Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Programming With Ghosts

Integrated Real-Time Versioning for Creative Coding

Author: Maximilian Mayer   (2739590)

1st supervisor: Mauricio Merano Verano
daily supervisor: Mauricio Merano Verano
2nd reader: Fernanda Madeiral

A thesis submitted in fulfilment of the requirements for
the joint UvA-VU Master of Science degree in Computer Science

December 11, 2023

Abstract

Context. In recent years, increasingly accessible tools have elevated *creative coding* from obscurity to mainstream appeal. However, the exploratory nature of creative work introduces numerous challenges that have yet to be solved. Sterman et al. (1) have shown that current version control systems like *Git* are insufficient in supporting exploratory software development. This calls for a new paradigm in the design of such systems.

Goal. This thesis aims to design and test a novel integrated programming environment, embedding exploratory version control as a key component into the workflow of creative coders.

Method. Based on current literature, we formulate a framework for exploratory tool design. Using this framework, we implement a creative coding editor with embedded version control, and evaluate it in a comparative user study. The study adopts a hybrid approach, combining quantitative survey questions with semi-structured interviews.

Results. We are able to demonstrate that small changes to the editing experience can significantly improve the exploratory process, in particular through the use of visual version comparison and increased iteration speed. However, our results indicate limited applicability outside of creative coding, though further investigation is needed.

Conclusions. In summary, we contribute a novel framework for exploratory tool design and a prototypical implementation in the context of version control. Our testing suggests significant value for creative coding, and prompts further investigation to generalize these results.

Contents

CONTENTS

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

Art is an incredibly flexible discipline. Creatives can produce innovative pieces under any condition, with every type of tool, and in unthinkable shapes. Art is about pushing the boundaries and creating something new, something no one has ever done before (3). This urge to experiment with new ways of (self-)expression ultimately led to the invention of *creative coding*, the process of creating art with code (4).

The idea itself is nothing new, and groups like *Compos 68*[1] have published their work since the 1960s. However, programming was a relatively new concept at the time, and a steep learning curve prevented creative coding from becoming a mainstream practice. Since then, programming languages like *Python*[2] have reduced the entry barrier significantly, and today a vibrant ecosystem of specialized tools supports creatives on their quest to explore the power of modern programming.

This has led to a surge in mainstream attention, and successful artists like Refik Anadol can sell their pieces for prices beyond 1 million dollars[3]. As an increasing number of people starts to engage with creative coding, its technological feasibility is put to the test. While specialized languages and frameworks like *Processing*[4] and *P5JS*[5] have improved accessibility, the need for more advanced tooling is growing continuously.

---

[1]`https://monoskop.org/Compos_68`
[2]`https://www.python.org/`
[3]`https://archinect.com/news/article/150309579/refik-anadol-s-casa-batll-nft-sells-for-1-38-million`
[4]`https://processing.org/`
[5]`https://p5js.org/`

1.1 Creative Exploration

A particularly daring challenge is related to the exploratory nature of creative tasks. While the conventional software development cycle is based on well-defined requirements, creatives are manoeuvring through an unknown space of potential solutions and have to identify their goals along the way. Sheil (5) faced similar struggles during the experimental design of artificial intelligence (AI) code and summarized them as *exploratory programming*. In contrast to the linear, requirement-driven process of traditional software engineering, this discipline lacks a pre-determined goal for its outcome. Consequently, exploratory programmers have to develop their goal in parallel with the actual code, exploring different ideas along the way (6).

As Sheil's experience demonstrates, this challenge is not unique to creative coding. Most disciplines related to data exploration, such as data science and AI design, are affected. In fact, software engineering also includes exploratory challenges, e.g. debugging code without knowing the exact problem source (6–8). Nevertheless, these problems are particularly pronounced in creative coding, as the artist's intuition entirely guides the outcome. As a result, creatives often have to experiment with different ideas and create several alternative versions to decide on their future trajectory.

1.2 Version Control for Exploration

The need to create a multitude of different versions has very practical consequences. These versions have to be managed carefully as they guide future design decisions. Doing so manually is likely to end up in a chaotic mess. Software engineers face similar challenges when documenting a project's development history and have created a large ecosystem of version control systems (VCSs). These tools enable efficient recording and management of versions and serve as a repository of past project states.

However, Sterman et al. (1) uncovered that most current VCSs are unsuitable for the exploratory nature of creative coding. These tools are optimized for a linear development cycle and lack crucial features. In particular, few tools enable rapid, parallel version access to compare past ideas. Instead, they track a linear history converging to a single result.

A more suitable approach is to treat past versions as an "idea repository" that can be used as inspiration throughout the development process. This has to be supported by a corresponding user interface, enabling visual version access and an exploration-focused workflow.

1.3   Goal

Based on the challenges outlined above, this thesis studies tool design through the lens of creative coding. The goal is to generalize important principles into a general-purpose framework for exploratory tool design and test the resulting ideas through a novel VCS. The final result is evaluated through a user study to validate the conception and design of both the framework and VCS.

Based on the previous experience of the authors with creative version control (9), the framework design follows two core principles: the effective exploration of solution spaces and meaningful (self-)reflection on the exploratory process. Both factors have been identified as significant contributors to conventional creative processes (1) but are insufficiently supported by many current programming tools (9). Consequently, they serve as the starting point for this thesis.

1.4   Research Questions

To achieve this goal, a set of research questions was formulated to guide this thesis. The first question focuses on building an initial design theory based on the specific properties of creative coding. Sub-questions **RQ1-1** and **RQ1-2** are designed to highlight interface design and reflection as core principles within this theory, both of which have a significant impact on explorative processes according to Mayer and Merino (9).

> **RQ1** How to bring explorative and creative processes to the core of programming tool design compared to existing tool design, and particularly compared to existing VCSs such as Git^1?
>
> **RQ1-1** How can interface design facilitate exploration in programming tools compared to existing solutions such as command line interfaces or text-based editors?
>
> **RQ1-2** How can programming tools enable reflection on and documentation of an artefacts' creation process, beyond providing a simple change log?

The second question is focused on the design of a novel VCS for creative coding based on the framework created for **RQ1**. This is necessary to evaluate the framework's validity and explore the specific challenges of VCS in more detail.

---

[1]`https://git-scm.com/`

RQ2 How can exploratory and reflective processes be integrated into software version control to facilitate creative coding beyond current support from existing versioning tools?

Finally, the third question aims to evaluate the framework and prototype compared to existing tools for creative coding, specifically, the *P5JS Web Editor*. This is done through a user study.

RQ3 How do tools facilitating reflection and exploration improve creative coding, compared to existing tools like the *P5JS Web Editor*?

1.5   Contributions and Structure

In chapter 2, we begin by defining some context for the design of VCSs and challenges related to exploratory and creative coding. Based on this background, we then construct a framework for tool design targeting exploratory coding in chapter 3. Using this framework, chapter 4 derives a specific design for a prototypical VCS. This system is evaluated employing a user study (chapter 5) and discussed in the context of the previously defined framework (chapter 6). Then, chapters 7 and 8 comment on threads to the validity of presented results and alternative approaches from existing literature, respectively. Finally, chapter 9 concludes this thesis with a final summary of results and suggestions for future research.

The main contributions of this thesis are the theoretical framework on tool design for explorative and creative coding, as well as a prototypical implementation in the context of VCSs. A user study provides insights into the feasibility and usability of the prototype. In the discussion, the benefits and problems of the prototype are related to the original framework, and suggestions for future tool design are derived.

2

Background

Creating an exploratory tool design framework necessitates a deep understanding of creative processes. Thus, this chapter characterizes the exploratory code of creative coding and extrapolates its impact on version control systems.

2.1 Exploratory Programming

In chapter 1, we briefly defined *exploratory programming* as a subset of programming tasks without predefined goals. Such tasks require practitioners to consider various outcomes and explore them through code. Kery and Myers (6) summarize this idea in two properties:

Property 1 *"The programmer writes code as a medium to prototype or experiment with different ideas."* (6)

Property 2 *"The programmer is not just attempting to engineer working code to match a specification. The goal is open-ended and evolves through the process of programming."* (6)

These principles point to a distinct difference in mindset between conventional programming and exploratory coding. Traditionally, programmers seek to model a predefined outcome as efficiently as possible. Meanwhile, exploratory programmers navigate an ambiguous solution space without defined boundaries to create innovative solutions. This requires them to leave established solution blueprints behind and overcome continually evolving challenges.

Property 1 expresses the general spirit of exploratory programming. Instead of building a product, programmers build ideas, testing and iterating on them repeatedly. According

to **Property 2**, this helps to build a deeper understanding of the domain and shapes the project goal.

2.1.1 The Need for Exploration

The necessity for exploration is not unique to creative applications. In practice, many common programming tasks are inherently exploratory. Kery and Myers (6) extract the following exploratory problem categories:

1. Playfully Learning Programming

2. Creative Tasks in Art and Music

3. Data Science

4. Certain Exploratory Tasks in Software Engineering (e.g., Algorithm Design)

These problems have specific goals, such as creating a beautiful picture or learning a new programming language. However, there is no universal blueprint for guaranteed success. Instead, programmers must build an intuitive understanding of their domain and develop a solution through trial and error.

Data science is an excellent example to illustrate this principle. Most abstract data sets do not readily provide comprehensive insights, and establishing the correct course of action is challenging. Thus, programmers must test different approaches to effectively leverage the data at hand. Crucially, their intuition guides them in selecting suitable analysis methods, and as they learn more about their task, this intuition improves.

2.1.2 The Exploratory Process

An unconventional new programming process emerges when considering these exploratory challenges in greater detail. Instead of designing software according to existing specifications, practical experiments drive the development. Programmers create hands-on prototypes and iteratively improve their design (6, 10).

Significantly, every new idea can evolve throughout the process and inspire future modifications. The ambiguous nature of exploratory coding means there is no definitive way to evaluate different approaches. Consequently, a previously rejected concept can serve as an opportunity in the face of a new problem, resulting in a non-linear development process guided by the developer's intuition. Within this process, all versions hold the same value, and together, they synthesize a clear picture of the solution space.

In contrast, conventional programming continuously evolves a single artefact, and every new version permanently replaces its predecessor. Past versions only serve as a fallback in case of irreparable corruption of the latest state, manifesting a linear development cycle.

These differences have substantial practical implications. Exploratory programmers frequently reiterate previous versions, appropriating them to a continuously changing environment. While necessary, frequently adapting code on such a fundamental level tends to degrade quality and affect oversight. Hence, exploratory programmers are in dire need of practical solutions to these challenges.

2.1.3 Exploratory Tooling

Naturally, the conventional software development process is not free of iteration. Modern agile methodology encourages fast, iterative production cycles with regular refactoring (11, 12). Concurrently, complex software components can require local use of exploratory patterns despite existing requirement specifications.

As a result, it seems reasonable to assume strong support for iteration and exploration in the existing tool ecosystem. In reality, few meaningful options exist. Most notable are the so-called "notebooks" (e.g., *Jupyter Notebook*[1], interactive environments integrating code and formatted documentation into a single file. They are used to document design intentions alongside the experimental code to improve oversight and long-term usability. However, the complexity of managing alternative versions remains.

The academic literature proposes additional solutions, for instance, programming patterns to streamline exploration (13) and specific programming languages for increased iteration speeds (10). However, in summary, tool support for exploratory programming remains sparse.

The lack of specialized tools is especially problematic for version management, as the non-linear exploratory process can lead to a somewhat chaotic version history. Currently, tools like *Git* are too linear in their design, preventing effective use of past versions in an exploratory sense. Instead, many creative coders pivot to manual approaches, such as recording alternative versions in separate files and commenting out code (1), risking code quality and maintainability.

Few attempts have been made to address this issue, most of them focusing on so-called "micro-versioning" or "local versioning". While conventional VCSs usually track changes per file, these tools enable versioning on a semantic level. For instance, *Variolite* allows

---

[1]https://jupyter.org/

users to select arbitrary blocks of code and makes saved versions available inside the editor directly (2). Mikami et al. (14) explored an alternative approach, automatically grouping individual changes into semantically meaningful versions.

However, while promising, most of these approaches are not production-ready and limit workflow flexibility. E.g., *Variolite* is only available for the discontinued *Atom* editor[1]. Consequently, most exploratory programmers continue to struggle with manual processes.

2.2 Creative Coding

Creative coding is extremely versatile. It enables the creation of pictures, videos, interactive graphics, music, and even physical experiences, e.g., when creating a large-scale programmable light installation. Practitioners can even perform real-time shows in the form of *live coding* (4, 15, 16).

All of these applications are highly exploratory. Creatives experiment with various approaches to achieve the desired result, and there are no objective measures to assess quality. The relationship between creative processes and exploration is well-known and has been studied extensively before (1, 17). However, creative coding has unique challenges beyond exploration, as discussed below.

2.2.1 Code and Creativity

While regarded as a technical discipline, programming heavily relies on creativity (1, 18, 19). Complex problems, such as designing powerful algorithms, frequently necessitate innovative and creative solutions to achieve the desired outcome.

However, in most cases, creativity is part of the process, not the result. In fact, most programs are created with predictability in mind (20, 21). Unexpected, creative outputs are undesirable and a sign of bad software design. The opposite is true for creative coding. Here, the developer strives for unique and striking results beyond the spectator's imagination. Under these conditions, even technically wrong code can match the posed expectations.

Consequently, the shape of creativity transforms with its purpose. While operating as a solution engine in the hands of a conventional programmer, it becomes the very purpose of programming for creative practitioners.

In practice, creative coding primarily manifests through its exploratory nature. Practitioners face the challenges of a non-linear workflow and ambiguous solution space. The

---

[1] https://github.blog/2022-06-08-sunsetting-atom/

growing suite of specialized tools for creative coding may improve accessibility; however, the varying types of created media are largely incompatible with existing tool chains (e.g., using *Git* to store large binary files such as videos) and introduce additional overhead.

Finally, creating artistic artefacts is fundamentally different from solving technical programming tasks. The adapted methodology can influence requirements for the development process, e.g., a programmer generating images needs a way to compare results visually rather than a conventional debugger.

2.2.2   Tools for Creative Coding

There are several design philosophies regarding creative coding tools, likely due to the flexibility and complexity of program code. These approaches range from conventional code editors to advanced node-based graph editors with UI-based workflows. Each method is optimized for different use cases and addresses a specific user group.

The "traditional" approach is most comparable to conventional programming. Tools like *Processing* are often based on general-purpose programming languages like *Java*[1] and provide powerful interfaces to create artistic artefacts. While highly versatile, the initial complexity of learning to program reduces accessibility. However, a code-first approach enables integration with existing tools and directly benefits from programming research, making it the ideal contender for this thesis.

Other approaches emphasize usability and wrap standard program code in simplified user interfaces. *vvvv*[2] uses a node-based interface to visually compose complex operations. This approach is equally powerful yet easier to learn. On the other hand, users are locked into the existing ecosystem with limited flexibility and configuration options.

Some hybrid approaches combine these concepts to enable a visual coding experience. An example is *Stamper* (22), a node-based code editor that introduces a visually structured editing experience.

However, most of these tools only address the creation of creative code. Peripheral tools such as VCSs have to cope with the variety of these existing tools and the complexity of exploration. As a result, the support tool ecosystem for creative coding is currently lacking.

---

[1] https://www.java.com/
[2] https://visualprogramming.net/

2.3   Version Control Systems

Producing a new artefact is commonly associated with creating versions. In most cases, these versions represent consecutive changes as the artefact develops, building up to a final result (1, 23). This applies to most exploratory and creative domains, just as to more technical fields like programming, engineering, and many handicrafts.

While recording and managing these versions is not required to create an artefact, practitioners can benefit significantly from doing so. Most importantly, saved versions serve as a fail-safe in case the created artefact is damaged or lost.

Furthermore, past versions document the artefact's creation in detail and can serve as the foundation for collaboration. The latter is crucial for digital applications such as programming, 3D modelling, and image editing. In these cases, team members can synchronize modifications efficiently by sharing a detailed editing history (24–26).

2.3.1   Version Management

Version management is concerned with the creation and exploitation of a version history. In particular, it is about recording meaningful sets of changes as versions and making them accessible for future use.

Solving these tasks is certainly challenging, in particular, as every domain has unique versioning requirements. Factors such as the version medium (e.g., physical sketches, digital files), workflow requirements (e.g., collaborative work, tool and space availability), and process limitations (e.g., exploratory process, linear process) have a substantial impact on any potential solution (1, 14, 27).

According to Sterman et al. (1), most creative applications still require manual versioning through sketches, photographs, handwritten notes, or, in the case of creative coding, copying code into new files and commenting code out. However, these procedures are error-prone and can result in significant chaos.

Tool-supported methods have been proposed for several use cases, often in the form of so-called *version control systems* (VCS). These tools provide means of recording and managing versions automatically, often digitally. However, only a few solutions are widely accepted, and existing ones are mostly limited to a single domain (e.g., (28–30)).

Programming poses as a standout example, with version control systems for program code rising in popularity since the introduction of the original *Source Code Control System* (SCCS) in 1975 (31). Since then, similar tools have become industry-standard, with version

management platforms like $GitHub$[1] reaching more than 100 million users[2]. The core design goal for these tools, are collaboration and failure-recovery, with a clear emphasis on linear development processes (9, 32, 33).

A key issue with these solutions is their unintuitive user interface. Users must overcome a steep learning curve to benefit from the most powerful features, and most users never reach their full potential (34, 35). Additionally, these VCSs only define the content and format of versions (e.g., source code files). Recording new versions remains a user responsibility (e.g., creating a commit in Git), which means they might forget to save important progress.

More automated approaches can be found in other digital domains, such as image editing or 3D modelling. In these cases, editing tools and version control are frequently heavily intertwined, such that each editing step is recorded as a new version automatically (29, 36, 37). However, the vast version histories generated in these cases can overwhelm the user, making it hard to identify specific versions.

On the other hand, physical versions present entirely new challenges. While meaningful versions are generated automatically (e.g., sketches, the actual physical object), managing them requires manual work (e.g., filing papers, moving objects). Tool support can only provide some fundamental guidance, and reconstruction of previous versions is labour-intensive (1, 38).

In summary, version management remains highly challenging, with different requirements for each domain, and a "one-size-fits-all" solution does not exist. Consequently, the following section extracts the most important requirements for version management in creative applications, specifically creative coding, to provide a baseline understanding of the requirements for this thesis.

2.3.2 Creative Version Control

A good way to start understanding version control in creative coding is by looking at versioning in the creative process in general. Sterman et al. (1) do so by means of several interviews, talking to industry professionals in different creative and/or exploratory roles, including artists, (creative) programmers, artisans, researchers, and others. Based on these interviews, they identify four ways in which effective version management can aid the creative process:

---

[1]`https://github.com/`
[2]`https://github.blog/2023-01-25-100-million-developers-and-counting/`

Palette The idea that old versions can serve as a palette of ideas and inspiration that can be reused repeatedly, even beyond the boundaries of a single project. Instead of using versions to recover from mistakes, they become the practitioner's personal repository of tools for new projects.

Freedom Similar to how programmers use version control, it can help creatives to recover from mistakes without losing too much progress. However, more importantly, the knowledge that one can recover from such mistakes gives creatives the freedom to explore their ideas without fear of destroying their current progress. This is immensely important, as exploration is at the core of creative tasks. A fear of exploring unusual or strange ideas is thus a serious limitation for creatives.

Fidelity A key challenge discussed previously is the format of a recorded version. It is not always trivial to decide what should be part of a new version, but intuitively, including as much detail as possible may seem logical. This is an excellent idea in many cases, as it allows one to retract past steps accurately, reducing ambiguity. However, Sterman et al. (1) argue that a reduction in version fidelity can lead to unexpected re-interpretation of old work, further aiding exploration. This supports spontaneity, variation, and adaptation, which can be essential for the creative process. As such, it should be possible for a practitioner to choose a suitable level of fidelity instead of being locked into a predefined system.

Timescale As mentioned before, version histories can be used to document an artefact and allow understanding of its creation after the fact. Sterman et al. (1) believe this should be emphasized further, making sure versions are kept available across multiple projects to allow for active reflection on the personal process. This way, creatives can rework their ideas and improve over time more efficiently.

These four themes are extracted from the description of the interviewees' personal workflows. Together, and they characterize the real-world requirements of creative practitioners. Interestingly, these interviews include several programmers from exploratory and creative domains. While most of them are aware of existing version control systems such as *Git*, some explicitly chose different versioning processes in their everyday workflows, and described these systems as unsuitable. One participant felt *Git* was designed to follow a single direction in order to make progress fast, while they were aiming at diversity instead. This points to a mismatch between traditional program code VCSs and the creative need for exploration.

This mismatch was further investigated in a preceding literature study on the topic (9). Based on the four concepts described above, a number of creative (coding) versioning tools from different domains were analyzed to understand how these tools address creative needs and in which way existing tools for programmers fall short. The most frequently addressed factors across all of these tools were **Freedom** and collaboration. Consequently, the technical foundations across these tools tend to be similar, although differences are implied by varying media types.

However, when considering the intended use of versions, major differences were identified: While programming tools often support a linear development process (e.g., *Git*), many creative tools aim at rapid, parallel version access and efficient reuse of old versions (1, 38, 39), both of which are linked to a **Palette** type of versioning. While there have been some attempts to achieve similar functionality for program code, they are rather limited and sparse (2, 14, 40). This imbalance indicates a potential future path for new creative coding versioning tools.

Furthermore, it should be noted that both **Fidelity** and **Timescale** were severely underrepresented in the results. This is true for creative approaches but even more so for programming-oriented methods. In case of **Fidelity**, this may be related to digital nature of most tools. Digital resources can easily be duplicated and stored, leading to a tendency of data hoarding (41). Additionally, it is rather difficult to define *low-fidelity* in media such as program code.

Management of versions across long timescales, on the other hand, seems to be an afterthought for most practitioners, especially in the programming domain, and many professionals openly acknowledge their lack of retrospection (1). Most tools approaching this issue are concerned with progress documentation through tutorial generation (42) and the efficient reuse of old components (40, 43). Rarely, tools were designed to spark reflection or inspiration based on old versions.

Based on these insights, VCSs for program code already provide sufficient peace of mind for creatives when saving versions regularly. However, their focus on linear development processes with no meaningful perspective on the reuse of old versions conflicts with the exploratory process driving many creatives. As such, the following two core principles should guide the design of version control for creative coding and exploratory programming in general:

Exploration of Solution Space

Firstly, to successfully support creatives on their journey through the uncertainty of their domain, versioning tools should allow for rapid versioning with little to no overhead. Instead of complicated command line interfaces (CLI) with multiple stages for version creation, it should be a one-click solution that works instantly.

Furthermore, versions should be placed front and center as a primary resource to work with. Version access should be rapid and visual to allow for instant comparison and reflection. Instead of creating a sequential history, each version should be equally important while maintaining the original structure of the project. And finally, versioning may not obfuscate the creative process through merging conflicts and unexpected states (e.g., detached head in *Git*), but enable fast iteration and quick experimentation.

It should be noted that these suggestions are supposed to represent the general ideals of improving exploration through the use of versions. The exact ways to achieve this may be specific to the use case at hand. However, generally, these suggestions aim to enable **Palette** versioning as described by Sterman et al. (1) without affecting the ability to recover past project state at any time.

Reflection

Secondly, reflection on personal achievements and processes should be considered key to creative success. Exploring a vast solution space in the search for the best results is challenging and heavily relies on the practitioner's intuition and experience (1, 44). Thus, it is crucial to understand the personal strengths and weaknesses, and how to improve. This is impossible without reflecting on previous work, and there is no better way to document this work than through a meaningful version of history. And even beyond that, such a history can serve as inspiration and invoke old ideas in a new context. Currently, these practices are somewhat neglected by professional programmers, and tools such as *Git* do not provide intuitive interfaces to do so (1). Much can still be done to improve in this area and fully embrace **Timescale** versioning.

2.3.3 The Interface Problem

Finally, we would like to point out that both of these recommendations, at their core, are interface problems. As mentioned previously, the technical foundations of different versioning systems may differ according to the used media types, but are conceptually similar. At the same time, modern version control for code is extremely powerful and

theoretically already supports many of the features necessary to enable effective exploration and reflection. Unfortunately, these tools are equipped with rather technical interfaces designed for efficient collaborative code development. These interfaces often do not meet the requirements of creative practitioners and harm the benefits these tools could bring to them. As such, much of this thesis is concerned with interface design.

3

Concept

After identifying the core challenges that must be addressed for creative version control in subsection 2.3.2, the following chapter develops a framework for tool design based on these challenges. The first step in doing so is to understand the role of confidence in exploratory processes, and how it can improve the final results.

3.1 The Confidence Crisis

Fast iterations are key for successful exploration. They are required to test out ideas quickly and identify good solutions in an otherwise underexplored solution space (1, 45–47). As such, increased iteration speed can be extremely beneficial for exploratory programmers. However, in many cases, fast iterations come at the cost of code quality. This is particularly true in an environment that requires many changes for the same piece of code. In these cases, programmers might be compelled to save time by skipping documentation, writing badly structured code, or neglecting version management under the premise of creating "a better version" with every change (6).

Some programmers deem versioning tools such as *Git* to be overly complicated and slowing down development. As such, they might opt for more rudimentary versioning concept, such as commenting out code or duplicating files (1, 2). This can result in even more chaos, and a lack of oversight. Either way, these practices hurt the exploratory process as soon as access to previous versions is required.

Dealing with these issues on an everyday basis, programmers might develop an aversion to change, lacking the confidence to experiment with vast or complex changes for the sake of preserving their progress (1). This can severely harm their ability to identify unique

and innovative solutions. This also explains the results of subsection 2.3.2, indicating that **Freedom** is the most popular benefit of versioning tools (9).

Traditional VCSs for program code have no satisfactory solution to this problem. Instead, they are designed to manage successive changes in a linear development flow, and lack interfaces that would enable the rapid and parallel version access required for exploratory processes. This results in a *crisis of confidence.*

3.2   Exploring with a Map

To gain a deeper understanding of this crisis, we can explore the following analogy: An explorer, wandering through unknown territories in search of treasure, wants to avoid getting lost. A good tool aiding his navigation would be a map. Unfortunately, no map is available yet. So instead of wandering about blindly, the explorer creates a map along his way, updating it whenever he uncovers something new.

If we now put a developer into the shoes of our explorer, we must imagine the unexplored solution space as our territory. The map is represented by the version history captured throughout the development, and allows us to understand our current path. Creating such a map can be a tedious process, and capturing a detailed representation of our surroundings requires a significant time investment. However, if we spend more time creating our map, we have less time to explore the solutions space, and we might miss valuable results.

There are several ways to face this dilemma. First and foremost, it is important to maintain a delicate balance between the exploration of new ideas and version management. Spending too much time on version management will compromise idea exploration. Meanwhile, having no version history at all can derail the development process in case of errors and other unexpected events. Unfortunately, finding the right balance is a reoccurring challenge with each new project. As such, it is difficult to design a tool around this balance alone.

If we return to our analogy instead, we can consider the following alternative: Instead of drawing the map himself, the explorer could hire an assistant. Under these circumstances, he could focus completely on his exploration, while still receiving a map.

Translating this idea to exploratory programming, the assistant comes in the form of a VCS. This version control assistant (VCA) records the complete development process in the background, while developers only mark the most important versions for improved accessibility in the future. As such, many manual versioning task can be eliminated, e.g., setting up project repositories, configuring included and excluded files, or saving versions

manually (35). Concurrently, such a system would guarantee safe storage of all progress at all times.

Based on this idea, we proposes the hypothesis that gathering version information optimistically in the background can benefit exploratory programming through an increase in confidence, while decreasing version management overhead compared to traditional VCSs such as *Git*. Instead of settling for a sparse set of manually creative versions, a hybrid version history combines in-depth documentation of the development process with rapid access to the most important versions selected by the developer. This process can be optimized even further by suggesting important versions automatically based on optional configuration information or heuristic analysis of the version history (14).

However, several limitations have to be considered. Firstly, we must acknowledge, the digital nature of such an assistant. As a consequence, it can only record digital artefacts, and could miss out on important version information generated beyond this limited space of resources. In fact, it is most likely such an assistant would focus on source code exclusively. Other artefacts such as architectural sketches, (handwritten) notes, and developer discussions are often too unstructured to be recorded automatically without user intervention (48).

Furthermore, the scope versions must be defined in advance. Tools such as *Git* track changes on a file-level, and provide line-by-line comparisons between versions (33). While this may be sufficient for some developers, others might prefer tracking changes for individual functions, or even characters in the source code. This could be configured to the user's needs, however, requires additional consideration.

Finally, tracking versions alone will not suffice in creating a helpful assistant. As mentioned in subsection 2.3.3, the most important challenge concerns the user interface, as it fundamentally defines how versions will be used. If the interface is not beneficial to an exploratory process, the tool's value is diminished.

3.3 The Ghost Framework

The concepts presented in the previous section are tailored specifically to version control systems. We generalize them under the term *Intuitive Assistive Automation* (IAA) in the following section, as we believe these ideas can benefit tool design for creative and exploratory domains beyond version control as well. These findings are then used to present the *Ghost Framework*, formulating core values and practices required to embed

IAA into programming tools. This framework is used in chapter 4 to implement a VCS prototype for creative coding.

3.3.1 An In-Depth Look: Intuitive Assistive Automation

The term *Intuitive Assistive Automation* immediately provides a useful hierarchy to understand the core principles behind the *Ghost Framework*:

Intuitiveness Intuitive use is widely understood as the unconscious application of prior knowledge to the interaction with a new system or product, increasing the effectiveness of said interaction (49, 50). Accordingly, *intuitiveness* describes a design quality facilitating such knowledge application. Techniques to foster intuitiveness include the use of well-known interface components and metaphor-based design (50–53). Tools following these principles can improve user understanding and, in turn, elevate their effectiveness and value to the user.

Assistance Numerous theories on effective tool design exist, and in certain scenarios, especially potent tools may necessitate the adoption of a predefined process to unlock their full potential (54). However, the exploratory process inherent in creative coding is built on the practitioner's intuition and experience (55–57), resulting in an extremely specialized process. As such, the *Ghost Framework* advocates assistive tool design that accommodates to pre-existing processes and adapts dynamically. Workflows disruptions should be avoided, and tools should seamlessly support the user with proactive suggestions.

Automation Finally, the *Ghost Framework* employs automation as a key enabler for an intuitive workflow actively assisting the user. Rather than offering a static toolbox for manual operations, the framework advocates for a data-driven, automated process informed by user's behaviour. Analyzing the user's workflow in real-time enables anticipation of future operations and helps to streamline the tool's use. Furthermore, non-destructive operations can be optimistically executed and precede the user's eventual needs. All of this may be done in the background, providing active assistance only when required. Context cues can be used to suggest operations when necessary.

Based on these ideas, the *Ghost Framework* strives for a new generation of powerful tools solving problems in real-time without disrupting existing workflows. This is particularly

important for exploratory tasks, but can benefit programmers independently of their domain. Tools should step up into an active role, and act as autonomous assistants, rather than passive toolboxes.

The name "*Ghost Framework*" captures the essence of IAA's assistive, background-oriented nature. Much like a ghost, these tools are operating discretely, out of sight, and interact with the user through subtle, context-sensitive prompts. They are the metaphorical "ghost in the machine", always seeking for opportunities to improve while remaining in the background.

3.3.2 IAA in Practice

Naturally, some existing tools already embody the principles of *Intuitive Assistive Automation*. A prime example is Microsoft's *IntelliSense*[1]. This advanced code completion tool continuously analyses the user's code, comparing it to existing documentation and other code segments. Based on the results, *IntelliSense* offers real-time completions while typing, including token names (e.g., variables or class names) and meta-information such as parameter definitions for functions. Additionally, the tool can display relevant documentation in-context, as demonstrated in Figure 3.1.

Figure 3.1: An example of *IntelliSense* in *Visual Studio Code*. The code completion suggests several class names directly under the edited line and presents a brief explanation on the right.

---

[1] https://code.visualstudio.com/docs/editor/intellisense

Thanks to its recent integration with *GitHub Copilot*[1], *IntelliSense* now also supports advanced AI-powered suggestions beyond simple token completion. *Copilot* can generate functional code snippets based on existing code and comments, formulate documentation, and provide additional functionality like code tests.

Crucially, these interactions are seamlessly embedded into the development workflow. Completions are instantaneously provided as the user types, without obstructing the coding process itself. Suggestions remain optional, and rejecting them requires no additional effort. Temporary documentation appears beneath the current code line, and disappears once editing stops. *IntelliSense* even learns from user behaviour, adapting suggestions to the user's coding style and the existing code base.

All of this makes *IntelliSense* an incredibly versatile tool, enriching the coding experience for many developers. Furthermore, its integration in the widely used *Visual Studio Code* editor[2] requires minimal setup, making it easily accessible to anyone.

Beyond programming, tools like Spotify's *Discover Weekly* feature[3] and password managers like *KeePassXC*[4] present similar IAA principles. Spotify curates weekly playlists for each user based on their listening habits, while password managers, often equipped with special browser plugins, record login data automatically. Once a login was saved, it can be automated in the future (see Figure 3.2).
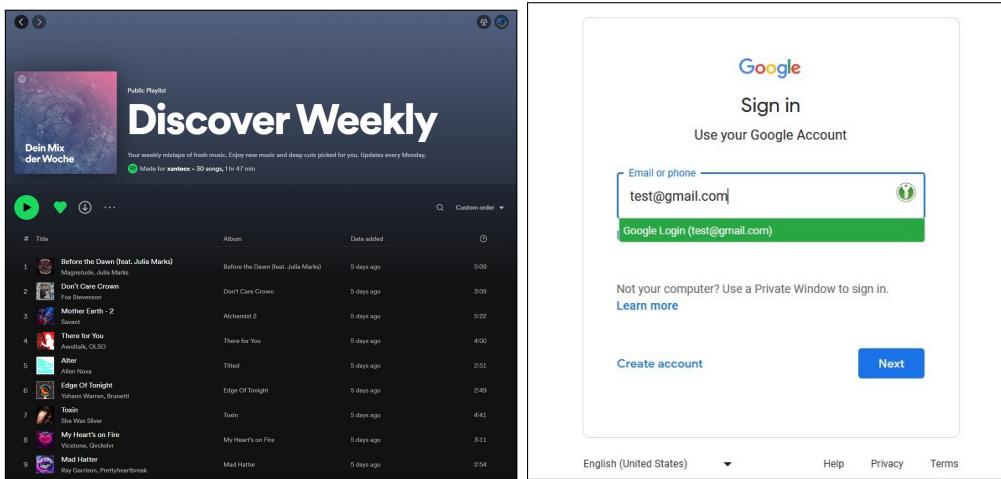
Figure 3.2: An example of Spotify's *Discover Weekly* playlist (left) and the auto-fill option of the password manager *KeePassXC* (right).

---

[1] https://github.com/features/copilot

[2] https://code.visualstudio.com/

[3] https://medium.com/the-sound-of-ai/spotifys-discover-weekly-explained-breaking-from-y our-music-bubble-or-maybe-not-b506da144123

[4] https://keepassxc.org/

Both tools operate unobtrusively in the background. If contextually relevant, they assist users without interrupting their routine. This proactive background support is characteristic for the IAA design philosophy.

### 3.3.3 Trusting in Ghosts

While the *Ghost Framework* is primarily designed for exploratory coding, it should be generic enough to accommodate various tool types. Consequently, concepts specific to VCSs are not incorporated. This also includes the two core principles of creative version control identified in subsection 2.3.2, exploration and reflection. Although important, they are too specific, and instead addressed when using the *Ghost Framework* to design a VCS prototype in chapter 4.

However, throughout this thesis, *confidence* emerged as a key factor for effective exploration. The observation of the *confidence crisis* detailed in section 3.1 is, in fact, a primary motivator for the design of this framework. As a result, we believe confidence deserves a central role within the *Ghost Framework*.

In practice, tools have to earn the user's trust, empowering developers to engage in experimentation without concerns or anxiety. Such trust can be cultivated through the following three principles:

Data Integrity First and foremost, maintaining data integrity is crucial. This is especially important for tools modifying code directly. For instance, Microsoft's *IntelliSense* must ensure that the insertion of code suggestions has no unexpected side effects. Preserving the user's data is exceedingly important.

Control Second, users must always retain control. A tool may prepare and propose operations, but explicit user consent should be mandatory and rejecting a suggestion the preferred default option. Additionally, the user should have the ability to overwrite tool operations and customize results. *IntelliSense*, for instance, integrates with the conventional undo/redo system in *Visual Studio Code*.

Transparency Finally, transparency is key to obtain the user's trust. This includes insights into collected data and predictable operations. If users have privacy concerns, they might hesitate to provide required information. Concurrently, it must be clear how operations work, and what their impact will be. This is a common issue with many tools using artificial intelligence, such as *GitHub Copilot*. These tools have no predictable outcome, and require manual supervision by the user (58).

Adhering to these principles allows users to understand and trust the tools. They can be sure of their code's safety and consistently understand the tool's impact on their project. Above all, they can take agency, and remain in full control of both their results and processes. That is an important step in building a synergistic relationship between user and tool to maximize their combined potential.

3.3.4 Building Tools with the Ghost Framework

After previously conceptualizing the *Ghost Framework*, we now propose guidelines on how to use it in practice. For these guidelines, the traditional separation of front- and backend in software design serves as a logical boundary. The frontend is concerned with intuitive, context-aware user interaction, while the backend handles background activities like data collection and analysis, as well as tool operations.

3.3.4.1 Backend Design: Data-Driven Tool Support

In the *Ghost Framework*, the backend is at a tool's heart. Whenever a developer is editing code, the backend analyses every step he takes, patiently waiting for an opportunity to help. Depending on the tool's requirements, the backend can do many different things. However, the most important tasks are discussed below.

Data Collection While not always required, many tools require certain data to work properly. For VCSs, this may be data representing code changes, while a tool like *IntelliSense* requires an index of existing variables, classes, and functions. In many cases, this data is constantly evolving. Then, the backend has to keep track of changes and update its knowledge of the project.

Data Processing Sometimes, raw data is insufficient. Additional processing of captured information may be required to unlock a tool's full potential. For instance, Spotify has to analyse a user's listening habits to learn their preferences. Similarly, a VCS will likely transform captured data into a format that enables efficient versioning. Processing data in advance can profoundly benefit tool performance, especially when expensive computation is required.

Tool Operations Most importantly, the backend performs any tool operations requested by the user. This can include database operations, on-demand data processing, advanced code modifications and more. The exact operations required are defined

by the tool's feature set, and all communication with the end user is performed through the frontend.

Context Analysis A fundamental idea of the *Ghost Framework* is to make tools autonomous *assistants*. Instead of passively waiting for the user's command, tools proactively provide support. However, to do so in an unobstructive and helpful way, the tool has to understand the workflow context. Consequently, the backend has to monitor the editing process closely, searching for contextual triggers before suggesting any operations.

Additionally, context knowledge can be used to personalize the user's experience by identifying patterns in their workflow. If they use the tool in certain ways, suggestions can be tailored to be more targeted and efficient.

Naturally, the requirements for each of these categories can differ significantly for different tools. Core features will likely translate to a set of tool operations, with data collection and processing supporting efficient execution of these operations. The context analysis is most susceptible to domain-specific requirements, and necessitates a deep understanding of the tool's usage patterns. However, this analysis plays a crucial role in elevating the tool to an active assistant. As a result, significant attention should be directed to its design.

The principles of *Intuitive Assistive Automation* have to be considered throughout the design, with an emphasis on assistive automation. Concurrently, data integrity and transparency has to be a primary concern when collecting, processing, and using data. At all times, the user should retain control over their data and work.

In practice, the design specifics depend on the unique tool requirements. Nonetheless, some general guidelines can be derived from literature on software design. Privacy-centric software design is particularly interesting when building software that instils confidence. For instance, Hoepman (59) advocates for design strategies that inherently enforce privacy and suggests concrete design patterns such as *access control* and *data breach notifications*. Similarly, Zieglmeier and Pretschner (60) present a framework for design-based transparency, with independent components for data monitoring and result verification to ensure accurate insights.

Literature on assistive systems with autonomous components is relatively sparse in the domain of programming tool design. However, robotics research can serve as inspiration. Meng and Lee (61) suggests a number of useful design principles, including continuous self-improvement through adaptive learning and robust operation across diverse condition by means of action variability.

Finally, operational accuracy and data integrity can be achieved through rigorous software testing. This allows the verification of quality attributes, and builds confidence in the tool's functionality (62, 63).

3.3.4.2   Frontend Design: User Interaction

The frontend is responsible for all user interactions, and communicates with the developer through a user interface. Fundamentally, this interface has to be designed around the IAA principles of *intuitiveness* and *assistance* to enable effortless, predictable workflows. Moreover, it has to inspire confidence through extensive user control and transparency. In this way, the interface directly adheres to the core values of the *Ghost Framework*, and should be compatible with exploratory processes.

However, these terms do not inspire specific design recommendations. Yet, in contrast to backend design, there is a multitude of universally applicable research on user interface design that can provide concrete guidance. Specifically, Ben Shneiderman has provided many insights into user interface design, focusing on *creativity support tools* in particular (64–66). He presented the idea of "*[d]esign with low thresholds, high ceilings, and wide walls*" (66), which serves as the foundation of interface design in the *Ghost Framework*.

Shneiderman used this metaphorical description to suggest three desirable attributes in tools designed for creativity:

Low Entry Barrier Any tool supporting creativity should be easy to use for novices. This way, the tool remains approachable, and can be adopted quickly. Shneiderman suggests this can be achieved by means of *multilayer interface design* (67, 68). Instead of building an interface presenting all features at once, the user starts with a limited selection of tool operations, enabling other features when needed. This is a common technique, and a well-known example are Google's advanced search options, that can be toggled by the user.

Shneiderman's *layer 1* is thereby representative of the most limited feature set a user will encounter when using the application (67). The main design objective in this layer should be simplicity, allowing novices to learn the tool's basic concepts and features. For the purpose of the *Ghost Framework*, we suggest an additional, optional layer 0, or *hidden layer*.

While some tools require a basic set of features at all times (e.g., a code editor), other tools are based on infrequent interactions instead (e.g., a version control system). In these cases, the *hidden layer* will conceal the tool completely, leaving the developer's

workflow unperturbed. Contextual suggestions serve as primary interaction basis, superseding manual user prompts. When accepting a suggestion, the tool may present additional layers required to execute the desired operation, and hide them after completion.

If designed correctly, the user should never have to open any layers manually. This way, the tool's visual complexity is reduced to a minimum. However, the tool's full feature set should be accessible manually as well, in case the context analysis fails to detect the user's needs automatically. This can be achieved through (context) menus or shortcuts.

Tool Depth When speaking about "*[...] high ceilings [...]*", Shneiderman indicates that a tool should be approachable without sacrificing complexity or depth. This makes sense, a tool sacrificing its core functionality by catering to novices can serve as an educational example, but is unlikely to aid advanced users.

Multilayer interface design can help to isolate complexity into separate interface layers, but the design of intuitive, yet powerful interfaces is highly domain specific. The literature on user interface design suggests some general patterns, such as design metaphors (52, 53, 69), but more specific recommendations can only be made in the tool's specific context.

Tool Scope Finally, Shneiderman's "*[...] wide walls*" refer to the tool's scope. He suggests that good tools should cover as much functionality as possible, and interface efficiently with other tools when required. This reduces friction and frustration generated by frequent tool switching, file conversions, and other chores required when working with different tools (66).

Again, *multilayer interface design* can be used to manage complexity introduced by the interaction of different features in the same tool. However, instead of building large, monolithic tools, the *Ghost Framework* advocates for a platform approach.

In practice, many practitioners already use well-established tools as part of their basic workflow (e.g., code editors or IDEs). Today, many of these tools support extensions, enabling real-time interaction between tools in a unified environment[1]. This allows developers to create their own, custom tool suite from a vivid environment of plugins and extensions. Tool designers can capitalize on these circumstances by emphasizing

---
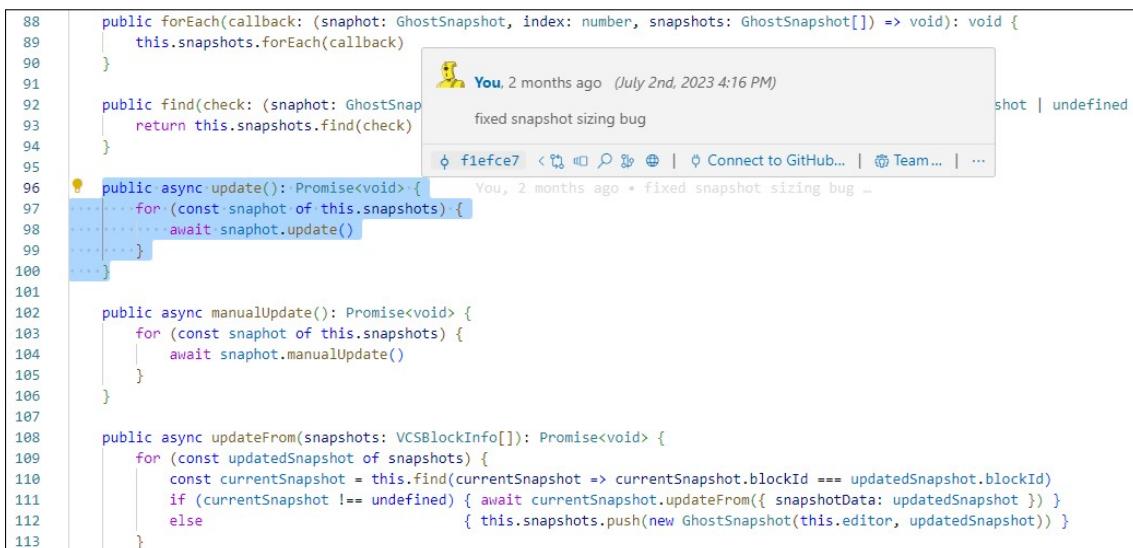
[1]An example is *Visual Studio Code*, with a large library of extensions: `https://marketplace.visual studio.com/vscode`.

key tool features, while basic functionality like editing code or file management are provided through the environment. Often, editor environments even provide direct interfaces to interact with the user's code efficiently.

In the context of *multilayer interface design*, the basic tool, for instance a code editor like *Visual Studio Code*, serves as *layer 1*, while extending tools start with a *hidden layer* on top of the editor.

While Shneiderman's theory guides the structure of the frontend, the specific design of individual user interface components remains underspecified. While generalized recommendations are lacking, anecdotal references can be used to gather some ideas. The previously mentioned *IntelliSense* is one such example, focusing on inline interactions by integrating suggestions directly into the code editor of *Visual Studio Code*. Similarly, documentation is provided directly underneath the active line, right in the user's zone of attention. Tools interacting with code might want to use similar concepts, as they prove extremely efficient in practice.

Figure 3.3: An example of *GitLens* in *Visual Studio Code*. The extension is based on *CodeLens* and embeds interactive information into the editor context.

Microsoft's *CodeLens*[1] is another example of contextual interaction found in *Visual Studio Code*. Figure 3.3 demonstrates how *CodeLens* embeds information directly into the editor without compromising the editor experience itself. Clicking on such an annotation reveals additional information. This is a common feature in many IDEs and can be adapted

---

[1] https://code.visualstudio.com/blogs/2017/02/12/code-lens-roundup

for different use cases, such as displaying versioning information (e.g. *GitLens*[1] or code metrics (e.g., *CodeMetrics*[2]).

However, interfaces do not always require innovative new shapes. Sometimes, a tool is powerful enough to justify manual interactions. A prime example is *IntelliJ*'s search bar shown in Figure 3.4. By gathering information about the project automatically, this search bar can provide extremely efficient access to various resources, and can speed up the user's process significantly. In turn, it has to be triggered by use of a shortcut. This is a fair trade-off, as it is hard to predict the user's intent to search. Nonetheless, designers have to be aware that introducing shortcuts and hidden menus reduces intuitiveness.
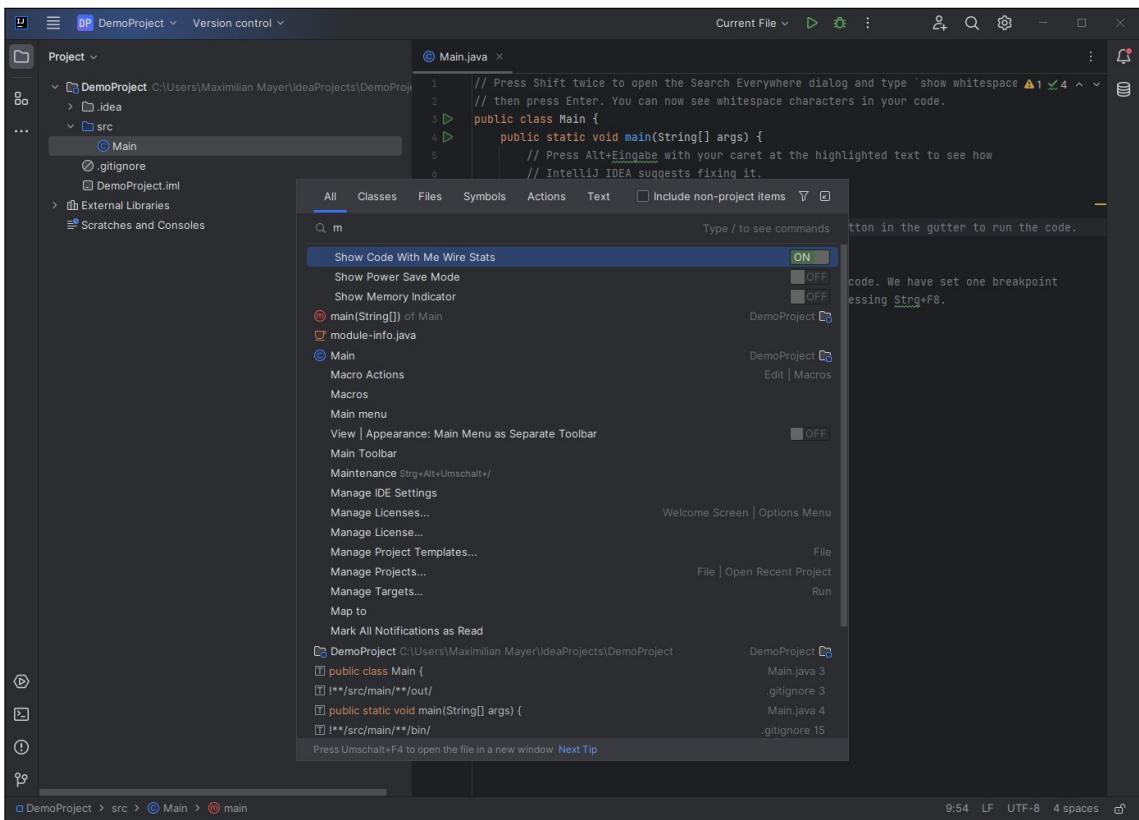
**Figure 3.4:** An example of *IntelliJ*'s powerful search bar.

3.3.4.3 Integration of Front- and Backend

Integrating the front- and backend effectively requires a coordinated design strategy throughout the tools' development cycle. Both components have to interact continuously, with the

---

[1] https://www.gitkraken.com/gitlens

[2] https://marketplace.visualstudio.com/items?itemName=kisstkondoros.vscode-codemetrics

backend serving as a processing engine for the frontend. Consequently, the frontend serves as a blueprint for the backend design.

Observing the *Ghost Framework* from the perspective of a conventional client-server architecture, the client (frontend) requires a set of features provided by a server (backend). As a result, the backend's interface should cater to the frontend requirements. This principle extends to other system attributes, including performance and stability. The frontend, being crucial to the user experience, must operate swiftly and reliably. In case of backend issues, the frontend should fail gracefully, offering users a clear understanding of any complications.

Originally, the *Ghost Framework* was designed for local deployment on the user's machine. However, remote backend server deployment is a viable option. In this case, some backend responsibilities shift to the frontend, particularly in data collection, as the backend no longer has access to the user's data. Otherwise, the fundamental framework principles remain intact.

Finally, it is advised to adopt suitable protocols to facilitate communication between frontend and backend. Such standardization streamlines communication and improves maintainability. A fitting example for programming tools is Microsoft's *Language Server Protocol* (LSP)[1]. Designed for programming-specific client-server interaction, LSP is widely deployed in tools like *Visual Studio Code* and *Eclipse*[2]. It offers robust features for programming tool design, including document synchronization and workspace management.

3.3.5 Limitations

The *Ghost Framework* is designed for intuitive tools with a high degree of automation. While it provides numerous guidelines to achieve this goal, several limitations remain. The following section will cover the most important challenges and pitfalls that should be avoided when applying the framework in practice.

Performance Overhead While proactive background activity takes a central role in the framework's design, it can be resource-intensive. Complex tasks, such as the context analysis, could degrade the tool's performance and affect responsiveness of the user's system. As a result, performance optimizations are crucial within the framework. Dynamic resource allocation should be employed to maintain system responsiveness and background tasks limited to a minimum where possible.

---

[1]https://microsoft.github.io/language-server-protocol/
[2]https://eclipseide.org/

Data Privacy While transparency and user data control are emphasized, continuous data collection always raises privacy concerns. To foster trust, the tool has to transparently communicate which data is collected, and for which purpose. Temporary data processing might be necessary to extract contextual triggers and suggestions, however, this data should be deleted afterwards. Recording data permanently should be reduced to a minimum, and always follow a clear purpose.

False Positives/Negatives Operating partly autonomous, the tool is susceptible to misinterpretation of user intent or context. This can result in faulty or lacking suggestions, disrupting the user's workflow and harming the tool's value.

As a result, the context analysis required thorough testing, and should employ reliable heuristics to predict intended operations. This requires a deep insight into the tool's user base.

Scalability Directly related to the performance overhead discussed earlier, tool scalability is at risk. Large projects might overload the system, resulting in a degraded user experience. This could be amplified if a tool includes collaborative features.

While performance optimizations can improve these issues, tool designers should consider a desirable project scope early on and take design decisions accordingly. Thorough testing at the desired scale is recommended.

Compatibility Finally, the *Ghost Framework* is designed to incorporate with existing tool platforms and enable interaction with other tools. This introduces crucial dependencies and can result in compatibility issues.

Tool designers should minimize external dependencies as much as possible. However, existing dependencies are likely to increase maintenance overhead, requiring frequent updates to accommodate the changing tool environment.

This list is by no means exhaustive. However, it should provide some insights into problems of the *Ghost Framework*. More practical issues will be revealed and discussed during the application of the framework in chapter 4.

4

Design

To test the *Ghost Framework* developed in chapter 3, this thesis proposes a prototypical version control system for creative coding. The following sections describe the tool's conceptualization and implementation, focusing on practical applications of the framework. A detailed evaluation of the results is presented in chapter 5.

4.1 The Ghost Editor

The manual, often complex versioning process of many current VCSs is a reoccurring struggle for exploratory and creative programmers. The resulting workflow disruptions can harm the creative process, and lead to frustration. The *Ghost Framework* promises to resolve this issue by automating the versioning process and minimizing manual user interaction. Analogue to the metaphorical map presented in section 3.2, the tool automatically records a version history, and offers easy access to the saved information.

A key concern of the *Ghost Framework* is the intuitive and seamless integration with the user's workflow. As such, an integrated solution is proposed that embeds version control in the conventional process of writing code. Furthermore, the development process is augmented with supportive features for creative coding to provide a comparable experience to existing creative coding editors such as the *Processing Editor*[1] and the *P5JS Web Editor*[2] (see Figure 4.1). The final result is called *Ghost Editor*, referencing the framework's name.

---

[1]https://processing.org/
[2]https://editor.p5js.org/
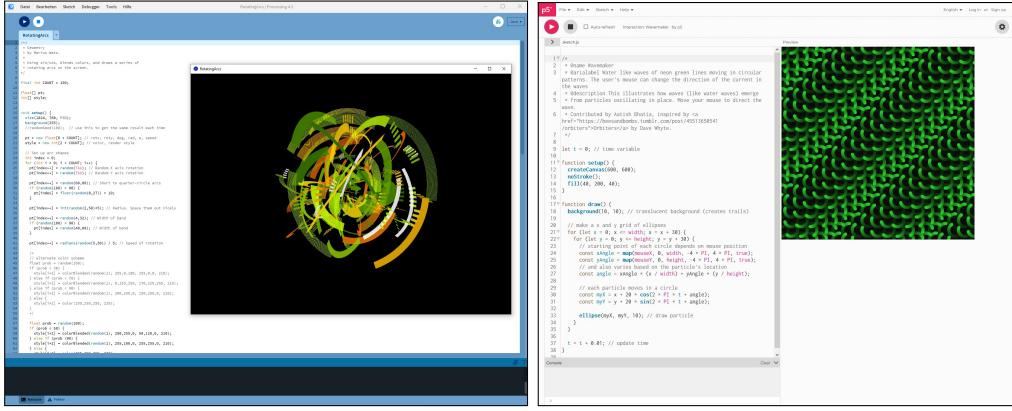
Figure 4.1: A screenshot of the *Processing Editor* (left) and *P5JS Web Editor* (right).

4.1.1 Core Concepts
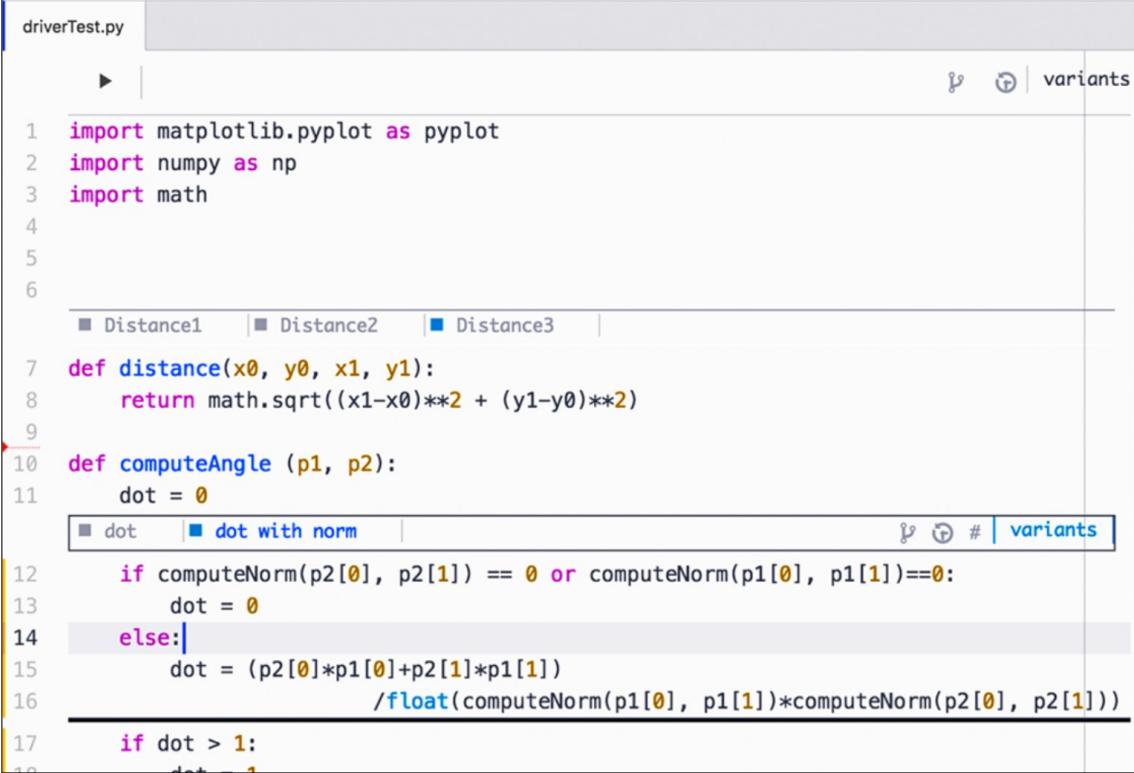
The versioning process itself is built upon real-time change tracking. Instead of generating versions on a manual user prompt (e.g., *Git* or *Subversion*[1]), the *Ghost Editor* creates new versions for every code modification automatically. This includes individual character insertion or deletion. Versions are stored on a per-line basis, compared to conventional file-based approaches, and every line has its own version history.

During the editing process, each line is treated as an individual unit, and changes are decomposed into individual line modifications (e.g., when inserting several lines at once, each inserted line receives its own version). Using the versions' timestamps, these individual line histories can be composed into a full file history. By selecting a suitable version for each line, the full code for any previous state can be reconstructed.

This process is part of the *Ghost Framework*'s autonomous backend component, proactively gathering information to provide assistive version control. While its design enables unsupervised real-time versioning with maximal resolution, it generates a multitude of meaningless or faulty versions, e.g., unfinished modifications when typing a variable name. The user interface has to reduce this complexity and offer intuitive user interactions. This is achieved through *multilayer interface design*, building on top of *Variolite*'s UI concept (2). As shown in Figure 4.2, *Variolite* conceptualizes an in-line UI for local versioning. This way, version can be accessed right from within the editor, and are available immediately.

The *Ghost Editor* expands on this idea in several ways. Most importantly, *Variolite* requires manual version creation and management. After selecting a block of code, the user can create new versions, and switch between them. Our method captures the entire

---

[1]https://subversion.apache.org/

Figure 4.2: A screenshot of *Variolite*, as presented by Kery et al. (2). It displays two nested inline versioning boxes with several saved versions each.

version history for each line proactively, and can combine these histories for any selection of lines. As a result, the full version history is available immediately after selecting a block of code, and users no longer have to fear losing any progress. However, visualizing the entire history of changes at once might be overwhelming. Consequently, the interface was decomposed into 5 individual layers:

Layer 0: Hidden Layer Initially, the code editor does not show any versioning-related interface components. Once the user wants to access the interface, they can select any code block, and use the right-click context menu to create a so-called "snapshot" (see Figure 4.3).

Layer 1: Code Highlight After creating a snapshot, the editor highlights the selected code permanently with a subtle box, as shown in Figure 4.4. This provides a continuous context cue for the user, and clearly indicates affected lines.

Layer 2: Version Interface Clicking into the code highlight for a snapshot unfolds the versioning interface. As a result, the user is always aware of available versioning
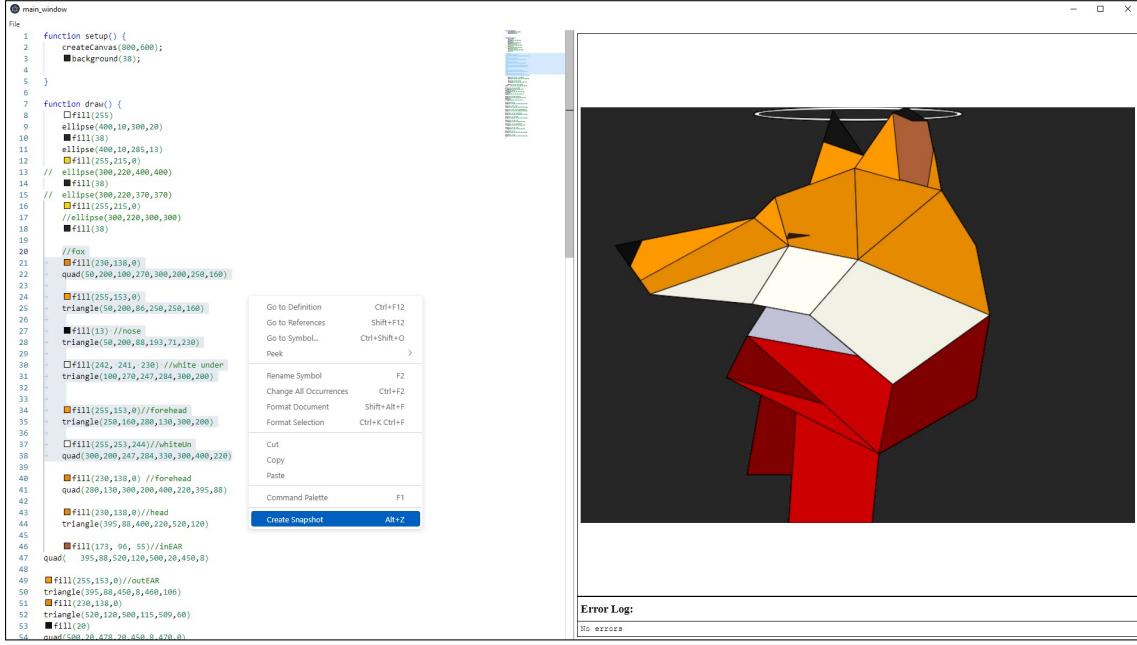
Figure 4.3: A screenshot of the *Ghost Editor*, showing the context menu used to create a snapshot.
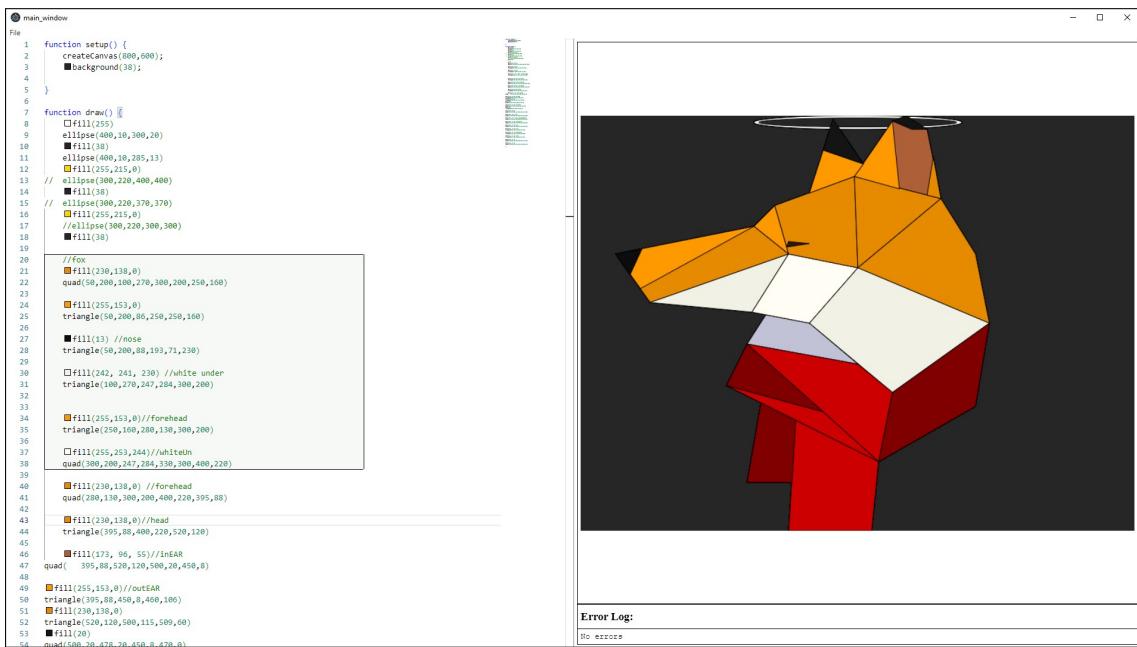
Figure 4.4: A screenshot of the *Ghost Editor*, showing the code highlight for a snapshot.
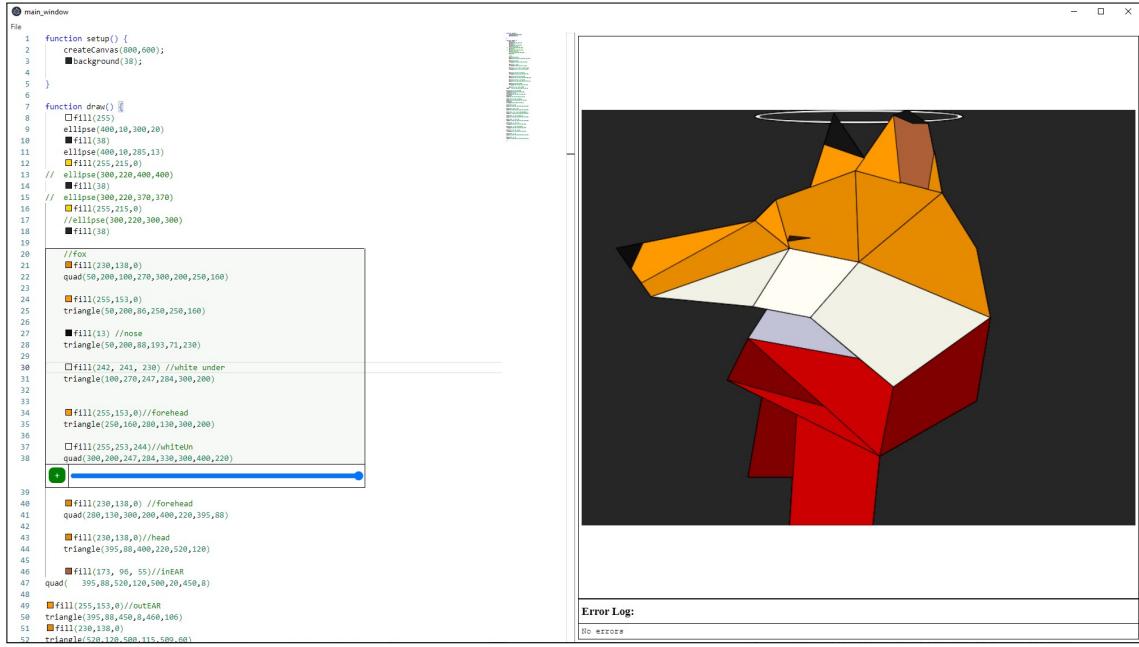
Figure 4.5: A screenshot of the *Ghost Editor*, showing the version interface. The green button will save the current state, while the blue slider allows the user to scrub through all past versions of the selected code block.

functionality when editing this code block. Figure 4.5 highlights the two operations accessible from this interface: The green "+" button saves the current version for faster accessibility, while the blue slider, a version timeline, allows scrubbing through every previous version of the code block.

Layer 3: Version View The version view opens next to the editor whenever the user saves a new version, and displays all manually recorded versions of the current code block. Versions are presented with a visual preview of their result and an AI-generated name and description (see Figure 4.6). The generated metadata transforms saving a version into a one-click operation and can be adjusted later. This enables creatives to save versions faster, encouraging experimentation with new ideas.

Layer 4: Version Editor Finally, a user can select versions in the version view to edit them in isolation. This can be helpful for quickly testing an idea in another version, without modifying the entire project. The goal is to enable faster iteration safely.

Modified versions can also be copied into the main editor, or duplicated for further experimentation. The interface is explained in more detail in Figure 4.7. It is worth
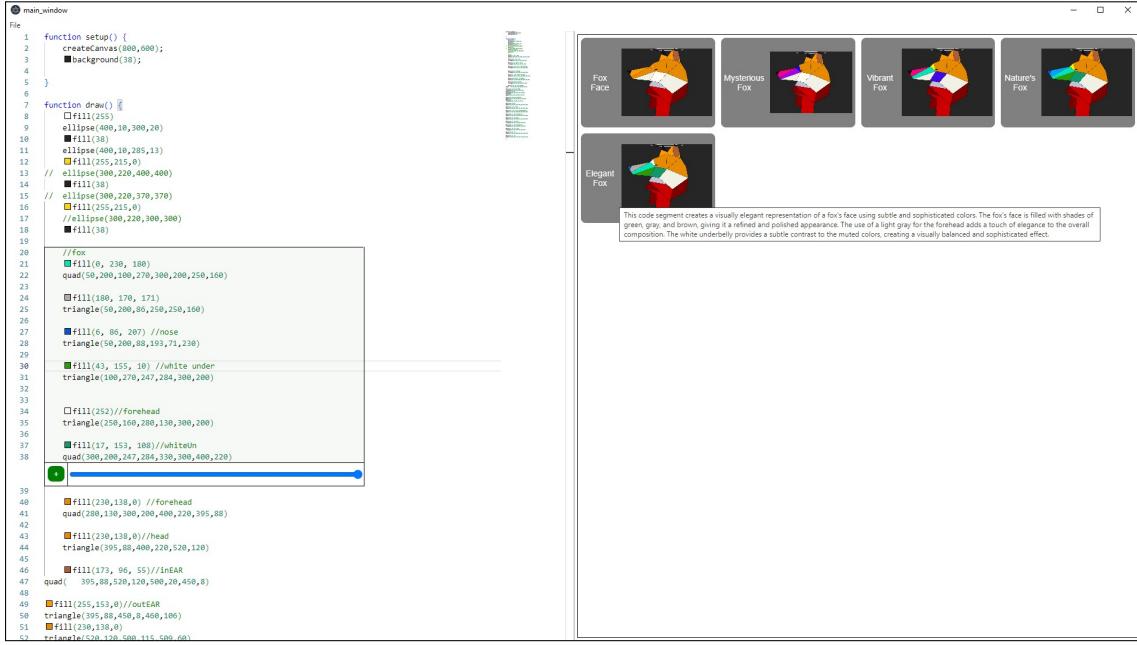
Figure 4.6: A screenshot of the *Ghost Editor*, showing the version view used to compare versions. Each version has an AI-generated name and description.
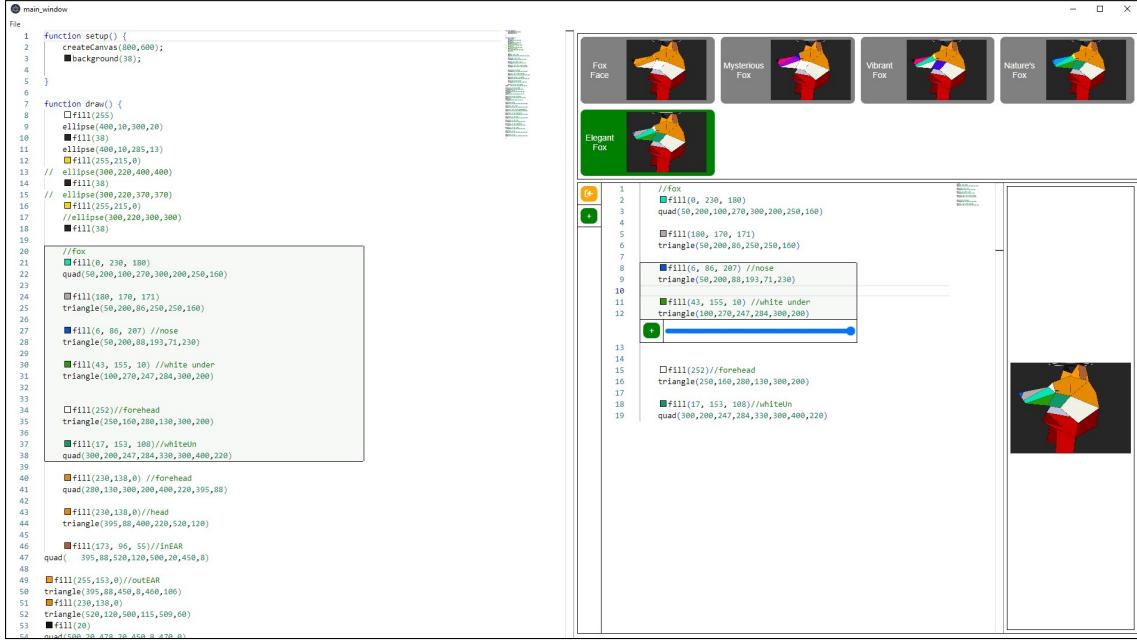
Figure 4.7: A screenshot of the *Ghost Editor*, showing the version editor used to modify versions independent of the main editor. The yellow button will load the selected version, the green one duplicates it.

noting that the version editor also supports versioning, and is synchronized with the overall version history for each line.

This interface is designed to hide the tool's complexity as long as possible, only revealing components when they are most useful. Crucially, manual user prompts are minimized through contextual layer selection. By providing visual triggers based on the editing context, the user is cued when they are most likely to interact with the tool. This should improve intuitiveness and reduce the entry barrier for new users.

Furthermore, the tool actively assists the user through its use of AI, by autonomously predicting names and descriptions for versions. This information is equivalent to commit messages in tools like *Git*, and crucial for effective communication in the development process. Nonetheless, these annotations are notoriously neglected by developers due to a lack of time and motivation (70, 71). This is a prime example of the *Ghost Framework*'s ambition to minimize workflow disruptions and support developers actively.

Concurrently, the only recorded information concerns the project's version history. This data can be freely accessed by scrubbing through the version timeline, and is stored locally. As a result, the user has full control over their data. However, all AI-based features will result in 3rd-party processing of some recorded version information. This must be clearly communicated prior to use.

Finally, the *Ghost Editor* proposes the novel idea of a dedicated version editor. This feature enables fast experimentation on varying versions, without the need to manually switch between these versions. The tool automatically integrates the modified code into the project and provides a real-time preview of the result. This can aid exploratory processes significantly, as it massively reduces versioning overhead for temporary experiments compared to traditional versioning methods (e.g., manual versioning in files, *Git*).

4.1.2   Auxiliary Creative Coding Concepts

Beyond the core versioning functionality, the *Ghost Editor* introduces several additional concepts to improve creative coding in general. These features are partly derived from existing creative coding editors, in particular the *P5JS Web Editor*, which is also used during the evaluation in chapter 5.

Real-Time Preview Many creative coding applications are related to visual artefacts, including images, videos, and interactive animations. Developers frequently observe this output to evaluate their current progress. Consequently, most creative coding
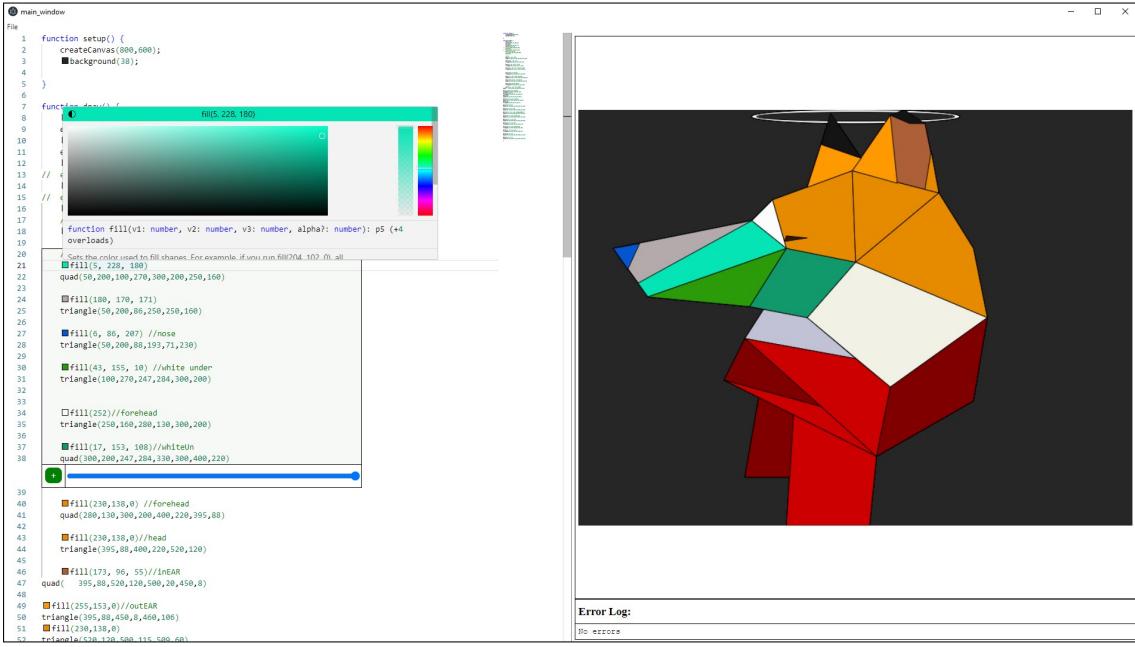
Figure 4.8: A screenshot of the *Ghost Editor*, showing the expanded colour picker and inline documentation for the *fill* function underneath.

editors contain a preview feature, executing the user's code on a button click. The *P5JS Web Editor* also includes an auto-refresh option, updating the preview automatically as soon as the user stops typing. The *Ghost Editor* develops this idea further with a real-time preview, refreshing almost instantly.

This preview enables a "what-you-see-is-what-you-get" workflow, allowing users to reflect on changes immediately. This feature is also used in the editor's version view, synchronizing every saved version with changes elsewhere in the project.

Colour Picker When working with visual artefacts, colours are of utmost importance. However, to many people, digital colour representations like the RGB colour space are rather unintuitive (72). As a result, the *P5JS Web Editor* provides various colour representations, including predefined colour names (e.g., "red", "lightblue") and hex codes. Most importantly, an embedded colour picker enables visual selection of colours from within the code editor.

This is an exceedingly useful feature, simplifying colour manipulation significantly. The *Ghost Editor* includes a similar feature, as shown in Figure 4.8.

Inline Documentation This thesis has frequently praised Microsoft's *IntelliSense* as a prime example of the *Ghost Framework*. As a result, *IntelliSense* is embedded in the
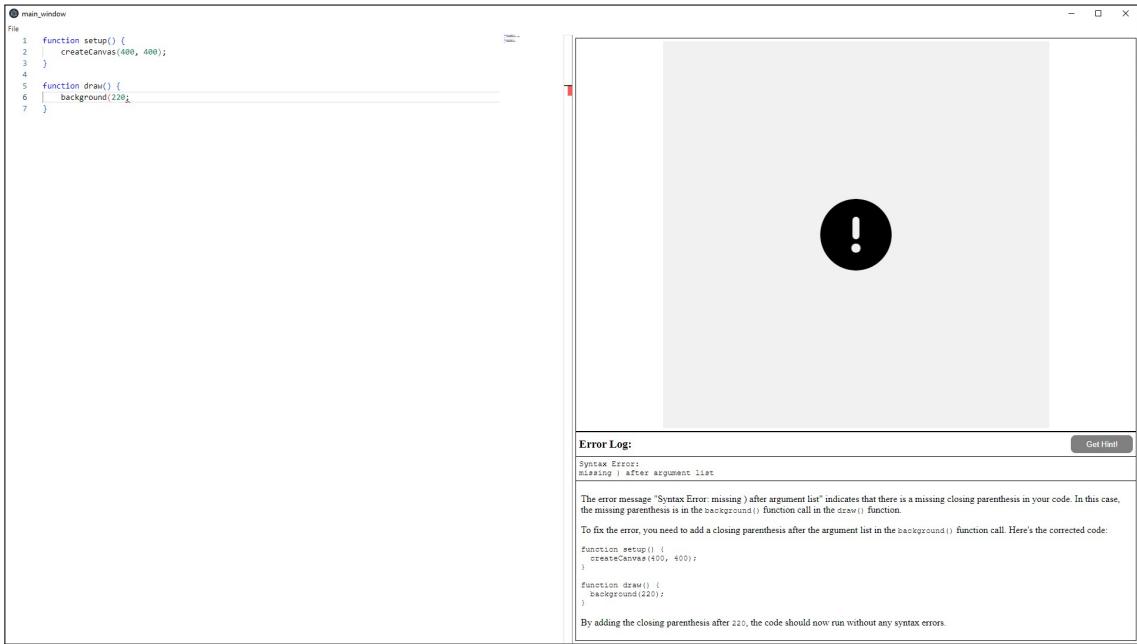
Figure 4.9: A screenshot of the *Ghost Editor*, showing a simple error hint underneath the error message.

Ghost Editor, providing the same inline documentation and code completion when editing code. Furthermore, the tool was configured to provide additional documentation related to creative coding.

AI-Based Error Hints Every programmer has to face bugs, and consequently error messages. However, these messages are frequently confusing and hard to comprehend. *Large language models* (LLMs) present themselves as a new opportunity to explore errors and bugs in the context of the original code (73–75). This can be extremely beneficial for new users, or professionals from other domains (e.g., artists or designers). To explore this idea, the *Ghost Editor* includes a feature to automatically explain errors using OpenAI's *GPT-3*.

The results are presented together with the original error, and can clarify cryptic messages, as shown in Figure 4.9. However, LLMs are by no means perfect, and validating their output is hard. Therefore, it is possible that provided hints are (partly) wrong.

This concludes the conceptualization of the *Ghost Editor*. Its design closely follows the principles of *Intuitive Assistive Automation*, predicting and simplifying version management through automated change tracking and contextual version access. The multilayered

user interface scales complexity based on the user's workflow, and AI-based automation encourages versioning for a more exploratory process. The technical design and implementation enabling these concepts in practice is illuminated in the following sections.

4.2 Architecture

The *Ghost Editor* is designed for a local deployment without remote components. Nonetheless, its architecture considers a dedicated front- and backend. The frontend is responsible for the user experience. This includes file management, code editing, result preview, version visualization, context analysis, and data collection. The backend, on the other hand, is concerned with data processing, version control, and data storage.
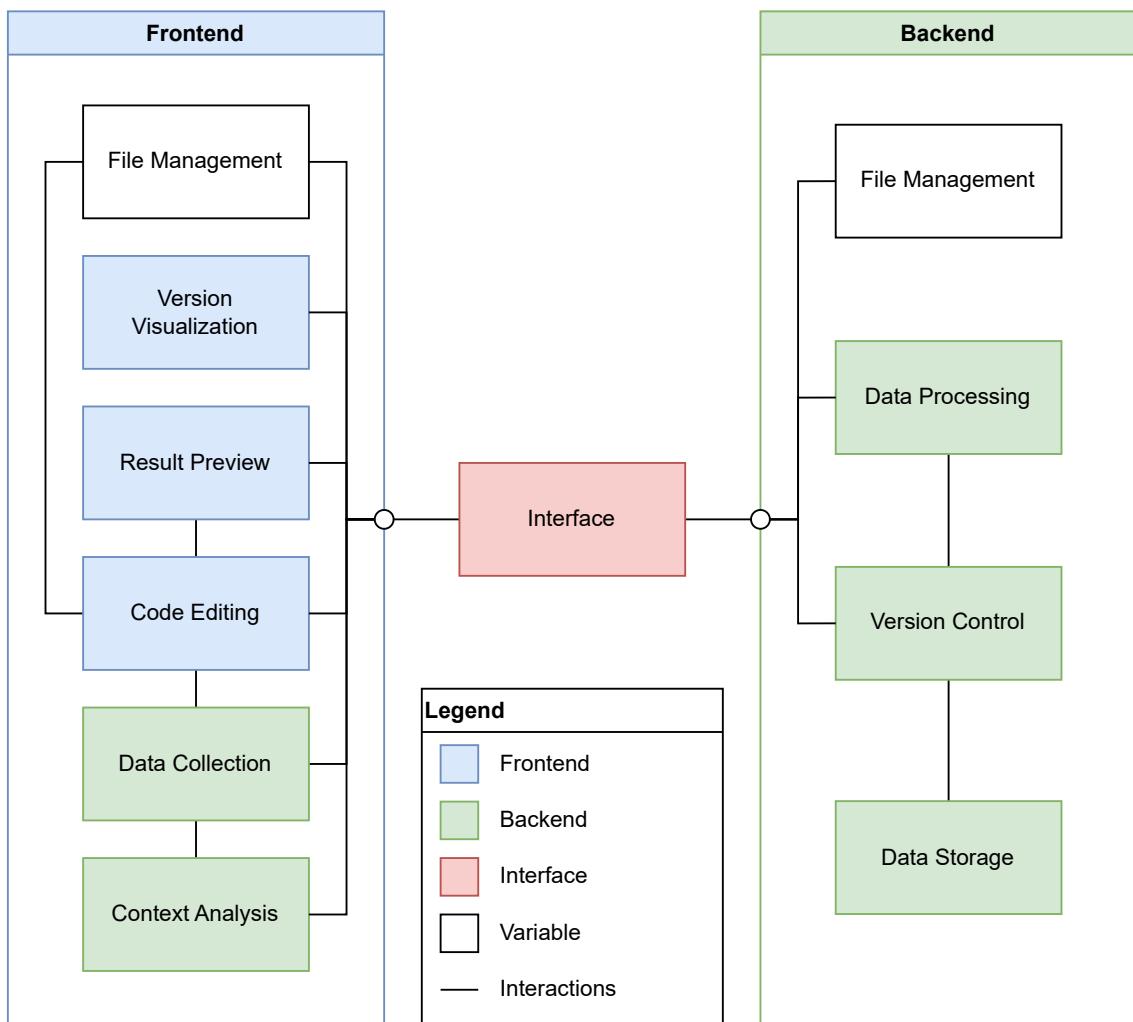
Figure 4.10: Responsibilities as distributed between front- and backend in the *Ghost Editor*. The colours indicate suggested responsibilities, according to the *Ghost Framework*.

Figure 4.10 visualizes these responsibilities. The colour coding highlights a mismatch between the editor's architecture and the *Ghost Framework*. While the framework considers data collection and context analysis a backend responsibility, the editor performs these tasks in the frontend due to technical limitations. The code editor provides an API required to collect real-time change data and context information unavailable to the backend. Consequently, the frontend processes this context information immediately, which also improves responsiveness. The change data is forwarded to the backend for further analysis. Depending on the specific architecture, this adjustment may be suitable for other applications of the *Ghost Framework*, if increased frontend complexity is acceptable.

Being built on top of *Electron*[1], the front- and backend run in different processes. As a result, there is no overlapping code between both components. All communication is performed through *Electron*'s inter-process-communication channels (IPC). The architecture of the individual components is detailed in the sections below.

4.2.1 Backend Architecture

Fundamentally, the backend operates as a server, providing a version control API to its client (frontend). Each interaction starts with the creation of a session, which identifies the client for all subsequent requests. Then, the client can load files for further versioning operations. The server loads these files into a database, and manages their history accordingly. Finally, files can be unloaded, and the session closed to end the interaction. Figure 4.11 visualizes these interactions in the form of a sequence diagram.

Persisting all versioning information in a database is crucial to ensure reliable and consistent operation across application restarts. The recorded information is strictly related to version management, and will not leave the user's system. It can be accessed transparently through the version interface, which allows access to all recorded versions.

To improve performance, the backend server additionally caches data loaded from the database in memory during use. This improves access times significantly, and is crucial to manipulate versions rapidly. In theory, this technique could be applied to writing operations as well, but due to time limitations this was not implemented. As a result, large insertion operations can take a few seconds (e.g., when inserting hundreds of lines at once). The user interface will indicate this by means of a loading animation.

---

[1]https://www.electronjs.org/
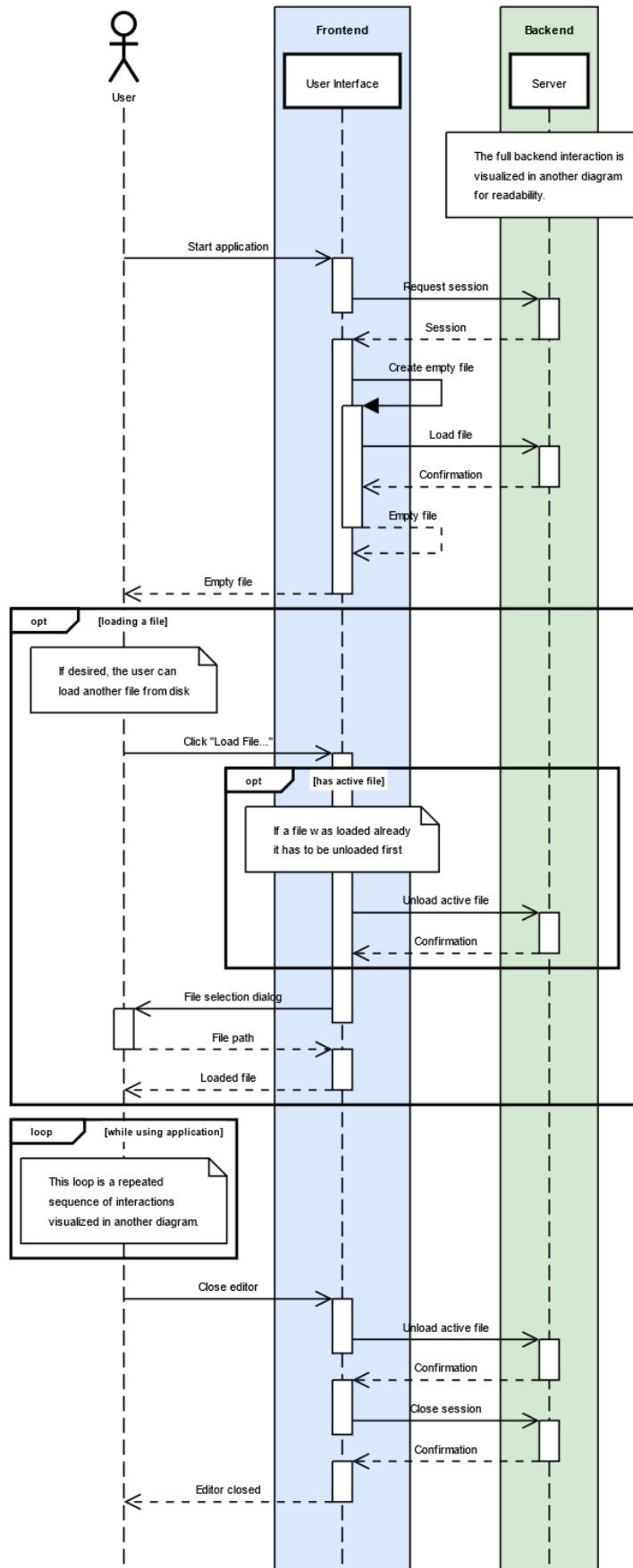
Figure 4.11: Sequence diagram of frontend interaction. All backend interaction is reduced to server requests. Figure 4.12 visualizes internal server interactions.

4.2.1.1 Functional Components

Internally, the backend relies on 5 core components: the server, a resource manager, a query manager, a cache manager, and finally a database client. The server is responsible for orchestrating any incoming requests, and uses the resource manager to load required session data. The query manager then sorts all incoming request to avoid merge conflicts when applying changes to the database. Once a query can be executed, it loads additional data through the cache manager. Any cached information is returned immediately, while missing data is loaded by the database client. New data will be cached for future use. If a query modifies the database directly, it will do so via the database client as well.

The full process is presented in Figure 4.12. As shown, the backend is completely driven by frontend request. This differs from the framework's original design, as data collection and context analysis were integrated into the frontend. Other tools may use the backend to trigger contextual suggestions.

4.2.1.2 Data Scheme

Throughout its operation, the backend uses a simple data scheme to manage versioning information (see Figure 4.13). This scheme is designed around line-based versioning, and treats each line as an individual versioning object with a dedicated version history. The details of this file representation are explained below.

File In the *Ghost Editor*, a file can be understood as a container object, used to reference its individual lines. It provides meta information such as the file path, but all versioning operations are performed on the lines directly.

Line Each line contains a list of versions, sorted by timestamp. Any user edits are decomposed in changes on individual lines, and will be added to this version history.

The order of lines within a file is maintained by the *order* property. This enables efficient line insertion by choosing the average *order* value of the pre- and succeeding lines.

Version A version consists of a timestamp and the full line content. Additionally, some versions are inactive, for instance when they represent the deletion of a line.
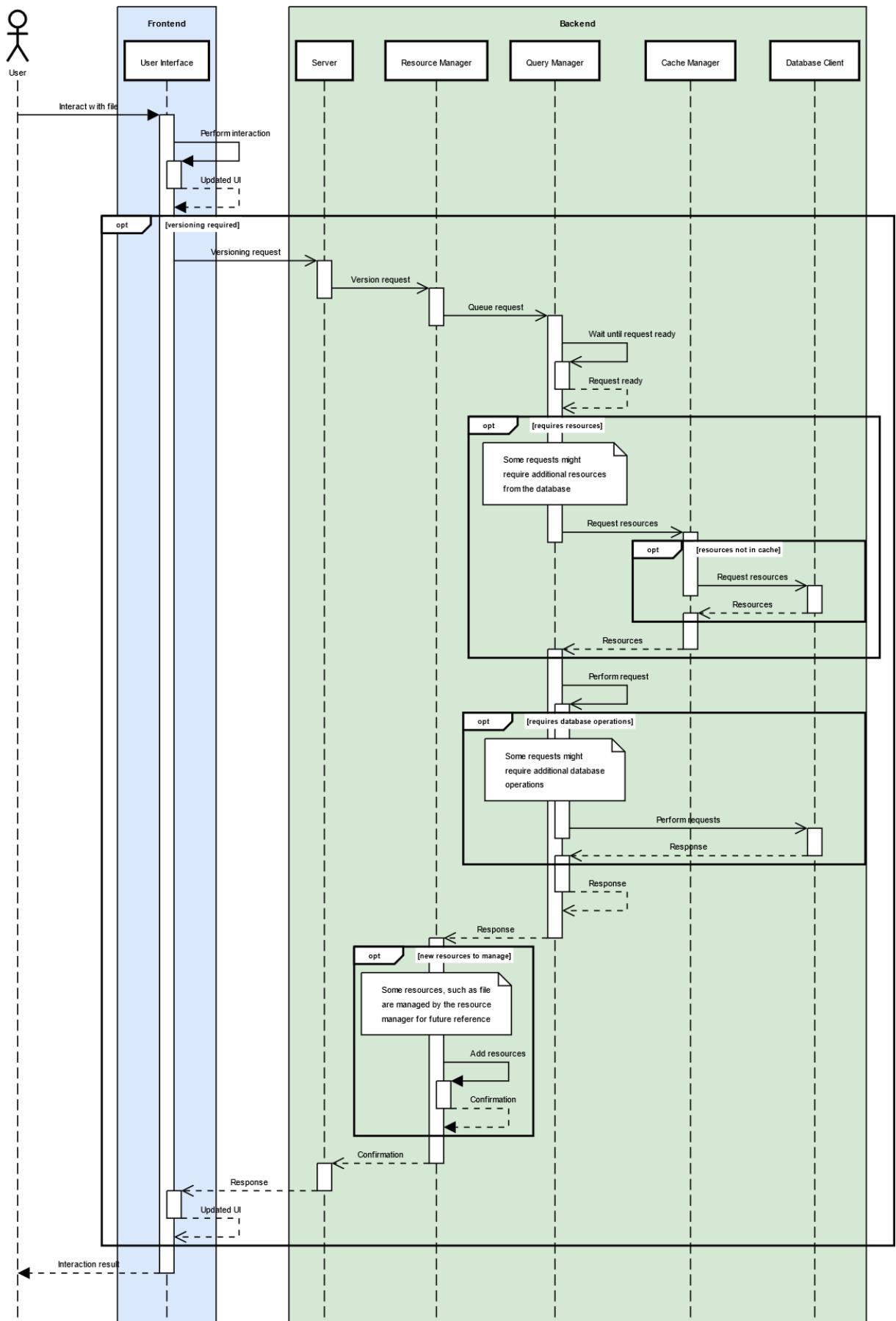
Figure 4.12: Sequence diagram of a single generic backend operation.
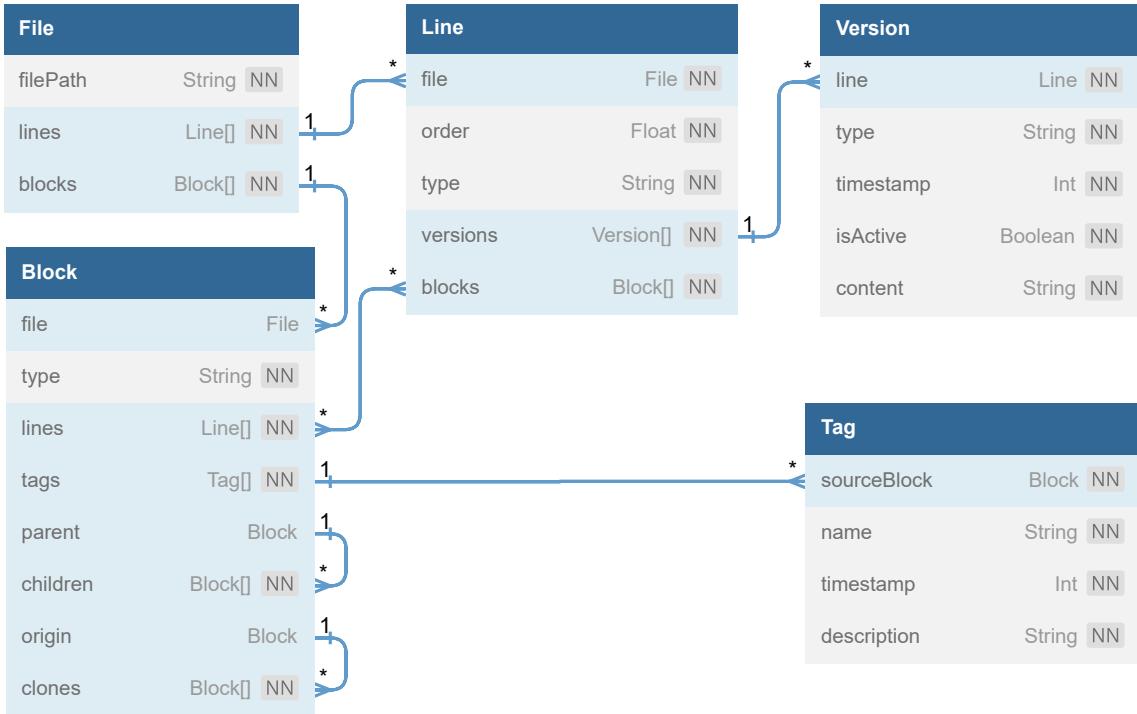
Figure 4.13: Data scheme for versioning information in the backend.

Beyond the file data itself, this scheme also accounts for local versioning by means of **Blocks**. Each block represents a selection of lines, and provides saved versions in the form of **Tags**. These tags contain a timestamp, which can be used to reconstruct the correct version for each individual line. As a result, tag storage is extremely efficient. In the user interface, blocks are equivalent to snapshots, and tags represent saved versions. The slider of the version interface is used to iterate through all timestamps fluently, offering access to every past state of a snapshot.

A final technicality concerns inserted and deleted lines. They are treated identical to other lines, however, their history starts and ends with an inactive version respectively. When an inactive version is selected by the user, these lines will provide no content to the assembled code, and thus visually disappear in the editor. They can be accessed again by selecting another version.

4.2.2 Frontend Architecture

Throughout the sequence diagrams in Figure 4.11 and 4.12, the frontend is depicted as a single unit, responsible for user interface, context analysis, data collection, and backend communication. In reality, these tasks are spread across numerous components, as shown

in Figure 4.14. Roughly, each interface layer is implemented by a single component. For the sake of complexity, the hidden layer is divided into two separate components, while the interface manager and snapshots are primarily responsible for the context analysis. Finally, the Ghost editor and the snapshot manager operate layer-independent, as they provide functionality across all layers. Each individual component will be described in more detail below.
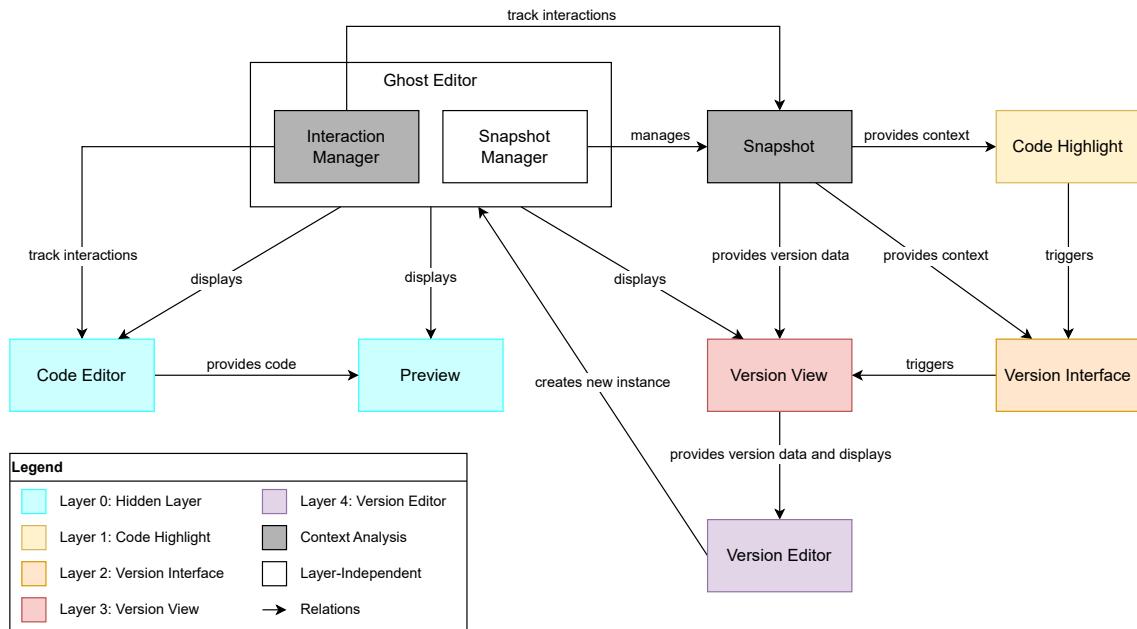
Figure 4.14: Frontend components and their relation, coloured according to their respective interface layers.

Ghost Editor The Ghost editor manages all other frontend components and loads the user interface on application startup. It serves as a central access point for information across components, and integrates with the interaction and snapshot managers.

Interaction Manager The context analysis is divided across several components. However, the interaction manager collects contextual events from different various sources and uniformly forwards them to the respective components for execution. This includes editing events, and snapshot interaction.

Snapshot Manager Whenever the user creates a snapshot, the snapshot manager takes over control. It synchronizes snapshots with the backend and forwards any interactions to the interaction manager, if necessary. The manager also provides snapshot access to all other components.

Snapshot A snapshot represents a local selection of code that is under version control. Each snapshot contains information about its position, the contained code, and created versions. This component interacts with the backend to create new versions and access existing ones, while it forwards any user interaction to the interaction manager.

Code Editor The code editor is the core editing tool of the *Ghost Editor*. It allows the modification of code, and provides various support features, including code completion, inline documentation, and a colour picker. Any code changes are relayed to the interaction manager. Furthermore, the editor provides an interface to embed inline UI elements.

Preview The preview is integrated with the code editor and renders the code in real-time. Any changes are applied almost instantly.

Code Highlight Each snapshot is visualized with a green box inside the code editor, using its inline UI feature. Whenever the user clicks into the code highlight, the version interface is loaded as well.

Version Interface The version interface is an extension of the code highlight and allows the user to save and access past versions. When saving a version, it also enables the version view.

Version View Saved versions for a snapshot can be viewed in the version view. It presents a list of all versions, including a small preview, and the version name. Clicking on a version opens the version editor.

Version Editor The version editor can be used to modify versions without loading them first. It instantiates a new Ghost editor, and recursively loads all other components as well. As a result, the snapshot feature is available in the version editor as well.

For the most part, each component implements a different layer of the *multilayer interface design* proposed in subsection 4.1.1. The Ghost editor integrates all of these components into a complete frontend. Each component has full access to the backend, and can perform versioning operations independently.

4.2.3 Interface Architecture

Electron's IPC-based communication dictates an asynchronous, event-driven interface between the front- and backend. As a result, an event handler in the backend translates frontend requests into function calls. The backend's response is transmitted via a corresponding answering event.

This interface could easily be adapted to any other communication protocol, such as *HTTP*. In this case, the events can be converted into *HTTP* requests, enabling a full client-server architecture. Being built on top of web technology already, this also means the editor could be converted into a conventional web app accessible through the browser. Furthermore, the editor could be transformed into a multi-client system with collaborative features, if desired.

4.3 Implementation

Originally, the *Ghost Editor* was designed as a plugin for *Visual Studio Code*. This would have enabled full project management, colour themes, plugin compatibility, and many other benefits that come with a mature code editor. However, the required plugin API lacks flexibility in terms of UI design, and caused the project to pivot. Instead, the editor is built on top of the *Monaco Editor*[1], Microsoft's open-source web-based code editor that also powers *Visual Studio Code*.

The *Monaco Editor* provides a fully featured code editor that easily integrates into any web application. It provides syntax highlighting, code completion through *IntelliSense*, inline documentation, efficient code search, and many other features to improve the editing experience. However, compared with *Visual Studio Code*, it is fully customizable and flexible enough for the requirements of the *Ghost Editor*.

Being a web-based component, the *Monaco Editor* has to run in a browser-like environment. However, most modern browsers only provide limit file management capabilities due to security concerns. As a result, the *Ghost Editor* was built on top of *Electron*, enabling the design of a native cross-platform web application with full file system access and the versatility of modern web development.

The editor is written in *Typescript*[2] to benefit from a strong type system, and leverages the vast ecosystem of *Node.js* to integrate various dependencies. The front- and backend use independent code bases, though they share the same technology stack.

---

[1] https://microsoft.github.io/monaco-editor/
[2] https://www.typescriptlang.org/

To enable local deployment, the backend uses a *SQLite* database[1], and accesses data through *Prisma*[2], a modern object-relational mapper (ORM) designed for tight integration with *Typescript*'s type system.

The frontend mostly uses custom *Typescript* code to generate UI components. The *Monaco Editor* additionally provides an API to embed UI into the editor itself, which is used for the inline UI components. Unfortunately, this strategy failed to scale with the project's ambitions, which led to a partial integration of *React*[3] components for some functionality, such as loading animations and the real-time preview.

The build pipeline for the *Ghost Editor* is based on *Electron Forge*[4], a first-party packaging tool for *Election*, and *Webpack*. This setup allows for a versatile build process with native support for all major operating systems, including *Windows*, *macOS*, and *Linux*. Both *Windows* and *macOS* have been tested and are verified to work seamlessly.

The project was developed using *Git*, and the final code is available on *GitHub*[5].

4.4 Limitations

The final editor runs mostly stable, with only few known bugs and no notable crashes throughout the evaluation phase. Performance is good, even on older hardware[6], with most operations completing instantly and a smooth editing experience overall. While these observations are a testament to the editor's functional design, it is not without fault. The most important limitations are discussed below:

Interface Complexity While the user interface of the *Ghost Editor* is designed for intuitive, fluent interactions, it provides a variety of novel concepts that might be foreign to new users. In particular, experienced users of external versioning tools like *Git* might find the direct integration into the code editor distracting, and the intent of contextual suggestions could be misunderstood. In particular, the large amount of intermediary versions accessible via the timeline can be overwhelming.

Additionally, the *Ghost Editor* requires an explorative mindset to be most effective. It is designed to value the development process as a whole, instead of the final result.

---

[1] https://www.sqlite.org/
[2] https://www.prisma.io/
[3] https://react.dev/
[4] https://www.electronforge.io/
[5] https://github.com/Xantocx/ghost-editor
[6] Tested on an *Intel Core i7-7500U* mobile CPU with 16 GB of RAM from 2017.

Users only interested in the latest version of their code might struggle to see the editor's value.

Scalability Throughout the evaluation phase, the editor performed well in small-scale projects. However, its lack of advanced file and project management is likely to affect scalability. While these issues can easily be addressed in the future, the fundamental versioning system might require significant modification to account for project-wide versioning. As of now, the editor does not support multi-file projects.

Compatibility While the *Ghost Framework* specifically encourages the use of an extension-based development environment, in its current form, the *Ghost Editor* does not support plugins or extensions without significant development efforts. In hindsight, a modified user interface design could have allowed for a *Visual Studio Code* integration, eliminating this issue. In its current state, developers are limited to the provided feature set.

Storage Overhead Most modern VCSs use some form of delta-based data format to store changes (76). The *Ghost Editor* record the full line content for every new line version. In combination with real-time versioning, this generates an enormous amount of version data, increasing storage usage significantly compared to established tools like *Git*. The was no issue during our small-scale tests, however, this could change when using the editor for extended periods of time.

Performance Overhead While the editor's performance is mostly acceptable, the current design includes some specific weaknesses. In particular, the decomposition of complex change operations into line-based versions and the subsequent database operations can be expensive, taking up to a few seconds. Similarly, AI-based tasks take an unpredictable amount of time, usually a few seconds as well.

The interface hides these issues behind a loading screen, but long loading times can lead to user frustration. Modifications to the current architecture could enable background processing for such operations, providing a more responsive user experience.

Most of these issues are related to the conceptual design of the editor, rather than its implementation. However, with some additional work, they could be resolved, e.g., by providing a built-in tutorial and tool tips to improve the user's understanding of the interface.

Naturally, this list is not exhaustive. A more practical evaluation will follow in chapter 5, involving real user feedback.

5

Evaluation

This chapter is concerned with the practical evaluation of the *Ghost Editor* compared to existing tools for creative coding by means of a hybrid user study. As part of this study, participants are requested to perform a basic creative coding task in an established creative coding editor and the *Ghost Editor*. Each task is followed by a short, semi-structured interview about the participant's personal experience with the respective editor. Both, the task execution and interview are recorded. Finally, the participants fill out a standardized form, quantifying their observations for statistic comparison. The goal of this study is to understand if and how the *Ghost Editor* can improve the established process for creative coding and, in particular, versioning.

Testing a code editor is a time-consuming task, particularly if the participants are not yet familiar with (creative) coding. As a result, the selected programming assignments are very small and focus on emulating the creative process as well as possible. This also explains the hybrid data collection approach: While quantitative data is optimal for visualizing general tendencies and preferences, it is nearly impossible for participants to express the fine subtleties of their experience after such a brief testing period quantitatively. The goal of the interview is to uncover some of these insights through dynamic follow-up questions.

The study is designed as a comparative scenario to give inexperienced users the chance to understand current challenges of creative coding before evaluating the proposed solution. The specific details of this scenario are discussed in the section below.

5.1   Study Execution

For the study experiment, the *P5JS Web Editor* was selected as an established to establish a baseline experience. Created by the designers of *P5JS*, it is powered by the same creative

coding library as the *Ghost Editor*, eliminating language-specific difference as a potential source of bias. Furthermore, the *P5JS Web Editor* served as a design blueprint for the *Ghost Editor*, making it a suitable reference for comparison.

Participants are interviewed individually by a single interviewer, either in person or via *Zoom*[1]. The interviewer introduces the environmental conditions of the study, and prepares the tasks for the participant. Each task follows the same process, beginning with a short introduction to the given editor and the *P5JS* library. Next, the interviewer guides the participant through their task and conclude with a short interview on their editing experience. This process is recorded. After the recording has ended, the participant is finally asked to fill out a set of survey questions on the used editor. This process is then repeated for the second editor, and finalized with a set of additional questions specific to the *Ghost Editor*'s expanded feature set (e.g., the integrated VCS).

Both tasks follow a similar structure, however, differ in content. For the first editor, the participant is requested to draw a simple house, followed by several modifications (e.g., adding a door, changing the daytime). In particular, some of these modifications require the reconstruction of previous versions. This should simulate the exploratory process of creative practitioners; testing out different ideas and returning to old versions repeatedly. For the second editor, this process is repeated, changing the subject to a car.

Throughout their coding session, participants can ask the interviewer for advice concerning the coding itself. This includes information about *P5JS*, and conceptual questions about their drawing. This is done to simplify programming, as the *P5JS* library is not the intended subject of this study.

5.2   Participants

To ensure a general alignment of interest, the population of participants is limited to creative practitioners and programmers interested in programming as a creative tool. Both parties have a basic understanding of the domain, and are likely to engage in creative coding naturally. Unfortunately, the time-frame of this thesis necessitates previous programming experience, eliminating creatives engaging in programming for the first time.

Given these constraints, the most appropriate source of interviewees are the computer science degrees at the *Vrije Universiteit Amsterdam* (VU Amsterdam), and make up the largest portion of participants. In total, 8 participants contributed to this study, all of

---

[1]`https://zoom.us/`

which had previous experience in computer science and programming, mostly through academic education. Unfortunately, no creative practitioners contributed.

All participants were briefed on the usage of their data in the context of this thesis, as well as future research projects. They consented to these use cases. Additionally, the study passed the self-check on ethical research design by the *Ethics review committee of the Faculty of Science* (BETHCIE) at the VU Amsterdam.

5.3 Collected Data

This study collects different types of data, ranging from well-structured survey questions to semi-structured interview recordings. For the purpose of reproducibility, the full catalogue of survey question can be found in Appendix A, together with a list of basic questions used to initiate interviews in Appendix B.

Both survey and interview questions are designed to reveal the participant's experience between the two editors. The goal is to identify specific differences and their impact on the overall usability. Versioning is an important factor, trying to understand the benefits and disadvantages of the novel versioning system proposed in this thesis. A full segment of questions is dedicated to this topic. The full survey responses[1] and interviews[2] are available online.

5.4 Evaluation Process

Due to the hybrid nature of data in this study, the survey responses are considered isolated from the interviews. This also ensures anonymity of survey responses. The specific evaluation processes are detailed below.

5.4.1 Survey Data

Due to the small sample size, the data was evaluated in aggregated form, using the average and median score as reference for each question. The standard deviation is used to further investigate the composition of these results. Using these metrics, the participants' overall experience with the editors can be quantified.

While the average was used as the guiding measure, the median and standard deviation serve as a tool to detect internal disagreement. In these cases, the individual responses

---

[1]Survey Results: `https://docs.google.com/spreadsheets/d/1ZE_l-UF83APa3QueIrqslxoi1Npo6SvP OMBfWJv8E8M/edit?usp=sharing`

[2]Interview Recordings: `https://surfdrive.surf.nl/files/index.php/s/VI9tXqZwslsmHYQ`

were reviewed in more detail. Textual responses serve as an additional source of reasoning when explaining such contradicting experiences.

The survey was split into several related categories (e.g., usability, feature set) and results analysed within this context. Additional information collected from our interviews was used to refine the interpretation of the numerical data.

5.4.2 Interviews

The interviews provide deep insights into the participants' perception of both editors. Using the survey data for a general sense of direction, interview responses can yield details about the participants' experience and reasoning. This information was used to contextualize numerical observations, and serve as the foundation for a discussion of these results.

While a more structured approach such as *Grounded Theory* (77) could help to extract more specific information from these interviews, the given time-frame did not allow the implementation of such a strategy. However, this may be a valuable addition in a future continuation of this work.

5.5 Results

This section presents a summary of the collected data, with a deeper discussion in the context of our research questions following in chapter 6. The data is divided into three, categories, starting with an evaluation of the participant demographics, followed by an editor evaluation and a detailed look into the novel versioning system of the *Ghost Editor*. All numerical data is extracted from the survey responses where participants rated their experiences, while qualitative observations are based on the interviews. Some additional qualitative data were generated by the long-form survey responses.

Many survey questions were rated on a scale from 0 to 10, in which case 0 always refers to a negative answer, while 10 is the most positive response. For instance, when asking "*How would you rate your coding experience?*", 0 refers to "*I have not programmed before*", while 10 is equivalent to "*I am highly skilled*".

5.5.1 Participant Demographic

Firstly, the participants are mostly male (7 out of 8), and between 18 and 25 years old (6 out of 8). This is likely a related to the high number of (former) computer science (CS) students (6 out of 8), a subject notorious for under-representation of women (78). Half of all participants have completed a Master's degree, 3 of which in CS, while another 3

completed a CS Bachelor's degree. Currently, 3 participants are still studying CS, while the other 5 are working in the industry, again in CS.

Considering their background, it is no surprise that all participants have a good previous knowledge of CS - specifically programming. 3 participants rated their own experience with a moderate 6 out of 10, the other 5 responded with a 9 or 10. When asked about their experience in detail, most participants revealed an affinity to software engineering and full-stack development. The most notable exceptions are data science and artificial intelligence, as well as IT-security (one participant each).

None of the participants have a particularly strong background in creative coding, with 5 participants rating their experience as 0. Two participants had heard of the discipline before (rated 1 or 2), and one participant responded with 6. Any previous experience was limited to the use of basic creative coding tools (no examples provided) and data visualization.

5.5.2   Editor Evaluation

Each participant answered an identical set of questions for each editor. The questions are mostly concerned with the general user experience, the editor's feature set, performance, and its learning curve. As the *P5JS Web Editor* does not have any built-in version control features, this topic is explored in more detail later (see subsection 5.5.3). The data for both editors is presented in Table 5.1, and is evaluated below.

5.5.2.1   P5JS Web Editor

In terms of usability, the *P5JS Web Editor* was rated with an average score of 7.97 out of 10. The median is 9, indicating an outlier at the lower end. This outlier arose from participant 7 who consistently rated the *P5JS Web Editor* worse than all other participants, sometimes up to 6 points lower than the second-lowest rating. In their textual explanation, they specifically referred to a lack of features and several bugs other participants did not experience, even though they were reproducible (e.g., the colour picker overwriting wrong code, unsafe execution of JavaScript functions that crashed the browser). Excluding their rating results in a usability score of 8.43 and a median of 9. The individual questions are rated similarly, with the installation process having a particularly high rating (8.875), likely because the editor is available as a website.

With an average score of 5.63 and a median of 6, the *P5JS Web Editor*'s feature set is the worst assessed category. Naturally, the absence of version control is a major contributor to

Categories & Questions		P5JS Web Editor			Ghost Editor		
		Avg.	*Median*	σ	*Avg.*	*Median*	σ
Usability	Installation Process	8.875	9	1.64	7.875	8.5	1.89
	First Impression	8	8.5	2.56	8.625	9	0.92
	Ease of Use	7.25	7.5	2.19	9.5	10	0.76
	User Interface	7.75	8.5	2.05	8.5	9	1.07
	Overall	7.97	9	2.12	8.625	9	1.31
Feature Set	Completeness	6.125	7	2.30	8.5	8	0.76
	Preview	9.375	10	0.92	9	9	0.76
	Code Editor	5.375	6.5	3.02	8.875	9	0.99
	Error Handling	6	6	1.51	8.125	9	2.30
	Version Control	1.25	0	1.75	8.625	10	2.77
	Overall	5.625	6	3.26	8.625	9	1.68
Performance	Performance	8.375	8	1.69	9.125	9.5	0.99
	Reliability	8.625	9.5	2.39	8.375	8	0.92
	Stability	9.25	9.5	0.89	9.5	10	0.76
	Overall	8.75	9	1.73	9	9	0.98
Learning Curve		8.5	9	2.33	8.25	8.5	0.89
Overall	*Rated*	7.25	8	1.83	8.625	8.5	0.744
	Computed	7.29	8	2.80	8.68	9	1.34

Table 5.1: Aggregated survey results for both editors. All ratings are provided as average (*Avg.*) and median over all participants, together with the respective standard deviation (σ). Each category has several subcategories, one for each survey question. The last row is split into *Rated* and *Computed*, which refers to the conclusive rating by the participants and the average result computed from all question ratings, respectively. Values are rounded to two digits, except if a value only has 3. Better results are marked in green, dark green highlights summarizing results.

this result, however, the code editor, error handling, and general feature completeness all scored between 5.3 and 6.2 as well. While the participant assessments of error handling were approximately unanimous (e.g., lack of meaningful debugging features), there were clear outliers for the editor and feature completeness. Participant 1 was particularly frustrated with the minimalistic code completion, while participant 4 noted the lack of proper project management features (e.g., full file system integration, external tool support). Only the preview was universally satisfying, with an average score of 9.375 and a median of 10.

In comments throughout our interviews, the lack of a visual coordinate system and the unpredictable code completion were two additional sources of confusions. Nonetheless,

participant 6 was completely satisfied with the editing experience, rating it a 10 out of 10. Yet, they would only recommend the editor to experienced programmers for the lack of a tutorial. All other participants recommended the editor to everyone, with only little prior experience required.

In terms of performance, there were little complaints, as reflected by the final score of 8.75. All three questions received an average score of at least 8, the median for reliability and stability is even 9.5. Reliability is affected by a single outlier; participant 7 complained about several bugs, as mentioned earlier.

The perceived learning curve was rated with an average of 8.5, all participants were able to quickly adapt to the new environment and created sketches with ease. Part of this is related to the accessible design of the *P5JS* library, however, the editor did not introduce major difficulties. Once again, participant 7 complained about several bugs and the confusing code completion.

In summary, the participants rated the editing experience with an average of 7.25 and a median of 8. This is extremely close to the unweighted average of all responses (7.29), and reflects an overall positive perspective. The editor was deemed useful, even though limited for larger projects. The lack of version control was a common complaint, as was the minimalistic approach to code tools, specifically the code completion. Generally, the editor was recommended for new programmers with some prior experience.

5.5.2.2 Ghost Editor

The *Ghost Editor* received a significantly better overall rating of 8.625, which again matches the computed average of 8.68 closely. Across all 3 categories, the *Ghost Editor* saw generally improved scores over the *P5JS Web Editor*. However, there were a few exceptions.

One shortcoming was the setup process. Built as a native *Electron* application, the editor requires an installation process. A corresponding installer was distributed and worked without issues for all participants[1]. However, this process cannot compete with the simplicity of a conventional web app. Furthermore, the result was affected by participant 5, who used a prepared device with the editor pre-installed, due to unrelated complications with their own device. As a result, they could not assess the installation process and rated it with a neutral 5 out of 10.

For all other usability questions, the *Ghost Editor* received superior results, with average scores of 8.625 and above. The relatively low standard deviations of approximately 1 and

---

[1]Only Windows systems were tested.

below also indicate a general agreement across the participants. Comments gave credit to an intuitive user interface, the editing experience, and integrated version control.

Feature-wise, the more powerful code editor was considered one of the greatest improvements of the *Ghost Editor*. The advanced code completion with inline documentation was positively noted by all participants, and the *Monaco Editor* was generally regarded as a modern alternative to the web editor. The lack of file and project management was criticized for both editors, however, the participants considered the overall features set of the *Ghost Editor* complete enough for most smaller projects and new programmers.

The AI-based error hints found little use during the limited testing time, but most participants acknowledged the concept positively. Only participant 5 voiced concerns that programmers with little prior experience might accept the AI-generated solutions blindly and lack the knowledge to reflect on potential issues.

The preview was rated with an average of 9 out of 10, slightly worse than for the web editor. While this result is still exceptional, the increased refresh frequency of the *Ghost Editor* produces error messages before the user stops typing. Some participants found this distracting.

The version control system was generally well received, with an average score of 8.625. Common concerns were mostly related to the novel user interface, which requires some experience for effective use. The greatest outlier is participant 5, with a rating of 2. They attempted to refactor their code into separate functions, which they felt interfered with the versioning system. Specifically, they missed the ability to move code together with its history by using the common copy-and-paste feature.

The other participants found the novel approach to version control intriguing, even though some participants did not see significant value outside creative coding. In their eyes, conventional tools like *Git* are already satisfactory. Additionally, the amount of accessible versions in a project can become large quickly and was noted to be overwhelming.

Performance was rated very similar to the web editor, with a brief overall improvement for the *Ghost Editor*. Reliability received better scores with the *P5JS Web Editor* (8.625 for the web editor, compared to 8.375 for our editor), but no explicit comments were made to clarify these ratings.

Finally, the expanded feature set of our editor requires some additional instructions, resulting in a slightly steeper learning curve. However, all participants agreed that a brief tutorial would suffice to introduce the advanced user interface. Generally, all participants would recommend the editor to users with at least some coding experience.

In summary, the *Ghost Editor* was able to improve on most evaluated points, albeit by a small margin. Given the generally positive experience participants had with the original web editor, this is a notable feat. The greatest overall improvement is related to the improved feature set, as the web editor only provides a very basic experience.

5.5.3 Versioning System Evaluation

To evaluate the novel features of the *Ghost Editor*, participants answered a number of additional questions regarding their usage of the versioning system. These questions are designed around the 4 additional interface layers used to integrate versioning into the editing experience, and are presented accordingly in Table 5.2.

Components		Usage Frequency				Rating		
		Never	*Occ.*	*Som.*	*Often*	*Avg.*	*Median*	*σ*
Versioning	Layer 1	0	5	3	0	7.875	8.5	1.89
	Layer 2	1	6	1	0	5.125	5.5	2.95
	Layer 3	0	4	2	2	8.375	8.5	1.30
	Layer 4	5	3	0	0	6	5.5	1.93
Error Hints		3	3	0	2	7.125	8	3.31

Table 5.2: Aggregated survey results for unique features of the *Ghost Editor*. Usage frequency is presented by the absolute number of participants voting for each option. The winning category is highlighted in green. The abbreviations *Occ.* and *Som.* stand for "occasionally" and "sometimes", respectively. All ratings are shown as average (*Avg.*), median, and with the respective standard deviation (*σ*). The different layers refer to the version control user interface, the error hints to the AI-based debugging suggestions.

5.5.3.1 Layer 1: Code Highlight

The code highlight is the visual component used to interact with versions of a code snippet. Each participant used this feature at least once - some more extensively than others. Overall, the highlight was perceived as helpful and unobtrusive by most participants. Interestingly, participant 4 noted that it actively encouraged them to interact with the versioning system. On average, the code highlight was rated 7.875, with a median of 8.5. Participant 1 and 5 were outliers with a rating of 5, commenting on the highlight as slightly distracting when editing code.

5.5.3.2 Layer 2: Version Interface

The version interface allows the user to access previous versions and save new ones. While all uses used the interface to save versions, the timeline feature was less popular. Most participants only used it occasionally, with participant 5 ignoring it completely. They did not see the need to access a version history of this granularity and were satisfied with their saved versions.

Overall, the opinions on the timeline differed significantly, as indicated by a standard deviation of almost 3. Participant 8 rated it with 10, claiming they would "love" to use it as part of their regular workflow. Participant 1 appreciated the concept, but only rated it 3 due to the cumbersome amount of versions accessible. This was concern was raised by several participants, as it made exploration of meaningful versions through the timeline much harder. Overall, the feature was rated with an average score of 5.125.

5.5.3.3 Layer 3: Version View

The version view got the best average rating of all features, with an average of 8.375 and a median of 8.5. The ability to compare versions visually was well-received, along with the automatic generation of names and descriptions. This simplified exploration of past ideas, and was used more frequently than any other feature.

5.5.3.4 Layer 4: Version Editor

The least popular feature of the *Ghost Editor* was the dedicated version editor. 5 out of 8 participants never used this feature, despite being introduced to it. Most participants voiced confusion about its use case, suggesting it might be more relevant in larger projects or as read-only access to previous versions. Participant 2 and 4 argued that the main editor would be sufficient for all required edits, and an additional editor would introduce unnecessary complexity to the user interface. Overall, the feature was rated with an average score 6 out of 10.

5.5.3.5 Error Hints

Another novel feature of the *Ghost Editor* are AI-based error hints. However, due to the trivial nature of our survey tasks, few participants found the opportunity to engage with this feature. Nonetheless, most participants showed interest in the concept, even suggesting a feature to automatically apply the suggested fix. However, participant 6 pointed out the danger of new programmers blindly accepting faulty suggestions. Participant 1 rated the

feature with 0 out of 10, as they encountered a bug that prevented its use. Overall, the error hints received an average score of 7.125, with a median of 8.

To conclude, the *Ghost Editor* was generally well-received and improved over the *P5JS Web Editor*, largely due to its more mature code editor and the integrated version control. The latter presented itself as a powerful tool with real-world use for creative coding. While the user interface integration has some potential for improvement (e.g., moving versions with copy-and-paste), the current state of the tool was deemed sufficient for use in real projects, albeit small ones due to the lack of project management features. Its application outside of creative coding is less clear, and is considered in more detail during the discussion.

6

Discussion

The core goal of this thesis is to answer the posed research questions in a satisfactory manner. The following discussion will attempt to do so based on the results of this research. We will begin with preliminary considerations that affect the following conclusions, then the questions are addressed in sequential order. The chapter concludes with some additional deliberations beyond the scope of these questions.

6.1 Preliminary Considerations

As many of the presented results are based on a user study, we have to consider the influence of participant demographics, as well as the general study design. The section presents the most important factors that may taint the following discussion.

6.1.1 Participant Bias

While we discuss topics at the intersection of arts and computer science (CS), all 8 participants have a CS background, mostly related to software engineering. To some degree this is necessary, as prior programming experience was required. However, conventional CS, and software engineering in particular, are, in many cases, requirement-driven disciplines. Consequently, some participants might struggle to appreciate the exploratory challenges of creative coding.

The evaluation tasks also suggested this discrepancy in practice, with most participants only fulfilling the minimum requirements for a given challenge, before awaiting further instructions. In contrast, the only data scientist in the study started experimenting on their own, without a need for clear guidance. This could be a testament to the exploratory

65

nature of their daily work. While these observations are not statistically significant, they reveal a general tendency.

As a result, tools designed specifically for exploration might receive less attention, which could bias the final results. Additionally, all participants are already experienced with sophisticated programming tools, and likely to expect a higher standard than a novice would.

Finally, most participants were recruited in their role as students as the *Vrije Universiteit Amsterdam*. The relation to the author's own studies at this university could affect the final results. Similarly, the impact of participant demographics (1 female and 7 male, age between 18-30) has to be considered.

6.1.2 Study Design

Unfortunately, testing programming tools is a time-intensive task. Due to time constraints and limited availability of participants, our study used a limited testing approach based on two simple programming tasks. However, these tasks are rather simple, and easily solved by an experienced programmer. Hence, the participants might not require advanced tools such a VCS as much as they would under different circumstances. This could affect their assessment of its validity.

Furthermore, the order of these tasks might affect their evaluation. During the survey, participants start by testing the *P5JS Web Editor*. Participants that have no experience with *P5JS* prior to this study will this be more comfortable with the library when testing the *Ghost Editor*, and might perceive the editor as more intuitive. A randomized design could have prevented that, however, the goal was to provide participants with an unbiased perception of the current state of creative coding tools before presenting our alternative approach.

6.2 RQ1: Exploration in Creative Tool Design

A key goal of the *Ghost Framework* is to design programming tools that can contribute to exploratory and creative processes effectively. Naturally, this is a very wide topic with many potential solutions. To narrow down the scope of this thesis, the issue is approached through two sub-questions discussed in the sections below.

6.2.1 RQ1-1: Exploration through Interface Design

For this thesis, the design of programming tools for exploration was framed as an interface problem. This perspective is based on the hypothesis that much of the functionality needed for exploratory programming already exists in other tools like *Git*. However, the user experience of these tools is optimized for a linear development process and is incompatible with exploration.

The *Ghost Framework* attempts to resolve this problem through intuitive assistive automation; tools that anticipate the user's actions and proactively produce solutions. This concept is founded in the observation that creative coding and similar tasks rely on a form of structured chaos to explore an unknown solution space effectively. Often, creative practitioners create a multitude of different versions and jump between ideas with no obvious pattern. This can create confusion in the documentation of their results (e.g., the famous "spagetti code" (79)). Tools that autonomously bring order into this chaotic environment without disrupting the existing workflow can be exceedingly valuable.

Version control is a natural topic to test this theory with, as it is fundamentally concerned with structuring and documenting program code efficiently. Additionally, conventional VCSs are often criticized for their bulky interfaces, (34, 35) and many exploratory programmers refuse to use them at all (1). *Multilayer interface design* serves as a good starting point to explore alternative solutions, and the *Ghost Editor* delivers the first observational data on this theory.

The user study revealed a general liking of our novel versioning approach, however, responses varied greatly between participants. While all of them acknowledged the advantages of comparing versions visually and streamlining version creations through automatic name generation, several participants stated they would rather return to conventional version control systems like *Git* for two main reasons:

Bloated Version History While the idea of a complete editing timeline seems intuitive at first, our implementation includes all previous versions with no way to filter for specific qualities (e.g., only versions that run without error). This makes history navigation hard, and resulted in frustration for several participants. Consequently, this feature was mostly neglected, and even posed a visual distraction when editing code. Hiding this feature in another interface layer and enabling some form of version selection might be a possible solution.

6. DISCUSSION

Little Added Value With access to the full version history being neglected, our versioning system essentially transforms into a conventional VCS with an inline user interface. The user can manually save versions, and access them from a separate interface. While the side-by-side version view is useful to compare different ideas, two participants claimed they would rather use *Git* for non-creative applications due to their advanced experience.

This raises concerns about the effectiveness of the proposed *Ghost Framework*. After all, tools are supposed to aid the user and simplify complex workflows seamlessly. By adding unused functionality, tools becomes increasingly complex without improving efficiency.

Luckily, these concerns did not apply to the general user interface concept. The inline interactions improved the versioning efficiency for most participants, and 3 participants specifically noted that it encouraged them to create versions more frequently. The AI-based generation of version names was seen as a major improvement over existing tools, as the selection of meaningful version descriptions can be tedious and discouraging. Additionally, the code highlight served as a constant reminder to actively interact with versions for at least one participant.

While several participants argued the novel interface requires some initial explanation, they all agreed it can be learned easily, and integrates naturally into the coding process. As a result, the *Ghost Editor* actively encourages exploration, and enables effective version comparison through its version view. In the context of creative coding, all participants agreed on its superior functionality compared to conventional VCS interfaces like *Git*.

Beyond creative coding, the aforementioned complaints remain a valid hurdle, especially as visual side-by-side comparison is unique to creative applications and less beneficial in traditional programming tasks. Nonetheless, several participants agreed on the potential of saving versions automatically. However, this should be done more selectively, e.g., by analysing the saved version history and extracting meaningful versions through heuristics.

In that sense, the *Ghost Editor* arguably emphasized automation too little, leaving the user with a large, unstructured version history to manage manually. However, the general concept of intuitive assistive automation seems to be aligned with the goal of improving the user's workflow seamlessly. In our case, all participants agreed that our editor is more efficient for creative coding than existing solutions, and half of them would use it beyond creative coding. This indicates a clear value, even when comparing the *Ghost Editor* to existing VCSs.

Naturally, it is hard to generalize these results due to the limited sample size and scope, however, they suggest that the *Ghost Framework* achieves its goal of encouraging exploratory behaviour in tools. Obviously, the real-world impact depends on environmental conditions, as certain tool types are more suitable to aid exploration than others. Nonetheless, our results are promising and serve as a starting point for additional research into exploratory tool design.

In the future, the framework should be tested in various domains to understand its value in a broader sense. Additionally, the *Ghost Editor* can serve as continued study to improve the framework by expanding it based on user feedback and evolving the framework accordingly.

6.2.2 RQ1-2: Reflection through Tools

Exploration requires an intuitive understanding of a given solution space. Such an understanding can be built through reflection on past experiments, which requires a practitioner to frequently revisit previous solutions and actively engage with them. However, most conventional tools are designed for a forward-looking linear programming process, and do not encourage this type of re-iteration. The *Ghost Framework* can serve as the foundation for alternative approaches.

Similar to the map analogy presented in section 3.2, the framework encourages the automatic curation of past ideas for effective re-iteration. Naturally, a VCS is the most suitable tool type to test this idea, and the *Ghost Editor* serves as an experimental approach. Specifically, the editor provides access to the precise editing history, and can replay it using the timeline slider. Manually curated versions are presented in the version view.

During our evaluation of the editor, the timeline feature did not receive overly positive feedback. Instead, it was deemed hard to navigate. However, all participants 8 claimed that the manual version view was a helpful overview of past work, and contributed to the understanding of previous ideas. The visual comparison made it easy to spot differences, and trace the evolution of the project. 3 participants specifically noted the benefits of local versioning, allowing for a more specific look at the past project trajectory.

Naturally, the small scale of the evaluation task makes it hard to fully comprehend the reflective value of our editor in the long-term. However, compared to existing tools, it provides a more efficient way to reconsider and compare old versions visually.

Relating these observations back to the *Ghost Framework*, it is hard to say whether it has a positive effect. While the timeline is more representative of the framework's concept, it did not contribute to reflective behaviour in a meaningful way. At the same time, this may

be related to similar weaknesses discussed in subsection 6.2.1, namely the overly exhaustive version history. An improved filtering system might also improve its value.

Concurrently, such a detailed history provides an excellent foundation for further analysis. For instance, the history could be used to extract particularly volatile segments of code, reoccurring sources of errors, and other metrics to improve the user's understanding of their own process. Providing this data in a contextual way might help users to avoid common mistakes.

To conclude, the current state of the *Ghost Editor* does not provide conclusive evidence towards the *Ghost Framework*'s value for process reflection. Presently, our editor fails to leverage the full potential of IAA and the testing scope is insufficient for evaluation. While there is some potential for meaningful improvements, additional work is required to evidently demonstrate this vision. A recommended starting point is the filtering of versions, as well as the addition of further analysis based on the full version history.

Overall, this thesis succeeded in proposing a tool design framework for creative and exploratory tasks. This framework can support efficient solution space exploration, and holds promise for process reflection and documentation. The *Ghost Editor* serves as a commendable first implementation, but reveals several weaknesses in its dedication to intuitive automation. The collected versioning information serves as a promising foundation for future analysis, however, the editor falls short in exploiting of this data. Whether this is a structural issue with the framework itself or implementation-specific is hard to evaluate on the basis of a single example. Yet, we believe that the evidence at hand justifies further exploration of the presented ideas. Both, the construction of further tools based on the proposed framework and the evolution of the *Ghost Editor* itself can yield valuable insights to fully understand the impact of the suggested ideas.

6.3 RQ2: Exploration and Reflection in Version Control

While the first research question discusses the integration of exploratory and reflective processes into tool design in general, **RQ2** targets VCSs specifically. The goal is to understand how VCSs can aid creative practitioners more efficiently, compared to existing tools like *Git*.

The version control system proposed in this thesis is based on the idea of automation. Instead of requiring manual interaction, our system records every change automatically. To use this information as versatile as possible, each line is treated as an individual object

with its own version history, enabling efficient local versioning for any code segment. In theory, this approach provides full flexibility and does not require manual user interaction.

However, in its current state, the *Ghost Editor* is not able to translate the promise of this system into practical value. As discussed in section 6.2, the overly precise version history resulted in frustration during testing, and the theoretical advantages of this method remain unrealized, for both exploration and reflection.

Nonetheless, there are two indications that the fundamental concept is worth further exploration. Firstly, 4 out of 8 participants stated interest in using a modified version of this system on a daily basis. In particular, they requested filtering options to reduce the amount of versions accessible through the timeline based on sensible criteria (e.g., no crashing versions). Secondly, the idea of saving versions with less user interactions is welcome to all 8 participants. Each participant mentioned the automatic generation of version names as a major advantage of the proposed system, as it reduces the barrier to create a new version.

Combining both observations, a future iteration of the *Ghost Editor* should focus on extracting meaningful versions automatically (e.g., by using heuristics and metrics of modification frequency), and simplify access to these versions by means of a more efficient timeline. Doing so enables users to fully focus on writing code, without having to fear the loss of valuable ideas. In line with the *Ghost Framework*, they should retain full control, being able to overwrite the system's suggestions whenever needed.

In its current state, the system still provides significant exploratory value for creative coding, however, it falls short for other applications. At least 2 participants specifically mentioned that they would prefer *Git* when dealing with more traditional programming tasks.

In summary, we were unable to verify the validity of real-time versioning for exploration and reflection based on the *Ghost Editor*. However, we found evidence for its potential benefits, and suggest a further exploration of this topic.

6.4 RQ3: Improvement over Existing Tools

Finally, the last research question addresses the direct comparison with existing tools designed for creative coding, specifically the *P5JS Web Editor*. In this regard, the survey results in Table 5.1 provide a clear conclusion: The *Ghost Editor* improves upon the web editor in 10 out of 13 evaluated categories.

The general verdict of the participants concluded that the *Ghost Editor* essentially provided the same feature set as the web editor, and added some convenience like the *IntelliSense* integration and AI-based code hints. As all of these features aid the development process and have basically no negative effects, the *Ghost Editor* should be preferred over the web editor.

Interestingly, only 2 out of 8 participants noted the version control as a specific reason for this conclusion, and participant 5 even considered it an optional addition rather than an integral part of the experience. Consequently, it seems the somewhat limited editing experience of the web editor was a greater issue during the evaluation tasks than version control. This is likely due to the simple nature of the performed tasks.

As a result, we cannot conclude whether the exploratory and reflective advantages the VCS was supposed to introduce had a significant impact on the editing experience. While the participants did not complain about interferences with their usual workflow, a more elaborate test setup might be required to understand the exact implications of this novel versioning system. Such a long-term test should also include creatively experienced participants to consider their specific needs in more detail, compared to the current examination through computer scientists.

6.5   Further Suggestions

Throughout the interviews, participants provided numerous additional suggestions on how to improve the creative experience of the *Ghost Editor*. However, these suggestions are not directly related to the topic of this thesis, and are only discussed briefly below. They might prove valuable for a future expansion of this project.

Version Quick Select One participant appreciated the ability to switch versions without leaving the editor, and suggested further simplifying the process through a quick-access bar next to the real-time preview. Users could use shortcuts to navigate the latest versions, without interacting with the user interface in the first place. This is a great idea, as it keeps recent modifications always in sight and accessible, increasing iteration speed even further.

Selective Version Display Currently, the version previews always display the whole image, even parts that are not selected by a snapshot specifically. A participant suggested creating an additional preview that only shows elements directly affected by the selected snapshot. This would improve comparability across local versions and

help to evolve individual elements of a complex artwork. However, technologically, it might be challenging to decompose the full project code in a way that is suitable for such a preview.

Coordinate Visualization Several participants struggled to determine the correct coordinates for elements on the canvas. They suggested different methods to support this process, including a labelled grid overlay on top of the preview, and the ability to click on the canvas to select specific coordinates automatically. These solutions would indeed improve the accessibility to novice programmers, and should be considered for a future expansion.

Drag & Drop At least 2 participants proposed the idea of a *drag & drop* system to position elements on the canvas visually, and generate corresponding code from the result. While this is a great idea for learning the *P5JS* library, such a system would be severely limiting when dealing with animation or sound design. However, it is worth considering this feature for educational purposes.

7

Threats To Validity

While the user study was carefully crafted to generate meaningful insights, there are a number of potential threats to the result validity. This section discusses them in detail.

The participant demographics of a user study can have a major influence on its outcome. In the case of this thesis, a high degree of homogeneity could result in one-sided results. All 8 participants have a background in academic computer science, and 6 are specialized in software engineering. Furthermore, all 8 are younger than 30, with 6 being younger than 25. Finally, only one participant was female, while the other 7 were male.

As a result, the presented findings might be biased towards a conventional software engineering mindset, which often conflicts with the exploratory processes observed in creative coding. The lack of experienced creatives in the study could emphasize this even further, especially as all participants had prior experience with alternative versioning systems.

Furthermore, 5 out of 8 participants are affiliated with the same institution as the researchers conducting the study. This could lead to an unconscious bias towards the researcher's work. In combination with the small sample size of participants, this may affect the final conclusions.

The proposed framework was evaluated on the basis of a single practical implementation. The discussion generalizes these observations, however, future implementations based on the *Ghost Framework* might arrive at different conclusions. Similarly, the results of this study only compare two creative coding editors, not version control in particular. While several additional questions were posed to evaluate the proposed VCS, comparisons to tools like *Git* are based on verbal responses only, instead of direct observations.

Additionally, the survey design is limited due to time constraints. While we tried to cover all important aspects of creative coding editors, the selection might be incomplete. Concurrently, the depth of participant responses is affected by their short hands-on experience with both editors. The performed evaluation tasks were rather simple and might not account for the full spectrum of challenges in creative coding. The choice to evaluate the *P5JS Web Editor* first might have introduced further bias, as participants are more comfortable with the *P5JS* library when testing the *Ghost Editor*.

The personal interpretation of the evaluated attributes could pose another challenge to validity. Participants might have a varying understanding of terms such as *usability*, *efficiency*, and *performance*. Using these terms in survey questions can lead to inconsistent observations. Similarly, predefined response options or scales might be interpreted differently across participants.

Finally, the collected results were interpreted in the context of this study. As the evaluation was performed by the designers of the *Ghost Editor* themselves, the final conclusions might suffer bias.

8

Related Work

While the domain of creative coding, and specifically the context of versioning, remains underexplored in academic literature, this thesis is not the first attempt at a creative version control system. In fact, a small but lively body of research has developed around version control for creative applications, and is continuously evolving. This chapter will present some of the most important works, and demonstrate how they differ from our approach.

One of the most comprehensive works on creative version control comes in the form of an empirical study by Sterman et al. (1). In a series of interviews with 18 creative practitioners from diverse domains, they established a model of 4 fundamental approaches to creative versioning, which we introduced in subsection 2.3.2 (see **Palette**, **Freedom**, **Fidelity**, and **Timescale** for more details). Based on their findings, this thesis postulates that creative version control has to resolve a *crisis of confidence* to boost exploratory behaviour. The *Ghost Editor* incorporates this idea into its fundamental design philosophy and serves as a first confirmation of Sterman *et al.*'s results.

A more practical approach is presented by Kery et al. (2). They present *Variolite*, an exploratory versioning tool, that served as inspiration for our *Ghost Editor*'s local versioning approach with an editor-embedded inline user interface. *Variolite* allows to manually record and access versions for any code segment, directly from within the editor itself. Our approach expands on this idea by proactively capturing all versions for each line, and dynamically composing a version history for any code segment from that. As a result, even unsaved version are available. Furthermore, the *Ghost Editor* is specialized for creative coding, while *Variolite* is mainly designed for data scientists.

8. RELATED WORK

A creative-first approach to version control is proposed by Burgess et al. (22). Their "artboard-like" programming environment *Stamper* embeds programming with a node-based graph editor. Assets, code snippets, and results are depicted as individual nodes and the execution flow is determined by visual links. The user can create multiple execution flows in parallel and visualize their outputs in a unified interface. While parallel version visualization is also supported by our *Ghost Editor*, saved versions cannot be modified in parallel. Additionally, *Stamper*'s visual editing process enables the intuitive composition of new versions from prior code snippets. However, the *Ghost Editor* provides a rich version history that enables access to all previous versions. This is impossible in *Stamper* due to visual space limitations. Combining both approaches, e.g., by embedding our versioning system into *Stamper*'s nodes, could be an interesting future project.

Instead of tackling creative coding directly, Ginosar et al. (80) suggest an interesting versioning tool for authoring multi-stage code tutorials. This form of educational code-writing starts with a simple code snippet that is continuously expanded to full complexity, often accompanied by explanatory text, to visualize programming concepts effectively. Their versioning system treats each new version as an individual stage, and is designed to propagate changes through all stages in a single operation. This can be extremely useful to creative coders that want to test an idea across several versions in parallel. While the *Ghost Editor* does not account for multistaged code in its user interface, our versioning system tracks each lines as an individual object, enabling the described behaviour by default. Furthermore, our system provides access to the full change history of each line, not just the saved versions. Expressing this ability in the user interface directly could improve the overall user experience for creative practitioners.

Hartmann et al. (81) present a completely different take on versioning. Instead of addressing temporal version control, e.g., by capturing past versions, they focus on parallel versioning and the exploration of alternatives. Their proposed tool enables the code-driven design of user interfaces (e.g., UI defined by code) and comes with a real-time preview. The user can switch dynamically between different versions of their code and tweak predefined parameters in a control panel without re-compiling the application. The goal is to simplify and speed up iterations. This approach differs significantly from our solution, as it is not concerned with a temporal version history. However, the ability to modify parameters in a control panel could be extremely useful for creative coders that want to test alternative configurations visually. A hybrid approach integrating both ideas is worth considering.

Finally, *GEM-NI* is a radically different approach to creative versioning that does not involve program code directly. Instead, Zaman et al. (82) designed a user interface for generative design, providing a variety of pre-built features that can composed in a node-based graph editor. Parameters are defined via a control panel, and the output is presented in real-time. Based on this editor, *GEM-NI* then provides a rich version history that enables parallel version editing and rapid access to alternatives. Most importantly, *GEM-NI* provides the ability to merge existing versions into new results quickly. A key advantage of *GEM-NI* is the reduced complexity of predefined nodes compared to full source code. As such, the *Ghost Editor* can be understood as an attempt to achieve a similar feature set for programming purposes. Yet, *GEM-NI* can serve as inspiration on how to expand our solution, in particular with regard to merging. The ability to compose different versions together can be extremely powerful to generate new ideas quickly, and is currently missing from the *Ghost Editor*.

To conclude, academic literature provides numerous interesting approaches to the problem of (creative) versioning. Within this vibrant landscape, the *Ghost Editor* contributes the idea of real-time versioning in a tightly integrated framework focused on assistive, automated tooling. While change logs and version tracking exist in other tools, these concepts often lack integration with the editing process itself, serving as a fallback in case of errors. Our solution tries to expand on this idea, embedding a rich version history at the heart of an exploratory process. Yet, our editor can still benefit from other ideas, including version merging and interactive control panels to further improve usability.

9

Conclusion

As creative coding gains momentum in the public conscience, the need for creative support tools has never been higher. Due to the exploratory nature of creative processes, version management is particularly essential to increase efficiency and generate better results. However, the current state of version management in software development is optimized for linear development cycles, and incompatible with creative requirements. This thesis attempts to understand the intricacies of creative coding, and translate them into the design of a novel version control system that embraces the exploratory nature of creative tasks.

Through in-depth analysis of current literature on the challenges of creative coding, the problem was framed as an interface design challenge on top of the mature version control ecosystem for software development. Much of the desired functionality already exists, however, current version control interfaces limit the parallel access capabilities creatives need.

Based on this observation, we contribute a framework for exploratory tool design based on intuitive assistive automation. Instead of creating a passive toolbox, the so-called *Ghost Framework* strives for proactive assistance based on behaviour prediction and the contextual suggestion of solutions. User trust is considered a key to the framework's success, as confidence can inspire users to explore innovative ideas without fear.

The framework was used to design the *Ghost Editor*, a creative code editor with integrated version control. In contrast to existing VCSs, the editor automatically records all changes and enables local version control for any code segment. The intuitive *multilayer interface design* integrates the VCS directly into the editor, and cues relevant interaction based on the current context.

In a final user study, the editor clearly outperformed a popular alternative for creative coding. The results evidently supported the principles of intuitive assistive automation as a way to seamlessly integrate complex interaction into an existing workflow. However, the VCS design suffered from an overly detailed version history. This hurts its practical usability, and should be addressed through automatic version prioritization, doubling down on the assistive nature of the tool.

Beyond this limitation, most participants praised the VCS's usability in the context of creative coding. However, a long-term study would be required to determine its value outside this limited domain.

We believe that the proposed framework provides a solid foundation for the design of powerful tools that can aid both exploratory and conventional programmers alike. By taking responsibility off the developer's shoulders, they can focus on their work freely, and produce better results. In the context of version control, our initial implementation provides a guideline for future adoptions of the framework, while also highlighting several pitfalls and mistakes that should be avoided.

However, the framework only serves as a starting point for exploratory tools, and close attention must be paid to the specific design to ensure compatibility with creative workflows. Additionally, the user study only serves as preliminary evidence for the quality of our results. Due to several limitations in its design and the participant demographics, further research is suggested to validate these observations. In particular, this research should be conducted on alternative tool designs based on our framework to test its generalizability.

In terms of version control for creative coding, the *Ghost Editor* introduces several concepts that could benefit creative practitioners. However, to reach its full potential, several adjustments are recommendable, including version filtering and prioritization.

To conclude, this thesis challenges the traditional notion of version control and pioneers a new path for exploratory tool design. Our novel framework promises a new generation of assistive tools, and we hope that our insights contribute to a more vibrant ecosystem of assistive support tools for all creative minds.

Bibliography

[1] Sarah Sterman, Molly Jane Nicholas, and Eric Paulos. Towards Creative Version Control. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW2):1–25, 2022. doi: 10.1145/3555756. ii, 2, 3, 7, 8, 10, 11, 12, 13, 14, 17, 67, 77

[2] Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, volume 10, pages 3025453–3025626, 2017. v, 8, 13, 17, 34, 35, 77

[3] Ellen Winner. *How art works: A psychological exploration.* Oxford University Press, USA, 2019. 1

[4] Kylie Peppler and Yasmin Kafai. Creative coding: Programming for personal expression. *Retrieved August*, 30(2008):314, 2005. 1, 8

[5] Beau Sheil. Datamation®: Power tools for programmers. In Charles Rich and Richard C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, pages 573–580. Morgan Kaufmann, 1986. ISBN 978-0-934613-12-5. doi: https://doi.org/10.1016/B978-0-934613-12-5.50048-3. URL `https://www.scienced irect.com/science/article/pii/B9780934613125500483`. 2

[6] Mary Beth Kery and Brad A Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29. IEEE, 2017. 2, 5, 6, 17

[7] TRG Green. Programming languages as information structures. In *Psychology of programming*, pages 117–137. Elsevier, 1990.

[8] Amy J Ko and Brad A Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, pages 301–310, 2008. 2

BIBLIOGRAPHY

[9] Maximilian Mayer and Mauricio Verano Merino. Towards version control for creative coding. *Unpublished*, 2023. 3, 11, 13, 18

[10] David W Sandberg. Smalltalk and exploratory programming. *ACM Sigplan Notices*, 23(10):85–92, 1988. 6, 7

[11] Martin Fowler, Jim Highsmith, et al. The agile manifesto. *Software development*, 9 (8):28–35, 2001. 7

[12] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999. 7

[13] Marcel Taeumel, Patrick Rein, and Robert Hirschfeld. Toward patterns of exploratory programming practice. *Design Thinking Research: Translation, Prototyping, and Measurement*, pages 127–150, 2021. 7

[14] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. Micro-versioning tool to support experimentation in exploratory programming. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 6208–6219, 2017. 8, 10, 13, 19

[15] *Code Art*, pages 3–24. Apress, Berkeley, CA, 2007. ISBN 978-1-4302-0310-0. doi: 10.1 007/978-1-4302-0310-0_1. URL `https://doi.org/10.1007/978-1-4302-0310-0_1`. 8

[16] Alex McLean and Geraint A Wiggins. Live coding towards computational creativity. In *ICCC*, pages 175–179, 2010. 8

[17] Brian Logan and Tim Smithers. Creativity and design as exploration. *Modeling creativity and knowledge-based creative design*, pages 139–176, 1993. 8

[18] Deborah K Smith, David B Paradice, and Steven M Smith. Prepare your mind for creativity. *Communications of the ACM*, 43(7):110–116, 2000. 8

[19] Rotem Israel-Fishelson and Arnon Hershkovitz. Studying interrelations of computational thinking and creativity: A scoping review (2011–2020). *Computers & Education*, 176:104353, 2022. 8

[20] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D Patel, Stephen A Edwards, and Edward A Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 137–146, 2008. 8

[21] William F McColl. Scalability, portability and predictability: The bsp approach to parallel programming. *Future Generation Computer Systems*, 12(4):265–272, 1996. 8

[22] Cameron Burgess, Dan Lockton, Maayan Albert, and Daniel Cardoso Llach. Stamper: An artboard-oriented creative coding environment. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–9, 2020. 9, 78

[23] David L Atkins. Version sensitive editing: Change history as a programming tool. In *International Workshop on Software Configuration Management*, pages 146–157. Springer, 1998. 10

[24] E. Carra and F. Pellacini. SceneGit: a practical system for diffing and merging 3D environments. *ACM Transactions on Graphics (TOG)*, 2019. doi: 10.1145/3355089.33 56550. URL `https://dl.acm.org/doi/abs/10.1145/3355089.3356550`. Publisher: dl.acm.org. 10

[25] J. Doboš and A. Steed. 3D revision control framework. *Proceedings of the 17th International Conference on 3D Web Technology, ACM*, pages 121–129, 2012. doi: 10.1145/2338714.2338736. URL `https://dl.acm.org/doi/abs/10.1145/2338714.2 338736`. Publisher: dl.acm.org.

[26] Christian Santoni, Gabriele Salvati, Valentina Tibaldo, and Fabio Pellacini. LevelMerge: Collaborative Game Level Editing by Merging Labeled Graphs. *IEEE Computer Graphics and Applications*, 38(4):71–83, July 2018. ISSN 1558-1756. doi: 10.1109/MCG.2018.042731660. Conference Name: IEEE Computer Graphics and Applications. 10

[27] Sandeep Kaur Kuttal, Anita Sarma, and Gregg Rothermel. History repeats itself more easily when you log it: Versioning for mashups. In *2011 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, pages 69–72. IEEE, 2011. 10

[28] Fabio Zünd, Steven Poulakos, Mubbasir Kapadia, and Robert W. Sumner. Story Version Control and Graphical Visualization for Collaborative Story Authoring. *Proceedings of the 14th European Conference on Visual Media Production (CVMP 2017)*,

10:10, 2017. doi: 10.1145/3150165.3150175. URL `https://doi.org/10.1145/3150` `165.3150175`. 10

[29] H. T. Chen, L. Y. Wei, and C. F. Chang. Nonlinear revision control for images. *ACM Transactions on Graphics (TOG)*, 30(4):105, 2011. doi: 10.1145/2010324.1965000. URL `https://dl.acm.org/doi/abs/10.1145/2010324.1965000?casa_token=WJ1Y` `gJS9dWIAAAAA:DZwkkIavyYH5vq1ShwEWwuOoh2XUgv7PqPNp5niZzYCBCnf4_sW7O_dM2p` `1dovumzyRKaO9zwIA3`. Publisher: ACM. 11

[30] S. Baltes, F. Hollerich, and S. Diehl. Round-trip sketches: Supporting the lifecycle of software development sketches from analog to digital and back. *2017 IEEE Working Conference . . .*, 2017. URL `https://ieeexplore.ieee.org/abstract/document/80` `91190/`. Publisher: ieeexplore.ieee.org. 10

[31] Marc J Rochkind. The source code control system. *IEEE transactions on Software Engineering*, (4):364–370, 1975. 10

[32] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. Version Control System: A Review. *Procedia Computer Science*, 135(CSCW2, Article 336):408–415, 2018. ISSN 1877-0509. doi: 10.1016/j.procs.2018.08.191. URL `https://doi.org/10.1016/j.pr` `ocs.2018.08.191`. 11

[33] NB Ruparelia. The history of version control. *ACM SIGSOFT Software Engineering Notes*, 35(1):5–9, 2010. ISSN 0163-5948. doi: 10.1145/1668862.1668876. URL `https://dl.acm.org/doi/pdf/10.1145/1668862.1668876?casa_token=eIe5LCq4` `7bEAAAAA:PJau_GfyRj1qchCKCgwLrTft_dd71fGL6mpj-NSzjJBIGKKEzDBOwRQb4RAFmh` `CPGz9yCcYyn972XQ`. Publisher: dl.acm.org Type: PDF. 11, 19

[34] Santiago Perez De Rosso and Daniel Jackson. What's wrong with git? a conceptual design analysis. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, Onward! 2013, pages 37–52, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509584. URL `https://doi.org/10.114` `5/2509578.2509584`. 11, 67

[35] Santiago Perez De Rosso and Daniel Jackson. Purposes, concepts, misfits, and a redesign of git. *ACM SIGPLAN Notices*, 51(10):292–310, 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984018. URL `https://doi.org/10.1145/3022671.2984018`. 11, 19, 67

[36] Hsiang-Ting Chen, Li-Yi Wei, Björn Hartmann, and Maneesh Agrawala. Data-driven adaptive history for image editing. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 103–111, 2016. 11

[37] Fabio Calefato, Giovanna Castellano, and Veronica Rossano. Recode: revision control for digital images. *Multimedia Tools and Applications*, 78:33169–33188, 2019. 11

[38] David Jones, Aydin Nassehi, Chris Snider, James Gopsill, Peter Rosso, Ric Real, Mark Goudswaard, and Ben Hicks. Towards integrated version control of virtual and physical artefacts in new product development: inspirations from software engineering and the digital twin paradigm. *Procedia CIRP*, 100:283–288, 2021. 11, 13

[39] Michael Terry and Elizabeth D Mynatt. Recognizing creative needs in user interface design. In *Proceedings of the 4th Conference on Creativity & Cognition*, pages 38–44, 2002. 13

[40] Andrés Felipe Gómez, Jean Pierre Charalambos, and Andrés Colubri. Shaderbase: A processing tool for shaders in computational arts and design. In *VISIGRAPP (2: IVAPP)*, pages 191–196, 2016. 13

[41] Cynthia J Gormley and Samantha J Gormley. Data hoarding and information clutter: The impact on cost, life span of data, effectiveness, sharing, productivity, and knowledge management culture. *Issues in Information Systems*, 13(2):90–95, 2012. 13

[42] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. Generating photo manipulation tutorials by demonstration. In *ACM SIGGRAPH 2009 papers*, pages 1–9. 2009. 13

[43] Jens Lincke, Robert Krahn, Dan Ingalls, Marko Roder, and Robert Hirschfeld. The lively partsbin–a cloud-based repository for collaborative development of active web content. In *2012 45th Hawaii International Conference on System Sciences*, pages 693–701. IEEE, 2012. 13

[44] Blair Subbaraman and Nadya Peek. p5. fab: Direct control of digital fabrication machines from a creative coding environment. In *Designing Interactive Systems Conference*, pages 1148–1161, 2022. 14

[45] Joel Chan and Christian D Schunn. The importance of iteration in creative conceptual combination. *Cognition*, 145:104–115, 2015. 17

[46] Alex Doboli and Anurag Umbarkar. The role of precedents in increasing creativity during iterative design of electronic embedded systems. *Design Studies*, 35(3):298–326, 2014.

[47] David C Wynn and Claudia M Eckert. Perspectives on iteration in design and development. *Research in Engineering Design*, 28:153–184, 2017. 17

[48] Robert Blumberg and Shaku Atre. The problem with unstructured data. *Dm Review*, 13(42-49):62, 2003. 19

[49] Carsten Mohs, Jörn Hurtienne, Martin Christof Kindsmüller, Johann Habakuk Israel, Herbert A Meyer, et al. Iuui–intuitive use of user interfaces: Auf dem weg zu einer wissenschaftlichen basis für das schlagwort „intuitivität". *MMI-Interaktiv*, 11(11):75–84, 2006. 20

[50] Alethea Blackler and Vesna Popovic. Towards intuitive interaction theory, 2015. 20

[51] Alethea Blackler, Vesna Popovic, and Douglas Mahar. The nature of intuitive use of products: an experimental approach. *Design Studies*, 24(6):491–506, 2003.

[52] Dennis C Neale and John M Carroll. The role of metaphors in user interface design. In *Handbook of human-computer interaction*, pages 441–462. Elsevier, 1997. 27

[53] John M Carroll, Robert L Mack, and Wendy A Kellogg. Interface metaphors and user interface design. In *Handbook of human-computer interaction*, pages 67–85. Elsevier, 1988. 20, 27

[54] Ana Viseu. A multidisciplinary approach to the mutual shaping process in electronic identities or "we shape the tools and thereafter they shape us" mcluhan. *Preprint*, 1999. 20

[55] Marie-Hélène Raidl and Todd I Lubart. An empirical study of intuition and creativity. *Imagination, Cognition and Personality*, 20(3):217–230, 2001. 20

[56] Emma Policastro. Creative intuition: An integrative review. *Creativity Research Journal*, 8(2):99–113, 1995.

[57] Judit Pétervári, Magda Osman, and Joydeep Bhattacharya. The role of intuition in the generation and evaluation stages of creativity. *Frontiers in Psychology*, 7:1420, 2016. 20

[58] Deep Ganguli, Danny Hernandez, Liane Lovitt, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova Dassarma, Dawn Drain, Nelson Elhage, et al. Predictability and surprise in large generative models. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, pages 1747–1764, 2022. 23

[59] Jaap-Henk Hoepman. Privacy design strategies. In *IFIP International Information Security Conference*, pages 446–459. Springer, 2014. 25

[60] Valentin Zieglmeier and Alexander Pretschner. Trustworthy transparency by design. *arXiv preprint arXiv:2103.10769*, 2021. 25

[61] Qinggang Meng and Mark H Lee. Design issues for assistive robotics for the elderly. *Advanced engineering informatics*, 20(2):171–186, 2006. 25

[62] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004. 26

[63] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016. 26

[64] Ben Shneiderman. Creating creativity: user interfaces for supporting innovation. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):114–138, 2000. 26

[65] Ben Shneiderman. Creativity support tools. *Communications of the ACM*, 45(10): 116–120, 2002.

[66] Ben Shneiderman. Creativity support tools: accelerating discovery and innovation. *Communications of the ACM*, 50(12):20–32, 2007. 26, 27

[67] Ben Shneiderman. Promoting universal usability with multi-layer interface design. *ACM SIGCAPH Computers and the Physically Handicapped*, (73-74):1–8, 2002. 26

[68] Bryan Clark and Jeanna Matthews. Deciding layers: Adaptive composition of layers in a multi-layer user interface. In *Proceedings of 11th International Conference on Human-Computer Interaction*, volume 7, 2005. 26

[69] Aaron Marcus. Metaphor design in user interfaces. *ACM SIGDOC Asterisk Journal of Computer Documentation*, 22(2):43–57, 1998. 27

[70] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, pages 2389–2401, 2022. 39

[71] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 709–712. IEEE, 2015. 39

[72] Alvy Ray Smith and Eric Ray Lyons. Hwb—a more intuitive hue-based color model. *Journal of graphics tools*, 1(1):3–17, 1996. 40

[73] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7, 2022. 41

[74] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *arXiv preprint arXiv:2304.02195*, 2023.

[75] Zachary Englhardt, Richard Li, Dilini Nissanka, Zhihan Zhang, Girish Narayanswamy, Joseph Breda, Xin Liu, Shwetak Patel, and Vikram Iyer. Exploring and characterizing large language models for embedded system development and debugging. *arXiv preprint arXiv:2307.03817*, 2023. 41

[76] James J Hunt, Kiem-Phong Vo, and Walter F Tichy. An empirical study of delta algorithms. In *International Workshop on Software Configuration Management*, pages 49–66. Springer, 1996. 52

[77] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006. 56

[78] Paul De Palma. Why women avoid computer science. *Communications of the ACM*, 44(6):27–30, 2001. 56

[79] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15Th european conference on software maintenance and reengineering*, pages 181–190. IEEE, 2011. 67

[80] Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, and Bjorn Hartmann. Authoring multi-stage code examples with editable code histories. In *Proceedings of*

the 26th annual ACM symposium on User interface software and technology, pages 485–494, 2013. 78

[81] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 91–100, 2008. 78

[82] Loutfouz Zaman, Wolfgang Stuerzlinger, Christian Neugebauer, Rob Woodbury, Maher Elkhaldi, Naghmi Shireen, and Michael Terry. Gem-ni: A system for creating and managing alternatives in generative design. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, pages 1201–1210, 2015. 79

Appendix A

Survey

This appendix includes the full survey for participants of the user study. It includes the preamble and answering options. The original survey was created with *Google Forms*.

Creative Coding Editor User Study

This survey is part of Maximilian Mayer's Master Thesis in Computer Science at the Vrije Universiteit Amsterdam. For this thesis, a code editor for creative coding was developed, incorporating novel versioning concepts, as well as a real-time preview for the P5JS library, and many other convenience features.

In this first section, we will provide some background information for this survey, and explain how we will use your data. Please read the following information carefully.

Goal and Nature of Data

The goal of this survey is to evaluate said editor. This will be done in the form of a comparison with a baseline, the well-established P5JS Web Editor. First, you will test this baseline editor, and fill out a portion of this survey. Then, you will test the Ghost Editor, and fill out the same questions again. The questions will ask you about your experience with the editors, and are designed to gain meaningful insights for this research project.

Some of these questions are of personal nature, including information about age, gender, education, and profession. Answering these questions is not required, but helps to gain a deeper understanding of the participant demographics for this study, and can be beneficial to the overall results. Thus, we would be glad if you chose to provide this information.

Other questions refer to the personal user experience while using the editors. This data can, and should, be opinionated. After all, the goal is to evaluate the user experience, and every user will have a different experience.

Data Collection and Processing

All data collected in this survey is anonymous, and there is no way for the researchers to relate it back to the individual taking this survey. The anonymous dataset will be used by us to understand the impact of our efforts, and processed only to deepen this understanding. Processing may include statistical tests to identify trends in the data, the creation and interpretation of graphs, and other research-related practices.

This final results may be available to the public as part of the final Master Thesis, as well as further research proceedings based on the Master Thesis. This may include the raw data provided (e.g., a spreadsheet containing the full data set, quotes of specific answers as examples in the thesis text), or processed forms of it (e.g., graphs depicting the survey demographics, classifications derived from text answer).

Finally, please consider that this survey uses Google Forms, a service provided by Google, and thus, an American company. The US Government has the right to forcefully acquire all data on Google servers, if they desire to do so.

Consent and Contact

If you do not wish that your data is part of this survey, please do not submit any data, as it might prove labor-intensive to remove your specific answers from the final data set. If you submit data to this survey, you consent to us using it in the ways outlined above.

If you have any questions or other requests, please do not hesitate to contact Maximilian Mayer directly, using this email: m.mayer@student.vu.nl

Otherwise, thank you for your participation!

Personal Information

Let's start with some personal questions! If you feel comfortable about it, tell us a bit about yourself, your background, and how you ended up in this study.

Note: The choice questions are marked as mandatory. If you prefer to not say, just choose "Prefer not to say". That simplifies the final data extraction process for us.

Question 1: Age

Answering options:

1. 18 - 25
2. 25 - 30
3. 30 - 35
4. 35 - 40
5. 40 - 45
6. 45 - 50
7. 50 - 55
8. 55 - 60
9. 60+
10. Prefer not to say

Question 2: Gender

Answering options:

1. Female
2. Male
3. Non-Binary
4. Prefer not to say

Question 3: Highest Completed Education

Answering options:

1. None
2. (High) School Education
3. Bachelor (Computer Science)
4. Bachelor (Arts/Design)
5. Bachelor (Other)
6. Master (Computer Science)
7. Master (Arts/Design)
8. Master (Other)
9. PhD (Computer Science)
10. PhD (Arts/Design)
11. PhD (Other)
12. Prefer not to say
13. Other

Question 4: Currently Studying

Answering options:

1. None
2. (High) School Education
3. Bachelor (Computer Science)
4. Bachelor (Arts/Design)
5. Bachelor (Other)
6. Master (Computer Science)
7. Master (Arts/Design)
8. Master (Other)
9. PhD (Computer Science)
10. PhD (Arts/Design)
11. PhD (Other)
12. Prefer not to say
13. Other

Question 5: Current Profession

Answering options:

1. None
2. Studying
3. Working in Industry (Computer Science)
4. Working in Industry (Arts/Design)
5. Working in Academia (Computer Science)
6. Working in Academia (Arts/Design)
7. Prefer not to say
8. Other

Question 6: Have you programmed before?

Answering options:

1. Never
2. Occasionally
3. Sometimes
4. Often
5. Prefer not to say

Question 7: How experienced are you with Computer Science, and specifically programming?

Answer on a scale from 0 to 10, with 0 meaning *No Experience* and 10 being *Highly Skilled.*

Question 8: Please describe your previous experience with programming and Computer Science in some more detail

Long-form text answer.

Question 9: Have you done creative coding before?

Answering options:

1. Never

2. Occasionally

3. Sometimes

4. Often

5. Prefer not to say

Question 10: How familiar are you with Creative Coding, and how much experience do you have?

Answer on a scale from 0 to 10, with 0 meaning *No Experience* and 10 being *Highly Skilled.*

Question 11: Please describe your previous experience with Creative Coding in some more detail

Long-form text answer.

Question 12: Anything else you think would help us with our research?

Long-form text answer.

Editor Evaluation

This section has to be filled out once for each editor.

In this section, you will answer some questions for the reference editor, the P5JS Web Editor. This editor is developed by the creators of the P5JS library, that is also at the heart of our own editor. As such, it is very prominent on their web page, and for many people, the first introduction to Creative Coding with P5JS.

To test this editor, you first perform some tasks to get acquainted with using the editor. Then you will answer the questions below. You will answer the same questions later for our own editor later as well. However, please answer these questions BEFORE testing the other editor, to ensure you are not biased by the comparative experience.

Most questions will require you to rate your experience on a scale from 0 to 10. In this case, 0 refers to the lowest possible rating, and is equivalent with a terrible user experience that would lead you to search for an alternative solution. 10 refers to an outstanding experience that leaves nothing to desired, and you would use the editor again at any time. A score of 5 means that you were not bothered by the experience, but you would not recommend it either.

Consider that most of these questions only refer to some small part of the overall experience. The answers don't have to reflect your overall decision on whether you would use the editor or not. Specific comments can be added at the end of this section, if you want to highlight certain elements that were particularly good or bad.

Instructions on how to test the editor will be given by the survey leader.

The editor can be found here: `https://editor.p5js.org/`
Additional references for P5JS: `https://p5js.org/reference/`

Please note:
To save sketches with this editor, you have to sign up on their page. You can also sign in with GitHub or Google. While this is recommended, you can also just save the code to files on your machine manually. If you do so, this should not affect your evaluation of the experience in general, though, as this is a limitation of the design as a web service.

Usability

Question 1: How was the installation/startup process of the editor?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Question 2: How would you rate your first impression? Was everything as expected, or did you miss/confuse something?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Question 3: How was the experience of creating your first program? Were you held up by something, or could you start immediately?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Question 4: How intuitive was the overall user interface (UI) of the editor? Did you find everything where you expected it?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Features

Question 1: How was the experience of editing code? Did you find all features you would expect?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Question 2: Did the preview behave as expected? Did it benefit or hinder your development process?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Question 3: Did you find all features you expect from a fully functional code editor? Did they work as expected?

Answer on a scale from 0 to 10, with 0 meaning *Unusable* and 10 being *Fully Featured*.

Question 4: How helpful was the editor in case of errors? Did you feel well-supported in debugging your code?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Question 5: Creating and managing many different versions is considered a crucial workflow for creative coding. Did the editor support you in this?

Answer on a scale from 0 to 10, with 0 meaning *No Support* and 10 being *Exceptional Support*.

Performance

Question 1: How did the performance of the editor influence the experience?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Question 2: How smooth and bug-free was the experience?

Answer on a scale from 0 to 10, with 0 meaning *Very Rough* and 10 being *Completely Smooth*.

Question 3: How stable did the editor run?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Learning Curve

Question 1: How quickly were you able to learn how to use the code editor effectively?

Answer on a scale from 0 to 10, with 0 meaning *Steep Learning Curve* and 10 being *Very Easy*.

Question 2: Would you recommend this code editor to beginners in the field of creative coding?

Answering options:

1. No, to no one.

2. Only with solid background knowledge.

3. To those with some experience.

4. Yes, to everyone.

5. Other (additional text option)

Overall

Question 1: How would you rate the overall experience of using the editor?

Answer on a scale from 0 to 10, with 0 meaning *Terrible* and 10 being *Exceptional*.

Open Questions

Question 1: Briefly describe your experience of using the editor.

Long-form text answer.

Question 2: What were the best features of the editor?

Long-form text answer.

Question 3: What did not work as expected?

Long-form text answer.

Question 4: What would you wish for to improve this editor?

Long-form text answer.

Question 5: Whom would you recommend this editor to?

Long-form text answer.

Ghost Editor: Feature Questions

In this section, we added some more specific questions towards special features that only our editor supports. They are designed to help us understand if they provide any additional value for users.

Again, remember: Most questions will require you to rate your experience on a scale from 0 to 10. In this case, 0 refers to the lowest possible rating, and is equivalent with a terrible user experience that would lead you to search for an alternative solution. 10 refers to an outstanding experience that leaves nothing to desired, and you would use the editor again at any time. A score of 5 means that you were not bothered by the experience, but you would not recommend it either. **Specifically, 5 also represents a lack of experience, e.g., if you did not use the feature at all.**

Consider that most of these questions only refer to some small part of the overall experience. The answers don't have to reflect your overall decision on whether you would use the editor or not. Specific comments can be added at the end of this section, if you want to highlight certain elements that were particularly good or bad.

Snapshots

Question 1: How often did you create snapshots of code blocks?

Answering options:

1. Never

2. Occasionally

3. Sometimes

4. Often

Question 2: How did you like the visual representation of snapshots in your code? Was it helpful or not?
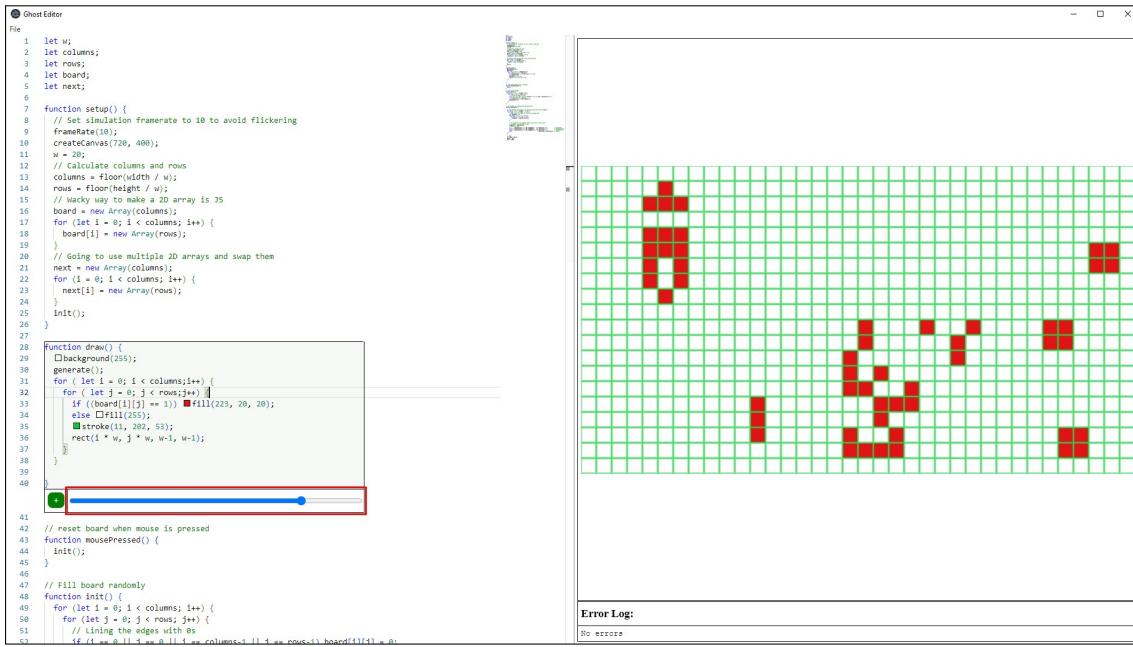
Answer on a scale from 0 to 10, with 0 meaning *Distracting* and 10 being *Beneficial*.

Question 3: Did the visual representation of these snapshots in the code impact your workflow?

Long-form text answer.

Timeline

Question 1: How often did you use the timeline feature to access previous versions for a snapshot/block of code?

Answering options:

1. Never
2. Occasionally
3. Sometimes
4. Often
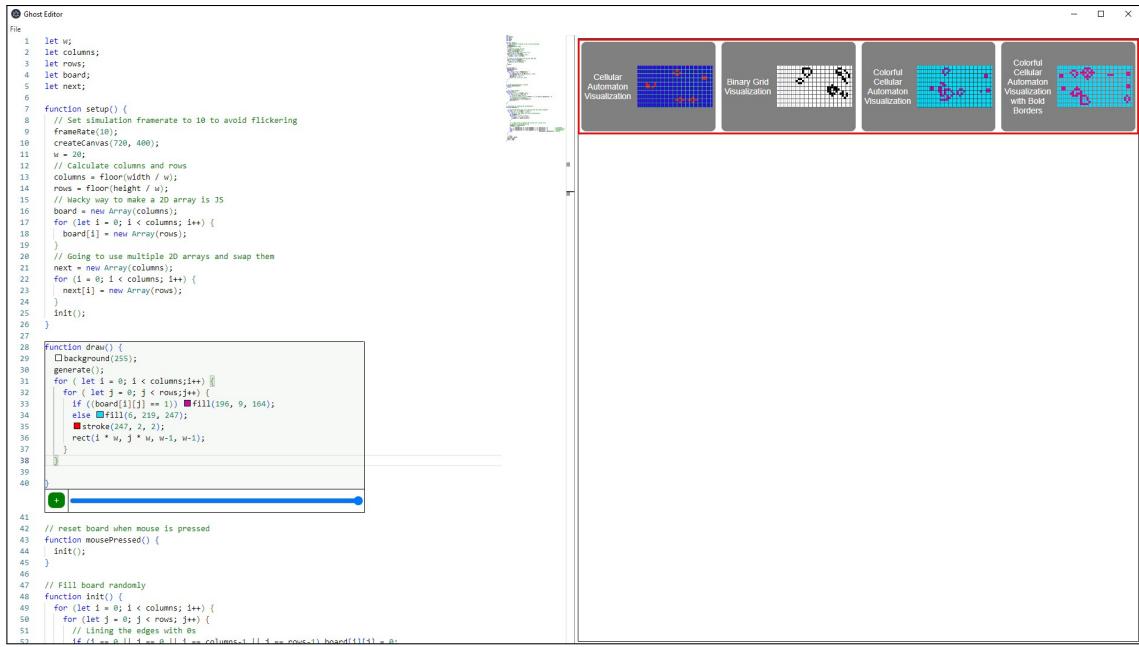
Question 2: How useful was the timeline to your workflow?

Answer on a scale from 0 to 10, with 0 meaning *Useless* and 10 being *Extremely Valuable.*

Question 3: How did you use the timeline, and how did that impact your workflow?

Long-form text answer.

Version View

Question 1: How often did you create saved versions and accessed them in the separate version view?

Answering options:

1. Never

2. Occasionally

3. Sometimes

4. Often

Question 2: How useful was the separate version view for you to compare versions and access old code snippets?
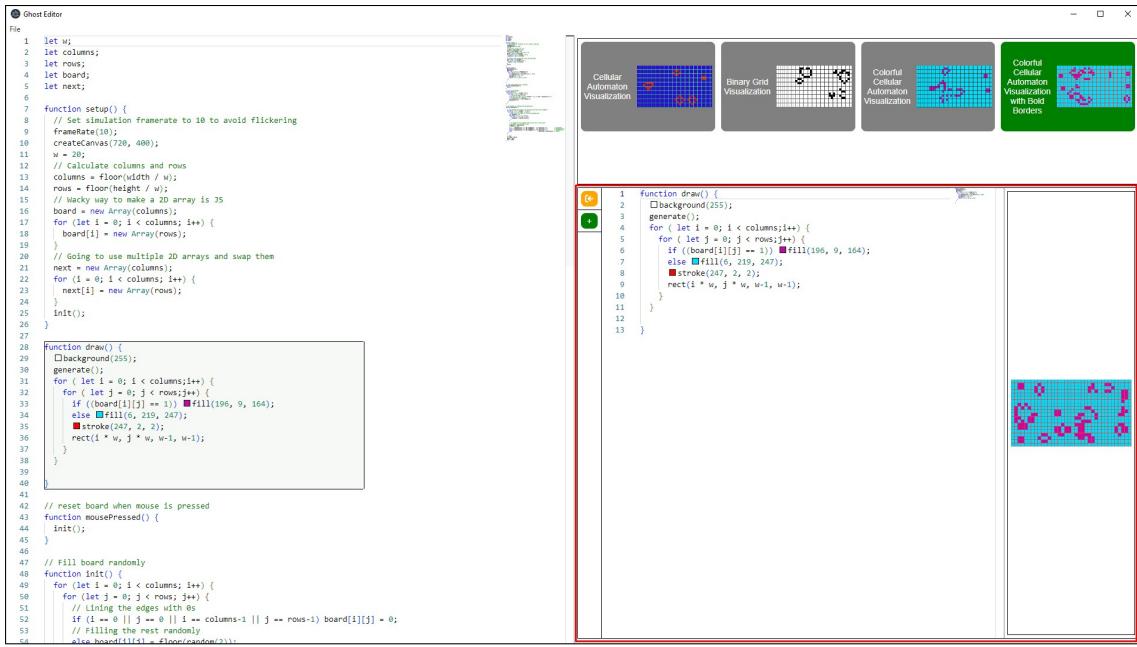
Answer on a scale from 0 to 10, with 0 meaning *Useless* and 10 being *Extremely Valuable.*

Question 3: How did you use the version view, and how did that impact your workflow?

Long-form text answer.

Version Editor

Question 1: How often did you use the version editor over the main editor to edit a specific version?

Answering options:

1. Never
2. Occasionally
3. Sometimes
4. Often
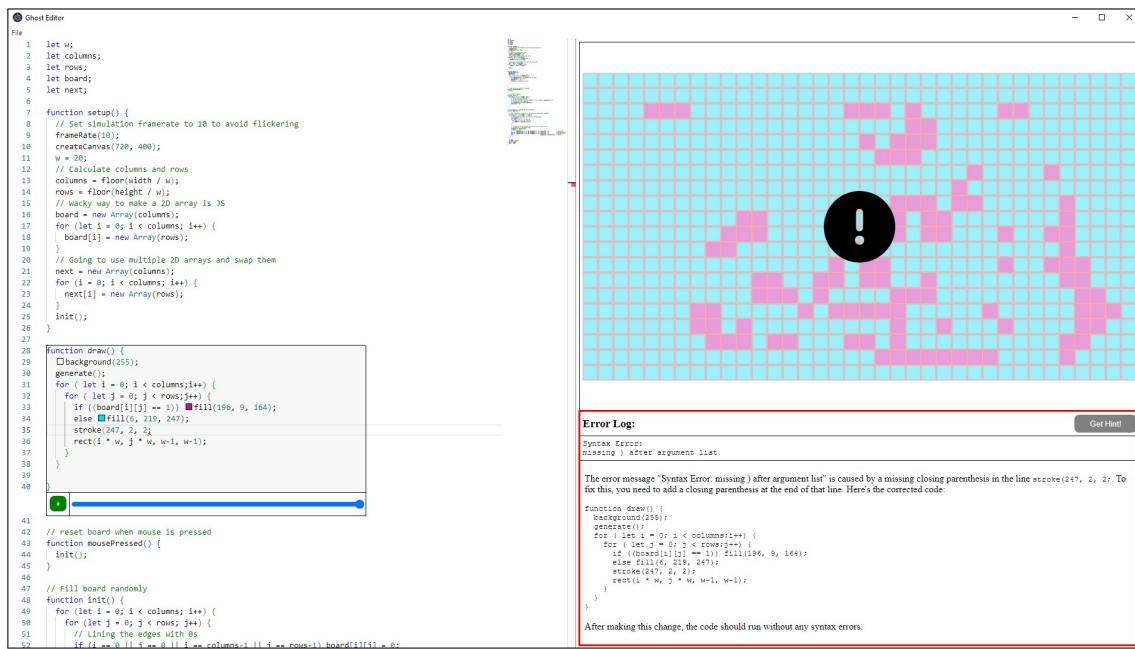
Question 2: How useful was the version editor, e.g., to focus on a smaller code segment when editing a version?

Answer on a scale from 0 to 10, with 0 meaning *Useless* and 10 being *Extremely Valuable.*

Question 3: How did you use the version editor, and how did that impact your workflow?

Long-form text answer.

Error Hints

Question 1: How often did you use the error hint feature to explain an error message in more detail?

Answering options:

1. Never

2. Occasionally

3. Sometimes

4. Often

Question 2: How useful were the error hints to you?

Answer on a scale from 0 to 10, with 0 meaning *Useless* and 10 being *Extremely Valuable.*

Question 3: How did error hints influence your workflow?

Long-form text answer.

Comparison and Final Thoughts

In this final section, you will now ask you to describe briefly about your overall opinion on the two editors, and how they compared for your workflow.

Question 1: Briefly compare your experience between the first and second editor.

Long-form text answer.

Question 2: What is your overall opinion on the presented editors after using them both? Do you have any specific thoughts you would like us to know for any future development?

Long-form text answer.

Question 3: Anything else you would like to let us know?

Long-form text answer.

That's it! Thank you so much for your time!

Your insights help us a great deal in improving our work in the future! Without you, this research would not have been possible. So thank you a lot for taking the time, and filling out this survey.

Now, all you have to do is click send in your results by clicking the button below! Again, should you have any questions, do not hesitate to contact Maximilian Mayer under this e-mail address:

m.mayer@student.vu.nl

I hope you have a great day! Thanks, and best regards,

Maximilian Mayer

Appendix B

Interview Questions

The following catalogue of questions was used as a baseline for interviews. However, due to their semi-structured nature, interviews evolved dynamically and did not always abide to this structure.

1. Which editor did you prefer?

   (a) How was the general experience of using the editor?

   (b) Was there any feature that stood out?

      i. How and why?

   (c) What was missing?

      i. How would you improve that?

   (d) How was the general experience of using the other editor?

      i. What made the other editor worse?

         A. How and why?

         B. How was the chosen editor better at these points?

      ii. What would have to change in the worse editor?

2. Did you use the snapshots feature in the Ghost Editor?

   (a) How and why?

   (b) What was useful?

      i. Specifically, the timeline feature?

      ii. Specifically, the version view feature?

      iii. Specifically, the version editor feature?

    (c) What was distracting/bad?

        i. Specifically, the timeline feature?

        ii. Specifically, the version view feature?

        iii. Specifically, the version editor feature?

    (d) Would you use the feature(s) outside creative coding?

        i. How and why?

3. Would you use either editor (as a whole) outside creative coding?

    (a) How and why?

    (b) What could be improved to make the editor(s) more suitable?