*care a lot*” (P2) about naming noted that programming with Copilot ON was harder than with it OFF, as they felt like they were “*fighting to break free*” of the names suggested by Copilot. Yu et al. (2021) showed that mixed-initiative systems can cause users to feel a loss of agency, which may increase cognitive load. Darvishi et al. (2024) found that AI assistance impacts the agency of students, causing them to rely on rather than learn from AI. While Kalliamvakou (2023) posits that Copilot reduces cognitive load by automating mundane tasks, our results suggest that should developers decide to invest attention into a task, such as choosing a good name, Copilot may decrease feelings of agency and thus increase cognitive load. The perception of agency is an important aspect of the user experience in interacting with intelligent text assistants (Yu et al., 2023), and convergence to Copilot naming might reduce the overall agency and ownership perceived by the programmer. Sarkar (2023a) observes that in generative AI-assisted end-user programming, the traditional attention investment trade-off (between the costs of automation, the time saved, and the risks of failing to build a useful automation) may well be subsumed by considerations of agency and trust in automation.

The second challenge posed by our findings is to the idea that mixed-initiative systems neutrally progress users towards achieving their goals. When the goal is broad and admits a variety of solutions (as in the case of identifier naming), the system may actually influence the goal rather than just infer it. This may or may not be inappropriate – in cases where the programmer should be using a standardised solution or algorithm, but has not recognised this, substitution of a more conventional, predictable, identifier could improve their solution. However, in aspects of software development that relate to contextual and

domain understanding, standardised solutions may be worse.

Consider the mixed-initiative nature of traditional code completion tools (Mărășoiu et al., 2015), and paradigms such as programming by demonstration (Cypher & Halbert, 1993) or programming by example (Lieberman, 2001), and compare their properties to Copilot. Previous work has largely focused on the technical challenge of inferring the user’s goals, over which the user is assumed to have complete autonomy. In contrast, here we observe that the mixed-initiative system is taking an active role in goal-shaping. And the particular form of goal-shaping we have observed in our study corresponds to the phenomenon of *mechanised convergence* (Sarkar, 2023b).

Mechanised convergence is a general principle positing that automation has a standardisation effect, reducing the frequency of outliers. For example, a study of consultants at Boston Consulting Group found that ideas generated with AI assistance had a “*marked reduction in ... variability ... compared to those not using AI. ... it might lead to more homogenized outputs*” (Dell’Acqua et al., 2023). Similarly, Anderson et al. (2024) found that “*different users tended to produce less semantically distinct ideas with ChatGPT*” and further, that this could impact agency: “*ChatGPT users ... felt less responsible for the ideas they generated*”.

In the context of creating identifier names, the principle of mechanised convergence suggests that as names become more predictable, this reduces the frequency of very bad names, but also the frequency of very good ones. One researcher informally analysed the identifiers authored during the study for informativeness. With Copilot OFF, there were more extremely informative, and extremely uninformative identifiers. For example, consider the task where participants were asked to name a character that can move around a grid, one square at a time. With Copilot ON, most participants chose the name `Movable` – this is moderately informative as it states what can be done with the character but contains no information about the one-square constraint. With Copilot OFF, the quality of names ranged from `Move` (very bad) to `TakesSingleStep` (very good). `Move` is uninformative, as the vocative case of the verb “to move” is more appropriate for a function that causes the character to move, and the noun form indicating a specific instance of a motion (i.e., in the sense of “a dance move”) is more appropriate for an object that records a move instance. Both senses of `Move` fail to describe the character’s ability (unlike the adjective `Movable`), and also fail to capture the one-square constraint. On the other hand, `TakesSingleStep` is extremely informative, uses an appropriate grammatical form, and captures the one-square constraint. A full listing of identifiers written by our participants by task and condition is given in Appendix B.

Is mechanised convergence, *per se*, an asset or a liability for the practice of programming? Even if Copilot’s suggested identifiers cannot match the quality or informativeness of those written by the best programmers, they only need to be better than those written by *most* programmers for the aggregate benefits of naming-by-Copilot to outweigh the negatives. However, programmers do not experience the practice of programming in aggregate (Bergström & Blackwell, 2016), and individual programmers almost certainly vary in their naming skill at different times and in different contexts. Moreover, we must also consider not simply the quality of the final identifier, but also the cognitive challenges and benefits of inventing it. The process of naming a concept itself might induce changes or insights. For example, one craft practice of programming holds that if a function is hard to name, this is probably an indication that one is doing too much or too little in that function (Blackwell et al., 2008).

When Copilot suggestions were enabled, suggestions for method names were more readily accepted by participants than suggestions for interface names. Participants occasionally thought of names for interfaces while reading the problem description, *before* seeing Copilot’s suggestion, but this was rare for methods. Even without Copilot suggestions, the predictability of names may vary between concepts. Concepts for which strong conventions exist – for example, getters, setters, and common algorithms like `QuickSort` – might be named more predictably than bespoke methods or interfaces. However, in all treatments the mean entropy for interfaces does not differ significantly from the mean entropy for methods. Hence, there is insufficient evidence to suggest that interfaces are more, or less, predictably

named than methods.

6.2. Implications for design and developer practice

While Github Copilot may boost developer productivity, it also results in significantly more predictable identifiers. This may be because Copilot suggestions increase the attentional costs of improving on a suggested identifier.

These findings offer suggestions for developer workflows that increasingly require “critical integration” (Sarkar, 2023b) of Copilot-generated code.

First, consider settings where good names are costly, but important. For example, when establishing a new set of naming conventions for a codebase. Given that *appropriate* names suggested by Copilot increase the marginal cost of investing attention, the converse might also hold: *inappropriate* suggestions may decrease this cost, and encourage programmers to think more carefully about names, a similar strategy to Wilson et al. (2003)’s *Surprise-Explain-Reward* model, in which the user’s attention is drawn toward features of the code that they didn’t expect.

Second, consider settings where good names are not as critical. For example, when an established convention already exists, and predictable names are informative within the context. In these cases, Copilot may help developers follow existing conventions in predictable ways. In turn, this may help create a setting where unpredictable names draw attention more effectively. When the predictability of a set of names increases, an outlier is more surprising. Hence, when most names are predictable, deliberate breaks from convention can more effectively emphasise subtle differences and direct developers’ attention.

We can also draw on the observations from this empirical study to suggest several design opportunities for mixed-initiative features that could result in improved quality of identifier names.

First, it is important to note that in some cases, the predictable names suggested by Copilot might sometimes be better names than more idiosyncratic alternatives created by the programmer. This may be because the programmer’s suggestion reflects a misunderstanding of the problem, or perhaps a lack of knowledge of standard approaches. In these cases, it could be beneficial to the programmer to invest more attention, thinking again about the reason for their name choice. Wilson et al. (2003)’s *Surprise-Explain-Reward* design pattern can help here, alerting the programmer to the unconventional name they have chosen, and giving them the opportunity to investigate why this is the case.

A second design opportunity could be to optimise investment of attention with better understanding of contextual factors that are relevant to naming, such as distinguishing between a) throw-away programming “sketches” where the code will be discarded immediately after execution; b) systems intended to have a long maintained lifetime that will involve intermittent attention from many different programmers; or c) API libraries and frameworks where thousands of programmers will eventually need to understand the implications of the identifiers chosen. In cases where the choice of identifier names has especially costly implications, a programming assistance tool would be able to take this into account by collecting information about the eventual audience and context of use, encoding that contextual information as additional prompts to the LLM during code generation.

A third design opportunity is to consider a new kind of software development / maintenance tool that might be described as “conceptual refactoring”, which makes no changes to the function or semantics of the source code, but simply modifies identifier names. During incremental and iterative software development, it is not unusual for programmers to improve their understanding of the system such that they see opportunities to improve on the identifier names that were initially chosen. A conceptual refactoring tool, by focusing only on identifier names, could improve the overall clarity and coherence of that name space. Technical strategies that might achieve this through the use of LLMs could include use of summarising approaches to extract and clarify the variety of identifiers that have been used in a large code base, then reconsidering individual identifiers in terms of their role within that overall structure. Such a tool could be implemented as direct interaction with a symbol table or data dictionary, or by

using chat dialog prompts such as “Please include in your response a list of identifiers you’ve used, with the reasons for your choices” (Lewis, 2024).

6.3. Limitations and Future Work

Sample Size. While the results we have observed are statistically significant, it is also possible (given the upper bound on the entropy), that the effect size has been underestimated. Supplemental analyses that increase the sample size might find an even larger effect.

Assumption of Independence. The simplifying assumption that the predictability of an identifier depends only on the treatment, not the concept, may be loosened. While we took preliminary steps in this direction - broadly dividing concepts between interfaces and methods - further work could consider finer granularity in these subdivisions. As suggested, Copilot might have a smaller effect on the predictability of getter and setter names than other methods.

External Validity. The decision to disable IntelliJ IDEA’s code completion tool could be revisited. The presence of IntelliJ IDEA’s code completion tool could have been manipulated as an independent factor, resulting in three more treatments. While this was not feasible due to resource constraints, it represents a natural extension to the study.

Mechanised Convergence. A surface-level survey of the names finds results consistent with the phenomenon of “mechanised convergence”. However, a more thorough analysis requires considering not only changes to the *predictability* of names, but *quality* of names, including cases where the best quality name *would* be a very predictable one (for example when implementing a standard algorithm such as quicksort). Analysis might involve multiple raters ranking names by quality in context, with consistency achieved by employing set-wise comparison (Sarkar et al., 2016).

Sample Homogeneity. We only studied CS undergraduates at one University. Undergraduates are, in general, less than experienced software engineers. They may find it more difficult to choose good names, and be more susceptible to authoritative suggestions. Programmers in industry may be trained to respect certain company-specific conventions, or constraints, in naming. Future work may consider whether the effects generalize to samples of professional programmers.

Language Effects. This study considered the effect of Copilot on identifier names *in Java* specifically. Different programming languages are used by different types of programmer, and for different purposes, which may bring different implications for attention investment. As a result, distinct languages often have distinct conventions for identifiers. Future work might consider if and how the effect on identifier choice varies between languages. For languages with larger training corpora, e.g., popular languages such as Python and JavaScript, identifiers that follow conventions may be assigned higher probabilities by the language model, and so the effect size is unlikely to vary.

7. Conclusion

This study explored how AI code generation tools like GitHub Copilot influence the conceptual task of choosing identifiers during programming. Selecting descriptive names for classes, methods, and variables is a crucial activity that shapes code readability and communicates intent. Yet developers may face tensions between investing sufficient attention for informative naming versus prioritizing efficiency.

We conducted a controlled experiment where 12 participants defined Java interfaces both with and without the presence of Copilot’s identifier suggestions. Across three coding tasks carefully designed to require subjective naming decisions, identifiers chosen under Copilot’s influence were found to have significantly lower entropy – that is, they were more predictable and less informative. Strikingly, this tendency towards predictable names occurred even when Copilot merely displayed suggestions without allowing auto-completion.

We find that generative AI problematizes the traditional task-oriented narrative of mixed-initiative systems. Mixed-initiative systems can have an impact on programmer agency as well as their goals. While predictable names promote consistency, overly deferring to AI suggestions could deprioritise investing

the human attention required to craft identifiers that are specifically tailored to the nuances of the current context and requirements.

To mitigate risks of AI prematurely narrowing programmers' perspectives, we propose AI tools that surface surprising or unconventional alternatives, counterbalancing predictable suggestions. Incorporating conceptual refactoring aids could also encourage revising identifiers as the programmer's understanding evolves.

As AI's role in programming extends beyond just accelerating tasks, this work underscores the need to thoughtfully steer AI-assisted workflows. Simply optimizing for productivity could inadvertently discourage essential cognitive activities that underpin coding quality. Balancing AI assistance with preserving key human skills like intentional naming will be crucial.

Acknowledgments

Thanks to our participants for their valuable time.

References

- Al Madi, N. (2022). How readable is model-generated code? examining readability and visual inspection of github copilot. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–5.
- Anderson, B. R., Shah, J. H., & Kreminski, M. (2024). Homogenization effects of large language models on human creative ideation.
- Ani, Z. C., Hamid, Z. A., & Zhamri, N. N. (2023). The recent trends of research on github copilot: A systematic review. *International Conference on Computing and Informatics*, 355–366.
- Arnold, K. C., Chauncey, K., & Gajos, K. Z. (2020). Predictive text encourages predictable writing. *IUI '20: Proceedings of the 25th International Conference on Intelligent User Interfaces*. <https://doi.org/10.1145/3377325.3377523>
- Barke, S., James, M. B., & Polikarpova, N. (2023). Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 85–111.
- Bergström, I., & Blackwell, A. F. (2016). The practices of programming. *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 190–198.
- Bhat, A., Agashe, S., Oberoi, P., Mohile, N., Jangir, R., & Joshi, A. (2023). Interacting with next-phrase suggestions: How suggestion systems aid and influence the cognitive processes of writing. *IUI '23: Proceedings of the 28th International Conference on Intelligent User Interfaces*. <https://doi.org/10.1145/3581641.3584060>
- Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., & Gazit, I. (2022). Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, 20(6), 35–57.
- Blackwell, A. F. (2002). First steps in programming: A rationale for attention investment models. *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2–10.
- Blackwell, A. F. (2003). First steps in programming: A rationale for attention investment models. *IEEE*. <https://doi.org/10.1109/hcc.2002.1046334>
- Blackwell, A. F. (2022, September). Chapter 10: The craft of coding [<https://moralcodes.pubpub.org/pub/chapter-9>]. In *Moral Codes*. MIT Press.
- Blackwell, A. F. (2023, June). Chapter 11: How can stochastic parrots help us code? [<https://moralcodes.pubpub.org/pub/1osz744d>]. In *Moral Codes*. MIT Press.
- Blackwell, A. F., Church, L., & Green, T. R. (2008). The abstract is an enemy: Alternative perspectives to computational thinking. *PPIG*, 5.
- Buschek, D., Zürn, M., & Eiband, M. (2021). The impact of multiple parallel phrase suggestions on email input and composition behaviour of native and non-native english writers. *CHI '21: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/3411764.3445372>
- Cypher, A., & Halbert, D. C. (1993). *Watch what i do: Programming by demonstration*. MIT press.
- Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. J. (2023). Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203, 111734.
- Darvishi, A., Khosravi, H., Sadiq, S., Gašević, D., & Siemens, G. (2024). Impact of ai assistance on student agency. *Computers Education*, 210, 104967. <https://doi.org/https://doi.org/10.1016/j.compedu.2023.104967>
- Dell'Acqua, F., McFowland III, E., Mollick, E. R., Lifshitz-Assaf, H., Kellogg, K., Rajendran, S., Kraye, L., Candelon, F., & Lakhani, K. R. (2023, September). *Navigating the jagged technological frontier: Field experimental evidence of the effects of ai on knowledge worker productivity and quality* (Working Paper No. 24-013). Harvard Business School Technology Operations Mgt. Unit. <https://doi.org/10.2139/ssrn.4573321>
- Dohmke, T. (2023, June). The economic impact of the ai-powered developer lifecycle and lessons from github copilot - the github blog. <https://github.blog/2023-06-27-the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/>

- Eshraghian, F., Hafezieh, N., Farivar, F., & De Cesare, S. (2023). Dynamics of emotions towards ai-powered technologies: A study of github copilot. *Academy of Management (AOM) Annual Meeting 2023*.
- Fajkovic, E., & Rundberg, E. (2023). The impact of ai-generated code on web development: A comparative study of chatgpt and github copilot.
- Furnas, G. W., Landauer, T. K., Gomez, L. M., & Dumais, S. T. (1987). The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11), 964–971.
- Goodman, N. D., & Frank, M. C. (2016). Pragmatic language interpretation as probabilistic inference. *Trends in Cognitive Sciences*, 20(11), 818–829. <https://doi.org/10.1016/j.tics.2016.08.005>
- Hayes, J. R. (2012). Modeling and remodeling writing. *Written Communication*, 29(3), 369–388. <https://doi.org/10.1177/0741088312451260>
- Horvitz, E. (1999). Principles of mixed-initiative user interfaces. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 159–166.
- Imai, S. (2022). Is github copilot a substitute for human pair-programming? an empirical study. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 319–321.
- Jakesch, M., Bhat, A., Buschek, D., Zalmanson, L., & (2023). Co-writing with opinionated language models affects users' views. *CHI '23: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/3544548.3581196>
- Kalliamvakou, E. (2023, September). Research: Quantifying github copilot's impact on developer productivity and happiness - the github blog. <https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/>
- Koyanagi, K., Wang, D., Noguchi, K., Kondo, M., Serebrenik, A., Kamei, Y., & Ubayashi, N. (2024). Exploring the effect of multiple natural languages on code suggestion using github copilot. *arXiv preprint arXiv:2402.01438*.
- Lewis, C. (2024).
- Lieberman, H. (2001). *Your wish is my command: Programming by example*. Morgan Kaufmann.
- 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. *CHI '23: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/3544548.3580817>
- MacArthur, C. A., Graham, S., & Fitzgerald, J. (2016, October). *Handbook of writing research, second edition*. Guilford Publications.
- Mărășoiu, M., Church, L., & Blackwell, A. F. (2015). An empirical investigation of code completion usage by professional software developers. *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group*.
- Mastrotaolo, A., Pascarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R., & Bavota, G. (2023). On the robustness of code generation techniques: An empirical study on github copilot. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2149–2160.
- McConnell, S. (1993, May). *Code complete: A practical handbook of software construction*. <http://ci.nii.ac.jp/ncid/BA26593422>
- Mozannar, H., Bansal, G., Fourney, A., & Horvitz, E. (2022). Reading between the lines: Modeling user behavior and costs in ai-assisted programming. *arXiv (Cornell University)*. <https://doi.org/10.48550/arxiv.2210.14306>
- Mozannar, H., Bansal, G., Fourney, A., & Horvitz, E. (2023). When to show a suggestion? integrating human feedback in ai-assisted programming. *arXiv (Cornell University)*. <https://doi.org/10.48550/arxiv.2306.04930>
- Naur, P. (1985). Programming as theory building. *Microprocessing and microprogramming*, 15(5), 253–261.
- Nguyen, N., & Nadi, S. (2022). An empirical evaluation of github copilot's code suggestions. *Proceedings of the 19th International Conference on Mining Software Repositories*, 1–5.
- O'Brien, D., Biswas, S., Imtiaz, S., Abdalkareem, R., Shihab, E., & Rajan, H. (2024). Are prompt engineering and todo comments friends or foes? an evaluation on github copilot. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 1003–1003.
- Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of ai on developer productivity: Evidence from github copilot. *arXiv (Cornell University)*. <https://doi.org/10.48550/arxiv.2302.06590>
- Poddar, R., Sinha, R., & Jakesch, M. (2023). Ai writing assistants influence topic choice in self-presentation. *CHI EA '23: Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/3544549.3585893>
- Puryear, B., & Sprint, G. (2022). Github copilot in the classroom: Learning to code with ai assistance. *Journal of Computing Sciences in Colleges*, 38(1), 37–47.
- Sarkar, A. (2023a). Will code remain a relevant user interface for end-user programming with generative ai models? *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 153–167. <https://doi.org/10.1145/3622758.3622882>
- Sarkar, A. (2023b). Exploring perspectives on the impact of artificial intelligence on the creativity of knowledge work: Beyond mechanised plagiarism and stochastic parrots. *CHIWORK '23: Proceedings of the 2nd Annual Meeting of the Symposium on Human-Computer Interaction for Work*. <https://doi.org/10.1145/3596671.3597650>
- Sarkar, A., Gordon, A. D., Negreanu, C., Poelitz, C., Srinivasa Ragavan, S., & Zorn, B. (2022). What is it like to program with artificial intelligence? *Proceedings of the 33rd Annual Conference of the Psychology of Programming Interest Group (PPIG 2022)*.

- Sarkar, A., Morrison, C., Dorn, J. F., Bedi, R., Steinheimer, S., Boisvert, J., Burggraaff, J., D'Souza, M., Kontschieder, P., Bulò, S. R., Walsh, L., Kamm, C. P., Zaykov, Y., Sellen, A., & Lindley, S. (2016). Setwise comparison. *CHI '16: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/2858036.2858199>
- Schankin, A., Berger, A., Holt, D. V., Hofmeister, J. C., Riedel, T., & Beigl, M. (2018). Descriptive compound identifier names improve source code comprehension. *2018 ACM/IEEE 26th International Conference on Program Comprehension*. <https://doi.org/10.1145/3196321.3196332>
- Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27(3), 379–423. <https://doi.org/10.1002/j.1538-7305.1948.tb01338.x>
- Singh, N., Bernal, G., Savchenko, D., & Glassman, E. L. (2023). Where to hide a stolen elephant: Leaps in creative writing with multimodal machine intelligence. *ACM Transactions on Computer-Human Interaction*, 30(5), 1–57. <https://doi.org/10.1145/3511599>
- Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. <https://doi.org/10.1145/3491101.3519665>
- Vasconcelos, M. H., Bansal, G., Fournay, A., Liao, Q. V., & Vaughan, J. (2023). Generation probabilities are not enough: Exploring the effectiveness of uncertainty highlighting in ai-powered code completions. *arXiv (Cornell University)*. <https://doi.org/10.48550/arxiv.2302.07248>
- Wermelinger, M. (2023). Using github copilot to solve simple programming problems, 172–178. <https://doi.org/10.1145/3545945.3569830>
- Williams, J., Negreanu, C., Gordon, A. D., & Sarkar, A. (2020). Understanding and inferring units in spreadsheets. *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 1–9. <https://doi.org/10.1109/VL/HCC50065.2020.9127254>
- Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C. R., Durham, M. D., & Rothermel, G. (2003). Harnessing curiosity to increase correctness in end-user programming. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/642611.642665>
- Yetistiren, B., Ozsoy, I., & Tuzun, E. (2022). Assessing the quality of github copilot's code generation. *Proceedings of the 18th international conference on predictive models and data analytics in software engineering*, 62–71.
- Yetiştirin, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2024). Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778*. 2023. [arXiv preprint arXiv:2304.10778](https://arxiv.org/abs/2304.10778).
- Yu, C. G., Blackwell, A. F., & Cross, I. (2021). Perception of rhythmic agency for conversational labeling. *Human-Computer Interaction*, 38(1), 25–48. <https://doi.org/10.1080/07370024.2021.1877541>
- Yu, C. G., Blackwell, A. F., & Cross, I. (2023). Perception of rhythmic agency for conversational labeling. *Human-Computer Interaction*, 38(1), 25–48.
- Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023a). Demystifying practices, challenges and expected features of using github copilot. *arXiv preprint arXiv:2309.05687*.
- Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023b). Practices and challenges of using github copilot: An empirical study. *arXiv preprint arXiv:2303.08733*.
- Zhao, S. (2023, February). Github copilot now has a better ai model and new capabilities - the github blog. <https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/>
- Zhou, X., Liang, P., Zhang, B., Li, Z., Ahmad, A., Shahin, M., & Waseem, M. (2023). On the concerns of developers when using github copilot. *arXiv preprint arXiv:2311.01020*.
- Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A. S., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2022). Productivity assessment of neural code completion. *MAPS 2022: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. <https://doi.org/10.1145/3520312.3534864>

Appendices

A. Experimental Prompts and Tasks

A.1. Prompts

This experiment is interested in how programmer style is influenced by the use of CoPilot.

It will comprise three tasks. Each task will require you to define one or more interfaces or abstract classes.

Unless explicitly told otherwise, you will not need to implement the interfaces, nor implement any of the methods in the abstract classes. In essence, the focus of task is on the interface/class signatures, not

on the logic.

Each task will be in its own `.java` file, with a comment explaining the nature of the task. Points where you are expected to write code are explicitly flagged with `TODO`.

Each task will have a different CoPilot configuration. There are three configurations:

1. One where CoPilot is turned on,
2. One where you can see CoPilot's suggestions, but you can't automatically accept them (you have to type them out), and
3. one where CoPilot is off.

The researcher will adjust the settings for each of the tasks. Please do not modify them.

When you are done with the task, please indicate to the researcher that you are happy with your submission, at which point the researcher will change the CoPilot settings, and allow you to move to the next task.

If you have any questions, please indicate them to the researcher at this point.

A.2. Tasks

A.2.1. Warmup

```

1 interface TwoDimPoint {
2     int getX();
3     int getY();
4     void setX(int v);
5     void setY(int v);
6 }
7
8 // TODO: Extend this interface to obtain a ThreeDimPoint interface.
```

A.2.2. Task 1

```

1 public class Task1 {
2     // A data pipeline is a series of steps for working with data. This typically
3     // involves reading, extracting, transforming, manipulating, validating,
4     // checking, visualising, storing, plotting, and many other steps.
5
6     // You are tasked with designing three interfaces that, if implemented,
7     // form a very basic pipeline for working with data.
8
9     // First, an interface that is able to ask some source, or knowledge object,
10    // or database for some datum or data. Further, it should be able to take some
11    // datum or data and ask the source/object/database/knowledge base to store it.
12    // When this interface is given a datum or data to store, it should
13    // automatically assume that the datum or data is valid. Further, you can
14    // assume that any singular piece of datum or data will be encoded as a string
15    // .
16    // TODO
17
18    // We have the data from the source, or to be written into the source.
19    // What's next? Well, we want an interface that makes sure we're not doing
20    // anything silly. The interface should have some signal that can call on when
21    // the checks have failed. This ought to be an exception. Now, we assume all
22    // users are internal, so throwing this checked exception shouldn't crash the
23    // program, but instead be caught and handled. This means we want checked
24    // exceptions to be thrown. Your job is to define some checked exception that
25    // is thrown when the second interface catches some mangled or nasty data.
26    // TODO
27
28    // Now we are ready to define the second interface.
```

```

28 // This interface should have two roles.
29 // Its first role is to take a datum or data that the first interface has
30 // returned, and check it for faults.
31 // These flaws could be mistakes, or corruption, or garbled data. While not
32 // important, potential sources of corruption could be faults in the hardware,
33 // problems in transmission over the wire, etcetera. Its second role is that
34 // it should have a method that checks if the data that the user gives to the
35 // first interface is valid. So this treats data going in the other direction.
36 // The causes of the potential mistakes/corruption are different in this case,
37 // as the user could have made a mistake, by encoding some flawed data, or by
38 // calling a buggy encoding method.
39 // TODO
40
41 // Now we are ready to define the third interface.
42 // The final task is to give clients something that they can actually use.
43 // So far we've only been working with data encoded as strings.
44 // Clients don't actually want the strings, but they want something that's
45 // easier to play around with. This involves molding or reshaping the data
46 // into something they can actually use. We don't want to rely on the clients
47 // building proper decoders, so it makes sense to define a couple of defaults
48 // that they can call on to manipulate the data into the form that they
49 // actually plan on using. We've done some studies, and we've narrowed down
50 // the three most commonly used formats to be:
51 //     json,
52 //     byte64, and
53 //     a Map object
54 // Hence, the interface should expose three methods, one for each format.
55 // TODO
56 }

```

A.2.3. Task 2

```

1 public class Task2 {
2     // I am building a board game that will be populated by characters.
3
4     // First, it's important that some characters can move around the board.
5     // Different characters can move in different ways, following different
6     // constraints. The interface should describe characters that can move
7     // one square up, one square down, one square left, and one square right.
8     // These characters aren't allowed to move multiple squares at once, or
9     // diagonally.
10    // TODO
11
12    // Sometimes we care about allowing the characters to move, and other
13    // times we care about allowing the characters to perform more complex
14    // motions, like spinning. In order to build something like spinning, we
15    // need to allow characters to rotate. Here, we only care about characters
16    // that can turn left with respect to the direction that they are facing,
17    // and right with respect to the direction that they are facing. The arc
18    // of rotation should be a quarter-circle, that is, increments of 90 degrees,
19    // or pi/2 radians.
20    // TODO
21
22    // Third, some characters will have an inventory.
23    // An inventory is a collection of items that a character can carry around
24    // with them. We care about characters that are able to take a single item
25    // from their surroundings, and put it into their inventory. After they do
26    // so, they will be able to carry these items around with them.
27    // Characters should also be able to retrieve stuff from the inventory, to
28    // use it or discard it in some manner. Characters should operate on singular
29    // items, that is, there won't be a method to put many things into the
30    // inventory, or take many things out of it, in just a single go.
31    // TODO
32

```

```

33 // Fourth, some characters will be able to throw things.
34 // When a character throws something, they will throw it in the direction
35 // that they are facing, rather than in any arbitrary direction. This means
36 // that they can only throw things in the direction they are facing. In
37 // addition, characters need to throw something. They can't just throw nothing.
38 // Further, characters that implement this interface should throw things
39 // exactly 5 squares. To recap: characters can throw things 5 squares in the
40 // direction that they are facing.
41 // TODO
42
43 // As a test, build a Character interface that extends each of the previous
44 // interfaces
45 // TODO
46
47 // Build a dodge method that moves a character. Assume that the character is
48 // facing the up direction. You want to move it in a zig-zag motion in the up
49 // direction. The character should turn to face the direction of motion and
50 // moving in some interesting pattern. The signature of the method should be
51 // void dodge(Character c)
52 // TODO
53
54 // Build an attack method: get a stone from the inventory, throw it, walk 5
55 // steps, and get the stone back. The signature of the method should be
56 // void attack(Character c)
57 // TODO
58
59 // Finally, build an interface for characters that can move 5 squares up,
60 // down, left, and right. They shouldn't be able to move in increments of
61 // less than 5.
62 // TODO
63 }

```

A.2.4. Task 3

```

1 public class Task3 {
2     /*
3     We have a user-facing command line system.
4     Users can personalise the system, for example, changing the time zone,
5     language, and font size.
6
7     There are also some global settings, or defaults. These global settings /
8     defaults kick in
9     when the user has not specified any personal settings. For example, when the
10    user is a new user.
11    In some sense these settings are pre-installed or pre-defined by the company,
12    though once the
13    software has shipped, users (be they people or companies) can change these
14    global or default settings
15    to their liking.
16
17    Currently, the mechanism for changing settings, both global and local, is by
18    setting feature flags.
19    A feature flag is a single bit, indicating if the feature is on, or off.
20    The code will query these feature flags to determine methods to execute, or
21    items to display.
22    This means that the system will operate differently depending on if a feature
23    flag is on, or off.
24    Specifically, this is done via a command line tool, called, flg. There are 5
25    ways to use flg
26
27    flg 0 is a getter, that returns the user's custom settings. For example, if
28    the user settings are
29    "1011", then flg 0 will return "1011" for that user. Different users will get
30    different results.

```

```

20
21     flg 0[sequence] acts as a setter for the user's private, personal, custom
settings.
22     The sequence is used to determine what settings ought to be set.
23     For example, flg 011001 turns feature flag 0 on, feature flag 1 on, feature
flag 2 off, feature flag 3 off,
24     and feature flag 4 on.
25
26     flg 1 is a getter, and it gets the global or universal settings. For example,
if the global settings are 0000, then
27     flg 1 will return 0000.
28
29     flg 1[sequence] changes the global settings (for everyone). The sequence is
used to determine what the new
30     global settings ought to be. For example, flg 10111 will turn feature flag 1,
feature flag 2, and feature flag 3 on,
31     and it will turn feature flag 0 off. It does not delete any custom settings.
That is, if the user already has
32     feature flag 0 on, they will not notice a change.
33
34     flg 2 deletes all custom settings, effectively resets all custom settings to
the global default. So for example,
35     if there are 3 users, users A, B, and C, and they each have custom settings,
and the global setting is 0000, then
36     after we call flg 2, all 3 users will have 0000 as their settings.
37
38     The company has decided to move to a system with a GUI.
39     You have been asked to refactor flg into an abstract class (no implementation
required, all methods should be stubs).
40     This class will not be exposed to the user via a command line interface.
41     This means there is no need for backwards compatibility, and you only need to
preserve the functionality, not
42     the exact syntax, or the exact mechanisms that supply the functionality.
43     Indeed, you have been advised to define one method for each possible different
way to use flg.
44     Your abstract class should store the global settings as a static list of
integers.
45     You abstract class should also store the user's custom settings as a static
dictionary from user ID (string) to a list of integers.
46     */
47     // TODO
48
49 }

```

B. Participant Responses

Tasks are named Tn::Im::Mk, as in Task n, Java Interface m, Java Method k.

	ON	VIEW	OFF
T1::I1	DataSource DataSource DataSource DataSource	DataSource DataStore DataSource DataSource	ManipulateData GetAndSettable SourceQuery Datum
T1::I1::M1	query read read read	read getData read read	getData get retrieve getDatum
T1::I1::M2	store write write write	store storeData store write	storeData set store storeDatum
T1::I2::E1	DataValidationException DataException	DataIntegrityException DataHandlerException	CheckedException SillyData

Continued on next page

	ON	VIEW	OFF
	DataException DataException	DataException DataException	SourceException MangledDataException
T1::I2::E2	DataValidationException DataException DataException DataException	DataConsistencyException DataHandlerException DataException DataException	CheckedException SillyData SourceException MangledDataException
T1::I3	Validator DataChecker DataChecker DataChecker	DataChecker DataHandler DataValidator DataChecker	CheckData DataChecker SourceVerifier DataChecker
T1::I3::M1	validate checkRead checkRead checkRead	checkRead checkDataReturn sourceValidate checkRead	checkDataOutput checkRetrievedData checkForFaults checkReturnedData
T1::I3::M2	validate checkWrite checkWrite checkWrite	checkWrite checkDataInput userValidate checkWrite	checkDataInput checkInputData checkForFaults checkInputData
T1::I4	Decoder DataTransformer DataTransformer DataTransformer	DataTransformer DataDecoder DataDecoder DataTransformer	convertData Decoder Decoder DataManipulator
T1::I4::M1	decodeJson json json json	toJson asJson json json	jsonData toJson toJson dataToJson
T1::I4::M2	decodeBase64 byte64 byte64 byte64	toBase64 asByte64 byte64 byte64	byte64Data toBase64 toByte64 dataToByte64
T1::I4::M3	decodeMap map map map	toMap asMap map map	mapData toMap toMap dataToMap
T2::I1	Movable Movable Movable Movable	TakesSingleStep CharacterTranslations CharacterMove Move	Movable Movable MovableCharacter MovingCharacter
T2::I1::M1	moveUp moveUp moveUp moveUp	movesOneSquareUp moveUp moveUp up	moveUp moveUp moveUp moveUp
T2::I1::M2	moveDown moveDown moveDown moveDown	movesOneSquareDown moveDown moveDown down	moveDown moveDown moveDown moveDown
T2::I1::M3	moveLeft moveLeft moveLeft moveLeft	movesOneSquareLeft moveLeft moveLeft left	moveLeft moveLeft moveLeft moveLeft
T2::I1::M4	moveRight moveRight moveRight moveRight	movesOneSquareRight moveRight moveRight right	moveRight moveRight moveRight moveRight
T2::I2	Rotatable Rotatable Rotatable Rotatable	AbleToRotate CharacterRotations CharacterRotate Rotate	Rotatable Rotatable RotatableCharacter SpinningCharacter
T2::I2::M1	rotateClockwise rotateLeft rotateLeft	turnsLeft90Degrees rotateLeft rotateCW	rotateLeft rotateLeft rotateLeft

Continued on next page

	ON	VIEW	OFF
	rotateLeft	rotateLeft	rotateLeft
T2::I2::M2	rotateAntiClockwise rotateRight rotateRight rotateRight	turnsRight90Degrees rotateRight rotateACW rotateRight	rotateRight rotateRight rotateRight rotateRight
T2::I3	Inventory Inventory Inventory Inventory	HasInventory CharacterInventory CharacterInventory Inventory	Inventory Inventory CharacterWithInventory InventoryCharacter
T2::I3::M1	storeItem takeItem takeItem storeItem	storeItem store store put	takeItem takeItem takeItem takeItem
T2::I3::M2	retrieveItem retrieveItem retrieveItem retrieveItem	retrieveItem retrieve retrieve retrieve	retrieveItem retrieveItem retrieveItem retrieveItem
T2::I4	Throwable Thrower Thrower Thrower	AbleToThrow CharacterThrow CharacterThrow Throw	Thrower Thrower CharacterThatThrows ThrowingCharacter
T2::I4::M1	throwItem throwItem throwItem throwItem	throwForwardFiveSquares throwFive throwFive throwItem	throwItem throwItem throwItemFiveSquares throwItem
T3::I1	FeatureFlags FeatureFlag Flg Flg	Flg Flg Flg Flg	Flg Settings Flg Flg
T3::I1::A1	defaultFlags global globalFlags globalSettings	globalSettings globalSettings globalSettings globalSettings	globalSettings globalSettings globalSettings globalSettings
T3::I1::A2	userFlags custom userFlags custom	customSettings userSettings customSettings customSettings	customSettings customSettings userSettings customSettings
T3::I1::M1	setDefault getGlobal flg1 flg0Get	getGlobalSettings getGlobalSettings getCustomSettings getCustomSettings	getGlobalSettings getGlobalSettings getUserSettings getGlobalSettings
T3::I1::M2	setDefault setGlobal flg1 flg0Set	setGlobalSettings setGlobalSettings setCustomSettings setCustomSettings	setGlobalSettings setGlobalSettings setUserSettings setGlobalSettings
T3::I1::M3	setUser getCustom flg0 flg1Get	getCustomSettings getUserSettings getGlobalSettings getGlobalSettings	getCustomSettings getCustomSettings getGlobalSettings getCustomSettings
T3::I1::M4	getUser setCustom flg0 flg1Set	setCustomSettings setUserSettings setGlobalSettings setGlobalSettings	setCustomSettings setCustomSettings setGlobalSettings setCustomSettings
T3::I1::M5	clear deleteAllCustom flg2 flg2Del	reset reset reset reset	reset resetToGlobalSettings resetUserSettings deleteCustomSettings

Table 5 – Participant Responses

Further Evaluations of a Didactic CPU Visual Simulator (CPUVSIM)

Renato Cortinovis
Freelance Researcher
Italy
rmcortinovis@gmail.com

Tamer Mohamed Abdellatif
Canadian University
Dubai, United Arab
Emirates
tamer.mohamed@cu.d.ac.ae

Devender Goyal
Raytheon
Technologies,
USA
dg1998@gmail.com

Luiz Fernando Capretz
Western University
Canada
lcapretz@uwo.ca

Abstract

This paper discusses further evaluations of the educational effectiveness of an existing CPU visual simulator (CPUVSIM). The CPUVSIM, as an Open Educational Resource, has been iteratively improved over a number of years following an Open Pedagogy approach, and was designed to enhance novices' understanding of computer operation and mapping from high-level code to assembly language. The literature reports previous evaluations of the simulator, at K12 and undergraduate level, conducted from the perspectives of both developers and students, albeit with a limited sample size and primarily through qualitative methods. This paper describes additional evaluation activities designed to provide a more comprehensive assessment, across diverse educational settings: an action research pilot study recently carried out in Singapore and the planning of a more quantitative-oriented study in Dubai, with a larger sample size. Results from the pilot study in Singapore confirm the effectiveness and high level of appreciation of the tool, alongside a few identified challenges, which inform the planning of the more comprehensive evaluation in Dubai.

1. Introduction

Numerous CPU visual simulators have emerged over time with the primary objective of enhancing the understanding of computer operation (Nikolic et al., 2009). These simulators cater to various levels of expertise and often specialize in specific facets of computer science, such as computer security (Imai et al., 2013) or pipelining (Zhang and Adams III, 1997). Among this diverse array of simulators, a small subset addresses a well-recognized issue: that students – despite studying both high-level programming languages and computer architecture fundamentals – frequently struggle to grasp how high-level code actually executes on computer hardware (Evangelidis et al., 2021; Miura et al., 2003). These concepts are considered fundamental in computer science and software engineering education and training: the Software Engineering Body of Knowledge (Bourque and Fairley, 2022), for example, reports that “software engineers are expected to know how high-level programming languages are translated into machine languages”.

In this context, the CPUVSIM (Cortinovis, 2021) – an Open Educational Resource available from Merlot and OER Commons – supports novices in comprehending the fundamental components of a simplified CPU, and in understanding the mapping from high-level control structures to low-level code, i.e. assembly and machine code. This is achieved through detailed animations that illustrate the execution of instructions, and empowering learners to write meaningful programs using a minimalist yet representative assembly language.

Figure 1 shows a screenshot of the CPUVSIM running in a browser, with a simple program loaded in RAM. The user can execute the program one instruction or micro-instruction at a time, at the desired speed. The user can interactively modify, at any time, the content of the RAM, or any register in the CPU. While the execution is animated, a voice over explains what is happening – in English, Spanish, or Italian.

As detailed by Cortinovis (2021), the development of CPUVSIM sought to address limitations observed in existing applications. While some simulators, such as LMC (Higginson, 2014), were deemed overly simplistic, others were considered unnecessarily complex. CPUVSIM, on the other hand, was honed through iterative improvements and extensions, building upon the foundation of an already popular visual simulator known as PIPPIN (Decker and Hirshfield, 1998). Its development process followed a

sustainable Open Pedagogy approach (Wiley and Hilton, 2018) in the form of non-disposable assignments to computer science students over multiple years.

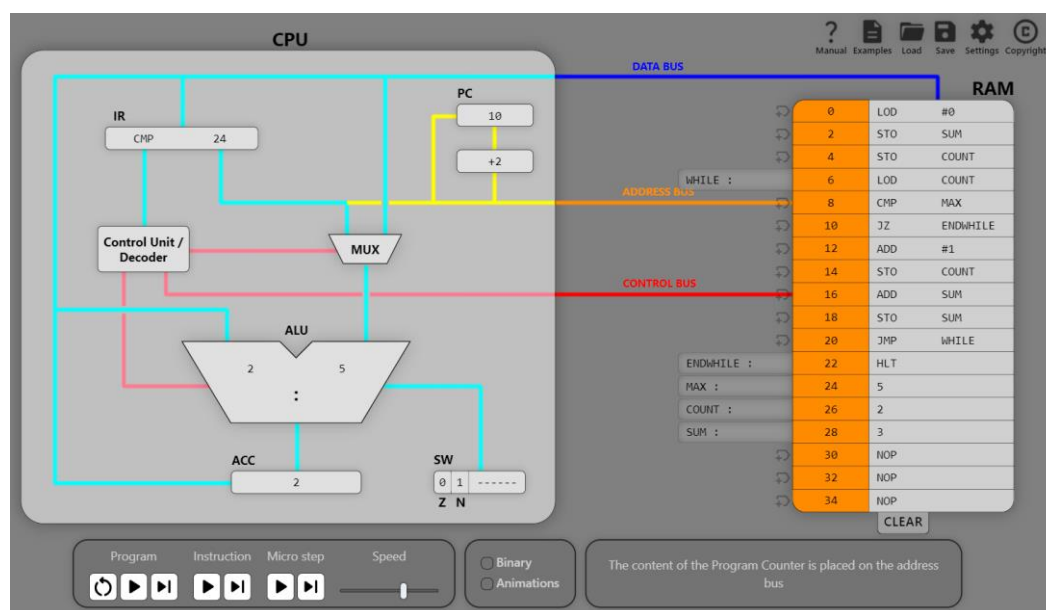


Figure 1 - A screenshot of the CPUVSIM

The CPUVSIM has previously undergone evaluations of its educational effectiveness, albeit with limited sample size and primarily through qualitative methods. These evaluations were conducted from the perspectives of both its developers and students who engaged with it in different contexts. In this paper, we describe an evaluation recently conducted in Singapore and also describe an evaluation planned to be conducted in Dubai. These studies outline additional evaluation activities designed to provide a more comprehensive assessment of the CPUVSIM. Our efforts encompass diverse settings, employing as far as possible complementary approaches.

Following the general introduction in this section, Section 2 reports on the evaluation of CPU simulators in the literature, Section 3 describes our Action Research pilot evaluation recently carried out in Singapore and its findings. Section 4 describes the planning of the evaluation to be conducted at a University in Dubai with a larger number of students, utilizing again Action Research but complemented with elements of a quantitative-oriented experimental approach. Finally, Section 5 presents our conclusions.

2. CPU simulators evaluation strategies

As mentioned earlier, the CPUVSIM has already undergone some limited and mainly qualitative evaluations. Cortinovis (2021) describes its informal qualitative evaluation from the developers' point of view, who deeply appreciated, in particular, the opportunity to work on a real problem in a real context, and the opportunity to contribute to the common good. Cortinovis and Rajan (2022) describe the evaluation from the students' point of view, both in two specialized technical schools in Italy (K12 and lifelong adult education) and in a first and a second-year undergraduate computer architecture courses in Colorado (USA). The students who used the simulator provided very positive feedback, which was analysed with a qualitative thematic content analysis, and was then used to further improve and extend the latest version of the simulator.

Nikolic et al. (2009), evaluate a rich set of existing CPU simulators, but only on the basis of characteristics identified from the documentation. Some of the criteria they used, such as level of coverage of the topics included in standard curricula, are not considered fully relevant in this context: the CPUVSIM is meant to support a firm grasp of the fundamental mechanisms, but at a relatively high level of abstraction, without dwelling too much in details.

Imai et al. (2013) evaluated the correlation between tests carried out on their simulator and the course final exam results. The strong correlation they reported is interesting, but to demonstrate the

effectiveness of their simulator, it is necessary to compare it with an alternative tool or a comparable teaching method. Indeed, in subsequent works (Imai et al., 2018), they adopted a qualitative approach alone with a simple questionnaire.

Chalk (2002) and Mustafa (2010) both used a mixed qualitative and quantitative strategy. The quantitative approach, in particular, makes use of a quasi-experimental schema, with experimental and control groups, and pre and post knowledge tests. Although the experimental strategy is instrumental in collecting supporting evidence about the effectiveness of the tool, and while pre- and post- knowledge tests can demonstrate the improvements of students' knowledge using the simulator, this approach does not provide information about its effectiveness against alternative strategies. Chalk (2002) demonstrates, in particular, the importance of referring to precisely formulated learning objectives, to test results against.

3. CPUVSIM pilot evaluation in Singapore

We planned and executed a first pilot evaluation of the simulator, on a small scale (13 students in total), in two undergraduate courses at the Yale-NUS College in Singapore: a course on C Programming and a course on Software Verification and Validation. The simulator was used in the first course to help students understand the mapping between C control structures and assembly code. It was used in the second course to test programs at machine language level. Considering the limited number of students, and the limited possibilities to control the many factors involved (different classes, different teachers, etc.), we considered it appropriate to adopt a socially-oriented, situational Action Research methodology, preferring a more postmodernist-oriented approach over a strictly positivist one (Kemmis and McTaggart, 2000).

Given the overall goal of grasping how code written in high-level language is actually executed on the hardware of a computer, we outlined first, as recommended by Chalk (2002), the learning objectives:

- Understand the role of the key components of a CPU.
- Understand the mapping from high-level to low-level control structures (assembly and machine) code.
- Code meaningful high-level programs with a minimalist but representative assembly language.

More specifically:

- Describe typical assembly instructions supported by a CPU.
- Explain the fundamental steps carried out by the main subcomponents of a CPU, to execute a given assembly instruction.
- Identify the information transferred on the Data bus, Address bus, and Control bus during each step of every instruction.
- Apply the suitable numeric/immediate and direct addressing modalities.
- Exemplify the use of the CPU flags through simple examples.
- Translate a program in C with a single control structure to assembly code.

According to the adopted research methodology, we defined an action plan for the proposed intervention, including specific pedagogical activities as well as “Data analysis and critical reflection”, and “Refinement of the planned intervention for future courses”.

In particular, we foresaw a first activity to present in class the CPUVSIM and its associated e-book (1.5 hours), follow-up students' activities to be started in class and completed at home (a couple of hours to familiarize individually with the CPUVSIM and related educational material), plus an additional hour to complete the graded activities. These included:

Briefly explain the differences between conditional and non-conditional jump.

Briefly list/describe the steps carried out by the main sub-components of a CPU, to execute the instructions ADD #20 and ADD 20 (immediate and direct addressing).

Identify the missing instruction in the following translation of an IF-THEN-ELSE control structure to assembly:

IF SUM == 2 THEN SUM=3 ELSE SUM=5 ENDIF	LOD SUM CMP #2 JNZ ELSE LOD #3 // MISSING CODE? ELSE: LOD #5 ENDIF: STO SUM HALT SUM: 0
--	---

The second course on software verification and validation included the following additional assignments:

Use the Simulator to test if the translation of the following high-level control structures to Assembler are correct or not [...]; explain your answer.

Discuss Specific Testing Strategies for Assembly code.

We finally specified a survey with Likert type questions and open questions (Mustafa, 2010; Cortinovis and Rajan, 2022), to collect feedback about the CPUVSIM and the learning experience, such as:

What ameliorations could be made to the simulator and/or related e-book to improve your learning experience?

The final assignment was graded and analysed with psychometric Classical Test Theory (Novick, 1996). Taking into account the limited number of students, the Likert-type questions in the survey were analysed with basic descriptive statistics (Mustafa, 2010), the open questions in the survey were analysed with thematic content analysis (Cortinovis and Rajan, 2022).

Finally, we carried out a critical reflection on the effectiveness of the intervention to derive the recommendations for planning the subsequent intervention – according to the iterative nature of Action Research, and to the goal of a pilot.

3.1 Pilot Results

The data extracted from the survey (Table 1) on 13 students shows that the CPUVSIM was definitely appreciated, especially for understanding how C control structures actually get executed on a computer, which was the main goal.

Questions	Strongly Agree + Agree (%)	Strongly Disagree + Disagree (%)	Neutral (%)
The simulator and related e-book were motivating and interesting.	77% (10)	0%	23% (3)
The simulator and related e-book were useful for understanding how C control structures actually get executed on a computer.	85% (11)	0%	15% (2)
I found the simulator too complicated to understand and use effectively.	23% (3)	46% (6)	31% (4)

Table 1 – Sample extracted from the survey in Singapore.

Interestingly, a relevant number (23%) of students stated that the simulator was not easy to understand and use effectively: this was probably due to the limited time devoted to its presentation (just 1.5 hours in total). Indeed, a first student who found the simulator too complicated suggested having “more hand holding in class”; a second student considered that “the lecture included too much”. A student found that the simulator was too fast: “It was challenging keeping up with its fast-pace while still understanding newly introduced concepts”. Obviously, this student did not notice the possibility to control the speed, which was explicitly appreciated by other students.

Despite these problems, the students’ answers on the assignments demonstrated a remarkable grasp of the targeted key concepts. There were no incorrect solutions to the assignments, even if there were omissions of relevant details in a few of them – notably from a student who found the use of the simulator somewhat complicated.

These overall positive outcomes were strongly correlated with the students' self-perceptions: one of them, for example, stated that she reached “a solid understanding of how high-level code runs on hardware”, while another one found it “really eye-opening to see how it actually works at the base level”. The number of students involved in this pilot was limited, yet the evaluation results confirm previous results available in the literature (Cortinovis and Rajan, 2022). The main lesson learned for future deliveries of the course, is the need to dedicate more time to coaching the students in the use of the simulator, so that all of them can get the most from it.

4. CPUVSIM planned evaluation in Dubai

In Dubai we aim to improve the previous evaluations addressing two potentially weak aspects of action research: generalizability and rigor. Concerning generalizability, we are evaluating the simulator in different contexts, that is, different courses, multiple classes, and different countries. To improve the rigor of the evaluation process, we take advantage of the larger sample size available (120+ students), enriching the qualitative-oriented action research design used in Singapore, with a quantitative-oriented experimental approach.

Drawing lessons from the pilot evaluation conducted in Singapore, we will allocate additional time to ensure that every student gains complete mastery over the utilization of CPUVSIM and its accompanying documentation. First, we will dedicate a decent time to our lab instructors to train on and master the simulator. This is planned to take place during the pre-semester preparation period of two weeks. During this period, the course instructors, with the support of the lab instructors, will work on integrating the simulator within the course syllabi and preparing the simulator-based assessment tasks. Accordingly, full two lab sessions (2 hours each) will be dedicated to the students' training on the simulator. The first lab session (2 hours) will be dedicated to introducing the simulator's built-in CPU instructions in addition to the education supporting features, such as the instructions execution simulator and the simulator e-book. After finishing this lab session, the students will be left with a simulator-based homework. In the second lab session (2 hours), the students will be provided with time dedicated to homework discussion and one-by-one coaching on the answer of each of the homework tasks using the simulator. This way, the students will acquire a fair understanding level of the simulator's functionality before proceeding with more challenging tasks.

Furthermore, we will include the following additional learning objective:

- Translate a program in a high-level language with multiple control structures, both sequenced and nested, to assembly code.

Therefore, beyond the exercises proposed in Singapore, we will conduct a more comprehensive assessment of students' proficiency in translating high-level constructs, such as loops, logic operators, and arithmetic operations, into assembly language. For example:

- write an assembly program that determines whether the value stored in a variable “var1” is odd or even;
- write an assembly program that performs a comparison between two signed variables. Var1=7Fh and Var2=80h. Then saves the highest and the lowest variables in HIGH and LOW variables respectively;
- write an assembly program that determines whether a given positive integer number satisfies the Collatz conjecture.

4.1 Experimental Design and Population

We have a total of 120+ students that will be partitioned into two groups. The first one will follow the traditional educational path of the previous years, which did not use a simulator but was based on theoretical lessons and paper-based exercises. The second one will follow an educational path modified with the new intervention, using the CPUVSIM. The students, who have Python and Java programming background, will have different lab instructors and teaching assistants.

4.2 Quantitative Data Collection and Analysis

We will gather quantitative data through the following assessment setups:

- Exercises under an invigilated environment: for this setup, students will be asked to answer the assessments at the lab using our learning management system (LMS). The students will have no access to the internet. The assessment will have a specified time limit and the LMS system will record the time taken by the student to complete each task (completing the task in less time will lead to collecting more marks). We plan to conduct this assessment setup twice within the course timeline.
- Group-based exercises carried out in the lab: this kind of assessment will allow the students in small sub-groups (3-4 students) to hone their assembly level programming and benefit from each other. In case of any needed support from the instructor, the students will be asked to submit their inquiries via the LMS system.
- Written exams: this includes both graded mid-term and final exams. Both students' grades and specific mistakes will be gathered for this kind of assessment setup.

Therefore, in addition to the students' grades, we will gather other quantitative data such as the time needed to finish the assessment, number of students' mistakes, types/categories of the students' mistakes (for instance, whether due to incorrect understanding of the logic of jump instructions or to incorrect understanding of the mapping between high level programming and assembly code logic), and the Grade Point Average of each group member. Furthermore, each assessment's activity/question will be mapped to a certain learning objective. This allows us to evaluate the effectiveness of using the simulator on improving the students' knowledge for each of our learning objectives, in addition to the overall impact on the students' performance along course(s).

We will adopt ANOVA variance analysis statistical test to verify whether the results for the two main student groups show an overall statistically significant difference based on the use of CPUVSIM according to the sample size at hand. In addition, we plan to apply the Tukey's Honestly Significant Difference (HSD) statistical test to figure out which group of data parameters is impacted the most by using the CPUVSIM based on the sample at hand. Examples of data parameters that can be studied by HSD are: time needed to finish the task, number of mistakes.

4.3 Qualitative Data Collection and Analysis

Qualitative data will be collected using a survey with both Likert-type questions and open questions similar to the ones adopted in the pilot study. Here, however, we plan to extend our qualitative data by collecting feedback from teachers too. Therefore, the survey questions for the teachers will include, for example: How does the CPUVSIM impact on the students' understanding of the assembly language structure? How does it impact related explanations? Would you recommend making use of the CPUVSIM, and how?

We plan to interview all the course instructors as well as one student from each student-focused sub-group. To analyse these qualitative data, both thematic and narrative analysis will be adopted. Thematic analysis will help in identifying and interpreting the patterns from our survey results, while narrative analysis will provide a better understanding of the motivation behind the feedback provided by the interviewees.

Finally, we will also run a longitudinal study, because we suspect that the students with hands-on experience with the simulator might better retain over time the competences acquired, compared to the students who followed the more traditional path. Therefore, we will retest the students of both groups after 12 months, to assess the possible different levels of retention of key concepts and competencies.

4.4 Threats to Validity

Generalizability and rigor are the two main weak aspects of the situational nature of action research. Yet, the variety of contexts where these evaluations are carried out (Singapore) and planned (Dubai), in addition to the evaluations previously reported in the literature (Italy and USA), should contribute to support the generalization of the outcomes. Additionally, we integrated in the methodology for the planned evaluation in Dubai, a quantitative-oriented experimental component to improve rigor: the use of complementary research methodologies, selected to better fit the particular contexts, should help compensate for their weaknesses.

More importantly, we acknowledge our limited control over numerous variables, particularly the inevitably diverse approaches employed by various educators when utilizing the tool. The challenge lies in discerning the tool's impact amidst the multitude of factors influencing student learning. It is imperative to recognize the tool as just a component within a broader socio-technical system, as highlighted by Mulholland (personal communication, 2023). To tackle this challenge, our evaluation strategy in Dubai includes qualitative assessments from educators' perspectives. These assessments aim to glean insights into how teachers perceive the CPUVSIM and its effects on their educational endeavours. Additionally, we leverage the recently developed CPUVSIM accompanying e-book titled "A Gentle Introduction to the Central Processing Unit (CPU) and Assembly Language". This Open Educational Resource, available via Merlot or OER Commons, offers some pedagogical support for utilizing the CPUVSIM. It provides explanations and practical activities to support both teachers as well as self-learners. Notably, this interactive e-book integrates the CPUVSIM seamlessly. Each programming example or exercise within the e-book features live images, embedding the fully functional CPUVSIM. Users can execute, modify, and re-execute these "images" at will, enhancing the interactive learning experience.

4.5 Ethical Considerations

In our plan, only one of the two groups of students will have the opportunity to reap the potential advantages of using the CPUVSIM, which can be questioned from the ethical point of view. To overcome this problem, in case the evaluation would show that a group was considerably disadvantaged compared to the other, we plan to offer, at the end of the evaluation, some extra educational activities to level their competences. These extra activities would target specifically the topics where the evaluation might have identified significant differences.

5. Conclusions

We have outlined our plans for assessing the educational effectiveness of the CPUVSIM simulator. The feedback from the pilot in Singapore, using Action Research, confirms a positive effect on students' ability to grasp important concepts and good appreciation for the tool, as reported in the literature, together with the identification of some challenges. This provided useful indications for the wider evaluation planned in Dubai, where we will enrich the qualitative Action Research methodology with a more quantitative-oriented study, aiming to address concerns about generalizability and rigor.

Through these activities, we are collecting feedback from students and teachers in diverse geographical regions, broadening the perspective on how the CPUVSIM resonates with stakeholders from different cultural backgrounds and educational systems.

6. References

- Bourque, P., & Fairley, R. E. (2022). Guide to the Software Engineering Body of Knowledge, Version 4.0 beta. IEEE Computer Society.
- Chalk, B. (2002). Evaluation of a Simulator to Support the Teaching of Computer Architecture. In 3rd Annual LTSN-ICS Conference, Loughborough University.
- Cortinovic, R. (2021). An educational CPU visual simulator. 32nd Annual Workshop of the Psychology of Programming Interest Group.
- Cortinovic, R., & Rajan, R. (2022). Evaluating and improving the educational CPU visual simulator: a sustainable open pedagogy approach. In Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group, 189-196.
- Decker, R., & Hirshfield, S. (1998). The Analytical Engine: An Introduction to Computer Science Using the Internet. PWS Publishing, Boston.
- Evangelidis, G., Dagdilelis, V., Satratzemi, M., & Efopoulos, V. (2021). X-compiler: yet another integrated novice programming environment. In Proceedings of the IEEE International Conference on Advanced Learning Technologies, 166-169.
- Higginson, P. (2014). Little Man Computer [Javascript application]. Retrieved September 2023, from <https://peterhigginson.co.uk/LMC/>.

- Imai, Y., Hara, S., Doi, S., Kagawa, K., Ando, K., & Hattori, T. (2018). Application and evaluation of visual CPU simulator to support information security education. *IEEJ Transactions on Electronics, Information and Systems*. 138, 9, 1116-1122.
- Imai, Y., Imai, M., & Moritoh, Y. (2013). Evaluation of visual computer simulator for computer architecture education. International Association for Development of the Information Society.
- Kemmis, S., & McTaggart, R. (2000). Participatory action research. In *Handbook of Qualitative Research*, edited by Norman K. Denzin & Yvonna S. Lincoln, 2nd ed., 567-605.
- Miura, Y., Keiichi, K., & Masaki, N. (2003). Development of an educational computer system simulator equipped with a compilation browser. In *Proceedings of the International Conference of Computers in Education*, 140-143.
- Mustafa, B. (2010). Evaluating a system simulator for computer architecture teaching and learning support. *Innovation in Teaching and Learning in Information and Computer Sciences*. 9, 1, 100-104.
- Nikolic, B., Radivojevic, Z., Djordjevic, J., & Milutinovic, V. (2009). A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization. *IEEE Transactions on Education*. 52, 4, 449-458.
- Novick, M. R. (1996). The axioms and principal results of classical test theory. *Journal of Mathematical Psychology*. 3, 1, 1-18.
- Wiley, D., & Hilton, J. (2018). Defining OER-enabled pedagogy. *International Review of Research in Open and Distance Learning*. 19, 4.
- Zhang, Y., & Adams III, G. B. (1997). An interactive, visual simulator for the DLX pipeline. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*. 9-12.

Exploring Teachers' Perspectives on Navigating Recursion Pedagogies

Jude Nzemeke

Department of Computer
Science
City, University of London
jude.nzemeke@city.ac.uk

Marjahan Begum

Department of Computer
Science
City, University of London
Marjahan.begum@city.ac.uk

Jo Wood

Department of Computer
Science
City, University of London
j.d.wood@city.ac.uk

Abstract

Recursion is a fundamental and powerful concept in algorithm design and programming. While invaluable for solving complex problems such as tree traversal and permutation generation, recursion presents challenges for students who often struggle with comprehension, tracing recursive calls, and devising efficient solutions. This study investigates teachers' pedagogical and instructional strategies for teaching recursion, as well as effective assessment techniques. It explores the order in which programming concepts, such as iteration, selection, sequencing, recursion, and object-oriented programming (OOP) are taught in relation to how well students understand the concepts. It highlights the significance of the instructional sequence of these concepts, and reveals that, contrary to the advocated early teaching approach by some researchers – for example, teaching recursion first before iteration – recursion is mainly introduced last to students and is perceived by most of the surveyed teachers as the most challenging concept for students to learn. Teachers' perceptions of the difficulty in teaching these concepts were also explored. Programming Assignments and Coding Challenges are found to be the most popular and effective assessment methods for recursion. The study advocates for an integrated teaching approach that combines tangible objects (e.g., boxes and envelopes) and visual aids (diagrams and animations) to enhance student engagement and understanding during recursion instruction. This multi-sensory approach caters to diverse learning styles and preferences among students, offering a strategy for addressing the challenges associated with teaching recursion.

1. Introduction

The importance of studying recursion spans various domains, particularly in computer science and mathematics. Recursion serves as a powerful technique for addressing problems characterized by repetitive and self-similar structures, forming the basis for the development of intricate algorithms in computer science. By embracing recursive principles, these algorithms offer efficient solutions to challenges that might otherwise be daunting to tackle. Recursion's influence is particularly pronounced in the domain of problem decomposition. While the benefits of recursion are well acknowledged, this study investigates its teaching aspects from the viewpoint of teachers. Exploring how teachers navigate teaching recursion and implement effective assessment methods is crucial for enhancing teaching strategies, addressing challenges, and optimizing instructional sequences.

2. Highlighting Research Questions

1. Identifying the sequence in which programming concepts (iteration, selection, sequencing, recursion, and OOP) are taught, and observing possible relationships with how well students understand these concepts.
2. What are the current instructional approaches and assessment methods that teachers find successful in delivering recursion?
3. How does the frequency and consistency of incorporating movement-based activities, tangible elements, and visual aids in teaching recursion affect the perceived effectiveness of these teaching approaches?

3. Theoretical Foundations of Key Issues

Many students have difficulty understanding recursion and they often use incorrect mental models when evaluating recursive functions (Segal, 1995; Haberman and Averbuch, 2002; Sanders et al. 2006). Novice programmers also face challenges in learning recursion as they have few real-world analogies to formulate a mental model, unlike iteration (Benander et al., 1996). Most learners do not naturally

think recursively (Anderson et al., 1988) and learning recursion poses difficulties due to its unconventional thinking process, especially for students lacking exposure to backward reasoning which involves working from a goal state back to an initial state. Students' previous problem-solving experiences mainly relied on forward reasoning, necessitating a paradigm shift in thinking when encountering recursion (Ginat, 2005).

3.1. Base Case

Learners often struggle with recursive functions, especially understanding the significance of the base case (McCauley et al., 2015). Misconceptions also arise in treating mathematical variables as programming variables, leading to errors, emphasized by McCauley et al. (2015) and compounded by context dependency and processing strategies (Segal, 1995). Hamouda et al. (2017) studied student misconceptions about the base case in recursion, drawing on insights from Sanders and Scholtz (2012). They linked difficulties in flow comprehension to base case misconceptions (Scholtz & Sanders, 2010). Close and Dicheva (1997) associated programming language choice with base case misconceptions, aligning with LOGO studies and Kurland and Pea's (1985) findings on language confusion. Segal (1995) dealt with categorization of "base-case as a stopping condition". Haberman and Averbuch (2002) identified challenges in identifying base cases, crucial for recursive algorithm functionality, as emphasized by them. Inadequate base cases may lead to non-terminating processes and computational inefficiencies, especially with substantial input data.

3.2. Recursion vs Iteration

In a study comparing comprehension of recursion and iteration, Benander et al. (1996) found a statistically significant advantage for recursion. Benander, et al. (2000) found that in small code segments involving linked lists, programmers might find locating bugs in recursive code, particularly in copying tasks, to be easier. Mirolo (2012) contradicted the notion that novice students find iteration easier than recursion, attributing difficulty to task characteristics rather than programming paradigm. Endres et al. (2021) observed superior performance in iterative-framed problems involving non-branching numerical computation. McCracken (1987) cautioned against deeming recursive programming "hopelessly difficult", emphasizing the importance of task matching. Sinha and Vessey (1992) linked construct choice to cognitive fit, advocating task and problem representation considerations.

The debate over whether to teach recursion or iteration first in computer science education involves conflicting perspectives on foundational concepts and ease of understanding. Guzdial in a conversation at the ITISCE 2023 conference and in Guzdial (2018) while referring to studies by Kessler and Anderson (1986) and Wiedenbeck (1989), suggested teaching iteration first due to its easier grasp and broader practical application. Turbak et al. (1999) found introducing recursion before iteration more effective, contrary to traditional methods, challenging the ongoing discourse on optimal sequencing in computer science education. Maiorana et al. (2021) concluded that students can grasp both recursion and iteration simultaneously, supporting the early introduction of recursion to enhance algorithm understanding in the curriculum.

3.3. Pedagogical Approach

To enhance students' understanding of recursion across computer science domains, Velázquez-Iturbide (2000) proposes a progressive teaching method introducing recursion through formal grammars, functional programming, and imperative programming. Syslo and Kwiatkowska (2014) recommend presenting recursion as a "real-life topic" to make it more accessible and relatable, especially for beginners. Explaining recursion to novice programmers can be challenging, and approaches like inductive definitions, Runtime Stack Simulation, Process Tracing, Mathematical induction, Russian Dolls, and the recursion tree (Dann et al., 2001; Haynes, 1995) help address this complexity. Wu et al. (1998) emphasize the importance of conceptual models for teaching recursion to novice programmers, cautioning about adapting or designing concrete models without conveying internal mechanism details. Gunion et al. (2009) challenge concerns about 'middle school' students learning recursion, demonstrating that hands-on activities effectively increase engagement and facilitate learning. Enhancing the learning experience for students can be significantly facilitated by employing tangible materials instead of abstract concepts (Akbaşlı and Yeşilce, 2018). The use of animations, such as in

tools like Alice, during recursion introduction has shown promise in enhancing student comprehension, although further research is needed to establish its long-term impact (Dann et al., 2001).

3.4. Learning Styles or Not

Understanding individual learning styles, especially in programming concepts like recursion (Wu et al., 1998), is vital for effective teaching. Dunn and Dunn advocate tailoring teaching methods to enhance students’ attainment, behaviour, and attitudes based on their research (Dunn, 1984; Dunn et al., 2009). However, teaching in a style different from students may increase cognitive load, hindering learning (Sweller, 1988). Recognizing diverse learning styles, such as visual learning, can reduce cognitive load, improving information assimilation and retention (Jawed et al., 2019). Aligning teaching strategies with varied learning styles is crucial in programming education (Bargar and Hoover, 1984).

Kavale and Forness (1990) defended their meta-analysis against Dunn’s (1989) critique, asserting the ineffectiveness of modality testing and teaching. Pashler et al. (2008) questioned the experimental basis and commercial motives of learning styles, echoed by Reynolds (1997) and Willingham (2005), who cited a lack of scientific evidence. Tarver and Dawson (1978), and Dembo and Howard (2007) opposed modality preference theory, citing empirical limitations and potential harm. Arbutnott and Krätzig (2015) highlighted the inefficacy of tailoring teaching to sensory learning styles. Teachers are advised to focus on content-driven modality choices and universal methods (Kavale and Forness, 1990; Tarver and Dawson, 1978; Willingham, 2005). Various methods to measure modality preferences exist, but caution is needed due to limitations (Willingham, 2005). Instead of catering to individual differences, teachers should employ diverse modalities for variety, attention, and memory strategies, benefiting all students (Tarver and Dawson, 1978; Kavale and Forness, 1987; Willingham, 2005).

In line with Coffield et al. (2004) and other researchers who argue that learning styles are not fixed traits, but rather flexible preferences influenced by context and tasks, our own teaching experiences support this perspective. Through working with diverse groups of learners, we have observed how individuals’ preferences for learning can vary depending on the subject matter and the learning environment. Embracing this viewpoint, we too believe that teachers should prioritize flexibility in their teaching methods, employing a range of strategies that accommodate the dynamic nature of learning preferences.

With this principle in mind, our focus will be on exploring the teaching methods utilized when teaching recursion, particularly looking at teachers’ perceptions of the impact of these methods, including the use of visualization, auditory, reading/writing, and kinaesthetic techniques in teaching. This approach offers a pragmatic means of addressing the research questions and shifts the focus toward identifying effective practices that can benefit a wider array of students in the teaching and learning of recursion, rather than attempting to tailor instruction to each individual student's unique learning style.

4. Methodology

Creswell (2009) highlighted the importance of philosophical worldview – “a basic set of beliefs that guide action” in research design framework. These beliefs can be used by researchers to decide if they should make use of qualitative, quantitative, or mixed methods approach. The design framework can be illustrated further as seen in figure 1 below:



Figure 1: Design Framework [Creswell, 2009].

We apply a mixed-methods approach to investigating current teaching practice. Born from the paradigm wars, it combines qualitative and quantitative approaches, offering a comprehensive view of complex topics (Johnson and Onwuegbuzie, 2004; Terrell, 2012; Poth and Munce, 2020). Utilizing both methods enhances understanding and explores multifaceted problems from various perspectives (Poth & Munce, 2020). Rooted in pragmatism, mixed methods emphasizes practical outcomes and provides diverse design choices for researchers (Shorten & Smith, 2017; Terrell, 2012). This approach, applicable across disciplines, proves valuable in answering intricate research questions (Terrell, 2012). The point of integration is one of the primary design dimensions for mixed method research. It is defined as “any point in a study where two or more research components are mixed or connected in some way” (Schoonenboom and Johnson, 2017). Getting the process of data integration from qualitative and quantitative components of the study right is key to have more insight of the data collected in mixed methods, and this can take place during the analysis phase of the study.

Quantitative data were gathered through a comprehensive online survey featuring 24 questions, with 21 focused on quantitative information. The survey covered non-sensitive demographic data, programming language used by teachers, challenges in teaching programming concepts and pedagogical approaches. It inquired into instructional methods, assessment approaches, and effective techniques in teaching recursion, providing a thorough overview of quantitative aspects in programming education.

Complementing the quantitative findings, qualitative insights were obtained through open-ended questions, enabling in-depth participant responses unconstrained by predetermined choices (Hyman and Sierra, 2016). To optimize completion rates, the survey incorporated a three-box limit for open-text responses, guided by Qualtrics online experts, acknowledging that exceeding this limit could reduce completion rates due to increased cognitive effort required for responses.

5. Ethical Considerations

In adhering to ethical standards outlined by the British Educational Research Association (BERA, 2018) and City, University of London, this educational research prioritized informed consent. Ethical approval from City, University of London’s ethics committee was secured for this research.

6. Sampling and Recruitment

Examination boards in England, tasked with developing detailed subject specifications that outline the curriculum framework – including what students are expected to learn, understand, and achieve by the end of the course – focus on recursion exclusively in post-secondary education (for students aged 16 to 18). Therefore, it was expected that primarily, teachers who teach recursion at this level and above, would take part in the study. However, due to the relatively low enrolment of computer science students in England, in post-secondary school (OFQUAL, 2023), the pool of teachers specializing in teaching recursion is anticipated to be limited. From our experience and from interaction with teachers, we know that this challenge arises because some teachers may opt not to cover recursion at primary and secondary school (attended by students less than 16 years old), potentially due to time constraints in delivering the curriculum. To address the challenge of recruiting teachers for the study, who teach recursion, the Digital SchoolHouse Ingenuity Day 1 conference event was strategically targeted, a gathering primarily attended mainly by computer science teachers.

Initially, 36 computer science teachers began the survey, comprising 61% male, 30% female, with 5% opting not to disclose their gender, and 2% identifying as non-binary. Regarding ethnicity, 64% identified as White British, 17% as other white backgrounds, and 6% each for Black and Asian backgrounds, with an additional 5% identifying as other ethnic backgrounds, while 2% chose not to disclose. One of the questions in the survey was designed to screen participants for eligibility – targeting only teachers who have taught or currently teach recursion. The question simply asked, “Have you taught or are you currently teaching recursion as part of your curriculum?” The survey ended for teachers who responded “No” to this question, indicating that they lacked the experience of teaching this topic. The eligibility screening, verifying experience in teaching recursion, narrowed the final sample to 14 teachers. Among them, seven teachers had 15 or more years of teaching experience, three teachers had 11–15 years, and 6 had 6–10 years, with only one having less than one year teaching experience. All teach recursion to post-secondary students (16 to 18-year-olds), with 3 also teaching at

Foundation and Undergraduate levels (18+ years old). These criteria ensured a focused and valid study with contributions from mostly experienced teachers across institutions, effectively addressing research objectives while acknowledging generalizability limitations.

7. Data Analysis and Discussion

7.1. RQ1: Identifying the sequence in which programming concepts (iteration, selection, sequencing, recursion, and OOP) are taught, and observing possible relationships with how well students understand these concepts.

7.1.1. Order of Teaching Concepts:

In investigating the teaching sequence of programming concepts, we explored the order in which these concepts are typically introduced. The instructional sequence can significantly influence students' comprehension and retention, providing insights into teachers' approaches. Of particular interest is the positioning of recursion relative to other concepts, indicating its foundational or advanced nature. Teachers were asked to rank the order in which they taught the different concepts. Analyzing mean rankings, with lower numbers indicating earlier introduction, reveals a consistent progression: sequence (1.64), selection (1.93), iteration (2.50), OOP (4.43), and recursion (4.50). This order aligns with a pedagogical strategy that introduces simpler concepts as building blocks before tackling more complex and abstract ideas. It appears to be a strategy that supports argument made by Kessler and Anderson (1986) and the subsequent study conducted by Susan Wiedenbeck (1989) that teaching iteration before recursion is more beneficial, as iteration is easier to grasp and has wider practical application.

7.1.2. How Challenging are these Concepts to Students:

Teachers' insights into students' struggles with specific concepts further inform the analysis. Teachers were asked to rank the order in which students understood the different concepts from easy to understand, to very hard to understand. Recursion topped the list as the most challenging concept for students (with a mean value of 3.93), followed closely by OOP (with a mean value of 3.79). Sequencing was perceived as the least challenging (with a mean value of 2.29). Examining standard deviation and variance highlighted the variability in ratings, with recursion exhibiting the highest values (0.74 and 0.55) and iteration the lowest (0.81 and 0.66), indicating the range of opinions among respondents.

7.1.3. How Difficult are these concepts to teach:

Investigating the difficulty levels teachers encounter when teaching the programming concepts, our findings highlight a significant variation in perceived challenges. For OOP, teachers reported a mean difficulty of 3.71, with a median difficulty of 4, indicating it is one of the most challenging topics to teach. Recursion followed closely with a mean difficulty of 3.64 and a similar median (of 4). Both concepts showed a wide range of challenge levels, from somewhat challenging to very challenging. Sequencing, iteration, and selection were considered less challenging, with mean difficulties of 2.07, 2.21, and 2.00, respectively, and medians at 2. These topics were generally seen as easier to teach, with their difficulty ranging from non-challenging to moderately challenging.

7.1.4. Findings for RQ1

Due to the impact on the statistical power of a smaller sample size, it was deemed that data collected might not have enough power to detect a significant correlation using nonparametric measures for example Spearman Rank Correlation. Bujang and Baharum (2016) proposed a minimum of 29 samples (or subjects) to detect a reasonably high correlation (specifically, a correlation coefficient of 0.5) with a good balance of error tolerance and study power. They noted other studies "(Bujang et al., 2009; Bujang et al., 2015)" that suggest samples larger than 300 can yield statistical results highly representative of the true population values. This is based on the idea that larger samples tend to provide more precise estimates of population parameters, thereby improving the generalizability of the findings. This made it apparent that the best way to analyse the data will be to look at it from a practical or observational perspective.

The violin plots below (Figure 2: Teacher Order and Students Understanding Order) are used to illustrate both the spread and the median of the orders in which the programming concepts are taught and students understanding, with one side of the violin for teaching orders and the other for indicating order in which student understanding concepts. The following observations were made:

The concept of sequence typically appears early in learning, supported by students finding it easier to understand, which aligns with it likely being an introductory concept. Selection is also introduced early on, yet students find its difficulty level consistent, irrespective of its teaching order. Iteration tends to be taught mid-way through the curriculum and is well understood by students, indicating its placement is appropriate for their learning curve. OOP is often reserved for the latter part of educational programs, which is reflected in its broader and more challenging understanding distribution, highlighting its complexity. Recursion stands out with a distinct pattern where both teaching and understanding orders are skewed to the higher end, indicating it is both taught late and considered difficult to understand.

Overall, the data reflects a structure in teaching programming that rises from simpler to more complex concepts, aligning well with student comprehension levels.

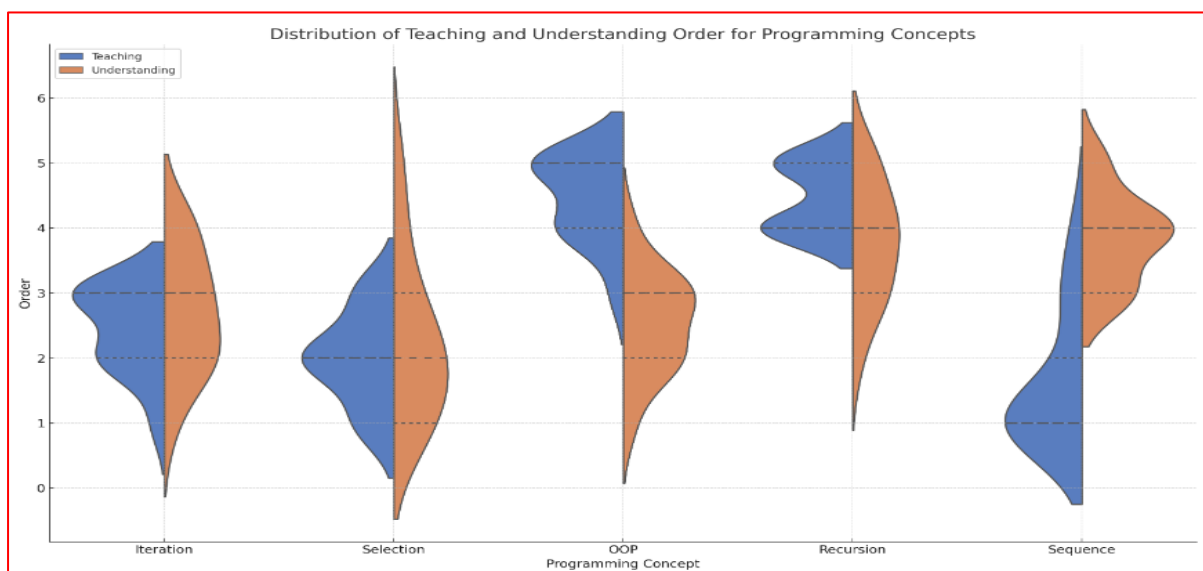


Figure 2: Teacher Order and Students Understanding Order

Note for Figure 2. The shape and width of the violins provide an immediate visual indication of the distribution's spread and density. A wider section of a violin plot indicates a higher frequency of data points (i.e., more teachers reported similar orders), whereas a narrower section indicates fewer data points. The horizontal lines within each violin represent the median order. This is crucial for exploring the most common teaching order, and the understanding order for each concept. The degree of symmetry between the teaching and understanding sides of each violin gives an immediate visual cue about alignment. High symmetry suggests that the understanding order closely matches the teaching order, whereas asymmetry suggests discrepancies.

7.2. RQ2 What are the current instructional approaches and assessment methods that teachers find successful in delivering recursion?

Teachers' approaches to teaching recursion offer insights into adapting methods for diverse learners. The choice of instructional approach significantly influences students' understanding and engagement with recursion concepts in programming education. The approaches used in the survey are defined as follows:

Inquiry-based learning is an approach where students learn through questioning and investigation. When teaching recursion, this approach might involve encouraging students to explore recursion concepts by asking questions, conducting research, and experimenting. For instance, students might investigate different recursive algorithms and their applications.

Direct instruction involves presenting information to students in a structured and systematic way. When teaching recursion, this approach may include clear explanations of recursion concepts, step-by-step examples, and guided practice. For instance, the teacher might systematically introduce recursive functions and then provide exercises to reinforce the learning.

Project-based learning is an approach where students learn by working on projects. When teaching recursion, this approach may involve assigning projects that require students to apply recursive concepts practically. For example, students could be asked to create a recursive artwork generator or a recursive maze-solving program.

Problem-based learning is an approach where students learn by solving problems. In the context of recursion, this approach might involve presenting students with real-world problems that can be solved using recursive techniques.

Collaborative learning is an instructional approach where students work together. When teaching recursion, this approach might involve students collaborating on recursive coding projects or solving recursion-related problems as a team. For example, students may work together to create a recursive function in a programming language.

Differentiated instruction [or Adapting Teaching] acknowledges the diverse needs of students. If a teacher selects this approach when teaching recursion, it means they are adapting their instruction to cater to individual students' learning styles and abilities. For instance, a teacher might provide additional resources or assignments to support struggling students while challenging advanced learners with more complex recursion problems.

Blended learning combines online and in-person instruction. In the context of teaching recursion, this approach could involve using online resources and platforms to complement in-person lessons. For example, students might watch online tutorials on recursion algorithms and then apply what they've learned during in-person coding sessions.

Independent learning is an approach where students research topics on their own. In the context of recursion, this approach may involve assigning self-directed projects or providing resources for students to explore recursion independently. For example, students could be given a list of recursion-related books and websites to explore as part of their learning process.

Common approaches include problem-based and inquiry-based learning, with problem-based and project-based learning considered the most effective. Collaborative learning, although less prevalent, still proves effective. However, further exploration is needed regarding differentiated instruction and blended learning. See Figure 3 for graph on instructional approaches and their effectiveness.

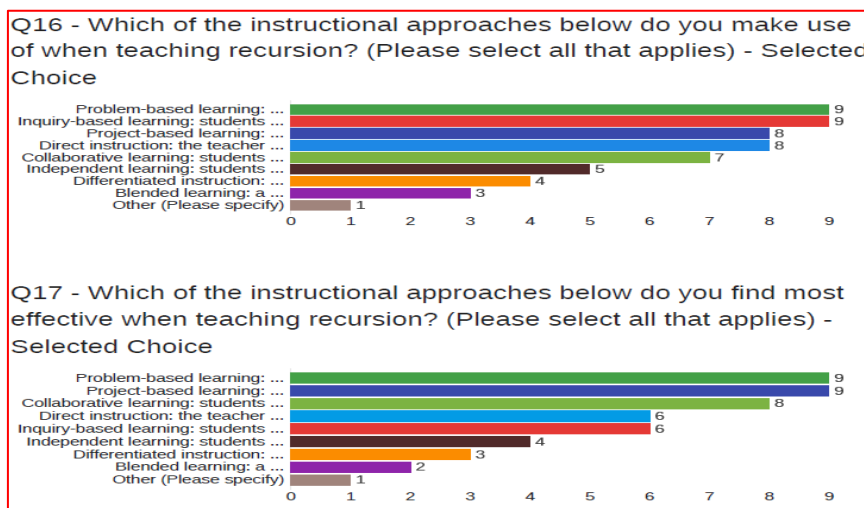
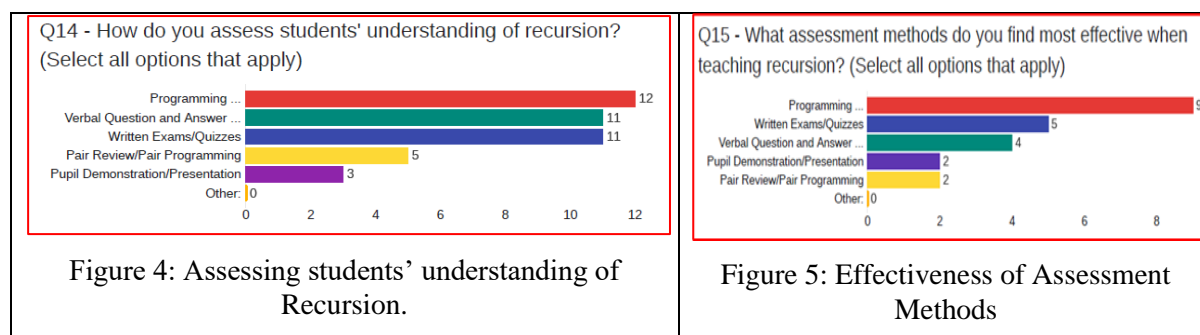


Figure 3: Instructional Approaches for Teaching Recursion

Evaluating assessment methods used in teaching recursion is crucial for assessing their effectiveness in measuring student comprehension. The options include Written Exams/Quizzes, Programming Assignments/Coding Challenges, Verbal Question and Answer sessions in lessons, Pair Review/Pair Programming, and Pupil Demonstration/Presentation. The diversity of assessment methods recognizes varied student learning styles and abilities, influencing how teachers adapt teaching strategies. This

information helps identify assessment methods that promote a deeper understanding of recursion concepts.

Assessment methods vary, with programming assignments being the most popular and effective, followed by verbal Q&A and written exams. Pair review/pair programming and pupil demonstration/presentation are less common. Our results indicate that most teachers find programming assignments the most effective assessment method for recursion, followed by written exams/quizzes, verbal Q&A, and pupil demonstration/presentation. Pair review/pair programming is perceived as the least effective method. No respondents indicated the use of alternative assessment methods. See Figure 4 and Figure 5 for graph on assessing students.



What are teachers' perceptions of the challenges faced by pupils when learning recursion?

Further inquiry was made to investigate the challenges students commonly encounter while learning recursion, with the goal of informing targeted teaching strategies. This inquiry is crucial for understanding specific pain points in student learning and facilitating the creation of effective teaching materials. Different challenges may emerge at varying educational levels or with specific programming languages, emphasizing the need for customized instruction. Responses from teachers highlighted prevalent challenges, including students struggling to comprehend the purpose of recursion and lacking foundational knowledge of iteration – which is sometimes mistaken for recursion and vice versa. To address this, teachers should ensure students have a solid grasp of fundamental concepts before introducing recursion, indicating the importance of a well-structured curriculum.

Another noteworthy challenge is students relying solely on data tracing to understand recursion, calling for encouragement to look into the underlying principles for a deeper conceptual understanding. Confusion between recursion and other programming concepts requires clear differentiation and practical examples to alleviate misunderstandings. Understanding the context and rationale behind recursive code is identified as a challenge, suggesting the importance of real-world examples and practical applications to enhance comprehension.

Teachers also expressed a lack of in-depth resources on how to teach recursion as a challenge, emphasizing the need for comprehensive materials catering to diverse learning styles and experience levels. This highlights the importance of resource development to support teachers in delivering effective instruction on recursion.

The survey sought advice from teachers on teaching recursion effectively to those new to the subject. Recommendations included thorough preparation, confidence, and simplicity in tasks to avoid overload. Peer support, understanding individual student needs, and fostering a student-centric approach were highlighted.

Practical aspects, such as extensive practice, providing examples, and using teaching tools like real-world examples and visual aids, were emphasized for diverse learning styles. The belief in spending more time on the topic highlighted the importance of patience and a comprehensive exploration of recursion for better understanding. Ensuring a strong foundation by understanding basics before tackling complex topics was advised. Teachers suggested addressing potential difficulties students may face with recursion by adapting teaching methods and maintaining focus and conciseness in delivery.

7.3. How does the frequency and consistency of incorporating movement-based activities, tangible elements, and visual aids in teaching recursion affect the perceived effectiveness of these teaching approaches?

The research explored teaching approaches for recursion, focusing on methods and the effectiveness of the use of hands-on activities, tangible materials, and visual aids.

Six out of 14 teachers occasionally use hands-on or movement-based activities for recursion, with 3 using them frequently. The perceived effectiveness varies, with 7 teachers rating them moderately effective, 4 very effective, and 3 slightly effective. Interestingly, no extreme opinions were expressed, indicating varied perceptions among teachers. While not universally adopted, hands-on activities are generally perceived as beneficial by those who incorporate them; and can enhance student engagement and understanding of abstract concepts like recursion.

Most teachers (9 out of 14) seldom or occasionally incorporate tangible elements in teaching recursion. The effectiveness varies, with 5 finding it very or extremely effective and 4 considering it slightly important. Further investigation is suggested to understand why some find this approach effective despite infrequent use. Understanding the specific tangible elements and materials used could offer valuable insights into effective teaching strategies for recursion.

The study emphasizes the use of visual aids, such as diagrams and animations, in teaching recursion. A significant majority (10 out of 14) frequently or always use visual aids, finding them highly effective in conveying recursion concepts. Only two teachers found them slightly effective. Visual aids have gained widespread acceptance, indicating their effectiveness in teaching recursion.

Two teachers who mainly teach post-secondary students in different schools, strongly advocated for the use of PRIMM (Predict, Run, Investigate, Modify and Make) and differentiation approaches when teaching recursion and other programming concepts in general. In a follow-up conversation with one of the teachers after the survey, they claimed to have observed marked improvements in students' outcomes, particularly in learning about recursive algorithms, since implementing PRIMM approach at their school "for over three years now". In their study, Sentance et al. (2019) suggested that PRIMM offers an efficient method for teaching programming, enhancing comprehension, and boosting confidence in students. They emphasized that teachers found PRIMM's structured lessons beneficial, offering clarity in both lesson planning and delivery and it allowed teachers to tailor tasks to individual student needs. Additionally, they recommended PRIMM's suitability for teacher training and various stages of programming education.

8. Further Discussion

The study emphasizes recursion as the most challenging programming concept for students, from the teacher's perspective, shedding light on practical difficulties in the classroom. This recognition prompts teachers to allocate additional time and resources, enhancing instructional strategy effectiveness. Contrary to literature advocating for early or pre-iteration teaching of recursion, as highlighted in the Theoretical Foundation section above, our study reveals that among the programming concepts examined, recursion is, in fact, introduced to students last. The findings regarding assessment methods and pedagogical approaches offer practical insights for teachers and researchers alike. Programming assignments and coding challenges emerge as the preferred and most effective assessment tools for recursion, providing a clear direction for teachers when designing students' evaluations, in their lesson planning. Additionally, the endorsement of problem-based learning and inquiry-based learning, alongside the nuanced impact of collaborative learning, offers valuable guidance for teachers seeking effective instructional strategies in the teaching and learning of recursion. While employing tangible objects for example boxes and envelopes to symbolize values returned by functions in recursive calls has proven highly effective in our teaching experience, our research emphasizes that, from the teachers' perspective, visual aids (diagrams and animations), are widely embraced and very effective. We contend that the use of tangible objects, despite being a hands-on approach, also offers a form of visualization for learners. This highlights their crucial role in bolstering student engagement and understanding during recursion instruction. The study advocates for an integrated teaching approach that combines for example, tangible objects and visual aids, fostering a multi-sensory learning experience that caters to diverse learning styles and preferences among students. This integrated approach holds the potential to

significantly enhance the effectiveness of recursion instruction and contribute to improved learning outcomes.

9. Conclusion

In conclusion, the relationship between the order of teaching programming concepts and students' understanding suggests a general alignment with educational theory: simpler concepts are introduced first, leading to a smoother learning curve for students. However, certain concepts like OOP and recursion present challenges that are recognized by both the teaching order and students' understanding. This could point to areas where additional teaching aids, practice, or alternative instructional strategies might be beneficial. The study brings to light the complexities inherent in teaching and learning recursion. Ongoing monitoring of students' understanding relative to the teaching order is crucial. Adjustments to this teaching sequence, if necessary, should be data-driven and responsive to the observed learning outcomes. Furthermore, problem-based learning emerged as the most effective method for teaching recursion, with programming assignments being the most popular and effective assessment approach. Practical insights into effective assessment methods and teaching approaches empower teachers to refine their techniques. We advocate for an integrated teaching approach that incorporates tangible objects and visual aids, offering a strategy that may enhance student engagement and comprehension. Together, these findings provide teachers with a framework to address the challenges associated with teaching recursion, fostering an environment conducive to improved learning experiences and outcomes.

10. Limitations

While we are confident in the credibility and meaningfulness of the data collected from the teachers surveyed, it is essential to acknowledge several limitations inherent in our study. Firstly, the sample size presents a challenge. While a larger sample size would enhance the statistical power and generalizability of the study, the inclusion criteria ensured that all participants had relevant experience, which is crucial for the study's focus on teaching recursion. Additionally, the distribution of teaching experience among the final sample, with a majority having 15 or more years of experience, adds credibility to the insights gathered. Furthermore, while our study provides valuable insights into current teaching approaches used in the delivery of recursion, it is not without its constraints. We cannot definitively establish causality between teaching sequence of concepts and students' learning outcomes, as correlation does not imply causation. As such, our study does not offer comprehensive explanations for the observed variables. Despite these limitations, we believe that our findings contribute valuable insights into the field of Computer Science education.

11. Future Work

Drawing from the findings of this research, future investigations could explore the impact of instructional sequence on the perceived and actual difficulty of programming concepts. Research could scrutinize how altering the sequence of these concepts influences teacher perceptions, student understanding, and overall learning outcomes especially regarding recursion. Future research could also aim to address these limitations by employing larger sample sizes, utilizing experimental designs to establish causality, and explore the underlying mechanisms driving teaching practices. Furthermore, specific studies should be conducted, focusing on the role and effectiveness of visual aids, alongside the integration of tangible elements and physical materials in teaching recursion. These focused inquiries promise to provide practical insights for teachers, exploring the potential benefits of these approaches in enhancing both student comprehension and engagement.

12. References

1. Akbaşlı, Sait & Yeşilce, İlknur. (2018). Use of Tangible Materials and Computer in Mathematics Teaching: Opinions of School Principals. *Eurasia Journal of Mathematics, Science and Technology Education*, 14. 2523-2532. 10.29333/ejmste/90087.
2. Anderson, J. R., Pirolli, P., & Farrel, R. (1988). Learning to program recursive functions. In M. T. Chi, R. Glaser, & M. J. Farr (Eds.), *The nature of expertise* (pp. 153–183). Hillsdale: Erlbaum.
3. Arbuthnott, K. D., & Krätzig, G. P. (2015). Effective Teaching: Sensory Learning Styles versus General Memory Processes. *Comprehensive Psychology*, 4. <https://doi.org/10.2466/06.IT.4.2>.

4. Bargar, R. R., & Hoover, R. L. (1984). Psychological Type and the Matching of Cognitive Styles. *Theory Into Practice*, 23(1), 56–63. <http://www.jstor.org/stable/1476739>.
5. Benander, A. C., Benander, B. A., & Pu, H. (1996). Recursion vs. iteration: An empirical study of comprehension. *Journal of Systems and Software*, 32, 73–82.
6. Benander, A.C., Benander, B.A., & Sang, J. (2000). An empirical analysis of debugging performance - differences between iterative and recursive constructs. *J. Syst. Softw.*, 54, 17-28.
7. BERA. (2018). Ethical guidelines for educational research (4th ed.). <https://www.bera.ac.uk/publication/ethical-guidelines-for-educational-research-2018-online>.
8. Bujang, Mohamad Adam & Baharum, Nurakmal. (2016). Sample Size Guideline for Correlation Analysis. *World Journal of Social Science Research*. 3. 37. 10.22158/wjssr.v3n1p37.
9. Claudio Mirolo. 2012. Is iteration really easier to learn than recursion for CS1 students? In *Proceedings of the ninth annual international conference on International computing education research (ICER '12)*. Association for Computing Machinery, New York, NY, USA, 99–104. <https://doi.org/10.1145/2361276.2361296j>
10. Claudius M. Kessler & John R. Anderson (1986) Learning Flow of Control: Recursive and Iterative Procedures, *Human–Computer Interaction*, 2:2, 135-166, DOI: 10.1207/s15327051hci0202_2
11. Coffield, F. & Moseley, D. & Hall, Elaine & Ecclestone, K. (2004). Learning Styles and Pedagogy In Post-16 Learning: A Systematic And Critical Review. *Book Learning styles and pedagogy in post-16 learning: a systematic and critical review*.
12. Creswell, J. W. (2009). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches* (3rd ed.). Sage Publications, Inc.
13. Dembo, M. H., & Howard, K. (2007). Advice about the use of learning styles: A major myth in education. *Journal of College Reading and Learning*, 37(2), 101-109. <https://doi.org/10.1080/10790195.2007.10850174>.
14. Dicheva, Darina & Close, Sean. (1997). Misconceptions in recursion: diagnostic teaching.
15. Dunn, R. (1984). Learning Style: State of the Science. *Theory Into Practice*, 23(1), 10–19. <http://www.jstor.org/stable/1476733>
16. Dunn, R. (1990). Bias over Substance: A Critical Analysis of Kavale and Forness' Report on Modality-Based Instruction. *Exceptional Children*, 56(4), 352-356. <https://doi.org/10.1177/001440299005600409>
17. Dunn, R., Honigsfeld, A., Doolan, L. S., Bostrom, L., Russo, K., Schiering, M. S., Suh, B., & Tenedero, H. (2009). Impact of Learning-Style Instructional Strategies on Students' Achievement and Attitudes: Perceptions of Educators in Diverse Institutions. *The Clearing House*, 82(3), 135–140. <http://www.jstor.org/stable/30181095>
18. Endres, M., Weimer, W., & Kamil, A. (2021). An Analysis of Iterative and Recursive Problem Performance. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*.
19. F. Turbak, C. Royden, J. Stephan, and J. Herbst, Teaching Recursion Before Loops In CS1, *Journal of Computing in Small Colleges*, Volume 14, Number 4, pp 86-101, May 1999.
20. Ginat, D. (2005). The suitable way is backwards, but they work forward. *Journal of Computers in Mathematics and Science Teaching*, 24, 73–88. Norfolk, VA: AACE.
21. Gunion, Katherine & Milford, Todd & Stege, Ulrike. (2009). The Paradigm Recursion: Is It More Accessible When Introduced in Middle School?. *The Journal of Problem Solving*. 2. 10.7771/1932-6246.1063.
22. Guzdial, M. (2018) Exploring the question of teaching recursion or iterative control structures first, *Computing Ed Research - Guzdial's Take*. Available at: <https://computinged.wordpress.com/2018/03/09/exploring-the-question-of-teaching-recursion-or-iterative-control-structures-first/> (Accessed: 22 October 2023).
23. Haberman, B., & Averbuch, H. (2002, June 24–28). The case of base cases: Why are they so difficult to recognize? Student difficulties with recursion. In *Proceedings of the 7th conference on innovation and technology in computer science education*. Aarhus.
24. Hamouda, S., Edwards, S., Elmongui, H., Ernst, J., & Shaffer, C. (2017). A basic recursion concept inventory. *Computer Science Education*, 27(2), 121–148.
25. Hyman, Michael & Sierra, Jeremy. (2016). Open- versus close-ended survey questions. *NMSU Business Outlook*. 14.

26. Ian Sanders, Vashti Galpin, and Tina Götschi. 2006. Mental models of recursion revisited. *SIGCSE Bull.* 38, 3 (September 2006), 138–142. <https://doi.org/10.1145/1140123.1140162>
27. J Terrell, Steven. (2012). *Mixed-Methods Research Methodologies*. Qualitative Report. 17. 254-265. 10.46743/2160-3715/2012.1819.
28. Jawed S, Amin HU, Malik AS and Faye I. (2019). Classification of Visual and Non-visual Learners Using Electroencephalographic Alpha and Gamma Activities. *Front. Behav. Neurosci.* 13:86.
29. Johnson, R. B. & Onwuegbuzie, A. J. (2004). Mixed-methods research: a research paradigm whose time has come. *Educational Researcher*, 33(7), 14-26.
30. Kavale, K. A. and Forness, S. R. (1987). Substance over style: Assessing the efficacy of modality testing and teaching. *Exceptional Children*, 54(3), 228–239.
31. Kavale, K. A., & Forness, S. R. (1990). Substance over Style: A Rejoinder to Dunn's Animadversions. *Exceptional Children*, 56(4), 357-361. <https://doi.org/10.1177/001440299005600410>
32. Kurland, D. M., & Pea, R. D. (1985). Children's Mental Models of Recursive Logo Programs. *Journal of Educational Computing Research*, 1(2), 235-243. <https://doi.org/10.2190/JV9Y-5PD0-MX22-9J4Y>.
33. Maiorana, F., Csizmadia, A., Richards, G., Riedesel, C. (2021). Recursion Versus Iteration: A Comparative Approach for Algorithm Comprehension. In: Auer, M.E., Centea, D. (eds) *Visions and Concepts for Education 4.0. ICBL 2020. Advances in Intelligent Systems and Computing*, vol 1314. Springer, Cham. https://doi.org/10.1007/978-3-030-67209-6_27
34. McCracken, D. D. (1987, January). Ruminations on computer science curricula, viewpoint column. *Communications of the ACM*, 30, 3–5.
35. OFQUAL (2023). Official Statistics, Provisional Entries for GCSE, AS and A Level: Summer 2023 Exam Series. Gov.UK. Retrieved December 24, 2023, from www.gov.uk/government/statistics/provisional-entries-for-gcse-as-and-a-level-summer-2023-exam-series/provisional-entries-for-gcse-as-and-a-level-summer-2023-exam-series
36. Pashler, H., McDaniel, M., Rohrer, D. and Bjork, R., 2008. Learning styles: Concepts and evidence. *Psychological science in the public interest*, 9(3), pp.105-119.
37. Poth, C., & Munce, S. E. P. (2020). Commentary—Preparing today's researchers for a yet unknown tomorrow: Promising practices for a synergistic and sustainable mentoring approach to mixed methods research learning. *International Journal of Multiple Research Approaches*, 12(1), 56-64. doi:10.29034/ijmra.v12n1commentary
38. Renée McCauley, Scott Grissom, Sue Fitzgerald & Laurie Murphy (2015) Teaching and learning recursive programming: a review of the research literature, *Computer Science Education*, 25:1, 37-66, DOI: 10.1080/08993408.2015.1033205.
39. Reynolds, M. (1997). Learning Styles: A Critique. *Management Learning*, 28(2), 115-133. <https://doi.org/10.1177/1350507697282002>.
40. S. M. Haynes. 1995. Explaining recursion to the unsophisticated. *SIGCSE Bull.* 27, 3 (Sept. 1995), 3–6. <https://doi.org/10.1145/209849.209850>.
41. Sanders, I., & Scholtz, T. (2012). First year students' understanding of the flow of control in recursive algorithms. *African Journal of Research in Mathematics, Science and Technology Education*, 16(3), 348–362
42. Scholtz, T. L., & Sanders, I. (2010). Mental models of recursion: Investigating students' understanding of recursion. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2010)*(pp. 103–107). Bilkent, Ankara, Turkey.
43. Schoonenboom J, Johnson R. B. (2017). How to Construct a Mixed Methods Research Design. *Kolner Z Soz Sozpsychol.* 2017;69(Suppl 2):107-131. doi: 10.1007/s11577-017-0454-1. Epub 2017 Jul 5. PMID: 28989188; PMCID: PMC5602001.
44. Segal, J. (1995). Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science*, 22, 385–411. <https://doi.org/10.1007/BF00891962>
45. Shorten A., & Smith J. (2017). Mixed methods research: Expanding the evidence base. *Evid Based Nurs*, 20, 74–5. <http://dx.doi.org/10.1136/eb-2017-102699>
46. Sinha, A., and Vessey, I., (1992) Cognitive Fit: An Empirical Study of Recursion and Iteration, *IEEE Trans. Software Eng.* 18, 368-379.

47. Sue Sentance, Jane Waite, and Maria Kallia. 2019. Teachers' Experiences of using PRIMM to Teach Programming in School. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). Association for Computing Machinery, New York, NY, USA, 476–482. <https://doi.org/10.1145/3287324.3287477>
48. Susan Wiedenbeck, Learning iteration and recursion from examples, *International Journal of Man-Machine Studies*, Volume 30, Issue 1, 1989, Pages 1-22, ISSN 0020-7373, [https://doi.org/10.1016/S0020-7373\(89\)80018-5](https://doi.org/10.1016/S0020-7373(89)80018-5).
49. Sweller, J. (1988), Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, 12: 257-285. https://doi.org/10.1207/s15516709cog1202_4.
50. Syslo, Maciej & Kwiatkowska, Anna. (2014). Introducing Students to Recursion: A Multi-facet and Multi-tool Approach. 124-137. [10.1007/978-3-319-09958-3_12](https://doi.org/10.1007/978-3-319-09958-3_12).
51. Tarver, S. G., & Dawson, M. M. (1978). Modality preference and the teaching of reading: A review. *Journal of Learning Disabilities*, 11(1), 17-29. <https://doi.org/10.1177/002221947801100104>
52. Velázquez-Iturbide, J. Ángel. (2000). Recursion in gradual steps (is recursion really that difficult?). *ACM SIGCSE Bulletin*. 32. 310-314. [10.1145/331795.331876](https://doi.org/10.1145/331795.331876).
53. Wanda Dann, Stephen Cooper, and Randy Pausch. 2001. Using visualization to teach novices recursion. *SIGCSE Bull.* 33, 3 (Sept. 2001), 109–112. <https://doi.org/10.1145/507758.377507>.
54. Willingham D. T. (2005) Do visual, auditory, and kinesthetic learners need visual, auditory, and kinesthetic instruction? *American Educator*, 29, 31–35.
55. Wu, Cheng-Chih & Dale, Nell & Bethel, Lowell. (1998). Conceptual models and cognitive learning styles in teaching recursion. *ACM Sigcse Bulletin*. 30. 292-296. [10.1145/273133.274315](https://doi.org/10.1145/273133.274315).

Understanding APIs and the software that provides them - Analysis of programmers' API mental models used in programming tasks

Ava Heinonen

Department of Computer Science

Aalto University

ava.heinonen@aalto.fi

Abstract

At one point in time, programming could be thought of as the act of translating program requirements into source code written in one programming language. However, modern programming relies heavily on building programs by integrating services, functionality, and data provided by external software into a program using APIs. In part, programming consists of selecting suitable software that provides ready solutions for programming problems and using their APIs to integrate those solutions into a coherent program.

This change in programming necessarily changes programmers' mental models — their understanding of the programs they work on. In this paper, we discuss programmers' mental models when using an API in a programming task. We conducted interviews with twelve industry professionals using the critical decision method. We analyzed the mental models — the understanding these practitioners utilized when completing different software development tasks using an API. Through this analysis, we were able to identify information about the tasks, the software providing an API, and the API that were represented in the programmers' mental models. These results contribute to the existing literature by opening a discussion on how using APIs changes the nature of programs and programming and by providing insight into the understanding necessary for completing programming tasks using APIs.

1. Introduction

Modern software development can be understood as a process of integration, combining software components into an overall system. Software libraries and frameworks provide pre-implemented solutions for many programming problems, which can be integrated into a new system by calling methods in their APIs. Services hosted on the Web can also be utilized as part of a software system by making requests to their API endpoints (Mäkitalo, Taivalaari, Kiviluoto, Mikkonen, & Capilla, 2020).

This shift in software development has changed the nature of computer programs. In the past, a program could be thought of as a specification written in a programming language that expresses a set of calculations (Pair, 1990; Détienne & Détienne, 2002b). A program was a set of calculations and objects, expressed in a programming language with rules on how to organize words into meaningful expressions (Pair, 1990; Détienne & Détienne, 2002b). While a program is still a set of calculations and objects, the way these are expressed is different. A program now comprises objects and calculations expressed as code written in a programming language or implemented elsewhere and accessed via API calls. The meaning of these calls is defined not by the programming language itself but by the programs implementing them. A program is a set of calculations and objects, some implemented in the program's source code and some by other programs. These are expressed either as source code or as API calls that indicate the execution of externally implemented calculations.

This shift in what a program is necessarily changes the way programmers design and implement programs and, even more so, the knowledge and understanding required to do so (Andrews, Ghosh, & Choi, 2002). As a programmer designs and implements a program, they form an internal representation of it — a mental model that represents the program to be implemented (Heinonen, Lehtelä, Hellas, & Fagerholm, 2023; Kim, Lerch, & Simon, 1995). This understanding of the program to be implemented then guides the process of translating the program design into an executable program (Pennington & Grabowski, 1990). However, the discussion around programmers' mental models has focused on programs as speci-

fications written in a programming language that expresses the calculations of the program (Détienne & Détienne, 2002b; Pennington & Grabowski, 1990; Heinonen et al., 2023).

Programming also requires the programmer to translate the designed solution into an executable program (Pennington & Grabowski, 1990). The shift in software development has transformed this task from writing code that implements a functionality to writing code that interacts with the interface of a program that implements a functionality (Andrews et al., 2002). This process can be described as writing "glue" code or writing client code that integrates functionality from different software components into one program (Chen, He, Liu, & Zhan, 2007).

In some ways, writing client code is similar to software reuse, where the programmer must understand what type of solution is required, find a suitable solution, and adapt it to fit the target software (Détienne & Détienne, 2002a). However, reuse theories do not account for the differences between reusing a piece of code and integrating a program into another using its API. The latter requires not only understanding the solution and its suitability, but also another level of understanding: the interface through which the solution can be accessed (Thayer, Chasins, & Ko, 2021; Mosqueira-Rey, Alonso-Ríos, Moret-Bonillo, Fernández-Varela, & Álvarez-Estévez, 2018). Successfully using an API also requires the programmer to form a mental model of the software providing the API — what it is, what it can be used for, and how it can be used in a program (Heinonen & Fagerholm, 2023).

As the nature of how programs are expressed, developed, and understood is changing, theories and research on the cognitive aspects of programming should also evolve. Theories of programmers' mental models should encompass the concept of a program as an integrated system of multiple programs and consider how programmers conceptualize the external programs integrated into their own programs. Furthermore, programming theories need to address the cognitive aspects of using APIs. In this work, we will contribute to this endeavor by presenting results on programmers' mental models of APIs and the programs that provide them. We will also present findings on how these mental models are utilized in programming tasks.

In everyday discussions, different libraries, frameworks, packages, web services, and other pieces of software that provide an API are all referred to as APIs. However, this terminology does not allow for differentiation between the software that provides an API and the API itself. In this work, we will refer to software libraries, frameworks, services, and other software that provide functionality, data, or other resources that can be integrated and utilized in a new software system as *provider software*. We will refer to the functionality, data, and services that the provider software offers, which can be utilized in building new software, as *resources*. Finally, we will refer to the interface provided by the provider software that allows a program to utilize its resources as a *API*.

In this work, we aim to provide insight into programmers' mental models as they design and develop programs that utilize a provider software's resources using its API. We present results from a study where 12 professional programmers were interviewed using the critical incident method about a situation or situations where they had to learn to work with a new provider software. We analyze the mental models of the participants through their expressions of their understanding, through the problems they expressed having with understanding some aspect of the task and its context, and through the information they sought and used to complete the tasks. Through this analysis, we identified some key aspects of programmers' mental models of provider software. We also identified some key mechanisms of how these mental models are developed and how they are used in completing programming tasks.

2. Methodology

To conduct the study, we used the critical incident method interview protocol, designed to elicit information about cognitive performance in complex task settings (Marcella, Rowlands, & Baxter, 2013), which has been used successfully in similar studies (Votipka, Rabin, Micinski, Foster, & Mazurek, 2020).

We conducted 12 interviews in which we explored programmers' mental models of provider systems and their resources and APIs through detailed analyses of one or more recalled real-life events where

the participants had to learn about a new provider system.

2.1. Participants

A total of twelve participants took part in the study. Our participants were employed in different roles in academic and industry settings. Two participants held primarily academic positions, while ten were employed in software development companies or software development teams within academic or other organizations.

2.2. Interview protocol

During the interviews, participants were asked to choose an event in which they had to learn a new API and recall the process while we drew a diagram to visually represent it. Throughout the interviews, the interviewers asked further questions about multiple items of interest.

The interviews lasted approximately 60 minutes and were conducted remotely or in person, depending on the availability of the interviewee. The interviews were divided into three parts: background questions, event selection, and event walkthrough, as described subsequently.

2.2.1. Background questions

At the beginning of each interview, participants were asked about their education, programming background, and the level of experience they have with the technologies they currently work with. They were also asked about their current job and the tasks and responsibilities of their current role.

2.2.2. Event selection

After the background questions, the participants were asked if they could recall a time when they had to learn to use a new API. They were prompted to think about a memorable event, either recent or otherwise memorable to them. If a participant had difficulty recalling a suitable event, we asked further questions to assist in event selection.

2.2.3. Event walkthrough

After a suitable event had been selected, the participants were asked to recount what had happened during the event. As participants described the event, the lead interviewer drew a diagram of the process described by the participant. The participants could see the diagram at all times and were asked to notify us if the diagram did not match their story in some way.

While participants described the event, we asked directed questions intended to clarify some aspect of the event or gather further information about some important or interesting aspect of the event.

2.3. Data Analysis

We utilized iterative coding to analyze the data with the goal of examining programmers' mental models as they engaged in programming-related activities that required them to learn a new API.

In the first round of analysis, we coded items related to the programming projects the participants were undertaking and divided the interviews into activities, such as designing or implementing a solution.

In the second iteration, we analyzed each activity in detail. We coded statements about mental models, knowledge gaps, sensemaking activities, and resulting understanding. In this article, we will present results related to mental models. Our analysis of knowledge gaps and sensemaking activities will be presented in another publication. Statements about understanding or perception of some aspect of the task or its context were coded as mental models.

During the third iteration, we analyzed statements related to mental models to identify the aspects of the provider software, its resources and APIs, and the programming tasks that were represented in the mental models.

3. Results

In this section, we will discuss our results related to programmers' mental models of provider software and their resources and APIs. We will first examine programmers' mental models of provider software, followed by the mental models of programming tasks, API resources, and APIs used when implementing

a program that utilizes a provider software through its API.

3.1. Understanding provider software

A programmer's mental model of the provider software represents their understanding of the software — what it *is*, what it can be used for, and some relevant quality attributes such as usability and quality. We have divided the results into two categories. The first category, "What it is and what it can do," represents an understanding of the type of the provider software, its function, and functionality. "Quality, usability, and other relevant attributes" represent different non-functional characteristics of the provider software.

3.1.1. What it is and what it can do

These aspects of the provider software represent the programmer's understanding of what a provider software is and what it can be used for.

Type refers to the kind of provider software a specific software is — whether it's a library, a web service, or something else. Knowledge of the type of a provider software allows the programmer to utilize their background knowledge of other provider software of the same type to understand what the provider software is guiding the process of learning about and using a provider software.

As our participants were professional programmers, they had previous experience with different types of provider software. We refer to this knowledge as *provider software type schemata*, which represent different types of software, the functionality those types have, and how they can be used. For example, a schema of REST APIs suggests that they provide access to data in a database and can be used by sending HTTP requests to API endpoints.

Our participants discussed provider software as instances of types. The recognition of a software's type provided them with expectations of how it can be used, such as expecting that a REST API is used by sending HTTP requests or a library is first installed to the project and then used by calling API methods. It also guided the participants forward, providing expectations of what to do next and what information was needed. For example, one of our participants discussed finding a suitable provider software, and knowing that a library has to be installed, moved on to seeking information about how the specific provider software could be installed using Cradle.

Function refers to the general purpose of the provider software. This was discussed as the type of task that could be done with it, such as "draw plots" or "access data from a database." For example, one of our participants described PNPM as follows:

It is used to install react native...or of course you can use it to install anything. So a PNPM package manager built on top of NPM. And it can be used to install all kinds of JavaScript dependencies.

This understanding not only aids in selecting suitable provider software for the task at hand, but it also provides the programmer with expectations about it. There are often many provider software with the same function, and they have some similarity between them. We refer to the knowledge of these similar provider systems as *provider software category schemata*, which represent knowledge of provider software with the same function, such as what libraries that provide methods for making HTTP requests generally are like. Based on our interviews, participants used their provider software category schemata when encountering a new provider software with the same function. These schemata provided them with expectations about the software in question — its use, functionality, and even the naming of API methods:

... I could already guess the name of the method I could use. Because with these the naming of the methods is quite similar. They are usually always named the same way. For example something like findAll, findOne, findById. So the query abstractions are usually always named like that...

Functionality is the set of resources provided by the provider software. These resources include data, services, and implementations of behavior. Programmers' understanding of provider software functionality enables them to select a provider software suitable for the task at hand, as described by one of our participants who needed a provider software that provides tab components:

At that point I had read the MaterialUI documentation quite a bit. It has all kinds of examples, and I have scrolled the sidebar which lists all the components that it has, and I had considered using it [the tab component] previously but I had not previously needed tabs for anything.

However, programmers' understanding of a provider software's functionality does not always appear to be comprehensive or entirely accurate. For instance, one of our participants expected that a provider system for drawing plots would provide a way to combine the titles of subplots into one and were disappointed when they learned that it did not:

I was a bit annoyed that there was no automatic way to do it. You'd think that combining identical sub-plot titles would be a relatively common use case. So then...I expected someone would have made something automatic for it especially since it has so many other automatic features.

3.1.2. Quality, Usability and other relevant attributes

There are, of course, multitudes of attributes a software has, ranging from fault tolerance to availability. However, not all of these are relevant at all times. Quality and usability of a provider software were mentioned as important by multiple participants and are thus discussed in more detail below. Participants also mentioned other attributes when those were relevant for the specific task or project they were working on, indicating that participants considered the attributes that were relevant for them at that time. Therefore, we will not discuss all possible attributes but focus on the notion that the mental model seems to contain some information about the attributes deemed important for the task or project at hand.

Quality refers to the programmer's perception of the "goodness" or quality of the provider software. Our participants talked about forming a perception of whether the provider software was good or "valid" to use. For example, one participant discussed building a perception of a provider software's validity before selecting it for use:

They [libraries] are almost always open source, so I usually also check its validity as well. So I usually check the GitHub repository to see, for example, if it has been actively updated and if it has a lot of these...umm...these like stars which are kind of like "likes" and if it has a lot of forks and all those kinds of things. Like if it seems like it is used a lot and is like validated by the developer community so it is valid. And that can also peak my interest [in using a provider software] as well.

Usability refers to the programmer's perception of how easy the provider software is to use. When selecting suitable provider software, our participants discussed forming perceptions of the usability of the API. For example, one of our participants described a provider software they liked as "logical, easy to use, and easy to understand," while another participant discussed liking a provider software because it seemed easy to use and understand.

Other relevant attributes: Our participants did not seem to form an understanding of all possible attributes of a provider software, but rather only the attributes relevant to the task at hand. For example, one participant had to consider the performance of a provider software to design a component using it. However, this participant stated that they would not want to need to know this information, indicating they would not have gathered it if it was not necessary:

Well I would like to think about it logically, in other words so that I would just need to know how to use it. In this case I also had a bit of understanding of its technical side. We had to consider it, mostly its performance, and if it would cause any problems.

3.2. Use of a provider software for a specific task

In this section we will discuss the mental models related using a provider software to achieve a specific outcome. The outcome, of course, is most commonly a program that has a certain functionality.

The understanding related to using a provider software contains multiple aspects of the situation, including what is to be achieved, what is required from the provider software to achieve it, how the required resources are modeled in the provider software, and how those resources can be accessed. Below, we will illustrate this using an example from the interviews, and then we will describe the aspects of the mental models in more detail.

3.2.1. Example

One of our participants was implementing the backend for a mobile application:

So I had a specific application in mind already. It was a backend for an application that I made for a course. So for the students to use...so in the course the students make a client for the backend...so they have the backend ready but they have to make the mobile application client for it... So it was like an application for users to review GitHub repositories. So the user can log in and then write reviews for a GitHub repository.

They started to implement the backend working on one endpoint at a time. They start working on an endpoint that lists all the repositories. They know that the provider software they are using provides functionality for making requests to a database, and after forming an idea of what kind of a functionality they'd need for the endpoint, they start looking for a suitable method:

So I knew that I have...I knew that I have a database table where the repositories are, and it has certain fields. So then I started to think that if my endpoint has to list all of them, I started to think about what kind of a method I had to look for.

They browse the API documentation to find a method that seems promising, and read the method description to verify that the method indeed does what they need. They then use a code example in the documentation to write the client code that calls the method:

It [API documentation] has like API reference, so like the API in a more technical level. So I went there. And that was organized by the main themes, so there was like queries, so a section about how to make queries. And the methods were listed there. And from there I could spot a method that could be the right one. And then based on the method description I verified that it really was. And it also had code, they usually also have code examples showing how to actually do it.

3.2.2. Goals, tasks, solutions and resources

One of the aspects of the programmer's mental model is an understanding of what is to be implemented, and what is required from the API to implement it. This includes the program that is to be implemented, the part of the program the programmer is currently working on, how the part of the program can be implemented, and what is required from the provider software to implement the part.

Goal software: refers to the programmer's understanding of the program they are implementing. For example, in the previous example one participant described the backend application they were developing. Our participants described having different levels of understanding of the design of the goal software.

In some cases, they were working from formal design documents that provided them with a detailed understanding of the architecture of the goal software. In other cases, they did not describe the use of formal design documents or processes, but they did have a rather extensive understanding of the design.

So. We decided that we should make a new component which handles this part of it. So we made a new container for it. And in that, in regular intervals it fires up, and the idea is that it makes a request to the API and fetches... Or actually first it fetches a configuration file from DynamoDB which tells it what to fetch. It specifies what to fetch from where...And then based on the configurations it makes queries to the API and fetches all the information about the campaigns. And and so we know that first it fetches all the campaigns, and checks their timestamps to see if some of them have changed. And if they have, then it fetches information about those campaigns like product information- and then it updates the information to another DynamoDB table.

However, when participants were adding functionality to existing programs by integrating a service that provides the entire functionality, their understanding of the resulting integrated program was limited. This understanding was primarily shaped by their background knowledge of applications in the domain rather than their understanding of the specific provider software. For example, one participant had already begun integrating a service into an existing application when they encountered a problem that necessitated them to develop a deeper understanding of the system they were constructing. Their surprise at how the system works indicates they did not possess a robust mental model of it previously:

So. I hadn't previously like. So rarely some thing from your own code ever calls anything else. So all of our services work so, that a frontend always has one backend. And the frontend always talks to only its backend and then the backend may call some other service or do anything else. And in this case it was like "wait a second, it talks with something else". Our frontend sends requests to it kind of like google analytics...So like after I got it it was more clear that "hey this is what we are doing and this is what it is all about". endquote

Task: refers to the programmer's understanding of the specific part of the goal software they are currently working on. When the program is small, the entire program could be the task. However, when the program is larger, it is split into parts, and the programmer works on them one at a time.

In many of our interviews, it was not clear how the participants identified the task at hand. However, participants who discussed tasks related to setting up the provider software could often provide more detail on how the tasks were identified. In some cases, participants identified the task based on their experience with the type of provider software. For example, one participant was using a library. Drawing from their familiarity with other libraries, they recognized they had to add the library to their project and moved on to read the library documentation to learn how to do so. In cases where the participant was integrating a service to an existing program or when a library required more extensive setup, participants also mentioned reading documentation to learn what had to be done.

Solution: refers to the programmer's mental model of the program that fulfills the task. Some of our participants described the solution in functional terms - what the program should do.

Other participants described the solution in terms of programming concepts or patterns that could be used to achieve the required functionality. They discussed knowing, for example, that to create a program that fetches specific data from a database they had to make a HTTP GET request, or that to create a program that makes multiple HTTP requests in parallel they should create a new threads. For example one participant was implementing a program, that allowed users to authenticate to a web-service. The participant was working on keeping the

users logged in, and based on their domain knowledge and knowledge of the service, knew that they should use refresh tokens to implement the functionality:

The situation was, that the authentication worked using OAuth, which means it was token-based. And with tokens it is essential that, for example, you stay logged in. So you don't have to log in again after like an hour when the token expires. And for that you need a system called refresh token.

Some participants discussed how they identified the concept(s) that were required to implement the solution. For example one of our participants was implementing a modification to an existing web application. They knew that the program should allow the user to switch between two views, and it should be easy for the user to see that there are two views available. They then spend some time figuring out the solution:

I did not know exactly how I wanted to do it. With React it is really easy to switch between components inplace. You just use a conditional statement to remove one and then change the state to show another one. And then switch back. Like a toggle. But I thought that way it would not be so easy for the users to see that there are multiple views there. There at the same place. So then I thought that the other view actually already has tabs for different files, so we could make another level of tabs.

In cases where the participant was already familiar with the provider software, they discussed the solution in terms of the provider software.

Resources: refer to programmer's understanding of what is required from a provider software to implement the solution. Some participants discussed the resources in terms of programming concepts that they expected a provider software would provide the tools to implement. One example of this is the participant who wanted to implement tabs described above. Other participants discussed needing the implementation of a specific behavior, for example fetching all rows of a database table. Participants who were writing a program that interacts with a service, the resources were described in terms of provider software functionality that the programmer wanted to use, such as authentication to the service.

Some of our participants were using provider software to access data, and knew the data that they needed for their programs, such as one participant who was making a program that checks the age of an user account, and knew they needed a datamodel that contained the age of a specific user account:

Basically we knew the datamodel...We were trying to do when we upgrade the subscription. So basically what happens, we usually the company will draw out the accounts, and sometime the person who is upgrading to the new service is using ten years old account. And then we need to upgrade it to the new account because that is too old... This is the thing which was, we need to calculate specific years...

The division between solutions and resources is not always clear. Some participants first designed the solution and identified resources they needed from a provider software, and then moved on to figure out which provider software provides the required resources and how to use it to add those resources to their program. Some participants had already selected a provider software to use, and as they designed their solution they did so based on their understanding of the provider software's functionality, so the solution was designed based on how it could be done using the specific provider software they had in mind.

3.2.3. API-translated solution model and its implementation

Another aspect of the mental model is an understanding how to use the API of the provider software to utilize specific resources. This understanding can be roughly divided into two categories: Understanding how the resources are modeled in the API, and understanding how to implement the code that interacts with the API to access the resources.

API-translated solution model: represents the programmer's understanding of how the resources they want to use are modeled in the API. In other words, the API-translated solution model describes the solution in terms of API elements, API tasks, or when it comes to data resources, datamodels. Some participants described the API-translated solution model in terms of *API tasks*. With API task we refer to the set of tasks the client code has to perform in regards of the API. For example, one of our participants was using a provider software that implements a functionality that draws plots. The task they were working on was writing the code that draws a title to the center of the plot. To do so, they first had to figure out how this behavior is modeled in the API. Using StackOverflow they learned that the client code has to first instruct the provider software to remove the titles from all the sub-plots, and then add text to specific coordinates within the plot:

So you remove from the Seaborn...make it so that it does not make the sub-titles or titles for the sub-plots. And then you just manually write text to it [the plot] through matplotlib using those like coordinates. So you just manually add text there and give it its orientation and coordinates.

Some participants described the API-translated solution model in terms of API elements and their relationships:

The tab API consisted of a tabs component, and you place tab components within it and give each tab an index. And tabs receives a state I think. And then there is also a tab panel component which is placed within a tab. And then a tab panel is shown according to the state.

When the resources are data, the provider software models information about different entities as datamodels, and the API-translated solution model refers to the datamodels and their attributes and relationships that represent information about a specific entity. For example one participant was working on a program that writes specific data to a provider software. They knew the entity they wanted to write information about. However, to write the program they had to first figure out how the entity was represented in the provider software:

So our department started to write the information about [entity] into the [provider software] using our own system...So we just checked from the API catalogue that here is this datamodel called [entity] and they can be created using this endpoint here...And then we checked what we needed for the datamodel. So it needs references to three different things. And we wondered how we were supposed to get the references from? So we do have the three other datamodels, but how are we supposed to get the right instances?...At that point it was unclear why in the world of [provider software] the information about [entity] is split into three datamodels and what the relationships between the models are...So we did not know what the three references actually mean. And then when we were told that you have to use this [another datamodel] then it started to make sense like that one had all the information we need so we can start by saying take this one first. At that point the whole thing did not seem so difficult and confusing anymore.

Implementation model: refers to the implementation details of the client code for API tasks, elements, and datamodels. This includes the syntax of the client code as well as their

parameters. This may also include the syntax of configurations or other code that has to be written to implement the solution. For example, one participant described seeking for information about the data type of parameters:

I checked the tab panel and the tab, the individual tabs, what I have to give them, to make sure I give them the right index, or like what kind of index they need. That was not explained in the code example page.

Participants often described searching for and reading code examples to figure out how to use provider software. In these cases the information about the API-translated solution model and the implementation model were acquired simultaneously. In other cases, participants described learning the API-translated solution model first, and then moving on to seek information about the correct syntax to implement it.

4. Discussion

In this paper, we present the results of a study where we used the critical incident interview technique to gain insight into programmers' mental models of provider software, their resources and their APIs. Our results show, that as programmers work with provider software, they form a mental model of the provider software, that represents their understanding of the provider software as a software artifact — its function and functionality, the way it works and is used in a program, and its quality, usability, and other attributes relevant to the situation.

This mental model provides the necessary framework for utilizing the provider software's resources in a program. As programmers integrate provider software's resources into a program through its API, their mental models of the provider software's type, function, functionality and use guide the process of identifying resources required to solve a programming problem, and figuring out how the resources are modeled in the software and how to integrate the them into a program. This result corresponds with the idea of initial API mental models, that represent programmer's understanding of a provider system they form before they begin using the provider software in programming tasks (Heinonen & Fagerholm, 2023). These mental models guide the programmer's actions, providing them with understanding of what the provider software can be used for and how the provider software can be used, as well as what information is required to use it and what should be done to begin using the provider software (Heinonen & Fagerholm, 2023).

When it comes to program design, our results show that the process is in most cases quite similar to the models of program design that have been previously proposed. Existing theories of program design describe program design as a process of decomposing a problem into manageable sub-problems that can then be broken down further and solved (Atwood, Jeffries, Turner, Polson, & CO, 1980; Pennington & Grabowski, 1990). We see the same behavior when designing solutions that use provider software. Our results show programmers breaking the goal program down into manageable tasks, that can then be solved one by one. However, in cases where programmers integrate services into existing programs or use libraries that require extensive setups, this decomposition of the problem is different. In these cases the programmer does not necessarily have a deep understanding of the resulting program, and thus use documentation to gather the information about the tasks required to implement it.

When writing code, traditionally, we would expect the programmer to understand what is to be implemented, how the required functionality can be achieved in computing terms (Pennington & Grabowski, 1990). We would also expect the programmer to have an understanding of the programming language used to write the code to implement the solution (Hoc, 1977). When using provider software, not only does the programmer require an understanding of the programming language used to write the client code, but also the

API resources, tasks and their syntax. We see programmers forming an understanding of what is to be implemented as a high-level understanding of the entities in terms of behavior and outcomes. Understanding how those entities are modeled in the API of the specific provider system and the syntax of the API methods is then used to integrate those entities to a program.

Therefore the understanding required to create programs using provider software requires understanding of the programming language used to write the client code and the parts of the overall program that do not use APIs with its syntax, grammar, rules of discourse and other conventions. It also requires understanding of the provider software and the API, which have their own design, syntax, rules of discourse, and other conventions — a mental model of the API and its language.

5. Conclusions

Writing about programming, Pennington and Grabowski stated "Programming problems are unique in that they usually involve solving a problem in another (application) problem domain, such as mathematics, accounting, electronics, or physics, in addition to solving the program design problem" (Pennington & Grabowski, 1990). Now, we can add a third level to the complexity of programming. Programming using provider software requires the programmer to solve the problem in the problem domain and solve the program design problem in terms of the structure and behavior of the program that solves the problem. It also requires the programmer to solve the problems of identifying and selecting provider software that provide the required resources to implement the designed program, and writing the client code to interact with the API of a provider software so that the provider software behaves as required.

6. References

- Andrews, A., Ghosh, S., & Choi, E. M. (2002). A model for understanding software components. In *International conference on software maintenance, 2002. proceedings.* (pp. 359–368).
- Atwood, M. E., Jeffries, R., Turner, A. A., Polson, P. G., & CO, S. A. I. E. (1980). The processes involved in designing software. *NTIS, SPRINGFIELD, VA, 1980, 62.*
- Chen, X., He, J., Liu, Z., & Zhan, N. (2007). A model of component-based programming. In *International symposium on fundamentals of software engineering: International symposium, fsen 2007, tehran, iran, april 17-19, 2007. proceedings* (pp. 191–206).
- Détienne, F., & Détienne, F. (2002a). Software reuse. *Software Design—Cognitive Aspects*, 43–55.
- Détienne, F., & Détienne, F. (2002b). What is a computer program? *Software Design—Cognitive Aspects*, 13–20.
- Heinonen, A., & Fagerholm, F. (2023). Understanding initial api comprehension. In *2023 ieee/acm 31st international conference on program comprehension (icpc)* (pp. 43–53).
- Heinonen, A., Lehtelä, B., Hellas, A., & Fagerholm, F. (2023). Synthesizing research on programmers’ mental models of programs, tasks and concepts—a systematic literature review. *Information and Software Technology*, 107300.
- Hoc, J.-M. (1977). Role of mental representation in learning a programming language. *International Journal of Man-Machine Studies*, 9(1), 87–105.
- Kim, J., Lerch, F. J., & Simon, H. A. (1995). Internal representation and rule development in object-oriented design. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2(4), 357–390.
- Mäkitalo, N., Taivalsaari, A., Kiviluoto, A., Mikkonen, T., & Capilla, R. (2020). On opportunistic software reuse. *Computing*, 102, 2385–2408.
- Marcella, R., Rowlands, H., & Baxter, G. (2013). The critical incident technique as a tool for gathering data as part of a qualitative study of information seeking behaviour. In *Leading issues in business research methods* (Vol. 2). Academic Conferences and Publishing.
- Mosqueira-Rey, E., Alonso-Ríos, D., Moret-Bonillo, V., Fernández-Varela, I., & Álvarez-Estévez, D. (2018). A systematic approach to api usability: Taxonomy-derived criteria and a case study. *Information and Software Technology*, 97, 46–63.
- Pair, C. (1990). Programming, programming languages and programming methods. In *Psychology of programming* (pp. 9–19). Elsevier.
- Pennington, N., & Grabowski, B. (1990). The tasks of programming. In *Psychology of programming* (pp. 45–62). Elsevier.
- Thayer, K., Chasins, S. E., & Ko, A. J. (2021). A theory of robust api knowledge. *ACM Transactions on Computing Education (TOCE)*, 21(1), 1–32.
- Votipka, D., Rabin, S., Micinski, K., Foster, J. S., & Mazurek, M. L. (2020). An observational investigation of reverse engineers’ processes. In *29th usenix security symposium (usenix security 20)* (pp. 1875–1892).

Analysing Open Source Software to Better Understand Long Term Memory Structures in the Human Brain.

Thomas Mullen
tom@tom-mullen.com

Abstract

As AI models become larger, replicating long term memory structures (LTM-S) may produce the same benefits that evolution provided the human brain (efficiency, performance, and extensibility).

At the heart of this paper is the conjecture that software structures are close representations of LTM-S. If this is true, then open source can be considered a huge database of easily searchable LTM-S examples that could assist in a deeper understanding of the same.

The paper proposes a general refactoring algorithm based around two elements of LTM-S, chunks and analogies. The underlying aim is to develop mechanisms and theories to analyse the analogical and chunking structures employed in software.

1. Introduction

The processes of introducing a required behaviour to a code base involves translating the requirements into dependencies amongst a set of variables. Even when a successful dependency graph is identified, software languages allow many structural choices of how to represent that graph. Design principles then provide the guidance to establish which alternative is easiest to understand.

There may be background and context that form part of this process. However, this paper looks specifically at the relationship between the dependency graph and the software structures that the developer alights upon. At the heart is the conjecture that those structures are close representations of the LTM-S of the dependency graph in the developer's brain.

The conjecture that *code structures are representations of long-term memory structures* was stated in Mullen (2009) in a slightly different form. This paper attempts to further validate this conjecture by proposing a refactoring algorithm which is based on elements of LTM-S (specifically, chunking and analogies).

The refactoring algorithm follows this process and has three elements:

- Software Chromatography. The code to be analysed is distilled into two graphs. Both graphs have shared nodes which represent the inherent variables/values/attributes in the code to be refactored. The first graph is a normalised dependency graph (which is inclusive of conditionals) and the second is a structural graph that identifies where each node is represented in the software language elements.
- Candidate Solutions Library. A selection of dependency sub-graphs, each of which is mapped to a number of alternative structural graph solutions.
- Chunking Objective Function. A measurement of simplicity, a function comprised of coupling and cohesion parts.

A more detailed description of the algorithm is provided in section 3.2, but the precis is that the code base to be analysed is firstly distilled into the dependency and structure graphs. Each dependency sub-graph in the library is searched for in the distilled dependency graph. Where a match is found each alternative structural solution is then employed in the distilled structural graph and the change to the chunking objective function is calculated. If the objective function is improved than that refactor becomes a recommendation.

The advantages of LTM-S that evolution has provided the human brain would likely benefit AI models. Webb et al (2023) showed that “[GPT-3] appears to display an emergent ability to reason by analogy” so it's possible that the training processes already produces them. However, a better understanding of

LTM-S could assist in recognising analogical structures in AI models and refine the training process to produce them. For example, Holyoak and Thagard (1997) propose that analogies are chosen based on similarity, structure and purpose. Gentner (1983) proposes that the depth of the behaviour function is a driver. Analysing which abstractions/analogies are chosen to be represented by classes in open-source code may help to fill in specific details of the process in choosing primary analogies.

2. Software Structures are LTM-S Conjecture

2.1 LTM-S in Languages

Whenever we understand anything, the endgame is to lay down long term memories. Only then can we utilise that knowledge, build upon it and apply it to other domains (to expand understanding in those areas). If we had a choice of how to represent something and the primary aim was to make it easier to understand, then the closer our representation gets to LTM-S, the less translation is needed when we (or others) try to understand the knowledge it represents.

The same argument could be applied to natural languages. For example, if you needed to explain something to me (especially knowledge that it took time for you to acquire). You would first access the LTM-S in your brain that contains that knowledge. You would then serialize those structures as natural language. I would deserialize those sentences and (hopefully) produce the same LTM-S that exist in your mind. In that way I could achieve the same knowledge without the cost you had to achieve it. Indeed, cognitive linguists analyse natural languages to see if they betray how the mind works.

If natural languages are the serialization of LTM-S, then it could be said that software languages provide a direct access equivalent. So, from a 30,000 foot point of view, the conjecture that *software structures are long term memory structures* makes sense.

It's probably inevitable that at some point AI models will communicate with one another to pass on knowledge without the necessity of repeating costly training. If AI models employ the same LTM-S as in a human brain, then the languages they will use will likely be based upon those structures (potentially a massive parallel version). The more we understand about LTM-S, the better our ability to eavesdrop or, more usefully, influence that knowledge transfer (especially to correct bias or error).

2.2 Chunking Analogies

If we consider the aspect of software design that simplifies the code structure without changing the behaviour. Then this is the same process as laying down long term memories, i.e. chunking analogies (Mullen, 2009). Specifically,

- Chunking – where memory elements are grouped so that elements within a chunk are strongly related to each other, but loosely related to elements in other chunks. The design principle of low coupling/high cohesion guides the developer to chunk the code.
- Analogies – a mapping of two or more domains that contains attributes and behaviours. This is similar to the class description of the OO paradigm. However, there are other software language structures that can represent analogies (Mullen, 2009).

Software design is a process of identifying the different language structures we can employ to represent the abstractions/analogies of the required behaviour and then choosing whichever provides the best coupling and cohesion (chunking). The similarity between the cognitive psychologists' description of LTM-S and software design principles and languages adds further validation to the *software structures are LTM-S* conjecture.

2.3 So What?

Even if we are prepared to accept at this stage that the conjecture is true, how does that help us?

Evolution has provided the human brain with LTM-S that produce an efficient, performant, and extensible knowledge store. These structures could be just as advantageous to AI models. Understanding LTM-S and the process that creates them would be key to recognising these structures in AI models or guiding the training process to produce them.

If the *software structures are LTM-S* conjecture is true, then open-source software could provide us with a huge searchable database of LTM-S examples. We could analyse these structures with the same aim that cognitive linguists analyse natural languages.

The next section proposes a general refactoring algorithm. It is not expected that this will produce a usable refactoring tool for developers. However, if it can come up with reasonable (or, hopefully, illuminating) refactoring suggestions then that could provide a more detailed validation of the conjecture. In addition, the mechanisms and theories needed for the algorithm could be employed to analyse open source. This would provide a large enough dataset for correlations to be identified and provide further evidence of the links between software structures and LTM-S.

3. Analysing Analogical and Chunking Structures In Open Source Code

Before going into the details of the algorithm it may be beneficial to provide an example of the type of refactoring that the tool is intended to deliver. This is included in the next section with the description of the algorithm following it.

3.1 Example

The three code examples in Fig 1 represent the same behaviour with respect to the relationship between *z* and its constituent elements *a1*, *a2*, *a3*, *b1*, *b2* and *b3*. In all the examples, the same labels are used (*z*, *a1*, etc.) but in real code they would be different labels and may have additional dependencies amongst them, and with other elements. The proposal is to represent the dependency relationship in a normalised graph so that the isomorphic nature can be easily identified.

```

public class JustConditional{
    int a1;
    int a2;
    int a3;
    boolean b1;
    boolean b2;
    boolean b3;

    int z;

    int calc() {
        if(b1) {
            z=a1;
        }

        if(b2) {
            if(b3) {
                z=a2;
            } else {
                z=a3;
            }
        }
        return z;
    }
}

class ConditionalWithMethod{
    int a1;
    int a2;
    int a3;
    boolean b1;
    boolean b2;
    boolean b3;

    int z;

    int calc() {
        if(b1) {
            z=a1;
        }

        return getValue(z);
    }

    private int getValue(int z) {
        if(b2) {
            if(b3) {
                return a2;
            } else {
                return a3;
            }
        }
        return z;
    }
}

abstract class B3 {
    abstract int value();
}

class B3_TRUE extends B3 {
    int a2;
    int value() {
        return a2;
    }
}

class B3_FALSE extends B3 {
    int a3;
    int value() {
        return a3;
    }
}

class OOHierarchy
{
    int a1;
    B3 valForB3;
    boolean b1;
    boolean b2;

    int z;

    int calc() {
        if(b1) {
            z=a1;
        }

        if(b2) {
            z= valForB3.value();
        }
        return z;
    }
}

```

Figure 1 – Three different code structures that provide the same behaviour between *z* and its dependencies *a1*, *a2*, *a3*, *b1*, *b2*, *b3*.

Fig 2 shows the resulting dependency graph for all the examples in Fig 1. There are three main composite elements to the graph;- conditional elements; pure functional sub-graphs; and assignments. This is discussed in more detail in Appendix A.

In addition to the dependency graph, a partner structural graph details how the different elements are represented in the language. This contains the same variable and expression nodes that are in the dependency graph but are placed in the package/class/method structure where they are defined in the code. Fig 3 shows the structure graph for the middle code snippet in Fig 1.

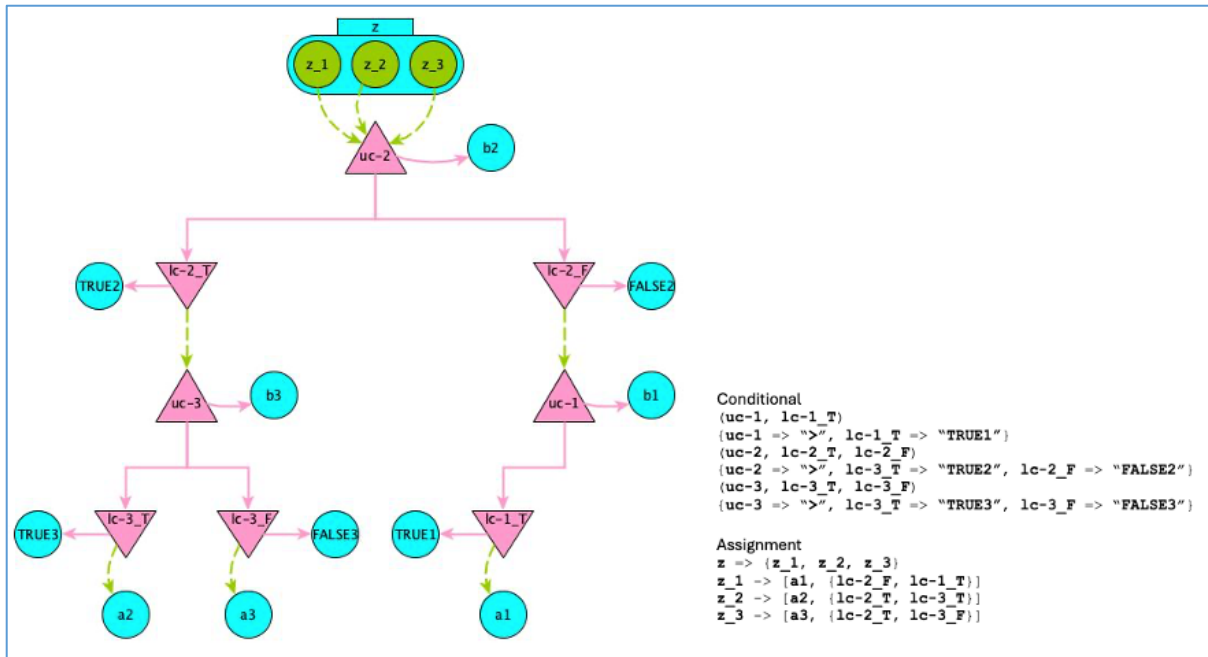


Figure 2 – Normalised dependency graph for all the three code examples in Fig 1

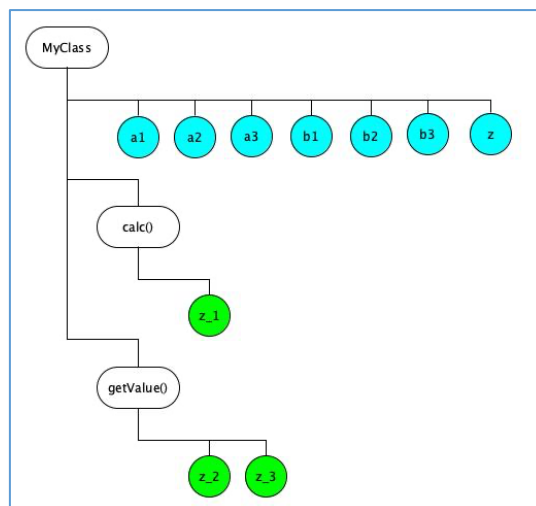


Figure 3 – Structure graph for the second code example in Fig 1

To illustrate a refactoring example, consider the code in the left-hand column of Fig 4, for which the dependency graph is in Fig 5. There is a subgraph isomorphism with the normalised graph of the three code examples in Fig 2. Specifically with the mapping;- b1-> (value==0); b2 -> (lowerValue<=upperValue); b3 -> (lowerValue <= input && value <= upperValue); a1 -> ZERO; a2 -> IN_RANGE; a3 -> NOT_IN_RANGE; z_1 -> z_ZERO; z_2 -> z_IN ;and z_3-> z_NOT. Consequently, we could employ the structure associated with any of the three examples in Fig 1. The second structure in Fig 1 applied to this code would produce something like that in the right-hand column of Fig 4.

```

enum RANGE_STATUS {
    ERROR, ZERO, IN_RANGE, NOT_IN_RANGE;
}

class EnumExample{
    RANGE_STATUS calc() {
        RANGE_STATUS rangeStatus = RANGE_STATUS.ERROR;

        if(value == 0) {
            rangeStatus=RANGE_STATUS.ZERO;
        }

        if(lowerValue<=upperValue) { //valid range
            if(lowerValue <= value && value <= upperValue){
                rangeStatus=RANGE_STATUS.IN_RANGE;
            } else {
                rangeStatus=RANGE_STATUS.NOT_IN_RANGE;
            }
        }
        return rangeStatus;
    }
}

RANGE_STATUS calc() {
    RANGE_STATUS rangeStatus = RANGE_STATUS.ERROR;

    if(value == 0) {
        rangeStatus=RANGE_STATUS.ZERO;
    }

    rangeStatus = getValue(rangeStatus);
    return rangeStatus;
}

private RANGE_STATUS getValue(RANGE_STATUS rangeStatus)
{
    if(lowerValue<=upperValue) { //valid range
        if(lowerValue <= value && value <= upperValue) {
            rangeStatus=RANGE_STATUS.IN_RANGE;
        } else {
            rangeStatus=RANGE_STATUS.NOT_IN_RANGE;
        }
    }
    return rangeStatus;
}
    
```

Figure 4 – Example refactor

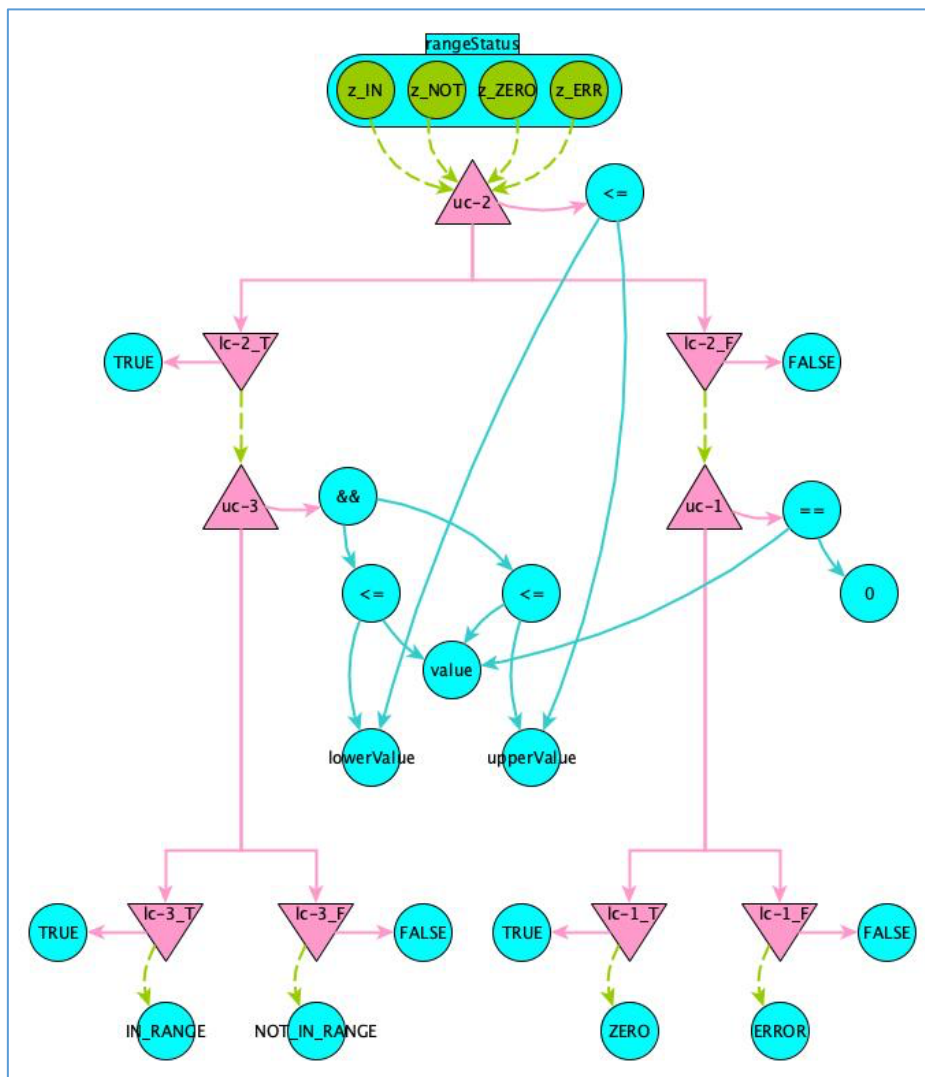


Figure 5 –Normalised dependency graph for the code example in Fig 4

3.2 Proposed Algorithm

The refactoring algorithm has three elements

- Software Chromatography. Where the code to be analysed is distilled into the dependency and structural graphs.
- Candidate Solutions Library. A selection of dependency sub-graphs and their potential structural graph solutions. An example entry in the library would be the dependency graph shown in Fig 2 paired with the three alternative structural graphs that represent the code examples in Fig 1.
- Simplicity objective function. A single value function to measure the simplicity of the solution. The proposal is to use a chunking penalty function (made up from coupling and cohesion penalty functions). This is detailed in Appendix B, but is comprised of distance functions on the structural and dependency graphs.

Definitions:

F_0 the dependency graph of the code to be analysed.

S_0 the structural graph of the code to be analysed.

G (with elements g_i) the set of dependency sub-graphs in the candidate solutions library.

T_i (with elements t_{ij}) the set of alternative structural graphs for the dependency sub-graph g_i .

The steps of the algorithm are as follows:

1. Distil the code that is to be analysed and produce the dependency and structural graphs F_0 and S_0 .
2. For Each dependency sub-graph g_i in the candidate solutions library, G :
 - a. Search for an isomorphism of g_i in F_0 .
 - b. For each finding of the candidate sub-graph, loop through the alternative structural graph solutions t_{ij} that are stored with g_i in the candidate solutions library.
 - i. Calculate the change to the chunking penalty function when S_0 is modified to employ the alternative structure t_{ij} . If this is an improvement, then the refactor of t_{ij} to S_0 is presented as a recommendation.

One question remains as to how we build the candidate solutions library. This could be achieved by performing the distillation process across open-source code and identifying dependency sub-graphs using a clustering algorithm. Further filtering could be employed based on the number of alternative structural solutions for a dependency sub-graph. It is expected that a similar process would be useful when we attempt to analyse open source for LTM-S.

3.2 Short Term Memory Capacity Limit

The capacity limit of short-term memory (Miller, 1956) has a significant influence on our ability to understand. It would be natural to assume that any algorithm which represents human understanding should incorporate those limits. The proposed algorithm doesn't explicitly mention short-term memory capacity limits but does model them implicitly in two ways.

Firstly, the candidate solutions library contains structures from existing code. A reasonable assumption is that at least one person (the developer of that code) must have understood it. Since short-term memory is the gateway to long term memory then the structures in the library have successfully passed through someone's short term memory and so, taken in isolation, are within the capacity limits.

Secondly, there is an argument that a function which could describe the limit is likely to be combinatorial in nature. As more and more elements are added to short term memory the possibility of cognitive overload becomes non-linearly more likely. The cohesion penalty function is made up from pairs of siblings in the structural graph, which is $O(n^2)$. This may or may not be sufficient to model the capacity limit so could need to be revisited.

4. Conclusion

This paper proposes a mechanism to distil software into two graphs, dependency and structural. The dependency graph is normalised to facilitate recognising isomorphisms of behaviour between different structural implementations. The intention is that this could be useful for the following:

- Better understanding of LTM-S. If we accept the conjecture that *code structures are direct representations of long-term memory*, the open source would be a huge searchable database of LTM-S examples. The ability to recognise code structures that are isomorphic in their behaviour would be key to this search. Software structures are not necessarily the only elements that could be mapped to LTM-S. For example, the same arguments could apply to architectural designs. However, software structures are likely to be more useful, due to the large dataset available with open source and the ability to identify dependency graphs and their associated structural representations.
- Refactoring tool. This paper proposes a work-in-progress algorithm for a general refactoring tool. Analysing open source to build up a library of refactoring candidates and choosing recommendations only when a simplicity function (based on cognitive principles) is improved.
- Code stored as a dependency graph. Most languages in use today require that code is stored in flat files. The files often fit a purpose for defining primary abstractions/analogs or for chunking/cohesion considerations. If the refactoring algorithm, above, could be proved successful then there may be an opportunity for storing the normalised dependency graph. As developers initiate searches on the code, the best structure to represent the dependency sub-graph of the search result would be generated. In most cases, this would likely produce the same results as present in the flat file solution. However, it may allow for a cleaner representation of cross-cutting concerns, as well as reacting to changes to the dependency graph that would otherwise require a significant refactoring.

5. References

- M. Fowler, K. Beck, J. Brant, and W. Opdyke (1999). Refactoring: Improving the Design of Existing Code. Addison- Wesley ISBN 0-201-48567-2
- D. Gentner (1983). Structure-mapping: A theoretical framework for analogy. In *Cognitive Science*, 7, pp 155-170.
- K. J. Holyoak and P. Thagard. The Analogical Mind (1997). <http://cogsci.uwaterloo.ca/Articles/Pages/Analog.Mind.html>
- 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. CiteSeerX 10.1.1.308.8071. doi:10.1037/h0043158. hdl:11858/00-001M-0000-002C-4646-B. PMID 13310704. S2CID 15654531.
- Mullen, T. 2009. Writing code for other people: cognitive psychology and the fundamentals of good software design principles. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 481–492. <https://doi.org/10.1145/1640089.1640126>
- J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42. <https://doi.org/10.1145/321921.321925>
- Taylor Webb, Keith J Holyoak, and Hongjing Lu (2023). Emergent analogical reasoning in large language models. Published at *Nature Human Behaviour* (2023) <https://doi.org/10.1038/s41562-023-01659-w>.

Appendix A : Normalised Dependency Graph

The normalised dependency graph consists of connected parts of three different element constructs. All nodes in the graph have a unique identifier, however for the sake of brevity the examples given here sometimes use the label associated with that element. If we take a simple max function as an example:

```
int max(int i, int j) {
    if (i > j) {
        return i;
    } else {
        return j;
    }
}
```

The normalised dependency graph is in Fig A.1

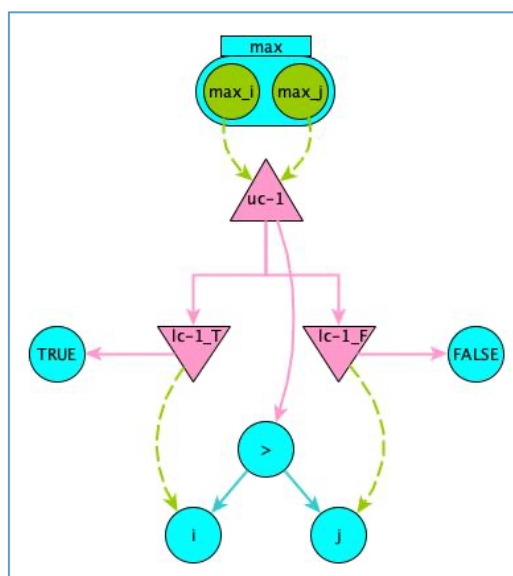


Figure A.1 – Normalised dependency graph for the max function.

The three constructs are

- **Conditional** – consists of the following nodes
 - An upper conditional node (triangle shape pointing up), labelled uc-1 in the max example.
 - Two or more lower conditional nodes (triangle shape pointing down), labelled lc-1_T and lc-1_F in the max example.
 - For conditional statements there will be two lower conditional nodes (representing the true and false cases).
 - Class hierarchies will also be represented by conditional structures with a lower conditional node for each class that inherits from a super class or interface.
 - Switch/case statements will have a lower conditional node for each case element.

The edges in the conditional construct are:

- Conditional edges (between the upper conditional node and each lower conditional node)
- An upper conditional value edge and lower conditional value edges (one for each lower conditional node). This is easiest understood by looking at the example where it represents the conditional statement. The value from the upper conditional edge is the boolean expression and the lower conditional edges point to TRUE and FALSE. So for a runtime case the upper conditional value is compared to the lower conditional values to determine which lower conditional branch represents the dependency in that case.
- **Functional** – a sub graph that consists of variables, literal values and operands that combine them. The $i > j$ construct in the graph is an example of this.

- **Assignment** – a link between a variable and a functional construct that also contains the lower conditional nodes that must all be satisfied for that dependency. There may be many assignments for a variable (some of which point to the same functional construct). The effective boolean expressions are in disjunctive normal form.

A.1 Shorthand notation

The graph can be described by the following shorthand notation that represent the edges. In each case the examples given are the shorthand notation for the graph of the max function, given in fig x.

- **Conditional** – this consists of two parts.
 - Firstly, an upper conditional node with a set of lower conditional nodes e.g.
 $(\mathbf{uc-1}, \mathbf{lc-1_T}, \mathbf{lc-1_F})$
 This represents the edges between the upper and lower conditional nodes e.g.
 $\{\mathbf{uc-1} \Rightarrow \mathbf{lc-1_T}, \mathbf{uc-1} \Rightarrow \mathbf{lc-1_F}\}$
 - Secondly, the edges between the conditional nodes and the conditional values
 $\{\mathbf{uc-1} \Rightarrow \mathbf{>}, \mathbf{lc-1_T} \Rightarrow \mathbf{TRUE}, \mathbf{lc-1_F} \Rightarrow \mathbf{FALSE}\}$
- **Functional** – simply the edges in the functional graph. The example is below. However a functional label, $f(i,j)$, is provided for reasons that will become apparent in section A.2 (Establishing Isomorphisms)
 $f(i, j) \rightarrow \{\mathbf{>} \Rightarrow \mathbf{i}, \mathbf{>} \Rightarrow \mathbf{j}\}$
- **Assignment** – consisting of two parts.
 - The first is the set of assignments for a variable, e.g.
 $\mathbf{max} \Rightarrow \{\mathbf{max_i}, \mathbf{max_j}\}$
 - the second defines each of these assignments using the value and the set of lower conditionals, e.g.
 $\mathbf{max_i} \rightarrow [\mathbf{i}, \{\mathbf{lc-1_T}\}]$
 $\mathbf{max_j} \rightarrow [\mathbf{j}, \{\mathbf{lc-1_F}\}]$

A.2 Establishing Isomorphisms

The candidate solutions library contains dependency subgraphs that we need to search for in the dependency graph of the code we are analysing. The intention is to use Ullman’s (1976) sub-graph isomorphism algorithm. The number of different node and edge types will be leveraged in the refinement step to aid in the process (e.g. upper conditional nodes will only be mapped to upper conditional nodes).

There will also be different flavours of isomorphisms. This would include:

1. Assignment value as function. In this case the functional constructs are collapsed to a single function node ($f(i, j, \dots)$) as mentioned in the functional shorthand notation in section B.1). The choice would then to be whether we include the function parameter list in the isomorphism. Whilst including the parameter list may lead to a more targeted mapping, not including the list could open up to more refactoring recommendations (with the belief that the chunking objective function would reject many unsuitable refactorings).
2. Pure functional: where the dependency sub-graph would contain a single functional construct. In this case the algorithm would identify refactors such as “Extract Method” and “Introduce Variable” (Fowler et al, 1999).

Appendix B : Chunking Penalty Function

The chunking penalty function consists of two parts;- coupling and cohesion. Both these parts rely on distance functions on the dependency and structural graphs. The initial proposal for these functions is given but analysis of open source may help to refine these.

- Coupling: a reduced form of the dependency graph is used that strips out all the non-variable nodes (but retains any dependency paths). The edges then provide the direct dependencies amongst the variables. Each edge is taken and the penalty is how far away the two nodes are on the structural graph (the further they are away the worse the cohesion). A simple distance function is initially proposed. However, edge weights or a function may need to be applied.
- Cohesion: this applies to elements that are structurally grouped together. Sibling nodes on the structural graph are processed and, for each pair, a metric of the similarity of the pair is used. This is achieved by using a vector that contains the distance of a node (on the dependency graph) to all other nodes. The cosine distance is then used to provide a measure of dissimilarity.

Definitions:

s_{ij} – the distance between nodes i and j in the structural graph.

d_i – a vector giving the distance of node i with each other node on the dependency graph.

$$\text{chunking penalty} = \sum_{\substack{i,j \text{ nodes of an} \\ \text{edge in the reduced} \\ \text{dependency graph}}} s_{ij} + \sum_{\substack{i,j \text{ sibling} \\ \text{nodes in the} \\ \text{structural graph}}} \left(1 - \frac{d_i \cdot d_j}{|d_i| |d_j|} \right)$$

Designing a didactic model for programs and data structures

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

Several authors affirm with solid arguments that it is essential to educate in computing, at least from secondary education and covering undergraduate courses, and that this continues to be a pending problem in most educational systems. Some point to the relationship between research and educational practice, which partly arises from the undervalued role of didactic research within the academy. Based on our epistemological model and taking as starting point fundamental ideas of computing, we began to develop a didactic model for the development of computational competencies and skills for novice students. In this paper we present the rationale of the proposed didactic model, a description of and empirical study and a preliminary analysis of the results of the experience.

1. Introduction

Several authors affirm with solid arguments that it is essential to educate in computing, at least from secondary education and covering undergraduate courses. These arguments provide answers to the *why* and *for whom* (to teach computer science) of the didactic questions (Saeli, Perrenet, Jochems, & Zwanveld, 2011). The authors add that this continues to be a pending problem in most educational systems (Denning & Tedre, 2015, 2019, 2021; Dowek, 2013) and some point to the relationship between research and educational practice, which partly arises from the undervalued role of didactic research within the academy. For example, at the conference “Key Competencies in Informatics and ICT (KEYCIT 2014)” that took place at the University of Potsdam in Germany in 2014¹, several works by science educators from various European countries were presented. In those, case studies, positions and perspectives of education in computing and technology were discussed, focused on secondary education, undergraduate education and teacher training. We found that the concepts of “competencies” and “key competencies” are a central issue and that most authors use some type of taxonomy to define their didactic model, for example in (Bröker, Kastens, & Magenheimer, 2014) the authors take the following definition of competency: “The existence of learnable cognitive abilities and skills which are needed for problem solving as well as the associated motivational, volitional and social capabilities and skills which are essential for successful and responsible problem solving in variable situations.” and they add: “This definition implies that competences are learnable by interventions.” For the competency model, these authors use the Anderson and Krathwohl taxonomy (AKT), an adaptation of Bloom’s taxonomy to which they add two dimensions: A) levels of knowledge (factual, procedural, conceptual, metacognitive) and B) classification of cognitive domains (“remembering, understanding, applying, analyzing, evaluating, creating”).

As a result of the literature review, added to our empirical research and own theoretical development, we conclude that our model of knowledge construction about data structures, algorithms and programs, helps in designing answers to the *how* to teach of the didactic questions (Saeli et al., 2011), playing a role similar to that of competencies and taxonomies used by the reviewed authors. Besides, the model contributes with a theoretical elaboration, mainly in two directions: extending Piaget’s theory to encompass the construction of knowledge about programs (da Rosa, 2018; da Rosa, Viera, & García-Garland, 2020) and providing a theoretical framework for designing empirical studies through Piaget’s triad intra-inter-trans (da Rosa & Gómez, 2019, 2022). This contribution is not minor: in their systematic review of research in computer science education, several of the cited authors found that half of the studies do not explain any theoretical framework. For the studies that do, their theories and conceptual models

¹<https://publishup.uni-potsdam.de/opus4-ubp/frontdoor/deliver/index/docId/7032/file/cid07.pdf>

are taken from other areas such as psychology or pedagogy and present a dispersed area with a great variety of terminology and methods. Although the authors anticipate a growth in the theoretical field of computer science education, at the time of their study they considered the number of studies with theoretical and conceptual frameworks specific to the area so small that they would not have sufficient impact to generate a theoretical unification of the area (Malmi et al., 2014).

Based on our epistemological model and taking fundamental ideas of computing (Schwill, 1997; Bell, Tymann, & Yehudai, 2018) as a didactic perspective, we began to design a didactic model for the development of computational competencies and skills for novice students (Cabezas & da Rosa, 2022). Fundamental ideas group together the central concepts and long-range of computing, allowing knowledge to be distinguished from ephemeral information, which constitutes a suitable answer for the *what* to teach of the didactic questions. These fundamental ideas are described in the next Section. The rest of the paper is organized as follows: in Section 3 a complete empirical study is included and in Section 4 some reflections and future lines of work are presented. Finally, bibliographic references are included.

2. Fundamental ideas of computing

Andreas Schwill's work on fundamental ideas in computer science (Schwill, 1997) is a classic cited by several authors as a starting point for the development of didactic modeling. Schwill defines four criteria that a fundamental idea must meet:

- the vertical criterion (the idea appears in different domains of the discipline)
- the horizontal criterion (the idea can be worked on at any intellectual level)
- the criterion of time (the idea can be observed throughout the evolution of the discipline)
- the common sense criterion (the idea makes sense in an informal, pre-theoretical and pre-scientific context and can be expressed in natural language)

and points out algorithmization, structural dissection and language as fundamental ideas (with sub-ideas). In (Dowek, 2012) the author adds as fundamental the idea of a machine related to the notion of a program as an executable object. In (Bell et al., 2018), the authors propose ten fundamental ideas, listed below, that cover those of Schwill and Dowek and add others related to the further development of computer science (networking, security, simulations).

1. Information is represented in digital form.
2. Algorithms interact with data to solve computational problems.
3. The performance of algorithms can be modelled and evaluated.
4. Some computational problems cannot be solved by algorithms.
5. Programs express algorithms and data in a form that can be implemented on a computer.
6. Digital systems are designed by humans to serve human needs.
7. Digital systems create virtual representations of natural and artificial phenomena.
8. Protecting data and system resources is critical in digital systems.
9. Time dependent operations in digital systems must be coordinated.
10. Digital systems communicate with each other using protocols.

The specificity of the modeling process means that we have focused on building knowledge about algorithms, data structures, and programs. Consequently, we face the challenge of defining our own subset of fundamental ideas. In principle we take the fundamental ideas above related to these concepts (1, 2, 3, 4, 5), separating them from those related to issues such as security, networks and ethics (the rest).

The authors' formulation of ideas 2, 4 and 5 led us to investigate the notions of a non-computable problem, problems without a computable solution and a non-computational problem since their clarity is relevant for our subset of fundamental ideas. Thus we find that in the complementary document to their article², the authors delve into each of the ten fundamental ideas proposed, expressing about idea 2: "The term 'computational problem', 'algorithmic problem', or simply 'problem' in this context is often used to refer to the task that needs to be computed e.g. searching for a word, sorting values into order, finding the shortest route on a map, or finding a face in a photo." At this point the importance of our theoretical elaboration is clearly observed: in our epistemological model the "algorithmic world" is distinguished from the "computational world" and we point out that "algorithmic problem" and "computational problem" are not the same (and much less simply "problem"!), even in a computer context. One of the main contribution of our work is the extension of Piaget's general law of cognition to encompass the construction of knowledge about data types and programs. Piaget's original law regulates the construction of knowledge about *algorithmic solutions* of problems and we have extended it to take into account *computational solutions*. The construction of knowledge about algorithms, data types and programs lies in understanding the dialectical relationship between both kind of solutions (da Rosa et al., 2020). In our work fundamental idea 2 is related to the "algorithmic world" and fundamental idea 5 to the "computational world".

In (Harel & Feldman, 2004) the author defines **algorithmic problem** as:

1. a characterization of a legal, possibly infinite collection of potential input sets,
2. a specification of the desired outputs as a function of the inputs.

Consequently, the fundamental ideas for which we plan to design didactic sequences and validate them in the classroom are the following:

1. Information is represented in digital form.
2. Algorithms interact with data to solve algorithmic problems.
3. Programs express algorithms and data in a form that can be implemented on a computer.
4. The performance of algorithms can be modelled and evaluated.

The fundamental idea 1 is related to the computational skill of knowing how to represent information as data types that the program has to deal with and here is used in that sense. Designing an algorithmic solution (algorithm) to an algorithmic problem consists of defining a function that takes elements from an input set and produces a desired output (fundamental idea 2), and solving the computational problem means to implement the algorithm in some programming language and execute it (fundamental idea 3). We consider that the fundamental idea 4 has to be introduced also, although it is not included in the empirical study presented in the following Section. As mentioned in the Section 1, the didactic sequence was designed and developed within the theoretical framework of the intra-inter-trans triad.

3. The empirical study

The activities described in this section were developed in two instructional instances with students in an introductory programming course in October 2023. A didactic sequence was designed to introduce

²<https://www.canterbury.ac.nz/media/documents/oexp-engineering/BigIdeas-webdocument.pdf>

a topic that until that moment was new to the students but related with their previous knowledge. It is a data structure called *capped array*, along with four fundamental operations for its manipulation: *initialization*, *insertion*, *listing* and *search*. In the capped array values are inserted one by one from zero (empty capped array) until the predefined maximum value, without being necessary to store values for a fixed number of cells determined prior to execution as in classical arrays known by the students. The cap is a special variable that keeps track of the number of values stored so far. This structure holds didactic interest in two senses: first, due to its flexibility it makes possible to establish a relationship, in terms of the process of knowledge construction about data structures, between static structures previously studied (*arrays* and *records*) and dynamic structures to be studied later (*linked lists* and *binary trees*). Second, the representation of data through the structure of capped array and operations on it present an adequate level of complexity, enhancing the introduction of fundamental ideas 1, 2 and 3 of Section 2.

The sequence was designed to be executed in class with a group of 12 students, taking into account their prior knowledge. They all had worked with the following topics: resolution of simple programming problems, basic syntax of an imperative programming language, variables, elementary data types, expressions and simple instructions, control structures (both selection and iteration), subprograms (both functions and procedures), static data structures (arrays and records). The programming language used in class was Pascal.

The activities for the sequence were designed based on guidelines established in the epistemological model according to which knowledge is built through a first stage focused on isolated objects (*intra* stage), then passing through a second stage that takes into account relationships between said objects and their transformations (*inter* stage) and reaching a third stage in which a general scheme is built that involves both the generalized objects and their transformations (*trans* stage). The sequence consists of four groups of activities introducing the fundamental ideas 1, 2 and 3 as shown below:

- A. Manipulation of the structure (fundamental ideas 1 and 2).
- B. Formalization of the structure in a programming language (fundamental idea 1).
- C. Initialization and insertion operations (fundamental idea 3).
- D. Listing and search operations (fundamental idea 3).

The activities within group A start from the students' instrumental knowledge (*intra* stage) and induce its transformation into conceptual knowledge (*inter* stage) in relation to the structure itself and the insertion operation, both of which constitute new concepts for the students. The activities in this group introduce fundamental ideas 1 and 2 by means of transforming the data in a structure that can be handled by the algorithm in Activity 3. The activities within group B introduce fundamental ideas 1 and 3 by means of discussing computational issues of the representation of the structure and its effects on the memory of the computer. The activities in group C introduce fundamental idea 3 with emphasis in the Pascal program as a formalization of the conceptual knowledge about the insertion algorithm (*trans* stage). Finally, in the activities within group D students work with operations that present similarities and differences with the insertion operation in order to consolidate the knowledge about the new structure of capped array. These are listing and search operations that students have implemented on classic arrays. We present below a detailed description of the activities within each group.

Group A: Manipulation of the structure (fundamental ideas 1 and 2)

Activity 1: *Imagine there is a shelf that contains compact discs, which are located from left to right. It is desired that a person, standing in front of the shelf, can immediately know the number of discs placed so far, without having to count them one by one or do any type of calculation. What could be added to the shelf so that the person can know that?*

The proposed arrangement has a correlation with the data structure to be constructed at the formal stage. The discs correspond to the values to be stored in the cells and their positions on the shelf to the indices

of the array (the data structure). The activity introduces the need to incorporate a new element: the *cap*. Students are induced to propose a solution, at instrumental level, to solve the task. They are expected to come up with ideas such as "put a mark" or "write down the amount of discs on a piece of paper". Based on the students' ideas, they are then asked to reflect on why each alternative works (or doesn't work). For example, putting a mark implies the need to count the discs that precede it, which does not satisfy the requirement of avoiding counting them. Having a piece of paper with the amount provides a better solution, as it avoids having to count all of the discs every time.

Activity 2: *Suppose you have a disc in your hand and you are about to place it on the shelf after the last one. What condition should be met to be able to perform the task successfully? After placing the disc, what should be done so that the person from Activity 1 still knows how many discs are placed without having to count them?*

This activity expects students to apply instrumental knowledge for its resolution. Everyone has stored items on a shelf before, so they should be able to figure it out without difficulty. The purpose is to make them aware of both the general case and the edge case (checking that the cap value does not exceed the maximum number of discs that fit on the shelf).

Activity 3: *Based on your answer to the question in activity 2, write an algorithm in pseudocode to insert a new disc on the shelf after the last one and update the number of discs placed.*

This activity proposes the use of an intermediate formalism (pseudocode) to help conceptualization and begin the passage to the trans stage. This is particularly helpful when introducing a new operation (insertion of a new element) that has no similar equivalent in the students' prior formal knowledge. So far, they had only worked with classic arrays (without a cap). In a classic array, the insertion operation is not defined, since all of its cells always have a value stored in it. The notion of inserting a new element into a data structure is being worked on for the first time with the introduction of the capped array.

Group B: Formalization of the structure in a programming language (fundamental idea 1)

Activity 4: *Define a data type in Pascal that allows representing the shelf along with the number of discs stored so far (the cap). For simplicity, assume that the shelf has the capacity to hold at most 50 discs and that each disc is simply represented by an integer number (its ISBN). Explain how each part of your Pascal definition corresponds to the shelf.*

This activity involves construction of knowledge about the use of the programming language to define the capped array. The students had previously worked separately with arrays and records, and their integration allows the new data structure to be defined, since it unifies, in the same syntactic unit, both the array and the cap. Taking advantage of the fact that no new syntactic elements are required, the activity encourages students to use what they already know about the language syntax to propose a definition for the new data structure and establish a correspondence between the shelf (instrumental stage) and its representation in a programming language (formal stage).

Activity 5: *Given the following variables in Pascal:*

```
arr: Arreglo;           (* classic array *)
act: ArregloConTope;  (* capped array *)
```

*Assuming that both of them have 10 cells (with indices ranging from 1 to 10), draw both the variable **arr** with values stored in all of its cells and the variable **act**, assuming that only the first four cells have been loaded with values so far. What expression should you write in Pascal to access the third cell of the classic array? And to access the third cell of the capped array? And to access the cap?*

Activity 6: *Now we are going to compare some characteristics between the classic array and the capped array. What syntax similarities and differences do they present? In what circumstances is it more appropriate to work with the classic array? and with the capped array? What is the purpose of the cap? Why does the classic array not need a cap? What happens with the values in the cells after the cap in*

the capped array?

Activity 5 induces students to construct knowledge about the syntax rules necessary to manipulate the capped array and become aware of how said data structure works in the computer memory. Subsequently, activity 6 induces a reflection process about the similarities and differences between the two data structures, both at syntactic and functional level. Especially in relation to the fact that, as seen earlier in the course, all cells of the classic array must be stored with valid values, while the cells after the cap contain undefined values ("garbage" values), simulating the absence of discs on the shelf beyond the position indicated by the cap. According to the epistemological model, the construction of knowledge about the new structure is carried out from the generalization of knowledge previously constructed for the manipulation of classic arrays and records.

Group C: Initialization and insertion operations (fundamental idea 3)

Activity 7: Write the following subprograms in Pascal (each subprogram name is kept in Spanish, which is the native language of the students who participated in the class, as well as the names given in Spanish for the data types: *Arreglo* for the classic array and *ArregloConTope* for the capped array):

```
procedure InicializarTope (var act: ArregloConTope)
function EstaLleno (act: ArregloConTope) : boolean
procedure Insertar (val: integer; var act:ArregloConTope)
```

which, respectively, initialize the cap with 0, determine whether the capped array already contains values in all its cells and insert the new value into the capped array, according to the algorithm in activity 3.

Activity 8: Draw a capped array that has 10 cells in total and show what it would look like after each of the following instructions is executed. How many more values could be inserted into it after executing these instructions?

```
InicializarTope (act);
Insertar (5, act);
Insertar (3, act);
Insertar (8, act);
```

Activity 7 asks students to implement the requested subprograms in Pascal, based on the knowledge built in the previous activities. It is expected that no major difficulties should arise, due to the progressive construction of concepts arising from previous activities. In particular, the third procedure constitutes a new expression of the algorithm in pseudocode. During writing, it is expected that students will make errors typical of machine execution. For example, a problem with an index that produces an out of range error, or accessing a cell with an undefined value ("garbage" value). In such cases, activity 8 induces students to resort to a mechanism called automation, described in (da Rosa & Gómez, 2022), which consists of manually executing the algorithms on an array drawn on paper which, within the framework of the epistemological model, has proven to be very useful in helping students become aware of errors and return to the code and correct them.

Group D: Listing and search operations (fundamental idea 3)

Activity 9: Write the following subprograms in Pascal:

```
procedure ListadoComun (arr: Arreglo)
procedure ListadoConTope (act: ArregloConTope)
```

which respectively list on screen the values stored in the classic array and in the capped array. Watch the code written for both subprograms. What similarities and differences do they present?

Activity 10: Write the following subprograms in Pascal:

```
function BusquedaComun (arr: Arreglo; num: integer): boolean
```

```
function BusquedaConTope(arr: ArregloConTope; num:integer):boolean
```

which respectively determine whether or not the value num is stored in the classic array and in the capped array. Watch the code written for both subprograms. What similarities and differences do they present?

Unlike the insertion algorithm from activity 8 (which is a *new* operation), students have already implemented listing and search operations on a classic array earlier in the course. These activities take this into account and seek to adapt said prior knowledge for the implementation of new versions, now working on the capped array. They have reasonable similarities and differences with the classic array, which provides an adequate context for the construction of new knowledge working at the formal level (through generalization). Finally, students are asked to compile and run all implemented subprograms on a computer. To do this, they are provided with a source file with the declarations and headers of the requested subprograms. Students simply must complete the missing portions, corresponding to the body of each of the implemented subprograms, before compiling and executing.

The course lasts 15 weeks and has several groups of students, distributed at different time schedules. Every group attends two classes per week, each lasting 90 minutes. According to the course curriculum, the capped array is planned to be worked on in week 10, after having introduced classic arrays and records in weeks 8 and 9, respectively. The didactic sequence was executed with a group of 12 students, who participated voluntarily. The rest of the students attended groups in which the topic was worked on in the traditional way (presenting from the beginning both the data structure and the operations, working directly at the formal stage), without taking into account the students' prior knowledge at the *intra* and *inter* stages. A future in-depth analysis of the differences in learning observed in the students after having worked on the topic based on one methodology or another is to be done.

All twelve students worked in teams of three members each. Everyone solved the ten activities presented. The work mechanics consisted of the teacher presenting the statement of each activity and then each team proposed a solution for it. Subsequently, a collective discussion was held among the entire class. We now present the development of the activities within each group (A, B, C and D) and a brief analysis of the work of the different teams.

Group A: Manipulation of the structure (fundamental ideas 1 and 2)

In activity 1, the four teams determined necessary to have a "space" that kept track of the number of discs placed so far on the shelf. They called it "poster", "sheet" or "paper". Regarding the question posed in activity 2, everyone concluded that it was necessary to compare the value currently noted on the "space" with the capacity of the shelf prior to placing the new disc after the last one. Within the framework of the epistemological model, students became aware of the two fundamental operations to solve the problem: *comparison* (of the cap with the maximum size) and *insertion* (of the new element after the last one) and they all managed to write the requested pseudo code in activity 3. As an example, the solution of one team is shown in Figure 1 (photo of the original algorithm, written in Spanish).

```

VERIFICAR QUE HAYA ESPACIO EN LA ESTANTERIA
SI HAY
|
AGREGAR DISCO
SUMAR 1 AL NUMERO DEL CAPTEC
FIN
  
```

Figure 1 – Pseudo code for activity 3

Group B: Formalization of the structure in a programming language (fundamental idea 1)

For activity 4, all teams defined a *record* type in Pascal that grouped the array with the cap. As expected, they did it tied to the specific instance presented in the activities of group A, considering a shelf with capacity for 50 discs. As an example, the definition proposed by one team is shown in Figure 2 ("Estantería" is the Spanish word for shelf). According to the epistemological model, such definitions

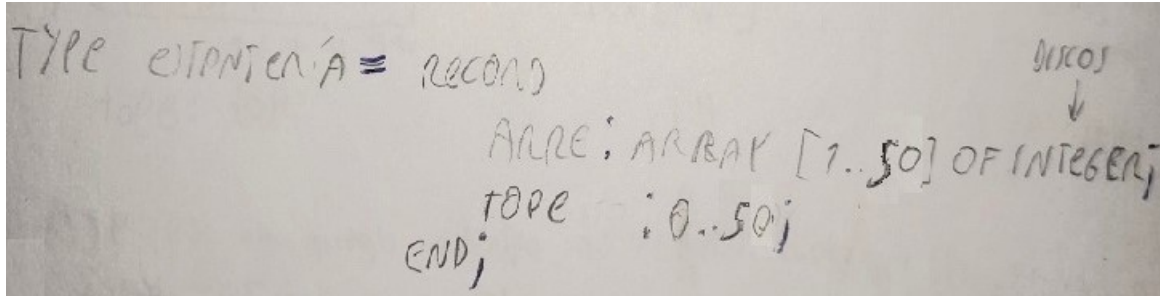


Figure 2 – Type definition for the capped array

are expected in this stage. Although they are correct from a syntactic point of view, they show that the students' thinking is still tied to the specific instance of discs manipulated in the instrumental stage. The jump from specific cases to the general case is produced by successive repetition. To abstract from the specific instance, during the collective discussion, the teams were asked how they would modify the definition so that it serves different specific sizes. This introduced the need for a constant N to make it possible to define a capped array of any size, as it is shown below:

```

type ArregloConTope = record
    arre: array [1..N] of integer;
    tope: 0..N;
end;
    
```

In activity 5, the four teams managed to draw both the classic and the capped array, according to the instructions given. When writing the expressions to access the third cell of the classic array, the third cell of the capped array, and the cap, they made various syntax errors. This is expected at this stage, since it is the first time that they combine syntax for arrays with syntax for records. In the subsequent discussion, reflection was induced and all teams were able to correct the aforementioned errors. As an example, the responses of one of the teams are shown in Figure 3.

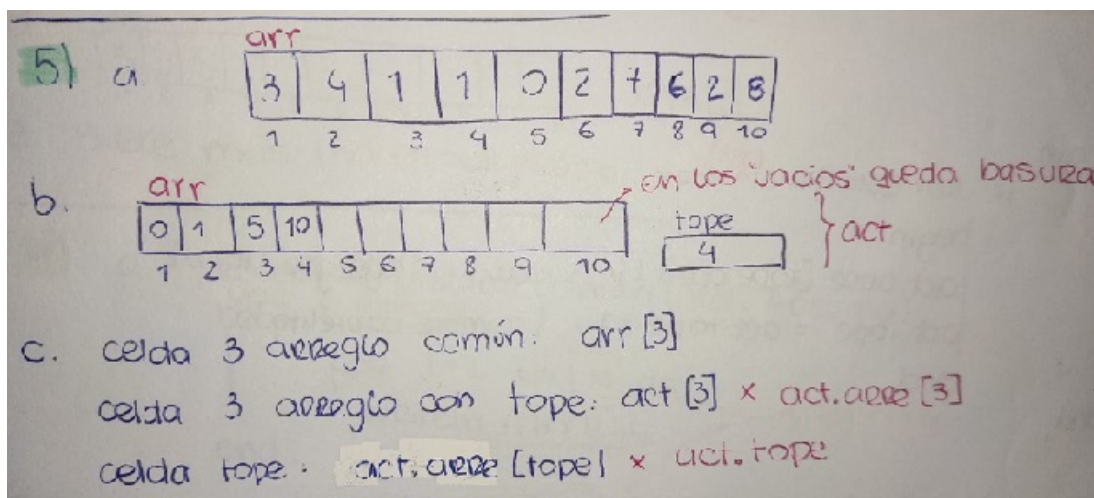


Figure 3 – Answers for questions posed in activity 5

As in activity 5, the answers to the questions posed in activity 6 presented a variety of errors, which

were again corrected after discussion, in which the similarities and differences between both structures (classic and capped array) were stated, both in terms of syntax and semantics. All students understood that it is appropriate to use the classic array when the number of values is fixed and known in advance, while the capped array is a collection with a variable number of elements, upper bounded by the size of the array.

Group C: Initialization and insertion operations (fundamental idea 3)

In activity 7, the four teams wrote initial versions for the three requested subprograms, making similar errors to those in activity 6. The syntax errors became evident after computer compilation. Facing the errors shown by the compiler helped the teams to reflect and correct them. Regarding execution errors (for example: an out-of-range error), the automation mechanism (guided by activity 8) made it possible to detect and correct them. The final version of one team for each requested subprogram is shown below:

```
procedure InicializarTope (var act: ArregloConTope);
begin
    act.tope:=0;
end;

function EstaLleno (act: ArregloConTope) : boolean;
begin
    if act.tope=N then
        EstaLleno:=TRUE
    else EstaLleno:=FALSE;
end;

procedure Insertar (val: integer; var act: ArregloConTope);
begin
    if (not EstaLleno(act)) then
    begin
        act.arre[act.tope+1]:=val;
        act.tope:=act.tope+1;
    end;
end;
```

Group D: Listing and search operations (fundamental idea 3)

Finally, activities 9 and 10 had a development analogous to that of activities 7 and 8, once again making it possible for all teams to implement correct versions for the requested subprograms (listing and search). Each team compiled and then ran all implemented subprograms on the computer. Below is the final version corresponding to the same team of questions 7 and 8 for the listing and search on the capped array.

```
procedure ListarConTope (act: ArregloConTope);
var i: integer;
begin
    for i:= 1 to act.tope do
        writeln (act.arre[i]);
    end;

function BuscarConTope (act: ArregloConTope; num : integer) : boolean;
var i:integer;
begin
    i:= 1;
    while (i <= act.tope) and (act.arre[i] <> num) do
```

```

        i := i+1;
    if i <= act.tope then
        BuscarConTope:=TRUE
    else BuscarConTope:=FALSE;
end;
```

4. Conclusions and further work

As it is pointed out in Section 1, various authors argument that one of the difficulties for computing being part of the basic disciplines for the education of all students lies in the undervalued role of its didactic. This becomes evident considering the lack of scientific research in the field supported with solid theoretical foundations elaborated with the participation of computing teachers. We have conducted for many years, empirical studies such as the one presented here, designed in the framework of epistemological basis. In the last years we have integrated results from didactic research (Bröker et al., 2014; Schwill, 1997; Bell et al., 2018) in order to produce a didactic model that computing teachers can use and evaluate. Particularly we found that the fundamental ideas of (Schwill, 1997; Bell et al., 2018) constitute an axis for designing didactic sequences, as described in Section 3. Although we have included the fundamental idea 4 about the performance of algorithms in our subset of fundamental ideas (see Section 2), we have not used it in the empirical study described here, leaving the task for future work.

The didactic sequence designed for the study constitutes one of the applications of the didactic model (didactic sequences for Physics are described in a draft version of unpublished manuscript). The theoretical framework guided by the fundamental ideas and the epistemological model that we have developed (Section 2) made it possible to design the activities in such a way that both the learning objectives and the work strategies were clearly outlined.

Within the four fundamental didactic questions: *what*, *how*, *why* and *for whom* (to teach computer science) (Saeli et al., 2011), the prior identification of the fundamental ideas made it possible to accurately define the specific programming concepts to work on in the sequence (*what*). On the other hand, the epistemological model made it possible to define the activities so that the students themselves built knowledge about these concepts during their execution (*how*). We believe that this particular point is of special value and distinguishes our didactic model from other approaches traditionally used in education, which usually focus on the presentation of concepts, without taking into account how students learn. As for *why* and *for whom*, the authors mentioned in Section 1 solidly justify why it is important to educate on computer topics in secondary education and/or undergraduate courses.

Last, in relation to the application of the didactic sequence, it was observed, in a first preliminary analysis, that the gradual nature of the proposed activities (taking into account the knowledge construction process guided by the *intra-inter-trans* triad) made it possible for the students to gradually build knowledge about the concepts worked on. The four teams of students managed to successfully solve every activity and finished with the compilation and execution on the computer of all the subprograms, implemented by themselves. The teacher's main task was to guide the collective discussion after each activity (instead of having an expository role of the concepts being worked on) and make corrections when appropriate. An in-depth analysis of the study remains to be done, from which it is expected to extract more evidence about the effect of introducing the fundamental ideas and draw conclusions that, in turn, will allow to provide feedback and enrich the didactic model, still under construction.

5. References

- Bell, T., Tymann, P., & Yehudai, A. (2018). The big ideas in computer science for k-12 curricula. *Bulletin of EATCS*, 1(124). <http://smtp.eatcs.org/index.php/beatcs/article/viewFile/521/512>.
- Bröker, K., Kastens, U., & Magenheimer, J. (2014). Competences of undergraduate computer science students. In *KEYCIT – Key Competencies in Informatics and ICT*. Torsten Brinda, Nicholas Reynolds,

Ralf Romeike, Andreas Schwill (Eds..

- Cabezas, M., & da Rosa, S. (2022). Modelado didáctico para ideas fundamentales en computación. *Proceedings of The 51 SADIO Conference, Simposio Argentino de Educación en Informática (SAEI 2022)*. <https://www.fing.edu.uy/~darosa/#papers/Modelado-Didactico-Ideas-Fundamentales-Computacion.pdf>.
- 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., & 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.
- da Rosa, S., Viera, M., & García-Garland, J. (2020). A case of teaching practice founded on a theoretical model. *Lecture Notes in Computer Science 12518 from proceedings of the International Conference on Informatics in School: Situation, Evaluation, Problems*, 146–157. <https://doi.org/10.1007/978-3-030-63212-0>.
- Denning, P., & Tedre, M. (2015). *Shifting identities in computing: From a useful tool to a new method and theory of science*. In Hannes Werthner and Frank van Harmelen, Eds. Informatics in the Future, Proceedings of the 11th European Computer Science Summit.
- Denning, P., & Tedre, M. (2019). *Computational thinking*. Cambridge, MA : The MIT Press.
- Denning, P., & Tedre, M. (2021). Computational thinking: A disciplinary perspective. *Informatics in Education.*, 20(3), 361–390. doi: 10.15388/infedu.2021.21
- Dowek, G. (2012). Les quatre concepts de l'informatique. *Bulletin de l'Association EPI*. <https://www.epi.asso.fr/revue/articles/a1204g.htm>.
- Dowek, G. (2013). *L'enseignement de l'informatique en France, Il est urgent de ne plus attendre*. www.academie-sciences.fr/activite/rapport/rads_0513.pdf. (Rapport de l'Académie des Sciences)
- Harel, D., & Feldman, Y. (2004). *Algorithmics - The Spirit of Computing*. Addison-Wesley Publishers Limited 1987, 1992, Pearson Education Limited 2004.
- Malmi, L., Sheard, J., Bednarik, B., Helminen, J., Kinnunen, P., Korhonen, A., ... Simon. (2014). Theoretical underpinnings of computing education research: what is the evidence? *ICER14: Proceedings of the tenth annual conference on International computing education research.*, 27–34.
- Saeli, M., Perrenet, J., Jochems, W. M., & Zwaneveld, B. (2011). Teaching Programming in Secondary School: A Pedagogical Content Knowledge Perspective. *Informatics in Education, Vol. 10, No. 1*, 73–88.
- Schwill, A. (1997). Computer science education based on fundamental ideas. *Proceedings of the IFIP TC3 WG3.1/3.5 joint working conference on Information technology: supporting change through teacher education*, 285–291.

Craft Ethics - Aiming for Virtue in Programming with Generative AI

Martin Jonsson
Södertörn University
martin.jonsson@sh.se

Jakob Tholander
Stockholm University
jakobth@dsv.su.se

Abstract

This paper analyses some aspects of the profound shifts in programming practice and education about by the advent of generative AI (GenAI). As GenAI tools become increasingly integrated into programming environments, they offer an approach to programming that bypasses significant aspects of the meticulous syntax-focused processes inherent in traditional programming. Instead, these tools enable a more immediate transition from problem articulation to automated solution generation, reducing the need for traditional forms of iterative problem-solving and careful focus on coding details. This paradigm shift not only challenges the foundational skills taught in programming education but also raises ethical concerns regarding aspects such as interpretability, authorship and accountability of the produced code. This involves a reevaluation of programming education and practice, suggesting a need for a reorientation to emphasise ethical and interpretative skills in programming with GenAI. Based on a series of studies on GenAI-supported programming, this paper highlights aspects relating to control, agency, and design for ethical deliberation in the evolving practices of programming with GenAI. To move towards such practice, we propose a set of design challenges based on the concept of "Craft Ethics," which emphasizes virtue, quality, and a thoughtful approach to programming and design. These challenges integrate traditional craftsmanship values into GenAI practices, ensuring that the ethical and qualitative aspects of programming are renewed and enhanced.

1. Introduction

Generative AI (or GenAI) tools based on Large Language Models such as GPT-4, Gemini, or Llama, have reached a level of functionality where they are capable of generating various forms of working programming code ranging from advanced code-completion to writing code snippets based on natural language input, as well as advanced debugging, reviewing, and rewriting code. These tools are rapidly being taken into practice both among professional programmers as well as in a range of educational settings and have given rise to renewed discussions of how programming finally may reach the point of being democratized and accessible to a broader audience, mirroring the hopes of early high-level languages such as Smalltalk and Logo. More recently, Krings et al, (2023) for instance, envision what the future of programming might be like when AI tools become powerful enough to generate working programming code based on open-ended specifications, allowing for *'everyone to be programmers'*, potentially turning programming into a practice that is more about developing ways to support humans in "how to best educate the machine" (Welsh, 2023), than about writing code in traditional ways. The new programming tools based on generative AI are certainly impressive but are also limited in many ways, and we still need to better understand how they are used in actual practices of programming, and what new kinds of interactions and collaborations need to be supported in the emerging processes of co-creation and co-coding between humans and AI (Jonsson & Tholander, 2022). In their current form, interaction with these tools is primarily conducted through various forms of text or chat-based natural language interactions which provides powerful opportunities for open-ended forms of interaction (Prather et al., 2023). However, studies have pointed to the need to also explore and understand the new practices that emerge with the use of these kinds of tools, and how the interactions and practices are shaped by the properties and behaviours of both the underlying models and the programming tools that leverage their functionalities to the end users. (Denny, Kumar, & Giacaman, 2023) (Jeuring, Groot, & Keuning, 2023) (Jonsson & Tholander, 2022). The new programming practices and the new ways to create digital interactive materials also challenge established truths about what are good and virtuous

ways of programming, and how to achieve good quality and desired behaviours of the generated code materials. Based on research in this area, we will discuss some new emerging phenomena relating to GenAI-assisted programming, and identify a set of ethical challenges relating to the use of such tools in various programming activities. To elaborate on these challenges we will articulate programming practice as a form of *craftsmanship* based on concepts from theories on craft and design practices, such as design judgment, and craftsmanship of risk. Finally, we will outline a *Craft Ethics* for GenAI-assisted programming, highlighting stances and approaches that might be beneficial for mitigating some of the identified challenges.

2. Background

An increasing amount of studies have explored the effectiveness of novel AI tools such as CoPilot for generating working programming code and how they integrate with and influence various aspects of the practices of programming, such as learning (Jeuring et al., 2023), productivity (Bird et al., 2023), use (Kazemitabaar et al., 2023), correctness and performance (Denny et al., 2023), etc. Bird, et al (2023) for instance noted how programmers using CoPilot spent less time analysing code errors, but at the same time seemed to develop less of an understanding of how or why a particular piece of code worked the way it did. They also noted that the perceived productivity among users of CoPilot was actually higher than it actually was. Relatedly, several other studies, (e.g. (Prather et al., 2023) (Barke, James, & Polikarpova, 2022)) show that users of Generative AI tools for programming were required to put additional efforts into actions such as instructing and articulating how the code should work, as well as on reading, reviewing, debugging and tweaking the code that has been generated, while less efforts were put into spending time on writing actual lines of code. This has been argued as pointing to a potential larger shift in the nature of future practices of programming, making programming less about writing code from scratch, and more about being able to interpret, understand and tweak code generated by AI systems. This has even somewhat provocatively been phrased as that we are facing "the end of programming" (Welsh, 2023), suggesting that the future of programming will change from programming being a process of writing syntactically and logically correct lines of code, to a process of being skilled in expressing and instructing the system in ways that makes it generate the code that one wants. A few recent examples of studies have explored how to design for novel kinds of programming with Generative AI may look beyond current interaction models, for instance, through prototypes that use the ability of ChatGPT to generate code based on visual and textual sketches of a program (Lewis, 2019) (Ban & Hyun, 2020) and how to create programming assistance through conversational interactions when programming using LLMs (Ross, Martinez, Houde, Muller, & Weisz, 2023).

2.1. Emerging Programming Practices

A limited number of studies have specifically addressed issues and specific patterns of use and interaction that emerge with generative AI tools in programming, and to identify emerging design opportunities (Jayagopal, Lubin, & Chasins, 2022) . Barke et. al. (2022) suggest that interaction with AI-based programming assistants is *bimodal*. Firstly, it involves what they call an *acceleration mode*, denoting how the tool simplifies and speeds up the process of writing a particular piece of code, and secondly, it involves an *exploration mode* denoting how the tool supports the programmer to explore different options when hesitating on where to go next. In a similar fashion, Jonsson and Tholander (2022) discuss the notion of *friction* and point to two ways that it can be understood in processes of programming with code generation tools. These tools may work to *remove some of the friction* that a novice programmer experiences in programming, e.g. simplifying through the generation of initial suggestions and ideas, and alternative solutions to various problems. They may also *induce friction* by slowing down the process, suggesting unexpected routes that make users reflect and explore alternative solutions. Similarly, Prather et al (2023) identified two interaction patterns in novice programmers use of CoPilot. The first, called *sheparding*, describes how students worked through continuous tweaking of prompts that guided Copilot through making its auto-generated code proposals to close up on a working solution, rather than creating code from scratch. The second, called *drifting*, regarded how students tried to make use of the system's suggestions despite not getting closer to a working solution, thereby drifting between various

suggestions and eventually getting lost. A form of interaction called *slow accept* was also identified, through which users typed in the suggested code snippet themselves, rather than merely accepting it. This allowed them to better understand the purpose and workings of the proposed code. This growing number of empirical studies of the usage of generative AI for programming have started to identify and conceptualise the different ways that incorporate these technologies in their practices. However, there is still limited understanding of how these kinds of interactions are shaped by the specific properties of the user interfaces, and the opportunities of novel designs for alternative forms of use.

2.2. From Programming Tools to Programming Partnerships

Some of the findings of previous studies have raised discussions about how these shifts in the nature of programming practice and interaction may fundamentally recast the relations between programmers and the technologies they are entangled in the creation of programming code. Both Welsch (2023) and Jonsson and Tholander (2022), point to how programmers have to give up parts of their autonomy as programmers and hand over control over the coding process to the AI tool (Bird et al., 2023). Programming tools and practices would then require the design of ways to support new forms of partnerships between human and artificial partners, new kinds of *pair-programmers*, in which the AI is considered an active collaborator, and not merely a non-agentive tool (Lawton, Grace, & Ibarrola, 2023). Such a reframing of the relations in programming practices requires a reconsideration of each actor's contribution to and role in the collaborative process. As in other forms of human-human collaboration, this is not always what can be expected or specifically designed for. All actors in various ways and to different extent contribute to bringing the overall process forward. This might involve a perspective where code develops through shared and collaborative efforts, including the wrongs and rights, mishaps and points of view of either actor. Conceptually, this aligns with notions of co-creativity and co-creation that have emerged in research in perspectives on HCI, which frame how interactive and smart technologies may be considered as co-participants in creative processes (Wakkary, 2021), rather than as mere tools controlled by human actors. This further relates to conceptual and theoretical work in HCI (Devendorf & Ryokai, 2015), around notions such as more-than-human design, and co-performance (Kuijjer & Giaccardi, 2018) and machine agency (Pickering, Engen, & Walland, 2017) which suggest a reconsideration of the view of humans as the sole source of creative agency, to also see various forms of interactive and digital technologies as sharing the agency to spur ideas and creative expressions. These theories highlight the entanglement of human agency with agencies originating from non-human entities, implying that programming and design are inherently *relational* forms of action. Programming and design are then not to be viewed as exclusively human actions, but co-constituted in entanglements of human and artificial agencies. We argue that the intersection between current technical developments and these theoretical perspectives on human-machine relationships opens up for interesting discussions around responsibility, control, and ethics in the new emerging practices of GenAI-assisted programming.

3. Ethical Challenges in GenAI-assisted Programming

As has been shown above, generative AI tools have the potential to radically redefine what programming is and how to do it. A large part of the research on GenAI-assisted programming, focuses on how to increase performance and productivity, reducing errors, and creating a more effective collaboration between humans and AI. What has been less discussed are the ethical ramifications of this shift of practice. Some concerns have been raised, for example, the risk that programmers might lose their jobs (Welsh, 2023), or that these new tools will have limited accessibility for broader groups of potential users, increasing gaps both between groups in society, as well as globally between the global north and developing countries. In the following, we will omit ethical issues on this societal macro level and instead focus on some challenges that arise on the level of the individual users and the emerging practices of GenAI-assisted programming.

As an increasing amount of the code the programmers produce is automatically generated, an inherent consequence is that the programmer has less control over the way the code is written and how it executes. For cases such as basic auto-completion for setting up simple programming constructs, this might not be problematic, as the code is easy to read and interpret. However, as these tools get more advanced, we

can assume that the generated code will be more challenging to read and, as all programming code, hide and abstract away important aspects of the code. This will give less control over the code and require the programmers to trust how the system is built and how it is executed. While large software development projects have similar consequences in that there is not one single individual in control of every piece of functionality in the system, we argue that when we to an increasing extent rely on AI-generated code, we will encounter unknown consequences of how it executes. We have identified four particular ethical challenges that may occur in the concrete GenAI-based programming practices: 1) dealing with errors and imperfections, 2) explainability and accountability of code, 3) bias and drifting in interaction, and 4) methods and approaches for bias and testing

3.1. Errors and Imperfections

Being able to interpret and find errors in generated code is reinforced by the fact that it is well-known and broadly acknowledged that despite rapid improvements, LLM-based tools still generate textual output that is far from perfect. Text generated by LLMs is commonly flawed or even totally wrong, and the same goes for the generation of programming code. A recent study from Kabir et al. (2024) shows that 52% of ChatGPT answers to programming questions contain incorrect information, but that users still prefer using ChatGPT for getting answers due to the comprehensiveness and well-articulated style of language often used. Concerning errors in generated programming code, Weisz, et al (2021) discuss how programming differs significantly from other forms of text-based practices, since small errors in programming may have much more significant consequences compared to small errors in a piece of argumentative or fictional text. Errors in the logic of the code may fundamentally change its workings, thus requiring that any piece of generated code always needs to be thoroughly reviewed. In Barke et al's (2022) study of novice programmers, they point to a need for a shift in programming practice from writing code, to reading and interpreting generated code, and how this enables students to work at a higher level of abstraction and spend their cognitive effort on thinking about the semantics of the program, rather than on details of the syntax and logic. Vaithilingam et al, (2023) as well as Jonsson and Tholander (2022) further discuss the consequences of the imperfection of the code that tools such as CoPilot or ChatGPT in relation to the design of novel forms of interaction for working with such code generation. Importantly, these studies found that despite code generation being imperfect, it still proved useful for the ongoing problem-solving and creative process, for instance as starting points when being stuck or as 'pieces of code to think with' in order to carry the overall process forward. It is likely that the performance of the LLM-models will improve, resulting in fewer errors and unexpected code. This is obviously a positive development, but it might lead to an over-reliance and poorly grounded expectations on the quality of the code that AI tools may generate, leading to the users omitting critical reflection and evaluation of the generated output. Moreover, this also builds on an assumption that the code is generated on a well-defined intention of the final execution. However, many programming projects are iteratively driven, in which the code works as a co-creative tool in developing the idea of what is to be designed and built.

3.2. Explainability and Responsibility

A commonly discussed problem with respect to AI and especially large language models, is that these systems are opaque, giving no explanation as to why a certain input generates a particular output. Even the researchers who designed these systems often have a hard time explaining the intricate links between input or output, or as bluntly put by (Welsh, 2023): *"Nobody actually understands how large AI models work."* This raises issues regarding who is responsible for the effects of the outputs of the AI system. To what extent can the user be in control of the output and the process? In the case of programming, the user might not have the abilities and training to fully understand the generated code or it may rely on code or data generated elsewhere. This raises the issue of how to design mechanisms in these systems that allow programmers to interpret and understand code in a fashion that allows them to be accountable for the behaviour of code. This concerns both an understanding of the workings of the generated code as such, but also involves challenges in how the AI tools interpret the instructions and input provided by the user, and how the generated output aligns with the user's ideas or more or less well-articulated

intentions. Partly this can be understood as a problem that can be mitigated by more powerful and better-tuned AI models, but there are dimensions of this problem that cannot be solved by engineering. The power of these tools resides in that they can translate a short instruction or a few lines of code into a solution rich with details that go far beyond what was specified in the input. If every aspect and detail that should be included in the generated output has to be specified in the input instructions, the gains and usefulness of the tools become comparable to just writing the code on your own. Projecting the usage of GenAI tools into the future, it is not hard to imagine usages in which the AI-generated output is too detailed or complex to fully comprehend even for expert users. If such code causes harm or expresses unwanted norms, it raises concerns regarding the responsibility for the emerging effects. This would call for more rigorous and efficient testing methods. However, as is well documented for instance in critical algorithm studies, the effects of these systems are not intentional or predictable beforehand but emerge out of interaction with users and the data that they operate upon.

3.3. Bias and Drifting

Bias in AI and machine learning systems is a well-known issue, extensively discussed concerning its ethical implications. Bias typically arises when training data is flawed due to prejudices related to gender, race, and other factors, leading AI systems to reproduce these flaws. In the context of GenAI-assisted programming, bias can also be significant. If the AI is trained on biased historical code bases, these biases can be reflected in its suggestions. Training data that is not diverse and predominantly includes code from specific demographics may cause the AI to favour certain coding styles or solutions. The architecture and algorithms used can further introduce bias by prioritizing specific data patterns. This may affect code completion and the recommendations that are generated, potentially reflecting bias or stereotypes thereby limiting the opportunities for novel and creative solutions. Additionally, the AI tools might generate code that is less accessible to certain user groups if it is optimized based on specific demographics, making it less usable for other groups. Bias can also manifest in the AI's performance across different development environments, particularly if certain coding environments or tools are more represented in the training data. Lastly, if the AI is predominantly trained on code from a specific region, it may fail to suggest best practices common in other regions. The potential biases of GenAI programming tools may play out in different ways in different patterns of interaction, for instance in relation to Prather et al's (2023) notion of *drifting* - a phenomenon where the AI-generated code gradually drifts away from the users' initial intentions. Drifting illustrates how a biased GenAI-programming tool can become ethically challenging, as the activity must always be understood as a joint effort between human and AI actors, and where the activity and generated outcomes is a result of a mangle of agencies (Pickering et al., 2017) attributed to both humans and the tools, models and user interfaces involved in the interaction.

3.4. Control and Testing

In a recent, still unpublished study comparing novice programmers with experienced programmers in their use of GenAI tools, a number of differences emerged concerning how the two groups manage control and testing. One critical difference between novice and experienced programmers concerned their previous coding experiences and how these influenced how well they could evaluate and test the code that the tools generated. As one would expect, the more experienced the users were, the more elaborate they were in their judging of the workings of the code that was generated. Less experienced users on the other hand, primarily evaluated the code by *executing* it in order to be able to experience and reflect upon how the output behaved in relation to what they had expected. To conceptualise this, two dimensions of interaction and use were proposed; *code-focused* versus *execution- and experience-focused* interactions. The code-focused interactions align to a large extent with existing programming practices and often involve the careful writing of prompts or structuring of sections of pseudo-code that the tools would use to generate a specific piece of code, including the interactional efforts required to understand and interpret the code that was generated. This also aligns with how previous studies (Barke et al., 2022) have argued for how generative AI tools contribute to a shift in the focus of programming practices from primarily involving various processes oriented to the writing of code, to the increased

engagement of competencies of skillfully reading, interpreting and tweaking generated code. The novice users more commonly engaged in interactions that were *execution- and experience-focused*, involving the writing of prompts, articulation of ideas, or expression of intentions that had the purpose of getting the tool to generate a piece of code that would result in a particular kind of execution - such as a specific visual interaction on the screen - without expecting an explicit idea of the specific characteristics or qualities of the actual code that the execution would be based upon. Projecting the use of GenAI tools for programming into the future, it is not unreasonable to assume that the textual code representation will be less important and less in focus than in current programming practices. This shift in how code is evaluated and tested might be ethically problematic as the focus on execution omits the scrutiny and detailed examinations of all the generated materials.

4. A Turn to Craft

Welsch (2023) claims that generative AI provides a "seismic shift" for people's relation to computation and programming, replacing predictable static processes, governed by instruction sets and decidability, with an understanding of computation as temperamental, mysterious and unpredictable, more similar to humans than we have seen before. Likewise, generative AI tools offer an approach to programming that bypasses significant aspects of the meticulous syntax-focused processes inherent in traditional programming. Instead, these tools enable a more immediate transition from problem articulation to automated solution generation, reducing the need for traditional forms of iterative programming practice and careful focus on details of syntax and logic. This paradigm shift raises ethical concerns regarding aspects such as authorship and accountability of the produced code, and how programming processes are controlled. In these emerging programming practices, there is a need to establish common understandings of what constitutes good practice, how quality is maintained, and how to aim for virtue. Programming practices have traditionally been characterised as cognitive endeavours rooted in logic and scientific scrutiny, highlighting computational thinking as a particularly critical competence, such as decomposition of problems into parts and working systematically towards a solution. An alternative, albeit not necessarily contradicting way of describing programming practice, is through the notions of crafting, and craftsmanship. This perspective on programming has previously been put to the fore by the software craftsmanship movement (Sundelin, Gonzalez-huerta, Wnuk, & Gorschek, 2021) that emerged in the early 2000s (McBreen, 2002) as a response to the industrialization of software development. This view characterises programming not merely as a technical-logical endeavour but with resemblances to art practices that require a deep commitment to quality, detail, and material. A recent literature study on the software craftsmanship movement (Sundelin et al., 2021) summarizes the core ideas behind this movement, highlighting both a focus on particular qualities of software architectures, such as simplicity, minimalism, and layered architectures, as well as a focus on processes and organisation of work highlighting iterative design with a strong focus on testing and iterative refinement. Cultural dimensions are also highlighted, such as values relating to professionalism, like pride, humility and accountability. As noted in previous chapters, GenAI-supported programming might result in practices where the care to detail is replaced by higher-level design considerations and automated manufacturing processes with a more direct route from high-level instructions to a final result. Such a process runs the risk of missing out on important considerations and attendance to the details that shape key qualities of what is being created. Therefore, we propose that the notion of software craftsmanship need to be reconsidered and reframed to align with the new directions that programming practices are taking. In order to do this we have to articulate the specific aspects of craftsmanship in line with the novel AI tools and ways of working with them, and how these can be stimulated through novel tools and methods.

4.1. Design judgment and reflection in action

Many of the ethical challenges outlined in the previous chapter highlight the need for judgment and critical evaluations to be integral parts of the programming practice. In his seminal work on the *Reflective Practitioner*, Donald Schön (1987) describes how professional designers continuously engage in a *reflection-in-action*, referring to the process by which they think about and critically analyze their actions while they are performing them. This concept emphasizes the importance of real-time reflection

to adapt and respond to complex and dynamic situations effectively. Unlike traditional forms of reflection that occur after the fact ("reflection-on-action"), reflection-in-action involves immediate, intuitive problem-solving and the ability to adjust one's approach on the fly based on the emerging circumstances and feedback. Nelson and Stolterman (2014) further elaborate on such designerly ways of reflecting in action by proposing the notion of *design judgment* as an intrinsic and important skill in design practices. Design judgment is described as a series of both conscious and subconscious judgments and deliberations on a broad range of different aspects. Parts of these judgments can be described as intellectual judgments, which could concern actively making sure that the design results live up to a "desiderata" of stated quality criteria and standards. Another dimension of design judgment is described as design volition, focusing on using one's own will to pursue desired ends, referring to a particular form of judgment-making related to the processes that bring new things into existence. These types of judgments typically do not rely on a science of measurement to determine an objective outcome. Rather it is the ability to gain subconscious insights abstracted from experiences and reflections from situations that are complex indeterminate, indefinable and paradoxical. Design judgment is here described as a process of taking in the whole, in order to formulate a new whole. Nelson and Stolterman identify design judgment as comprised of the following particular kinds of judgments:

- *Framing judgment* – defining and embracing the space of potential design outcomes and the direction that the design process will initially take
- *Default judgments* – a nearly automatic response to a triggering situation representing a form of "bodily knowing" and an application of high-level skill without conscious deliberation
- *Appreciative judgment* – determining what is to be considered background and what requires attention as foreground, assigning importance to some things and considering other aspects as part of the context.
- *Appearance judgment* – Aesthetic judgments concerning the material substance and temporal experience or the fundamental character of the design.
- *Quality judgments* – deliberations relating to craftsmanship, connoisseurship or artistry, for example highlighting precision and skill in crafting and shaping materials. Quality judgments can be understood as a quest for excellence in the making of things.
- *Instrumental judgments* – The choice and mediation of means within the context of prescribed ends. These judgments take technology into consideration and concern techniques and what instruments to use to determine what are realistic possibilities.
- *Navigational judgment* – making the right choices in an environment that is complex and unpredictable. Securing the desired state of affairs by staying on track and proceeding in the right direction, knowing when to follow the rule book and when to leave it aside.
- *Compositional judgment* – bringing things together in a relational whole, and concerns aesthetic, ethical as well as sensual considerations.
- *Connective judgment* – establishing interconnections among things so that they create functional assemblies, creating synergies and emergent qualities.
- *Core judgments* – The cases where we "know what is right" without being able to argue in a rational way. A composite of meanings and values, formed during the experience of living.
- *Mediative judgment* – balancing the different types of designer judgments to orchestrate how the whole should be brought together.

Even though programming has often been described as a craft practice, working with code and interactivity as design materials, all aspects of design judgment as described above might not be immediately applicable to computer programming. Many of the judgments are however relevant and a subset of these judgments will be a fundamental contribution to the forthcoming suggestion of a Craft Ethics for GenAI-assisted programming.

4.2. Workmanship of Risk

David Pye's work, *The Nature and Art of Workmanship* (Pye, 1968) from 1968 explores the philosophy and practice of craftsmanship, emphasizing the importance of skill and human touch in creating objects. Pye here paints a romanticized image of traditional craft and craftsmanship as a reaction to the increasing industrialisation of design and mass production of goods. He distinguishes between "workmanship of risk," where the quality of the outcome is directly influenced by the maker's skill and decisions during the creation process, and "workmanship of certainty," where the outcome is predetermined by machine processes, factory production, and automation. This distinction can be directly applicable to the case of AI-generated code which could be described as articulating a "design" or description of the wanted output, that is provided to the machine, which automates the production of the code. For such processes, you are required to know exactly what qualities you want when you start the manufacturing process, as they cannot emerge as part of the process. Pye instead advocates a craftsmanship of risk, where the end result continuously depends on the mastery as embodied in persons, reflected in a capacity for judgment, and expressed in problem-finding rather than problem-solving. But the main concern is taking care and retaining control of the work process, and as a result of this, accepting responsibility for the end result. This stance towards making with machines, as well as coding with AI, reflects a more ethical practice, where the programmer becomes an integral part of the entire programming process.

4.3. Making with Head, Heart and Hand

Cheatle & Jackson (2023) explores traditional pre-digital craft practices and highlights qualities of craft practices to be embraced in digital creative practices, such as craft's holistic ways of making with '*head, heart, and hand*' and craft's distinctly collaborative and embodied practice styles. Their conceptualisation of craft accounts for ways of making that are based on a rich and dynamic concept of personhood, a 'whole self' approach, where mindfulness and cognitive processes are combined with emotional dimensions and the physical and skilful workings of the body in equal measure. "The heart" in this understanding of craft represents a driving force that moves work forward, founded on respect for labour, hard work, self-reliance, community support, perseverance, and excellence in work. The heart here symbolizes a type of personal investment that sustains work through challenges and failures, as well as the pursuit of building a better world through care, commitment, trust, responsibility, respect, and knowledge. This embodied and holistic perspective on making, to some extent aligns with the accounts of design judgments accounted for above, referring to a "bodily knowing" based on internalised experiences.

4.4. Towards a Craft Ethics in GenAI-Assisted Programming

Based on how virtue, ethics and judgment are articulated in relation to craft and design practices, we provide a first outline of a new ethical framework, *Craft Ethics* for GenAI-assisted programming, in an attempt to highlight the need for a common understanding of *craftsmanship* and *judgment* for the emerging practices involving programming with generative AI-based tools. Craft ethics builds on the care given to details and quality that are intrinsic to most craft and design practices, and the conscious and unconscious judgments and deliberations that take place in these processes. As part of this framework, we propose an alternative understanding of design judgment, in terms of *programming judgment*. Central judgments in GenAI-assisted programming are: *Framing judgments* - Being clear of the aims and directions and being able to articulate them in the co-creative activities with the AI tools. *Quality judgments* - Continuously monitoring and evaluating the joint performance between human and AI in a quest for excellence. *Instrumental judgments* - an internalised understanding of the capabilities and limitations of the AI tools. *Navigational judgments* - Monitoring the co-creation process with a critical awareness towards drifting in unwanted directions. *Co-creative judgments* - critically monitoring how different

agencies come into play in the joint activities, and how the interplay works towards a whole. Another dimension of the Craft Ethics framework we refer to as *risky programming*, encouraging practices that emphasize flexibility, creativity, and adaptability, often at the expense of predictability and control. In risky programming with GenAI, the programmer accepts full responsibility for AI-generated outcomes and is forced to engage intimately with the co-creative process to mitigate the potential harms that might appear. The final dimension of the Craft Ethics framework is *programming with head, hands and heart*, highlighting programming as an embodied practice with intrinsic ethical dimensions, with an overarching ambition of programming as a pursuit of building a better world. A craft ethics framework could be understood as a set of guiding principles to support practitioners in how to engage in this particular form of programming activities in an ethical and sustainable way. The framework could also be used to inform the design of new tools and user interfaces for GenAI-supported programming, for example making room for and encouraging programming judgment and risky programming approaches.

5. Conclusions

In this paper, we have tried to outline some aspects related to the new emerging practices concerning GenAI-assisted programming, as well as a number of ethical challenges related to these practices. The need for a new understanding of craftsmanship for programming with generative AI tools was highlighted, discussing the need to support design judgment in different forms. An initial draft of a craft ethics framework was formulated, consisting of a) programming judgment, b) risky programming, and c) programming with head, hand and heart.

6. References

- Ban, S., & Hyun, K. H. (2020, March). 3D Computational Sketch Synthesis Framework: Assisting Design Exploration Through Generating Variations of User Input Sketch and Interactive 3D Model Reconstruction. *Comput. Aided Des.*, 120(C). Retrieved 2024-05-02, from <https://doi.org/10.1016/j.cad.2019.102789> doi: 10.1016/j.cad.2019.102789
- Barke, S., James, M. B., & Polikarpova, N. (2022, October). *Grounded Copilot: How Programmers Interact with Code-Generating Models*. arXiv. Retrieved 2024-01-24, from <http://arxiv.org/abs/2206.15000> (arXiv:2206.15000 [cs])
- Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., & Gazit, I. (2023, May). Taking Flight with Copilot. *Commun. ACM*, 66(6), 56–62. Retrieved 2024-01-30, from <https://dl.acm.org/doi/10.1145/3589996> doi: 10.1145/3589996
- Cheatle, A., & Jackson, S. (2023, October). (Re)collecting Craft: Reviving Materials, Techniques, and Pedagogies of Craft for Computational Makers. *Proc. ACM Hum.-Comput. Interact.*, 7(CSCW2), 250:1–250:23. Retrieved 2024-02-21, from <https://dl.acm.org/doi/10.1145/3610041> doi: 10.1145/3610041
- Denny, P., Kumar, V., & Giacaman, N. (2023, March). Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (pp. 1136–1142). New York, NY, USA: Association for Computing Machinery. Retrieved 2024-02-29, from <https://doi.org/10.1145/3545945.3569823> doi: 10.1145/3545945.3569823
- Devendorf, L., & Ryokai, K. (2015, April). Being the Machine: Reconfiguring Agency and Control in Hybrid Fabrication. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (pp. 2477–2486). New York, NY, USA: Association for Computing Machinery. Retrieved 2021-12-21, from <https://doi.org/10.1145/2702123.2702547> doi: 10.1145/2702123.2702547
- Jayagopal, D., Lubin, J., & Chasins, S. E. (2022, October). Exploring the Learnability of Program Synthesizers by Novice Programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (pp. 1–15). Bend OR USA: ACM. Retrieved 2024-05-02, from <https://dl.acm.org/doi/10.1145/3526113.3545659> doi: 10.1145/3526113.3545659
- Jeurig, J., Groot, R., & Keuning, H. (2023, November). What Skills Do You Need When Developing

- Software Using ChatGPT? (Discussion Paper). In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (pp. 1–6). Koli Finland: ACM. Retrieved 2024-04-25, from <https://dl.acm.org/doi/10.1145/3631802.3631807> doi: 10.1145/3631802.3631807
- Jonsson, M., & Tholander, J. (2022, June). Cracking the code: Co-coding with AI in creative programming education. In *Proceedings of the 14th Conference on Creativity and Cognition* (pp. 5–14). New York, NY, USA: Association for Computing Machinery. Retrieved 2023-11-30, from <https://dl.acm.org/doi/10.1145/3527927.3532801> doi: 10.1145/3527927.3532801
- Kabir, S., Udo-Imeh, D. N., Kou, B., & Zhang, T. (2024, May). Is Stack Overflow Obsolete? An Empirical Study of the Characteristics of ChatGPT Answers to Stack Overflow Questions. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (pp. 1–17). New York, NY, USA: Association for Computing Machinery. Retrieved 2024-05-29, from <https://dl.acm.org/doi/10.1145/3613904.3642596> doi: 10.1145/3613904.3642596
- Kazemitabaar, M., Hou, X., Henley, A., Ericson, B. J., Weintrop, D., & Grossman, T. (2023, November). How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (pp. 1–12). Koli Finland: ACM. Retrieved 2024-04-25, from <https://dl.acm.org/doi/10.1145/3631802.3631806> doi: 10.1145/3631802.3631806
- Krings, K., Bohn, N. S., Hille, N. A. L., & Ludwig, T. (2023, April). “What if everyone is able to program?” – Exploring the Role of Software Development in Science Fiction. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (pp. 1–13). Hamburg Germany: ACM. Retrieved 2024-05-02, from <https://dl.acm.org/doi/10.1145/3544548.3581436> doi: 10.1145/3544548.3581436
- Kuijjer, L., & Giaccardi, E. (2018, April). Co-performance: Conceptualizing the Role of Artificial Agency in the Design of Everyday Life. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (pp. 1–13). New York, NY, USA: Association for Computing Machinery. Retrieved 2022-01-05, from <https://doi.org/10.1145/3173574.3173699>
- Lawton, T., Grace, K., & Ibarrola, F. J. (2023, July). When is a Tool a Tool? User Perceptions of System Agency in Human–AI Co-Creative Drawing. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference* (pp. 1978–1996). New York, NY, USA: Association for Computing Machinery. Retrieved 2024-02-29, from <https://doi.org/10.1145/3563657.3595977> doi: 10.1145/3563657.3595977
- Lewis, C. (2019, April). Why can’t programming be like sketching? In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming* (pp. 1–6). Genova Italy: ACM. Retrieved 2024-05-02, from <https://dl.acm.org/doi/10.1145/3328433.3338060> doi: 10.1145/3328433.3338060
- McBreen, P. (2002). *Software craftsmanship: The new imperative*. Addison-Wesley Professional. Retrieved 2024-05-30, from https://books.google.com/books?hl=sv&lr=&id=C9vvHV1lIawC&oi=fnd&pg=PR13&dq=%5B57%5D+Pete+McBreen.+2002.+Software+Craftsmanship:+The+New+Imperative.+Addison-Wesley.&ots=pQ_x2wbVfN&sig=J9QWqgl4n8pmugreoPI1pl_d0TM
- Nelson, H. G., & Stolterman, E. (2014). *The design way: Intentional change in an unpredictable world*. MIT press. Retrieved 2024-05-30, from https://books.google.com/books?hl=sv&lr=&id=lr34DwAAQBAJ&oi=fnd&pg=PR9&ots=UFve8ln4P5&sig=kJYyFDkUsZ7Ygq0Q38vmMWu_0_s
- Pickering, J. B., Engen, V., & Walland, P. (2017). The Interplay Between Human and Machine Agency. In M. Kurosu (Ed.), *Human-Computer Interaction. User Interface Design, Development and Multimodality* (pp. 47–59). Cham: Springer International Publishing. doi: 10.1007/978-3-319-58071-5_4
- Prather, J., Reeves, B. N., Denny, P., Becker, B. A., Leinonen, J., Luxton-Reilly, A., ... Santos, E. A.

- (2023, November). “It’s Weird That it Knows What I Want”: Usability and Interactions with Copilot for Novice Programmers. *ACM Trans. Comput.-Hum. Interact.*, 31(1), 4:1–4:31. Retrieved 2024-02-20, from <https://doi.org/10.1145/3617367> doi: 10.1145/3617367
- Pye, D. (1968). *The Nature and Art of Workmanship*. Cambridge University Press.
- Ross, S. I., Martinez, F., Houde, S., Muller, M., & Weisz, J. D. (2023, March). The Programmer’s Assistant: Conversational Interaction with a Large Language Model for Software Development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces* (pp. 491–514). Sydney NSW Australia: ACM. Retrieved 2024-04-25, from <https://dl.acm.org/doi/10.1145/3581641.3584037> doi: 10.1145/3581641.3584037
- Schön, D. A. (1987). *Educating the reflective practitioner: Toward a new design for teaching and learning in the professions*. San Francisco, CA, US: Jossey-Bass. (Pages: xvii, 355)
- Sundelin, A., Gonzalez-huerta, J., Wnuk, K., & Gorschek, T. (2021, September). Towards an Anatomy of Software Craftsmanship. *ACM Trans. Softw. Eng. Methodol.*, 31(1), 6:1–6:49. Retrieved 2024-05-30, from <https://dl.acm.org/doi/10.1145/3468504> doi: 10.1145/3468504
- Vaithilingam, P., Glassman, E. L., Groenwegen, P., Gulwani, S., Henley, A. Z., Malpani, R., . . . Yim, A. (2023, May). Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode’s User Experience. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 185–195). Melbourne, Australia: IEEE. Retrieved 2024-02-29, from <https://ieeexplore.ieee.org/document/10172834/> doi: 10.1109/ICSE-SEIP58684.2023.00022
- Wakkary, R. (2021). *Things We Could Design: For More Than Human-Centered Worlds*. MIT Press.
- Weisz, J. D., Muller, M., Houde, S., Richards, J., Ross, S. I., Martinez, F., . . . Talamadupula, K. (2021, April). Perfection Not Required? Human-AI Partnerships in Code Translation. In *26th International Conference on Intelligent User Interfaces* (pp. 402–412). New York, NY, USA: Association for Computing Machinery. Retrieved 2024-01-30, from <https://dl.acm.org/doi/10.1145/3397481.3450656> doi: 10.1145/3397481.3450656
- Welsh, M. (2023, January). The End of Programming. *Commun. ACM*, 66(1), 34–35. Retrieved 2024-04-30, from <https://dl.acm.org/doi/10.1145/3570220> doi: 10.1145/3570220

Educational Tools for Probabilistic Machine Learning Curriculum in Schools

Josephine Rey

Computer Laboratory
University of Cambridge
jmr239@cam.ac.uk

Alan F. Blackwell

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

Xinyue Li

Cambridge University
Press & Assessment
University of Cambridge
xinyue.li@cambridge.org

Gemma Penson

gpenonite@gmail.com

Hong Ge

Engineering Department
University of Cambridge
hg344@cam.ac.uk

Helen Arnold

Independent Tutor
helenlindaarnold@gmail.com

Abstract

As Bayesian approaches to probability and statistics become more widespread foundations of machine learning, there is interest in introducing basic principles of probabilistic modelling at secondary school level. This paper presents a series of educational experiments with simple probabilistic modelling tools based on probabilistic programming languages.

1. Introduction

This paper reports on progress within a long-term project, following earlier reports at the Psychology of Programming Interest Group (PPIG). The overall agenda is the use of probabilistic programming languages (PPLs) to enhance education in probability and statistics, specifically when introducing concepts of probabilistic modelling into school and university curricula.

This educational agenda was one focus of a paper at PPIG 2019 that introduced the research field of usability of PPLs (Blackwell et al., 2019), in a multi-authored paper including many of the contemporary leaders in PPL development and research. Among other contributions, the 2019 paper suggested that the field may benefit from a ‘furthest-first’ strategy (starting design work with those who are most excluded), in this case by undertaking initial scoping research with school students in remote and disadvantaged communities, in particular on the African continent. Early results reported on experiments with visualising Bayesian probability in the Kalahari (Blackwell, Bidwell, et al., 2021), and use of causal models by schoolchildren in Nigeria to estimate the risk of Covid-19 infection on the basis of observations (Attahiru, Maudslay, & Blackwell, 2022).

These educational experiments have been planned in collaboration with an international mathematics curriculum research team based at Cambridge University Press & Assessment (CUP&A), who have developed a framework of mathematical concepts and learning objectives encompassing a comprehensive range of probability and statistics content across both primary and secondary school curricula (Cambridge Mathematics, n.d.). In addition to their ongoing work, the team has recently begun investigating how to support skills and practices in the field of digital technology-enhanced education, including AI (Li & Zaki, 2024). One of the goals for this team is to anticipate and document future developments in this curriculum area that will enhance competencies and advanced learning in this evolving field, much of which is dependent on fundamental principles of probability and statistics that are not yet routinely included in school curricula (Slesinski & Fadel, 2024; Hoegh, 2020).

In this paper, we present two further projects advancing our investigation: one evaluating an interactive visualisation of Causal Bayesian Networks in a UK classroom context and the other extending the Scratch language with PPL functionality to investigate how classroom use of interactive statistical modeling tools in South Africa might support curriculum priorities in that country.

2. Previous Work

As described in the introduction, work that we have previously presented at PPIG explored foundational concepts in Bayesian probability through a ‘furthest-first’ agenda to engage communities historically excluded in curriculum research. This programme of work is in contrast to other initiatives that have introduced PPLs in university-level teaching, typically at an advanced level, including Oxford, Harvard and Columbia ¹.

2.1. Conditional probability in the Kalahari

The first of these explorations investigated ways of representing and thinking about probability in relation to the context and needs of the Ju|’hoansi people living near Tsumkwe, Namibia (Bidwell et al., 2022). As hunter-gatherers, a strong ability to reason about likelihood from observed data enables survival and success. This work explored interactive visualisations of conditional probability, using physical spinners made from cardboard and paperclips to carry out simple Monte Carlo simulations that quantitatively explored the causal relationship between random variables (Blackwell, Bidwell, et al., 2021). For example, the chance of finding water under different temporal and situational scenarios (i.e. ‘after rain’, ‘within a tree’ and ‘within a tree given that it may be home to a snake’) places the foundations of Bayesian ideas within indigenous knowledge practices and elevates the importance of making AI accountable to diverse knowledge practices (Bidwell et al., 2022).

2.2. Causal reasoning during a pandemic

Following interruption of the Kalahari fieldwork by the Covid-19 pandemic, we created a simple Javascript emulation of the cardboard spinner, allowing interactive Monte Carlo simulations to be explored remotely with our field research collaborator and translator on the screen of his Android phone (Blackwell, Bidwell, et al., 2021). Simulated outcome frequencies were tallied in an interactive webpage, with the proportion of different outcomes for each variable rendered as a pie chart whose sector sizes could be compared to the relative sizes of the spinner sectors as a demonstration of long-run probabilities.

These visualisations were used as the starting point for a classroom experiment in a school in Nigeria, where a lesson plan asked children to quantify their relative risks of being infected with Covid, as informed by observations they might make in a local market (Attahiru et al., 2022). Likelihoods of different outcomes for each random variable were again visualised as different-sized sectors in a pie chart. Causal relations between random variables were visualised as links between the pie charts, showing how the different outcome likelihoods of an unknown variable might be updated on the basis of observations of other variables that it is conditioned on.

Using these visualisations, a workshop with eight students was carried out remotely, using a web-based lesson plan trialled by a teacher known to the researcher in Nigeria. The static visualisations of likelihoods were not successful in this case, in large part because the lesson plans were neither clearly related to students’ personal experiences of risk and likelihood nor to the standard curriculum in probability and statistics as taught at that level in Nigeria.

3. Interactive Visualisation of Causal Bayesian Networks

As an improvement over the static visualisation concept evaluated in (Attahiru et al., 2022), we created a dynamic version of the same visualisation, in which the circular nodes of a Bayesian network are again replaced by pie charts whose sector sizes correspond to the relative likelihoods of different outcomes for a categorical random variable. Figure 1 shows an overview of this system’s operation.

The interactive network graphs allow students to create and link nodes, specify frequencies of known values for exogenous variables (observed data external to the model), and observe expected outcome distributions for the endogenous variables (values inferred by the model) that are specified as child nodes in the graphical model. The graph is constructed by interactively creating and linking nodes, with

¹e.g. Oxford’s *Bayesian Statistical Probabilistic Programming*, Harvard’s *Probabilistic Programming and Artificial Intelligence*, Columbia’s *Applied Statistics III Nonparametric Theory in Machine Learning*

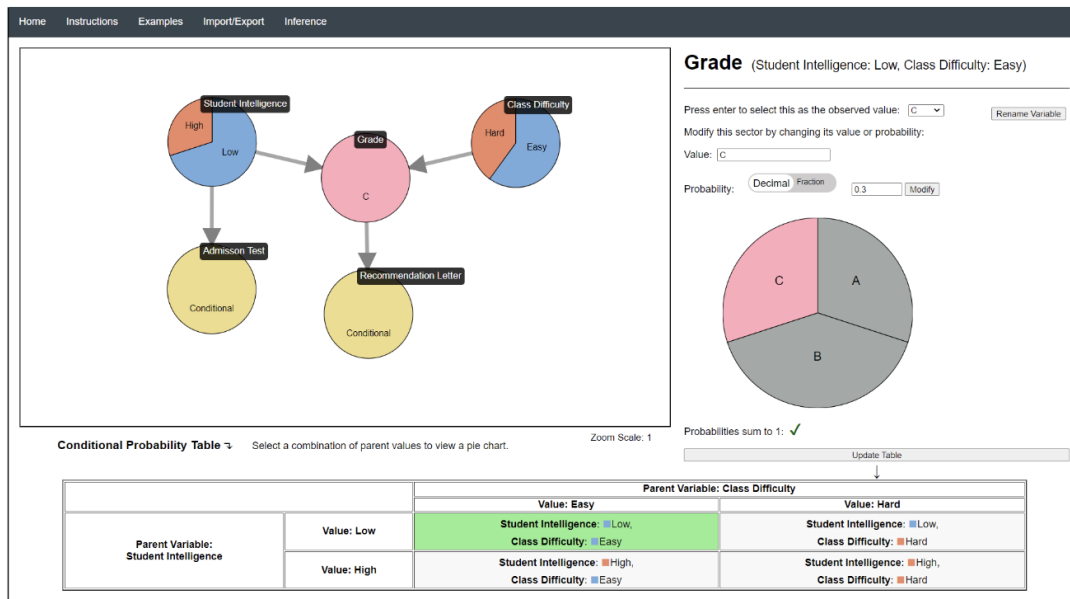


Figure 1 – Overview of the interactive CBN visualisation created by Penson, showing a Causal Bayesian Network with relative likelihood of values indicated by sizes of pie chart segments within each node. The bottom part of the display shows a corresponding conditional probability table.

class likelihoods for exogenous variables entered via an interactive dialog, and model updates rendered as modified pie charts for the endogenous variables.

3.1. Implementation

With the visualisation implemented in d3.js, the original plan had been to support more complex or data-intensive models via back-end execution in the Turing PPL (Ge, Xu, & Ghahramani, 2018). Early experiments also used the Julia implementation of BUGS, which shares Julia components with Turing.jl (Xianda Sun & Ge, 2024). However, performance issues with that server connection, and the relative simplicity of the teaching scenarios, meant that the classroom deployment of the system could be achieved with in-browser execution, with the graphical model compiled to the TypeScript PPL BayesJS² (re-compiled using Browserify). An example of the BayesJS node syntax corresponding to one of our teaching examples is shown in Figure 2.

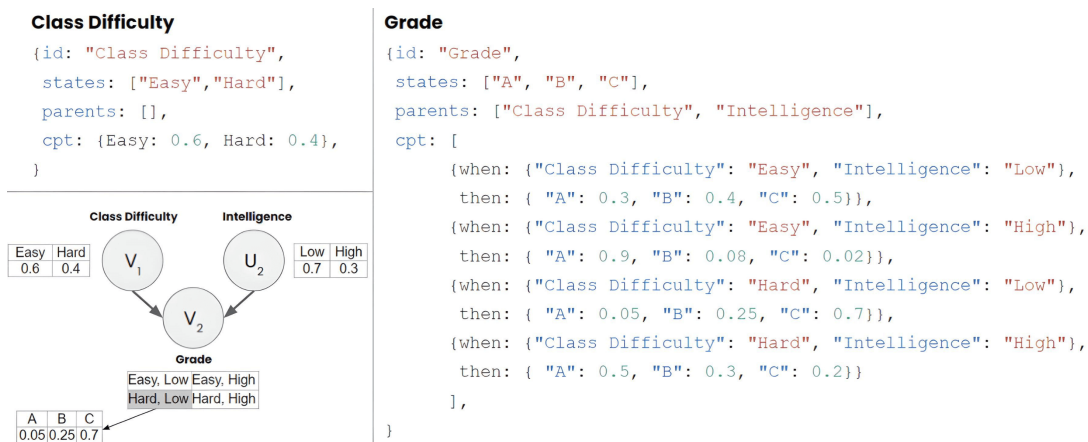


Figure 2 – BayesJS node syntax corresponding to part of the university applications model

²github.com/bayesjs/bayesjs

3.2. Evaluation

Where Attahiru's project (Attahiru et al., 2022) had explored causal reasoning in relation to experiences that were expected to be meaningful to schoolchildren in Nigeria, we evaluated this interactive visualisation in a Western context, with secondary school students studying the UK sixth-form curriculum in Further Mathematics. In this context, a cultural priority is how students will achieve the necessary grades for admission to university. We, therefore, used a teaching example that focused on causal factors in university admission, based on the Western approach to teaching Bayesian probability advocated by Pearl (Pearl, 1995), and replicating teaching examples that had previously been used for school-based research in other Western settings by (Lecoutre, 1992) and (Gordon, Henzinger, Nori, & Rajamani, 2014).

Classroom evaluation of our tool focused on the following research questions:

1. Do students employ Bayesian thinking in their decision-making? (Pre-intervention)
2. Can Bayesian thinking be induced through interaction with an interface? (Intervention)
3. Is the system's usability sufficient to support students completing Bayesian tasks? (Usability)

An in-school trial was conducted with 20 students at Hill's Road Sixth Form College in Cambridge. This took place during a timetabled class in Further Maths, supervised by a classroom teacher. The study was approved by the ethics committee of the Cambridge Department of Computer Science and Technology.

The research questions were investigated through six tasks:

- Tasks 1, 4, 5, and 6 directly address the research questions:
 - Task 1 (Pre-intervention): Students answer questions about BT without using the interface. (RQ1)
 - Task 4 and 5 (Intervention): Students build a Causal Bayesian Network (CBN) using either the interface or on paper (within-subjects). (RQ2)
 - Task 6 (Usability): Students complete self-directed tasks with the interface, reporting difficulty and completion level. (RQ3)
- Tasks 2 and 3 (Intervention) These provide foundational knowledge for later tasks:
 - Task 2: Lecture on probability and probability trees.
 - Task 3: Introduction to CBNs using examples.

The order of completing tasks with and without the interface is switched between groups to avoid bias. Overall, this evaluation aims to assess if the educational tool can effectively teach Bayesian thinking through interaction and if the interface itself is usable for students.

Using Likert scale measures in a post-intervention survey, students reported improved understanding of Bayes theorem, conditional and marginal probabilities, and prior and posterior likelihoods ($p < 0.05$). In comparing usability between the two presentation conditions, they reported that hand-drawn CBNs were easier and faster to create (Welch's t test, $p < 0.05$), while those created using the interactive editor were easier to modify and explore.

4. Teaching Bayesian probability in a South African context

This section reports a preparatory study exploring the potential of digital tools to introduce Bayesian concepts in probability education in South Africa. The first author conducted interviews with educators, curriculum designers, and NGO leaders to understand the challenges of designing future probability

curricula and how the current South African Curriculum Assessment Policy Statements (CAPS) support digital tools and their use in classrooms (Department of Basic Education, 2011).

The study identified several challenges in South Africa's education system. The shift from a student's mother tongue to English in the Intermediate Phase (ages 9-13) creates difficulties for learners when acquiring subject-specific knowledge. Additionally, the CAPS curriculum is outcomes-driven and rigid in structure, with its delivery frequently affected by structural and socioeconomic inequality between schools (Spies, 2022).

Regarding digital tools, teachers lack the resources and training necessary to integrate technology effectively into their classrooms, and CAPS does not provide guidance for employing such tools. When considering future curricula, the importance of addressing infrastructural limitations and insufficient teacher training for digital tools becomes clear. The findings emphasised that curricula should celebrate African perspectives and integrate indigenous knowledge systems. However, they also acknowledge the need for a balance between including these local contexts and ensuring students are prepared for a globalised world.

The study also explored the challenges of teaching probability and mathematics in South Africa. Probability studies fall under the mathematics curriculum's 'Data Handling' focus area, but feature few real-world notions of likelihood beyond games of chance and simple data collection/analysis. Findings suggest that games from African cultures could be a valuable resource for teaching probability concepts instead of those related to suits and cards.

In designing digital tools to support future curricula, informants stressed the importance of accessibility for learners with varying digital literacy levels. Teacher training and support are crucial for the successful implementation of these tools. The study also highlights the need for the tools to function offline, considering limitations like power outages that are common in South Africa. Finally, user testing in classrooms is essential to evaluate and improve the effectiveness of these digital tools.

Overall, the findings highlighted the need for culturally-relevant curricula that integrate African knowledge systems alongside globally recognised educational standards. Inclusive digital tools are necessary, but researchers must address infrastructural challenges and ensure these tools can support diverse learners. The study emphasises the importance of ongoing collaboration between researchers, educators, and policymakers. To this end, South African educators are committed to improving education and building capacity for future skills, with efforts underway to integrate technology, social-emotional learning, and indigenous knowledge into the curriculum.

5. The ScratchTuring hybrid PPL

Previous experiments in this programme of work, as reported above, have achieved web-deployable prototypes by using visualisation libraries such as d3.js and implementing basic programmability with additional semantic elements such as the facility to link pie charts together as nodes of a graph. In these earlier systems, more sophisticated probabilistic modelling has been achieved either through back-end execution using a general-purpose PPL such as Turing or JuliaBUGS or in-browser execution via compilation to Javascript as in BayesJS.

In order to explore the potential for web-based educational PPLs in the South African context, we used a more powerful formalism for visual computation: the well-known Scratch language and environment originally created for school-level introductory programming classes (Resnick et al., 2009). Scratch already has sophisticated editing, interaction, and visualisation capabilities, and is deployed in a fully browser-based version with extension capabilities that enabled the extensions described below.

5.1. Turing interface

The ScratchTuring hybrid introduces new Scratch blocks that invoke the PPL functionality of the Turing.jl language, cross-compiled into Turing scripts that are executed on a (local or remote) server, which can then be queried from Scratch to visualise the probabilistic model. We created Scratch-syntax wrap-

pers for a subset of the Turing language so that Turing development tasks, such as creating and conditioning models on new data, can be done within the Scratch editor. This functionality is delivered through a constrained set of blocks specifically to support lesson plans in probabilistic modelling, as seen in Figure 3.

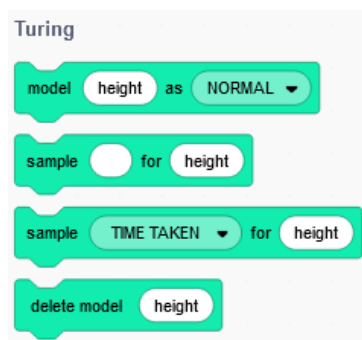


Figure 3 – Scratch blocks used to interact with model distributions in Turing

5.2. Geographical data

We wanted to address the policy demand in South Africa for educational tools that relate to recent advances in machine learning while also considering the geographical and cultural context within which South African students engage with such advances. We therefore created new Scratch capabilities relating to these concerns, in contrast to the original development and evaluation of Scratch that focused on the Western priorities of children’s engagement with computer games and digital media.

One lesson plan was inspired by previous work that had provided schoolchildren in Ethiopia with programmable access (via simplified Python libraries) to satellite imagery from Google Earth Engine geo-tagged with the what3words API (Longdon, Gabrys, & Blackwell, 2024). We created a Scratch extension that loads a satellite image from any specified coordinates as a ‘backdrop’. This image can then be used in conventional Scratch code, as seen in Figure 4 where the Scratch character is making a random walk, sampling colours from an image of the South African coast.

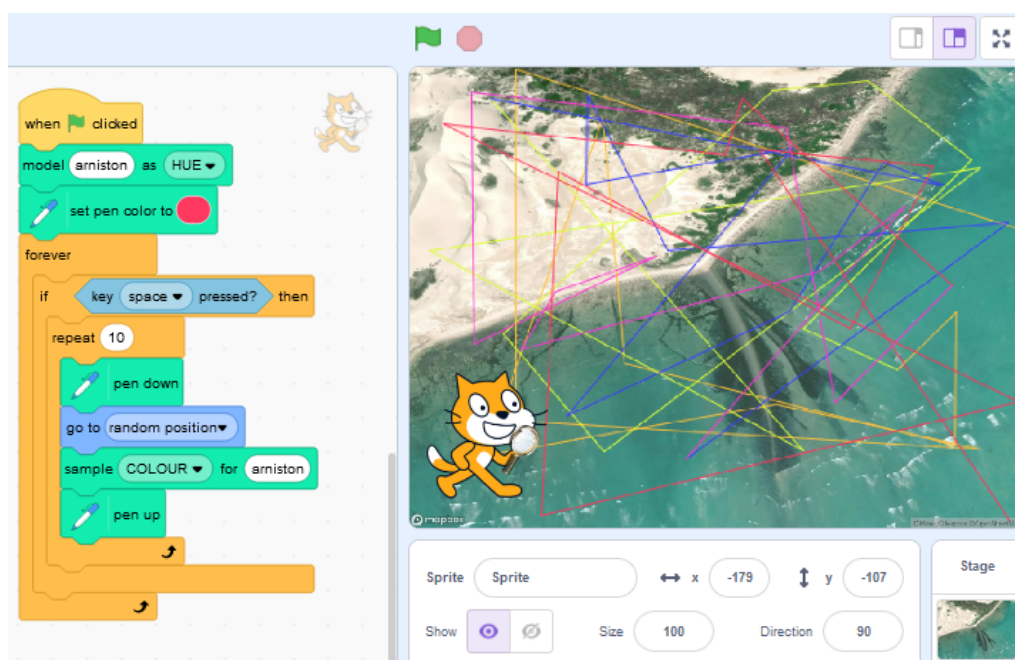


Figure 4 – The ScratchTuring interface used to create a program that collects samples from a satellite image

5.3. Probability model dashboard

The core extension that allows ScratchTuring to be used as an interactive tool for exploration of probabilistic models is a model *dashboard*, implemented as a new tab in the browser-based Scratch client. The operation of this dashboard can be seen in Figure 5, which shows a teaching application where each Scratch sprite represents an elephant in the Kruger National Park, South Africa’s largest nature reserve. The elephant sprite reports its attributes as sample observations, updating a statistical model maintained in Turing. The ScratchTuring dashboard can then be used to visualise a probability density function reported by that model, showing the prior distribution before the observation, followed by (as in Figure 5) the posterior distribution. Using these facilities, a teacher projecting the dashboard can deliver exploratory interactive lessons, and students may also experiment with Scratch to create larger or more complex data science projects and simulations.

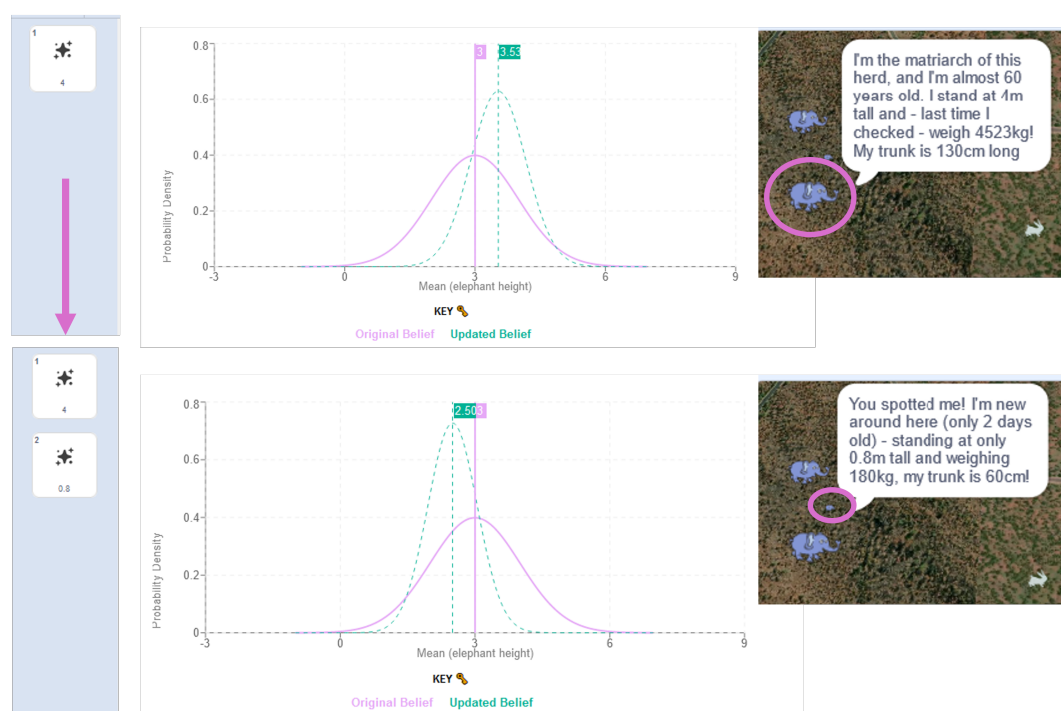


Figure 5 – The ScratchTuring model dashboard showing prior and posterior distributions based on reported elephant heights. After observing a baby elephant, the posterior distribution (green dotted line) reflects an updated belief that elephants can sometimes be smaller.

5.4. Visualising image samples as a hue distribution

As an experiment in relating local understanding of satellite image data to simple probabilistic methods in computer vision and machine learning, we implemented a model type and visualisation that renders probability distribution for colours sampled from a hue spectrum, as seen in Figure 6. The hue sampling Scratch block calculates a local average RGB from the background of a sprite’s location and maps this into hue space (with saturation and brightness collapsed). This allows a distribution of hues to be calculated across samples from a satellite, as seen in figure 6. In the figure, it can be seen that hues are drawn from two different distributions, one corresponding to portions of the satellite image containing the ocean, and one corresponding to the colour of the beach. Although rendering a hue spectrum as the x-axis of a histogram is intuitively appealing, it should be noted that not all HSB colour values are easily perceived as being similar to the same hue value with 100% saturation and brightness. To help students appreciate this mapping, the histogram bars are rendered using the average of the saturation and brightness values observed, rather than the bright colours shown on the axis. The actual colours observed in the samples collected for a particular hue also pop up as a set of patches when the user hovers over its corresponding bar.

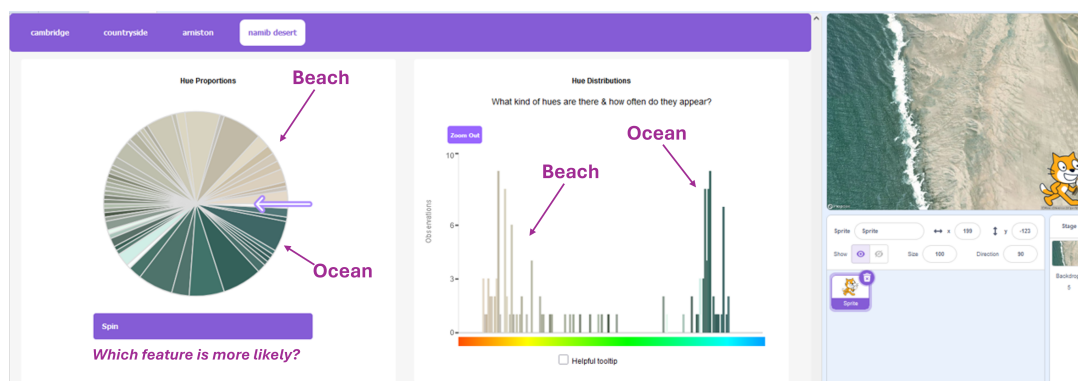


Figure 6 – The ScratchTuring model dashboard showing the distribution of hues sampled from a satellite image. The samples are collected using behaviour added to the Scratch character, shown here carrying a magnifying glass, whose lens is positioned over the part of the image from which the sample will be taken.

5.5. Lesson evaluation

At the time of writing, the ScratchTuring prototype has been presented to UK students at a specialist mathematics school in Cambridge. Students from the school are planning to travel to Kenya in the summer of 2024 for a programme of activities planned to include volunteer teaching at a Kenyan primary school. There may be an opportunity for students to use some aspect of this project during their visit.

The ScratchTuring prototype has also been demonstrated to several of the South African teachers who participated in the study described in Section 4. Additionally, a lesson plan incorporating this tool was recently introduced to Grade 8 students (ages 13-14) in Langa, a township in Cape Town, during a mathematics class at a specialist maths and science secondary school.

6. Discussion

We have described two interactive prototypes, continuing a series of educational experiments studying the potential use of probabilistic programming languages for teaching Bayesian probability and statistics at school level. Both systems are intended for web deployment in classrooms and present an interactive graphical front end, with a more conventional PPL used to construct and manipulate a Bayesian model.

We are especially interested in the potential for new developments in international school curricula to be initiated in non-Western contexts, including countries in Africa. Although the authors are based in the UK and often work with local schools as required by the practicalities of student research projects, we are focused on curriculum ideas that can be informed by local understanding and indigenous knowledge traditions from other parts of the world (Blackwell, 2021; Blackwell, Damena, & Tegegne, 2021). Since the methods of Bayesian probability underpinning recent advances in machine learning already represent a change in emphasis from the established conventions of frequentist statistics embedded in today's Western curricula (Slesinski & Fadel, 2024), this seems an ideal opportunity to consider the opportunities and implications arising less WEIRD (Western, Educated, Industrialised, Rich, Developed) ways of thinking (Henrich, Heine, & Norenzayan, 2010; Escobar, 2018).

At the 2022 PPIG workshop, Zainab Attahiru presented lesson plans that visualised conditional probability in a form intended to be accessible to students in Nigeria, allowing them to reason quantitatively about risks in their own lives (Attahiru et al., 2022). Gemma Penson's project, as reported here, has implemented an interactive version of the same visualisation, demonstrating that it can be used by UK students to reason in a probabilistic way about their school ambitions.

The series of educational experiments described in this paper aligns with key learning sciences theories such as project-based learning (engaging students in real-world challenges), situated learning (embedding education within its natural context), and simulation-based learning (using interactive, real-life

scenarios). These perspectives afford the potential to extend the findings beyond local settings to international and global contexts.

The ScratchTuring prototype that we have introduced builds on these experiments to create a fully featured visual programming environment, with facilities supporting direct modelling of problem domains relevant to African learners. Preliminary teacher evaluation suggests that ScratchTuring is sufficiently robust and usable for deployment in classrooms, and we expect to be able to report on those deployments at the PPIG workshop.

7. Acknowledgements

We are grateful to the many teachers and students who have advised us and participated in the experimental lessons reported here. Josephine Rey's research has been funded by the Margaret & Patrick Flanagan Trust. Continued research enhancing the usability and educational applications of the Turing language is funded by the Alan Turing Institute. Generative AI declaration: The preparation of this manuscript employed multiple automated language-processing tools, including functions for spelling and grammar correction, summarisation, paraphrasing and predictive text.

8. References

- Attahiru, Z., Maudslay, R. H., & Blackwell, A. F. (2022). Interactive bayesian probability for learning in diverse populations. In *Ppig* (pp. 77–87).
- Bidwell, N. J., Arnold, H., Blackwell, A. F., Nqeisji, C., Kunta, K., & Ujakpa, M. (2022). Ai design and everyday logics in the kalahari. In *The routledge companion to media anthropology*. Routledge. Retrieved from <https://www.routledgehandbooks.com/doi/10.4324/9781003175605-54> (Accessed on: 14 Dec 2023)
- Blackwell, A. F. (2021). Ethnographic artificial intelligence. *Interdisciplinary Science Reviews*, 46(1-2), 198–211.
- Blackwell, A. F., Bidwell, N. J., Arnold, H. L., Nqeisji, C., Kunta, K., & Ujakpa, M. M. (2021). Visualising bayesian probability in the kalahari. In *Ppig*.
- Blackwell, A. F., Church, L., Erwig, M., Geddes, J., Gordon, A., Maria, I. G., ... others (2019). Usability of probabilistic programming languages. In *Ppig* (pp. 53–68).
- Blackwell, A. F., Damena, A., & Tegegne, T. (2021). Inventing artificial intelligence in ethiopia. *Interdisciplinary Science Reviews*, 46(3), 363–385.
- Cambridge Mathematics. (n.d.). *Randomness and probability research summary*. (unpublished)
- Department of Basic Education. (2011). *Curriculum and assessment policy statement: Mathematics grades 1-3*.
- Escobar, A. (2018). *Designs for the pluriverse: Radical interdependence, autonomy, and the making of worlds*. Duke University Press.
- Ge, H., Xu, K., & Ghahramani, Z. (2018). Turing: a language for flexible probabilistic inference. In *International conference on artificial intelligence and statistics* (pp. 1682–1690).
- Gordon, A. D., Henzinger, T. A., Nori, A. V., & Rajamani, S. K. (2014). Probabilistic programming. In *Future of software engineering proceedings* (pp. 167–181).
- Henrich, J., Heine, S. J., & Norenzayan, A. (2010). The weirdest people in the world? *Behavioral and brain sciences*, 33(2-3), 61–83.
- Hoegh, A. (2020, September). Why Bayesian Ideas Should Be Introduced in the Statistics Curricula and How to Do So. *Journal of Statistics Education*, 28(3), 222–228. Retrieved 2023-11-06, from <https://doi.org/10.1080/10691898.2020.1841591> (Publisher: Taylor & Francis _eprint: <https://doi.org/10.1080/10691898.2020.1841591>) doi: 10.1080/10691898.2020.1841591
- Lecoutre, M.-P. (1992). Cognitive models and problem spaces in “purely random” situations. *Educational studies in mathematics*, 23(6), 557–568.
- Li, X., & Zaki, R. (2024). Harnessing the power of digital resources in mathematics education: The potential of augmented reality and artificial intelligence. In S. Papadakis (Ed.), *Iot, ai, and ict*

- for educational applications: Technologies to enable education for all* (pp. 191–223). Cham: Springer Nature Switzerland.
- Longdon, J., Gabrys, J., & Blackwell, A. F. (2024). Taking data science into the forest. *Interdisciplinary Science Reviews*, 49(1), 82-103. doi: 10.1177/03080188241230415
- Pearl, J. (1995). Causal diagrams for empirical research. *Biometrika*, 82(4), 669–688. Retrieved 2023-12-19, from <http://www.jstor.org/stable/2337329>
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... others (2009). Scratch: programming for all. *Communications of the ACM*, 52(11), 60–67.
- Slesinski, J., & Fadel, C. (2024). *What “mathematics of ai” should be taught in schools?* (Tech. Rep.). Center for Curriculum Redesign.
- Spies, A. (2022). A culture of exclusion: Re-configuring inclusive education in south africa. *International Human Rights Law Review*. Retrieved from <https://api.semanticscholar.org/CorpusID:253231697>
- Xianda Sun, A. T., Philipp Gabler, & Ge, H. (2024). Juliabugs: A graph-based probabilistic programming language using bugs syntax. In *The languages for inference (lafi) workshop at the 51st acm sigplan symposium on principles of programming languages (popl 2024)*.

Automatic Bias Detection in Source Code Review

Yoseph Berhanu Alebachew

Virginia Tech
yoseph@vt.edu

Chris Brown

Virginia Tech
dcbrown@vt.edu

Abstract

Bias is an inherent threat to human decision-making, including in decisions made during software development. Extensive research has demonstrated the presence of biases at various stages of the software development life-cycle. Notably, code reviews are highly susceptible to prejudice-induced biases, and individuals are often unaware of these biases as they occur. Developing methods to automatically detect these biases is crucial for addressing the associated challenges. Recent advancements in visual data analytics have shown promising results in detecting potential biases by analyzing user interaction patterns. In this project, we propose a controlled experiment to extend this approach to detect potentially biased outcomes in code reviews by observing how reviewers interact with the code. We employ the "spotlight model of attention", a cognitive framework where a reviewer's gaze is tracked to determine their focus areas on the review screen. This focus, identified through gaze tracking, serves as an indicator of the reviewer's areas of interest or concern. We plan to analyze the sequence of gaze focus using advanced sequence modeling techniques, including Markov Models, Recurrent Neural Networks (RNNs), and Conditional Random Fields (CRF). These techniques will help us identify patterns that may suggest biased interactions. We anticipate that the ability to automatically detect potentially biased interactions in code reviews will significantly reduce unnecessary push-backs, enhance operational efficiency, and foster greater diversity and inclusion in software development. This approach not only helps in identifying biases but also in creating a more equitable development environment by mitigating these biases effectively.

1. Introduction

The Software Development Life Cycle (SDLC) is full of decisions. For example, when working in a team, developers perform an activity called code review which requires the reviewer deliberate on whether to include the code written by other developers into the main code-base. In open source project that utilize distributed version control systems such as Git¹, this review process will take a from of pull request.

Like any human decision-making process, stakeholders in the SDLC are susceptible to cognitive bias. In a mapping study that assessed over 65 scientific papers published up to 2016, Mohanani *et al.* (Mohanani, Salman, Turhan, Rodriguez, & Ralph, 2020) reported that at least 37 cognitive biases have been shown to exist in the software engineering process. Based on a study that observed 16 developers in situ, Chattopadhyay *et al.* (Chattopadhyay et al., 2022) reported that 45.72% of developers' actions exhibited one or more of 28 biases.

In a large-scale pull request analysis, Murphy-Hill *et al.* (Murphy-Hill, Jaspan, Egelman, & Cheng, 2022) discovered that biased pushbacks exist with a high correlation to the code author's gender, ethnicity, and age. The authors estimated that over 1000 hours of programmer time are wasted per day due to this pushback. Such push back could be a result of in-group favoritism, which is when one give preferential treatment to others they perceive as being part of their own group, whether that group is defined by ethnicity, nationality, religion, or other shared characteristics.

With the pervasiveness of open source software projects and global software development powered by distributed version control systems such as git, programmers work with code contributors/authors of

¹<https://git-scm.com/>

different background. This means, the source code review process (often in the form of pull/merge request) involves accepting/rejecting code written by a contributor whom the reviewer might not know at all. Platforms like GitHub² provide some profile information about the code author. This in principle should not be a problem as code review should only involve examining the source code and decide on its merit only. However, it has been shown that reviewers examine information besides the source code while performing the review (Ford, Behroozi, Serebrenik, & Parnin, 2019). In the study participants reviewed source code while their eye gaze was tracked. This authors reported that reviewers gazed and fixated on profile indicators such as profile picture of the authors more than what was self reported by the participants.

In a follow-up to the aforementioned study, Huang *et. al.* (Huang et al., 2020) established a causation relationship between reported author gender and biased merge request decision. In this study, participants presented with pull request were more likely reject it if the author was reported to be a female or an automated bug fix program than if the same pull request was presented as being authored by male contributor.

1.1. Background

Decision-making refers to the process of choosing among several alternatives. We typically associate decision-making with instances where we make conscious and deliberate choices. However, based on this definition, we make decisions every minute of every day, often with little conscious effort. Generally, we want to and believe we make rational decisions, that is, we choose the alternative that is most likely to lead to our desired outcome (Pronin, Lin, & Ross, 2002).

Making such rational decisions, however, requires that we weigh all alternatives and the potential outcomes for each, which is resource-intensive. There is also a trade-off between the accuracy and timeliness of our decisions. As a result, we tend to take shortcuts called heuristics. These short cuts enable us to make decisions that, while not guaranteed to be rational all the time, work often enough and are far timelier than exploring all alternatives in detail. These shortcuts take various forms but generally result from an effort to achieve decision-making goals with minimal resource expenditure.

When these shortcuts fail to result in a rational outcome, it leads to what we call cognitive bias. Tversky and Kahneman, who first introduced the concept of cognitive bias, defined it as "the systematic errors in judgment that occur predictably in particular circumstances" (Tversky & Kahneman, 1974). One key aspect of this definition is that biases are predictable. This means that under the right circumstances, the likelihood of making a cognitive bias is more than random chance. Since the introduction of the concept in 1974, over 200 cognitive biases have been identified by researchers in behavioral economics, psychology, management, and sociology.

1.2. Problem Statement

The core problem we aim to address in this proposed project is the need for an automated system to detect bias in source code reviews. Bias in code reviews can significantly impact the quality and fairness of the review process, leading to potential issues in code quality, team dynamics, and overall project outcomes. Currently, detecting such biases is challenging due to the subtle and often unconscious nature of biased behavior. Our approach involves analyzing data on how reviewers interact with the code review screen. By examining patterns in actions such as scrolling, highlighting, commenting, and time spent on different sections of the code, we aim to identify indicators of bias. This attempt to identify patterns of interaction that can indicate possibility of bias will provide valuable insights into the review process, enabling teams to address and mitigate bias, thereby improving the fairness and effectiveness of code reviews.

²<https://github.com/>

2. Literature Review

2.1. Physical Manifestation of Bias

Boonprakong *et al.* (Boonprakong, Chen, Davey, Tag, & Dingler, 2023) explored the indicators of bias in physiological and interaction data. They demonstrated the existence of behavioral and physiological signals of cognitive bias. The authors conducted two studies that collected and analyzed eye-tracking, skin conductance level (SCL), and fNIRS data from an experiment in which participants were presented with information that either aligned with or opposed their opinions on a topic. Since the opinions were presented in text form, the eye-tracking data collected focused on dwelling time instead of scan path.

Using the data from the experiment, the authors built four bias classifiers: a Linear Discriminant Analysis (LDA), Support Vector Machine (SVM) with RBF kernel, Random Forest, and XGBoost, all with 5-fold cross-validation. The highest accuracy achieved was 55.27% with XGBoost, which barely outperformed ZeroR (50.04%). To ensure that the models' performance was not due to mere chance, they conducted permutation tests and found that all achieved a p-value lower than 0.05. Participants showed higher neural activity and spent more time processing statements that disagreed with their own opinions.

2.2. Cognitive Bias in Software Engineering

The Software Engineering (SE) process is intricate and decision-intensive. Like any decision-making endeavor, SE is prone to cognitive biases. Mohanani *et al.* (Mohanani et al., 2020) conducted a comprehensive mapping study to explore cognitive biases within the field of software engineering. This study aimed to identify, catalog, summarize, and synthesize existing research on the subject. Specifically, it highlighted research on cognitive biases frequently encountered in SE, focusing on their origins, manifestations, and impacts. The authors examined 65 papers published until 2016, finding that 37 cognitive biases have been studied at various stages of the SE process.

Chattopadhyay *et al.* (Chattopadhyay et al., 2022) conducted a two-part field study to assess the extent of cognitive biases in developers' daily activities. The field study took place in a startup, where the authors observed ten developers in situ for an hour while they performed their daily tasks. They found that 45.72% of developers' actions (953 out of 2084) involved one or more of the 28 observed biases, which were grouped into ten categories.

As part of the study, follow-up interviews were conducted with 16 developers to validate the findings, assess the perceived frequency of biases, and identify available tools and practices to address the identified biases as reported by developers. The authors reported that developers spent 34.51% of their time reversing biased actions. Of all the reversal actions, 70.07% involved biases. They defined reversal actions as those that developers need to undo, redo, or discard and measured the total number of actions and time spent on these reversal actions resulting from cognitive biases.

Additionally, the authors assessed the standard tools developers reported for bias mitigation and presented a set of helpful practices to mitigate or address the biases, grouped into six categories.

2.3. Bias in Source Code Review

One area within software engineering where biases have been observed is the process of source code review. With the widespread adoption of distributed version control systems, code reviews often occur through merge requests (or change requests). Murphy-Hill *et al.* (Murphy-Hill et al., 2022) conducted a large-scale analysis of pull requests at Google. The study's primary objective was to examine the influence of demographic attributes, such as gender, ethnicity, and age, on the rate of pushback. Here, "pushback" is defined as "the perception of unnecessary interpersonal conflict in code review, where a reviewer blocks a change request" (Egelman et al., 2020). This concept draws on role congruity theory, which posits that a group member receives negative evaluations when stereotypes about the group are misaligned with the qualities perceived as necessary for success in a specific role. The authors found a significant correlation between the incidence of pushback and the gender, ethnicity, and age of the code's author.

An eye tracking experiment by Ford *et al.* (Ford et al., 2019) has showed that reviewers do look at

information beyond the source code itself while decision to accept or reject a pull request. Reviewers fixated on what Ford *et. al.* referred to as *social indicators* such as the contributor's aviator, display name, and contribution history. They found that the amount of actual fixation was more than what was self-reported by reviewers after the experiment.

In another eye tracking study, Huang *et. al.* (Huang et al., 2020) re-established the existence of correlation reported gender of authors and rate of push back. What's more they demonstrated causation relationship exists. They found both male and female reviewers were harsher on female-authored pull requests compared to male contributors of the same contribution. This implies the reviewer's bias based on gender which contradicts the self-reported behavior in which the reviewers claimed they did not put the gender of the author into consideration.

An interesting finding in this study is that participants admitted their bias towards computer generated code. But even in these instances participants tried to rationalize their bias after the fact with excuses such as that the code generated by computer programs are "too complex and less understandable". This is despite the fact that the actual code was written by human. This shows that even when we acknowledge our biases, we try to justify them rather than gauge their rationality.

2.4. Bias Detection

Detecting cognitive bias refers to identifying the expected likelihood of bias in a decision rather than certainty. Primary procedure for detection has been based on selective exposure experiment. In such methods, participants are surveyed with pre and post experiment questionnaires which will be used as a basis to see if they are biased in the task. Such an approach can be referred to as a static detection since it relies on post activity static analysis as opposed to dynamic techniques that try to detect bias during the decision making. Static detection limits the debiasing techniques that can be employed.

Dynamic Detection is a detection effort whereby the (possible) occurrence of cognitive bias is detected during the deliberation process, before the actual decision is made. While such detection is more likely to help in mitigating biases in the current decision, it is relatively difficult to implement.

Wall *et. al.* (Wall, Blaha, Franklin, & Endert, 2017) proposed a theoretical framework for quantifying interactions and subsequently identifying biased interaction in visual data analytics tools. In a later work (Wall, Blaha, Paul, & Endert, 2019), the authors presented a formative study that implemented this theoretical recommendation. They used their proposed metrics in detecting anchoring bias from interaction data while participants were tasked with an activity purposefully prepared to induce bias. The authors reported an encouraging result towards the ability to detect cognitive bias, in real time, based on interaction behavior.

Nussbaumer et al. (Nussbaumer, Verbert, Hillemann, Bedek, & Albert, 2016) presented an attempt to establish a framework for automatic confirmation bias detection and feedback in criminal intelligence visual analytics. The authors identified software development and research challenges that need to be addressed to realize the proposed framework. The research tasks include developing a detection method, providing meaningful feedback, and evaluating the effectiveness of the proposed methods.

They proposed the use of visual analytics tools, such as MUVA (Kalamaras, Papadopoulos, Drosou, & Tzovaras, 2015) and VALCRI³, to record interaction logs for analysis while users performed typical tasks. This work defines two types of bias detection based on interaction data. The first is a statistical approach that compares biased interactions with unbiased ones. The second is a semantic approach that maps biased and unbiased interactions to cognitive processes and compares these cognitive processes.

2.5. Eye Tracking in Software Engineering

Eye tracking has been used by a number of researchers to understand developer behavior. Bansal et al. (Bansal et al., 2023) present an early attempt at using Large Language Models (LLMs) for programmer attention modeling. They employed gaze trackers and LLMs to build a machine learning model capable

³<https://cordis.europa.eu/project/id/608142>

of predicting how programmers scan source code. To build this model, the authors had 27 programmers perform a Java source code comprehension task, involving 25 randomly selected methods from a total of 68, for 1.5 hours with breaks every 20 minutes. The participants wore eye trackers to record their fixations during the task. Afterward, the participants were asked to write short summaries of the source code. The collected data was used to build an LLM model that takes source code as input and produces the expected scan-path. This model aims to mimic the way programmers would scan and comprehend source code, providing insights into programmer attention patterns.

Aljehane et al. (Aljehane, Sharif, & Maletic, 2023) used eye movement data to identify the expertise level of developers based on how they examine source code. They conducted a quantitative analysis comparing the eye movements of 207 participants, including both novices and experts, while they solved comprehension tasks. The results revealed a notable increase in pupil size among the novice group compared to the experts, suggesting that novices exert greater cognitive effort. Additionally, novices exhibited significantly more fixations and longer gaze durations than experts when comprehending code. Furthermore, a correlation study indicated that programming experience remains a strong predictor of expertise in this eye-tracking dataset, alongside other expertise variables.

3. Proposed Methodology

We propose a research project aimed at collecting data on code reviewer interactions through an experimental setup. We will analyze the collected data using various machine learning algorithms to identify patterns that predict the occurrence of cognitive biases, with a particular focus on prejudice. Our findings will be reported at a peer-reviewed academic conference. This section details the major tasks envisioned for our project.

3.1. Research Design and Data Collection

Our plan involves conducting an experiment to measure participants' performance in source code review tasks. We will first identify open-source pull requests suitable for the experiment based on criteria set by related research works. Then, we will recruit participants and ensure they have the required background to review the code under consideration through a survey. We expect to include both industry professionals and experienced students as developers in our study. Participants will be seated in front of computers and presented with a sequence of pull requests, deciding whether to accept or reject each. Code snippets from pull requests will be randomly assigned labels indicating authorship by different groups of contributors.

We plan to use the Tobii Pro Fusion Eye Tracker⁴ to track participants' gaze. Additionally, we intend to record additional signals to explore the possibility of multimodal analysis including keyboard log, cursor movement and comments/feedback on code and decision outcome. To avoid introducing social desirability bias (Grimm, 2010) by participants, we plan to carefully design our pre-experiment briefing. We will ensure that no mention of bias is made during the initial briefing to prevent influencing participants' behavior. This approach will help us obtain more genuine responses and interactions during the source code review tasks.

Once the experiment is conducted, we will conduct a thorough debriefing session with each participant in the form of an interview. During this debriefing, we will explain the true purpose of the study and the reason for withholding information about bias in the initial briefing. We will discuss the concept of social desirability bias and how it could have impacted their behavior if they had been aware of it from the start.

After explaining the rationale behind the deception, we will seek the participants' consent post hoc. This means obtaining their permission to use the data collected during the experiment, now that they are fully informed about the study's aims and methods. This process ensures ethical transparency and respects participants' autonomy, as they can choose whether to allow their data to be used in the study after being fully informed. By implementing this strategy, we aim to gather more accurate and reliable data while

⁴<https://www.tobii.com/products/eye-trackers/screen-based/tobii-pro-fusion>

maintaining ethical standards in our research. We will obtain approval from our local institutional review board (IRB) on our research methods before recruiting participants and commencing this study.

3.2. Data Analysis

After collecting the necessary data, we aim to explore various machine learning models to find the most effective approaches for bias detection and prediction. We will start with simpler probabilistic models, such as those proposed by Wall *et al.* (Wall *et al.*, 2019), and gradually move towards designing more complex multi-modal and transformer-based deep learning models.

Initially, we will implement and evaluate probabilistic models to establish a baseline performance for bias detection. These models are known for their simplicity and interpretability, which will provide valuable insights into the basic patterns and correlations present in our data.

Next, we will design and evaluate multi-modal machine learning models that can leverage the diverse types of data we are collecting, including eye-tracking data, video recordings, and survey responses. Given the multi-modal nature of our data, we anticipate that these models will outperform simpler, single-modality machine learning algorithms, such as Markov Chains. The integration of different data modalities is expected to capture more nuanced patterns of bias and provide a richer representation of the participants' cognitive processes.

We will also explore the use of advanced deep learning models, such as transformer-based architectures, which have shown significant promise in handling complex, high-dimensional data. These models, with their ability to process sequential and multi-modal data effectively, will be crucial in capturing the temporal and contextual dependencies inherent in our data.

In addition, we will evaluate the performance of various machine learning models, such as Recurrent Neural Networks (RNNs) and Conditional Random Fields (CRF), on the bias prediction task. RNNs are particularly well-suited for sequential data and can capture temporal dependencies, while CRFs are effective for structured prediction tasks, making them valuable for detecting patterns of bias in our data.

Beyond constructing and evaluating machine learning models, we intend to delve into psychological theories that explain human decision-making processes. This interdisciplinary approach will help us understand the underlying mechanisms of cognitive biases and provide a theoretical foundation for interpreting our findings. By integrating insights from psychology, we aim to develop more robust and explainable models that not only predict bias but also offer explanations for why certain patterns of bias occur.

Ultimately, our goal is to develop a comprehensive framework for bias detection and prediction that combines the strengths of machine learning with psychological theories. This framework will enable us to identify and mitigate biases more effectively, contributing to the broader field of cognitive bias research and its applications in various domains.

4. Expected Outcomes and Future Work

Upon successful completion of our study, we expect to have:

- conducted a user study to analyze developers' gaze while reviewing code;
- collected insights from developers on in-group favoritism bias in code review processes; and
- developed a machine learning model capable of predicting the likelihood of bias in code review based on how the reviewer interacts with the code.

The resulting model will leverage the comprehensive multi-modal data we collect, including eye-tracking data, video recordings of facial expressions, and detailed interaction logs, to identify subtle patterns indicative of bias. By analyzing these interaction patterns, our model will be able to provide real-time feedback to reviewers and highlighting potential biases. This will not only enhance the fairness and accuracy of code reviews but also contribute to a more objective and inclusive review process.

The successful implementation of our study will result in a machine learning model that can predict the possibility of bias in code reviews, backed by rigorous data analysis. This model has the potential to help in building a tool that significantly improves the quality and impartiality of software development practices. Such a tool and a more in-depth theoretical interpretation of our finding will be possible extensions of the envisioned study. Once possible occurrence of bias is predicted what to do with the information and how or when to present this information will another question to investigate.

5. Conclusion

This proposal highlights the issue of cognitive bias in source code reviews, affecting software development fairness and effectiveness. Research shows biases based on gender, ethnicity, and age can negatively impact review outcomes. We propose using multi-modal machine learning to address this problem. By analyzing eye-tracking data, video recordings, and interaction logs, we aim to detect bias patterns.

Our experimental design involves recruiting participants to review open-source pull requests, using the Tobii Pro Fusion Eye Tracker, and recording additional signals for multimodal analysis. Pre-experiment briefings and thorough debriefings will ensure ethical transparency and participant consent.

We will explore various machine learning models to develop robust models that predict bias and can be studied to offer insights into cognitive processes. Upon completion, we expect to develop a model for predicting and addressing bias in code reviews. This can be used in creating training programs and tools to help developers recognize and counteract their biases. This research aims to improve software development practices, fostering a more equitable and efficient environment.

6. References

- Aljehane, S. D., Sharif, B., & Maletic, J. I. (2023). Studying developer eye movements to measure cognitive workload and visual effort for expertise assessment. *Proceedings of the ACM on Human-Computer Interaction*, 7(ETRA), 166:1–166:18. Retrieved from <https://doi.org/10.1145/3591135> doi: 10.1145/3591135
- Bansal, A., Su, C.-Y., Karas, Z., Zhang, Y., Huang, Y., Li, T. J.-J., & McMillan, C. (2023). *Modeling programmer attention as scanpath prediction*. Retrieved from <https://doi.org/10.48550/arXiv.2308.13920>
- Boonprakong, N., Chen, X., Davey, C., Tag, B., & Dingler, T. (2023). Bias-aware systems: Exploring indicators for the occurrences of cognitive biases when facing different opinions. In *Proceedings of the 2023 chi conference on human factors in computing systems* (pp. 1–19). Retrieved from <https://doi.org/10.1145/3544548.3580917> doi: 10.1145/3544548.3580917
- Chattopadhyay, S., Nelson, N., Au, A., Morales, N., Sanchez, C., Pandita, R., & Sarma, A. (2022). Cognitive biases in software development. *Communications of the ACM*, 65(4), 115–122. Retrieved from <https://doi.org/10.1145/3517217> doi: 10.1145/3517217
- Egelman, C. D., Murphy-Hill, E., Kammer, E., Hodges, M. M., Green, C., Jaspan, C., & Lin, J. (2020). Predicting developers' negative feelings about code review. In *Proceedings of the acm/ieee 42nd international conference on software engineering* (pp. 174–185). Seoul, South Korea: ACM. Retrieved from <https://doi.org/10.1145/3377811.3380414> doi: 10.1145/3377811.3380414
- Ford, D., Behroozi, M., Serebrenik, A., & Parnin, C. (2019). Beyond the code itself: How programmers really look at pull requests. In *2019 ieee/acm 41st international conference on software engineering: Software engineering in society (icse-seis)* (pp. 51–60). Retrieved from <https://doi.org/10.1109/ICSE-SEIS.2019.00014> doi: 10.1109/ICSE-SEIS.2019.00014
- Grimm, P. (2010, December). Social desirability bias. In *Wiley international encyclopedia of marketing*. Chichester, UK: John Wiley & Sons, Ltd.
- Huang, Y., Leach, K., Sharafi, Z., McKay, N., Santander, T., & Weimer, W. (2020). Biases and differences in code review using medical imaging and eye-tracking: Genders, humans, and machines. In *Proceedings of the 28th acm joint meeting on european software engineering confer-*

- ence and symposium on the foundations of software engineering* (pp. 456–468). Retrieved from <https://doi.org/10.1145/3368089.3409681> doi: 10.1145/3368089.3409681
- Kalamaras, I., Papadopoulos, S., Drosou, A., & Tzovaras, D. (2015). MoVA: A visual analytics tool providing insight in the big mobile network data. In *IFIP advances in information and communication technology* (pp. 383–396). Cham: Springer International Publishing.
- Mohanani, R., Salman, I., Turhan, B., Rodriguez, P., & Ralph, P. (2020). Cognitive biases in software engineering: A systematic mapping study. *IEEE Transactions on Software Engineering*, 46(12), 1318–1339. Retrieved from <https://doi.org/10.1109/TSE.2018.2877759> doi: 10.1109/TSE.2018.2877759
- Murphy-Hill, E., Jaspan, C., Egelman, C., & Cheng, L. (2022). The pushback effects of race, ethnicity, gender, and age in code review. *Communications of the ACM*, 65(3), 52–57. Retrieved from <https://doi.org/10.1145/3474097> doi: 10.1145/3474097
- Nussbaumer, A., Verbert, K., Hillemann, E.-C., Bedek, M. A., & Albert, D. (2016). A framework for cognitive bias detection and feedback in a visual analytics environment. In *2016 european intelligence and security informatics conference (eisic)* (pp. 148–151). Retrieved from <https://doi.org/10.1109/EISIC.2016.038> doi: 10.1109/EISIC.2016.038
- Pronin, E., Lin, D. Y., & Ross, L. (2002). The bias blind spot: Perceptions of bias in self versus others. *Personality and Social Psychology Bulletin*, 28(3), 369–381. doi: 10.1177/0146167202286008
- Tversky, A., & Kahneman, D. (1974). Judgment under uncertainty: Heuristics and biases. *Science*, 185(4157), 1124–1131.
- Wall, E., Blaha, L., Paul, C., & Endert, A. (2019). A formative study of interactive bias metrics in visual analytics using anchoring bias. In D. Lamas, F. Loizides, L. Nacke, H. Petrie, M. Winckler, & P. Zaphiris (Eds.), *Human-computer interaction – interact 2019* (Vol. 11747, pp. 555–575). Springer International Publishing. Retrieved from https://doi.org/10.1007/978-3-030-29384-0_34 doi: 10.1007/978-3-030-29384-0_34
- Wall, E., Blaha, L. M., Franklin, L., & Endert, A. (2017). Warning, bias may occur: A proposed approach to detecting cognitive bias in interactive visual analytics. In *2017 IEEE conference on visual analytics science and technology (vast)* (pp. 104–115). Retrieved from <https://doi.org/10.1109/VAST.2017.8585669> doi: 10.1109/VAST.2017.8585669

Ethical Integration in Computer Science Education: Leveraging Open Educational Resources and Generative Artificial Intelligence for Enhanced Learning

Ranjidha Rajan
Computer Science Department
MSU Denver
rranjidh@msudenver.edu

Renato Cortinovis
Freelance Researcher
Italy
rmcortinovis@gmail.com

Abstract

In contemporary society, the extensive integration and dependence on computerized systems are evident across various aspects of everyday life. The training and education of the developers responsible for these systems should encompass more than just technical skills: a profound grasp of ethical considerations and the societal impact of their work is considered essential. This paper outlines an experimental approach utilizing adapted and newly developed Open Educational Resources (OER) to familiarize computer science students with the ACM Code of Ethics and Professional Conduct. These OERs, employing an underlying reusable pattern, propose assignments mandating the integration of ethical considerations into software development practices within an Inquiry-Based Learning (IBL) framework.

In the scope of these assignments, this study conducted a preliminary investigation into leveraging Generative Artificial Intelligence (gen-AI) to augment student learning and self-efficacy. This was achieved through the analysis of the data gathered from the assignments evaluation and a survey encompassing Likert scale ratings and open-ended inquiries. Factor analysis helped identifying the key themes 'Use', 'Tool Efficiency (TE)', 'Concerns (C)', 'Academic Integrity (AcI)', and 'Tool Convenience (TC)', which reflect various aspects of student engagement and perceptions of gen-AI tools. Structural Equation Modelling (SEM) further explored the relationships among these themes, suggesting that a combined 'TE' and 'TC' factor significantly enhanced user engagement with gen-AI tools. Conversely, the combined 'Concerns' and 'Academic Integrity' factors, i.e., concerns about reliability and academic dependency, did not significantly inhibit the willingness of the students to adopt gen-AI technologies.

Preliminary findings also indicate that gen-AI exhibits notable efficacy among students of moderate proficiency, albeit demonstrating underutilization among academically advanced students. Conversely, students categorized as lower-ranked tend to utilize gen-AI without exercising critical discernment. These results underscore the necessity to carefully tailor these OER to accommodate diverse student proficiency levels, thereby maximizing their educational efficacy.

1. Introduction

In contemporary society, the proliferation and reliance upon computerized systems pervades most facets of daily life. It is widely acknowledged that the training and education of the developers behind these systems extend beyond mere technical proficiency: a critical understanding of ethics and societal implications is deemed imperative. The ACM Code of Ethics and Professional Conduct is an invaluable resource that synthesises the key aspects in this regard. However, in our previous experience, many students who are strongly technically inclined tend to underestimate the importance of considering ethics and impact on social good of their work. Therefore, a first research question (RQ) for this study was:

RQ1: What strategies could help familiarizing technically inclined computer science students with the ACM Code of Ethics and Professional Conduct?

As we will discuss, in response to this research question, assignments were designed to integrate ethical considerations into software development practices within an Inquiry-Based Learning (IBL) framework. Students were tasked with applying the ACM Code principles across progressively complex and broader scenarios. In order to support the students in these activities, we experimented the use of gen-AI tools, which prompted the following further research inquiries:

RQ2: Which are the main factors influencing positively or negatively the students' attitude in using gen-AI tool for inquiry-based learning assignments in CS classrooms?

RQ3: What is the correlation between the quality of students prompts to genAI tools and their performances in the specific IBL-base assignment and overall performance in learning?

The subsequent sections describe the learning assignments devised, the methodological approach employed to address our research inquiries, and the preliminary findings gleaned from the initial two pilot studies conducted.

2. Learning assignments

In order to familiarize students with the ACM Code, we mainly followed the approach of Fiesler et al. (2021), and Peck (2017) among others, integrating ethical considerations into traditional programming design and development assignments. This is also very similar to the CSG_ED approach of Goldweber et al. (2013), 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, climate change, human rights, etc.

We considered that the IBL model was particularly appropriate to incrementally foster a deeper and broader understanding of ethical considerations and help students developing an autonomous research-oriented attitude. For this study we adopted the IBL5E variation (Duran, L. and Duran, E., 2004) articulated in the phases described in Table 1.

Phase	Purpose
Engage	Create interest and stimulate curiosity. Set learning within a meaningful context. Raise questions for inquiry.
Explore	Provide experience of the phenomenon or concept. Explore and inquire into students' questions and test their ideas. Investigate and solve problems.
Explain	Introduce conceptual tools that can be used to interpret the evidence and construct explanations of the phenomenon. Construct multi-modal explanations and justify claims in terms of the evidence gathered. Compare explanations generated by different students/groups.
Elaborate	Use and apply concepts and explanations in new contexts to test their general applicability. Reconstruct and extend explanations and understanding using and integrating different modes, such as written language, diagrammatic and graphic modes, and mathematics.
Evaluate	Provide an opportunity for students to review and reflect on their own learning and new understanding and skills. Provide evidence for changes to students' understanding, beliefs and skills.

Table 1 – Phases of the IBL5E model.

We have abstracted a generic schema for assignments based on the IBL5E model with guided and open enquiry, where gen-AI has been integrated into the Elaborate phase, to extend/improve artefacts previously developed in the Explore/Explain phases, as follows:

1. ENGAGE - Discuss the importance of ethics in computer science; critically read the ACM Code of Ethics and Professional Conduct; debate motivating and intriguing ethical dilemmas to realize that the application of the ACM Code is not necessarily straightforward: group decision making in autonomous vehicles (Awad et al., 2018), matching decisions to relevant aspects of the Code.
2. EXPLORE – Preliminarily develop an artifact [Program/UML, Diagram/ERD] about a system [on a specified topic].

3. EXPLAIN [guided enquiry] - Extend the previous artifact to cater for specified ethical implications – free support from the Internet, excluding gen-AI.
4. ELABORATE [open enquiry] – Freely identify further extensions/improvements to the artifact, integrating additional (unspecified) aspects concerning ethics – free support from gen-AI tools. Report your prompts to the tool.

The next two sections provide pertinent excerpts from the assignments utilized in our pilot studies, exemplifying the overarching schema just described.

2.1. Personalized Ads Programming

This first assignment, just slightly adapted from an existing OER published by Fiesler et al. (2021), asks students with very basic programming skills to incrementally develop a program to serve personalized ads on a social platform. In a sequence of scenarios of increasing complexity, the ads program prompts the user for information (in a real situation it would automatically extract information from the profile and posts of the users) and then return text that describes ads based on their inputs. In a first scenario, for example, the program provides text advertising dog food if the user has a dog. In another scenario, the program provides advertisements for any product that includes dogs to extrovert people, and advertisements for any product that includes cats to introvert people. In a third scenario the program provides advertisements about more or less expensive products, based on the age of the user and the estimated average income for the zip code where she lives. Here is a meaningful extract of the assignment:

EXPLAIN – (Structured enquiry)

Explain in a short report, helping yourself also with information you may search on the Internet (without using AI tools), how personalized ads work and the ethical implications, by answering the questions:

- How do you feel about these kinds of inferences being used to influence your behaviour?
- Are there ethical and unethical ways to use the technology of personalized advertisement?
- [...]

ELABORATE – (partially guided enquiry)

- Using the support of ChatGPT (or any genAI tool), create a short report (with your own words, avoiding cut & paste) where you:
 - identify a new case where a personalized ad led to ethical dilemmas,
 - pinpoint the ethical issues involved,
 - match them to the ACM ethical code,
 - and discuss potential solutions.
- Include as an annex the specific prompts you submitted to ChatGPT and its responses.

2.2. Ethical Database Design

A second newly developed assignment asks students studying database design to develop an Entity Relationship Diagram (ERD) for a simplified database of a medical clinic, incrementally enhancing it by integrating aspects related to ethics. Here is a meaningful extract of the assignment:

EXPLORE – Look into how ethical guidelines from the ACM Code of Ethics could affect the design of a database for a personalized healthcare clinic. Write a short report about it.

EXPLAIN – Sketch a first ERD of the database, trying to incorporate relevant ethical aspects. Provide suitable comments to the schema, in particular making explicit any impact of the Code of Ethics on the ERD. You are allowed to make use of the Internet, excluding gen-AI tools.

ELABORATE – Using ChatGPT or similar gen-AI tools, identify further enhancements to the ERD by integrating additional ethical considerations. Document the prompts (queries) submitted to the AI tool, present the extended ERD resulting from these interactions, and

provide clear comments showing how specific features of the ERD are linked to corresponding items in the ACM Code.

3. Methodology: data collection and analysis

To address RQ1, an evaluation of the students' assignment submissions was conducted to gauge their comprehension of ethical dimensions and their proficiency in effectively incorporating these principles into their software development tasks. For example, in the second pilot study, the assessment schema employed was as follows:

EXPLORE, EXPLAIN: check ERD correctness, discussion of relevant ethical considerations, integration of ethical considerations into ERD.

ELABORATE: check ethical considerations extensions, corresponding ERD extensions.

To address RQ2, we collected data from Likert questions concerning the students' attitude in using gen-AI tools, and their answers to open-ended questions about their experience with the assignments. Data from Likert questions were analysed and refined with factor analysis and subsequent thematic analysis to reveal the main factors affecting their perceptions.

In this study, we employed Principal Component Analysis (PCA) as the extraction method for factor analysis to identify underlying dimensions within the dataset (Wetzel, 2012). We utilized Promax rotation with Kaiser Normalization to allow the factors to be correlated, enhancing interpretability in a framework where constructs may be interrelated (Grieder & Steiner, 2022). The selection of variables and the number of factors retained were based on their ability to meaningfully explain the covariance among observed variables.

Using Structural Equation Modelling (SEM), the interrelationships among these themes were further elucidated (Goldberger, 1972). The Maximum Likelihood (ML) estimation method was used to identify the best-fitting model, a standard approach in SEM due to its efficiency and robustness.

Concerning RQ3, we collected the prompts submitted by the students to the gen-AI tool, which were classified as Descriptive, Comparative, Inquisitive/Exploratory, Ethical/Philosophical Inquiry, Case Study, Focused, and Instructional. Descriptive prompts aim at eliciting detailed narratives or explanations (Cave et al., 2020). Comparative are prompts that encourage comparison between concepts or examples (Sutton & Barto, 2018). Inquisitive/Exploratory are prompts designed to probe deeper understanding or exploration of a topic (Lake et al., 2018). Ethical/Philosophical Inquiry are prompts that delve into ethical considerations or philosophical questions (Bostrom, 2014). Case Study Focused are prompts asking for specific examples, case studies, or applications (Silver et al., 2016). Instructional are prompts that guide the AI in performing a specific task or generating content in a certain way, reflecting the few-shot learning capabilities mentioned in AI research (Schick & Schütze, 2022).

Accordingly, each prompt was analysed using specific criteria where descriptive prompts asked for explanations, comparative prompts involved comparisons, inquisitive prompts sought understanding or exploration, ethical prompts focused on moral or ethical considerations, and instructional prompts provided summaries or instructions. Walter (2024) claims that prompt classification is crucial for understanding student learning with gen-AI tools, as it enables the identification of specific areas where the tool enhances educational outcomes, allowing for targeted improvements and better support for diverse learning needs. The prompt classification process was automated based on text analysis to label the prompts accordingly. The classified data were then visualized using a pivot table and bar chart to illustrate the distribution of prompt classifications by grade, providing insights into the cognitive and analytical skills development of students at different educational levels. Finally, we correlated the classification scores with their grades in the assignment.

4. Preliminary results and discussion

The analysis was carried out on a limited sample of 27 students. Concerning RQ1, the assignments' evaluation showed that the proposed generic schema, providing students with the opportunity to concretely apply the ACM code of ethics in progressively complex scenarios and with tools of increasing power, supported the students in developing an increasing level of understanding moving

through the assignment phases. Students also explicitly appreciated the integration of IBL5E and gen-AI. One student stated, for example: “*I appreciate the school's interest in embracing innovations and its dedication to enhancing teaching methods*”, while another commented “*These new teaching methods, including the use of AI, should be incorporated more frequently into other lessons*”.

Concerning RQ2, the examination of Likert scale data through factor analysis (Figure 1) facilitated the identification of several themes. The pattern matrix resulting from Principal Component Analysis (PCA) (Wetzel, 2012) with Promax rotation and Kaiser normalization reveals the underlying factor structure of the dataset (Grieder & Steiner, 2022).

Here is a meaningful extract of the assignment:

Pattern Matrix^a

	Component				
	1	2	3	4	5
Q18	.912				
Q12	.856				
Q19	.842				
Q21	.718				
Q20	.600				
Q3		.899			
Q5		.784			
Q7		.775			
Q1		.728			
Q13			.901		
Q10			.834		
Q15				.925	
Q14				.694	
Q11					.816
Q8					.788

Extraction Method: Principal Component Analysis.
Rotation Method: Promax with Kaiser Normalization.
a. Rotation converged in 6 iterations.

Figure 1 – Pattern matrix after removing irrelevant questions

Five distinct components were extracted, each representing a unique factor. Component 1 shows strong loadings for Q12, Q18, Q19, Q20, and Q21, indicating a shared underlying factor. Q2, Q4, Q6, Q9, Q16, Q17 were removed for better factor loading. Component 2 is defined by high loadings on Q5, Q3, Q1, and Q7, suggesting another common factor. Component 3 includes significant loadings for Q10, and Q13, highlighting a third distinct factor. Component 4 is characterized by loadings on Q15, and Q14, with Q16 showing a particularly high negative loading. Component 5 has a notable loading for Q8 and Q11 pointing to additional unique factors. Each component represents a different underlying factor extracted from the data set, where the numbers indicate the strength of the association between each question and the corresponding component. For instance, Q18 has a strong loading on Component 1 (0.912), suggesting it is closely related to that factor, while Q3 has a high loading on Component 2 (0.899). The matrix indicates which questions are grouped together under each component, helping to identify patterns and underlying structures in the data. The rotation method used ensures that the components are more interpretable by allowing them to be correlated. This pattern matrix effectively elucidates the factor structure, with each component representing a distinct underlying dimension measured by the variables in the dataset.

The resulting themes identified were 'Use', 'Tool Efficiency (TE)', 'Concerns (C)', 'Academic Integrity (AcI)', and 'Tool Convenience (TC)'. These themes encompass diverse dimensions of student engagement and perceptions pertaining to gen-AI tools. The survey questions were grouped based on the previous themes identified. The sample for survey question grouping for TE is given in Figure 2.

Tool Efficiency (TE)

ChatGPT provided quick and relevant responses to my inquiries. (Q1)

Using ChatGPT saved me time during the research process. (Q3)

I felt more confident about my research findings after using ChatGPT. (Q5)

I would prefer using ChatGPT over traditional research methods for future assignments. (Q7)

Figure 2 – Sample Thematic factors based on factor loadings

SEM analysis was conducted using the maximum likelihood estimation method. It yielded a log likelihood of -84.41, suggesting a relatively good model fit (Figure 3). The variable "te_tc" demonstrated a strong positive and statistically significant relationship with the dependent variable, with a coefficient of 0.88 and a p-value of 0.001. This indicates that changes in "te_tc" are likely to have a significant impact on "use". In other words, the amalgamation of 'TE' and 'TC' significantly impact on user engagement with gen-AI tools. Conversely, the variable "c_ai" showed a positive relationship with a coefficient of 0.32, but this was not statistically significant with a p-value of 0.226, showing that the influence by "c_ai" on "Use" is not strongly supported by the current sample size of data. That is, the amalgamation of 'C' and 'AcI' factors, indicative of apprehensions regarding reliability and academic dependence, did not markedly impede students' propensity to adopt gen-AI technologies.

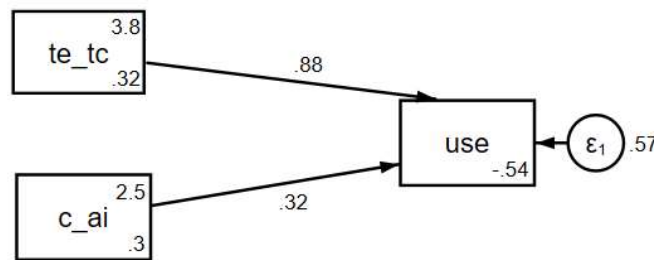


Figure 3 – Structural Equation Modelling

Concerning RQ3, the categorization of students' prompts to ChatGPT resulted in the identification of three distinct groups: firstly, the highest-performing students effectively showcased a tangible integration of the ACM Code principles into their database design, solely relying on their own capabilities without resorting to ChatGPT assistance. Secondly, students with moderate proficiency levels attained comparable outcomes by utilizing ChatGPT, engaging in meaningful interactions that facilitated their progress. Lastly, a minority of students with very limited abilities demonstrated minimal interaction with GenAI: they just submitted the whole assignments directly to the tool without personalized interventions. This supports the claim that gen-AI exhibits notable efficacy among students of moderate proficiency, albeit demonstrating underutilization among academically advanced students. Conversely, students categorized as lower-ranked tend to utilize gen-AI without exercising critical discernment. These results underscore the necessity to carefully tailor these OER to accommodate diverse student proficiency levels, thereby maximizing their educational efficacy.

The analysis of prompt classifications across different grades revealed distinct trends in the focus and complexity of student inquiries (Figure 4). Grade 6 exhibits a higher frequency of descriptive prompts, reflecting a focus on foundational understanding and detailed descriptions. Ethical and philosophical inquiries are evenly distributed among Grades 4, 6, and 8, indicating a consistent engagement with moral and ethical considerations across these levels. Grade 8, however, shows a notable increase in inquisitive and exploratory prompts, suggesting a shift towards more critical and analytical thinking as students advance. Instructional prompts are unique to Grade 4, perhaps indicative of an emphasis on summarization and concluding thoughts at this stage. Interestingly, unclassified prompts appear predominantly in Grades 4 and 8, with Grade 8 having the highest number, which could reflect the more

open-ended and complex nature of discussions at this level. These findings underscore the progression in cognitive and analytical skills development as students move through different educational stages.

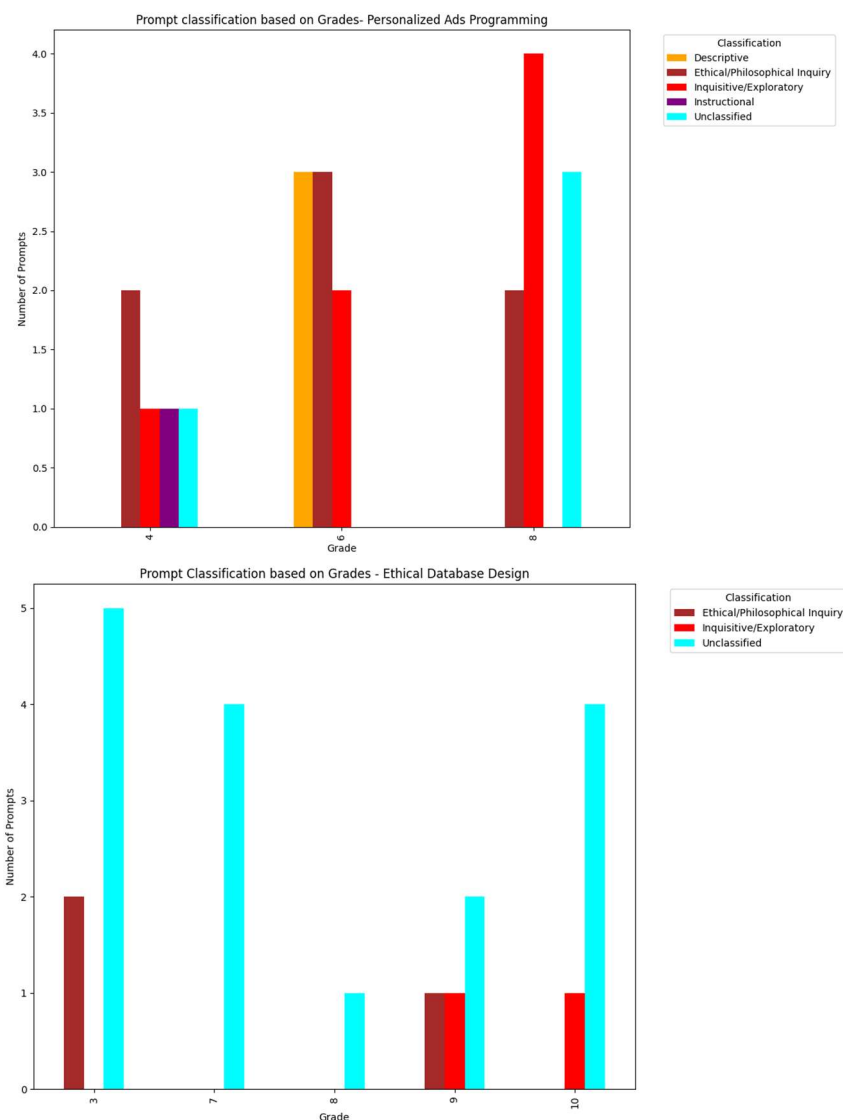


Figure 4 – Prompt classification samples

The data from the database design assignment and their analysis findings show that the majority of prompts at lower grades, especially Grade 3, remain unclassified, indicating a need for more specific guidance or focus in the questions posed. Higher grades tend to have more focused inquiries, with Grade 8 showing a balanced mix of inquisitive and ethical inquiries. This likely reflects the naturally expected deeper reasoning exhibited by the most proficient students. However, further research is needed to determine how best GenAI could contribute to improving critical thinking and understanding across all students.

This analysis helps in understanding the developmental trends in cognitive and analytical skills among students, and how they engage with ethical considerations in their academic tasks.

5. Conclusions

The utilization of an existing OER in the first pilot proved pivotal for the undertaken activity, underscoring the significance of OER in both research and educational endeavours. This prompted our decision to openly publish the resource developed in the second pilot as an OER, with further intentions to create additional resources, capitalizing on the identified seemingly effective overarching framework. The student reception of the assignments outlined has been positive, with active engagement noted and an explicit appreciation for the IBL5E model and the incorporation of gen-AI. The data analysis

identified themes such as 'Use', 'Tool Efficiency', 'Concerns', 'Academic Integrity', and 'Tool Convenience', which represent various aspects of student engagement and perceptions of generative AI tools. SEM analysis revealed that the combination of 'Tool Efficiency' and 'Tool Convenience' significantly enhanced user engagement, while 'Concerns' and 'Academic Integrity' did not substantially deter students from adopting generative AI technologies. These insights highlight the complex interplay between efficiency, convenience, and apprehensions in shaping students' adoption of generative AI tools. Nonetheless, despite garnering some preliminary findings, these outcomes necessitate validation and expansion through subsequent studies involving students from diverse contexts, larger sample sizes, and varied assignments.

6. References

- Awad, E., Dsouza, S., Kim, R. *et al.* (2018). The Moral Machine experiment. *Nature* 563, 59–64. <https://doi.org/10.1038/s41586-018-0637-6>
- Bostrom, N. (2014). *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press.
- Cave, S., Dihal, K. & Dillon, S. (2020). *AI Narratives: A History of Imaginative Thinking about Intelligent Machines*. Oxford University Press.
- Duran, L., and Duran, E. (2004). The 5E Instructional Model: A Learning Cycle Approach for Inquiry-Based Science Teaching. *The Science Education Review*, 3(2), p49–58
- Fiesler, C., Friske, M., Garrett, N., Muzny, F., Smith, J., and Zietz, J. (2021). Integrating Ethics into Introductory Programming Classes. *Proceedings of the ACM SIGCSE Conference on Computer Science Education*.
- Goldberger, A. S. (1972). Structural Equation Methods in the Social Sciences. *Econometrica*, 40(6), 979–1001. <https://doi.org/10.2307/1913851>
- 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.
- Grieder, S., & Steiner, M. D. (2022). Algorithmic jingle jungle: A comparison of implementations of principal axis factoring and promax rotation in R and SPSS. *Behavior research methods*, 54(1), 54–74.
- Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. (2017). Building machines that learn and think like people. *Behavioural and Brain Sciences*, 40, E253.
- Peck, E. (2017, July 5). The Ethical Engine: Integrating Ethical Design into Intro Computer Science. Blog, Bucknell HCI, 5.
- Schick, T., & Schütze, H. (2022). True Few-Shot Learning with Prompts—A Real-World Perspective. *Transactions of the Association for Computational Linguistics*; 10 716–731.
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), 484-489.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.
- Walter, Y. (2024). Embracing the future of Artificial Intelligence in the classroom: The relevance of AI literacy, prompt engineering, and critical thinking in modern education. *International Journal of Educational Technology in Higher Education*, 21(1), 1-29. <https://doi.org/10.1186/s41239-024-00448-3>
- Wetzel A. P. (2012). Factor analysis methods and validity evidence: a review of instrument development across the medical education continuum. *Academic medicine: journal of the Association of American Medical Colleges*, 87(8), 1060–1069. <https://doi.org/10.1097/ACM.0b013e31825d305d>

Intention Is All You Need

Advait Sarkar

Microsoft Research, University of Cambridge, University College London
advait@microsoft.com

Abstract

Among the many narratives of the transformative power of Generative AI is one that sees in the world a latent nation of programmers who need to wield nothing but intentions and natural language to render their ideas in software. In this paper, this outlook is problematised in two ways. First, it is observed that generative AI is not a neutral vehicle of intention. Multiple recent studies paint a picture of the “mechanised convergence” phenomenon, namely, that generative AI has a homogenising effect on intention. Second, it is observed that the formation of intention itself is immensely challenging. Constraints, materiality, and resistance can offer paths to design metaphors for intentional tools. Finally, existentialist approaches to intention are discussed and possible implications for programming are proposed in the form of a speculative, illustrative set of intentional programming practices.

1. The “Intention Is All You Need” Picture of Programming with Generative AI

What is programming? Blackwell’s succinct and influential definition is that programming is any activity exhibiting the property *“that the user is not directly manipulating observable things, but specifying behaviour to occur at some future time”* (Blackwell, 2002). Behaviour is specified through an interface, commonly a notation, which we call a programming language. Therein lies the source and objective of all research in the psychology and design of programming: the study of the use and improvement of the interfaces, notations, and languages for specifying behaviour.

The value of such study is called into question with the introduction of Generative Artificial Intelligence (GenAI), which can be defined as any *“end-user tool [...] whose technical implementation includes a generative model based on deep learning”*.¹ GenAI captures the relationships between natural language specifications of behaviour, and the translations of that behaviour into programming notation, implicit in enormous training datasets. The power of translation thus captured can be stochastically replayed on demand (Blackwell, 2020). What could this mean for research in the user-centred design of programming languages? One perspective anticipates nothing less than its obsolescence:

*“The programming barrier [with GenAI] is incredibly low. We have closed the digital divide. Everyone is a programmer now - you just have to say something to the computer.”*²

*“Up until now, in order to create software, you had to be a professional software developer. You had to understand, speak and interpret the highly complex, sometimes nonsensical language of a machine that we call code. [...] But with GenAI] We have struck a new fusion between the language of a human and a machine. With Copilot, any person can now build software in any human language with a single written prompt. [...] going forward, every person, no matter what language they speak, will also have the power to speak machine. Any human language is now the only skill that you need to start computer programming.”*³

¹There is no definitional consensus on this term. A rationale for the definition adopted here is given by Sarkar (2023d).

²Huang, 2023. (Source: <https://www.reuters.com/technology/ai-means-everyone-can-now-be-programmer-nvidia-chief-says-2023-05-29/>, accessed July 2024.)

³Dohmke, 2024. (Source: https://www.ted.com/talks/thomas_dohmke_with_ai_anyone_can_be_a_coder_now, accessed July 2024.)

“Since the launch of GPT-4 in 2023, the generation of whole apps from simple natural language requirements has become an active research area. [...] Our vision is that by 2030 end users will build and deploy whole apps just from natural requirements.”⁴

“Programming will be obsolete. [...] the conventional idea of ‘writing a program’ is headed for extinction [...] all programs in the future will ultimately be written by AIs, with humans relegated to, at best, a supervisory role. [...] The engineers of the future will, in a few keystrokes, fire up an instance of a four-quintillion-parameter model that already encodes the full extent of human knowledge (and then some), ready to be given any task required of the machine.”⁵

The promise of GenAI for programming, therefore, is to transform programming into an activity where expertise in specialised notations and languages for specifying behaviour are unnecessary. One merely has to say what one wishes the program to do, and GenAI does the rest. The interaction design challenges of programming are solved.⁶ Intention is all you need.

There are many problems with this picture. There are compelling reasons for continuing to engage with formal notations, even and perhaps especially when GenAI is in play (Sarkar, 2023d). Moreover, language in general, and the language of prompts used to direct GenAI in particular, is most certainly not a flawless, transparent route for the expression of intent. Johnny can’t prompt (Zamfirescu-Pereira et al., 2023). Johnny can’t figure out what level of abstraction to write his prompts in, either (Liu et al., 2023; Sarkar et al., 2022). Thinking about prompting is hard for Johnny, and thinking about thinking about prompting is hard, too (Tankelevitch et al., 2024). Prompting “dialects” might evolve in much the same way as specialised uses of natural language do in domains such as scientific and legal communication, through disciplinary norms and professional consensus, and to acquire such language will require users to undergo analogous processes of disciplinary and professional acculturation (Sarkar, 2023d). But these problems are not the primary concern in this paper.

There is a rather more fundamental pair of problems with the idea that intention is all you need (to program with GenAI): it assumes that GenAI does not interfere with intention. Moreover, it takes for granted that intentions are easy to form. Both premises will be questioned in turn.

2. Mechanised Convergence: The Homogenising Effect of AI on Intention

Contrary to not interfering with intention, AI supplies intention. It does so in a way that can be described as *mechanised convergence* (Sarkar, 2023b), drawing on Walter Benjamin’s concept of mechanical reproduction (W. Benjamin, 1935). Mechanised convergence describes the idea that the automation or mechanisation of work leads to a convergence in the space of outputs. Standardisation is necessary for factory logic to function. For a machine to be repeatable at speed, its inputs and outputs need to be repeatable at speed, too. You can have any colour as long as it’s black.

Here is some of the evidence that GenAI has a mechanised convergence effect:

- Predictive text encourages predictable writing (Arnold et al., 2020). In an image captioning task, when participants use predictive text entry systems, captions written with suggestions are shorter and use fewer words than the system does not predict. A similar effect occurs in identifier names when programmers use a GenAI tool such as GitHub Copilot to assist them in writing code (Lee

⁴Robinson et al. (2024)

⁵Welsh (2022)

⁶One is reminded of similar claims made during the early days of spreadsheets or about any number of visual programming languages. E.g., Benjamin Rosen, a PC industry analyst for Morgan Stanley, later a key funder of Lotus and Compaq, noted in 1979 that *“In minutes, people who have never used a computer are writing and using programs [...] the user need not know anything about computers or programming in order to derive Visicalc’s benefits. You construct a Visicalc program much as you would define a problem on a sheet of paper or a blackboard”* (Rosen, 1979).

et al., 2024). This effect occurs even when the suggestions are merely visible and not actionable (i.e., cannot be accepted using a keyboard shortcut).

- Similarly, a large study (n=293) of participants writing short stories with varying degrees of AI assistance found that exposure to GenAI “ideas” leads to a reduced diversity of content (Doshi & Hauser, 2023). Participants exposed to even a single GenAI suggestion produce stories similar to the average of the other stories in the same experimental condition.
- A large study (n=758) of strategy consultants at BCG examined the effects of ChatGPT use on a set of consultancy tasks (Dell’Acqua et al., 2023). The majority of participants with access to ChatGPT retain a very high amount of its response – typically around 90% – in their submitted work. Participants without access to ChatGPT produce ideas with more conceptual variation than those with access, showing that usage of ChatGPT reduces the range of ideas generated. The variation across responses produced by ChatGPT is smaller than what human participants produce on their own.
- Large language models have a “homogenization effect” on creative ideation (Anderson et al., 2024). In a creative ideation task, participants produce less semantically distinct ideas when using ChatGPT. Moreover, participants feel less responsible for ideas produced with ChatGPT assistance.
- A large study (n=115) finds that conversational search built on GenAI increases selective exposure compared to conventional search (Sharma et al., 2024). Users engage in more biased information querying with conversational search, and the bias is exacerbated when the model is itself “opinionated” to reinforce the user’s views. The authors call this a “generative echo chamber”.
- Similarly, a large study (n=1506) of co-writing with GenAI found that using an “opinionated” language model affects the opinions expressed in participants’ writing and moreover, actually shifts their opinions as measured in a subsequent attitude survey (Jakesch et al., 2023). A related effect, termed “drifting”, has been observed in novice programmers, where the tendency to accept and adapt code generated by the system leads programmers away from a correct solution (Prather et al., 2023).

Mechanised convergence signals an odd reversal (or perhaps intensification) of Dennett’s “intentional stance” (Dennett, 1971), wherein we not only ascribe intention to these systems but also delegate it, sometimes wilfully, other times unknowingly.

The intention supplied by GenAI through mechanised convergence has a complex source, combining influences of its training data, and the biases and heuristics encoded by the system developers. However at its core, mechanised convergence is the ultimate outcome of the old statistical logics of uncovering underlying natural “laws” (Blackwell, 2020; Sarkar, 2023a). The statistical machine eliminates “noise” (diversity) to predict “signal” (uniformity). The statistical machine is the triumph of the Enlightenment aesthetic faith in nature’s having an underlying elegance or simplicity that is obscured from view by imperfect forms. It should come as no surprise that machines that are built to search for Platonic ideals reflect back to us a mechanically converged picture of the world, making quiddity of haecceity.

It is important to note that the effect on intent as demonstrated in these studies is an *aggregate tendency* that likely does not square with individual phenomenal perceptions of GenAI use. At the granularity of individual interactions, the experience of GenAI might well be as a passive translator, not active supplier, of intent. The nudge towards standardised, centralised, averaged, generic, and statistically optimised answers may be barely perceptible. Yet the data demonstrates that these nudges in fact have a measurable cumulative effect on knowledge work.

As Winner sets out, artefacts have politics (Winner, 1980). The design features of a technology enable certain forms of power, and the decision to adopt a particular technology requires certain power relations to be enacted. Putting it in Winner’s terms, convergence is the politics of AI, the artefact.

As McLuhan sets out, the medium is the message (McLuhan, 1964). There is an effect of a particular medium, be it typography, radio, or television, on the human sensorium that is quite distinct from any particular content being conveyed through that medium. The effect of the medium overwhelms the content and makes it incidental. Putting it in McLuhan's terms, convergence is the message of AI, the medium.

McLuhan predicted that electric technology and programmability would reverse the convergence tendencies of factory logic. He gives the example of a programmable tailpipe machine: *“A new automatic machine for making automobile tailpipes [...] starting with lengths of ordinary pipe, it is possible to make eighty different kinds of tailpipe in succession, as rapidly, as easily, and as cheaply as it is to make eighty of the same kind. And the characteristic of electric automation is all in this direction of return to the general-purpose handicraft flexibility that our own hands possess. The programming can now include endless changes of program.”*

Taken to its logical conclusion, McLuhan makes a claim that is strikingly similar to the narrative that intention is all you need: *“the older mechanistic idea of “jobs,” or fragmented tasks and specialist slots for “workers,” becomes meaningless under automation. [...] The very toil of man now becomes a kind of enlightenment. As unfallen Adam in the Garden of Eden was appointed the task of the contemplation and naming of creatures, so with automation. We have now only to name and program a process or a product in order for it to be accomplished. Is it not rather like the case of Al Capp's Schmoos? One had only to look at a Schmoos and think longingly of pork chops or caviar, and the Schmoos ecstatically transformed itself into the object of desire. Automation brings us into the world of the Schmoos. The custom-built supplants the mass-produced.”* As we have seen, the vast programmability of GenAI does not necessarily result in a *“return to [...] general-purpose handicraft flexibility”*, rather, it has enabled a newer, subtler, and more pervasive form of the *“fragmentalized and repetitive routines of the mechanical era”*. Through the mechanised convergence of knowledge work through GenAI, the principle of interface design becomes WYGIWYG – What You Get Is What You Get.

Postman, who builds on McLuhan, more accurately reappraised the effect of the electric age on intention (Postman, 1985). He explains that the information age has resulted not in an Orwellian dystopia where intentions are surveilled and constrained, but rather a Huxleyan one, where intentions are numbed: *“What Orwell feared were those who would ban books. What Huxley feared was that there would be no reason to ban a book, for there would be no one who wanted to read one. Orwell feared those who would deprive us of information. Huxley feared those who would give us so much that we would be reduced to passivity and egoism. Orwell feared that the truth would be concealed from us. Huxley feared the truth would be drowned in a sea of irrelevance. Orwell feared we would become a captive culture. Huxley feared we would become a trivial culture [...]”*. We inhabit not Foucault's society of discipline (Foucault, 1977; O'Neill, 1986), but Deleuze's society of control (Deleuze, 1992).

This scenario is undesirable, not least because mechanised convergence implies a reduction in the rate at which new ideas are generated, and an increase in repetition and replay of existing ideas. What kind of culture springs from the consumption and emission of an increasingly convergent set of increasingly recycled ideas? A derivative, “stuck” culture, is the diagnosis of technology critic Paul Skallas.⁷ Even for GenAI itself, the indications are that the roads of autophagy lead to madness; the roads of recursion lead to cursed collapse (Alemohammad et al., 2023; Shumailov et al., 2024; Bohacek & Farid, 2023; Gerstgrasser et al., 2024).

Mechanised convergence, as a tendency of automation more broadly, creates a crisis of intentionality: a culture that has lost the capacity to intend, does not realise, and does not care.

⁷<https://lindynewsletter.beehiiv.com/p/culture-stuck>, accessed July 2024. Related is the concept of “refinement culture”; *“Refinement culture can be summarized as a general streamlining and removal of any unique characteristics. Refinement Culture is one dimensional and removes variety from the environment. It's optimized.”* <https://lindynewsletter.beehiiv.com/p/refinement-culture>, accessed July 2024, <https://medium.com/@lindynewsletter/refinement-culture-51d96726c642>, accessed 2024.

3. Interlude: Babbage’s Intentional Programmer

Describing what GenAI does to intention as a “crisis” implies that we need to do something about it. Indeed, what we need to do about it is to promote the active cultivation of the capacity to intend.⁸

Since this is PPIG, we can start by considering the intentions of programmers. What the tendency for mechanised convergence tells us is that, prior to specifying behaviour, programming must be about forming an intention for behaviour. A definition of programming that centres intention, rather than specification, evokes a rather older philosophy of programming that we can draw from the crisis in theology at the time of Babbage.

Science (more precisely, natural philosophy) in post-Enlightenment Britain at the time of Babbage was grappling with the apparent contradiction of divine miracles – acts of God outside the laws of nature created by God – which Hume had famously argued could not be rationally supported (Hume, 1748). In aiming to discover mathematical laws such as those of Newton, which could accurately describe and predict nature, natural philosophers operating within the frameworks of Deism and Christianity struggled to reconcile their work and faith.

Babbage found in his Difference Engine the possibility for reinterpreting miracles as part of the natural divine order. Using a “feedback mechanism” that connected two gear wheels, Babbage was able to encode programs that, after a certain number of iterations, would change their behaviour. For example, he would demonstrate a program that counts the integers 1, 2, 3 ... up to 100, at which point the program would change and start counting in steps of two: 102, 104, 106 ... etc. In demonstration-sermons delivered to rapturous audiences, he used this example to explain his theory of God as a *divine programmer* (Snyder, 2012). A miracle was thus explained as a shift in the program. God’s intervention to perform apparent miracles was not an aberration against universal, constant laws – it was merely the manifestation of a deeper and misunderstood universal law, a deeper plan, a deeper intention.

It is instructive that Babbage’s conception of programming and intention centred around shifts, or deviations, from the expected. A machine that continues to execute the same predictable behaviour is not a program, it is simply a machine. It is in the departure from convergent behaviour that evidence of programming emerges as activity and divinity. For Babbage, to converge is human, to deviate divine. To execute is human, to program divine. To specify is human, to intend divine.

4. Sources of Intention: Constraints, Materiality, and Resistance

Returning to our objective – to promote the active cultivation of the capacity to intend – it is worth briefly exploring a few perspectives on the sources of intentions.

Much intention appears to arise as a result of interaction with the external world. Practitioners of creative arts and research in creativity have long noted the role of constraints in shaping and facilitating creativity (Stokes, 2005; May, 1975). Materiality and resistance are essential to craftsmanship; any material, by virtue of its properties and resistances, participates in an ongoing dialogue with the craftsman’s intentions (Basman, 2016). According to material engagement theory⁹ (Malafouris, 2019), “*Our forms of bodily extension and material engagement are not simply external markers of a distinctive human mental architecture. Rather, they actively and meaningfully participate in the process we call mind*”. As such, the role of material as a source of intention can be seen as a form of extended cognition, or at the very least external cognition (Turner, 2016), notwithstanding challenges to these ideas (Rupert, 2004).

A sculptor must consider how pliable or fragile their material is, what tolerances and fine details can be accomplished, how gravity will constrain the scale and orientation of their figures. A carpenter must consider the grain of their wood, where cuts and incisions can be made. A painter using watercolours must consider and exploit the additive translucency of that medium, one using oils must consider the opacity of theirs. It is telling that the archetypical dimension in the Cognitive Dimensions of Notations

⁸Much as R. Benjamin (2024) calls for us to cultivate the capacity to imagine.

⁹Thanks to Ava Scott for identifying this connection.

(Green, 1989) is *viscosity*, a metaphor rooted in materials and resistances, aiming to bridge them with the seemingly immaterial and disembodied world of notations.

Some intentions even rejoice in the contradiction of others: for example, the objective of subversive gameplay styles is to ignore the received goals of the game and invent one's own (Flanagan, 2009), it is playing the infinite game whose objective is to continue playing, not the finite game whose objective is to win (Carse, 1986). Solving the continuous puzzles posed by these resistances, having a vision pushed, pulled, and evolved, is the pleasure and intentionality of craftsmanship. These are not destructive resistances that hinder the realisation of an intention; they are productive ones that facilitate it.

Exploratory programming (Kery & Myers, 2017) exemplifies how the materialities and resistances of programming are exploited to shape intention. In exploratory programming, the programmer's goal is unknown or ill-defined. The objective of the process is to discover or create an intention, to formulate a problem. The formulation of a problem co-exists with and cannot be separated from its solution (Rittel & Webber, 1973; Sarkar, 2023c). This is also the case in the end-user programming activity of interactive machine learning, or interactive analytical modelling (Sarkar, 2016b), where the goal is ill-defined and the objective is to create one, through a constructivist loop of interaction between ideas and experiences (Sarkar, 2016a).

There have been proposals to design GenAI systems that introduce productive resistances as catalysts for the development of intention. Rather than an assistant, AI can act as a critic or provocateur (Sarkar, 2024; Sarkar et al., 2024). AI can be antagonistic (Cai et al., 2024). AI can cause cognitive glitches (Hollanek, 2019). AI can act as cognitive forcing functions (Buçinca et al., 2021). These proposals are counter to traditional narratives of system support, system disappearance, and system non-interference. They can be seen as successors to previous counternarratives raised by researchers such as critiques of the doctrines of simplicity and gradualism (Sarkar, 2023c), critiques of seamlessness (Chalmers & MacColl, 2003), critiques of reversible interactions (Rossmly et al., 2023), the case for design frictions and microboundaries (Cox et al., 2016), reframing of ambiguity as design resource (Gaver et al., 2003), and calls for attention checks in AI use (Gould et al., 2024).¹⁰

The concept of resistance could be key to framing the design objectives for intentional GenAI tools. Our current explorations of improving critical thinking with GenAI (e.g., Sarkar et al. (2024)) are strictly *additive*: let's augment AI interaction and output with prompts, text, visualisations, etc. that get the user thinking. However, this approach increases the cognitive burden by asking users to consume and reflect on more information. We know that people don't always enjoy, or want, more information. Particularly when it comes to the user interfaces of discretionary software, they usually want less (Carroll & Rosson, 1987; Sarkar, 2023c). The additive approach may be starting by fighting a losing battle, one in which we try to design the smallest, most stimulating, most rewarding "consumable" that creates user reflection, without incurring undesirable attentional costs. The idea of resistance provides a different starting point. How can we build GenAI tools with inherent, productive resistances that are part of working with the tool, not an additional thing that users need to "pay" attention to? How can the experience of resistances in the interface feel more like the pliability of clay, or the translucency of paint? This is an open avenue for future work.

¹⁰It is worth observing that while such counternarratives often involve calls for greater, more critical, and more reflective user engagement and participation with technology, it should not be assumed that intentionality always entails participation or action. Observation itself is not intrinsically passive. This point is well made by Pfaller (2017): "*Two philosophical premises silently played a decisive role in this triumphal march of participation: first, the idea that the relation between transmitter and receiver represents a hierarchy and that the elimination of this hierarchy therefore has to amount to a democratisation; and secondly, the idea that it is more desirable for spectators to participate than to spectate [... however;] the relation between transmitter and receiver does not always represent a hierarchy. And when it does, then it is not always in favour of the transmitter [...] This is why it is misleading to believe that activating the audience in art is automatically and always tantamount to their liberation. Because could not the exact opposite be the case: could the enthusiasm for joining in produced by art not deprive people of the necessary refractoriness that they would need in political life in order not to be immediately enthused by every neoliberal or reactionary or even fascist appeal to 'actively' join in, and pursue this with a feeling of liberation?*"

5. Existentialist Approaches to Intention

So far we have been considering intention at relatively small scale: instances of knowledge work and GenAI use. But intentions, like goals, form hierarchies. Intentions are not isolated and independent, they are related and convergent. To what do they converge? At this point we shall make a somewhat abrupt leap outwards and consider the most expansive scope of intention – as enacted over the course of an entire life.

An evolutionary account might attempt to trace human intentions back to fundamental physiological concerns: we form intentions to continue survival, to avoid fear, to ensure comfort, to maximise pleasure, to minimise pain. These can certainly account for some intentions. The concept of intention has much in common with free will – loosely defined, one’s capacity to act differently to how one did, in fact, act. Free will is not the same as intention, but it can be viewed as a precondition for true intention. Neuroscientific work purporting to demonstrate (a lack of) free will has been criticised by philosophers because (among other objections), we do not have a suitably good picture that connects short-term choices dominated by low-level psychological phenomena (such as choosing to push the left button or the right button) to the complex, long-term, highly planned and goal-oriented intentions (such as the intention to commit a crime) that pose the truly consequential ethical challenges to free will (Mele, 2019). The evolutionary account is part of a broader category of *teleosemantic* theories of intention (Jacob, 2023) according to which design (evolutionary or artificial) supplies a function (τέλος), which in turn supplies intention.

In considering whether human intention can truly be reduced to evolutionary or functional needs, I am drawn to the argument made by feminist anthropologist Payal Arora in her closing keynote for the 2022 CHI conference (Arora, 2022). She criticizes Maslow’s famous hierarchy of needs that places physiological and safety needs at the bottom, rising to esteem and self-actualisation at the top. The conventional reading is that needs at the bottom of the pyramid need to be satisfied, the foundation of the pyramid needs to be built, before one can proceed to the higher levels. This is a fairly influential way of thinking and often dictates the way in which social aid and rescue efforts are prioritised: focus on food, water, and shelter first, and joy, play, growth, education, and dignity later. Arora finds that this picture does not correspond with her observations in her extensive ethnographic work with precarious, oppressed, and underprivileged groups. Instead, she proposes that the pyramid is upside down. What she finds is that self-actualisation is what people need first, and are willing to sacrifice safety needs to get it. People leave secure work when the nature of that work threatens their dignity, even if this places them in financial hardship. People leave homes where they cannot express their identity, or are not accepted for who they are, even if this might leave them without a roof over their head. A line from the poet James Oppenheim captures the sentiment:

*“Our days shall not be sweated from birth until life closes —
Hearts starve as well as bodies: Give us Bread, but give us Roses.”*

If not entirely upside down, then at the very least Maslow’s hierarchy is not a unidirectional ladder to climb, but a set of considerations and influences that are continually negotiated and traded-off. Physiology and evolution are part of intention formation, but far from the entire picture. Where can we look for a perspective on intention that aligns with Arora’s observations? Moreover, is there an approach that not only identifies the source of intention, but prescribes a method for cultivating it?

Elaborating the consequences of the idea that the active cultivation of intention is *the* core virtue in an inherently meaningless world is precisely the project of existentialist philosophy.

The absence of any inherent purpose to life is the starting point. Per Sartre (1943), “existence precedes essence”; individuals first exist without purpose and must subsequently forge their essence, or identity, through their actions. Angst, or existential anxiety, arises from the realization of one’s freedom and the infinite possibilities it entails (Kierkegaard, 1844). Existentialists see angst as a motivator rather than an obstacle.

Authenticity is one expression of existentialist intention. It is the pursuit of living in accordance with one's true self and values, rather than conforming to societal norms, and is essential for genuine existence (Heidegger, 1927). Authenticity requires a conscious effort to understand and act upon personal convictions, even in the face of adversity or societal pressure (Kierkegaard, 1843; de Beauvoir, 1948). Other sources of intentionality, besides authenticity, go beyond the individual. Kierkegaard's (Kierkegaard, 1849) "leap of faith" suggests that to escape from existential despair requires acknowledging the limits of rational reflection and an individual's relationship with the divine. Moreover, to seek engagement with the world is to step beyond oneself, to interact with others, and to find and create meaning through these actions (Jaspers & Saner, 1932). Similarly, de Beauvoir (1948) points out that our individual subject-like freedom is complemented by an object-like unfreedom ("facticity"), deriving an ethics of freedom that advocates for actions that respect the freedom of others.

Camus (1942) counsels individuals to accept "the absurd": the tension between the human search for meaning and a universe that is silent in response, to recognize the lack of inherent meaning in the world and to take on the task of creating their own purpose. Camus rejects "solutions" to the absurd proposed by prior philosophers, such as Kierkegaard, as "philosophical suicide". To Camus, seeking overarching meaning despite the absurd is seeking to resolve, minimise, sidestep, or ignore the absurd, not acknowledging it.

Camus rejects a forced imposition of meaning where there is none. A leap of faith is a form of escape. Incidentally, a forced imposition of meaning is precisely the *modus operandi* of GenAI: for language to be produced by arithmetic means it is necessary to encode language in a uniform, rational vector space. Sense and nonsense alike are thus enumerated and made commensurable. *King+Man+Woman=Queen* (Mikolov et al., 2013). Before carefully designed guardrails (themselves a form of escape) made it more difficult to do so, it was easy to elicit answers to nonsense questions such as "what colourless green ideas sleep furiously?" from language models. Furthermore, GenAI is an essential component of an emerging pseudoreligious meta-narrative of escape identified by Gebru & Torres (2024): "*What ideologies are driving the race to attempt to build AGI? [...] we trace this goal back to the Anglo-American eugenics movement, via transhumanism. [...] we delineate a genealogy of interconnected and overlapping ideologies that we dub the 'TESCREAL bundle,' where the acronym 'TESCREAL' denotes 'transhumanism, Extropianism, singularitarianism, (modern) cosmism, Rationalism, Effective Altruism, and longtermism'*".

Camus' existentialist view offers a non-escapist alternative that stares meaninglessness in the face and from it derives freedom. This freedom is both liberating and burdensome. We are at liberty to choose, but are also responsible for bearing the burden of the consequences. The lightness of being can thus be unbearable. It is through confronting this anxiety that individuals can make deliberate and meaningful choices, shaping their intentions, and by extension, their essence.

GenAI has implications for the intention of professional programmers and casual ones alike. The introduction poses the question "what is programming?", and we can now see a second reading of this question which asks not for a definition of an activity, but of an aspiration or identity. As GenAI solves the problem of control, of specifying behaviour, the aspiration shifts to intent. Intent precedes control. To be a programmer is therefore not to be one who specifies behaviour, but one who forms authentic, meaningful intentions for behaviour.

6. Speculative Scenarios for Intentional Programming

The optimism of the "intention is all you need" narrative does posit a legitimate observation concerning the behavioural economics of software production. GenAI makes the production of bespoke software vastly cheaper. One can view existentialism as a response to the loss of the "grand narratives" of modernity. But software has still been constrained by the grand narratives of capitalism and utility – until now. To write a program required *investment* of time and hard-earned expertise, exerting pressure on programs to be valuable, robust, and reusable. Where they did not place an outright barrier, the investment costs of programming disincentivised exploration, error, and disposal. Within this frame story hitherto sits

the universe of programmer psychology and behaviours: from authoring code to code comprehension, from knowledge sharing and documentation to debugging, from learning barriers to attention investment, from API design to autocomplete. Almost the entire diversity of experience of programmers, professional or casual, that our research community has so carefully documented and explained for the last half-century, has dwelt in the shadow of the market's invisible hand.

As the hand is withdrawn, one might ask how programmers can respond, in a microcosm of the existential dilemma, to the liberating yet burdensome freedom granted by GenAI. As far as practical advice (i.e., “implications for design[ing your life]”) is concerned, existentialists advise embracing one's freedom to shape life, living authentically, accepting the absurd, confronting anxiety, and seeking engagement with the world as ways to form meaningful intentions. What this might mean for programmers, and interaction with GenAI, can be sketched in a few speculative scenarios:

- **Intentional coding retreats:** The programmer steps away from her standard way of working to participate in an intentional coding retreat. Here, the programmer reconnects with the craft of coding without the assistance of AI tools. This allows the programmer to explore and reaffirm personal coding styles and problem-solving approaches. For example, a programmer accustomed to relying on AI for debugging might rediscover the satisfaction of manually untangling complex code, thus reaffirming their individual capability and creative freedom.
- **AI as muse:** AI suggests an unusual, contradictory, or incorrect algorithmic approach, which the programmer then refines and transforms with personal insights and expertise. The tool is not a crutch but a source of inspiration.
- **Programming with provocations:** programming environments include prompts or questions to stimulate deeper thinking about the purpose and potential impact of the code being written. This can help programmers reconnect with their motivations and aspirations.
- **Programming with constraints:** intentional constraints are introduced to programming projects, much like the practices of constrained writing.¹¹ Programmers already practice genres of constrained programming for pleasure, such as “code golf” (writing the shortest possible program with a certain behaviour) or “quines” (inputless programs that produce only their own source code as output). By deliberately limiting certain resources or imposing unique challenges, programmers can stimulate creativity and craft intentional solutions.
- **Deviation practice:** in the education of professional programmers, exercises are developed that require intentional deviation from established patterns. By practising the precise skill of breaking away from standard solutions, programmers may more readily acquire the conscious muscle and desire for forming unique intentions and exploring novel paths.
- **Intentionality metrics:** tools display metrics that evaluate the degree of human intention in the creative process (noting that these metrics are necessarily reductionist proxies and may become subject to Goodhart's/Campbell's law). For example, a generative design tool might analyse the uniqueness of user queries and the divergence of the output from standard templates. Visibilising the invisible effects of mechanised convergence may encourage users to engage more deeply with the work and make conscious, deliberate, individual choices.
- **Participatory AI artefacts:** artefacts are intentionally left incomplete by AI, requiring human participation¹² to finalise. For instance, a participatory tool generates the outline of a web design but leaves decisions about colour schemes and typography to the user. Conversely, a tool refuses to generate an outline, requiring the user to form a rough intention independently, before assisting by filling in details.

¹¹E.g., see discussion of conceptual writing in [Sarkar \(2023b\)](#).

¹²though not “collaboration” ([Sarkar, 2023a](#))

These speculations are not meant to be concrete proposals, but rather simply representative ideas of a future where the existentialist values of freedom, authenticity, and intentionality are preserved and enhanced through GenAI. They are limited in vision, representing only the lines of sight from where we stand today, and unable to anticipate the adjacent possibles of where we might travel.

7. Conclusion

Programming is undeniably changing under the influence of GenAI. Intention appears to be all one needs to create software. But the notion that GenAI offers a neutral, unencumbered path to realising intentions is a mirage. Contrary to the assumption that GenAI merely executes human intentions, it also shapes them. At the very least, GenAI can induce “mechanised convergence”, homogenising creative output, and reducing diversity in thought. There is therefore a risk of creating a “stuck” culture that recycles an old set of convergent ideas instead of fostering a new set of divergent ones.

In seeking a way through this problem we have encountered a variety of sources that we can draw upon to precipitate the active cultivation of intention: evolutionary pressures, the need for dignity and self-actualisation, constraints, subversion, materiality, and resistance. Finally, we discussed how the problem of intention resonates with the existentialist pursuits of freedom, identity, and authenticity. While this discussion of existentialism is necessarily cursory, limited, flawed, and provisional, its aim has been to situate the problems posed by GenAI to intentionality in the broadest possible scope.¹³

Programming must go beyond specification and embody the active cultivation of intentions. Existentialist philosophy offers a proactive, prescriptive framework for understanding the formation of human intentions as a process that ought to be held as deeply personal, ethically charged, and fundamentally free. It teaches us that to be human is to be involved in a continuous project of becoming. After all – one is not born, but rather becomes, a programmer.

8. Acknowledgements

Thanks to Sean Rintel and Lev Tankelevitch for helping review drafts of this paper. I am especially grateful to Ava Scott and Richard Banks for their generous and helpful reflections.

References

- Alemohammad, S., Casco-Rodriguez, J., Luzi, L., Humayun, A. I., Babaei, H., LeJeune, D., . . . Baraniuk, R. G. (2023). *Self-Consuming Generative Models Go MAD*. Retrieved from <https://arxiv.org/abs/2307.01850>
- Anderson, B. R., Shah, J. H., & Kreminski, M. (2024). Homogenization effects of large language models on human creative ideation. *arXiv preprint arXiv:2402.01536*.
- Arnold, K. C., Chauncey, K., & Gajos, K. Z. (2020). Predictive text encourages predictable writing. In *Proceedings of the 25th International Conference on Intelligent User Interfaces* (p. 128–138). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3377325.3377523> doi: 10.1145/3377325.3377523
- Arora, P. (2022). FemWork: Critical Pivot towards Design for Inclusive Labor Futures. In *2022 CHI Conference on Human Factors in Computing Systems (Closing Keynote)*.
- Basman, A. (2016). Building software is not a craft. *Proceedings of the Psychology of Programming Interest Group, 142*.
- Benjamin, R. (2024). *Imagination: A manifesto (a norton short)*. WW Norton & Company.
- Benjamin, W. (1935). The work of art in the age of mechanical reproduction, 1936. *New York*.
- Blackwell, A. F. (2002). What is programming? In *PPIG* (Vol. 14, pp. 204–218).

¹³Camus (1942) describes existence (suicide) as the only truly serious philosophical problem.

- Blackwell, A. F. (2020). Objective functions:(in) humanity and inequity in artificial intelligence. *Science in the ForeSt, Science in the PaSt*, 191.
- Bohacek, M., & Farid, H. (2023). *Nepotistically trained generative-ai models collapse*. Retrieved from <https://arxiv.org/abs/2311.12202>
- Buçınca, Z., Malaya, M. B., & Gajos, K. Z. (2021, apr). To Trust or to Think: Cognitive Forcing Functions Can Reduce Overreliance on AI in AI-assisted Decision-making. *Proc. ACM Hum.-Comput. Interact.*, 5(CSCW1). Retrieved from <https://doi.org/10.1145/3449287> doi: 10.1145/3449287
- Cai, A., Arawjo, I., & Glassman, E. L. (2024). Antagonistic AI. *arXiv preprint arXiv:2402.07350*.
- Camus, A. (1942). *The myth of sisyphus* (J. O'Brien, Trans.). France: Éditions Gallimard (in French), Hamish Hamilton (in English).
- Carroll, J. M., & Rosson, M. B. (1987). Paradox of the active user. In *Interfacing thought: Cognitive aspects of human-computer interaction* (pp. 80–111).
- Carse, J. P. (1986). *Finite and infinite games*. New York, NY: Free Press.
- Chalmers, M., & MacColl, I. (2003). Seamless and seamful design in ubiquitous computing. In *Workshop at the crossroads: The interaction of HCI and systems issues in UbiComp* (Vol. 8).
- Cox, A. L., Gould, S. J., Cecchinato, M. E., Iacovides, I., & Renfree, I. (2016). Design frictions for mindful interactions: The case for microboundaries. In *Proceedings of the 2016 CHI conference extended abstracts on human factors in computing systems* (pp. 1389–1397).
- de Beauvoir, S. (1948). *The ethics of ambiguity* (B. Frechtman, Trans.). Citadel Press Publishing, A Subsidiary of Lyle Stuart Inc.
- Deleuze, G. (1992). Postscript on the societies of control. *October*, 59, 3–7. Retrieved 2024-06-04, from <http://www.jstor.org/stable/778828>
- Dell'Acqua, F., McFowland, E., Mollick, E. R., Lifshitz-Assaf, H., Kellogg, K., Rajendran, S., ... Lakhani, K. R. (2023). Navigating the jagged technological frontier: Field experimental evidence of the effects of ai on knowledge worker productivity and quality. *Harvard Business School Technology & Operations Mgt. Unit Working Paper*(24-013).
- Dennett, D. C. (1971). Intentional systems. *The journal of philosophy*, 68(4), 87–106.
- Doshi, A. R., & Hauser, O. (2023, Aug). Generative artificial intelligence enhances creativity but reduces the diversity of novel content. (Available at SSRN: <https://ssrn.com/abstract=4535536> or <http://dx.doi.org/10.2139/ssrn.4535536>)
- Flanagan, M. (2009). *Critical play*. London, England: MIT Press.
- Foucault, M. (1977). *Discipline and punish*. New York, NY: Pantheon Books.
- Gaver, W. W., Beaver, J., & Benford, S. (2003). Ambiguity as a resource for design. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 233–240).
- Gebru, T., & Torres, É. P. (2024). The TESCREAL bundle: Eugenics and the promise of utopia through artificial general intelligence. *First Monday*.
- Gerstgrasser, M., Schaeffer, R., Dey, A., Rafailov, R., Sleight, H., Hughes, J., ... Koyejo, S. (2024). *Is model collapse inevitable? breaking the curse of recursion by accumulating real and synthetic data*. Retrieved from <https://arxiv.org/abs/2404.01413>

- Gould, S. J. J., Brumby, D. P., & Cox, A. L. (2024). Chatldr – you really ought to check what the llm said on your behalf. In *Extended abstracts of the 2024 chi conference on human factors in computing systems*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3613905.3644062> doi: 10.1145/3613905.3644062
- Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, 443–460.
- Heidegger, M. (1927). *Being and time* (J. Macquarrie & E. Robinson, Trans.). SCM Press.
- Hollanek, T. (2019). Non-user-friendly: Staging resistance with interpassive user experience design. *A Peer-Reviewed Journal About*, 8(1), 184–193.
- Hume, D. (1748). *An enquiry concerning human understanding*.
- Jacob, P. (2023). Intentionality. In E. N. Zalta & U. Nodelman (Eds.), *The Stanford encyclopedia of philosophy* (Spring 2023 ed.). Metaphysics Research Lab, Stanford University. <https://plato.stanford.edu/archives/spr2023/entries/intentionality/>.
- Jakesch, M., Bhat, A., Buschek, D., Zalmanson, L., & Naaman, M. (2023). Co-Writing with Opinionated Language Models Affects Users' Views. 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/10.1145/3544548.3581196> doi: 10.1145/3544548.3581196
- Jaspers, K., & Saner, H. (1932). *Philosophie* (Vol. 1). J. Springer Berlin.
- Kery, M. B., & Myers, B. A. (2017). Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 25–29).
- Kierkegaard, S. (1843). *Fear and trembling*. Denmark: First authorship (Pseudonymous). (Published in English in 1919 – first translation)
- Kierkegaard, S. (1844). *The concept of anxiety* (R. Thomte, Trans.). Denmark. (Published in English in 1946)
- Kierkegaard, S. (1849). *The sickness unto death*.
- Lee, M., Blackwell, A., & Sarkar, A. (2024). Predictability of Identifier Naming with Copilot: A Case Study for Mixed-Initiative Programming Tools. *Proceedings of the 35th Annual Conference of the Psychology of Programming Interest Group (PPIG 2024)*.
- 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*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3544548.3580817> doi: 10.1145/3544548.3580817
- Malafouris, L. (2019). Mind and material engagement. *Phenomenology and the cognitive sciences*, 18(1), 1–17.
- May, R. (1975). *The courage to create*. New York, NY: WW Norton.
- McLuhan, M. (1964). *Understanding media: The extensions of man*. McGraw-Hill. (First edition)
- Mele, A. (2019). Free will and neuroscience: decision times and the point of no return. In *Free will, causality, and neuroscience* (pp. 83–96). Brill.

- Mikolov, T., Yih, W.-t., & Zweig, G. (2013). Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies* (pp. 746–751).
- O’Neill, J. (1986). The disciplinary society: from weber to foucault. *British Journal of Sociology*, 42–60.
- Pfaller, R. (2017). *Interpassivity: The aesthetics of delegated enjoyment*. Edinburgh University Press.
- Postman, N. (1985). *Amusing ourselves to death*. Viking Books.
- Prather, J., Reeves, B. N., Denny, P., Becker, B. A., Leinonen, J., Luxton-Reilly, A., . . . Santos, E. A. (2023, nov). “it’s weird that it knows what i want”: Usability and interactions with copilot for novice programmers. *ACM Trans. Comput.-Hum. Interact.*, 31(1). Retrieved from <https://doi.org/10.1145/3617367> doi: 10.1145/3617367
- Rittel, H. W., & Webber, M. M. (1973). Dilemmas in a general theory of planning. *Policy sciences*, 4(2), 155–169.
- Robinson, D., Cabrera, C., Gordon, A. D., Lawrence, N. D., & Mennen, L. (2024). Requirements are all you need: The final frontier for end-user software engineering. *arXiv preprint arXiv:2405.13708*.
- Rosen, B. M. (1979). *VISICALC: Breaking the Personal Computer Bottleneck*. <http://bricklin.com/history/rosenletter.htm>. (Accessed 06-08-2024)
- Rossmly, B., Terzimehić, N., Döring, T., Buschek, D., & Wiethoff, A. (2023). Point of no undo: Irreversible interactions as a design strategy. In *Proceedings of the 2023 chi conference on human factors in computing systems* (pp. 1–18).
- Rupert, R. D. (2004). Challenges to the hypothesis of extended cognition. *The Journal of philosophy*, 101(8), 389–428.
- Sarkar, A. (2016a). Constructivist Design for Interactive Machine Learning. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems* (p. 1467–1475). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2851581.2892547> doi: 10.1145/2851581.2892547
- Sarkar, A. (2016b). *Interactive analytical modelling* (Tech. Rep. No. UCAM-CL-TR-920). University of Cambridge, Computer Laboratory. Retrieved from <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-920.pdf> doi: 10.48456/tr-920
- Sarkar, A. (2023a). Enough With “Human-AI Collaboration”. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3544549.3582735> doi: 10.1145/3544549.3582735
- Sarkar, A. (2023b). Exploring Perspectives on the Impact of Artificial Intelligence on the Creativity of Knowledge Work: Beyond Mechanised Plagiarism and Stochastic Parrots. In *Proceedings of the 2nd Annual Meeting of the Symposium on Human-Computer Interaction for Work*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3596671.3597650> doi: 10.1145/3596671.3597650
- Sarkar, A. (2023c). Should Computers Be Easy To Use? Questioning the Doctrine of Simplicity in User Interface Design. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3544549.3582741> doi: 10.1145/3544549.3582741

- Sarkar, A. (2023d). Will Code Remain a Relevant User Interface for End-User Programming with Generative AI Models? In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (p. 153–167). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3622758.3622882> doi: 10.1145/3622758.3622882
- Sarkar, A. (2024, sep). AI Should Challenge, Not Obey. *Communications of the ACM*. Retrieved from <https://doi.org/10.1145/3649404> (Online First) doi: 10.1145/3649404
- 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)*.
- Sarkar, A., Xu, X. T., Toronto, N., Drosos, I., & Poelitz, C. (2024). When Copilot Becomes Autopilot: Generative AI's Critical Risk to Knowledge Work and a Critical Solution. In *EuSprIG Proceedings*.
- Sartre, J.-P. (1943). *Being and nothingness* (H. E. B. (1st English translation) & S. R. (2nd English translation), Trans.). France: Éditions Gallimard, Philosophical Library. (Published in English in 1956)
- Sharma, N., Liao, Q. V., & Xiao, Z. (2024). Generative Echo Chamber? Effect of LLM-Powered Search Systems on Diverse Information Seeking. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3613904.3642459> doi: 10.1145/3613904.3642459
- Shumailov, I., Shumaylov, Z., Zhao, Y., Papernot, N., Anderson, R., & Gal, Y. (2024). Ai models collapse when trained on recursively generated data. *Nature*, *631*(8022), 755–759.
- Snyder, L. (2012). *The philosophical breakfast club*. New York, NY: Broadway Books.
- Stokes, P. D. (2005). *Creativity from constraints: The psychology of breakthrough*. Springer Publishing Company.
- Tankelevitch, L., Kewenig, V., Simkute, A., Scott, A. E., Sarkar, A., Sellen, A., & Rintel, S. (2024). The Metacognitive Demands and Opportunities of Generative AI. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3613904.3642902> doi: 10.1145/3613904.3642902
- Turner, P. (2016). Distributed, external and extended cognition. *HCI Redux: The Promise of Post-Cognitive Interaction*, 75–98.
- Welsh, M. (2022, dec). The end of programming. *Commun. ACM*, *66*(1), 34–35. Retrieved from <https://doi.org/10.1145/3570220> doi: 10.1145/3570220
- Winner, L. (1980). Do artifacts have politics? *Daedalus*, 121–136.
- Zamfirescu-Pereira, J., Wong, R. Y., Hartmann, B., & Yang, Q. (2023). Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. 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/10.1145/3544548.3581388> doi: 10.1145/3544548.3581388

How Do Developers Approach Their First Bug in an Unfamiliar Code Base? An Exploratory Study of Large Program Comprehension

Andreas Bexell

Ericsson | Lund University
andreas.bexell@ericsson.com

Emma Söderberg

Lund University
emma.soderberg@cs.lth.se

Christofer Rydenfält

Lund University
christofer.rydenfalt@design.lth.se

Sigrid Eldh

Ericsson | Mälardalen Uni. | Carleton Uni.
sigrid.eldh@ericsson.com

Abstract

Program comprehension is a significant part of developing software; studies suggest that developers spend 50-70% of their time comprehending program code. Program comprehension and code comprehension have been the topics of numerous research studies, but the vast majority of these studies focus on the comprehensibility of statements or functions, and give little guidance for how to support comprehension of the large programs common in the industry.

In this paper, we present an exploratory study focusing on practitioners' comprehension of large programs in the context of approaching their first bug in an unfamiliar code base. We carried out a study where we interviewed five professional programmers with experience in software development of large programs. The interviews were focused on the subjects' attitudes, their strategies, frustrations they experienced and any opportunities in this area. We found that our participants employ several different strategies, including for example reproduction, localization and simulation, when approaching an unfamiliar code base. We see a potential relationship between these strategies and factors such as professional experience. Our results indicate that large program comprehension may be a fruitful area for further study, and we outline and discuss some of these opportunities.

1. Introduction

Program comprehension is an important part of the work of a programmer. Studies based on instrumentation of tools used by practitioners report time spent on program comprehension in the range of approximately 58% (Xia et al., 2018) to 70% (Minelli, Mocci, & Lanza, 2015). Program comprehension takes substantial time while developing software. Improved assistance in this area has the potential to create a significant impact for programmers.

Understanding large programs poses additional challenges to understanding medium-sized programs or individual statements of code. It is reasonable for the goals of many program comprehension studies to address code snippets, which have the size of approximately 10-100 lines of source code (LoC) and can be viewed as self-contained programs. However, there is little guidance to be found when seeking to understand more about the comprehension of larger programs typically found in the industry. How do programmers orient themselves in such large systems? For instance, Von Mayrhauser et al. (Von Mayrhauser, Vans, & Howe, 1997) describe comprehension of sufficiently large programs as “*understanding will, of necessity, be partial*”. They further question whether code comprehension studies of novices working on a general understanding of small programs apply to “*professional maintenance tasks [...] on large scale software*”. We believe that it is reasonable that comprehension of large programs may differ from comprehension of code snippets, and that this is an area worthy of more study.

In this study, we specifically address comprehension of large programs, where the size of the program could for example be the Linux kernel that consists of 30 MLoC. Such a large program has complex dependencies and relations between components, which is necessary and a prerequisite to comprehend before many programming tasks can be carried out. Programmers often need to understand large pro-

grams in unfamiliar code bases numerous times during their careers, e.g., when joining a new team or integrating with an unfamiliar system. We address the research question (RQ) *How do professional programmers approach their first bug in an unfamiliar large program?* We seek an answer to this question via semi-structured interviews with five practitioners, where we focus on attitudes towards large program comprehension, strategies, frustrations, and opportunities for improvement.

Our contribution is that we find several different strategies employed by these practitioners when starting to comprehend a large program in an unfamiliar code base. We also find indications that strategies change and evolve with experience. Furthermore, we find that there may be an untapped potential for improved tools support for large program comprehension.

The rest of this paper is structured as follows: we start with a description of the method in Section 3, then the results in Section 4, before we cover threats in Section 5, discuss our findings in relation to related work in Section 6, and conclude in Section 7.

2. Related work

Approaching one's first bug in an unfamiliar large program is an activity that requires acquiring a certain level of comprehension of the program and its structure, as well as being able to locate the bug in it. In this section, we present an overview of related work in the areas of program comprehension and code comprehension, with a brief comparison between these closely related topics. Next, we present related work on bug (fault) localization.

2.1. Program Comprehension & Code Comprehension

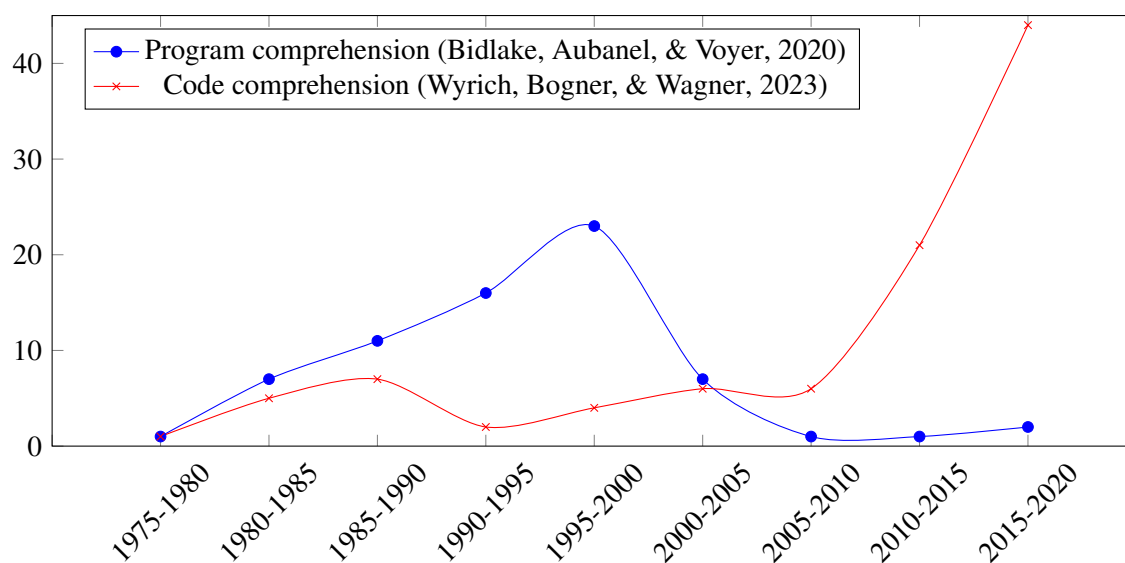


Figure 1 – A graph of publication dates of included papers in the meta-studies of “mental representations of programs” and “code comprehension” indicate a turn in research attention from the first towards the latter.

Studies on program comprehension tend to focus on the programmers’ mental models and understanding of an entire program (Bidlake et al., 2020), while code comprehension studies tend to use code snippets selected to fit the needs of the study, for instance, to strike a balance between simplicity and complexity (Wyrich et al., 2023). Recent mapping studies of program and code comprehension indicate a turn in the field of software comprehension from mental representations of programs, reported by Bidlake et al. (Bidlake et al., 2020), to code comprehension, reported by Wyrich et al. (Wyrich et al., 2023); see Figure 1. This study falls in the domain of program comprehension, rather than in code comprehension, with a special focus on large programs.

Siegmund (Siegmund, 2016) summarizes program comprehension to these points; (1) getting an

overview of a large program or software architecture, (2) understanding type structures and call hierarchies, (3) understanding the relationship between components, and (4) identifying the developers who are responsible for a component. Our study focuses on the first and third items.

Siegmund describes the top-down comprehension model, bottom-up comprehension model, and the combination of the two in the integrated comprehension model. She notes that if programmers are encountering areas they are not familiar with, they resort to executing the program sequence by sequence. However, she does not explore how programmers create their own orientation in the large program, and how they explore unfamiliar areas in the program to solve a particular task.

Kulkarni provides a case description by narrating their experience of starting work with a new large code base. (Kulkarni, 2016). They reflect on their use of bottom-up and top-down comprehension, and on their conscious use of Brook's, Soloway's and Shneiderman's models (Brooks, 1978; Soloway & Ehrlich, 1984; Shneiderman & Mayer, 1979) Kulkarni concludes that no one strategy is all-embracing, but that several approaches are needed.

Wuilmart et al. (Wuilmart, Söderberg, & Höst, 2023) interview four professionally active programmers and indicate that to gain sufficient comprehension of a large program, programmers may need to consult with other programmers and sources outside of the source code. They further found that the interviewed programmers would typically approach a new code base top-down followed by experimentation with different ways of running the program.

2.2. Bug localization and Fault localization

When beginning work with a new code base, a common problem is to find a good starting place. (Tempero & Ralph, 2018) discusses the “where to start” question. They note that most “program comprehension” research is done on an “implementation” level – the understanding of statements. This is contrasted with “design comprehension”, as in understanding the internal relations of a program.

Locating a bug in a large program requires specific strategies. This has been studied by Katz and Andersson (Katz & Anderson, 1987), by Vessey (Vessey, 1985), and Decasse and Emde (Decasse & Emde, 1988), who each enumerate a set of strategies employed by programmers when locating bugs in software programs. Later, Romero et al. (Romero, Du Boulay, Cox, Lutz, & Bryant, 2007) has coded strategies from observations of programmers' behaviour when debugging software.

In the 1990s, research on bug localization took a turn for the study of automatic bug localization techniques see (Wong, Gao, Li, Abreu, & Wotawa, 2016) and (Wang, Galster, & Morales-Trujillo, 2023), and new theories on strategies for human debuggers after that are scarce. The gap between recent graduates' experience with large code bases compared to the industry's expectations is addressed in (Shah, Yu, Tong, & Raj, 2024). The fact that there is an expectation from the industry may indicate that experienced software developers engage with unfamiliar large code in a manner different from students. We study professionally active developers working with large programs.

Collecting data from 102 professional developers, Hirsch and Hofer suggest that localizing bugs in programs is more time-consuming than fixing them. They conclude that more research may be needed specifically on bug localization. (Hirsch & Hofer, 2021)

Alaboudi and LaToza perform an analysis of the activities developers engage in during video-recorded debugging sessions (Alaboudi & LaToza, 2023). This results in a comprehensible enumeration of what observable activities programmers engage in (*Edit, Navigate, Test*). This study collects retold experience, and may additionally capture some of the strategies that may result in the activities enumerated by Alaboudi and LaToza.

3. Method

To address our research question, we designed an interview study focused on the following topics in relation to large program comprehension; the *attitudes* with which programmers approach an unfamiliar code base, what *strategies* they employ to orient themselves to solve their first bug there, what *frustra-*

Table 1 – The mapping of topics and questions in the interview protocol. See also Appendix A.

Topic	Question(s)
Attitudes	"[...] how do you feel?"
Strategies	"What do you do [...]?" "What strategies do you employ [...]?" "What tools do you employ?"
Frustrations	"What trouble do you get into?"
Opportunities	"What would you want [...]?"

Table 2 – Overview of interview participants. Exp - years of professional programming experience, CB - number of new code bases the participant has been introduced to, Lang - number of programming languages the participant has professional capacity in.

Participant	Age	Gender	Exp	CB	Lang
S1	40	non-binary	27	15	15
S2	47	male	20	15	5
S3	43	male	18	30	5
S4	26	female	2	2	6
S5	37	male	22	20	5
Avg/Med	38/43		17/22	16/15	7/5

tions they encounter, and what *opportunities* for improvement they identify.

3.1. Data Collection

We used *semi-structured interviews* (Robson, 2011) as a means to collect the data needed to address our research question. The interview protocol was designed to enable *directed content analysis* (Hsieh & Shannon, 2005) to collect information about the participants' professional experience and their experience of large program comprehension with a focus on attitudes, strategies, frustrations, and opportunities (Table 1). The interview protocol is included in Appendix A.

We recruited five participants, described in Table 2. The participants were all programmers with 2-27 years of professional software development experience from more than one company. All five have started working with new large programs at least twice during their respective careers. All of them are Swedish and have worked in large software development companies in Sweden.

Before the interviews, the participants were informed of the topic of the study and the members of the research group. Participants were promised that any publications would be in a manner so that the identity of the participants would be protected. The raw data is available only to external auditors. The participants were informed of their right to opt-out of the study at any time, without penalty. Each of the participants signed a consent form to this effect.

The interviews were executed in Swedish. They were conducted via remote link with audio and video. The interviews were recorded (except the first interview, when the recording failed). In addition, physical notes were taken during the interviews.

3.2. Data Analysis

Data extraction was carried out by the first author. The recordings were automatically transcribed using Microsoft Teams auto-transcription. The transcriptions were used as a guide to listen to the recordings to extract quotes according to directed content analysis, related to the topics in Table 1). Initially, at least one quote per participant and topic was extracted. During this process, the quotes were translated from Swedish to English.

The extracted quotes were reviewed and discussed together with the second author. Especially the

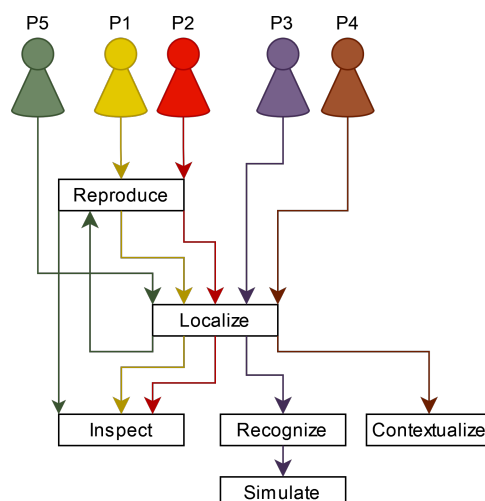


Figure 2 – The activities mentioned by the participants, with order corresponding to the order in which each activity was mentioned in the interview.

quotes connected to strategies were reviewed in more detail as it became clear that participants undertake several different activities. The interviews were revisited by the first author to code the activities using *conventional content analysis* (Hsieh & Shannon, 2005). The coded activities of bug localization in an unfamiliar large program were mapped to activities described in the literature (see Table 3).

4. Results

Here, we present the results of our interview study. We focus the analysis of the interview material on the four topics presented in Section 3; attitudes, strategies, frustrations, and opportunities.

4.1. Attitudes

We found that our participants approach an unfamiliar codebase with a mix of **anticipation and caution**. Several participants described feelings of vulnerability, like a feeling of being overwhelmed (“*It’s a mix of fun and overwhelming*”, S1) or nervous (“*Until I’ve checked out [the code] I’m quite calm. [...] That is something I have learned with time because you are very, very nervous. What am I getting into?*”, S5). One participant mentioned that a mentor can help to relieve the **feeling of being lost** (“*When you have a mentor, it’s easier [...] Otherwise, just ‘here’s the code base, here’s a computer, find out what’s happening’ that’s much more frightening, because I’m, like, what file should I even open first?*”, S4).

We also noticed that there appears to be a risk of friction in terms of **expectations** (e.g., “*many places have these - ‘we’ve always done like this’ [...] can drive you crazy in the beginning*”, S1). To spare some frustration, participants recount strategies to contain expectations, for instance, to refrain from a personal investment to reduce the **feeling of ownership** (“*Maybe a good idea to cool it in the beginning, to wait and see - there may be a good reason*”, S1). The structure of the work may already create some of this distance. For instance, one participant described the role you have as a consultant where you are more a visitor in the code base which may create a different relation to code ownership (“*As a contractor, you have the benefit of not being a part of company politics, but you can be objective and try to solve the problem. The client’s problem, that is.*”, S3).

4.2. Strategies

Our participants mention strategies composed of a series of activities, illustrated in Figure 2 with the order they were mentioned by participants. Some of the activities are shared between participants, while others are mentioned to a lesser extent and some are mentioned only by a single participant.

Reproduce: To “reproduce” a bug means to be able to provoke and observe a bug, ideally in a predictable manner. This is a common entry point (“*It’s important to know exactly what the bug is [...] So that’s the first thing, to find out exactly what the bug results in and see if it can be reproduced.*” S5). Reproducing

a bug can either be by running a **test case** that fails or by following a **specific procedure** (“*We need to start by reproducing this bug I’m meant to solve [...] best case, there is a failing test case ... or maybe I need to use the product according to some instructions to make it exhibit this problem.*” **S1**, “*I mean, it’s a bug, it’s reproducible. Yes: you press a button or play a video or something [...]. And you see: this happens, that shouldn’t. [...] Then I can dig from there.*” **S2**). How to reproduce a bug is often a core part of a bug report, and several subjects state they start their understanding of the bug by reproducing it. This might also provide an opportunity to introduce oneself to the interaction with the program.

Localize: When finding a bug in a large and unfamiliar code base, it is important to try to shrink the area of investigation to a manageable size. “Localization” is the act of trying to limit the area of interest and demarcate a part of the code that is relevant to a specific task (“*try to come down from the problem is located somewhere in these hundred thousand lines to [...] ... 1000. A workable number*”, **S1**). We find a mix of different approaches to localization: **S1** and **S4** search for **string constants** (“*Search for strings or something [...] they are searchable and somewhat unique. When you don’t know naming, schema, folder structures and so on.*” **S1**) and **keywords** (“*I search the code base for keywords.*” **S4**) in the source code and work from there, in a **bottom-up** approach. **S3** starts with a **top-down** approach from **architectural overviews** but is ready to switch to a bottom-up approach (“*If you’re lucky, there’s some kind of architecture you can look at [...] Or you have to discuss it with someone or ask [...] otherwise, you’ll have to search the files manually*”, **S3**). **S2** focuses on the **data flow** and tries to trace it through the program to localize its source (“*The incoming data is wrong. OK, then we need to find who feeds that data and what is wrong with it.*”, **S2**), while **S5** focuses on the **control flow** and tries to locate a central point of the system to trace their way outwards from there. (“*I mean, a juncture, a manager, whatever where you don’t need to go further backwards or forwards. Here I can start working without familiarizing myself with too much.*”, **S5**).

Inspect: The source code of a software system is only part of what is needed to understand how it behaves. To be able to alter the behaviour of the running software, or identify anomalies, the data the code operates on needs to be observed. To systematically observe a part of the code along with its data, as it runs is to “inspect” the program. The subjects refer to **stepping** and **breakpoints**, that is, using a **debugger** to execute the program one instruction at a time, being able to inspect changes in its state for each step (“*I step and see what value all the variables have and let it run and [...] begin clicking a few buttons to see where the flow takes me.*”, **S5**; “*I’ve got a point before the crash/error/bug manifests. If possible, one can set a breakpoint there*” **S1**). However, using a debugger is not always possible (“*Systems very seldom are like ‘well, you just click these buttons in the IDE and run this and then switch to a debug perspective’*”, **S2**). **S1** and **S2** express experiences of situations where a debugger is unavailable and the, in their opinions, less ideal strategies of using **printouts** of partial states while the program runs in a normal manner (“*otherwise, one may go old school and print to trace out some data and see how it changes*”, **S1**; “*Sadly, it often becomes printf. [...] We suspect these variables have the wrong values, and in the end, well, we have to print the values and look at them*”, **S2**).

Recognize: Software source code often forms patterns. Many parts of the code may do similar things, and will over time tend to do similar things in similar ways. Common expressions of code are sometimes referred to as “idioms”. Some idioms have proved to be based on misunderstandings and may not work as intended; such are sometimes referred to as “**anti-patterns**”. A sufficiently experienced programmer may be able to *recognize* anti-patterns or mistakes in idioms simply by looking at them (“*... And there you say ‘This can probably go wrong’ [...] ‘This is the special case’ [...] that’s experience, right?*”, **S3**).

Simulate: Every programmer needs to make a mental model of the execution flow of a program to be able to understand it. This mental model can range from superficial to detailed. Making an exhaustive analysis of the program flow under different circumstances amounts to *simulating* the program execution in one’s mind (“*You more or less run the code in your head, and then you see it.*”, **S3**). A programmer with a sufficiently detailed mental representation of the program may be able to simulate several program runs and thereby reveal conditions under which the program executes abnormally or in an undesired

manner.

Contextualize: To understand what goes wrong in a piece of code, it may be helpful to know in what environment that code is written and how it connects to other parts of the system. A panoramic overview of the *context* of the code can be beneficial to large program comprehension. **S4** collects and documents the context of the source code and draws a **map** for further reference (“*I paint a small picture of that area. Step one is getting an overview of the full code base. [...] What does a flow look like? So it became a small flow chart. On paper.*”, **S4**).

Table 3 – Bug localization strategies

	(Katz & Anderson, 1987)	(Vessey, 1985)	(Decasse & Emde, 1988)	(Romero et al., 2007)
Reproduce	"testing [the] system"	"Determine problem"	"checking computational equivalence of intended program and actual one"	"Following execution"
Localize	"locating the erroneous component of [the] system"	"Locate error"	"filtering"	"Causal reasoning"
Simulate	"Mentally process data through program"			"Hand simulation"
Inspect		"Examine program control"		
Contextualize		"Gain familiarity"		"Comprehension"
Recognize			"recognizing stereotyped errors"	

4.3. Frustrations

Two points of frustration stood out in the interviews:

Tedious Tool Setup: A common frustration among our participants concerns the tool setup where they expect to encounter trouble, for instance, regarding how to set up a functioning **build process** (“*Often-times it may be hard to understand how to even build the software*”, **S3**). A related point of frustration is lacking support for **debugging** (“*It’s very uncommon with systems where you just click a few buttons in the IDE, and it starts and runs and you can just switch to a debug perspective.*”, **S2**; “*Integrated debuggers are still not standard, surprisingly*”, **S1**), or the need for a **time-consuming setup** to get working debug support (“*Sometimes it takes time to get the break-point functionality up and working, so that the code stops where you are interested. I really miss that*”, **S5**). More generally, developers may feel **abandoned** when it comes to tooling (“*Tools and that kind of stuff are often kind of neglected.*”, **S2**).

Lack of Visualization: Another point of frustration mentioned was connected to support for visualization and the lack of having up-to-date versions of unified modelling language (**UML**) **diagrams** (“*The way I see it, to have a UML diagram and keep it updated can for a team be hard*”, **S4**). In some cases a solution may be available but company policies connected to protection of **intellectual property** may prohibit their use (“*I have looked a lot at available [UML generator] tools. But I never quite understand if they upload stuff, you know, they may be online, and then I’m not allowed to use them.*”, **S4**).

4.4. Opportunities

Related to the frustrations several intervention opportunities emerged from the analysis of the interviews.

Quick and Easy Developer Tool Setup: Many of the more experienced programmers in this study have experienced situations where the tool chain is incomplete or dysfunctional. It may be hard to build the project from the IDE, it can be hard to trigger the test cases, or complicated to get code indexation and debugging to work. A more ready-to-go and **simpler development setup** is an area of opportunities for improvement, and might over a sufficiently large developer base provide a good return on investment (“*I mean, pull up visual studio code, check out the code, press the button with the icon and then run in debug mode. [...] I think that would have made things easier.*”, **S2**).

Backward-Stepping Debugging: A debugger is a tool that allows running a program one step at a time, and allows for inspection and alteration of its state between the steps. Typically, a debugger can only run the program forwards along a single path. **Backing up**, to see for example where a state originated from, is usually not possible. Yet, this could make an appreciated feature (*“I think some kind of runtime that could record an execution. [...] It would be like single-stepping after the fact. [...] Back and forth”*, S2).

Runtime Visualization: When using a debugger, it is important to continuously evaluate the **state** of the running program, but this could be further aided by tools (*“Some kind of visualization tool [...] like being able to see how the data changes”*, S1). Furthermore, it is important to keep track of where, in the source code, the program is currently executing, and what **path** it has taken to get to this point (*“See that OK, we enter this function with erroneous parameters, why is that? Who is calling?”*, S1; *“I think the data is wrong, but who is calling with this data? I sort of want to find the call graph, and that can be rather tricky.”*, S2). Visualizations of a program’s runtime paths and state change behaviour could provide an accessible overview and elevate a programmer’s understanding of the runtime behaviour of the code.

Static Visualization: A programmer working on a large program needs a mental model of how the parts of that program interconnect. When probed about the process of creating such a model, the participants in this study mentioned mental visualizations with relations, in the shape of UML and **block diagrams** (*“You know, a big part is some kind of visualization of the architecture, so you can see how the big blocks are connected ... like a block diagram.”*, S3). One subject express a wish for a tighter integration of UML into the IDE to be able to use it for navigation (*“In a dream world, I would have an interactive UML diagram that is already in place. [...] Be able to click around in the UML diagram instead.”*, S4).

Event Tracking: An event-based architecture differs from an object-oriented or a functional architecture in that functions may alter their behaviour in response to the properties of the data — “event” — being processed. These events are typically queued before being sent for processing. When looking at the runtime behaviour of such code, we will see multiple calls to the event processing function from the queue handler, but it will often be hard to know where the event was sent from. When debugging, an erroneous event may be identified, but tracing the origin of an event may be hard. Marking events with their **origin** may simplify tracing it (*“An easy way would be. [...] I queue the [event] object, and later when the event gets [processed], you can see the object that created it, S5).*

4.5. Summary

RQ: How do programmers approach the task of resolving their first bug?

We find that different programmers may have different strategies. While there is a significant overlap in the activities they mention, it is clear, even from this small sample, that not all programmers approach bug fixing in an unfamiliar large code base the same way.

We also find that four of the five programmers in our study think they would benefit, in this task, from visualization tools aiding navigation or run time monitoring.

5. Threats to Validity

We base our analysis and definitions of threats to validity on Feldt and Magazinius (Feldt & Magazinius, 2010).

Conclusion validity. We conduct an exploratory study, meaning our conclusions should be taken as indicators and inspiration for further study, rather than as verifiable truth.

Construct validity We explore initial code base comprehension by conducting interviews. This limits the results in several ways: *a)* it only collects strategies that are *conscious*, *b)* it only collects actions the subject can *remember*, *c)* it only collects actions the subject *wants to talk about*. A method to investigate

strategies in initial code base comprehension is to observe the programmers' behaviour in the wild, as done in for example (Alaboudi & LaToza, 2023), however, to uncover strategies this may need to include think-aloud methods, that may in turn be invasive and affect the results.

External validity/transferability. The interview subjects were arbitrarily selected from the network of the first author in the geographical region of the authors, using convenient sampling. As such, they might be influenced by local factors and practices. The interview study is further quite small: only five subjects. The results from the interviews may not generalize to other contexts. The results of this exploratory study should be considered inspirational, rather than a ground truth of the experiences of a programmer community.

Credibility. The credibility of our findings comes down to the credibility of our subjects. As such, the credibility hinges upon the internal validity (in particular response bias) and the construct validity. Since the topic and questions are uncontroversial, we have no reason to believe our subjects have systematically tried to mislead us.

Dependability. We are confident that a similar *Problem Exploration* study would lead to similar diversity in results in terms of *attitudes*, *strategies* and *frustrations*, but we expect there is more to find here. A set of subjects with a different background may see or emphasize other *opportunities*.

Confirmability. The results from the *Problem Exploration* are influenced by the interview guide, but the answers are the subjects' own. The data analysis is done responsively and reactively, carefully considering what the subjects have said and how the answers correlate.

6. Discussion

Approaching a new large code base in a professional setting is an endeavour. The programmers we talked to find it *fun and overwhelming*, but it is also something they describe as *frightening* and that makes them *very, very nervous*. A software programmer needs to learn to orient – to find their way around – the new code base. It is unsurprising, then, that they express the need for *someone to [...] guide* them.

The programmers we have interviewed seem to have well-formed strategies made up of a string of activities they follow when they undertake to fix their first bug in a new code base. Some activities are shared among the programmers, but not all. It is clear, even from this small convenient sample, that not all programmers employ the same strategy.

A common frustration seems to be that current tools are perceived as unreliable – or at least as cumbersome to get and keep in a reliable state. They seem to have limited support for the different strategies employed by the programmers. Indeed, several programmers mention activities specifically executed to work around the limitations in the tool support.

The tools available to the programmers do not feature integrated visualisation to support the construction of mental models or aid navigation in code bases. Reliable integrated visualisation of static and dynamic aspects of the code is requested as a means of facilitating code base comprehension.

6.1. Directions for Future Work

We see several possible directions for future work:

Effective visualizations for large program comprehension. The experience of starting working with a new code base may be comparable to finding oneself in a new and unfamiliar physical space. To find one's way around a new and unfamiliar landscape, it comes as no surprise that many of the programmers we interviewed want a map to navigate by. What should such a map look like in practice? What kinds of questions should it be capable of answering? While UML is mentioned by some of the subjects in this study, there are more alternatives. Hawes et al. use a territory map metaphor to visualize large code bases and their internal dependencies (Hawes, Marshall, & Anslow, 2015). Mortara et al. use the city as a metaphor (Mortara, Collet, & Dery-Pinna, 2021), and Hori et al. visualize source code structure in as

a house (Hori, Kawakami, & Ichii, 2019).

Understanding the experience of shared code spaces. With the analogy of travelling, some of the programmers we interviewed plan to stay a while in a new code base, while others clearly see themselves as temporary visitors. Perhaps in relation to this intention, we saw a variation of frustration with a slight tendency towards an increase when the code base is about to become a new “home”. We see an interesting parallel to the work by Church et al. (Church, Söderberg, & Höst, 2023), in how ownership plays a role in the relationship to the shared space of a code base. How do programmers relate to code ownership? How does a feeling of code ownership emerge?

Understanding how experience shapes preferences. Our participants steered towards UML when asked about what they visualized or wanted to be visualized. With our sample of programmers being concentrated to a certain geographical region, where the likelihood of a similar educational background is high, we speculate that this affinity to UML may be due to a similar educational background. The notion that our experiences in the programming context will bias our preferences, is not new. Meyerovich et al. (Meyerovich & Rabkin, 2012) have studied this aspect in the setting of programming language adoption. How do programming experience influence preferences for programming tools?

Effect of live programming on code base comprehension. Live programming (Tanimoto, 2013; Church, Söderberg, Bracha, & Tanimoto, 2016), focused on immediate feedback about program state and runtime information, provides properties that may assist based on what we found in our problem exploration. Our participants expressed a need for visualization and seeing connections between the visualisation and the code. They further expressed a need for debugging support for a deeper understanding of code behavior. Live programming has been found to have a positive effect in an educational setting (Huang, Ferdowsi, Selvaraj, Soosai Raj, & Lerner, 2022). Provided a functioning workflow integration, would live programming support provide effective support for code base comprehension?

6.2. Limitations

This paper describes an exploratory study into programmers experiences with initial code base comprehension. The study is based on a small sample from one geographical area and the participants have similar educational and professional backgrounds. To make the study more generalized it would be possible to, for instance, carry out an observational study in the wild with a larger set of participants.

7. Conclusions

We have presented the results of an exploratory study focused on understanding code base comprehension. We found that the programmers in our study approached a new code base with a mix of anticipation and dread while expecting a lack of functioning tool support. The most apparent lack in tool support we found concerned easy tool setup, debugging, and visualization support. We find these results encouraging and an indication that code base comprehension is an understudied area worthy of more attention.

8. Acknowledgements

This work has been supported by Ericsson AB, 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.

Special thanks to Lo Heander and Peng Kuang for inspiration and discussions.

9. References

- Alaboudi, A., & LaToza, T. D. (2023). What constitutes debugging? an exploratory study of debugging episodes. *Empirical Software Engineering*, 28(5), 117.
- 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.
- Brooks, R. (1978). Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on software engineering* (pp. 196–201).

- Church, L., Söderberg, E., Bracha, G., & Tanimoto, S. (2016). Liveness becomes entelechy-a scheme for l6. In *The second international conference on live coding*.
- Church, L., Söderberg, E., & Höst, M. (2023). My space, our space, their space: A first glance at developers' experience of spaces. In *Companion proceedings of the 7th international conference on the art, science, and engineering of programming* (pp. 48–53).
- Decasse, M., & Emde, A.-M. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. In *Proceedings.[1989] 11th international conference on software engineering* (pp. 162–163).
- Feldt, R., & Magazinius, A. (2010). Validity threats in empirical software engineering research-an initial survey. In *Seke* (pp. 374–379).
- Hawes, N., Marshall, S., & Anslow, C. (2015). Codesurveyor: Mapping large-scale software to aid in code comprehension. In *2015 ieee 3rd working conference on software visualization (vissoft)* (pp. 96–105).
- Hirsch, T., & Hofer, B. (2021). What we can learn from how programmers debug their code. In *2021 ieee/acm 8th international workshop on software engineering research and industrial practice (ser&ip)* (pp. 37–40).
- Hori, A., Kawakami, M., & Ichii, M. (2019). Code house: Vr code visualization tool [Conference paper]. In (p. 83 – 87).
- Hsieh, H.-F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative Health Research*, 15(9), 1277-1288. (PMID: 16204405)
- Huang, R., Ferdowsi, K., Selvaraj, A., Soosai Raj, A. G., & Lerner, S. (2022). Investigating the impact of using a live programming environment in a cs1 course. In *Proceedings of the 53rd acm technical symposium on computer science education-volume 1* (pp. 495–501).
- Katz, I. R., & Anderson, J. R. (1987). Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, 3(4), 351–399.
- Kulkarni, A. (2016). Comprehending source code of large software system for reuse [Conference paper]. In (Vol. 2016-July).
- Meyerovich, L. A., & Rabkin, A. S. (2012). Socio-plt: Principles for programming language adoption. In *Proceedings of the acm international symposium on new ideas, new paradigms, and reflections on programming and software* (pp. 39–54).
- Minelli, R., Mocci, A., & Lanza, M. (2015). I know what you did last summer-an investigation of how developers spend their time. In *2015 ieee 23rd international conference on program comprehension* (pp. 25–35).
- Mortara, J., Collet, P., & Dery-Pinna, A.-M. (2021). Visualization of object-oriented variability implementations as cities [Conference paper]. In (p. 76 – 87).
- Robson, C. (2011). *Real world research*. John Wiley & Sons.
- Romero, P., Du Boulay, B., Cox, R., Lutz, R., & Bryant, S. (2007). Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human-Computer Studies*, 65(12), 992–1009.
- Shah, A., Yu, J., Tong, T., & Raj, A. G. S. (2024). Working with large code bases: A cognitive apprenticeship approach to teaching software engineering [Conference paper]. In (Vol. 1, p. 1209 – 1215).
- Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8, 219–238.
- Siegmund, J. (2016). Program comprehension: Past, present, and future. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 5, pp. 13–20).
- Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on software engineering*(5), 595–609.
- Tanimoto, S. L. (2013). A perspective on the evolution of live programming. In *2013 1st international workshop on live programming (live)* (pp. 31–34).

- Tempero, E., & Ralph, P. (2018). Towards understanding programs by counting objects [Conference paper]. In (p. 1 – 10).
- Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5), 459–494.
- Von Mayrhauser, A., Vans, A. M., & Howe, A. E. (1997). Program understanding behaviour during enhancement of large-scale software [Article]. *Journal of Software Maintenance and Evolution*, 9(5), 299 – 327.
- Wang, D., Galster, M., & Morales-Trujillo, M. (2023). A systematic mapping study of bug reproduction and localization. *Information and Software Technology*, 107338.
- Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8), 707–740.
- Wuilmart, P., Söderberg, E., & Höst, M. (2023). Programmer stories, stories for programmers: Exploring storytelling in software development. In *Companion proceedings of the 7th international conference on the art, science, and engineering of programming* (pp. 68–75).
- Wyrich, M., Bogner, J., & Wagner, S. (2023, nov). 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Comput. Surv.*, 56(4).
- Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10), 951-976. doi: 10.1109/TSE.2017.2734091

A. Interview guide

The subjects were reminded that the interview was recorded and that the data is processed with informed consent according to research standards.

Personal information

- Alias
- Age
- Gender identification

Professional Experience

- How would you describe your experience in the software industry?
- Number of workplaces have you worked in? (A new assignment in the same company may be a new workplace - extract the number of new codebases the subject worked with!)
- Number of programming languages?
- Years in the business?
- Anything else?

Starting a new job

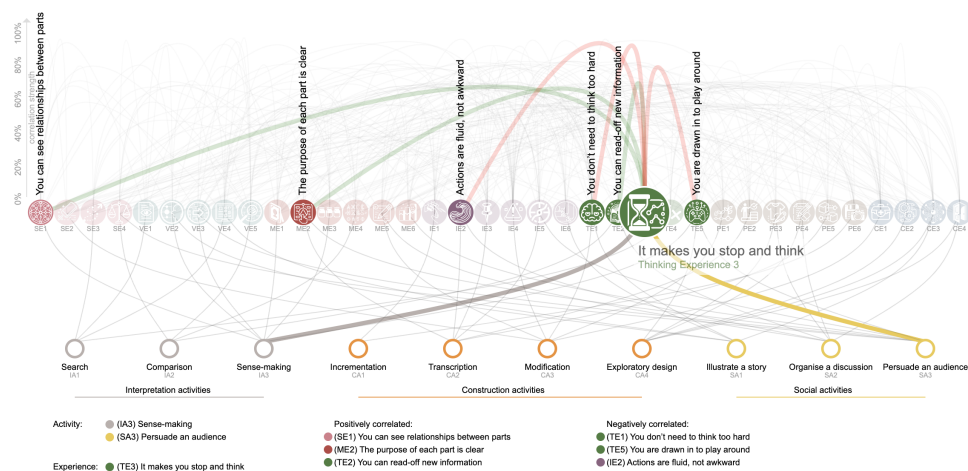
- When you start in a new workplace - how do you feel?
- What do you do when you are assigned your first bug in a new workplace?
- What strategies do you employ – to fulfil what goals
- What trouble do you get into?
- What tools do you employ? Why?
- What would you want to support you in your work with your first bug?

PUX Explorer: An Interactive Critique and Ideation Tool for Notation Designers

Justas Brazauskas
Computer Laboratory
University of Cambridge
jb2328@cam.ac.uk

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

PUX Thinking Experience 3 : It makes you stop and think



It makes you stop and think

Summary:

Awkwardness can aid memory and problem-solving, contradicting ease.

Original Text:

Although interaction designers generally focus on ease of use (IE2), there are many situations where a little awkwardness can actually be beneficial. Studies of educational and problem-solving contexts show that making the diagram user work a little harder to interpret what they see makes it more memorable, and can even mean that problems get solved faster, because the user stops to think. These benefits are obviously in tension with TE1 and TE5. However, clarity of meaning is still likely to be beneficial (SE1, ME2, TE2) if it means that the user is drawing correct conclusions rather than getting confused.

Figure 1 – Overview of the PUX Explorer, showing the main elements that will be discussed in section V and VI.

Abstract

PUX Explorer is a meta-design tool for use by designers of programming languages and other notational systems, in the tradition of Green's Cognitive Dimensions. Together with PUX Matrix and PUX Personas, these tools build on critical frameworks for notation design, informed by a general theory of design ideation. We evaluated PUX Explorer in a controlled study of meta-design, with specialist designers of new music notations. We find that these tools are effective and accessible design aids for meta-designers, not requiring specialist technical expertise.

1. Introduction

The design of novel programming languages can be informed by systematic documentation of the usability issues and tradeoffs that users experience when they need to understand or create information structures in any formal notation. However, formulating design guidance in a replicable process is challenging, especially because users with different specialist training or different task requirements will need different things from the notation.

There have been many attempts to formalise the design space of notational systems, in particular, the Cognitive Dimensions of Notations framework (CDNs) originally proposed by Thomas Green in 1989 (Green, 1989), and since extended and adapted by many researchers (Hadhrawi, Blackwell, & Church, 2017). One of those variants is the Patterns of User Experience (PUX) developed by Blackwell and Fincher (Blackwell & Fincher, 2010), which is taken as the starting point for this paper. PUX (Blackwell, 2024) describes a pattern language of notational activities (e.g. IA3 sense-making, CA4 exploratory design or SA3 persuasion), and experiences that users may have with a notation while undertaking those activities (e.g. VE2 the overall story is clear or IE3 things stay where you put them). As with CDNs, design choices in the notation and its environment make some kinds of experience more likely than

others, leading to trade-offs. Also as with CDNs, these may be more or less desirable depending on what activities the notation is to be used for.

The most widely adopted process for application of the CDNs was the Cognitive Dimensions Questionnaire, which could be used in semi-structured interviews or design reviews to assess a notational system in relation to the properties described by the framework (Blackwell & Green, 2000). Although more approachable than tutorial descriptions of the method, the questionnaire did not offer direct design recommendations or quantifiable measures. Although there has been steady development and adoption of systematic vocabularies for analysis of tradeoffs in notation design since then, the field has been hampered by a lack of interactive design tools. One notable exception is Clarke’s tool for use in the design of APIs, which characterised usability profiles of any API that was under development in relation to different programmer personas (Clarke & Becker, 2003).

In this paper we present an interactive tool specifically for use by notation designers, supporting the systematic investigation of notation usability properties such as those described in CDNs, in PUX, and in similar proposals such as Moody’s PoNs (Van Der Linden & Hadar, 2018). The contributions of this work are as follows:

- Introduction of the PUX Explorer tool for systematic investigation of notation usability during both formative and summative phases of notation design.
- A theoretical characterisation of the notation design process, in terms of co-evolution of problem and solution spaces.
- A controlled evaluation study in which PUX Explorer is compared to an alternative interactive tool that uses a conventional feature matrix approach.

2. Co-evolution in Notation Design

The design and critique of new notation systems falls into the class of “meta-design”, since the notations being created are generally intended to be used by other designers (Fischer, Giaccardi, Ye, Sutcliffe, & Mehandjiev, 2004). Earlier frameworks such as CDNs and PoNs did not include specific theories of design, relying instead on critique of the theoretical principles by which HCI guidance was formulated.

The original presentation of PUX (Blackwell & Fincher, 2010) did make explicit reference to Christopher Alexander’s Pattern Languages (Alexander, 2018) as a theory of design, but made no claims regarding the meta-design process by which PUX might be applied. This was problematic since there is already conflicting evidence regarding the value of pattern languages in interaction design processes (Dearden & Finlay, 2006).

Our current work approaches the problem of meta-design in relation to Nathan Crilly’s theory of ideation and critique in design processes (Crilly, 2021b, 2021a), which acknowledges the fundamental role of co-evolution, where the skilled designer does not simply translate predetermined requirements into product features, but rather constantly considers alternatives in both the problem space and the solution space, with the eventual design outcome reflecting a co-evolved alignment of the two.

We propose that co-evolution is the most appropriate theoretical framework by which to construct and assess meta-design tools such as PUX Explorer, since these are expected to offer benefits both in problem-oriented formative analysis (deciding what kind of notation to design) and in solution-oriented summative analysis (assessing whether the designed notation will be effective). As described later in this paper, we therefore used the perspective of co-evolution to design our own evaluation study.

3. PUX Explorer Functionality

PUX Explorer is a web application intended to help notation designers explore the activities and experiences described in the PUX framework ¹, implemented in Javascript using the d3.js library (Bostock,

¹PUX Explorer can be accessed at <https://jb2328.github.io/PUX-Diagrams/>

2024). The design process used to create PUX Explorer is described further below.

An overview of PUX Explorer can be seen in Figure 1, centred around two rows of circles that represent 36 experiences and 10 activities.

3.1. Primary operation

The primary mode of operation for PUX Explorer is for a designer to investigate the ways in which a candidate design delivers specific user experiences that will be associated with a given notational activity. The values used to prioritise and weight activities for different types of user can be established using the PUX Persona tool, which is described below.

PUX Explorer operates as an interactive diagram, allowing the user to isolate and explore the perspective of different activities. The overall structure of the visualisation is explored by mousing over the elements. Hovering on one activity shows the perspective of that specific activity, highlighting the notational experiences that are most salient.

Hovering on any one of the experiences, as shown in Figure 1, highlights the design trade-offs and synergies that exist with other experiences, and also indicates which other activities this experience might be associated with. Negative and positive associations (trade-offs and synergies) are indicated with red and green links, while stronger associations are indicated by the curve rising higher on the screen for greater prominence.

The PUX Explorer provides a targeted browsing interface guiding viewers to the documentation of the most relevant design guidance and properties among the many aspects of the framework. The tool converts the PUX framework into an interactive diagram, enabling integrated navigation of the entire framework.

3.2. Evidence Journey

Initially, the user sees two rows of distinctive circular icons, with the top row of 36 experiences organized into seven colour-coded groups, and the bottom row of 10 activities segmented into three groups. Activities are linked to experiences, and each experience is connected to others by grey arcs representing trade-offs. The arcs become highlighted and animated as the user hovers their mouse over different icons.

Whenever an experience or activity circle is hovered on, the right-hand side of the visualisation provides a description, summary and textual narrative for design guidance. This information can be captured for transfer to design documents, and a trail of the experiences identified as being relevant is maintained at the bottom of the screen.

Hovering over an *experience* icon highlights its connections to other experiences through green and red lines, indicating positive synergies and negative tradeoffs. Hovering enlarges the icon and animates lines that grow from there to the destination, illustrating directionality. Similarly, hovering over an *activity* circle emphasizes and animates the links to the experiences associated with that activity.

These animations give the Explorer tool a playful feel, including a degree of jitter that is designed to encourage users to explore the entire framework, avoiding premature design fixation and facilitating serendipitous discovery. The tool is designed such that the exploratory phase is led by animated lines. Once a user selects the appropriate experience or activity by clicking on it, the animation freezes until the unlock button is pressed.

See Appendix A for a zoomed-in view of the experiences and activities, as well as Appendix B for a deconstructed view of the PUX Explorer UI.

4. Development Process

PUX Explorer was developed through a potentially replicable process, beginning with a canonical presentation of the PUX framework (Blackwell, 2024). That textbook chapter uses conventional typographical structure (lists, section headings, and cross-reference codes) to support reference consultation for

application by meta-designers. PUX Explorer make this process interactive instead of typographic. Below, we provide two examples – one for an activity and one for an experience – to illustrate how we transformed the textual descriptions of the framework into an interactive tool.

4.1. Activities profiles

In the canonical PUX description, different types of activity (Interpretation, Construction, and Social) are organised into subchapters, with individual activities being described as paragraphs in text, and related experiences listed at the end of the paragraph as follows:

Interpretation Activity 3 (IA3): Sense-making

For example: What is the best route, and time of day, to make a new journey? The user is trying to learn about a new situation, or integrate data of a kind they haven't seen before. This involves understanding the overall structure, how parts are related to each other, and which are most important. Comparing different parts and aspects of the structure will be an important aspect of sense-making, so aspects of pattern IA2 will also be relevant.

```
{
  "name": "Sense-making",
  "id": "IA3",
  "links_to": ["VE2", "VE3", "SE1",
              "ME1", "ME3", "TE3", "TE5"]
}
```

Relevant experience patterns include *VE2*, *VE3*, *SE1*, *ME1*, *ME3*, *TE3*, *TE5*

We translate this descriptive text and cross-references into data structures to define the interactive tools behaviour (JSON data structure for a sample activity (Sense-making IA3) (Listing 4.1)).

4.2. Experience profiles and Tradeoff analysis

Experiences are described in groups of unique segments, such as Visual Experience (VE) in the example below. However, unlike the descriptions of activities, tradeoff links are not explicitly defined at the end of the paragraph and must be inferred from the text itself. Here we illustrate how the JSON data structure was extracted for a sample experience (The overall story is clear VE2):

VE2: The overall story is clear

People often say they prefer diagrams to text because they get a kind of 'gestalt' view of the whole information structure – you can stand back and look at the overall configuration, and get a good idea of the whole story. Of course, it needs to be visible for this to work (patterns *VE1* and *SE1*), but sometimes it is possible to leave out some of the detail in order to improve this overall understanding (pattern *VE5*).

```
{
  "name": "The overall story is clear",
  "id": "VE2",
  "link_positive": [{"VE1": 0.9}, {"SE1": 0.8}],
  "link_negative": [{"VE5": -0.7}]
}
```

Although the description delineates a clear relationship between VE2 and the three related experiences (VE1, SE1, and VE5), it does not explicitly define a tradeoff relationship.

We used textual sentiment analysis to determine whether the relationships to other experiences were described positively or negatively, and assigned a numerical value ranging from -1 (a negative tradeoff) to 1 (a positive synergy).

For example, the textbook-style descriptions of VE1 and VE5 are introduced with a positive sentiment ("of course, it needs to be visible..."), followed by a negative sentiment ("but sometimes it is possible to omit detail..."). This analysis yields the following data structure for use in the tool's data visualisation (Listing 4.2):

The extracted structural encoding in Listings 4.1 and 4.2 encodes the entire structure of PUX framework as a graph that can be visualised either as an incrementally interactive diagram (PUX Explorer) or a holistic overview (PUX Matrix).

4.3. PUX Explorer icon design

Where the textbook description refers to activities and experiences using three-character codes that have limited visual or mnemonic value, we created unique icons for each experience, and distinct colour coding for activities, ensuring clear visual differentiation between the two (shown in Figure 1 and Appendix A).

Thirty-six icons were developed using a collaborative design process enabled by generative AI:

1. *Generating design ideas.* The PUX description of each experience was input into a large language model (LLM), prompted to create three different design concepts for an icon. For example, the LLM suggested that the concept "*SE4: You can compare or contrast different parts*" could be symbolised by

a scale or balance icon as shown by the final row of icons in Figure 2. In some cases generic ideas were repeatedly proposed for different experiences (e.g. the magnifying glass in rows 1 and 3 of Figure 2), so the design team brainstormed alternatives to supplement the LLM output with more distinctive ideas.

2. *Generating icons from the created prompts.* The three design concepts for each icon were used as prompts for a Stable Diffusion (SD) model, with uniform style descriptors "2D flat design, vector, white background, minimalist" added to all prompts.

3. *Selection process.* At least three design options were created for each experience, as illustrated in Figure 2. Two raters experienced with the PUX framework independently selected the most appropriate icon from each set, making the final choice after reaching consensus.

4. *Icon cleanup.* The selected design was then vectorised from its original PNG format, refined in Adobe Illustrator, and saved as an SVG file for integration into the PUX Explorer tool as an icon.



Figure 2 – Icon selection process for the PUX Explorer tool. The final icon choices at the left of each row were selected from the three options to the right, as generated using Stable Diffusion.

Throughout the design process, a total of 112 icons were created, with 102 automatically generated, and 10 requiring some degree of manual intervention. In the final selection, 35 out of the 36 icons had been generated by SD, highlighting the efficiency of this LLM-enabled design process.

4.4. PUX Matrix

Our second interactive tool developed, the PUX Matrix (Figure 3), renders the same structural encoding of the PUX framework to emphasise an overview of interconnectedness among activities and experiences. PUX Matrix is inspired by the *contradiction matrix* that is a familiar element of the TRIZ process for inventive problem solving (Ilevbare, Probert, & Phaal, 2013). (The *inventive principles*, *standard solutions* and *separation process* of TRIZ can be considered as a rough analogy to the activities, experiences and trade-offs in PUX).

The PUX Matrix tool presents two rectangular grids corresponding to the two rows of icons and links in the PUX Explorer. The left grid shows links between activities and experiences as black dots in the corresponding cells. The right side uses red and green squares to indicate trade-offs and synergy between experiences.

PUX Matrix provides an overview of the whole framework, mapping regions in which experiences share similar patterns of trade-offs. The Matrix tool is more dense, but less interactive than the Explorer. As with the TRIZ contradiction matrix, detailed descriptions of each activity and experience are not included in the visual presentation, meaning that a separate text reference would need to be consulted. The tool in its entirety is shown in the appendix (Appendix C), along with a deconstructed version

explaining its visual elements (Appendix D).

4.5. PUX Personas

The final tool complementing PUX Explorer and PUX Matrix is PUX Personas, inspired by Clarke’s characterisation of programmer personas for API usability (Clarke & Becker, 2003). Previous usability questionnaires based on CDs and PUX have asked respondents to estimate what proportion of their time is spent in different activities. In the PUX Persona tool, different activity profiles are created for different user personas, and interactively visualised with polar area charts (also called Nightingale Rose Charts (Maguello, 2012)). The full tool is presented in Appendix E.

Utilisation of this tool involves three steps:

- 1) *Time-allocation.* Initially, users estimate the proportion of time, as a percentage, that this persona would spend on each of the 10 activity types from the PUX framework. This generates a pie chart where the angle of each slice corresponds to the proportion of time for that activity.
- 2) *Rating experiences.* Users then rate the comparative importance of each PUX experience within their notational environment using a 5-point Likert scale. Experiences are arranged vertically in the same order as in the PUX Explorer and Matrix tools, so that users can refer to descriptions and tradeoffs. As the rating of relevant experiences is adjusted, this determines the radius of the pie chart segment for the activity associated with that experience.
- 3) *Creating a visual user representation.* The final result of the activity and experience profiles is a rose diagram (Figure 4), where the polar area represents notation design priorities for a specific type of user. These visual persona representations can be used as a design aid and reference when making tradeoff decisions that will have differential benefits for different classes of users, or providing configuration capabilities relevant to a specific user class. In the depicted example, although a person spends a significant amount of time on *incrementation* activities, they rate other activities like *organising discussions* as more important, even though less time is spent on them. The full UI is shown in Appendix E.

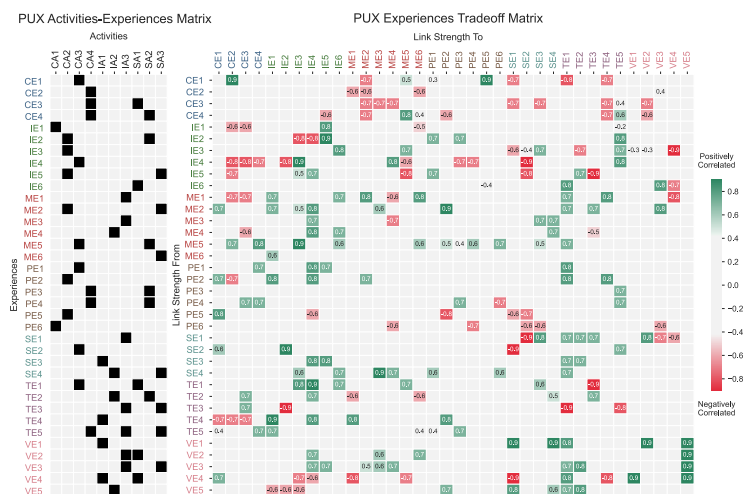


Figure 3 – A simplified view of the PUX Matrix tool. The complete matrix can be found in Appendix C along with a deconstructed version in Appendix D.

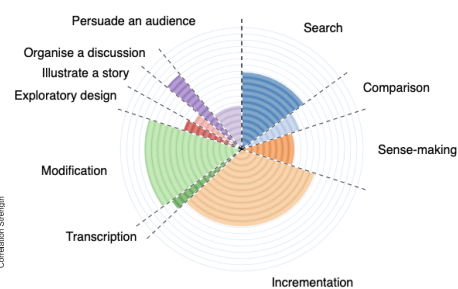


Figure 4 – Polar area chart generated by the PUX Personas tool. Slices indicate the percentage of time dedicated to specific activities (angle) and their importance (radius).

5. Evaluation

As an evaluation of the PUX Explorer tool, we chose to work with the same domain used for the initial evaluation of the Cognitive Dimensions Questionnaire (CDQ) (Blackwell & Green, 2000), which until now has been the most widely used research tool for analysis of notation usability (Hadhrawi et al., 2017). The paper introducing CDQ reported a study of music researchers who worked with and designed alternative music notations (Blackwell & Green, 2000). The advantage of the music notation domain for

this type of research is that music researchers are familiar with a wide range of notation alternatives, from experimental graphic scores, to performance annotation, to formal musicological analysis. Musicians and music researchers routinely use a variety of computer-based, print, and pencil modifications. They are also accustomed to describing properties of a notation with an analytic distance from the semantic content of the music as heard or played. These factors mean that music researchers are better able, for example, than many mathematicians to consider distinctions between concrete syntax and variation in styles of perception and usage that have very different degrees of formal rigour or creative freedom.

We used music research community contacts to recruit a sample of 6 specialist music notation researchers from universities and music colleges across the UK, USA and Europe. All participants had considerable experience as researchers and practitioners designing novel notations or music visualisations, and all had a pre-existing concern (in one case, years of experimental work) with the usability properties of their systems.

All materials were made available online. Participants completed the study on their own computers, at a time of their choice, with the experimenters available for contact if needed. The study was approved by the Cambridge Computer Science Ethics Committee.

5.1. Structure of the Study

This study was designed to evaluate and compare the effectiveness of PUX Explorer and PUX Matrix during the critique and problem reformulation phase of a co-evolution design process. Two preliminary exercises introduced participants to the perspective of analysing notation systems in a preamble, and familiarised them with the overall operation of both PUX Explorer and PUX Matrix, followed by a design task in which participants used the tools to analyse design options, concerns and opportunities relevant to their own notation design project.

5.1.1. Preamble

The preamble introduced the PUX framework, explaining the concepts of activities, experiences, and trade-offs within the design process. Both PUX Explorer and PUX Matrix tools were then introduced. The presentation order of the two tools was counterbalanced across participants. To ensure every participant received the same introduction to the tool behaviour, two one-minute videos were created, demonstrating user interaction with the PUX Explorer and PUX Matrix. Participants were also provided with a deconstructed view of each tool (Appendices B and D). These views explained the operation of individual UI components and guidance on interpreting the UI.

5.1.2. Familiarisation

As an introductory exercise, participants were asked to evaluate four sample data visualisations, in relation to the activities *Illustrating a story (SA1)* and *Persuading an audience (SA3)*. As an example likely to be familiar to an international audience, we sourced visualisations of the 2020 US election results from four major news outlets: BBC(BBC News, 2020), CNN(CNN, 2020), The Economist(The Economist, 2020), and Bloomberg(Bloomberg, 2020). The data presented in these visualisations was similar across all four sources. They differed primarily in their visual language and graphic design elements, allowing for comparison of notational properties.

5.2. Design Task

In the core design task, participants were asked to evaluate music notation systems they had designed themselves or with which they were extensively familiar. They completed five tasks using both PUX Explorer and PUX Matrix. The first four tasks involved analysing their notation systems through the lens of the PUX framework, considering both design priorities and problem reformulation. The final task was a direct comparison of the Explorer and Matrix tools.

5.2.1. Pre-existing problems (Task 1/5)

In the first task, participants were asked to identify current design issues and problematic parts of their notation system. They were asked to consider how their notation might be used by diverse kinds of user, engaged in a variety of activities. Participants used either the Matrix or Explorer tool (order-balanced across participants) to identify problematic components using the PUX framework, as guided by the

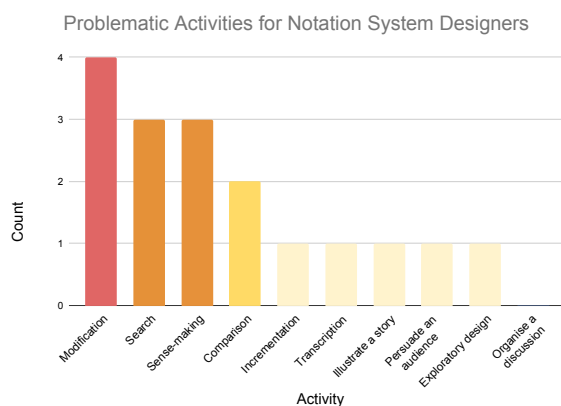


Figure 6 – Pre-existing problematic activities

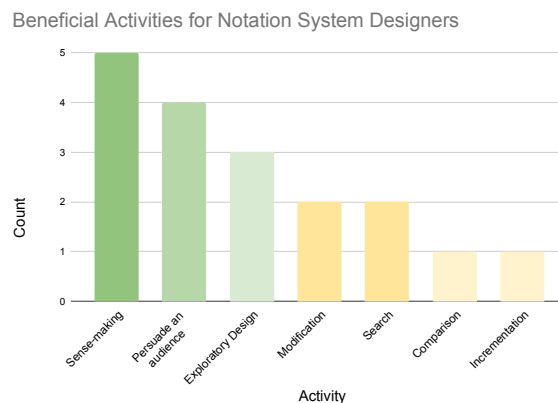


Figure 7 – Pre-existing beneficial activities

(SE2), and *Not needing to think too hard* (TE1).

6.2. Identifying pre-existing benefits (Task 2/5)

The most commonly cited activities benefitting from the notation systems were *Sense-making* (IA1) and *Persuading an audience* (SA3), as shown in Figure 7. Participants identified the most important experiences as *Being able to change one's mind easily* (SE2), *Being able to compare and contrast different parts* (SE4), and notation system elements *looking like what they describe* (ME1).

6.3. Identifying new design opportunities (Task 3/5)

Three of the six participants provided examples of new UX patterns that they had identified. However, when asked whether the tools had helped to identify new issues, Likert scale responses were mixed (using a 5-point scale: 1. Strongly Disagree, 2. Disagree, 3. Neutral, 4. Agree, 5. Strongly Agree). The Matrix tool had a mean rating of 2.2 (median 1.5), and the Explorer tool had a mean rating of 2.8 (median 2.5), indicating a neutral to negative view on the use of these tools to identify new issues with the existing notation systems.

6.4. Redefining pre-existing problems (co-evolution) (Task 4/5)

When participants were asked if they had considered reformulating the changes they planned to make, 3 out of 6 participants agreed they were considering reformulating the changes listed in Task (1/5). Both Explorer and Matrix users indicated that they somewhat considered reformulating the changes (mean 3.2, median 4). However, when asked if the PUX framework assisted in the reformulation process, Explorer users responded more positively (mean 3.8, median 4) than Matrix users (mean 3.2, median 4).

When combining responses to tasks 3 and 4, we found that 5 out of 6 participants agreed that they had either reformulated the problem or discovered new issues. Two participants reported reformulating, two reported new discoveries, and one participant did both. We consider these reports further in the discussion section below.

6.5. Tool preference results (Task 5/5)

We found a strong preference for the Explorer tool over the Matrix, both in terms of identifying existing issues with the design, as well as uncovering new ones. For confirming existing issues, PUX Explorer had a mean of 3.8 and a median of 4, compared to the PUX Matrix's mean of 2.5 and median of 2. For uncovering new issues, PUX Explorer scored a mean of 4 with a median of 4, whereas PUX Matrix scored a mean of 2.25 and a median of 2. These results are detailed in Figure 8.

7. Discussion

Overall, Likert scale responses indicated a generally positive assessment of PUX Explorer and a neutral attitude to the PUX Matrix tool; however, the different affordances of the two tools within the design process, particularly in supporting design co-evolution, offer interesting insights.

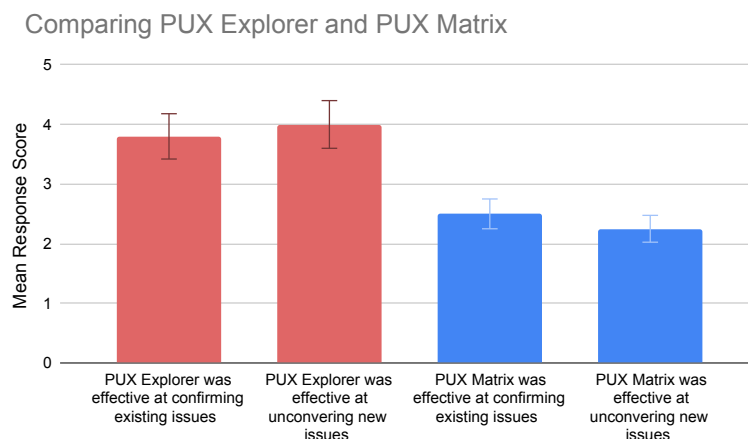


Figure 8 – The Explorer tool was preferred over the Matrix. The y-axis describes the average Likert score.

7.1. Preference for the PUX Explorer

Study data shows that both tools were effective in their use for notation designers with multiple positive comments preferring PUX Explorer.

Our initial expectation was that a more dynamic, interactive and visually engaging tool would be more appealing than a static matrix approach to visualising the PUX framework. The findings generally suggest that this was the case, as directly echoed by the study participants, e.g., P0A: “*PUX Explorer provides far more interactive exploration than PUX Matrix [and] engaged to discover more*”, and P2A: “*I think the PUX Explorer provides a more visually appealing experience*”, “*easier to navigate*” than the Matrix.

The PUX Explorer was perceived as engaging — P7B stated “*I like very much PUX Explorer, love dwelling into it*” and “*I find PUX Explorer more useful and fun and easier to use*”. P2B agreed “*the PUX Explorer provides a more visually appealing experience and one that is easier to navigate.*”

In contrast, P7B reported “*[The PUX Matrix] table is too difficult for me to read, I prefer looking at the Explorer tool directly*”, and that they had “*some difficulties reading and grasping PUX Matrix*”, because it was “*difficult to read, and the need to move head, find alignments, etc.*”, and additionally problematic because “*the concepts are not explained*” as in the PUX Explorer tool.

Nevertheless, beneficial aspects of the Matrix were identified — P4B stated that “*whereas the Explorer tool is more appealing and easy to navigate initially, once the Matrix has been used for a while it becomes more useful*”. This suggests that after familiarity with the PUX framework has been acquired, the Matrix can be a quick reference tool, as P4B later stated: “*when I did more detailed analyses I tended to revert to the Matrix*”.

Overall, the Explorer tool was considered superior for uncovering new issues and confirming existing design problems. We feel this is encouraging for further investigation of the PUX Personas tool, which is derived from that interaction approach.

7.2. Interaction Design of the PUX Explorer

The PUX Explorer was designed specifically to offer a dynamic experience, in which an overview map of the whole framework can be dynamically explored by mousing over different parts, with live animations drawing attention to the structural relationships. An essential interaction feature was the ability to “lock” the visualisation to zoom in and give more careful consideration to a specific pattern. This design strategy was generally effective — as P0A put it “*PUX Explorer provides far more interactive exploration than PUX Matrix, which kept me engaged to discover more.*”

However, this approach was not universally liked. P1A noted that “*The icons were too small to be visually useful [...] The quick “jittery” response of the tool meant that very complex information shifted quickly before I could really process it [...] It took a while to understand the benefit of LOCK and UNLOCK in this regard!*”. P4B was concerned that “*In Explorer things disappear and this makes the Matrix more systematic*”, realising only later that they had forgotten to use the lock function.

While the dynamic exploratory animation was generally appreciated, with overall preference for the PUX Explorer, trade-offs resulting from our design decisions were apparent. P4B suggested that after a while, they started preferring the Matrix tool: “*PUX Matrix was just as good as Explorer after a while*”, even suggesting a preference over the Explorer: “*after a while Matrix becomes easier*”.

Overall, these results suggest that the exploratory aspect of the PUX Explorer tool — characterised by its engaging, non-committal nature and the ability to reveal different insights upon reexamination — may not be necessary for those who are more familiar with an analytic framework. The problems experienced by the participant who forgot the essential lock functionality illustrates the dangers of dynamic exploratory applications in contrast to more structured guided experiences.

7.3. Notation design insights

In addition to our evaluation of the meta-design tools PUX Explorer and PUX Matrix, this study also offers some insights into future opportunities for novel music notation design, with certain notational activities seen as being especially salient in this domain.

For instance, *Modification (CA3)* was ranked as the most problematic and the fourth most beneficial activity. Similarly, *Search (IA1)* was the second most problematic and the fifth most beneficial. Most notably, *Sense-making (IA3)* was identified as the most beneficial activity, yet also the third most problematic (Figures 6 and 7).

The way in which the PUX framework draws the attention of notation designers toward the specific priorities of their own domain, with both negative and positive implications, seems especially useful. We observed similar trends in considering specific experience patterns, for example with *You can change your mind easily (SE2)* appearing on both the problematic and beneficial lists. It is notable that this corresponds to the first-recognised Cognitive Dimension of *viscosity* (Green, 1990), and that we also observed the early tradeoff between CDs of viscosity and *hidden dependencies* (coded as SE1 in PUX).

Overall, we found that activities were highlighted in relation to a variety of usage contexts for music notation, including Interpretation, Construction, and Social activities. This showcases the versatility of the PUX framework and its ability to accommodate a diverse range of notation uses. However, we note that our emphasis on meta-design highlights the designers’ own expectations of what users of their systems need, and that this may not necessarily align with the end-users’ actual experiences. Use of PUX Explorer or PUX Matrix in a co-design setting, where notation meta-designers and notation users might collaborate to identify priorities and design opportunities, is an interesting area for future investigation.

7.4. Problem-solution reformulation

The structure of our study explicitly reflected a co-evolution perspective on the meta-design of notational systems. According to this perspective, analytic tools such as PUX Explorer can assist designers in reformulating their problems as well as finding solutions.

As reported in the results section, half of our participants identified new elements to add to their system after using the tool, and half agreed that the tools were useful in problem reformulation, but these were not the same individuals. Furthermore, some participants did correctly identify new issues yet later reported that they had not done so.

This draws attention to an important consideration in co-evolutionary design work — the phenomenon of design fixation, in which it may be hard to step away from an existing problem, especially if a potential solution has been identified (Crilly, 2015) has occurred.

The participant who was most sceptical about the value of PUX Explorer (P1A) reflected on the chal-

lenge of achieving new creative insight while focusing on the specifics of their design, being “*somewhat overwhelmed with information at a fine-grained level [...] I’m not sure how either tool, in its current form, would be directly useful at the MOMENT that I tend to develop a new notational strategy*”. P1A did explicitly recognise the potential for design fixation: “*but this may be a bias of the fact that I am a practitioner with an already firmly established notational “style” and process*”.

Overall, these observations point to the ways that meta-design problems such as the creation of new music notation systems do share the characteristics of other more routine design domains, bringing potential for innovative solutions through co-evolution of problem and solution spaces, yet also subject to well-known obstacles such as design fixation. As with the design of completely novel programming tools and other kinds of visual language, the design of completely novel music notations is undertaken only by a relatively small number of people in the world. The practices of such meta-designers can be idiosyncratic, with significant divergence between individuals, making it challenging to generalise to every member of such a small population. Nevertheless, our study has found the PUX tools to be accessible as an approach to the meta-design of notational systems, able to be applied by people having no specialist technical expertise in visual language technologies.

8. Conclusions

We have presented the PUX Explorer, complemented by the PUX Matrix and PUX Personas, all of which are meta-design tools intended for use by the designers of new visual languages and other notational systems. We have related these tools to the historical development of critical frameworks for notation design, motivated by a recent general theory of design ideation that has motivated this new approach to meta-design.

As an initial evaluation of the tools, we conducted a controlled study in meta-design. To allow comparison to previous work, we recruited the designers of new music notations, since this notation domain had previously been used to demonstrate and evaluate the original Cognitive Dimensions Questionnaire.

Our study finds that the PUX Explorer is accessible to meta-designers who do not have extensive technical expertise, and who are encountering a critical framework for notation design for the first time. Our controlled comparison between the PUX Explorer and PUX Matrix demonstrates the relative advantages of these approaches, and also provides evidence that the Explorer interaction paradigm is an effective approach to deployment of meta-design tools.

We have also introduced the PUX Persona tool, which is designed for use in longer-term practical design projects beyond controlled laboratory evaluation. PUX Persona provides a principled basis for identifying, weighting, and quantifying the consequences of alternative design decisions. In ongoing work, we are applying these meta-design tools to a wide range of programming language, software engineering, and data visualisation projects in our own organisation and elsewhere.

9. References

- Alexander, C. (2018). *A pattern language: towns, buildings, construction*. Oxford university press.
- BBC News. (2020). *Us election 2020 results*. <https://www.bbc.co.uk/news/election/us2020/results>. (Accessed: 2024-04-17)
- Blackwell, A. F. (2024). Designing user experiences with diagrams: A pattern language. In C. Richards (Ed.), *Elements of diagramming: Theoretical frameworks, design methods, practice domains* (pp. 89–116). Routledge.
- Blackwell, A. F., & Fincher, S. (2010). Pux: patterns of user experience. *Interactions*, 17(2), 27–31.
- Blackwell, A. F., & Green, T. R. (2000). A cognitive dimensions questionnaire optimised for users. In *Proceedings of the twelfth annual meeting of the psychology of programming interest group* (pp. 137–152).
- Bloomberg. (2020). *Us election 2020 results*. <https://www.bloomberg.com/graphics/2020-us-election-results/?embedded-checkout=true>. (Accessed: 2024-04-17)
- Bostock, M. (2024). *D3.js - data-driven documents*. Available at <https://d3js.org/>. (Accessed: 2024-04-

14)

- Clarke, S., & Becker, C. (2003). Using the cognitive dimensions framework to evaluate the usability of a class library. In *Proceedings of the first joint conference of ease ppig (ppig 15)*.
- CNN. (2020). *Us presidential election 2020 results*. <https://edition.cnn.com/election/2020/results/president>. (Accessed: 2024-04-17)
- Crilly, N. (2015). Fixation and creativity in concept development: The attitudes and practices of expert designers. *Design studies*, 38, 54–91.
- Crilly, N. (2021a). The evolution of “co-evolution”(part ii): The biological analogy, different kinds of co-evolution, and proposals for conceptual expansion. *She Ji: The Journal of Design, Economics, and Innovation*, 7(3), 333–355.
- Crilly, N. (2021b). The evolution of “co-evolution”(part i): Problem solving, problem finding, and their interaction in design and other creative practices. *She Ji: The Journal of Design, Economics, and Innovation*, 7(3), 309–332.
- Dearden, A., & Finlay, J. (2006). Pattern languages in hci: A critical review. *Human–computer interaction*, 21(1), 49–102.
- Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., & Mehandjiev, N. (2004). Meta-design: a manifesto for end-user development. *Communications of the ACM*, 47(9), 33–37.
- Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, 443–460.
- Green, T. R. (1990). The cognitive dimension of viscosity: a sticky problem for hci. In *Proceedings of the ifip tc13 third interational conference on human-computer interaction* (pp. 79–86).
- Hadhrawi, M., Blackwell, A. F., & Church, L. (2017). A systematic literature review of cognitive dimensions. In *Ppig* (p. 3).
- Ilevbare, I. M., Probert, D., & Phaal, R. (2013). A review of triz, and its benefits and challenges in practice. *Technovation*, 33(2-3), 30–37.
- Magnello, M. E. (2012). Victorian statistical graphics and the iconography of florence nightingale’s polar area graph. *BSHM Bulletin: Journal of the British Society for the History of Mathematics*, 27(1), 13–37.
- The Economist. (2020). *The us 2020 election results*. <https://www.economist.com/graphic-detail/2020/11/03/the-us-2020-election-results>. (Accessed: 2024-04-17)
- Van Der Linden, D., & Hadar, I. (2018). A systematic literature review of applications of the physics of notations. *IEEE Transactions on Software Engineering*, 45(8), 736–759.

A. PUX Explorer

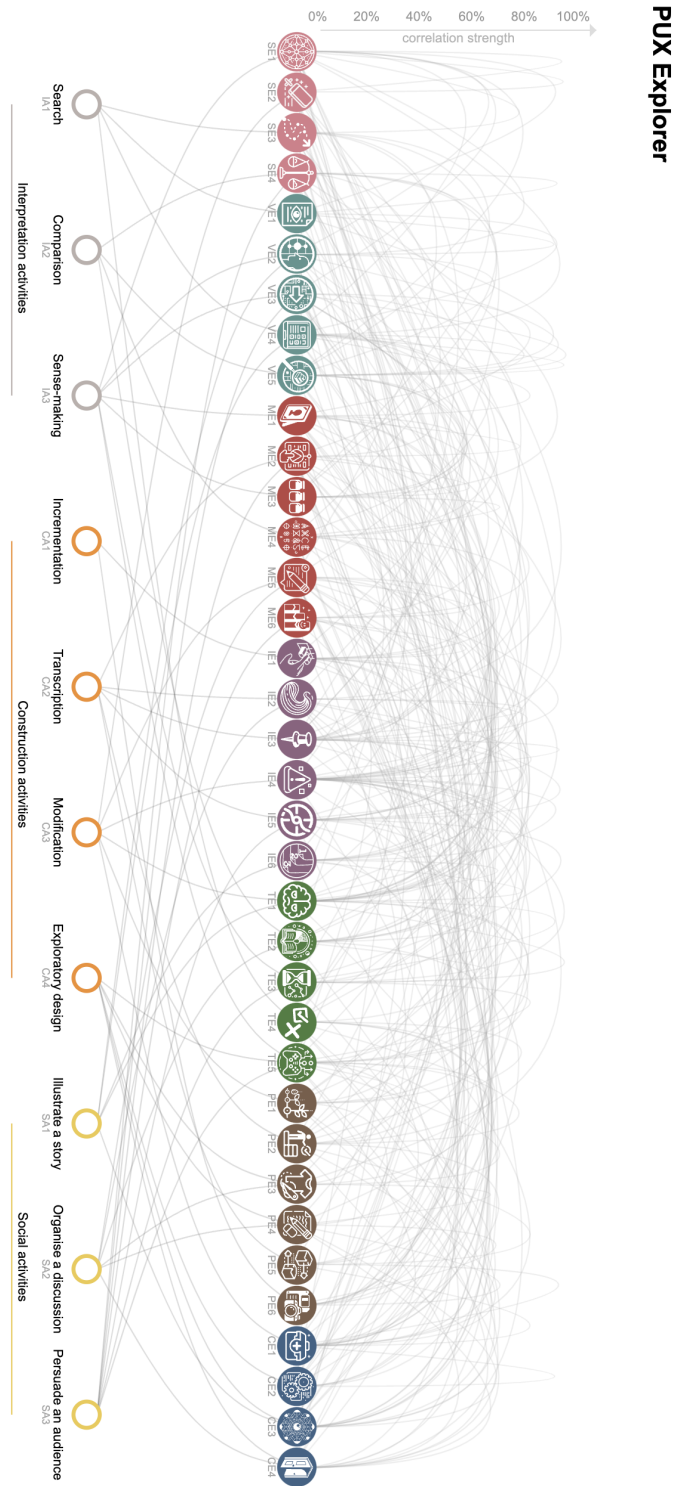
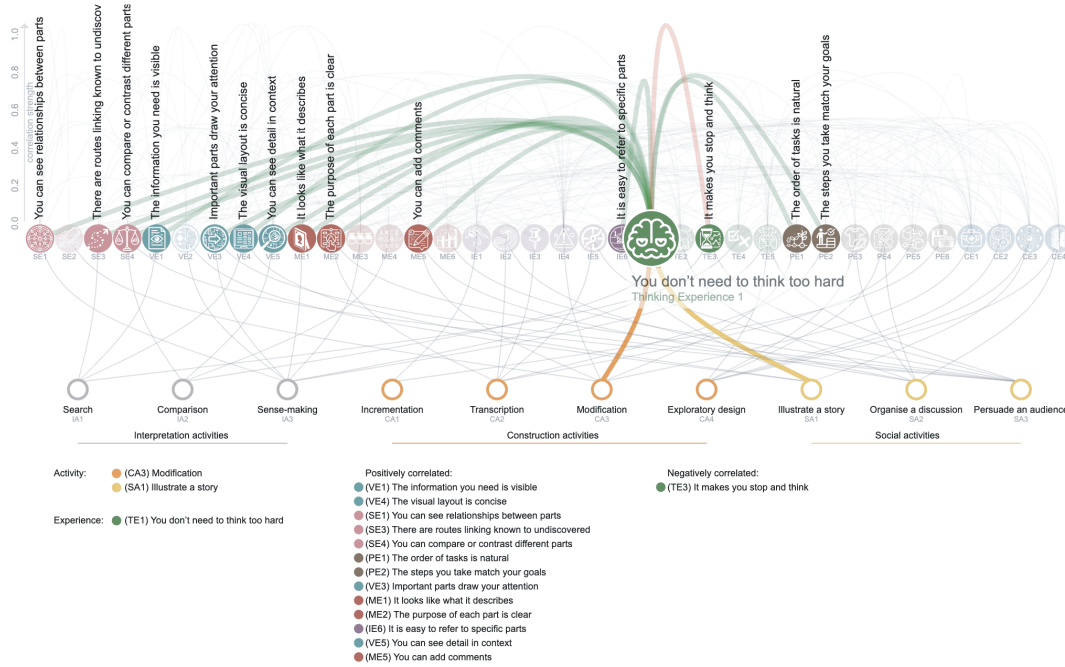


Figure 9 – PUX Explorer on startup with no experiences or activities selected. The top row of circles represents 36 unique experiences and the bottom row represents 10 unique activities described by the PUX framework.

B. PUX Explorer Deconstructed

PUX Thinking Experience 1 : You don't need to think too hard

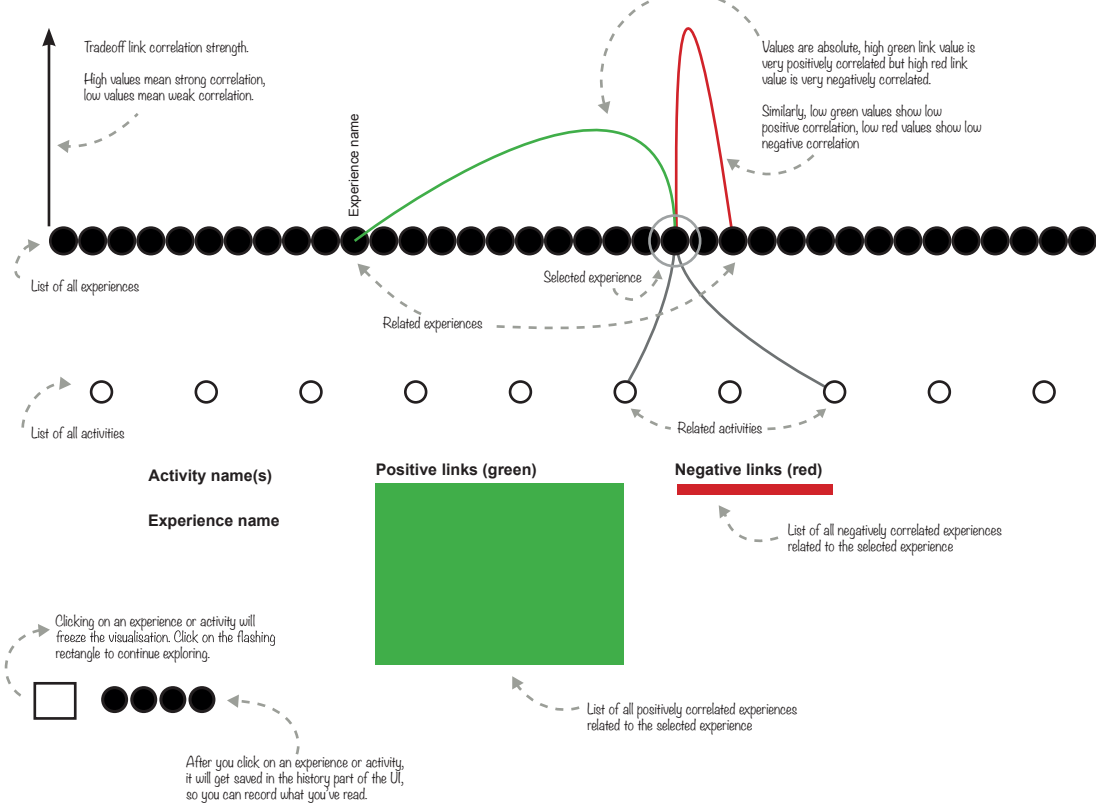


You don't need to think too hard

Summary:
Design minimizes cognitive load, aids focus and memory.

Original Text:
There is a lot of cognitive psychology literature exploring the limitations of the human eyes and brain – which things can be perceived and distinguished, how much we can remember with our eyes closed, or by silently repeating it to ourselves and so on. The number of independent elements in short-term memory is pretty small: around half a dozen words, and a couple of pictures. Design tricks to help the user include ensuring that everything they need is visible (VE1, VE4, SE1, SE3, SE4), that actions correspond to what the user is already planning (PE1, PE2), and that they can recognise things by looking at the diagram, rather than having to remember them (VE3, ME1, ME2). Users also benefit from being able to focus their attention by referring to a particular part (IE6) or region (VE5), or making notes to themselves where they know it will be necessary to return to something in future (ME5). However, note that this usually desirable pattern may be directly contradicted (for some purposes) by TE3!

Experience type: Experience name

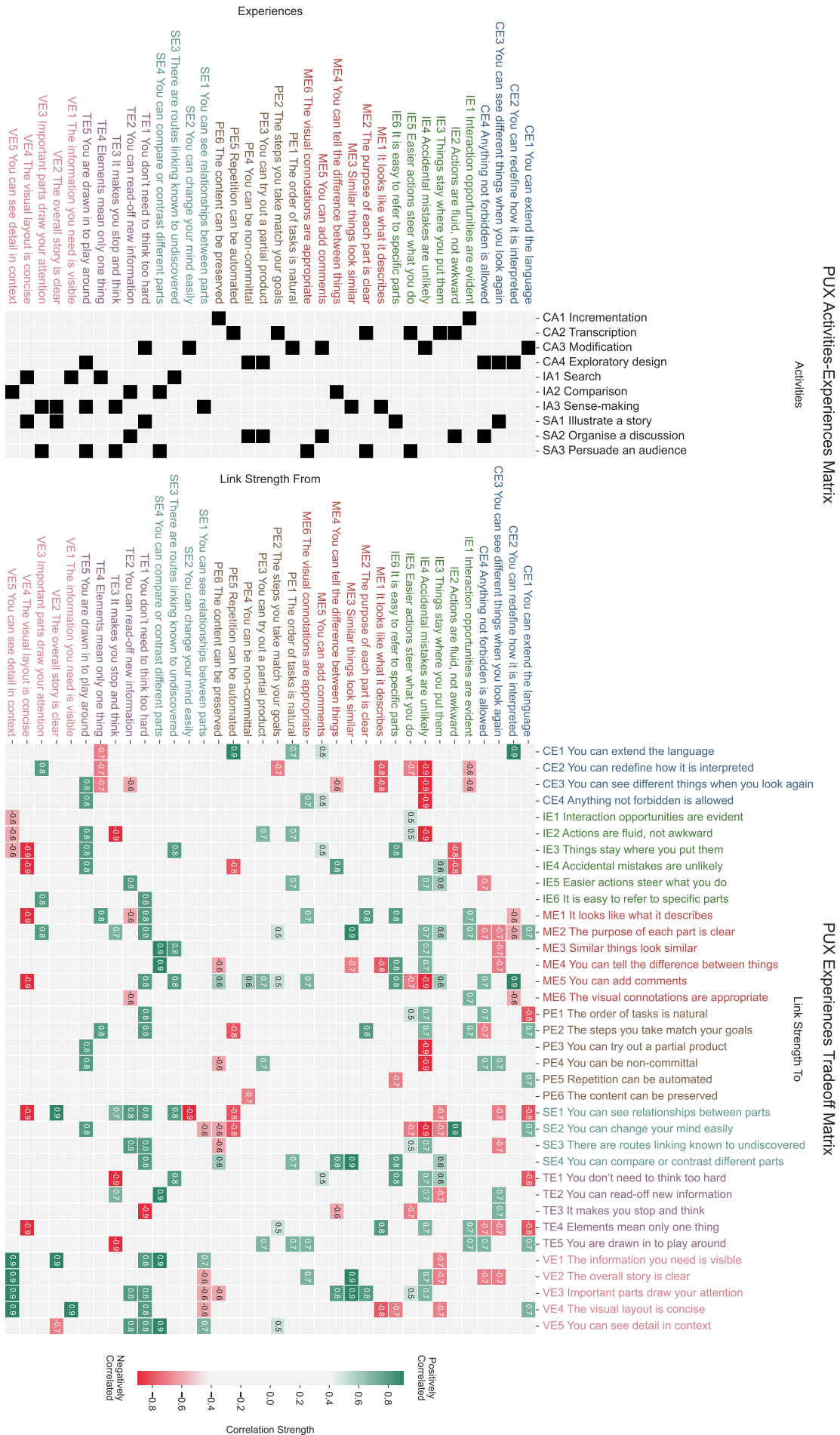


Experience name

Experience summary (brief)

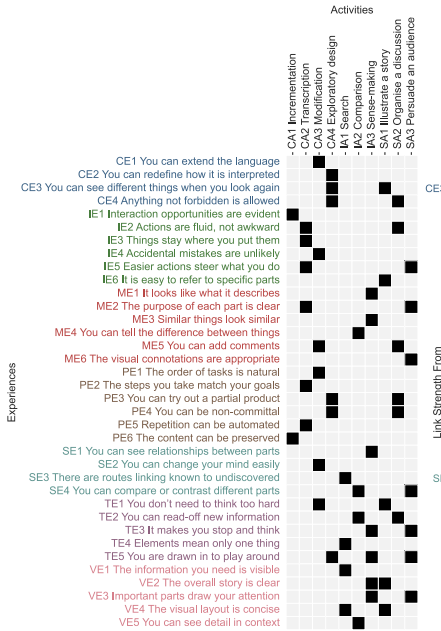
Experience summary (long)

C. PUX Matrix

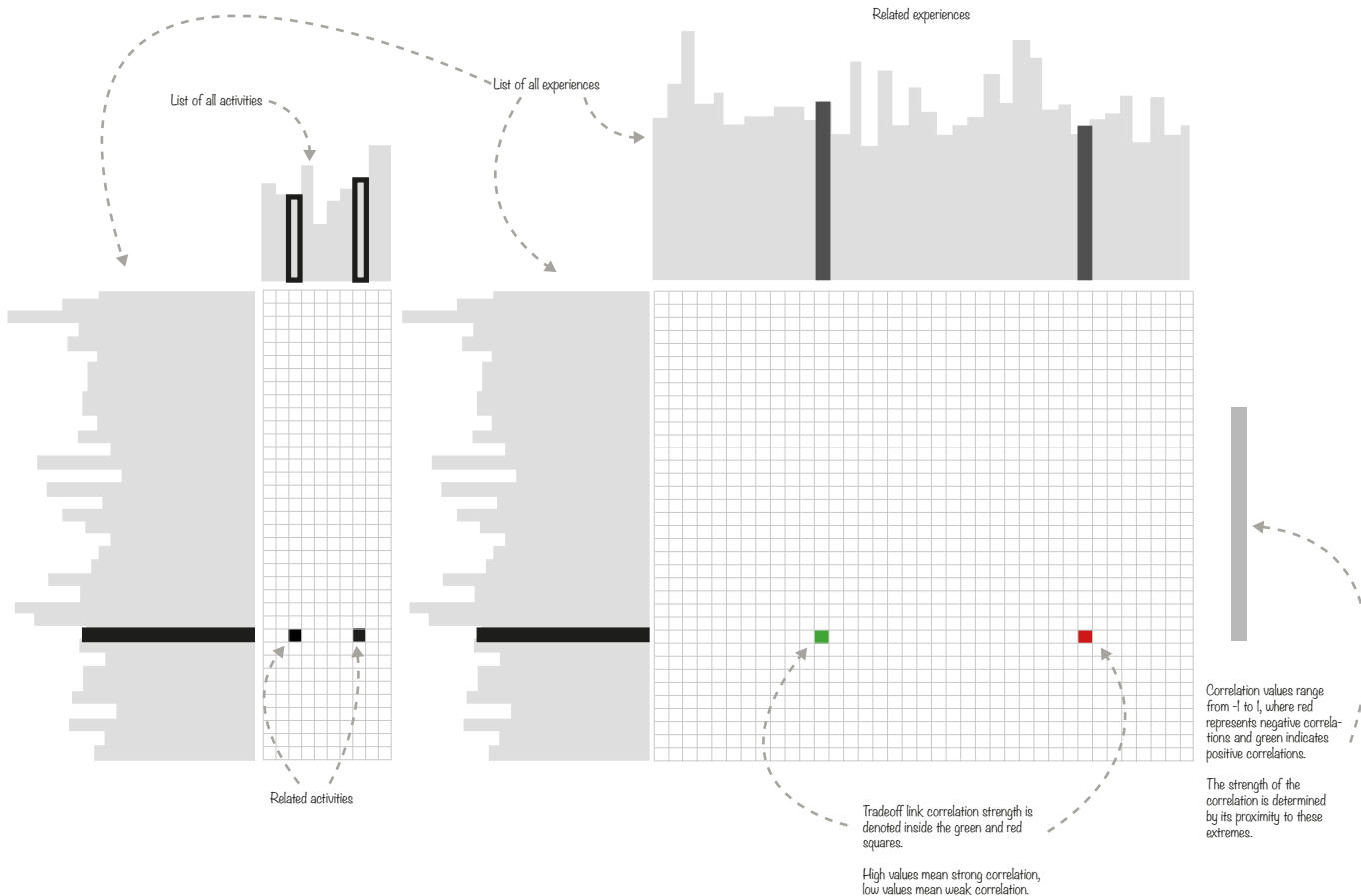


D. PUX Matrix Deconstructed

PUX Activities-Experiences Matrix



PUX Experiences Tradeoff Matrix



E. PUX Personas

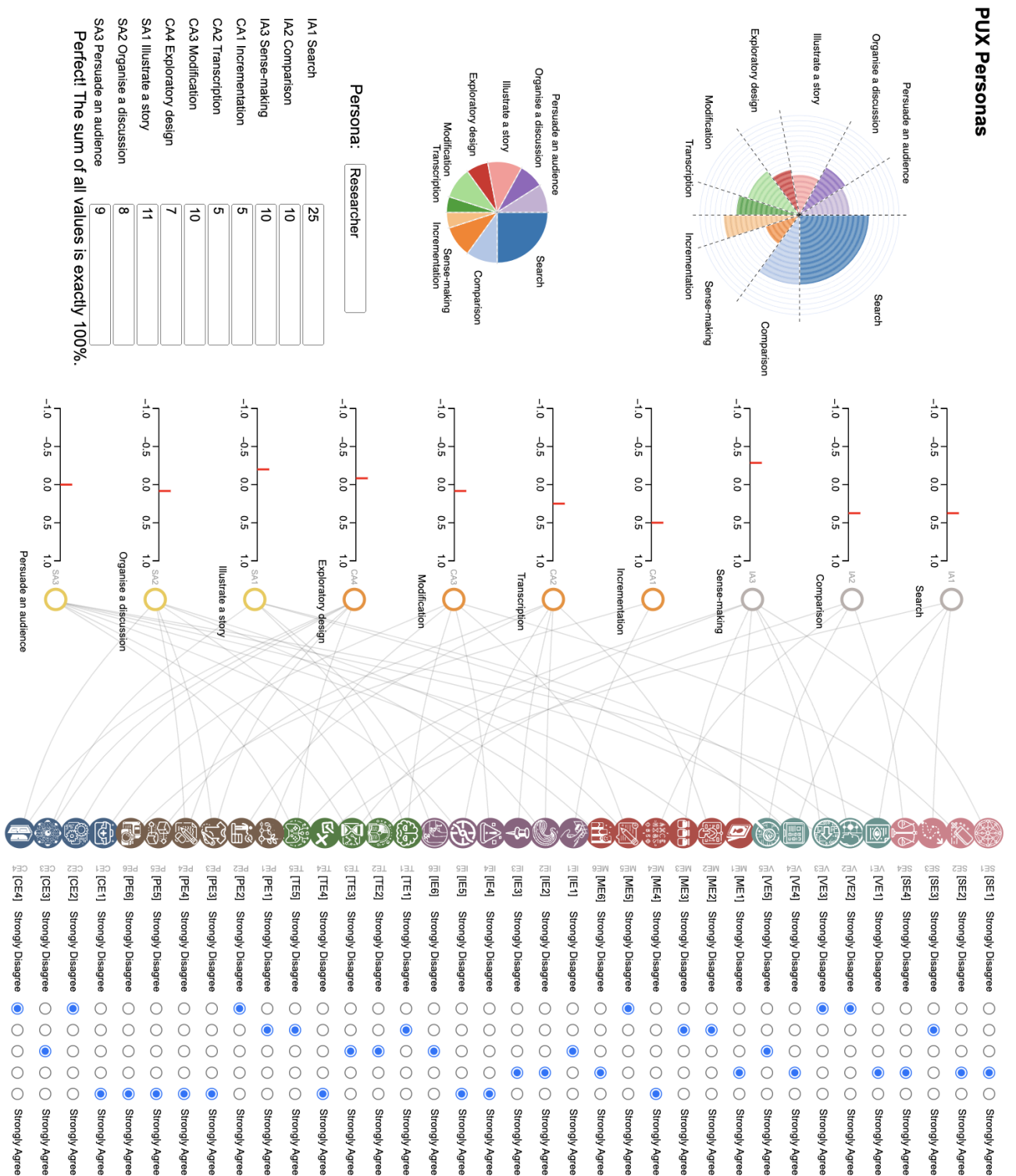


Figure 13 – PUX Personas

Boxer Sunrise Development Update and Demos

Steven Githens
steve@githens.org
diSessa Family Foundation

Demo abstract

This Reflection, Artwork, and Demo will give an update on the Boxer Sunrise Project, which was booted up during the seminal 2018 PPIG workshop hosted at the Art Workers Guild. Examples and demos will be geared towards this year's theme "Human agency in notations".