NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO PEDRO BRANCO DOS SANTOS

BSc in Computer Science

USER INTERFACE FOR LIVE PROGRAMMING APPLICATION CONSTRUCTION

MASTER IN COMPUTER SCIENCE AND ENGINEERING

SPECIALIZATION IN MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

September, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

USER INTERFACE FOR LIVE PROGRAMMING APPLICATION CONSTRUCTION

JOÃO PEDRO BRANCO DOS SANTOS

BSc in Computer Science

Adviser: João Ricardo Viegas da Costa Seco
Associate Professor, NOVA University of Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING
SPECIALIZATION IN MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2025

User Interface for Live Programming Application Construction

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.2.5) [26].

Acknowledgements

First, I would like to thank my family for their unconditional support throughout this journey. Their encouragement and belief in me were the main reason I was able to carry on. I would also like to thank my dog, Kiko, for keeping me company during long hours of work and always brightening my mood with his unconditional love.

I am deeply grateful to my friends, who gave me a way to unwind and provided much-needed distractions during stressful times. Their companionship and understanding were invaluable. A special thanks goes to Daniel Macau, who not only helped me with the technical aspects of my work but also provided motivation when I needed it most.

I would like to thank my adviser, Professor João Costa Seco, for his guidance and support throughout this project.

Finally, I would like to express my gratitude to Professor João Lourenço for creating this LaTeX template, which greatly facilitated the writing process and without which this document would not have been possible.

Abstract

In today's technology-driven society, as software becomes ubiquitous in everyday life, fostering new approaches to software development is essential for improving efficiency, quality, and usability. Traditional live programming environments often pose challenges: they are difficult for novices to approach and offer limited support for reactivity and incremental development. While live programming provides real-time feedback, most systems remain constrained by code-centric interaction models.

This thesis presents the design and implementation of a live programming environment that innovates data-centric application development through visual interaction. The system enables applications to be constructed and evolved incrementally, intertwining immediacy with dual modes of interaction: direct manipulation of visual elements and a command console for textual development. Program elements such as variables, actions, definitions, and HTML views are represented as nodes in a directed acyclic graph, making dependencies explicit and navigable. This visual composition is tightly coupled with real-time execution, so that every modification is instantly reflected in the running application.

The environment emphasizes usability and clarity through organization, filtering, and search mechanisms, while maintaining expressiveness for advanced users. By abstracting away the syntax of the underlying language, it allows developers to work productively without prior familiarity with it, while still rewarding general programming knowledge.

A user study with computer science students demonstrated the effectiveness of the approach: participants successfully built applications, reported high satisfaction and ease of use, and completed tasks with confidence.

This thesis contributes to the fields of live programming and user-centered design by presenting a scalable environment that lowers the barrier to application development while broadening how applications can be created, tested, and refined.

Keywords: live programming, GUI, user-centered design, visual programming, reactivity

Resumo

Na sociedade atual, impulsionada pela tecnologia, à medida que o software se torna ubíquo, novas abordagens ao desenvolvimento são essenciais para melhorar eficiência, qualidade e usabilidade. Os ambientes tradicionais de live programming apresentam desafios: são de difícil acesso para principiantes e oferecem suporte limitado à reatividade e ao desenvolvimento incremental. Apesar de fornecerem feedback em tempo real, a maioria continua centrada no código.

Esta dissertação apresenta o desenho e implementação de um ambiente de live programming que inova o desenvolvimento de aplicações data-centric por interação visual. Permite construir e evoluir aplicações de forma incremental, combinando imediatidade com dois modos de interação: manipulação direta de elementos visuais e uma consola de comandos para desenvolvimento textual. Elementos como variáveis, ações, definições e vistas HTML são representados como nós num grafo acíclico direcionado, tornando dependências explícitas e navegáveis. A composição visual está acoplada à execução em tempo real, refletindo instantaneamente cada modificação na aplicação.

O ambiente enfatiza usabilidade e clareza com mecanismos de organização, filtragem e pesquisa, mantendo expressividade para utilizadores avançados. Ao abstrair a sintaxe subjacente, permite que programadores trabalhem produtivamente sem conhecimento prévio, embora beneficiando de conhecimentos gerais de programação.

Um estudo com estudantes de informática demonstrou a eficácia da abordagem: os participantes construíram aplicações com sucesso, relataram elevada satisfação e facilidade de utilização e completaram tarefas com confiança.

Esta dissertação contribui para live programming e design centrado no utilizador, apresentando um ambiente escalável que reduz barreiras ao desenvolvimento de aplicações e amplia as possibilidades de criação, teste e aperfeiçoamento destas.

Palavras-chave: programação ao vivo, interface gráfica, design centrado no utilizador, programação visual, reatividade

Contents

List of Figures

Listings

Acronyms

DAG Directed Acyclic Graph *(pp. 4, 12, 30, 33, 35, 48, 72)*

GOD Goal-Oriented Design *(p. 14)*
GUI Graphical User Interface *(pp. 1, 4, 7, 13, 14, 20, 21, 35, 71)*

IDE Integrated Development Environment *(pp. 7, 9, 33)*

LP Live Programming *(pp. 1–4, 6–9, 11, 15, 18–20, 22, 27, 32, 33)*

UCD User-Centered Design *(pp. 4, 13–17, 34)*
UI User Interface *(p. 29)*
UX User Experience *(pp. 13, 14, 16)*

VR Virtual Reality *(p. 36)*

1 Introduction

1.1 Motivation and Context

Software has become ubiquitous in modern society, supporting activities that range from professional tasks to everyday domestic routines. This ubiquity has led to a growing demand for development tools that enable the creation of reliable software in a faster and more accessible manner. Traditional development practices generally follow the *code-compile-deploy* cycle [9], in which developers write code, compile it, and execute it to observe the resulting behavior. Although this cycle is standardized and efficient [42, 4], it introduces a temporal and cognitive gap between the modifications performed by the programmer and their effect on the application, which can hinder productivity and slow down experimentation.

In addition to limiting rapid prototyping, the code-compile-deploy cycle presents difficulties in the continuous evolution of software. As applications mature, they must be frequently adapted to fix defects, extend functionality, or respond to changes in the technological environment. These updates often create risks of service disruption and inconsistencies between the application's logic and its persistent state. Maintaining consistency typically requires additional mechanisms, which increase system complexity and may reduce overall reliability.

Live Programming (LP) [33] has emerged as a paradigm intended to address such challenges by ensuring that programs remain continuously running while providing real-time feedback as modifications are made. This paradigm reduces the latency between programmer actions and system responses, fostering a closer alignment between program logic and observable behavior. As a result, LP has been shown to improve productivity, support exploratory development, and enhance understanding of system dynamics.

This thesis is developed in the context of the Meerkat project, a research effort aimed at rethinking how modern applications are built. Meerkat is a tierless, reactive, and LP language designed to support the safe and incremental construction of interactive systems [36, 9]. Its central vision is that applications should be developed as live programs, where a Graphical User Interface (GUI) is automatically generated from underlying

services,whether local or cloud-based, each maintaining its own data storage. By following reactive programming principles, Meerkat models applications as dataflows in which changes propagate automatically, enabling systems that respond dynamically to user interactions and data updates.

Our team has contributed to this broader project in multiple ways: some members have worked on the language's semantics and consistency [13], others have explored prototype implementations in Go and Rust [40, 52], and additional efforts have extended Meerkat into distributed environments [48]. What remains underdeveloped, however, is a cohesive visual environment capable of integrating these components in an intuitive, interactive manner and making them accessible to both programmers and non-programmers.

To address this gap, our contribution is a live visual programming environment that brings Meerkat's live programming model into a direct manipulation interface. By replacing textual programming with an intuitive visual workspace, users can construct, modify, and observe application behavior through visual building blocks such as data structures, views, actions, and modules. The system models application logic as a reactive, directed acyclic graph, where edges represent dependencies between components. While this environment removes the need to learn textual syntax and clarifies program structure, users still engage with core concepts such as data types and structures, shifting the focus toward conceptual modeling, reducing syntax errors, and providing immediate visual feedback.

1.2 Problem

While the principles of reactivity and LP have clear benefits in the development of software due to continuous execution of programs and real-time feedback for developers, most LP environments utilize an interaction model that is primarily textual. This creates a strong dependence on knowledge of programming syntax, which makes such systems less accessible to users with limited programming experience and creates a barrier to broader adoption of the paradigm. Conversely, existing visual environments either target introductory programming education [45] or focus on narrow application domains, offering limited applicability in general-purpose software development. This gap highlights the need for a visual LP environment that can support the construction of complete and reactive applications.

Developing such an environment, however, presents significant challenges. One difficulty lies in reconciling visual manipulation with the reactive semantics of the language, ensuring that every interaction corresponds to meaningful program behavior while preserving safety and consistency. Another challenge concerns scalability: applications must be represented in a visual form, often as a graph of interconnected elements, but such structures can quickly become complex and difficult to navigate. Maintaining clarity requires mechanisms for visualization, filtering, and reorganization to prevent information overload. In addition, the environment must represent a wide range of system aspects,

including data structures, state changes, logical operations, and presentation views, in a way that remains comprehensible without reducing their expressive power. This prompts the challenge of designing visual abstractions to capture the semantics of a reactive system. A further difficulty arises from the need to support intuitive user interaction, so that applications can be developed incrementally even by users without knowledge of the underlying syntax, requiring interaction techniques that guide the creation of language elements through visual means. Finally, there is an inherent tension between accessibility and expressiveness: while the environment should lower the entry barrier for novice users (users that are not familiar with the underlying programming language), it must also support the flexibility required for advanced development.

1.3 Objective

In our thesis, the primary goal is to design and implement a graphical interface that, integrated with the back-end, creates an environment extending the principles of LP into a visual domain. This environment enables users to construct web applications incrementally through direct manipulation rather than traditional textual programming. Users create and manipulate visual nodes that correspond to language constructs, such as variables, actions, queries, and presentation views, while the system automatically generates and executes the underlying code. In this way, application development becomes a visual process, where assembling and connecting nodes directly reflects the behavior and structure of the program.

While the main focus is on building applications, the environment also emphasizes clarity and structural visibility. Users can freely create, move, edit, and delete nodes or entire structures, making it easy to experiment and iterate on the application design. Visual development can quickly become complex, so the system includes tools for organization, filtering, and search, allowing users to focus on relevant fragments without losing sight of the overall program. By combining flexible manipulation with these navigation aids, the environment ensures that the structure remains understandable and manageable as the application grows.

Another key objective is to preserve the immediacy characteristic of LP. Changes in the visual environment are instantly propagated to the underlying program and its execution, so the effects of every manipulation are visible in real time. This immediacy is reinforced by the dual modes of interaction provided: a graphical workspace for node-based manipulation and a console for direct textual input. Together, these two entry points allow developers to alternate seamlessly between visual construction and code-level interaction, intertwining the accessibility of direct manipulation with the precision of textual commands. The result is an environment where experimentation, iteration, and incremental development are not only possible but continuous.

Ultimately, our environment aims to encourage participation in software creation. By letting users build applications visually and interactively, it provides an intuitive entry

point for those without experience with the underlying language, while still offering enough expressiveness and control for users familiar with programming syntax. In this way, it aims to supports a wide range of developers, from beginners in the language exploring concepts to experienced users refining complex applications.

1.4 Contributions

The main contributions of this work are as follows:

- The design of a GUI that supports multiple types of interactions and a visual language that represents language components as nodes and their dependencies as edges, forming a Directed Acyclic Graph (DAG).
- A visual programming environment connected to a runtime system, enabling users to incrementally develop data-centric applications through the manipulation of visual blocks while ensuring that changes in the code and its visual representation remain fully synchronized.
- Development of simplified menus and interactive techniques to visualize app construction, enabling programmers without syntax knowledge to engage in software development without writing code.
- A comprehensive study of the state of the art in related products and important concepts.

1.5 Document Structure

In this section we present a description of what each chapter in this document discusses. This thesis is organized as follows:

- Chapter 1 introduces the context, motivation, problem, goals and main contributions of this thesis, while also providing an overview of its overall structure to guide the reader.
- Chapter 2 reviews the state of the art in LP, GUIs, User-Centered Design (UCD), direct manipulation, and low-code software, highlighting their strengths, weaknesses, and relevance to this work.
- Chapter 3 presents the Meerkat language and prior prototypes, describing its core semantics and illustrating how it provides the foundation for the proposed visual programming environment.
- Chapter 4 outlines the research methodology, identifies the target users, specifies system requirements, identifies development prototypes, and discusses the technologies selected to meet both usability and technical goals.

- Chapter 5 explains the technical approach in detail, describing the system's architecture, the design of the visual language, and the integration of the GUI with the reactive runtime.
- Chapter 6 demonstrates the system through examples and case studies, showing how applications can be incrementally developed and evaluated within the environment.
- Chapter 7 reports on the user testing phase, covering the evaluation methodology, participant demographics, performed tasks, and results that assess usability and effectiveness.
- Chapter 8 summarizes the findings of the thesis, reflects on its limitations, and outlines promising directions for future research and system improvements.

1.6 AI Tools

Over the development of this document, various Artificial Intelligence tools were used for various purposes, such as:

- **ChatGPT [56]**: This tool was used mainly for organizing original text, due to having some doubts if the structure was intuitive and if we were getting our point across, with an occasional help searching for concepts and examples. It was also used with troubleshooting some code issues, as it was able to provide some insights on how to solve them.
- **Elicit [57]**: Used for finding references for concepts utilized throughout the document and summarize some extensive literature.
- **SciSpace [62]**: Used in the earlier stages prior to developing this document, to help interpret the contents of the documents that we were suggested to read and were essential to understand the context of the work that would be developed in the future.

2

State Of The Art

In this chapter, we introduce some of the work related to context of this thesis. First, we go in-depth on the context of LP, what it is, how it works, what are the main goals and what to have in mind when utilizing it, showing some background work done on the matter. We also take a look at research regarding Reactivity, User Interfaces and User Centered Programming, since they are concepts that are also essential for development. These concepts lay the foundation for the choice on which the solution to our problem was built on. We also analyze work on solutions to analogous problems that can be helpful to achieving our proposed goal, acknowledging their limitations and highlighting their possible insight on the problem.

2.1 Live Programming

In all traditional approaches to programming applications, most of an application's development was done via writing code in a text editor, executing/compiling the code and seeing the result/changes in the interface (code-compile-deploy cycle) [9]. This caused developers to want a more dynamic and responsive way to represent state changes, prompting the development of a new paradigm, LP. LP is an innovative programming approach where a program is executed and updated continuously as it is being written, allowing users to receive real-time instant feedback as they modify code, allowing a dynamic way to develop and debug applications. Research on LP [19] suggests that this innovating way of development is highly beneficial for students and children, that by visualizing the immediate changes made by their own actions, they will be more engaged in their work and aware of their mistakes hence making learning more interactive and enjoyable. However, LP is not exclusively for children and students, also being highly useful for experienced programmers in all contexts of application development.

In terms of application debugging, debugging a program is a difficult task due to the multiple factors involved, such as the complexity of the application state, the interplay between concurrent processes, hidden dependencies between components, and the challenge of reproducing errors consistently across different execution contexts. However,

there are examples of LP that propose an improvement and possible solution to these setbacks, such as UNFOLD [20], allowing developers to see real-time traces of their code execution while they edit it. When interacting with an interface, every state change is documented in a timeline, where each state number is associated with a piece of code, as shown in Figure 2.1. In situations where the GUI representation does not align with the programmer's intentions or expectations, resulting in an undesirable outcome, one can simply observe what piece of code is associated with the undesired state, analyze what exactly is wrong and make corrections. While still a proof of concept, it is a clear demonstration of the usefulness of LP in debugging, improving overall user efficiency.

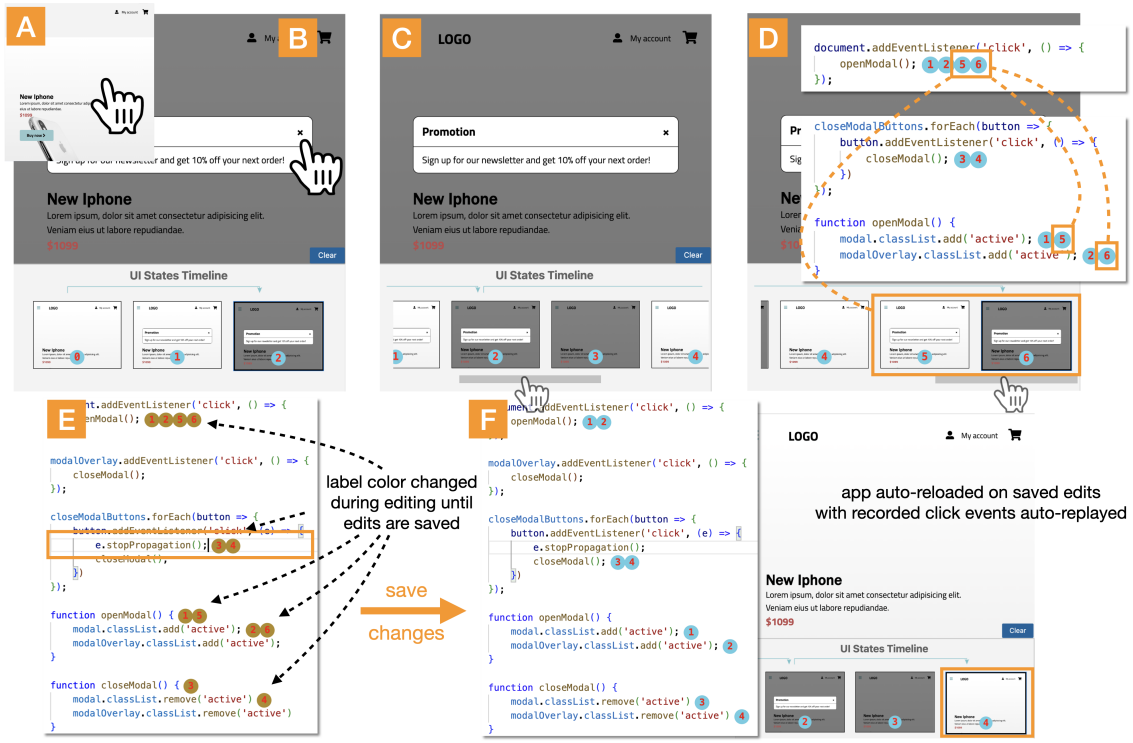


Figure 2.1: Graphical demonstration of UNFOLD, where each page and transition is assigned to a certain piece of code, allowing for a more intuitive debugging experience. [20]

Another example is Light Table [51], an open-source Integrated Development Environment (IDE) that emphasizes real-time feedback and interactivity. It allows developers to see the results of their code changes immediately, without needing to switch contexts or run separate build processes. However, these live previews are context-based, meaning that the type of feedback is different depending on the type of object that is being manipulated. For instance, in Figure 2.2, the editor displays JavaScript code for rendering a cube, while the right-hand side provides a live visualization of the result. Beyond these contextual previews, Light Table also introduces the concept of watches, which serve as a next-generation alternative to traditional print debugging. By attaching a watch to an expression, developers can continuously monitor its value as the program executes, with updates being streamed back into the environment in real time by creating separate

processes that run the interpreters of language being used, and return the results back to the editor. This allows for seamless tracking of important variables without interrupting the workflow. Furthermore, Light Table supports direct manipulation of code during execution: developers can change any parameter that influences the cube's movement or appearance, and the preview immediately reflects these modifications.

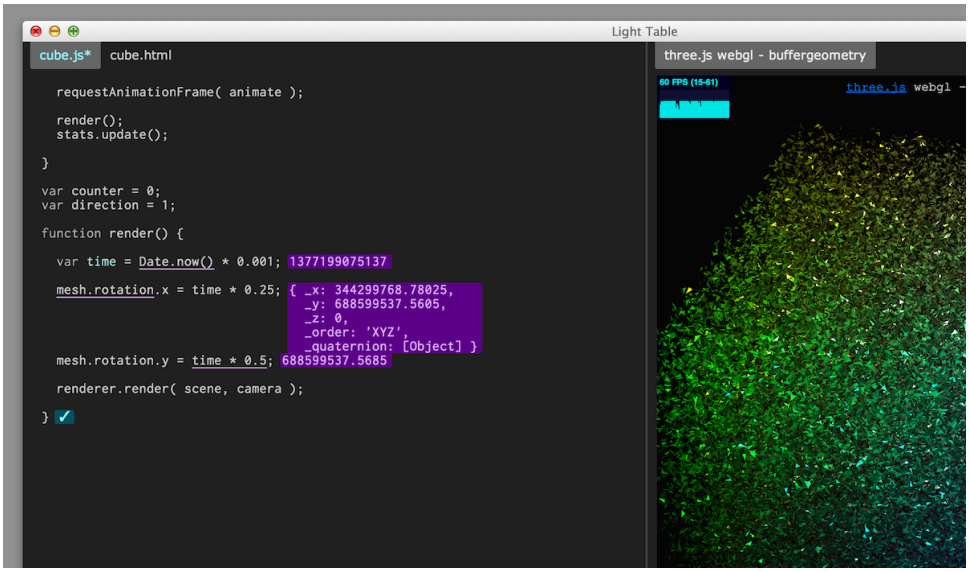


Figure 2.2: Light Table interface showing a JavaScript editor (left) and a live 3D cube preview (right). [51]

However, while allowing for manipulation of code during execution, both of these examples are limited to causing state changes by modifying the code itself, not allowing for direct manipulation of the graphical representation. An approach that aims to connect these alternatives is the programming language Circa [12]. Circa is a LP language and platform designed to integrate the act of coding with the state of the running program. Unlike traditional environments, where runtime and source are loosely coupled, Circa embraces a dataflow-based model of computation, representing programs as directed graphs of terms. Each term specifies a function and its inputs, enabling developers to trace values back to their origins, re-evaluate isolated fragments of code, and visualize execution paths.

A core aspect of Circa is its flow-based introspection. When rendering a scene, the programmer can click directly on a graphical element to inspect the computation that produced it. The runtime re-executes the program in a special pure-only mode, recording all relevant calls (e.g., `draw_sprite()`) and mapping them to screen coordinates. Once the appropriate call is identified, Circa captures the program's intermediate state (the stack), which stores frames with values and references to source code (Figure 2.3). This enables the environment to present a filtered code view containing only the expressions that contributed to the selected element's properties, making it easier to understand and modify complex behavior.

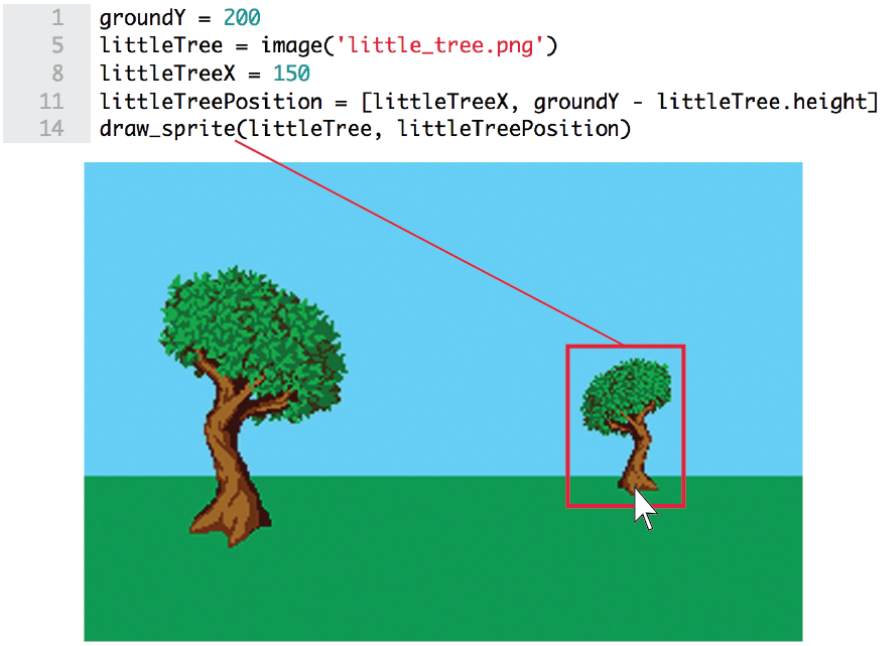


Figure 2.3: Code view is filtered around one call to draw_sprite(). [12]

Beyond introspection, Circa introduces a distinctive feature: feedback-based editing. Rather than limiting changes to textual or graphical code representations, Circa allows developers to express desired outcomes directly through interactions with the program's output. For instance, a user might drag a sprite to a new position on the canvas. The system interprets this manipulation as feedback, propagating the change backwards through the dataflow graph in order to propose or apply the corresponding code modifications, as shown in Figure 2.4. This mechanism allows for a hybrid way of application development depending on the users needs, whether it be visual or textual.

However, some developers are used to their own code editors and development environments, not being open to transition into a LP IDE. Another compelling example of a LP environment that aims to address those issues is Clerk [43], a system built on Clojure [44], a dynamic general-purpose programming language, that brings together the interactivity of REPL development, the structure of literate programming, and the immediacy of computational notebooks. Unlike usual browser-based notebooks, Clerk lets developers bring their own editor, integrating seamlessly with familiar tools like Emacs, VSCode, or any text editor, while still providing live feedback and interactive visualizations. This design allows programmers to maintain their accustomed workflow without being locked into a dedicated IDE.

Clerk transforms ordinary Clojure namespaces or Markdown documents into interactive notebooks, where top-level forms are treated as live code cells and block comments are rendered as rich prose, including support for LaTeX, Markdown, and visualizations. This approach preserves the source code as legal Clojure files, enabling them to live in version control and be used in production contexts, all while providing the benefits of a notebook

```
1   groundY = 200
5   littleTree = image('little_tree.png')
8   littleTreeX = 104.5
11  littleTreePosition = [littleTreeX, groundY - littleTree.height]
14  draw_sprite(littleTree, littleTreePosition)
```

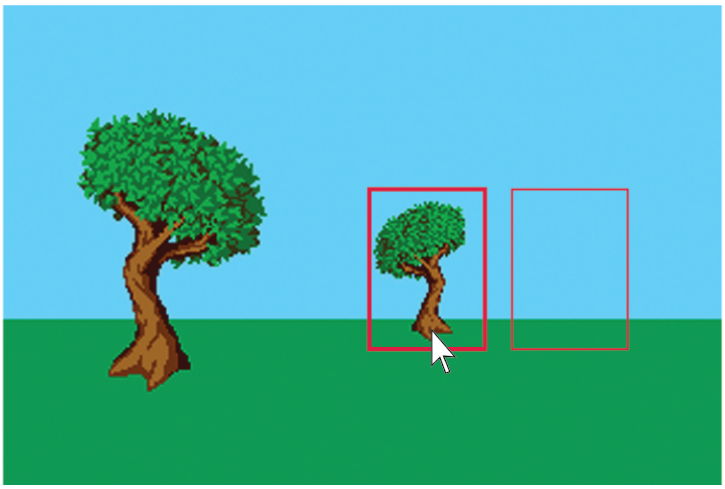


Figure 2.4: Feedback-based editing in Circa. Direct manipulation of the program's output, such as moving a sprite, propagates changes backward through the dataflow graph to suggest or apply corresponding code modifications. [12]

environment. Developers can evaluate individual forms, inspect results, and immediately see the effects of changes, creating a tightly coupled feedback loop (Figure 2.5).

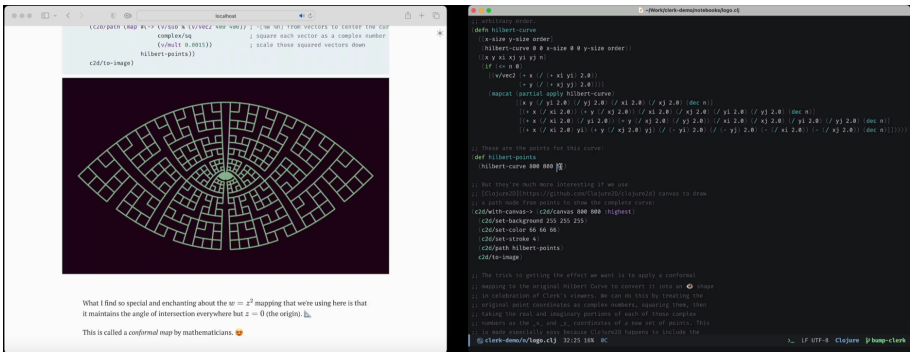


Figure 2.5: Clerk notebook displaying live evaluation of Clojure code alongside rendered output. [43]

A standout feature of Clerk is its incremental computation and caching system. The environment tracks dependencies between forms, so that when a form changes, only the affected computations are re-evaluated. Results are stored in an in-memory cache and optionally an on-disk cache, allowing large or expensive computations to be reused efficiently across sessions. This ensures feedback is practically instantaneous, even for complex notebooks, supporting iterative exploration and rapid refinement of algorithms and visualizations.

Beyond reactive evaluation, Clerk's moldable views allow developers to define how data should be presented. Built-in viewers handle tables, plots, LaTeX, images, and

hierarchical data, while custom viewers can be composed to create interactive, domain-specific representations (Figure 2.6). Viewers can even embed small plots, sparklines, or visual cues directly inside tables, providing rich insight without leaving the notebook. This capability makes Clerk suitable not just for data science, but for documentation, library visualization, exploratory programming, and even small interactive apps.

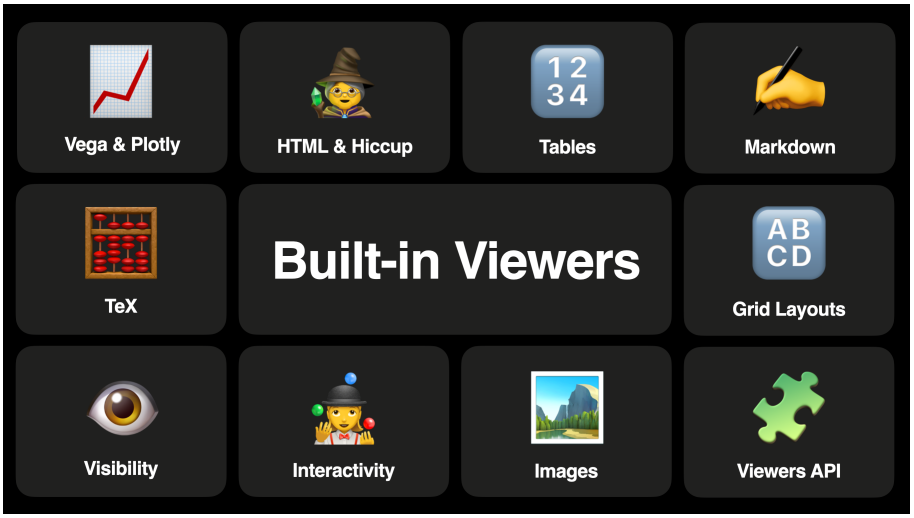


Figure 2.6: Custom viewers in Clerk enable interactive and composable visualizations directly from Clojure data structures. [43]

By preserving the workflow of traditional Clojure development and enhancing it with live interactivity, Clerk demonstrates a modern approach to LP that balances immediacy, flexibility, and integration with existing tools.

2.2 Reactivity

A defining characteristic of intuitive visual environments is their ability to provide *instant feedback*. When users manipulate elements in the interface, changes should be reflected immediately in the running system. This principle of reactivity does not, by itself, constitute live programming, but it is a critical foundation for creating environments that feel direct, responsive, and exploratory. In the context of LP, such responsiveness amplifies the benefits of immediacy, enabling developers not only to see the effects of their changes as they program but also to build a more intuitive understanding of system behavior as it evolves.

Ensuring such reactivity is relatively straightforward in a single-machine setting, but becomes more challenging in *distributed environments*, where updates must propagate consistently across multiple components, devices, or users. In these cases, it is important to understand how reactivity can be preserved without sacrificing responsiveness or correctness. Our initial goal was to implement a prototype connected to a distributed backend. While our work does not address the guarantees and challenges of maintaining reactivity in distributed systems, understanding these mechanisms remains highly

relevant. Research on distributed reactivity provides valuable insights into consistency models, synchronization strategies, and update propagation, all of which are crucial for scaling the principles explored in this thesis beyond a single runtime environment.

One such example is the work of Aceto et al. on *Runtime Instrumentation for Reactive Components (RIARC)* [2]. RIARC introduces a decentralized monitoring algorithm designed to observe reactive systems at runtime while maintaining their responsiveness. Instead of relying on a central monitor, RIARC attaches lightweight *tracers* to system components. These tracers are small observer modules that watch a specific part of the system and collect events as they happen. They are organized in a DAG and forward events to one another using next-hop routing, meaning that each tracer only needs to know which tracer to send an event to next rather than the full path to the final monitor. This approach allows events to reach their destinations efficiently while preserving the correct order (trace soundness). The system is also elastic: new tracers can be created for new components, and unused ones removed dynamically. Although primarily intended for runtime verification, RIARC demonstrates how reactive monitoring can be achieved in a non-intrusive, adaptive, and performance-preserving manner.

Another approach to maintaining reactivity in distributed settings is illustrated by *Historiographer*'s reactive propagation semantics [13]. The key challenge here is that in reactive programs, definitions automatically update in response to changes in their dependencies. For instance, consider the simple program in Listing 2.1:

Listing 2.1: A simple reactive program.

```
1 var x = 1.0, y = 1.5;
2 def avg = (x + y) / 2.0;
3 def diff = x - y;
4 def arith_mean_change = diff / avg;
```

Whenever `x` or `y` changes, the definitions `avg`, `diff`, and `arith_mean_change` must be recomputed. In a distributed setting, however, concurrent updates to `x` and `y` can reach different nodes in different orders. Without care, this can lead to inconsistencies: one node may apply `x`'s update first, another may apply `y`'s first, and the dependent definitions could momentarily diverge.

Earlier systems addressed this with strict coordination, often relying on mutual exclusion to enforce a single global order of updates. While safe, this is costly: parallel transactions must frequently block each other, harming scalability. Other approaches relaxed guarantees, providing only eventual consistency.

Historiographer sidesteps this trade-off. The key idea is that each node maintains a *reactive history*, recording the causal relationships between transactions. Instead of forcing all nodes to observe the exact same order of updates, nodes are free to observe different orders, so long as those orders are consistent with their ancestors. When conflicts arise, updates are merged and applied in *batches*, ensuring that dependent definitions (like `arith_mean_change`) always see a consistent state. This breaks the assumption that every

transaction must be applied one at a time, but allows the system to avoid heavy locking while still ensuring correctness.

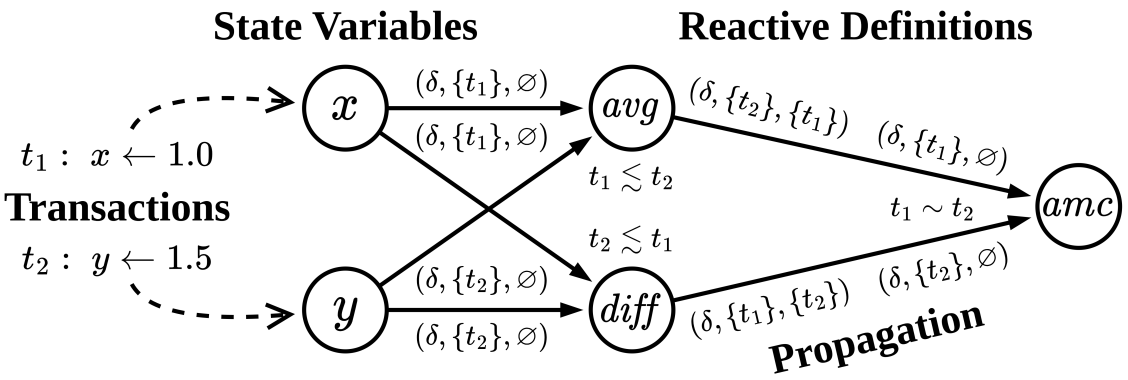


Figure 2.7: Transaction execution and reactive propagation in Historiographer [13]. Each node maintains a reactive history to enforce ordering constraints, applying updates in consistent batches rather than strict one-at-a-time sequencing.

This batching-based propagation guarantees two crucial consistency properties: *glitch freedom* (a definition never observes only part of a transaction's effects) and *history preservation* (updates respect causal and ancestor ordering). In practice, this means that even in distributed, asynchronous settings, reactive programs remain stable and consistent without the performance cost of global locks.

Overall, reactivity in distributed systems ensures that changes propagate quickly and consistently, allowing users or developers to receive immediate, accurate feedback. Achieving this requires mechanisms that maintain responsiveness, preserve the correct order of events, and adapt dynamically to changes in the system. Understanding these principles is essential for designing visual environments and graphical interfaces that remain intuitive, reliable, and responsive even as complexity grows.

2.3 Graphical User Interface

In this section we analyze the concepts behind GUIs, User Experience (UX), and UCD, which were essential for the development of this thesis. These concepts have influenced our design choices to ensure that the system provides an intuitive interaction flow, minimizes cognitive load for users, and aligns with established usability principles. By grounding the development process in these frameworks, we were able to prioritize user needs, improve accessibility, and create an interface that supports both efficiency and satisfaction.

2.3.1 Methodologies

Given that our work mainly revolved around the development of a GUI, we considered essential to find the best methodology of how to design and develop it correctly, the requirements, and the main concepts behind an interface that provides a good UX.

There are various methodologies on how to design a GUI, but we aimed to find the one that is more in line with our proposed goal: an interface that is meant for all types of users, from non-programmers to experts, that allows interactions via direct manipulation of the visual symbols available. To achieve this feat in a successful way, it is imperative that we have a good UX. When considering a good UX design, the essential concept we should take into account is usability [39, 22], which evaluates the effectiveness, efficiency, and satisfaction with which specified users can achieve specified goals in a particular environment. To understand if a design has a high degree of usability, we have to consider various attributes [23, 30]:

- **Memorable**: Is the system easy to remember, or do users have to relearn it each time they use it?
- **Learnable**: Can new users easily figure out how to use it?
- **Efficient**: Once learned, does it have a high level of productivity?
- **Effective**: Does it get the job done?
- **Satisfactory**: Is it pleasant to use?
- **Desirable**: Do people want to use it?
- **Useful**: Does it do something users need?

There are various methodologies that take usability into consideration, like Goal-Oriented Design (GOD). GOD [27] is a methodology based on the assumption that a thorough analysis of user goals and understanding of user intentions make it possible to understand the meaning of user activity and thereby create more suitable and high-quality products. Goals give motivation to users to achieve something. By understanding these goals instead of the tasks or features needed to complete, we can understand what are the expectations and pursuits of the users, which facilitates identifying the types of activity that are truly relevant to the design of the interface.

Another example that is further discussed in section 2.4 is UCD [27], a methodology that focuses on analyzing user needs and capabilities to create products tailored to their requirements. It involves:

- Identifying the target audience at the start of development.
- Understanding user psychology to design from their perspective.
- Researching competitors to identify strengths and weaknesses.
- Iterative design based on user feedback and continuous testing.

Successful applications of UCD requires constant communication with users to answer key questions about their expectations, usage conditions, and unmet needs. Designers create and refine prototypes through repeated testing cycles until user feedback no longer produces significant improvements, ensuring that product development is driven by user needs, making interfaces more intuitive, functional, and user-friendly.

A further example is Emotional design, a methodology that tries to transform human emotional experiences and sensations into specific properties of a product. By doing this, it aims at the design of products that not only allow users to effectively achieve their final goals but also evoke positive emotions when interacting with them. Emotional design recognizes that usability alone is not enough, as users must feel engaged, satisfied, and even delighted when using a product. Therefore, to create a successful computer product like an interface [49], it is necessary to consider three important components:

- **Technical support**: ensuring the product functions efficiently and reliably.
- **Information transmission**: using a visual language embodied in the interface to communicate effectively with users.
- **User perception**: designing interactions that are intuitive, engaging, and emotionally resonant.

Since the developers in this methodology do not communicate directly with users, the interface serves as the mediator between the two, being crucial to designing the behavior of the program in a way that elicits a positive emotional response from users.

2.4 User Centered Programming

Although LP is an innovative concept, it primarily caters to experienced programmers. Many efforts in this field present demonstrations aimed at revolutionizing the programming experience, but these often lack a well-defined roadmap and fail to address critical challenges [10] such as managing side effects, handling stochastic inputs, and visualizing complex data structures, instead, focusing only on simplistic use cases, giving only the illusion of groundbreaking innovation, emphasizing the need for a shift in focus toward the needs of end-users rather than professional programmers, a methodology we sought out to employ in the development of this thesis.

UCD is a methodology that prioritizes the end-users needs, goals, and experiences throughout the design process [24, 1]. It aims to create intuitive, easily understandable systems that require minimal documentation and training, with a high degree of usability [24, 64] . UCD aims to involve users at all stages of development, from requirements gathering to usability testing, with some approaches integrating users as design partners on equal footing (Cooperative design) [1, 7].

To achieve a system that follows the UCD process (usually iterative, repeating phases until the product is perfectly refined), the general phases are [55, 65]:

- **Context of Use Analysis:** Determine the primary user groups, their objectives, specific needs, and the environment in which the product will be utilized.
- **Requirements Specification:** Define comprehensive technical and functional requirements for the product, facilitating the development process and guiding the establishment of clear design objectives.
- **Design and Development:** Develop iterative design solutions informed by the product goals and requirements, incorporating cycles of prototyping, testing, and refinement to enhance usability and functionality.
- **Product Evaluation:** Conduct thorough usability assessments and gather user feedback systematically throughout the design process to ensure alignment with user needs and expectations.

In UCD, requirements specification is a crucial step that ensures the final product aligns with user needs, business goals, and technical constraints by systematically gathering, analyzing, and documenting essential requirements. If the process does not have a well-defined approach to identifying user needs, it risks leading to a product that is difficult to use, fails to meet business objectives, or encounters technical limitations. To ensure this does not occur, it is needed that designers follow some methods and procedures [67]:

- **User and Market Research:** Utilizing surveys, interviews and other user research tools, understanding the who (user personas) and what (competitor analysis, context, goals, and pain points) is crucial.
- **Prototyping the Requirements:** Creating personas and scenarios ensures that requirements reflect real-world usage, utilizing methods such as use cases, task analysis, paper prototype, and usability testing.
- **Defining the Requirements Document:** Balancing functional, business, UX, and technical requirements helps keep the product aligned with business goals while maintaining usability.
- **Writing the Specification:** Structuring all findings into a clear, organized document makes it easy for developers, designers, and various stakeholders to follow.

The outcome is a documented set of user requirements for the future new system or revised system which serves as a guideline for design and development.

To employ a user centered approach process, there are multiple practical models that we can consider, but only some effectively align with the principles of UCD. For instance, the Waterfall model [32] (Figure 2.8), while structured and sequential, lacks the flexibility and iterative feedback loops essential for UCD. In Waterfall, user involvement is typically confined to the initial requirements-gathering stage and the final evaluation phase. This limited interaction prevents the incorporation of iterative user feedback, making it difficult

to adapt the product to evolving user needs or address usability issues during development. As a result, the final product often risks misalignment with user expectations.

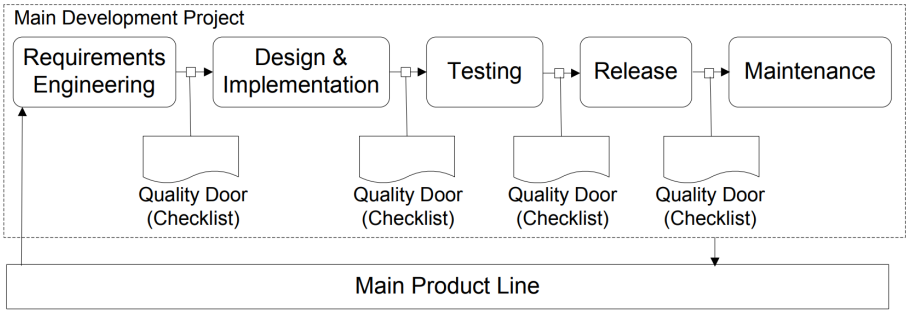


Figure 2.8: Waterfall Software Development Model. [32]

In contrast, iterative models, such as the Spiral model [66] and other Agile methodologies, are inherently more suitable for UCD. As shown in Figure 2.9, the Spiral model emphasizes an iterative cycle of planning, prototyping, testing, and refinement, allowing users to engage at every stage of development. This enables continuous feedback, usability testing, and risk mitigation, ensuring the product evolves to meet user needs effectively. Similarly, Agile prioritizes collaboration, adaptability, and iterative development. Regular user input is gathered through sprints or incremental releases, enabling frequent refinements based on real-world usage and feedback. These iterative approaches foster a dynamic relationship between developers and users, ensuring the design remains user-focused and responsive to changing requirements, which is central to achieving the goals of UCD.

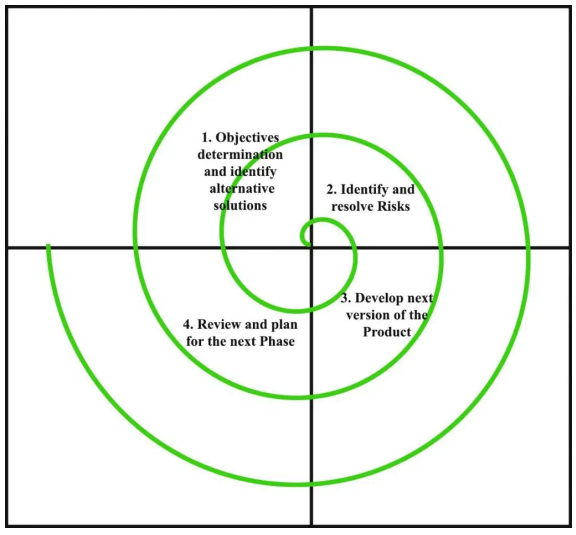


Figure 2.9: Spiral Software Development Model. [66]

This approach aims to create a more comprehensive and user-friendly programming environment, which led to the development of various works such as the Chorus project [10].

The Chorus project aims to create a standalone programming environment that breaks

free from the constraints of traditional programming technologies, fostering experimentation and innovation, particularly for non-programmers seeking to develop applications. By emphasizing the needs of end-users, Chorus strives to deliver a LP experience that is both practical and user-friendly. Central to this approach is the use of a meta-document, which allows programming without a strict division between users and programmers, capturing higher-level intentions and simplifying the programming process to feel less like traditional coding.

Additionally, Chorus introduces a user-friendly method for integrating live database programming, proposing a unified model that simplifies application development while enhancing the overall LP experience.

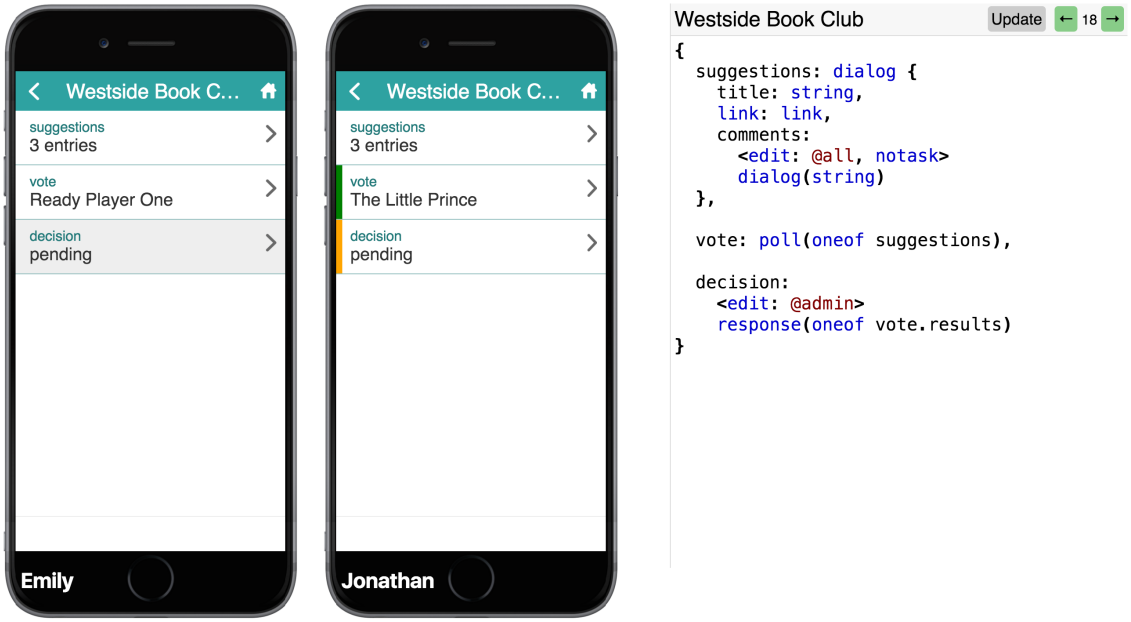


Figure 2.10: Two users are simulated collaborating through the Chorus document defined by the textual syntax on the right. [10]

We continue this analysis on non beginner friendly programming by considering the challenges faced by both beginner and experienced programmers in understanding and manipulating the state of a program.

In the work of Thorgeirsson et al. (2024) [37] there is a interesting metaphor for this issue, where chess grandmasters often play without looking at the board, justified by their strong visual memory (visualizing each game state in their minds), being able to play multiple games at once even while blindfolded. This ability is caused by the fact that no matter the skill level, chess players are accustomed to not only constantly seeing the game state, but also to directly manipulate it when exploring new moves or variations.

This is not the usual case for programmers, who are used to modifying a piece of code while having only a mental model of the program's execution alongside the actual code, not the real time state changes of said program. However, this "lack of knowledge" can be beneficial to experienced programmers, given that visual scaffolding (preview/view

changes as they are made) can distract developers, increasing cognitive load, and might be better suited for beginners.

To exemplify, in Algot [38], which employs programming by demonstration, users are allowed to specify inputs and perform actions in the intended execution order without writing traditional code, aiming to bridge the gap between user intentions and system feedback. Algot represents program state as a directed graph, which enhances the visibility of operations and simplifies some programming concepts for learners.

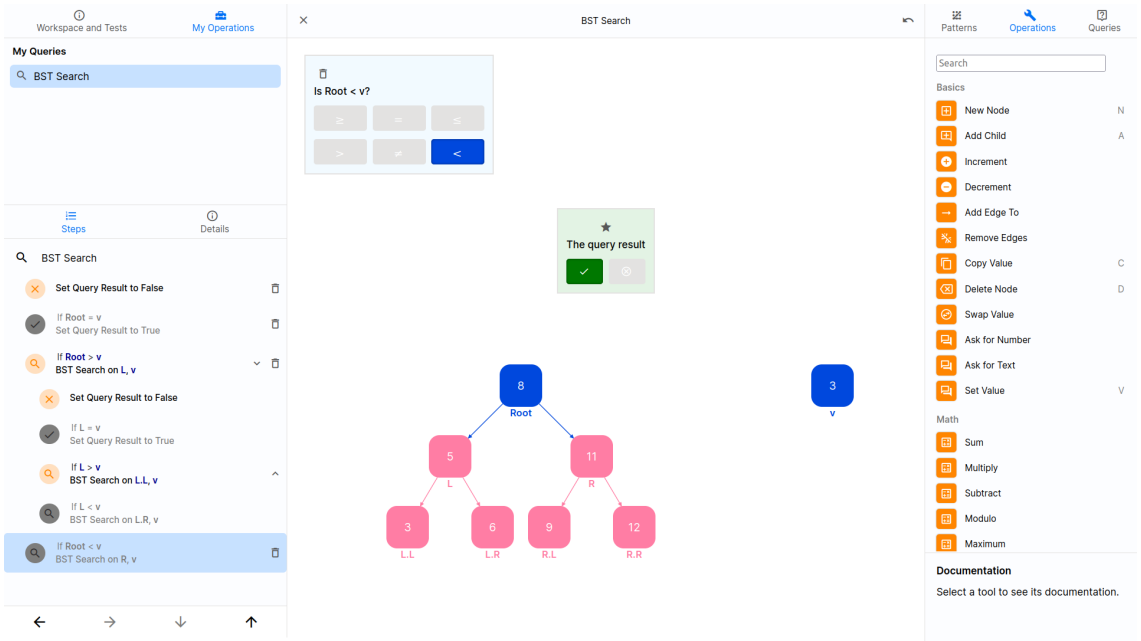
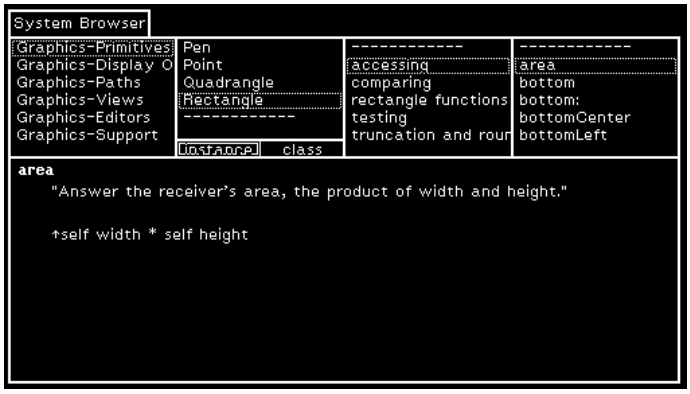


Figure 2.11: A screenshot of an Algot operation that checks whether the input value v exists in the binary search tree with root Root. The system includes support for example-based programming (all nodes have concrete values), an interactive semantic representation of the program (the sidebar on the bottom left), and a system for users to define their own queries. [38]

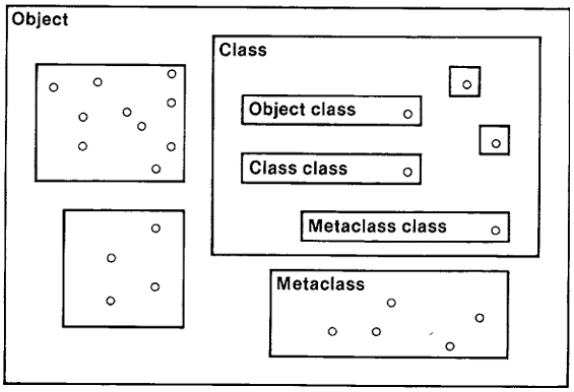
This emphasizes that LP, while a highly helpful model for application development and debugging, should also try to accommodate the needs of beginners and non-programmers.

2.5 Direct Manipulation

In this section we go in depth at the concept and some examples of direct manipulation of visual elements for software development. Direct manipulation is especially relevant in the context of this work because it provides a natural way for users to interact with programming environments through visual elements rather than abstract code alone. By enabling intuitive interaction, and a closer connection between actions and their results, direct manipulation principles serve as a foundation for the design of visual and live programming systems such as the one explored in this thesis.



(a) System Browser. [34]



(b) Organization of classes and instances.

Figure 2.12: Smalltalk80 system [16]

When talking about the direct manipulation of visual elements and its impact in software development, it is imperative to discuss what is considered by many to be their foundation, Smalltalk. [15] Smalltalk is a object-oriented programming language developed in 1970s that introduced a new way of thinking when developing software, emphasizing the use of objects and messages for programming. Its first stable release, Smalltalk-80 [16], emerged in 1980 and represented a significant advancement in the field of object oriented languages and GUIs by providing a fully integrated environment for interactive software development. Smalltalk-80 features a visual interface that allowed users to interact with the system through visual elements rather than traditional text commands. This system included a variety of tools such as a system browser for class management (Figure 2.12a), a debugger for error handling, a change manager for version control, and the concept of hot swapping, the ability to alter the running code of a program without needing to interrupt its execution that would later be the concept on which most LP systems would be built on, all of which contributed to a more intuitive, engaging and interactive user experience.

In later years, with the goal of building a educational software that could be used and programmed by non programmers while perfecting and taking advantage of the dynamic and incremental programming of its ancestor came Squeak [21], an open source

implementation of the Smalltalk. Squeak [5] uses as its interface Morphic, a UI construction environment that employs graphical objects, referred to as "Morphs," to facilitate GUI development. Over that GUI model lies another UI framework, Etoys, a child-friendly constructivist learning environment that employs visual programming, allowing users to create programs by manipulating program elements graphically rather than in a conventional text editor. In Etoys users can use an array painting tools in order to design

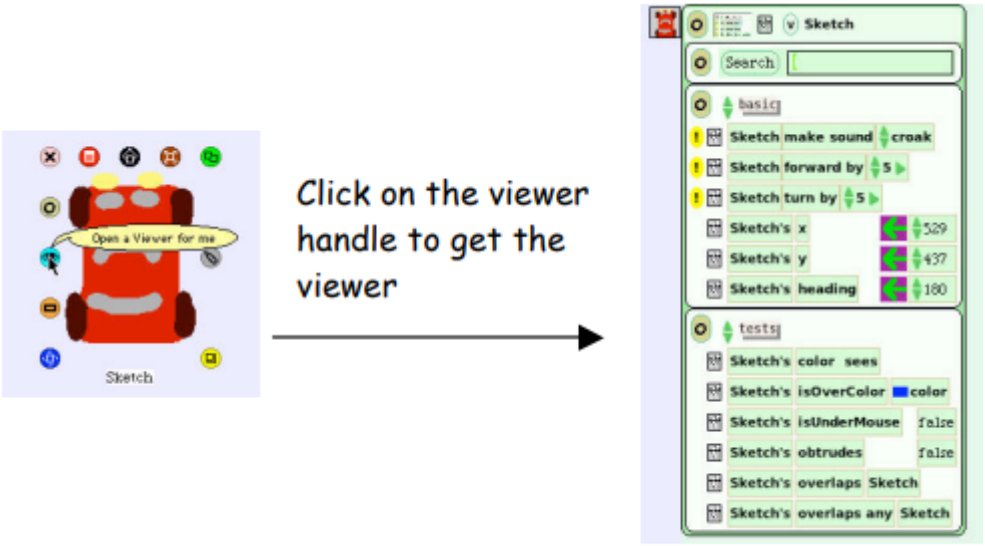


Figure 2.13: Etoys environment [50]

an object to their liking, which can be freely manipulated around the interface. Every object in Etoys contains their own properties and behavior, and we can manipulate those as well by using the various oval shapes surrounding said object called Handles [50]. In figure 2.13 we can also view the Viewer tab associated with the current object being manipulated. This tab provides a comprehensive interface for examining the object's properties, characteristics, and capabilities. It enables users to modify attributes such as position, rotation, and direction, while also offering the functionality to utilize pre-existing scripts or develop new ones. This feature supports experimentation and facilitates the creation of animations in an intuitive and user-friendly manner.

Another example of direct manipulation is Scratch [28] [60] , a visual programming language inspired by Squeak designed to make coding accessible and intuitive for users of all ages. Built upon the principles of Squeak, Scratch also allows creation of programs while introducing several innovations, such as a built-in block-based programming interface that eliminates syntax errors, and features specifically tailored to support creative learning and storytelling, not needing a separate learning environment. This visual approach makes programming more accessible to young learners, encouraging users to experiment with different programming elements and concepts through hands-on experimentation and peer-to-peer collaboration, enabling them to share ideas and learn from one another. While demonstrating strides in the direct manipulation of graphical elements and having the type

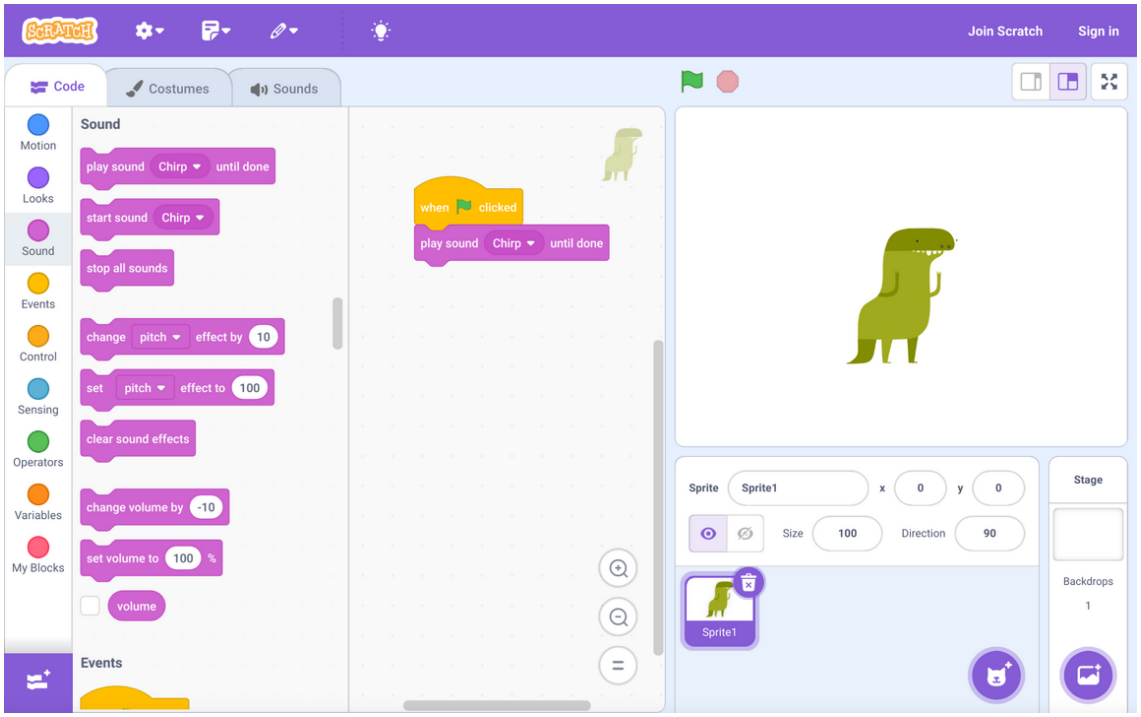


Figure 2.14: Simple Scratch Project

of interaction that we could take inspiration from, Squeak and Scratch are implementations tailored to children, more as a introduction to programming as a concept rather than incremental application development, lacking in almost all factors needed to developing what is proposed for the thesis.

Nonetheless, they gave inspiration to many, driving significant advancements in the realm of direct manipulation interfaces. One notable example is App Lab from Code.org [41], an interactive platform designed for both beginners and more advanced users to create web applications through a combination of visual design tools and text-based JavaScript coding. App Lab facilitates a smooth transition from block-based programming to real code, making it an ideal educational tool for understanding core programming concepts while still engaging with direct manipulation techniques.

App Lab allows users to design user interfaces using a drag-and-drop visual editor where elements like buttons, text inputs, and images can be placed and customized interactively, while also being able to interchange the block-based code with traditional JavaScript code with a switch of a button for more experienced users. As users modify properties of these elements, such as changing text labels, adjusting colors, or repositioning components, App Lab reflects these changes immediately in the code, fostering a bidirectional connection between the design view and the source code.

Another example is the work of Schuster et al. (2016) [35], a implementation of LP by example, which enables updates to the code of running programs through direct manipulation of the program's user interface, rather than editing the source code.

The prototype illustrated in figure 2.16 [59] uses a dynamic string origin analysis

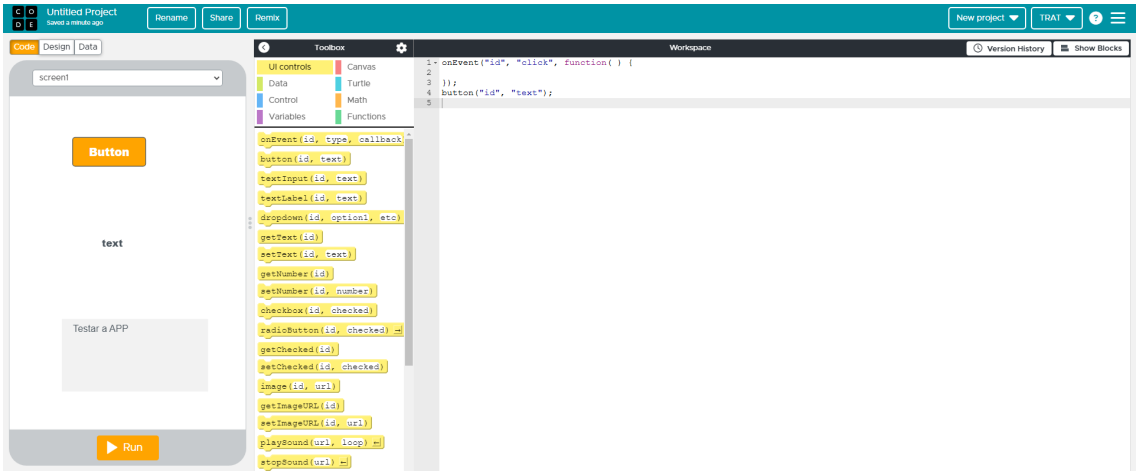


Figure 2.15: Code.org App Lab interface showing the visual UI builder alongside the JavaScript code editor, with tabs to design elements and add data structures [41]

that enables the system to track the relationship between the HTML output displayed to the user and the corresponding string literals in the underlying JavaScript source code. When a programmer interacts with and modifies the HTML output, the system uses this analysis to identify which string literals in the source code need to be updated to reflect these changes. While mainly focused on the modification of string literals, this type of flexibility in terms of interacting with HTML to change source code and vice-versa serves as a more intuitive and direct way of modifying running programs, providing a proof of concept that direct manipulation of user interfaces can be a helpful way to construct web applications.

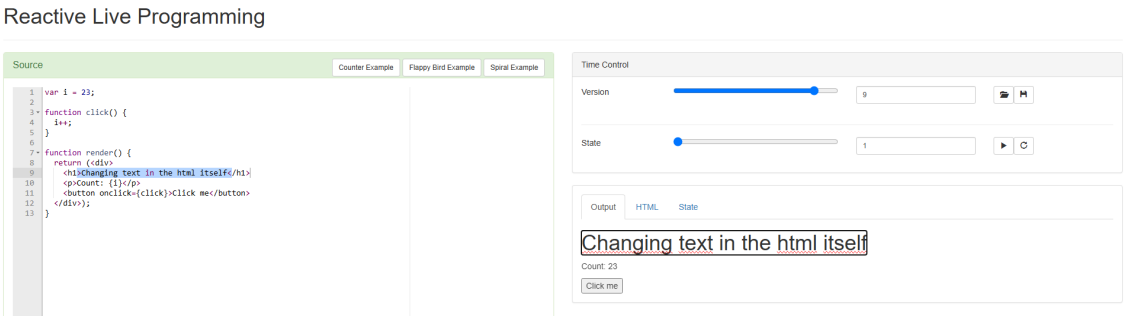


Figure 2.16: Prototype that allows direct changes to the html page. [59]

2.6 Low-Code Software

In our proposed thesis, we aimed to develop an application that moves beyond traditional programming toward a more interactive approach, allowing direct manipulation of graphical interface elements. For this reason, we considered the low-code paradigm worth exploring, since it shares certain interactive features aligned with our goals. Low-code [58]

facilitates application creation with minimal manual coding, typically providing a user-friendly drag-and-drop interface, visual tools for defining data structures and application logic, and prebuilt components to streamline development. While our work is not a low-code system, it aligns with some of these principles, particularly visually simplified development.

A related concept is no-code, which eliminates the need for programming skills entirely. No-code development can be highly accessible and enable rapid application creation, but it has notable limitations, such as restricted customization, difficulty handling complex use cases, and potential challenges with scalability and performance. These constraints highlight why low-code platforms, which balance visual development with extensibility, are more relevant for studying interactive development approaches.

Several prominent low-code platforms illustrate these principles:

OutSystems [17] is a cloud-based platform for building and deploying web and mobile applications within a single environment. It emphasizes fully interactive visual components, prebuilt or user-defined, representing data models such as database tables and variables, and allows developers to define interactions between them, reducing reliance on traditional coding and accelerating development.

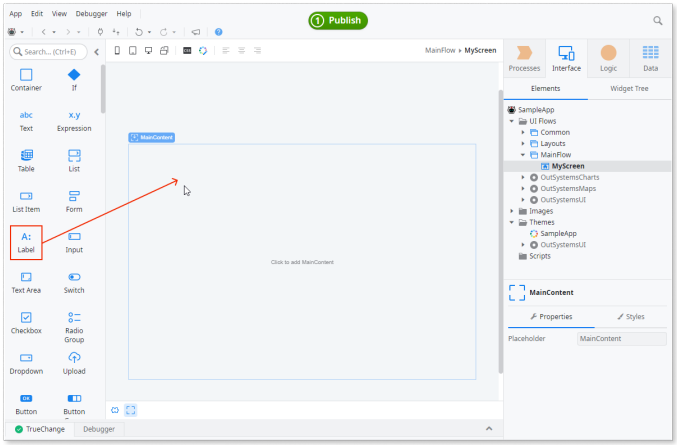


Figure 2.17: OutSystems User Interface [17]

Mendix [53, 18] combines visual modeling with extensibility for professional developers. Its web-based Mendix Studio allows business users to build applications via a WYSIWYG (what you see is what you get) drag-and-drop interface, while the desktop-based Mendix Studio Pro lets experienced developers extend functionality with Java and JavaScript. Mendix also supports seamless cloud deployment with multi-cloud compatibility, promoting collaboration between technical and non-technical stakeholders.

Microsoft PowerApps [61, 47] provides a rapid development environment for custom business applications. Canvas apps allow freeform drag-and-drop creation, while model-driven apps use prebuilt components for structured, component-based development. PowerApps integrates with cloud services via Dataverse and supports responsive deployment across devices, combining flexibility with structured design for diverse business

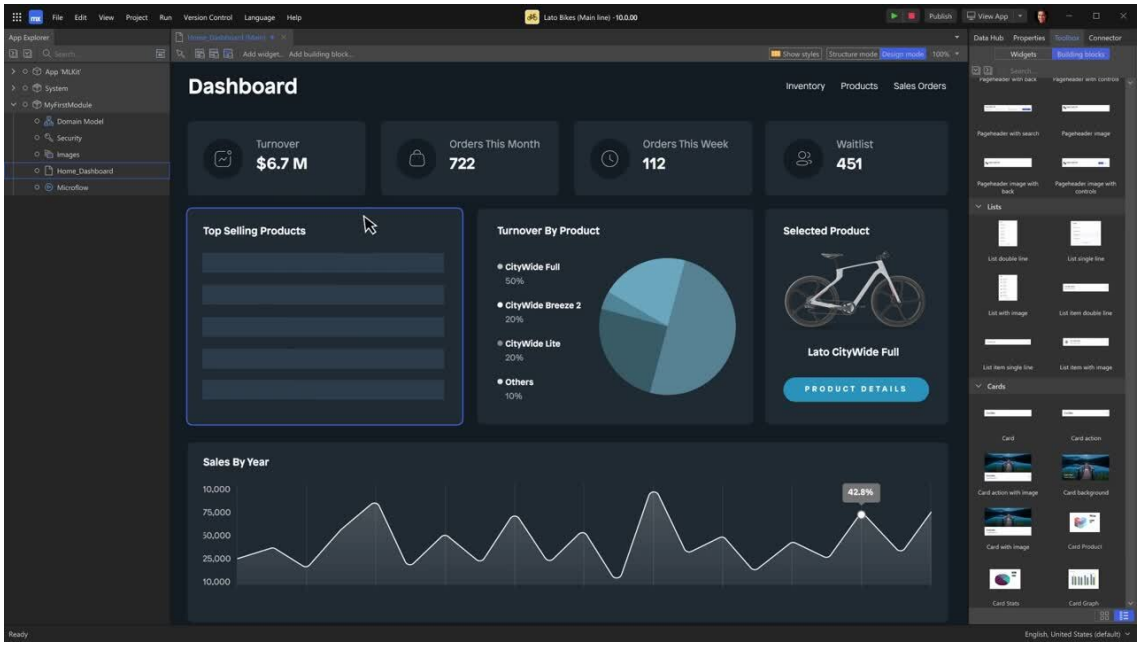


Figure 2.18: Mendix Studio Interface [53]

needs.

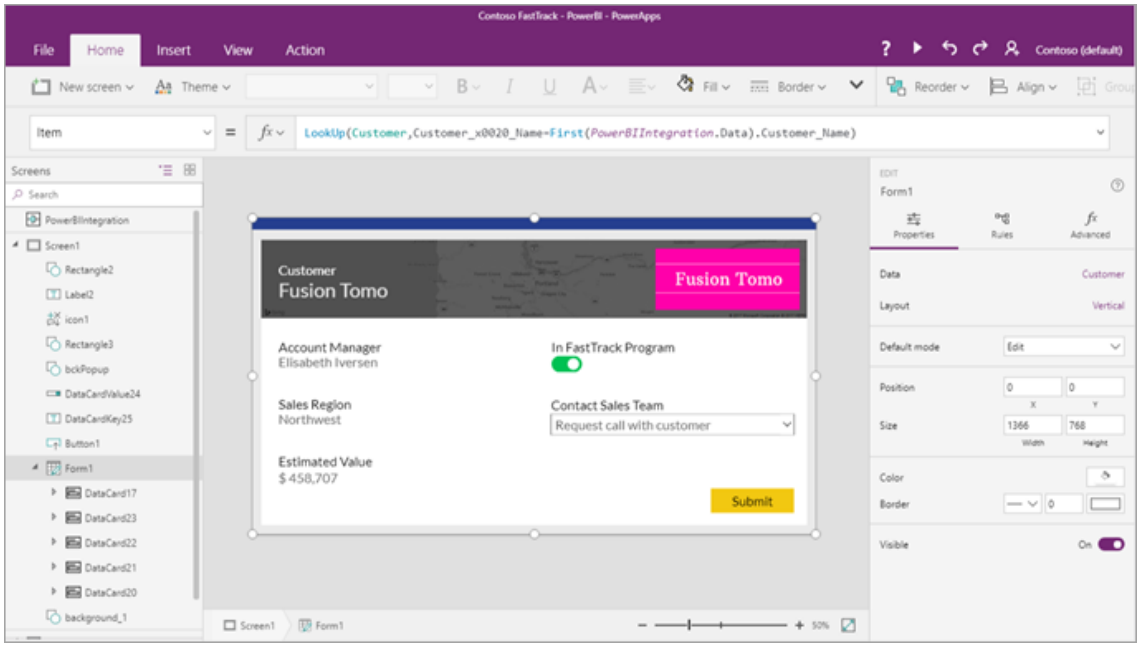


Figure 2.19: PowerApps Interface [47]

Toddle [63] is an open-source visual development platform aimed at professional teams. As a low-code tool, it allows application creation without traditional programming, while leveraging concepts from major frameworks like Angular, Vue, and React. Users build applications using a visual, component-based interface, with the platform handling underlying code generation, providing both speed and performance comparable to conventional frameworks.

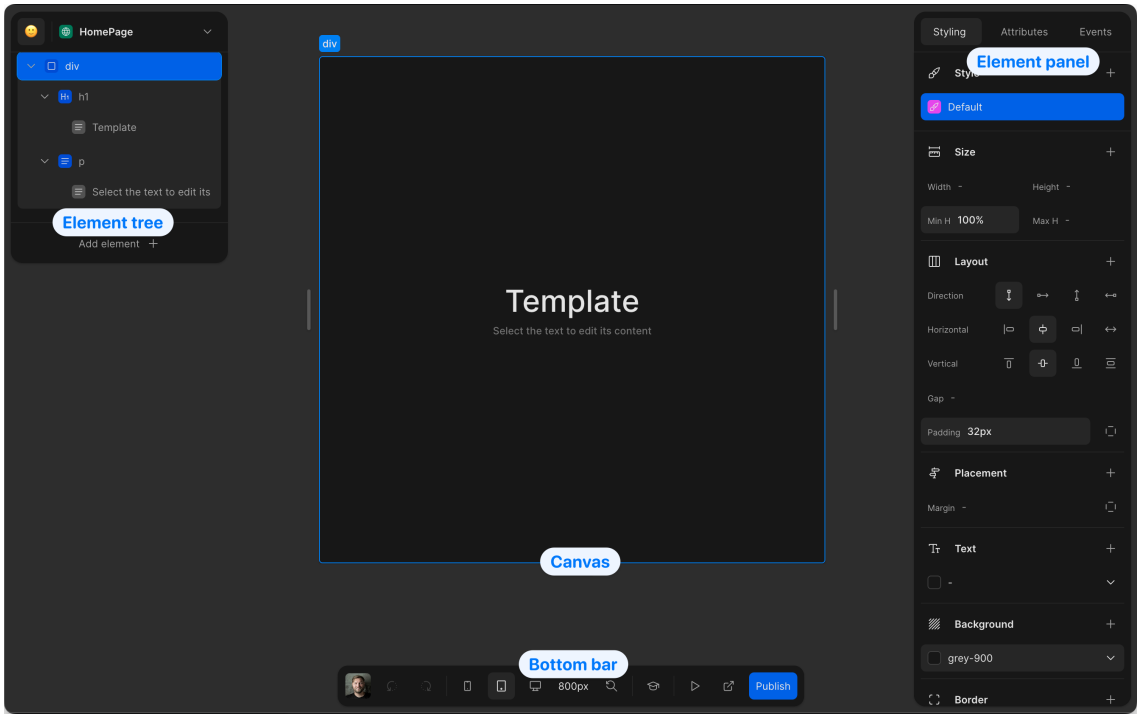


Figure 2.20: Toddle Editor [63]

In conclusion, low-code software is immensely helpful as it bridges the gap between technical and non-technical users, enabling faster and more accessible application development. By utilizing visual tools, drag-and-drop interfaces, and pre-built components, it reduces the need for extensive coding knowledge while maintaining the flexibility to address complex use cases.

2.7 Previous Work

Our work builds upon previous efforts to develop a visual programming environment for the Meerkat language. In this section, we analyze a prototype developed by Alves [3], which, despite its limitations, provided a foundational reference for understanding how visual programming can be applied to web development and guided the design of a more complete environment.

The prototype consists of a graphical interface in which developers place symbols representing variables, actions, and data transformations onto a canvas (Figure 2.21), enabling application construction in a manner similar to our approach.

Applications are constructed by dragging visual symbols into a canvas and connecting them to define dependencies, such as linking a variable to an action or a data transformation (Figure 2.22). This allows users to build programs by directly manipulating visual elements rather than writing code textually.

While it also employed a DAG-based representation, its visual language focused on generic data types displayed in the sidebar, such as numbers, text, database tables,

expressions, and HTML views. Nodes displayed limited information. For certain non-data-structure nodes, some details were hidden by default. This limited not only the user's ability to quickly grasp the structure and state of the program and slowed interactive exploration, but also minimized the learning of the underlying language, as code statements were not visible as applications were developed. Dependencies between symbols are encoded via arcs, forming a directed graph that models the program's logical structure.

In contrast, our system's visual language was designed to closely mirror the constructs of the Meerkat programming language. Each node represents a specific language feature, including variables, definitions, actions, database tables, HTML pages, or modules, while edges similarly represent dependencies between them. Nodes display relevant information directly, including names, values, and data contents, providing a clearer view of the program's structure and state.

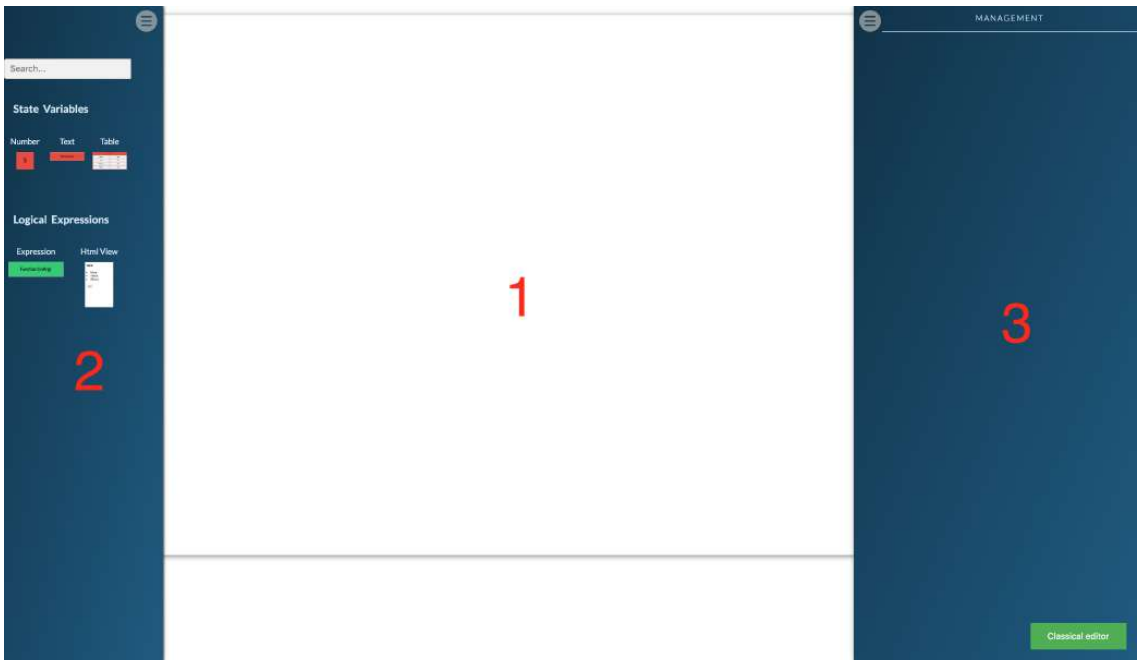


Figure 2.21: Prototype graphical interface structure. [3]

In this system, as users manipulate symbols, the system generates corresponding LP code and evaluates it incrementally on a server. Updated values are sent back to the client via WebSockets, enabling real-time observation of changes without requiring manual code entry.

Despite its conceptual contributions, the prototype exhibits several limitations when compared to our system. Unlike our environment, it allows connections that do not conform to the underlying language semantics. It lacks support for grouping symbols into modules, which is an important language feature, preventing further scalability. Users cannot filter elements by type, and the system does not provide automatic layout for complex applications, making navigation cumbersome. Additionally, there is no way to alternate between textual and visual input, limiting flexibility for users who prefer

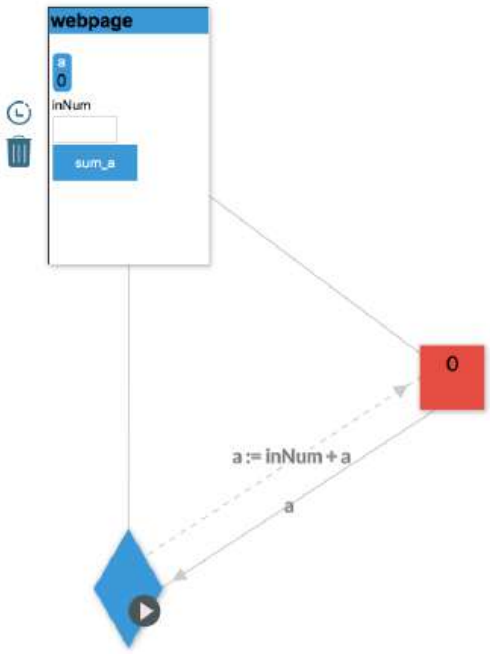


Figure 2.22: Example of a program managing an integer with an HTML view. [3]

a hybrid workflow. The prototype is also not designed for mobile devices, restricting access on tablets or smartphones. In contrast, our system enforces valid dependencies, supports modular components, offers filtering and layout features, integrates textual and visual programming, and is compatible with mobile devices, further enhancing usability, scalability, and flexibility.

By overcoming these limitations, our system delivers a more comprehensive and practical environment for visual programming in Meerkat, facilitating clearer program representation, easier navigation, and smoother integration between visual and textual development workflows.

3

Meerkat

In this chapter we present the Meerkat language [9, 36], which forms the foundation of the visual programming environment developed in this work. We introduce the essential features of Meerkat and highlight the constructs most relevant for its integration into a visual programming environment. We begin with the top-level operations that form the backbone of the language and then present the broader expression language, which supports functional and reactive composition. We also illustrate these concepts with a simple running example, showing how variables, definitions, actions, and HTML can be seamlessly combined to build interactive applications.

Alongside the language, we describe `Live Programming`, the original web-based prototype, which offered a live coding experience and served as a precursor to the visual environment presented in this work.

3.1 Language

The underlying base language used for application development in the visual interface in this work is the Meerkat language [9, 36]. It is a live, tierless programming language designed to support incremental, reactive web development, unifying application logic, data, and User Interface (UI) description in a single runtime environment. Meerkat allows developers to evolve running systems by updating definitions, inserting new components, and executing changes without restarting or reloading the application. It emphasizes type safety and dependency tracking to ensure that updates do not break the live execution state.

The language introduces three fundamental top-level operations, all parametric over an expression language (**e**) that supports reactive and functional-style constructs [11]:

- **var x = e** – Defines a state variable `x` with an initial value given by the expression `e`. This variable represents persistent application state and is modified only by explicit assignment operations (called actions). Additional persistent structures, such as database tables, follow similar semantics.

- **def x = e** – Defines a pure, reactive computation `x`, whose value is derived from expression `e` and automatically updated whenever its dependencies change. These are used to define views, expressions, and derived computations.
- **do e** – Executes an action value denoted by `e`. Actions cause side effects, such as modifying state variables or table entries, and must be explicitly triggered (e.g., by user interaction). They are not reactively re-executed.

The full syntax includes a broader set of constructs:

Listing 3.1: Top-level operations and declarations

```
o ::= r
    | do e -- Interaction Operation

r ::= var a = e -- State Variable
    | def a = e -- Reactive Definition
    | atomic {r*} -- Composition of Operations
    | module a<param*> -- Module Definition
    | when(e)
    | with a(e*) {i* r*} -- Activate Module
    | @a {i* r*} -- Redefine Module
    | table a {(x: t)+} -- Table Declaration
    | delete a -- Remove Name
    | deleteall -- Remove All Names

i ::= import a as a
    | from a in a where e
    | import first? e as a (default e)?

p ::= usid | t*? a -- Module Parameters
t ::= number | string | boolean -- Base Types
```

The expression language (`e`) supports standard programming constructs including arithmetic and logical operators, `if-then-else`, `match`, `let`-bindings, and higher-order functions such as `map`, `foreach`, and `iter`. A special class of expressions produce `action` values, which encapsulate side-effecting changes such as assignments (`action {x := e}`).

Beyond these core constructs, Meerkat provides higher-level abstractions that integrate naturally with its reactive model. A key construct is the **table** declaration, which introduces a persistent data structure with a declarative schema. Tables behave similarly to relational database tables and support actions like `insert`, `update`, and `delete`. Table queries and dependent computations are reactive by default: changes to table contents automatically trigger reevaluation of affected definitions.

The language also treats HTML elements as first-class values. In the expression language described in [9, 36], Meerkat supports direct embedding of HTML using familiar tag-based syntax inside `def` expressions. This enables interface components to be expressed declaratively and updated reactively in response to data changes, eliminating the need for separate templating or rendering logic.

Under the hood, Meerkat tracks dependencies between all top-level definitions using a DAG. This ensures glitch-free reactive propagation: updates flow through the DAG in

Listing 3.2: Expression language

```
e ::= a | b | x | #a | a@a | usid
    | action {assign*}
    | <tag attr*>e*</tag>
    | e op e | not e
    | let x = e in e
    | e ? e : e | if e then e else e | in a then e else e
    | (x+) => e | e e | [e*]
    | iter e e | foreach(x in a with y = e) e
    | map(x in a) e | get x in a where e
    | match x with x::xs => e | [] => e
    | {(x': e)+} | e.x | linkto a e+ | str e | workspace_path

attr ::= attrName = e

assign ::= a := e
       | insert e into a
       | update x in a with e where e
       | delete x in a where e

op ::= + | - | * | / | %
    | == | != | > | < | >= | <=
    | and | or
    | :: | ++
```

topological order, and the runtime prevents illegal operations such as deleting a node still required by another. All reactive computations are statically checked to ensure termination and correctness of update order.

The language also supports Modules [25], which provide isolated, named environments where data and state can be declared and shared internally, but remain inaccessible from outside. This design enables controlled access and fine-grained sharing among user groups. To ensure global uniqueness, modules internally prefix identifiers with their name. They also support selective access to external data via an import mechanism based on lenses: bidirectional views that allow reactive access and manipulation of external state while preserving encapsulation and security. Meerkat includes support for general, filtered, and conditional imports with fallback defaults, enabling flexible yet safe inter-module interactions.

To illustrate the language in practice, consider the simple example of a counter application, as shown in Listing 3.3. The program begins by declaring a state variable `counter`, initialized to zero. An action named `inc` is defined to increment this variable by one whenever triggered. Two additional reactive definitions, `plus5` and `times2`, demonstrate how values can be derived from the counter, automatically updating whenever the counter changes. The view `counterPage` presents a simple HTML interface that integrates these elements and allows the user to seemingly visualize and interact with the application. It displays the counter's current value and provides a button to invoke the increment action. The interface also includes an input field pre-filled with the current counter value, alongside a button that updates the variable based on the user's input.

In summary, Meerkat is a unified, reactive language tailored to web application development, offering first-class reactivity, live execution semantics, and safe evolution of

Listing 3.3: Implementation of a counter with derived values and update input

```
1
2  var counter = 0
3  def inc = action { counter := counter + 1 }
4  def plus5 = counter + 5
5  def times2 = counter * 2
6
7  def counterPage =
8      <div>
9          <h1>"Counter"</h1>
10
11         <p>"Counter: " counter</p>
12         <button doaction=(inc)>
13             "Increment"
14         </button>
15
16         <br/>
17         <br/>
18
19         "New value: " <input type="number" id="newCounter" value=counter/>
20         <button doaction=(action { counter := #newCounter })>
21             "Set counter"
22         </button>
23     </div>
```

state and structure. Its design makes it especially suitable for integration into visual LP environments like the one described in this work.

3.2 Live Programming

The original development environment for Meerkat is a web-based prototype [25, 9, 29] named *Live Programming* that allows users to write and execute Meerkat code in a browser. This prototype provides a live coding experience, where changes to the code are immediately reflected in the running application, enabling rapid prototyping and iterative development. Applications are constructed incrementally, through modifications to existing features, the addition of new functionality, or updates to the application's data schema. Each change is statically verified by the interpreter to ensure the system's consistency, while the runtime system evaluates statements as they are written and propagates their effects through the application.

Upon entering the environment (Figure 3.1), users are presented with a command prompt for writing Meerkat statements. Commands containing syntax errors are immediately invalidated, with error feedback displayed to guide the user in correcting them. Successfully executed commands are added to a structured list, making all variables, definitions, actions, and HTML components visible and easy to access. Selecting an element in this list reveals additional options in a side panel, such as deleting the element, viewing its definition through a QR code, or executing an action. When actions require parameters, input fields are dynamically displayed to allow the user to supply values before execution. For HTML components, the panel offers direct navigation to the corresponding page.

The side menu also includes utilities for resetting the workspace and accessing a

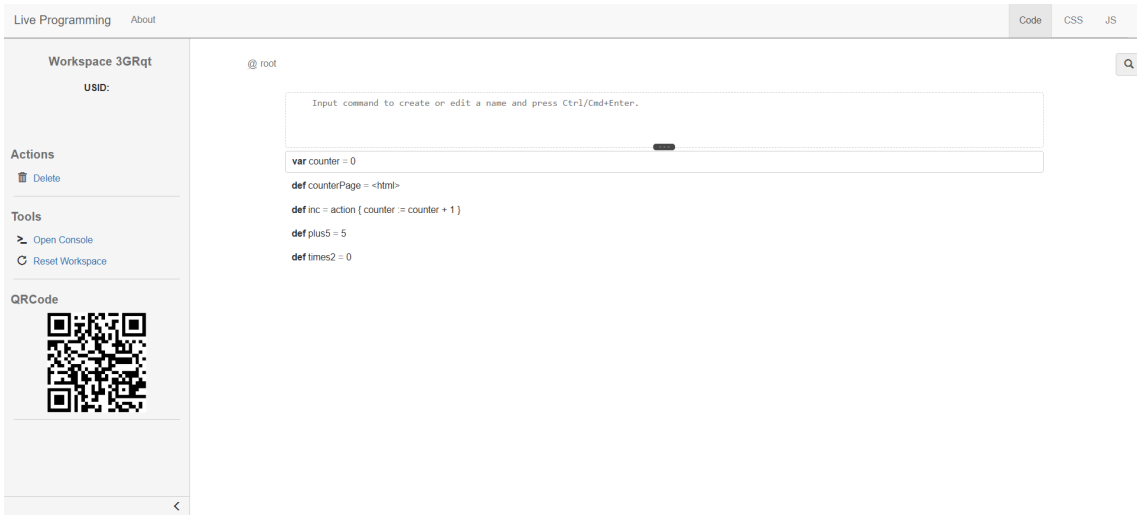


Figure 3.1: The *Live Programming* interface for Meerkat.

console with command history. A search bar enables quick lookup of elements by name, and dedicated tabs are provided for editing Meerkat code alongside CSS and JavaScript, supporting customization of application style and behavior. For users familiar with the language and its syntax, the prototype offers a concise yet powerful interface for developing Meerkat applications.

The architecture combines a runtime system running on a remote server with the browser-based IDE. The IDE enables programming of both the behavior and the user interface of a web application using the Meerkat language, while the runtime maintains application state, manages subscriptions, and delivers updates to connected clients via WebSockets. Through this mechanism, any modification to a subscribed element automatically triggers recomputation and update of dependent components.

To support the interaction between the text editor and the runtime system, the environment uses a REST interface: `GET` requests retrieve the value of an element, such as a number, an action, or an HTML page, while `POST` requests send commands to the runtime. These commands can introduce new variables, definitions, and HTML components, update existing ones, or execute actions that modify persistent state. Behind the scenes, the interpreter enforces type safety and coordinates reactive updates across the DAG of dependencies, while the database component stores both source code and persistent application data. Together, these elements enable a continuous, LP workflow where the system remains available even as applications evolve.

4

Methodology

In this chapter we present the methodology we used to design and implement our visual environment, focusing on user-centered design principles and practices. We discuss the model, target users, requirements, design choices, development phases, and some initial sketches that guided our design process.

4.1 Model

The development of the prototype followed an iterative model, in which the system was gradually refined through cycles of design, implementation, and evaluation. Rather than attempting to define all requirements and features from the start, the process emphasized incremental improvements, where each iteration produced a working version of the interface that could be tested and adjusted.

Throughout this process, we adhered to the principles of UCD. The design decisions were guided by the needs and expectations of the target users, ensuring that usability was prioritized alongside technical functionality. Early sketches and mockups were employed to explore possible interface layouts, followed by the implementation of partial prototypes that allowed us to validate interaction mechanisms such as drag-and-drop composition, direct manipulation of nodes, and the visualization of dependencies. Feedback gathered from these iterations informed subsequent refinements, reducing complexity in critical areas and ensuring that the final design remained both accessible to novice users and expressive enough for more advanced use cases.

By connecting an iterative model with UCD principles, we aimed for the system to evolve in a way that balanced technical feasibility with usability, allowing us to converge toward an interface that aligned with the overall goals of the project while remaining grounded in the practical needs of its intended users.

4.2 Target Users

When considering the design and development process, the first key aspect discussed in section 2.4 is identifying the target users of our future system. We considered 3 main types:

- **Beginner Developers**: Users with little programming experience who rely on the GUI to create and test simple programs while gradually learning coding concepts.
- **Experienced Developers**: Users familiar with programming who leverage the system's direct manipulation and live feedback features to streamline development and debugging.
- **Non Programmers**: Citizen developers who will not be writing any code and are primarily looking to develop applications per their needs, not worried about any construction details. They usually have experience with software like Microsoft Excel or Google Sheets.

The following type of users were initially identified given that our approach intended to create an environment that requires little to no programming knowledge. Throughout the design process, we tried considering the needs and expectations of each of these users, but as we progressed, we realized Non Programmers would not be able to use the system as it is due to the nature of the environment and prior knowledge needed, so we focused on Beginner and Experienced Developers, as they are the most likely to benefit from the system. Testing was conducted only with Experienced Developers, so we can not guarantee that the system is usable by Beginner Developers, but we are hopeful that the design is intuitive enough to allow them to use it with minimal guidance, since none of the users that conducted the testing had any prior contact with the language and the interface, as is discussed further in chapter 7.

4.3 Requirements

In order to ensure that the design and development of the user interface meets the needs of the target users, the second key aspect in user-centered design was to establish a set of requirements. To begin, we first need to idealize how the environment will visualize our construction model. Since the Meerkat language follows a dataflow programming model, described in chapter 3, we wanted to visually represent the underlying DAG, where nodes would map to programming elements and the edges would represent dependencies between each of the components, and as the graph grew in size and complexity, a visual layout would automatically reorganize nodes to ensure clear visibility.

We first decided on the interface layout, which would consist of a central area where the user can interact with the system and visualize the graph, a toolbox with various components that can be dragged and dropped into the central area, a filter menu that

allows users to change the abstraction levels of the interface components and a text console so users could visualize language syntax and write code if they desired. The toolbox would contain various visual components representing various language components. The filter menu would allow users to toggle between which type of element was to be highlighted. A search bar would also be included to allow users to search for components.

We then aimed to identify the main components that would be represented in the interface and in the toolbox, each corresponding to a distinct symbol. We considered the most relevant operations of the language, as described in Section 3.1:

- **Variable nodes**: store values that can be used or altered by other components.
- **Definition nodes**: derive new values from existing components. These include operations such as computing the size of a database, deriving values from a variable, or creating database views to focus on specific elements.
- **Database table nodes**: represent structured data collections essential for managing information in web applications.
- **HTML page nodes**: represent web pages that can be displayed to the user. They are crucial for visualizing data and providing a user interface for the application. These nodes can connect to variables or database tables to display their content.
- **Action nodes**: represent operations that modify data, such as inserting, updating, or deleting rows in a database table. They can be connected to other nodes to trigger updates or changes in the system.
- **Module nodes**: allow grouping of related components into an encapsulated unit. Modules can contain all other nodes, supporting modular application design that simplifies service management and extension.

Once all language components were defined, we focused on another important feature that was originally proposed. We wanted the system to be compatible with touchscreens and Virtual Reality (VR) devices, so we wanted to ensure that the interface was designed with this in mind, allowing users to interact with the system in a more intuitive way. We also wanted to ensure that the interface was responsive and adaptable to different screen sizes, so that it could be used on a variety of devices, so the original sketching of the interface was done with this in mind. We experimented with some sketches that focused on an approach similar to previous work [3] but with some original features, as shown in sketches 4.1 and 4.2, but we concluded that this approach was not ideal, as the menus would take up much of the screen space and would be unfit for non-desktop devices, so we decided to focus on a design with floating menus, that could be collapsed or expanded depending on the user's needs, as shown in sketch 4.3. This sketch would ultimately be the basis for the design of the interface, despite occasional adjustments made during the development process. In line with the visual clarity goals from previous work [3], we also

adopted the idea that each node should have a distinct shape and color depending on its nature, allowing users to quickly identify a component's type at a glance, as depicted in the sketches.

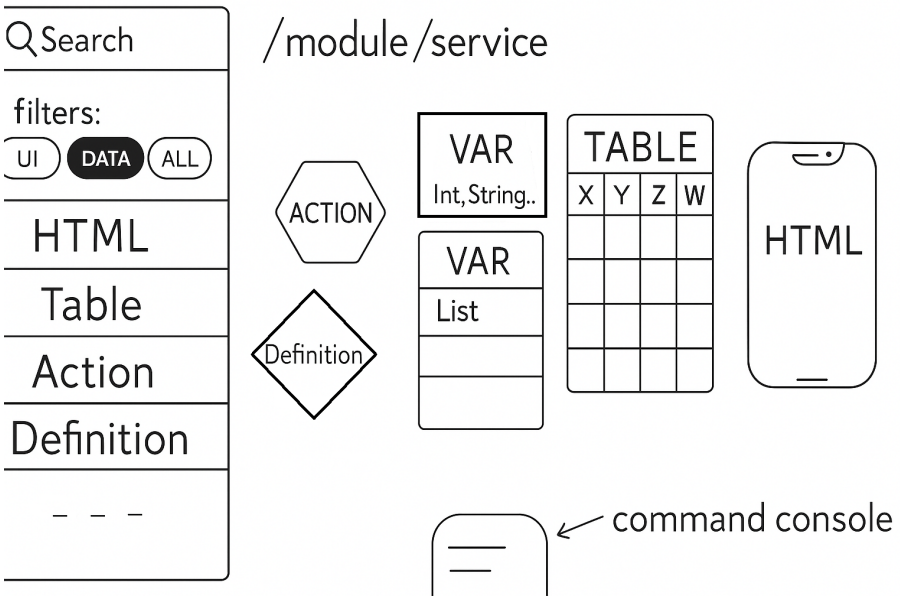


Figure 4.1: Graphical Interface sketch, with collapsible non floating menus, all elements and their designs.

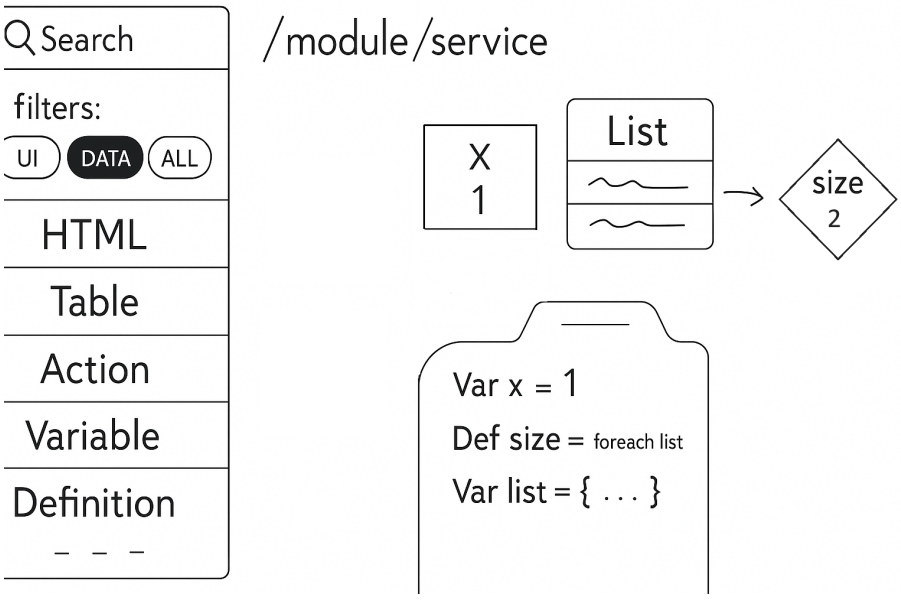


Figure 4.2: Sketch of utilizing the command console in the environment.

Concluding that a basis for our design was set, we proceeded to identify which of the components would cause the most confusion for novice users, so we could focus on simplifying the process of creating them. We identified that actions and tables would be the most complex components to create, as they require a deeper understanding of the

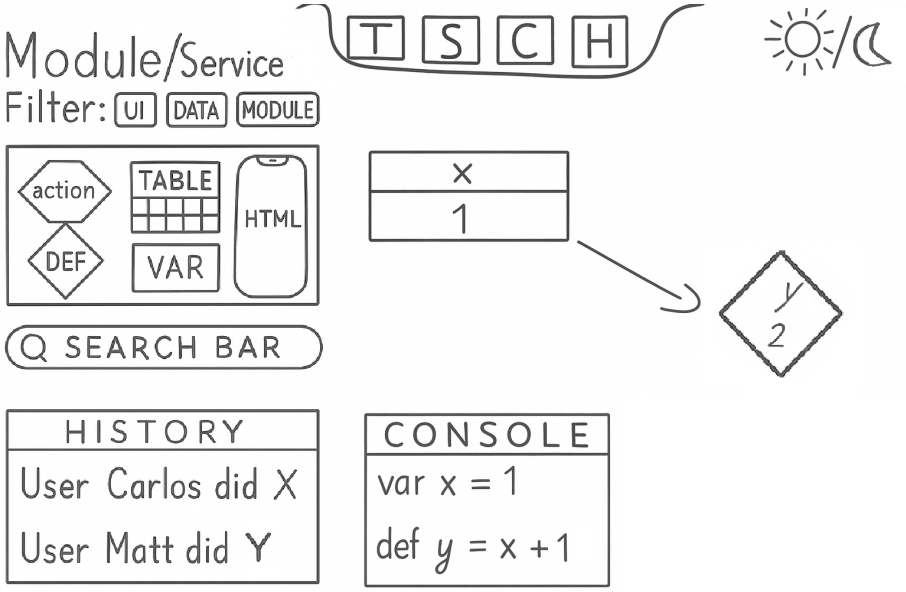


Figure 4.3: Alternative sketch with floating menus.

language and its syntax. Variables, while simpler, might also benefit from a dedicated creation menu to streamline their setup.

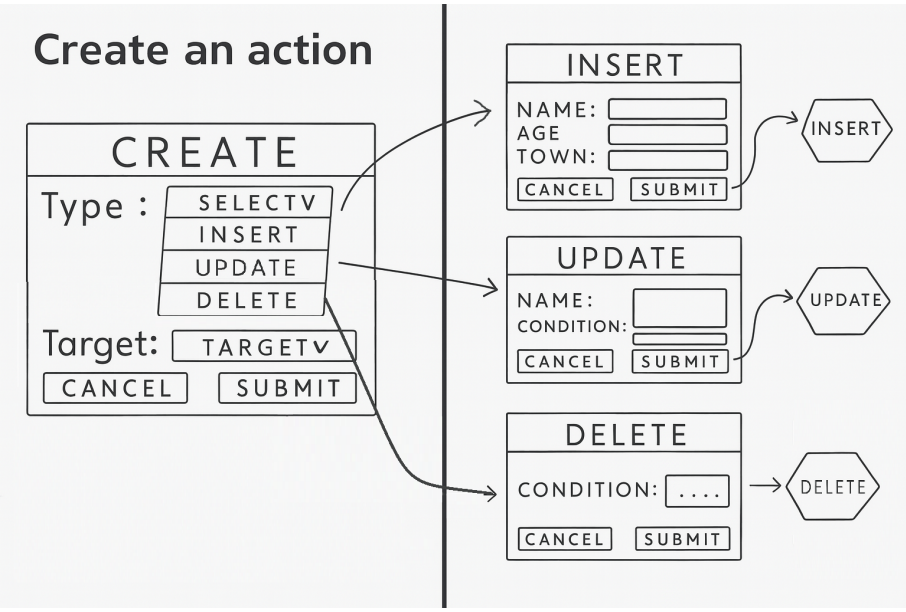


Figure 4.4: Sketch of creating an action

For this reason, we focused on designing dedicated menus for these components, as shown in sketches 4.4 and 4.5. The table menu provided a text input for naming the table, along with a control to add columns one by one. Each new column could be configured with a name and type, allowing the user to incrementally define the table schema before placing it in the workspace.

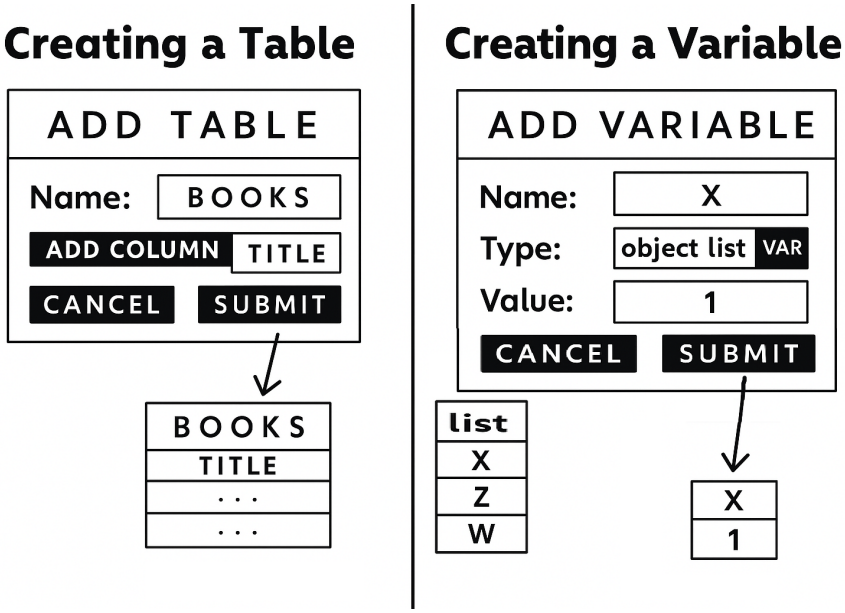


Figure 4.5: Sketch of creating a table and a variable.

The action menu included a dropdown to select the target of the action, which dynamically adjusted the menu options according to the selected component type. For example, when the target was a table, the available actions included inserting new rows, updating existing entries, deleting specific rows, or clearing the entire table. Additional fields and conditions appeared contextually to guide the user in specifying the operation parameters without needing to remember the exact language syntax.

The variable menu was kept simple, providing a field to specify the variable name, select the data type, and optionally set an initial value. This allowed users to quickly create new state variables and link them to other components in the application.

The remaining components, such as HTML pages and modules were deemed straightforward enough to be created directly via drag-and-drop from the toolbox, without requiring a simplified specialized creation menu. Further in the development process, we realized that the definition node would also benefit from a dedicated menu, as it was not intuitive to novice users how to create them, so we created a menu similar to the one for actions, allowing users to select the target node and specify the computation logic.

To explicitly state the requirements, we identified a list of tasks that the user should be able to perform in the interface, which would help us define the requirements in terms of functionalities. We identified the following tasks:

- Create various nodes, such as variables, definitions, actions, database tables, views and HTML pages via drag and drop.

- Freely manipulate the created nodes, allowing users to move them around the interface.

- Connect components by dragging an arrow from one component to another or to the empty environment, resulting in varying interactions. For example, connecting a node to a HTML represents a display of information concerning the node on the HTML page, while connecting a variable to a definition represents inserting that variable into the environment of said definition, allowing computations of a value derived from that variable.
- Support grouping various components into modules.
- Ensure that a node can only be deleted if it has no interactions that compromise the system as a whole.
- Support the editing, deletion, and interaction of nodes.
- Support executing actions that trigger data updates.
- Support search and filter mechanisms that allow users to quickly find and focus on specific components within the workspace.
- Support writing language code in the console, causing state changes in the interface.

Before proceeding with the development, we wanted to sketch an example of how a simple application could be created in the proposed interface, showcasing some the functionalities proposed above. The example, shown in sketch 4.6, illustrates the process of building a small message management application in six steps: creating a database table to store messages, defining an action to insert new rows into the table, executing the action to add data, creating a definition to compute the size of the table, adding an HTML page to present the data, and finally applying a filter to display only HTML components. This sketch was intended to provide a concrete visualization of how the development workflow would unfold in the system.

4.4 Development

The prototype was developed through an iterative process, where each cycle introduced new functionality while refining the interface according to user-centered principles. Each iteration built upon the previous one, gradually increasing the expressiveness, usability, and flexibility of the system.

First Iteration - Establishing the Foundations

The first cycle focused on laying the groundwork for the prototype, ensuring that the most essential components were available and functional. At this stage, the goal was to create a minimal but complete environment where users could already test simple interactions.

Key features introduced in this iteration:

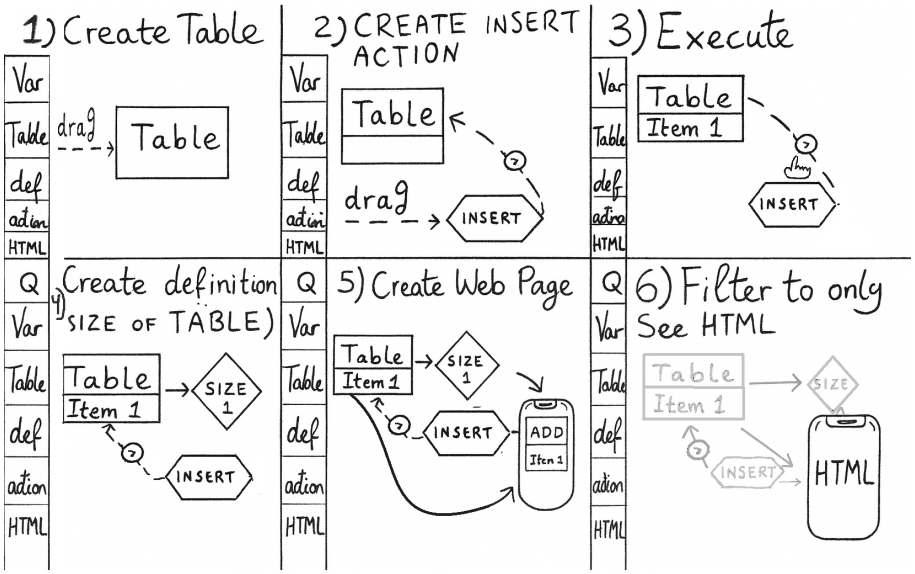


Figure 4.6: Sketch example of developing a system that manages messages.

- Basic interface layout with a central workspace, toolbox, and console.
- Creation and management of the core node types: variables, definitions, actions, and HTML views.
- Drag-and-drop support for placing and organizing nodes in the environment.
- Execution of simple actions (e.g., incrementing a counter).
- Immediate visualization of values inside the environment without requiring an HTML view.
- Console integration for displaying generated code and supporting direct code input.

Second Iteration - Expanding Interactivity

The second cycle moved beyond the initial foundations to enrich interaction and support more complex constructs. The focus here was on extending expressiveness and enabling automatic code generation through node connections.

Enhancements included:

- Introduction of advanced structures such as tables and modules.
- Support for connections between all node types, establishing the full dependency graph.
- Automatic generation of HTML fragments from connected nodes.
- Parameter handling in actions, with modal dialogs automatically generated for values prefixed with `in`.

- Dedicated creation menus for complex nodes like actions, definitions and tables to streamline user interaction.

Third Iteration - Refinement and Usability

The final development cycle emphasized refinement of the user experience. By this point, the system already supported most of the intended functionality, so improvements focused on making the interface more usable and adaptable to different contexts.

The main contributions of this iteration were:

- More complex HTML rendering, including lists and dynamic data-bound elements.
- Node rearrangement tools and layout improvements for better workspace organization.
- Filtering and search mechanisms to focus on specific node types and simplify navigation.
- Touchscreen compatibility and responsive design, ensuring the prototype could be used across different devices.

5

Technical Approach

In this chapter, we describe the technical approach taken to implement the visual programming environment. We begin with an overview of the system architecture and behavior of the environment, followed by a detailed explanation of how the visual language maps to the underlying programming constructs. Finally, we describe the main components of the interface and their functionalities.

5.1 Overview

The architecture of the system follows a reactive, client–server model. A visual interface runs in the browser, allowing users to define and modify applications through drag-and-drop components or textual commands. This interface is synchronized with a runtime system [29], which evaluates user-submitted statements, manages application state, and propagates reactive updates in real time. Communication is handled over WebSockets, ensuring low-latency, bidirectional synchronization.

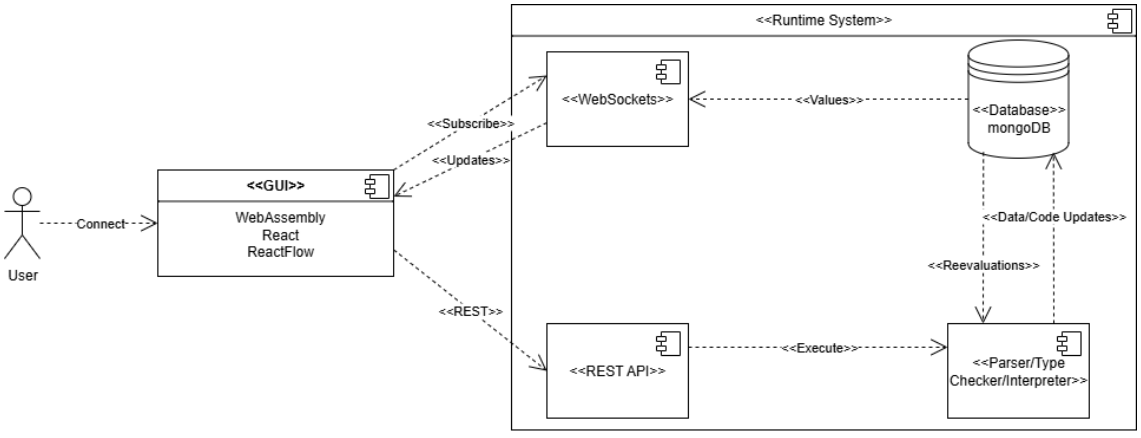


Figure 5.1: Overview of the system architecture. Adapted from [29].

When a user connects to the interface, a unique namespace is generated, providing an isolated environment for that session. The runtime maintains a subscription list of active namespaces, each associated with a set of connected clients. Upon connection, the frontend

subscribes to the generated namespace via WebSockets and receives the corresponding state and code history. As the user incrementally adds or updates components, changes are streamed through the same channel, keeping the interface in sync with the backend.

Each addition or modification of a language component represents a code statement, which is sent to the runtime for evaluation. The runtime processes these statements, updating the application state and computing any necessary reactive changes. The results are then pushed back to the interface, which updates the visual representation accordingly.

In the original prototype, described in section 3.2, the user would write statements such as:

Listing 5.1: Example of user statements in the original prototype

```
var counter = 0
```

This would be sent to the runtime in the form of a POST request:

```
POST { "exp": "var counter = 0" }
```

If the statement was valid, the code would be stored in a database, the runtime would respond with a `200 OK` status, and WebSockets updates would be received with information regarding all current components. The interface would then update to show the new variable and its value. If the statement was invalid, the runtime would return an error message, which the interface would display to the user.

Our interface builds upon this foundation by adding a visual layer that abstracts away the need for direct code input. Users can create and manipulate components through drag-and-drop interactions, with the interface generating the corresponding code statements behind the scenes. As the user builds their application visually, an incremental stream of code statements is sent to the runtime, ensuring that the visual representation and the underlying code remain synchronized.

For example, consider the following sequence of statements:

Listing 5.2: Statements representing a simple counter application

```
POST { "exp": "var counter = 0" }
POST { "exp": "def plus5 = counter + 5" }
POST { "exp": "def inc = action { counter := counter + 1 }" }
```

To achieve these components, the user can drag and drop the corresponding visual elements from the sidebar into the Environment. For instance, dragging a *variable node* into the Environment prompts a modal window where the user specifies the variable name and initial value. Once confirmed, the system generates the equivalent code statement and sends it to the runtime. If no syntax errors occur, the client receives the update via WebSockets and generates the corresponding node, such as the counter variable shown in Figure 5.2.

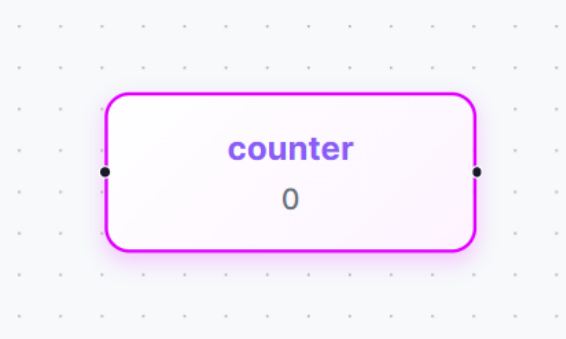


Figure 5.2: A counter variable created through the interface and rendered in the Environment.

Once the variable node is created, additional elements such as definitions and actions can be added. Definitions require an existing element to depend on, while actions require a target element that they can modify. In this case, the definition node computes the numeric expression `counter + 5`. For the action, the modal allows the user to define both the target (the `counter` variable) and the assignment operation itself (incrementing the value by 1). All nodes contain their name and varying information depending on their type, which is further described in section 5.2.

After processing all the statements from Listing 5.2, the Environment appears as shown in Figure 5.3.

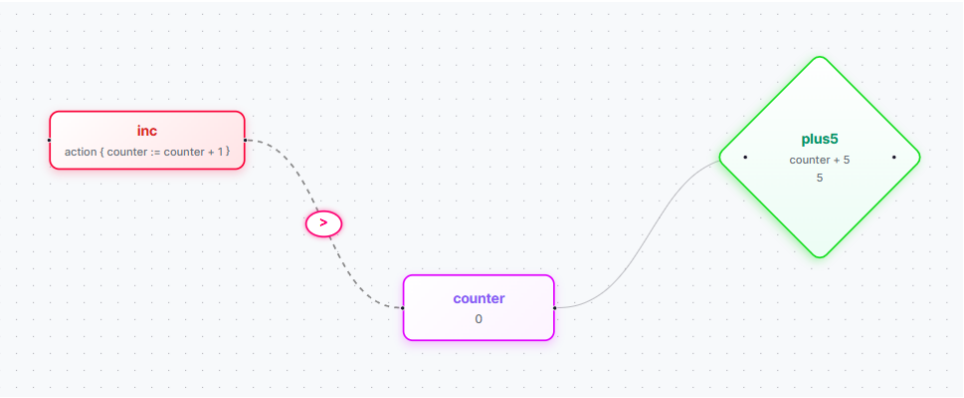


Figure 5.3: Environment state after processing the counter example.

As nodes are instantiated, the interface sends a `GET` request per node to the runtime to compute the dependencies between components.

The runtime responds with a list of dependencies, which the interface uses to create directed edges between nodes, visually representing their relationships. In this case, the definition node depends on the variable node, while the action node targets the variable node, producing a default edge and an action edge, respectively. These edges are aimed at helping users understand how components relate to each other and how data flows through the application, further explained in section 5.2. If the user were to try and attempt to delete the variable node, generating the following request:

```
POST { "exp": "delete counter" }
```

The system would first check for any dependencies. Since both the definition and action nodes depend on the variable, the runtime would reject the deletion request, returning an error message indicating that the variable cannot be deleted due to existing dependencies.

In order to execute the action, the user can click the button embedded in the action edge, which sends a `POST` request to the runtime to execute the action:

```
POST { "exp": "do inc" }
```

The runtime then processes the action, updating the value of the `counter` variable from 0 to 1. This change is then propagated back to the interface via WebSockets, which updates the variable node to reflect the new value. The definition node automatically recalculates its value based on the updated counter, changing from 5 to 6.

It is important to note that some actions may require parameters, which the user must provide before execution. In the present example, the action `inc` requires no parameters and is therefore executed immediately upon clicking the button. Consider a variation of the counter action defined as:

```
def sum = (inNum) => action { counter := counter + inNum }
```

In this case, `inNum` is a parameter. Parameters are typically identified by the parser when analyzing the action's specification. In particular, parameters prefixed with `in` are automatically recognized as inputs. If the user were to click the action edge button for this modified action, a modal dialog would appear, prompting the user to enter a value for `inNum`, as shown in Figure 5.4. The action would only execute after the user provides this input and confirms the action.

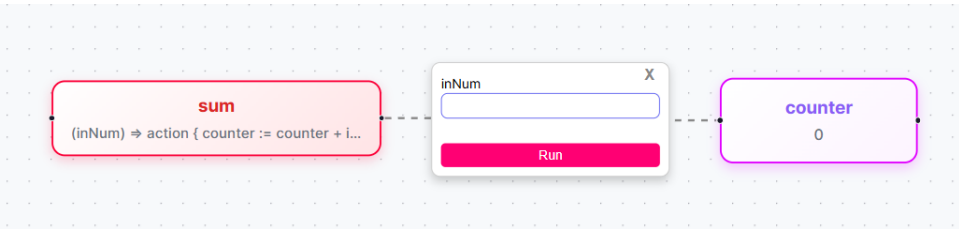


Figure 5.4: Modal dialog prompting the user to enter a value for the action parameter `inNum`.

The previous example illustrated the construction of a simple application within the Environment. However, interacting with the program solely through the diagram is limited. To enable direct user interaction, we introduce the *HTML page node*, which provides a web-based interface for visualizing data and triggering actions.

When first created and instantiated, the HTML page consists of an auto-generated minimal container with a user defined name, generating the following code statement:

```
def counterPage =
  <div class="template1" id="counterPage">
    <h3 class="main-title">"counterPage"</h3>
  </div>
```

As other nodes are connected to it, fragments of HTML are automatically generated and inserted into the page definition. This process allows the interface to evolve incrementally as the user builds the program.

For instance, connecting a variable node such as `counter` produces a block that displays its current value:

```
<div class="single-value" id="counter">
  <label>"counter"</label>
  <div>counter</div>
</div>
```

Definitions are handled in a similar way: attaching a definition node results in a labeled block that shows the derived value. In the case of actions, the generated fragment provides interactive elements. A simple action creates a button that, when pressed, triggers the corresponding update:

```
<form>
  <button doaction=(inc)>
    "inc"
  </button>
</form>
```

If the action requires parameters, the environment automatically generates input fields so that users can supply arguments before execution. Tables, on the other hand, are rendered as lists of rows and columns, making their contents directly visible in the page. Once the HTML page is fully constructed, it can be previewed directly within the interface by clicking a button on the node, as shown in Figure 5.5.

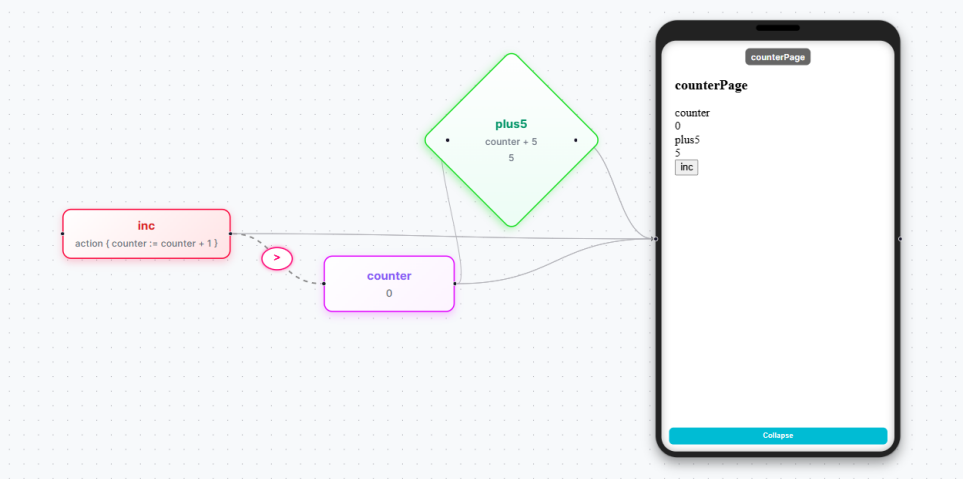


Figure 5.5: Environment of counter example with an HTML page.

5.2 Visual Language

A key component of our system is the visual language that underpins the programming environment. Our visual language is designed to map Meerkat language constructs to visual components, enabling users to create applications without needing to write traditional code. This section describes the vocabulary and the rationale that guided its design. We then describe the grammar of our visual language, which is essential for understanding how elements interact with each other.

When deciding how to represent the language, we considered different paradigms commonly used in visual programming, such as flow-based models and block-based representations. While these alternatives are effective in their own contexts, they were not entirely suited to our goals. Block-based languages are useful for reducing syntax errors and introducing programming concepts, but they impose a rigid structure that does not naturally align with the semantics of the Meerkat language. Flow-based languages focus primarily on sequencing and control flow, which only partially captures the nature of our system. Since the Meerkat language inherently models programs as a DAG, where nodes represent elements of the system and edges represent dependencies between them, we determined that a graph-based representation would be the most faithful and accurate way to visualize its computational model. This choice ensures that the visual constructs remain consistent with the underlying language semantics, while still providing an intuitive interaction model for users.

5.2.1 Vocabulary

The vocabulary domain of our visual language consists of the components that represent the underlying DAG of the programming language, which are the nodes and edges.

The nodes represent a language component of the language described in section 3.1. Since each node has a unique identifier and carries varying information depending on its type, we needed a way to easily distinguish between them in an intuitive fashion. We initially focused solely on distinguishing nodes by assigning different shapes depending on the language component. This approach is slightly flawed in the fact that it does not scale well, as the number of shapes is limited and if the language expands, the design becomes unsustainable, as identified in previous work [3], and also in the fact that some shapes are not optimal to display information.

As discussed in section 4.3, we aimed to represent variables, definitions, actions, database tables, HTML pages and modules. To understand how to properly represent them, we first must consider the domain of programming elements that our nodes will represent:

- **State Variables** Used to store basic pieces of information within the system. Three types are supported:
  - *Integer* – represents whole number values;

  - *String* – represents sequences of text;
  - *Boolean* – represents truth values (`true`/`false`).
  - *List* – represents ordered collections of values, which can be of any type, including other lists.

- **Definitions** Nodes that derive new values or views from existing data. Examples include:
  - *Numeric Expression* – computes a transformation over integer variables;
  - *Size* – returns the number of rows in a table;
  - *Map* – applies a transformation to each entry of a table, producing a new collection.

- **Actions** Operations that modify system state or trigger changes. The available actions include:
  - *Assignment* – updates a variable with a new value or expression;
  - *Insert* – adds a new row into a table;
  - *Delete* – removes a set of table entries;
  - *Clear* – empties all rows from a table;
  - *Update* – alters one or more entries in a table;
  - *Composition of Operations* – combines multiple actions into a sequence that executes step by step.

- **Database Tables** Structured collections of rows and columns, where each row represents an entry and columns define the fields of that entry.

- **HTML Pages** Elements of the interface that render web pages, allowing data and interactions to be presented directly to the user.

- **Modules** Encapsulated groups of components that can be parameterized and reused, supporting modular design of larger applications.

Despite the concerns regarding utilizing shapes, we still considered it an acceptable approach for representing some of the nodes (e.g., definition nodes are rhombus-shaped, while HTML nodes resemble a mobile phone). This visual differentiation helps users quickly identify the nature of each component. Variable, Definition, and Action nodes display their names and values. Other nodes, such as HTML pages, show their names along with a button that loads an HTML page when clicked. Table nodes display their name and the table's rows and columns, allowing users to view the data structure directly within the Environment. Modules simply display their name and a button to access the module's contents.

In the case of variables and actions, both of them have the same shape of a rectangle when represented in the Environment, as the use of the originally thought of shape in actions, which was the hexagon, would hinder visibility of information. So, in order to distinguish them and other nodes even further, we assigned a unique color to each node type. Variables are represented in purple, definitions in green, actions in red, database tables in blue, and modules in yellow. The case of HTML pages is slightly different, as they are represented as a mobile phone shape in the Environment, despite being depicted with an orange rectangle in the toolbox menu, which we deemed a more intuitive representation of the component since the user actively interacts with the content inside the phone screen.

The choice of colors and shapes was not made with any particular logical cognitive reasoning, but rather based on our own preferences and the fact that we wanted to create a visually appealing interface. We acknowledge that this may not be the most optimal solution for all users, as some may have difficulty distinguishing between certain colors or shapes. Nonetheless, we considered that this color and shape combination was sufficient to distinguish between the different node types in our visual language. In Figure 5.6, we present examples of the visual symbols used in the Environment. Although the symbols on their own may not appear immediately intuitive, their association with the corresponding elements in the toolbox makes their meaning clear. Moreover, any initial difficulties are expected to quickly fade as users become familiar with the interface through practice.

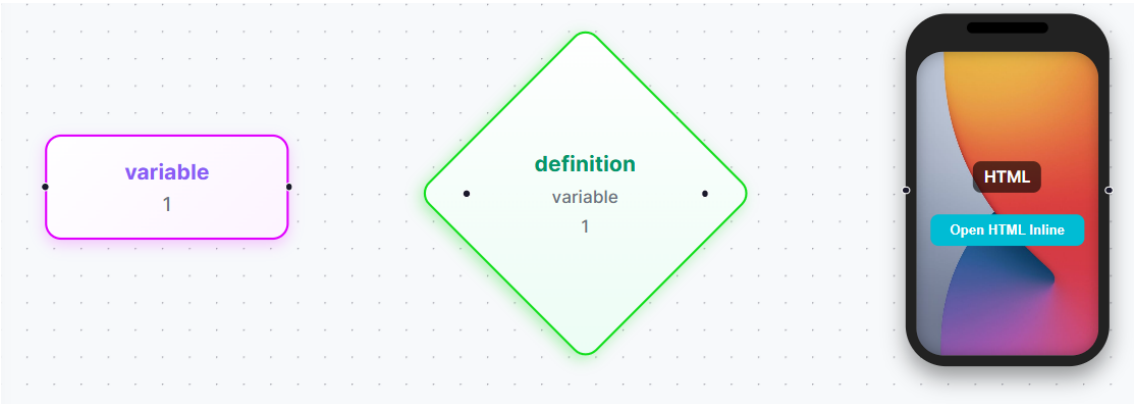
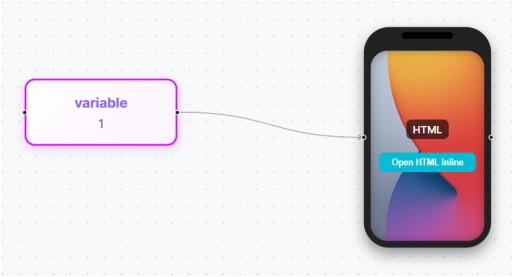


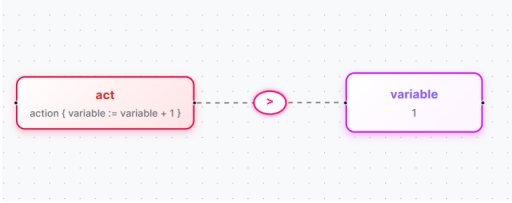
Figure 5.6: Examples of visual symbols in the Environment: a Variable node, a Definition node, and an HTML page node.

The second component of the vocabulary domain is the edges, which represent dependencies between nodes. These edges are directed, with a defined source and target, and are visually represented as arrows in the Environment. As illustrated in Figure 5.7, there are two types of edges: *action edges* and *default edges*. Action edges are edges represented with dashed lines that link actions to other nodes, such as variables or definitions, and include an embedded button that allows the user to directly trigger the corresponding action, visually displaying the resulting flow of data. Default edges, in contrast, are simple grey directed edges that connect all other node types and primarily serve to represent structural dependencies, helping users understand relationships between components

and determine whether a node can be safely deleted.



(a) Default edge representing a data dependency.



(b) Action edge representing an executable action dependency.

Figure 5.7: Types of edges in the Environment.

5.2.2 Grammar

The visual grammar defines the set of rules that constrain how nodes and edges can be composed within the Environment. While the vocabulary specifies the visual symbols and their categories, the grammar governs their valid interactions, ensuring that connections between nodes are semantically meaningful.

In our language, dependencies are always represented as directed edges, where the left-hand element acts as the source and the right-hand element as the target. These dependencies encode both data flow and the possibility of state modifications.

At a high level, the following rules describe the general structure of the language:

- Variables → Definitions
- Tables → Definitions
- Definitions → Definitions
- Definitions → Actions
- Actions → Variables
- Actions → Tables
- Variables, Tables, Definitions, Actions, HTML Pages → HTML Pages

These rules ensure that stateful elements, such as variables and tables, can serve as inputs to derived computations (definitions). Definitions themselves can be chained, allowing for expressions to be built on top of other expressions. Definitions may also feed actions, providing the necessary values or conditions for their execution. Actions, in turn, are the only elements capable of directly modifying state, either by updating variables or by altering table contents. Finally, any element in the system, can be routed to an HTML page, either for visualization in the case of data or interaction in the case of actions

exposed as triggers in the user interface. This means that any node can be interpreted as HTML markup and rendered directly to the user. Importantly, this domain also includes the HTML page element itself, allowing for a hierarchical construction of interfaces where one HTML page may depend on other pages or elements to compose its content.

To refine the dependency domain further, we specify node-level grammar rules that capture the semantics of each element type:

- Number → Numeric Expressions
  (a numeric expression depends on one or more numbers)
- Table → Map, Size
  (a table can be mapped or counted to produce new results)
- Expression (Definition) → Assignment, Insert, Delete, Update
  (definitions can provide values or predicates to actions)
- Assignment → Variables
  (an assignment modifies the value of a variable)
- Insert, Delete, Update, Clear → Tables
  (actions modify the state of a table)
- Variables, Definitions, Tables, HTML Pages → HTML Pages
  (any data element may be displayed on a web page)
- Actions → HTML Pages
  (actions are exposed as triggers in the interface, rather than modifying the page itself)

Each rule reflects a design choice intended to preserve semantic consistency. For example, a numeric expression can only depend on numbers, while a map or size definition can only depend on a table. Similarly, actions like Insert or Delete are restricted to modifying tables, while assignments are limited to variables. These constraints prevent invalid connections, ensuring that relations without semantic meaning (such as an Insert action targeting a number) cannot be expressed in the system.

In previous work [3], when users attempted to establish an invalid dependency, the system still displayed the connection but highlighted the target node in grey to indicate that the relation was semantically invalid. In contrast, our approach enforces grammar constraints at the interaction level, preventing users from creating invalid dependencies altogether. As such, these connections are never visually represented, leading to a clearer and more consistent Environment.

By enforcing these rules, the grammar ensures that the visual Environment remains faithful to the semantics of the underlying language. It guides users toward valid constructions while preventing nonsensical or contradictory dependencies, ultimately making the system both expressive and semantically consistent.

5.3 Used Technologies

The prototype interface was implemented in *JavaScript* using React, with React Flow [54] handling the node-based editing features. React Flow is a library for creating interactive graphs, providing tools for node manipulation, edge creation, and real-time layout updates. User interactions with the graph, such as adding or connecting nodes, are captured in React Flow and sent to the backend.

Due to initial aspirations to connect to a distributed *Rust* backend, further discussed in section 8.1, part of the interface was implemented in *Rust* and compiled to *WebAssembly*. This section communicates with the backend via WebSockets and REST, processes updates, and synchronizes the program state with the React Flow interface to ensure that changes are reflected instantly in the visual workspace.

5.4 Visual Programming Environment

Our visual programming Environment is designed to facilitate the development of web applications through a user-friendly interface that abstracts away the complexities of traditional coding.

The interface consists of four main elements:

- **(1) Environment:** A dynamic canvas [54] where components are instantiated either via direct code input or drag-and-drop from the Sidebar. It visualizes node dependencies and serves as the primary space for composing and editing applications.
- **(2) Sidebar:** A structured panel for browsing and inserting components such as variables, definitions, actions, and views.
- **(3) Console:** A command-line interface for users familiar with the language, enabling direct code execution without graphical input.
- **(4) Search/Filter Bar:** Utility bars that assist in locating and filtering components within large applications.

Figure 5.8 displays the interface structure. The **Environment** is the core of the interface, where users can visually manipulate components. New components can be created by dragging items from the **Sidebar** into the Environment, or by typing commands in the **Console** which allows users to write and execute code directly. This Console is particularly useful for experienced developers who prefer traditional coding methods or need to perform complex operations that may not be easily achievable through the visual interface. The Console also assists users in learning the underlying language by displaying the equivalent code for creating or modifying components and executing actions in the Environment. The Console can also be used to paste whole application code, which is then parsed and executed in the Environment. This allows users to quickly test and iterate

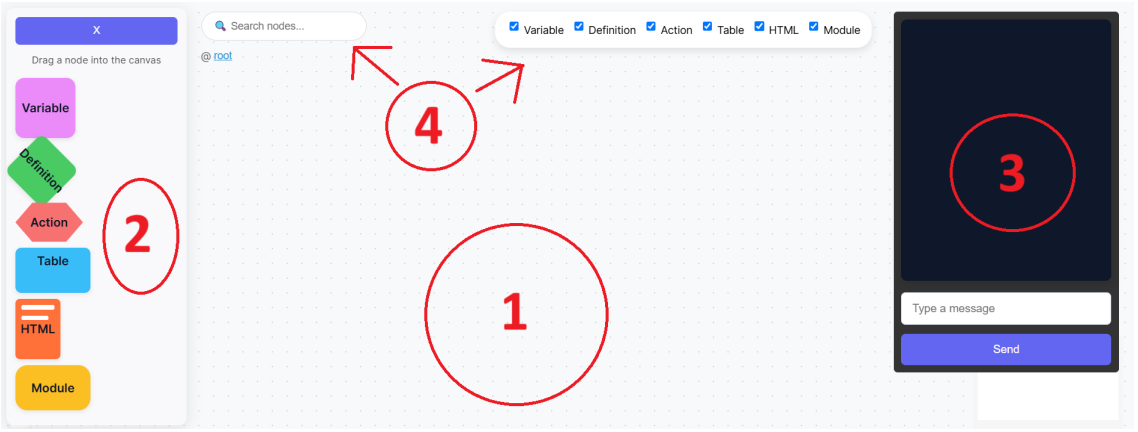


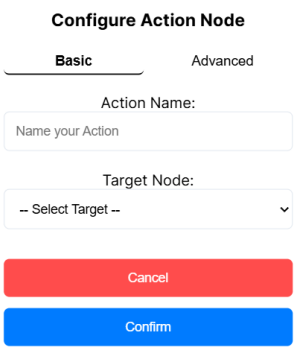
Figure 5.8: Interface structure.

on their code without needing to manually create each component through the visual interface.

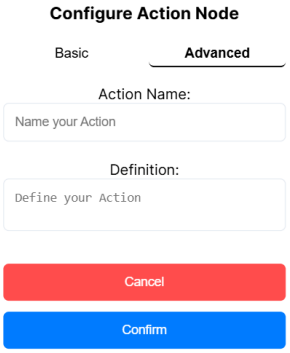
Using the Console requires knowledge of the underlying language, whereas the drag-and-drop functionality is designed to be intuitive for users with varying levels of programming experience. Each component in the Sidebar is represented by a unique symbol, a label, and a distinct color, making it easy to identify different types of components at a glance.

Dragging an item from the Sidebar into the Environment prompts a modal dialog for specifying the component's name and relevant parameters (e.g., for a definition or action). In some modals, there are both *Basic* and *Advanced* tabs, allowing users to choose between a simplified or detailed view of the component's properties. The *Basic* tab narrows down the required inputs for creating the component, while the *Advanced* tab provides full flexibility at the cost of requiring familiarity with the language. This design caters to both novice and experienced users, enabling them to interact with the system at their preferred level of detail, as illustrated in Figure 5.9. Upon confirmation, a `POST` request is sent to the runtime with the component's specification. The runtime responds with updates over WebSockets, which result in the new component being instantiated in the Environment in the form of a node.

Users can modify or delete nodes by double-clicking them, which opens a similar modal dialog. Confirming an edit or deletion sends a corresponding `POST` request to the runtime. Whenever components are added or changed, the interface issues a `GET` request to compute the node's dependencies. These dependencies determine which edges are instantiated in the Environment, visually representing the connections to other nodes. When an action node targets another structure and a valid dependency is detected, an *action edge* is created. All other types of dependencies are displayed via a default edge. Clicking an action edge triggers a `POST` to execute the associated action, accompanied by an animation on the edge to indicate execution. In the case that the action requires parameters, a modal dialog appears in the middle of the edge, prompting the user to enter



(a) Modal dialog with the *Basic* tab.

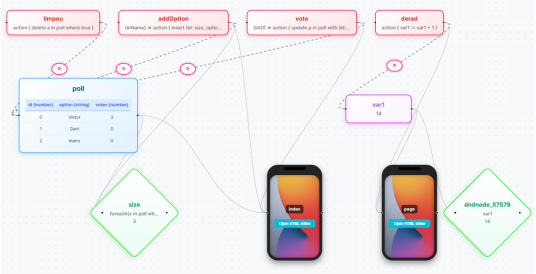


(b) Modal dialog with the *Advanced* tab.

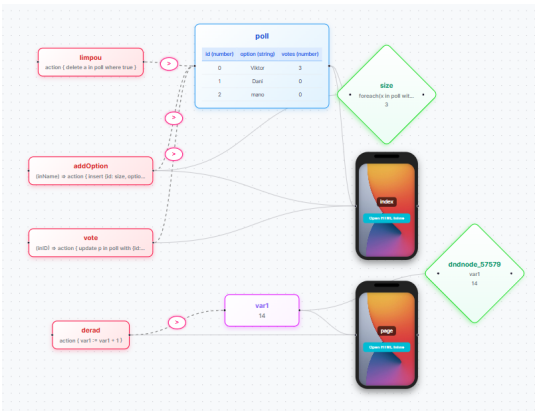
Figure 5.9: Comparison of the *Basic* and *Advanced* tabs when creating a component.

the necessary values. Once the action is executed, the Environment updates to reflect any changes in the state of the application.

As nodes and edges are generated, a visual layout algorithm organizes them in the Environment, ensuring a clear and intuitive representation of the application's structure. Figure 5.10 represents the Dagre layout algorithm [46] applied to different orientations, which is designed to minimize edge crossings and maintain a logical flow, making it easier for users to understand the relationships between components. Users can also manually rearrange nodes by dragging them, allowing for further customization of the visual layout.



(a) Vertical orientation.



(b) Horizontal orientation.

Figure 5.10: The Dagre layout algorithm applied in two orientations.

To simplify adding definition nodes, a basic quality of life feature was implemented where users can drag an edge from an existing variable node to an empty space in the Environment. This automatically creates a definition node initialized with the variable's value and an edge between the variable and the definition, allowing users to quickly create definitions without manually entering the value.

As in standard code editors, the interface contains a search bar that allows users to quickly locate components by name. This is particularly useful in larger applications where

the number of components can become unwieldy. When a search begins, a dropdown list appears with all components containing the given keyword, and once selected, the Environment scrolls to the corresponding component for quick access. A similar feature is available via the built-in minimap inside the Environment, which provides a simplified overview of the entire application and allows users to navigate directly to components by clicking on them.

The interface also includes a filter bar that allows users to adjust the abstraction level of the displayed components. This is particularly useful for managing complexity in larger applications, as it enables users to focus on specific aspects of their code without being overwhelmed by irrelevant details. For example, users can choose to view only data structures, as shown in Figure 5.11.

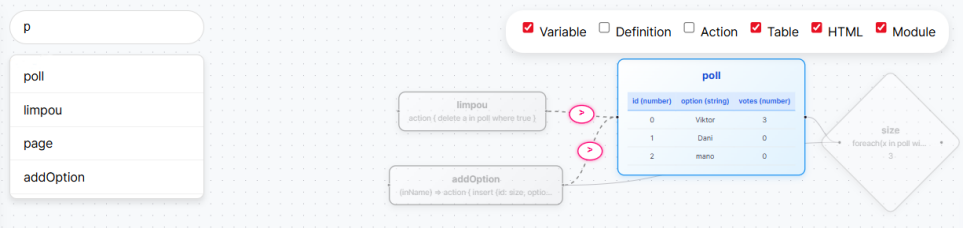


Figure 5.11: Search and filter features, aiding application visualization.

The interface also supports *Modules*, allowing users to group related elements into reusable and nestable isolated environments. Modules can accept parameters that dynamically influence the behavior of their internal nodes, enabling scoped and context-sensitive logic. For example, the value of a variable inside a module may depend on parameters defined by its parent module. When a module is created, it is instantiated as a new node in the Environment, containing its name and a button to access the given module.

When a user clicks on a module node, the Environment updates to display the contents of that module, effectively switching the context to that module. This allows users to work within the module while maintaining access to the main application context. The system automatically tracks the current module context, ensuring that any changes made within a module are isolated from the rest of the application until explicitly defined otherwise.

To navigate between modules, the interface provides a directory-like structure at the top of the Environment. It shows the current module's name and a breadcrumb trail of parent modules leading it to the root directory, illustrated by Figure 5.12. Clicking a parent module navigates back to it. This allows users to work within modules while maintaining easy access to the main application context.

When a parameterized module is accessed, a modal dialog prompts the user to specify the parameter values. This mechanism not only allows users to retrieve values defined in terms of the provided parameters but also to define new values that can be utilized within the module. If no parameters are specified, all components inside the module expose all possible values based on each parameter's default state. To refine this set, the user can enter a parameter keyword in the modal, and the system filters the displayed component

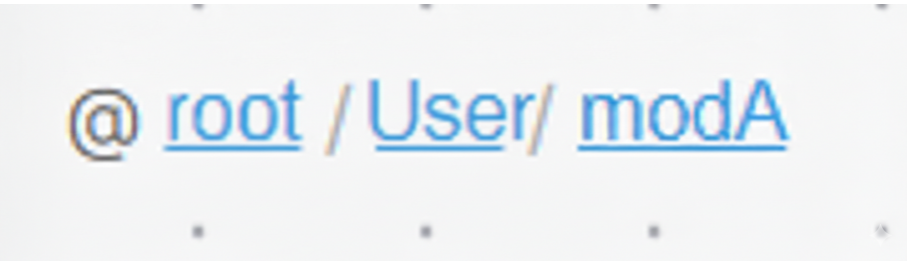
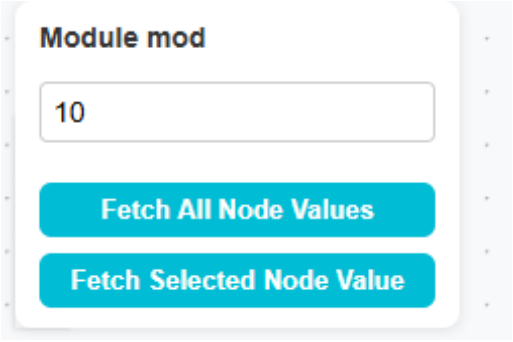
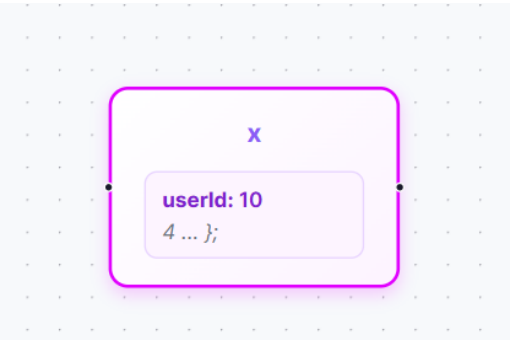


Figure 5.12: Breadcrumb navigation in the Environment (@root/User/modA), highlighting that the user is currently in the modA module.

values accordingly. This allows users to focus on relevant components and quickly find the information they need.



(a) Parameter modal with userId = 10



(b) Variable node with x = 4 for userId = 10

Figure 5.13: Parameterized module workflow

When a user creates or updates a component while specifying a parameter in the parameter modal, the system associates the component's properties with the given parameter value. For example, as illustrated in Figure 5.13, defining `var x = 4` with the parameter `userId: 10` assigns this value of `x` only within the context of `userId = 10`. This mechanism ensures that component behavior and data can be tailored to specific parameterized contexts .

Finally, the system enforces two-way interactivity: changes made through the Console are immediately reflected in the Environment, and vice versa. This ensures that both code-driven and visual workflows remain synchronized, offering a fluid, real-time development experience entirely within the browser.

6

Examples and Case Studies

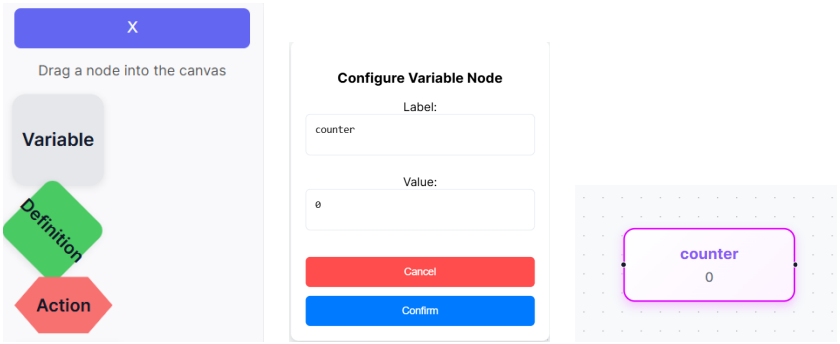
In this chapter, we explore several examples and case studies that illustrate the practical applications of utilizing the interface, providing insight into the web application development process.

6.1 Example 1: Counter Application

In this example, we create a simple counter application that allows users to increment and reset a counter value. To start, we first need to create a new **variable** to store the counter value. As shown in Figure 6.1a, this is done by dragging a new **variable** box into the **Environment**. This action triggers the appearance of a modal box (Figure 6.1b), where we define the `variable`'s name and initial value. For our counter application, we name it `counter` and set its value to 0. Once confirmed the following code statement is sent to the server:

```
1 var counter = 0
```

Once checked that no syntax errors or illegal operations occurred, the **variable** is created and rendered in the **Environment**, as shown in Figure 6.1c.



(a) Dragging a **variable** node into the **Environment**.

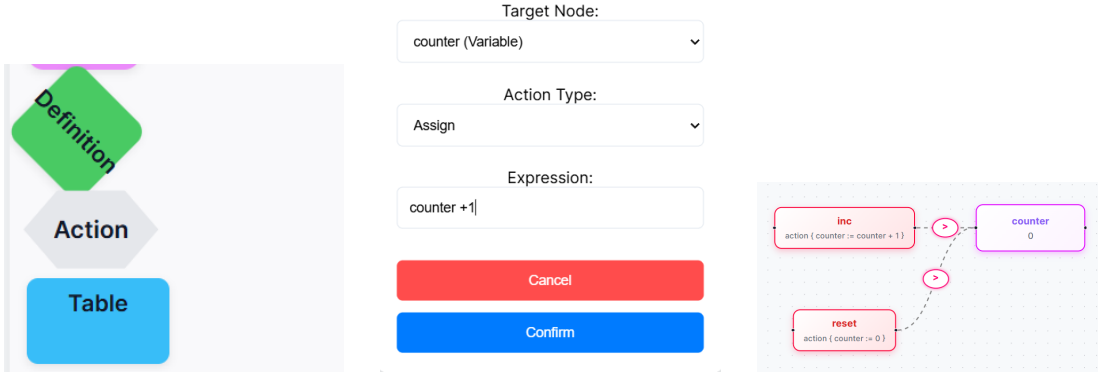
(b) Defining the `variable`'s name and initial value.

(c) The `counter` variable appears in the **Environment**.

Figure 6.1: Creation of the `counter` **variable**.

This `counter` variable is persistent: it stores application state and can be modified by actions. Its value is automatically available to other definitions or HTML views connected to it.

Next, we create an **action** to increment the counter. As illustrated in Figure 6.2a, the user drags a new **action** node into the **Environment**. In the modal that appears (Figure 6.2b), we define the action's name as `inc`, select the target `counter` variable, and specify its logic to increase the value by 1. Similarly, we create a `reset` action to set the counter back to 0 (Figure 6.2c).



(a) Dragging a new **action** node into the **Environment**.

(b) Selecting the action's target and defining its logic.

(c) The `reset` action appears in the **Environment**.

Figure 6.2: Creation of **actions** to increment and reset the counter.

By creating these actions, the following these code statements are generated:

```
def inc = action { counter := counter + 1 }
def reset = action { counter := 0 }
```

Actions are side-effecting definitions: when triggered, they modify the state of the variable they target. In this case, the `inc` action increments the counter, while `reset` restores its initial value. However, actions do not execute automatically; they must be triggered by the user.

With the **variable** and **actions** defined, we can freely execute actions by clicking the buttons located on the edges connecting action nodes to their targets. Clicking the button on the `inc` edge updates the counter immediately. To expose this logic through a web page, we introduce a **HTML View** node. We drag a new **HTML View** into the **Environment** and name it `index`. These nodes can be configured either by writing full manually writing HTML strings or by connecting nodes manually. In this example, the second approach is used (Figure 6.3), where both the `counter` and `inc` nodes are connected to `index`.

As node are connected to the HTML node, HTML code is generated incrementally to the page, resulting in the following code statement, producing an interactive web page displaying the counter and its value, and a button to execute the action:

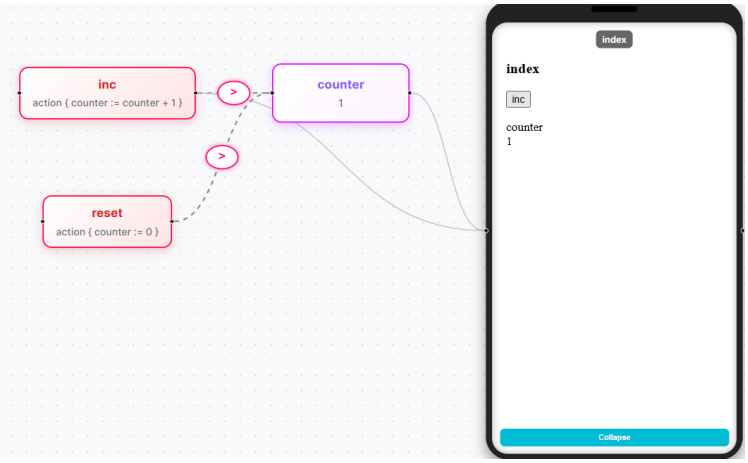


Figure 6.3: Final HTML page showing the counter and buttons.

```
def index =
  <div class="template1" id="index">
    <h1 class="main-title">"index"</h1>
    <form>
      <button doaction=(inc)>
        "inc"
      </button>
    </form>
    <div class="single-value" id="counter">
      <label>"counter"</label>
      <div>counter</div>
    </div>
  </div>
```

6.2 Example 2: Poll Voting Application

This example is slightly more complex, involving a poll voting application. Users can vote on predefined options and add new options, each tracked with a unique identifier. To begin, we create the necessary data structures. Dragging a **table** node from the **Sidebar** into the **Environment** and specifying the fields (Figure 6.4) generates the following code statement:

```
table poll {
    id: number,
    option: string,
    votes: number
}
```

Once confirmed, it is instantiated. The table defines the structure of the poll: every row corresponds to one voting option, with its identifier, name, and vote count.

When adding new voting options to our poll, we need to guarantee candidates with similar names and vote counts can be distinguished. To ensure unique identifiers, we add a `size` definition that tracks table length (Figure 6.5). Since there is a definition type to compute what we need, the user doesn't need to specify anymore logic besides the target

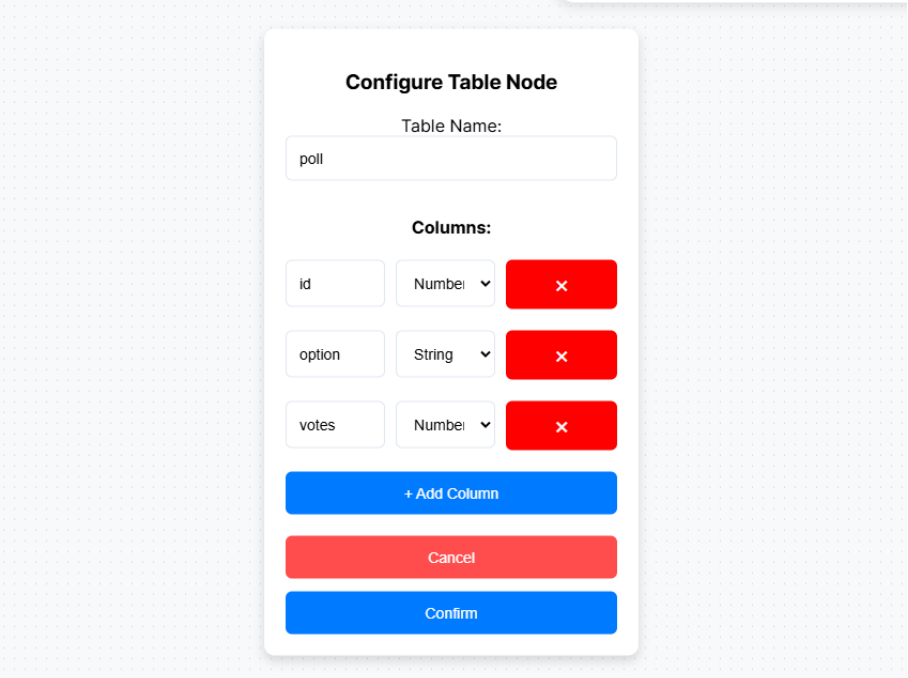


Figure 6.4: Creating the poll **table**.

to be tracked. This allows new entries to be assigned increasing identifiers.

```
1 def size = foreach(x in poll with y = 0) y + 1
```

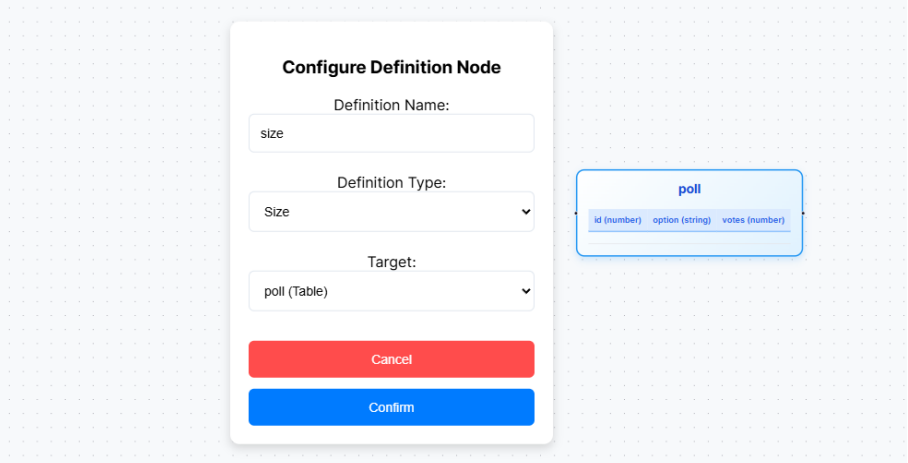


Figure 6.5: Creating a definition for unique IDs.

Next, we create **actions** to add new options and vote on existing ones. Figure 6.6 shows creating the `addOption` action.

The user selects the target **table** (`poll`) and parameters for insertion. We also use a similar method to create the the `vote` action, that increments votes for a selected option by locating its identifier in the table. The generated code statements are shown below.

Once created, both actions receive an action edge that connects them to the table, allowing the execution of their respective actions, as show in Figure 6.7.

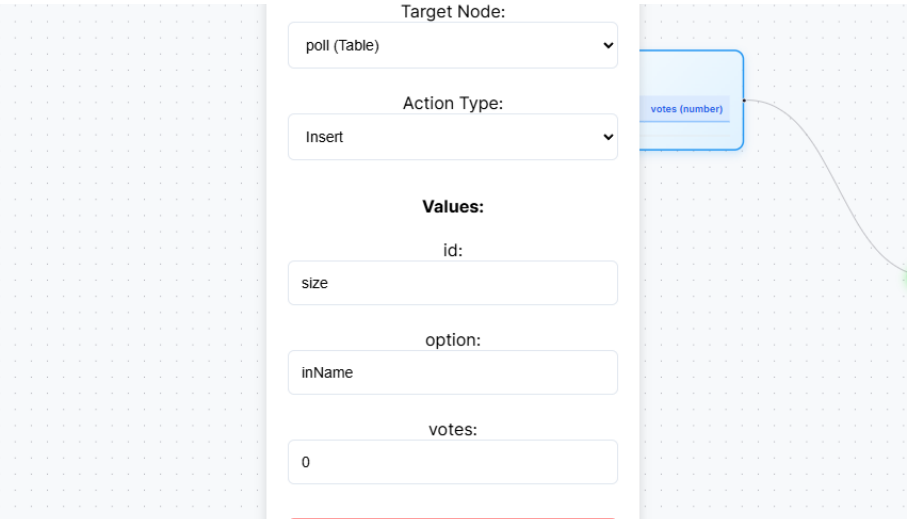


Figure 6.6: Creating the `addOption` action.

```
def addOption name =
  if name == "" then
    action {}
  else
    action {
      insert {
        id: nextId,
        option: name,
        votes: 0
      } into poll,
    }

def vote inID =
  action {
    update p in poll
    with {
      id: p.id,
      option: p.option,
      votes: p.votes + 1
    }
    where p.id == inID
  }
```

The user can execute actions and view the corresponding updates directly in the **Environment**. To make the poll accessible through a browser, we extend the system with an **HTML View** node. In contrast to the previous approach (Figure 6.3), where nodes were manually connected to the HTML node, here we manually paste the HTML code for the page, and the dependencies are automatically generated in the form of edges. Here is the code for both the `pollList` and `index` views:

In this example, we create two HTML pages, as shown in Figure 6.8. The first page is intended for regular users, allowing them only to view the poll and cast their votes. The second page functions as an admin interface, where, in addition to voting, new options can be added to the poll. The `pollList` definition generates a list of all entries in the

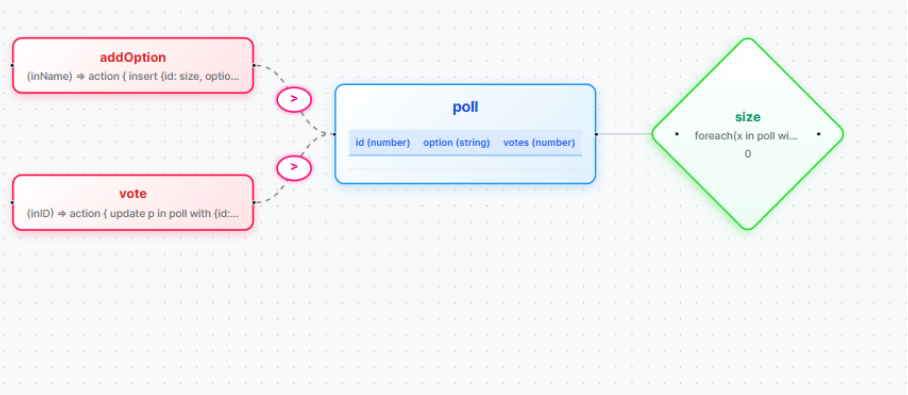


Figure 6.7: **Environment** with the `vote` action.

```
def pollList =
  <ul>
    (map(p in poll)
      <li>
        <b>(p.option)</b>
        <br/>
        Votes: (p.votes)
        <button doaction=(vote p.id)>Vote</button>
      </li>)
  </ul>

def index =
  <div>
    <h1>Vote in the Poll!</h1>
    pollList
    <form>
      <input id="newOption"/>
      <button doaction=(addOption #newOption)>Add Option</button>
    </form>
  </div>
```

`poll` table, displaying each option along with its current number of votes and a button to trigger the `vote` action. The `index` view integrates this list with a header and a form for adding new options. Submitting the form calls the `addOption` action, immediately updating both the table and the displayed list.

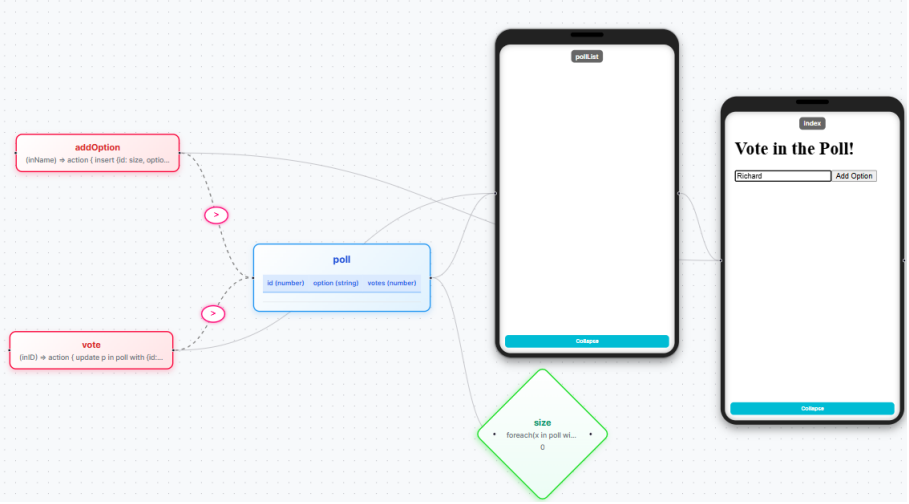


Figure 6.8: Creating a new poll entry via the HTML page.

7

Evaluation

In this chapter, we describe the user testing phase conducted to evaluate the usability of the visual programming environment developed in this work. We begin by outlining the objectives and evaluation method, followed by a description of the participants and tasks involved in the study. Finally, we present and analyze the results obtained from the testing sessions.

7.1 Objectives and Evaluation Method

The primary goal of this user testing phase was to evaluate the **usability** [39, 22, 23] of the graphical Live Programming (LP) environment developed in this work. The system is intended to lower the barrier of entry to reactive web application development through a visual interface and immediate feedback. To assess its effectiveness in practice, we conducted a **formative evaluation** [8] involving participants with programming backgrounds.

The evaluation followed a **task-based usability testing** [14] methodology, supported by the **think out loud protocol**, where participants verbalize their thoughts, decisions, and reactions while performing a series of representative tasks using the system , providing insight. This method helped us collect **qualitative insights** (e.g., confusion points, interface expectations). As participants completed the tasks, various **quantitative data** (e.g., task success rate, completion time, observed errors) were measured, allowing a detailed analysis of the usability performance of the system and the identification of potential areas for improvement.

In addition to observational data, participants were asked to complete a context-adapted version of the **System Usability Scale (SUS)** [6] questionnaire after finishing all tasks, providing a standardized assessment of perceived usability.

7.2 Participants

Participants were recruited on a voluntary basis by inviting students who expressed interest in the study. Participation was unpaid, and all participants were informed about the purpose of the study and their right to withdraw at any time. No sensitive personal data were collected, and given the minimal risk involved, formal ethical approval was not required.

The study involved a total of 10 participants, selected to reflect the target audience of the system: individuals with at least basic programming knowledge, but not necessarily experts in reactive or low-code programming. All participants were male, aged between 22 and 29, with an average age of 24. Nine were computer science students in their 3rd to 5th year of undergraduate or master's studies at NOVA School of Science and Technology or Instituto Superior Técnico, and one was a doctoral student at NOVA School of Science and Technology. None of the participants had prior interaction with the system; however, all had programming skills and at least a basic understanding of web development.

7.3 Tasks

The overhead of introducing a new programming language was minimized by giving participants a clear context before each test. At the beginning of every session, we briefly explained the purpose of the application and highlighted some of the core features of the language behind it. This framing helped participants understand what to expect and reduced confusion, allowing them to focus on the interaction itself rather than struggling with unfamiliar concepts. Then, we described the activities. The description below is taken directly from the instructions provided to participants.

Activity 1: Building a Simple Poll Application

In this activity, the participant is asked to build a simple poll application using the graphical interface. A step-by-step guide was provided to assist participants in completing the tasks. This activity aims to evaluate the participant's ability to translate domain-specific functionality into graphical operations using the GUI. We measure the time required to complete each step and observe any usability issues or errors.

The sequence of steps is as follows:

1. Create a database **Table** named `poll`, with fields `id`(number), `option`(text), and `votes`(number).
2. Add a **Definition** named `size` that computes the number of entries in the `poll` table.
3. Add an **Insert Action** named `addOption` that adds a voting option to `poll` , with the following fields:
    - `id` set to the value of `size`,

- `option` set to the parameter `inName`,
- `votes` initialized to `0`.

4. Add an **Update Action** named `vote`, which increments an option's **a** votes in `poll`, with the following fields and condition:
   - field `id` set to `a.id`,
   - field `option` set to `a.option`,
   - field `votes` set to `a.votes + 1`,
   - **condition** is set to a.id == inID.
5. Create a **HTML View** node to interact with the application called `index`.
6. Manually connect the `poll` table and the `addOption` action to `index` by dragging the handles on the right side of both structures to the left handle of the HTML View.
7. Execute the `addOption` action inside the HTML by adding a new option, Viktor.

Activity 2: Build a Counter Apllication

In this activity, the participant must utilize what he has learned while performing Activity 1 to build a Counter web application and successfully increment the counter at least once.

7.4 Results

We now present the results collected during the testing sessions. One of our primary goals was to assess the effectiveness and usability of the system by comparing not only the time it took participants to build a small application using the interface versus hand-coding, but also how this performance compares to prior systems designed for similar visual programming tasks.

We originally considered running a comparative evaluation between the visual and textual versions of programming environments that followed the Meerkat language, with participants familiar with the syntax. However, due to the lack of participants with these skills, this would have required training them to a sufficient level of proficiency in the textual language, which would have made the study too long and impractical in the scope of this work. Conducting tests with participants without prior syntax knowledge would have been even more time-consuming, and given the limited availability of the participants in our study, it was not feasible to carry out such an evaluation. However, such a comparison could provide valuable insights in future work, both for measuring productivity differences with experienced users and for assessing the learning curve of newcomers when using visual versus textual environments.

To enable a meaningful comparison, we draw on data from a previous visual prototype [3], where users were asked to construct a simple Task List application using a

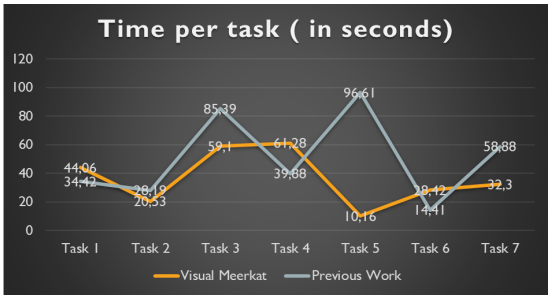
step-by-step guide. While the structure and domain of that task differ slightly from our Activity 1, the required operations are analogous: creating and managing data tables, defining computed values, adding insert/update actions, linking these to an HTML view, and executing interactions. Both activities involve a comparable number of conceptual steps, including modeling data, wiring behaviors, and building a user interface, making their timing results a relevant point of comparison.

Table 7.1 shows the measured times (in seconds) for each task of Activity 1, grouped by participant. As observed, task completion times generally exhibit only slight deviations; however, in some cases, the deviation exceeds the average. This was typically due to participants misreading instructions or experimenting with alternative approaches they found intuitive. Despite this, no significant usability issues were observed during the test.

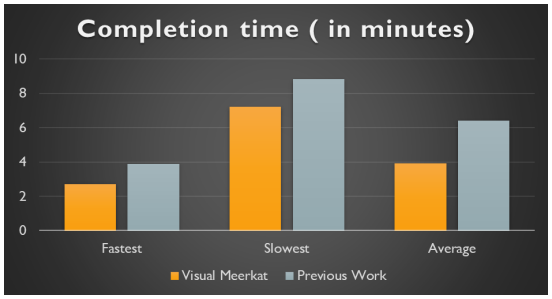
Tasks / Participants	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**
Task 1	39.79	43.11	38.27	48.06	55.82	29.09	32.12	86.00	41.74	26.55
Task 2	24.39	24.05	25.12	17.39	12.41	15.37	17.92	27.67	23.88	17.07
Task 3	81.00	51.37	56.92	45.16	31.72	58.64	61.00	105.00	59.27	40.87
Task 4	49.34	50.28	49.70	66.74	28.17	114.00	49.36	83.00	54.22	68.00
Task 5	10.64	9.40	9.24	9.76	17.35	7.74	10.28	6.16	9.25	11.74
Task 6	29.80	13.87	17.20	8.59	8.67	32.38	57.46	47.99	24.13	44.07
Task 7	15.82	25.82	12.18	20.39	10.68	9.88	41.53	78.00	32.67	76.00
Total (in sec)	250.78	217.90	208.63	215.09	164.82	266.10	269.67	433.82	245.16	284.30
Total (in min)	≈ 4.2	≈ 3.6	≈ 3.5	≈ 3.6	≈ 2.7	≈ 4.4	≈ 4.5	≈ 7.2	≈ 4.1	≈ 4.7

Table 7.1: Measured times for Activity 1

In both evaluations, all participants were able to complete all tasks, indicating that both systems were effective and the testing successful. As shown in Figure 7.1, when comparing the times per task and total completion times to those reported in the prior system [3], we observe a noticeable improvement.



(a) Time per task compared to previous work.



(b) Completion times: slowest, fastest, and average.

Figure 7.1: Comparison of task performance with previous work. (a) shows the time per task, and (b) shows the summary statistics of completion times.

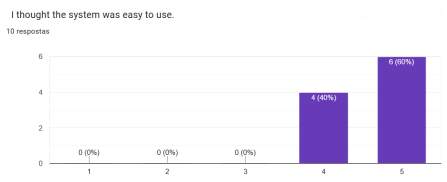
In the previous Task List activity, participants took an average of approximately 6.4 minutes to complete the application (ranging from ~4 to ~9 minutes). In contrast, participants using our system completed the poll activity in an average of 3.9 minutes, with a range from approximately 2.7 to 7.2 minutes. While the tasks are not identical, the structure and complexity are sufficiently comparable to suggest that our system enables

users to build reactive web applications more efficiently. This efficiency gain (nearly 40% reduction in average completion time) demonstrates the potential of our visual interface to streamline development even for users unfamiliar with the underlying model. In terms of qualitative insights, a few participants noted some lack of clarity in certain parts of the task list. Regarding the interface itself, more than half of the participants commented during testing that it felt intuitive and easy to use.

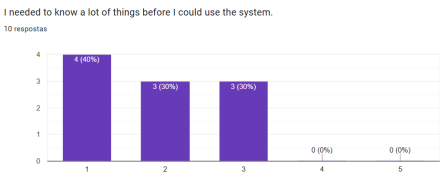
After completing all tasks, participants were asked to respond to a usability questionnaire to provide feedback on the system they had just used. All responses were given using a 5-point Likert scale [31], where 1 stands for *strongly disagree* and 5 stands for *strongly agree*. The questions targeted perceived complexity, usefulness, ease of use, integration, innovation, and overall satisfaction. The results are displayed in Figures 7.2 and 7.3.

In terms of perceived ease of use, all participants rated the system positively. A majority (60%) strongly agreed that the system was easy to use, and the remaining 40% agreed. When asked whether the system was easy to learn, 70% strongly agreed and 30% agreed. Similar results were observed regarding interface organization, with 70% strongly agreeing the elements were well arranged, and 30% agreeing. As for complexity, most participants did not perceive the system as hard to grasp. 80% strongly disagreed with the statement "I found the system unnecessarily complex," indicating they found it simple and straightforward. When asked whether they needed to know a lot before using the system, 40% strongly disagreed, 30% disagreed, and 30% were neutral, suggesting that prior knowledge requirements were considered low to moderate overall.

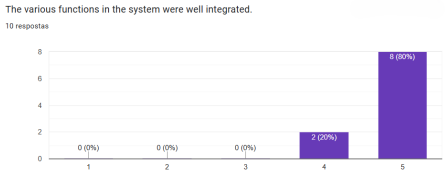
Participants also evaluated the coherence and integration of system features positively. 80% strongly agreed that the various functions in the system were well integrated, while the remaining 20% agreed. Regarding confidence while using the system, 30% strongly agreed and 60% agreed, while only one participant (10%) gave a neutral response. In terms of usefulness, 90% of participants strongly agreed that the system may be useful, and 10% agreed. A similarly strong response was observed on innovation in the context of web development, with 70% strongly agreeing and 30% agreeing. Finally, participants expressed overall satisfaction: 80% strongly agreed they were satisfied with the system, and 20% agreed. The results suggest the system was perceived as easy to learn and use, with low complexity, high usefulness, and strong user satisfaction.



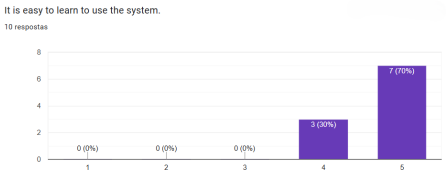
(a) Q1



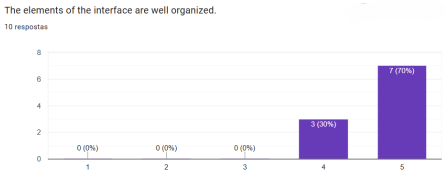
(b) Q2



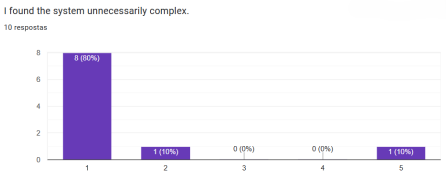
(c) Q3



(d) Q4

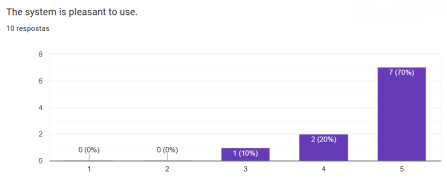


(e) Q5

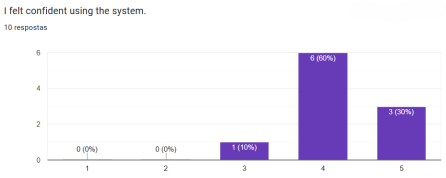


(f) Q6

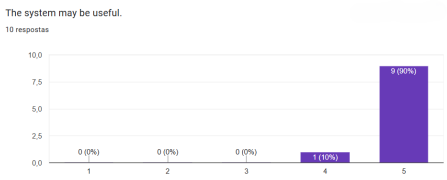
Figure 7.2: Responses to usability questionnaire (SUS), questions Q1–Q6.



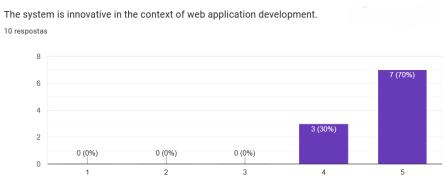
(a) Q7



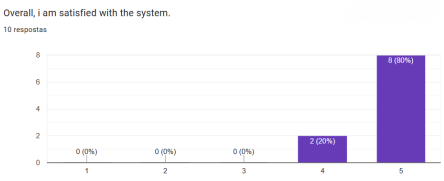
(b) Q8



(c) Q9



(d) Q10



(e) Q11

Figure 7.3: Responses to usability questionnaire (SUS), questions Q7–Q11.

8

Conclusion

In this chapter we present the conclusions of our work, including the limitations we faced, future work, and final remarks.

8.1 Limitations

When we proposed the project, we intended the final prototype to be connected to a distributed environment, allowing the user to interact with the system in real-time and providing immediate feedback while also guaranteeing state consistency and concurrent use by multiple clients. We initially implemented the interface based on a distributed RUST backend, however, we concluded that due to its unfinished state and lack of essential features and language components, we were unable to continue to implement the interface in order to create a fully functional prototype. So, in order to continue the project, we utilized the runtime system that was used for the *Live Programming* prototype discussed in section 3.2 and prior implementations [3], adding new features and making adjustments as they were needed. The current prototype is centralized, meaning that all interactions are processed on a single server, which limits scalability and fault tolerance.

We also proposed a specific visual layout for the interface, that would reorganize nodes in a way that would mimic the various applications layers, such as UI, Data, and Presentation, and apply progressive visual abstraction: when zoomed out, nodes are shown in simplified form, while zooming in reveals increasingly detailed information. While having layouts to ensure the interface is intuitive, we were unable to implement this specific type of layout feature in the current prototype due to limitations in the framework used. Compatibility with mobile screens was achieved, whereas support for virtual reality was proposed but not fully implemented due to time constraints.

8.2 Conclusion

As proposed initially, our environment supports simplifying web application development through a unified visual interface. By tightly coupling a reactive runtime with a GUI,

developers can incrementally build applications using direct manipulation of high-level constructs such as variables, views, and actions without having to write code, all visualized within a DAG to reinforce data flow and dependency clarity.

The user evaluation results were very positive in assessing system usability, which was our main concern. All participants were able to successfully complete the assigned tasks, and the average completion time was lower compared to a previous prototype [3]. Participants praised the system and rated it highly in terms of ease of use, learnability, and overall satisfaction, with most considering the interface intuitive and the prior knowledge requirements low to moderate. Although our evaluation focused on computer science students with prior programming experience, these results make us hopeful that our approach to a more intuitive interface for dynamic web development may support users beyond experienced developers. Future studies will be needed not only to assess its usability among users with little or no technical background but also to directly compare the visual environment with textual programming environments to evaluate differences in productivity, learning curve, and overall efficiency.

Despite certain development limitations, we conclude that the thesis successfully achieved its primary goal. We consider this project a contribution to the exploration of more accessible, visual-first approaches to reactive web development, aiming to bridge the gap between novice-friendly tools and expert-level control.

8.3 Future Work

Despite meeting most of the expected requirements and resulting in a functional prototype, several important features remain for future development. A key priority is completing the integration with a distributed environment, where the client and server operate as a unified system. This would allow the prototype to fully leverage distribution, ensuring stronger state consistency and supporting concurrent use by multiple clients.

A second future development is adding compatibility with virtual reality, enabling more versatile and immersive ways of interacting with the system. Since the interface was designed with device versatility in mind, this extension is essential for providing a more complete user experience.

Another important feature would be to implement an interactive *HTML* editor, allowing users to visually design and modify web pages within the system. Features such as drag-and-drop functionality, style templates, and real-time previews would make it possible for both novice and experienced users to create and customize application views efficiently without writing code manually.

Finally, we consider beneficial the inclusion of a visual layout mechanism, as discussed in Section 8.1, that organizes nodes according to application layers, such as UI, Data, and Presentation, and supports progressive abstraction. At broader zoom levels, the interface would present a simplified overview of the system's architecture, while zooming in would reveal more detailed information about individual components and their connections.

Such a feature would give users a clearer understanding of application structure, reduce visual clutter, and facilitate navigation through complex projects.

Bibliography

[1] C. Abras, D. Maloney-Krichmar, and J. Preece. In: *Encyclopedia of Human-Computer Interaction*. Ed. by W. Bainbridge. Thousand Oaks, CA: Sage Publications, 2004 (cit. on p. 15).

[2] L. Aceto et al. *Runtime Instrumentation for Reactive Components (Extended Version)*. 2024. arXiv: 2406.19904 [cs.SE]. URL: https://arxiv.org/abs/2406.19904 (cit. on p. 12).

[3] G. G. V. R. Alves. *Interactive and Live Program Construction*. Master's thesis. 2019-12. URL: https://run.unl.pt/handle/10362/92287 (cit. on pp. 26–28, 36, 48, 52, 67, 68, 71, 72).

[4] T. Böttger et al. *Open Connect Everywhere: A Glimpse at the Internet Ecosystem through the Lens of the Netflix CDN*. 2018. arXiv: 1606.05519 [cs.NI]. URL: https://arxiv.org/abs/1606.05519 (cit. on p. 1).

[5] N. Bouraqadi and S. Stinckwich. "Bridging the gap between morphic visual programming and smalltalk code". In: *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*. ICDL '07. Lugano, Switzerland: Association for Computing Machinery, 2007, pp. 101–120. ISBN: 9781605580845. DOI: 10.1145/1352678.1352685. URL: https://doi.org/10.1145/1352678.1352685 (cit. on p. 21).

[6] J. Brooke. "SUS: A quick and dirty usability scale". In: *Usability Eval. Ind.* 189 (1995-11). URL: https://www.researchgate.net/publication/319394819_SUS_--_a_quick_and_dirty_usability_scale (cit. on p. 65).

[7] *Design at Work: Cooperative Design of Computer Systems*. CRC Press, 2020-10. ISBN: 9781003063988. DOI: 10.1201/9781003063988. URL: http://dx.doi.org/10.1201/9781003063988 (cit. on p. 15).

[8] A. Dix et al. *Human-Computer Interaction*. 3rd ed. Harlow, England: Pearson Education Limited, 2004. DOI: 10.5555/1203012 (cit. on p. 65).

[9] M. Domingues and J. C. Seco. "Type Safe Evolution of Live Systems". English. In: *Proceedings of the Workshop on Reactive and Event-based Languages & Systems (REBLS'15)*. Pittsburgh, Pennsylvania, United States: ACM, 2015. URL: https://novaresearch.unl.pt/en/publications/type-safe-evolution-of-live-systems (cit. on pp. 1, 6, 29, 30, 32).

[10] J. Edwards, J. Chen, and A. Warth. *Live End-User Programming: A Demo/Manifesto*. Rome, Italy, 2016. URL: https://2016.ecoop.org/details/LIVE-2016/12/Live-end-user-programming-a-demo-manifesto (cit. on pp. 15, 17, 18).

[11] C. Elliott and P. Hudak. "Functional Reactive Animation". In: *International Conference on Functional Programming*. 1997. URL: http://conal.net/papers/icfp97/ (cit. on p. 29).

[12] A. Fischer. "Introducing Circa: A dataflow-based language for live coding". In: *2013 1st International Workshop on Live Programming (LIVE)*. 2013, pp. 5–8. DOI: 10.1109/LIVE.2013.6617339 (cit. on pp. 8–10).

[13] J. Freeman and T. Zhou. "Historiographer: Strongly-Consistent Distributed Reactive Programming with Minimal Locking". In: *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH 2023. Cascais, Portugal: Association for Computing Machinery, 2023, pp. 31–33. ISBN: 9798400703843. DOI: 10.1145/3618305.3623597. URL: https://doi.org/10.1145/3618305.3623597 (cit. on pp. 2, 12, 13).

[14] J. L. Gabbard, D. Hix, and J. E. Swan. "and Evaluation of Virtual Environments". In: *IEEE Comput. Graph. Appl.* 19.6 (1999-11), pp. 51–59. ISSN: 0272-1716. DOI: 10.1109/38.799740. URL: https://doi.org/10.1109/38.799740 (cit. on p. 65).

[15] A. Goldberg. *SMALLTALK-80: the interactive programming environment*. USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN: 0201113724 (cit. on p. 20).

[16] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0201113716 (cit. on p. 20).

[17] D. Golovin. *OutSystems as a Rapid Application Development Platform for Mobile and Web Applications*. Lahti, Finland, 2017. URL: https://www.theseus.fi/handle/10024/132267 (cit. on p. 24).

[18] F. Gürcan and G. Taentzer. "Using Microsoft PowerApps, Mendix and OutSystems in Two Development Scenarios: An Experience Report". In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (2021), pp. 67–72. URL: https://api.semanticscholar.org/CorpusID:245388434 (cit. on p. 24).

[19] C. M. Hancock. *Real-Time Programming and the Big Ideas of Computational Literacy*. 2003. URL: https://dspace.mit.edu/handle/1721.1/61549 (cit. on p. 6).

[20] R. L. Huang, P. J. Guo, and S. Lerner. " UNFOLD: Enabling Live Programming for Debugging GUI Applications ". In: *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024-09, pp. 306–316. DOI: 10.1109/VL/HCC60511.2024.00041. URL: https://doi.ieeecomputersociety.org/10.1109/VL/HCC60511.2024.00041 (cit. on p. 7).

[21] D. Ingalls, T. Kaehler, and J. Maloney. "Back to the future: the story of Squeak, a practical Smalltalk written in itself". In: *SIGPLAN Not.* 32.10 (1997-10), pp. 318–326. ISSN: 0362-1340. DOI: 10.1145/263700.263754 (cit. on p. 20).

[22] International Organization for Standardization. *Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts*. International Organization for Standardization, 2018. URL: https://www.iso.org/standard/63500.html (cit. on pp. 14, 65).

[23] S. Krug. *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability*. Always learning. New Riders, 2014. ISBN: 9780321965516. URL: https://books.google.pt/books?id=qahpAgAAQBAJ (cit. on pp. 14, 65).

[24] D. P. Lanter and R. Essinger. "User-Centered Design". In: URL: https://api.semanticscholar.org/CorpusID:29847471 (cit. on p. 15).

[25] T. D. P. G. Q. Lopes. "Language-Based Data Sharing in Web Applications". Master's thesis. NOVA University of Lisbon, 2017-08. URL: http://hdl.handle.net/10362/24099 (cit. on pp. 31, 32).

[26] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[27] V. N. Lukin, A. L. Dzyubenko, and Y. B. Chechikov. "Approaches to User Interface Development". In: *Programming and Computer Software* 46.5 (2020), pp. 316–323. ISSN: 1608-3261. DOI: 10.1134/S0361768820050059. URL: https://doi.org/10.1134/S0361768820050059 (cit. on p. 14).

[28] J. H. Maloney et al. "The Scratch Programming Language and Environment". In: *ACM Trans. Comput. Educ.* 10 (2010), 16:1–16:15. URL: https://api.semanticscholar.org/CorpusID:9744698 (cit. on p. 21).

[29] J. P. C. Mateus. "Runtime Support System for an Incremental and Reactive Web Programming Language". Master's thesis. MA thesis. NOVA University of Lisbon, 2015-09 (cit. on pp. 32, 43).

[30] J. Nielsen. *Usability engineering*. San Francisco, Calif.: Morgan Kaufmann Publishers, 1994. ISBN: 0125184069 9780125184069. URL: http://www.amazon.de/gp/product/0125184069/ref=oh_details_o02_s00_i00 (cit. on p. 14).

[31] G. Norman. *Likert scales, levels of measurement and the "laws" of statistics*. Vol. 15. 5. Springer, 2010, pp. 625–632. DOI: 10.1007/s10459-010-9222-y. URL: https://doi.org/10.1007/s10459-010-9222-y (cit. on p. 69).

[32] K. Petersen, C. Wohlin, and D. Baca. "The Waterfall Model in Large-Scale Development". In: *Product-Focused Software Process Improvement*. Ed. by F. Bomarius et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 386–400. ISBN: 978-3-642-02152-7 (cit. on pp. 16, 17).

[33] M. Santolucito, W. T. Hallahan, and R. Piskac. "Live Programming By Example". In: CHI EA '19 (2019), pp. 1–4. DOI: 10.1145/3290607.3313266. URL: https://doi.org/10.1145/3290607.3313266 (cit. on p. 1).

[34] P. Sawyer, A. Colebourne, and I. Sommerville. "Object-Oriented Database Systems: a Framework for User Interface Development". In: 1992-01, pp. 25–38. ISBN: 978-3-540-19802-4. DOI: 10.1007/978-1-4471-3423-7_3 (cit. on p. 20).

[35] C. Schuster and C. Flanagan. "Live Programming by Example: Using Direct Manipulation for Live Program Synthesis". In: URL: https://api.semanticscholar.org/CorpusID:11618446 (cit. on p. 22).

[36] J. C. Seco and J. Aldrich. "The Meerkat Vision: Language Support for Live, Scalable, Reactive Web Apps". In: *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2024-10, pp. 54–67. ISBN: 9798400712159. DOI: 10.1145/3689492.3690048. URL: https://dl.acm.org/doi/10.1145/3689492.3690048 (cit. on pp. 1, 29, 30).

[37] S. Thorgeirsson, O. Graf, and Z. Su. "The Hidden Program State Hurts Everyone". In: *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2024-10, pp. 266–274. ISBN: 9798400712159. DOI: 10.1145/3689492.3689813. URL: https://dl.acm.org/doi/10.1145/3689492.3689813 (cit. on p. 18).

[38] S. Thorgeirsson and Z. Su. "Algot: An Educational Programming Language with Human-Intuitive Visual Syntax". In: *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2021, pp. 1–5. DOI: 10.1109/VL/HCC51201.2021.9576166 (cit. on p. 19).

[39] L. Windlinger and D. Tuzcuoğlu. "Usability Theory: Adding a User-Centric Perspective to Workplace Management". In: *A Handbook of Management Theories and Models for Office Environments and Services*. Ed. by C. Riratanaphong et al. Routledge, 2021, pp. 173–183. DOI: 10.1201/9781003128786-15. URL: https://doi.org/10.1201/9781003128786-15 (cit. on pp. 14, 65).

Webography

[40] *Another demo/prototype of Historiographer with support for code updates with coordination on a per-node basis.* [Accessed 02-02-2025]. URL: https://github.com/invpt/hig-proto (cit. on p. 2).

[41] *App Lab | Code.org — code.org*. [Accessed 01-02-2025]. URL: https://code.org/tools/applab (cit. on pp. 22, 23).

[42] N. T. Blog. *How We Build Code at Netflix — netflixtechblog.com*. [Accessed 30-01-2025]. URL: https://netflixtechblog.com/how-we-build-code-at-netflix-c5d9bd727f15#:~:text=Our%20deployment%20strategy%20is%20centered,Amazon%20Machine%20Image%2C%20or%20AMI. (cit. on p. 1).

[43] *Clerk: Moldable Live Programming for Clojure — px23.clerk.vision*. [Accessed 31-01-2025]. URL: https://px23.clerk.vision/ (cit. on pp. 9–11).

[44] *Clojure — clojure.org*. [Accessed 31-01-2025]. URL: https://clojure.org/ (cit. on p. 9).

[45] S. Crighton. *LiveCode in Education — livecode.com*. [Accessed 23-01-2025]. URL: https://livecode.com/education/ (cit. on p. 2).

[46] *dagrejs — npmjs.com*. [Accessed 20-03-2025]. URL: https://www.npmjs.com/package/dagrejs (cit. on p. 55).

[47] DaveBeasley. *Introduction to Power Apps - Training | Microsoft Learn — learn.microsoft.com*. [Accessed 14-01-2025]. URL: https://learn.microsoft.com/en-us/training/modules/get-started-with-powerapps/1-powerapps-introduction?source=recommendations&ns-enrollment-type=learningpath&ns-enrollment-id=learn-bizapps.create-powerapps (cit. on pp. 24, 25).

[48] *Distributed Meerkat*. [Accessed 04-02-2025]. URL: https://github.com/heng-zhong-2003/meerkat_distributed (cit. on p. 2).

[49] *Emotional Design or the Secret of the Fourth Wave*. [Accessed 06-02-2025]. URL: https://uexpert.ru/emotsionalnyj-dizajn-ili-tajna-chetvyortoj-volny/ (cit. on p. 15).

[50] *Etoys Tutorial*. URL: https://etoysillinois.org/files/Etoys%20tutorial%201.pdf (cit. on p. 21).

[51] C. Granger. *Light Table — lighttable.com*. [Accessed 28-01-2025]. URL: http://lighttable.com/ (cit. on pp. 7, 8).

[52] *Historiographer prototype in GO*. [Accessed 04-02-2025]. URL: https://github.com/invpt/historiographer/tree/rewrite (cit. on p. 2).

[53] *Low-code Application Development Platform | Mendix — mendix.com*. [Accessed 14-01-2025]. URL: https://www.mendix.com/ (cit. on pp. 24, 25).

[54] *Node-Based UIs in React - React Flow — reactflow.dev*. URL: https://reactflow.dev/ (cit. on p. 53).

[55] *Notes on User Centered Design Process (UCD) — w3.org*. [Accessed 7-01-2025]. URL: https://www.w3.org/WAI/EO/2003/ucd (cit. on p. 15).

[56] OpenAI. *ChatGPT*. [Accessed 22-01-2025]. 2024. URL: https://openai.com/chatgpt (cit. on p. 5).

[57] Ought. *Elicit: The AI Research Assistant*. [Accessed 22-01-2025]. 2024. URL: https://elicit.com (cit. on p. 5).

[58] *OutSystems: Low-code development: The ultimate low-code guide*. [Accessed 9-01-2025]. URL: https://www.outsystems.com/low-code/ (cit. on p. 23).

[59] *Reactive Development Environment — levjj.github.io*. [Accessed 8-01-2025]. URL: https://levjj.github.io/rde/ (cit. on pp. 22, 23).

[60] *Scratch - Imagine, Program, Share — scratch.mit.edu*. [Accessed 3-01-2025]. URL: scratch.mit.edu (cit. on p. 21).

[61] tapanm-MSFT. *What is Power Apps? - Power Apps — learn.microsoft.com*. [Accessed 14-01-2025]. URL: https://learn.microsoft.com/en-us/power-apps/powerapps-overview#power-apps-for-developers (cit. on p. 24).

[62] *The Fastest Research Platform Ever*. [Accessed 22-01-2025]. URL: https://typeset.io/ (cit. on p. 5).

[63] *toddle.dev - The Visual Web Framework — toddle.dev*. [Accessed 01-02-2025]. URL: https://toddle.dev/ (cit. on pp. 25, 26).

[64] *Usability — digital.gov*. [Accessed 7-01-2025]. URL: https://digital.gov/topics/usability/ (cit. on p. 15).

[65] *User-centered design - Wikipedia — en.wikipedia.org*. [Accessed 7-01-2025]. URL: https://en.wikipedia.org/wiki/User-centered_design (cit. on p. 15).

[66] *What is Spiral Model in Software Engineering? - GeeksforGeeks — geeksforgeeks.org*. [Accessed 24-01-2025]. URL: https://www.geeksforgeeks.org/software-engineering-spiral-model/#example-of-spiral-model (cit. on p. 17).

[67] *What is User Centered Design (UCD)? — interaction-design.org*. [Accessed 07-02-2025]. URL: https://www.interaction-design.org/literature/topics/user-centered-design#4_phases_in_user-centered_design-2 (cit. on p. 16).