Gil Godinho Vieira Rodrigues Alves

Bachelor in Computer Science and Informatics Engineering

Interactive and Live Program Construction

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Teresa Romão, Assistant Professor, NOVA University of Lisbon

Co-adviser: João Ricardo Viegas da Costa Seco, Assistant Professor, NOVA University of Lisbon

Examination Committee

Chairperson: João Baptista da Silva Araújo Junior, Assistant Professor, FCT NOVA
Raporteur: Ana Paula Afonso, Assistant Professor, FC UL
Member: Teresa Romão, Assistant Professor, FCT NOVA

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2019

Interactive and Live Program Construction

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my dear mother and sister, who have always been there for me. To my beloved nephews, who have given me so much joy in the hardest of times. And for you, the light and inspiration that has kept me going.

Abstract

In the highly technological and advanced society we live nowadays, it is essential to explore new development approaches in order to increase the efficiency and flexibility with which software is built. Our work focuses on the design and conception of a live graphical environment to allow for incremental and interactive construction of web applications through visual manipulation interactions.

Our research is introduced in the context of a prototype, *Live Programming*, that provides a style of incremental and agile development of web applications, allowing for efficient updates of code and data. However, the construction of a web application through the existing coding environment is still slow and not as flexible as one would wish. This is due to the fact that its user interface is based on text editors, resulting in a heavy reliance on computer code to build these applications.

The goal of our work consists on the conception of a visual construction model and graphical environment that interacts with the *Live Programming* system, allowing to incrementally develop web applications through the manipulation of visual symbols on the screen. The user does not need to program: instead, our tool automatically generates code according to the user's manipulation of the visual components. The user must then be able to visually define the data model, queries, logical operations and presentation views (for example, *html* pages). We aim, as well, at idealizing and proposing creative and convenient techniques to program visualization and methods to visually organize the structure of a program, in order to help the user comprehending the relationships between elements and their responsibility within the system. This way, developers leverage an agile and interactive approach to efficiently deal with increasingly demanding requirements throughout development.

Keywords: agile, interaction, GUI, visual manipulation, software development, iterative design, web applications

Resumo

Na sociedade tecnologicamente avançada em que vivemos hoje em dia, é essencial que se explorem novas abordagens ao desenvolvimento, de modo a aumentar a eficiência e flexibilidade com que se constrói *software*. O nosso trabalho foca-se no design e concepção de um ambiente gráfico e *live*, que permita uma construção incremental e interactiva de aplicações *web* através de interações de manipulação visual.

A nossa investigação é introduzida no contexto de um protótipo, *Live Programming*, que fornece um estilo incremental e *agile* de desenvolvimento de aplicações *web*, permitindo atualizações eficientes de código e dados. Contudo, a construção de aplicações *web* através do ambiente de programação existente ainda é lento e não tão flexível como se desejaria. Isto deve-se ao facto de a interface do utilizador se basear em editores de texto, resultando numa forte dependência de código para construir estas aplicações.

O objetivo do nosso trabalho consiste na concepção de um modelo de construção visual e ambiente gráfico que interage com o sistema *Live Programming*, permitindo um desenvolvimento incremental de aplicações web através da manipulação de objetos visuais no ecrã. O utilizador não precisa de programar: em vez disso, a nossa ferramenta gera código automaticamente, de acordo com a manipulação dos components visuais por parte do utilizador. O utilizador deve ser, então, capaz de definir, visualmente, o modelo de dados, queries, operações lógicas e vistas de apresentação (como por exemplo, páginas *html*). Temos também o objetivo de idealizar e propor técnicas criativas e convenientes para a visualização de programas e métodos para organizar a estrutura dos mesmos, de modo a ajudar o utilizador a perceber as relações entre elementos e as suas responsabilidades dentro do sistema. Deste modo, os *developers* tiram partido de uma abordagem *agile* e interativa para lidar, de forma eficiente, com o aumento de exigência de requisitos ao longo do processo de desenvolvimento.

Palavras-chave: *agile*, interação, interfaces gráficas, manipulação visual, desenvolvimento de *software*, desenho iterativo, aplicações *web*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we provide a brief and general description of the proposed research, presenting the fundamental key aspects to comprehend the overall purpose of the dissertation. We first specify the context in which our work can be integrated. We then proceed to describe the motivation behind and define the main challenges to overcome. We finish by pointing out the relevant aspects of the followed approach, provide a list with the key contributions of the research and detail the structure of the document.

1.1 Context

This research is one of the contributions of the project CLAY (ref. PTDC/EEICTP/4293/2014), an initiative from the NOVA LINCS (NOVA Laboratory for Computer Science and Informatics) research unit, hosted in Departamento de Informática on Faculdade de Ciências e Tecnologias da Universidade de Lisboa (FCT-UNL). CLAY consists of a family of prototypes that provide agile mechanisms to incrementally and safely evolve software systems, aiming to cope with increasingly demanding requirements throughout development and continuous growth of customer's standards.

The project has its origin on a prototype called *LiveWeb* [28], which allows for the easy express interaction between components as well as the the definition of basic safety properties.

Our work and research focuses on a prototype called *Live Programming* [2], which combines a reactive programming model and a live coding environment to build and maintain software systems [14]. This environment provides a style of incremental and agile development of web applications that allows for efficient updates of code and data. The updates are subject of static verification [26], as a means of ensuring a safe evolution of software. The underlying programming model follows a reactive data-flow paradigm,

that captures data dependencies between programming elements. The effects of updates propagate to the relevant pieces of data, ensuring these are constantly kept up-to-date [27]. Unlike traditional programming environments, where code modifications are only observable when the program executes [34], this live coding setting allows for constant and immediate feedback on the effects of updates [32]. Immediate feedback is translated into higher efficiency throughout development, which leads to an improvement in the developer's productivity [32]. Moreover, updates occur without service disruption [26], which means users can continuously interact with the system, even when updates take place. It is important to mention that this environment provides feedback on the programmer's actions through a REPL (or *read-evaluate-print loop*) [33] mechanism, that evaluates and displays the results of code statements as the developer writes them.

Our work aims to take one step further: designing and building a visual construction model and graphical environment that interacts with the *Live Programming* system, allowing to incrementally develop web applications through the manipulation of visual symbols on the screen. The final prototype must support an agile construction of applications, allowing to express interactions and dependencies between components through simple graphical manipulation, as it happens, for instance, with *Low-Code* solutions [22].

1.2 Motivation

In a highly technologically developed society like the one we live nowadays, it is crucial to consider and explore new development approaches, in order to increase the efficiency and ease of software development. Throughout the years, many software development methods appeared and evolved as a means to optimize the process of building and deploying quality software in the business world. In a typical corporative environment, there is a need for well-defined strategies and methodical approaches to software development, in order to achieve a set of stipulated goals respecting both time and financial restrictions.

A very popular traditional software development method, introduced very early on, is the *Waterfall* model [40]. This model consists of a structured progression of well-defined activities throughout the development process. The main idea behind is to define, at the beginning of the project, a set of requirements the system has to comply with and step through each one of the activities in order to fulfil these requirements. Although it brings several advantages, due to the corporative world's inner nature of continuous evolution, traditional methods such as this one present a major drawback: its inability to adapt to change [40]. This is a disadvantage since it is fairly common for previously defined requirements to suffer changes and adjustments during the development process. Therefore, it is essential to efficiently adapt to new demands.

The need for an adaptive approach [40] (instead of a predictive one) is what motivates the adoption of agile methodologies in software development. The idea is to promote an incremental development that includes frequent deliveries of functional software pieces,

while efficiently deal with non-anticipated code modifications [40]. We aim at contributing to a paradigm shift from the old *code-compile-deploy* cycle [26] into an incremental and interactive construction of software, where the reactive properties of programs and the constant feedback from the live environment promotes faster releases of products.

As mentioned earlier, the *Live Programming* system provides an agile and live coding approach to software development. However, the construction of an application through its coding environment is still slow and not as flexible as one would wish. This is due to the fact that the user interface is based on text editors, resulting on a heavy reliance on computer code to build web applications. Our work aims to idealize and design visual and interactive construction methods and combine them with the current live and agile environment of *Live Programming*, in order to provide the user with a more effortless and efficient way of building web applications. Since it is not necessary to write any code, a tool such as this one may, as well, influence users with no programming background and software developing skills to build their own products from scratch, thus contributing to an increased investment opportunity in the software market.

1.3 Problem

With our work, we intend to design and build an interaction model and graphical environment to support the construction and maintenance of web applications through simple data and code manipulation. The intended final product consists of a user interface that interacts with the *Live Programming* system, through which the user incrementally builds applications by relating visual symbols. The user does not need to program: our tool automatically generates code according to the user's manipulation of the visual components. The user must then be able to visually define the data model, queries, logical operations and presentation views (for example, *html* pages).

At any given time, such an environment provides a visual representation of the application being built (and not a sequence of lines of code as before). We aim, as well, at idealizing and proposing creative and convenient techniques to program visualization and methods to reorganize the structure of a program, in order to help the user to better understand the relationships between elements and their responsibility within the system.

Because our user interface is integrated within a live environment [32], the effects of actions (such as adding a new component or executing a function) are immediately visible to the user. With our graphical environment, these effects result in a rearrangement of components in the screen, in such a way users clearly perceive their impact on the overall structure. Moreover, each increment (or update) throughout development leaves the system in a safe state [32] (for example, ensuring syntax errors do not occur). The current *Live Programming* contains an *Interpreter* that verifies code updates, ensuring safety invariants are not broken.

The development process occurs in fast and simple modifications (or increments), for instance, ensuring the user has full visibility [37] of the system's current state and defining clear relations between actions and their effects on the system's state. Moreover, when managing data, it is necessary to ensure that the effects of the updates immediately propagate to all the relevant pieces of data in the application [27], just like in the original model of *Live Programming*.

It is crucial to define the target users of our system, since it is an important aspect to be taken into account throughout the design and development process. When considering the general purpose and high-level goals of our system, there are three groups of target users to consider:

- Type A. These users have, at least, basic programming knowledge and have built web applications in the past. These users aim at optimizing their productivity throughout development and aspire to efficiently deliver updates.
- Type B. These users have no programming knowledge and do not know how to build software. They are focused on the final result and do not wish to worry about construction details.
- Type C. Users of this type have no programming knowledge and do not know how to build software. They aim to acquire basic skills in software development and understand the mechanisms involved in the conception of applications.

Since our approach does not require programming knowledge, users of type B and C should be considered as well. However, we center our design process around users of type A, and our prototype is intended to be used, primarily, by developers who aspire to increase their productivity when building products.

The intended prototype offers the developer a pleasurable [36] user experience [9] as he, effortlessly, defines visual components and interactions amongst them, leveraging of an agile methodology to build fully functional web applications.

1.4 Approach

As previously mentioned, our work contributes to the conception of a new interaction model. It focuses on idealizing interaction strategies for an interactive and visual approach to software development. The work process is divided into two main parts: a more practical one, that consists of the actual implementation of a prototype, and a more theoretical one, that consists on idealizing and sketching techniques to program visualization.

The computer prototype includes two main components: a graphical environment, through which the user interacts with the system, and a visual language, that the user uses inside of the environment. The visual language allows for the clear and easy expression

of relationships amongst programming elements to define the responsibility of each one within the system. In this language, a system is visually represented by a directed graph, where nodes map to programming elements and arcs map to dependencies between them. The structure of a web application is evolved by incrementally defining new programming elements and relationships between these in the diagram. The environment allows for the evolution of an application's state by means of actions. When a code or data update occurs, the environment is responsible for rearranging and displaying the nodes on the screen, in such a way that a user clearly understands the effects on the behaviour or internal state of the system.

The initial step consists of exploring related software, described in chapter 4, and analysing their interaction models to gather useful insights for when designing ours. The design process is based on a *User-centered approach* and the *usability* attributes must be analysed and selected according to the target population of the system. It is essential to define the set of high-level tasks the users are able to do, which is useful to produce initial paper sketches of the graphical environment.

Since a crucial part of the design consists of defining a visual language, it is important to analyse principles used when structuring a *visual notation* [35]. The initial step is defining the *visual vocabulary* of the language, that corresponds to the set of visual symbols the language works with. The following step consists of defining the *visual grammar* of the language, that specifies what relationships are allowed by the language.

Finally, each step is integrated within an iterative design process, in which interaction ideas are document, sketched and evaluated throughout design.

1.5 Contributions

The key contributions of this dissertation can be summarized as follows:

- the conception of a visual construction model that allows to incrementally build web applications through direct manipulation of symbols on the screen;
- the design and implementation of a graphical user interface that enables the interaction with the *Live Programming* system;
- the creation of new interaction techniques to visualize an application's construction process and state;
- a categorization and study of software related products available in the market, according to important identified properties in the context of the work.

1.6 Document Structure

Our document consists of nine chapters: chapter 1 provides a general overview of the proposed system and research, where we briefly discuss the main motivation behind

the work and the approach to be adopted; the two following chapters, 2 and 3, consist, respectively of important concepts regarding *Live Programming* and human-computer interaction foundations to better understand our design process; in chapter 4, we explore a list of related software tools and we identify important features of each one; chapter 5 describes some of the design methodologies used and provides some initial sketches of the design; in chapter 6, we provide a description on each component that is part of our tool; chapter 7 consists of a set of case studies to exemplify the process of software construction with our tool; in chapter 8, we describe the methodology used for evaluating our computer prototype and we present the results gathered during evaluation; finally, we present our final remarks and future work in chapter 9.

Chapter 2

Live Programming

Our work focuses on the CLAY project research leading to the design and development of a prototype called *Live Programming* [2]. This chapter aims to identify and describe the relevant components of this prototype. The initial section contains a general overview of the system. The following section is centred around the prototype's programming model. The last section provides a summary of the whole development environment and describes the architecture of the runtime system that supports the construction and execution of applications.

2.1 Overview

The research developed in the context of the project CLAY, aims to open new perspectives regarding the way that web application are developed nowadays, promoting the adoption of new agile development tools. It aims to take a step forward in traditional software development methods towards an innovative approach to deal with increasingly demanding requirements during development [14].

Code updates are a natural part of a software system's life cycle [14] and fairly common throughout the process. Those updates can frequently lead to a decrease in the efficiency of software development, as the structured progression of software life cycle activities introduces major gaps between code refactoring and the perception of modifications on the overall state of the system (those effects are only observable while the program executes or after its deployment). Firstly, this means that it is possible for individuals to use the system in the presence of errors, which leads to a bad user experience and, consequently, to a decline in the number of active users. Secondly, code updates cause disruption of service, which means that clients cannot use the system when modifications take place.

CLAY introduces a new prototype called *Live Programming* [2]. This prototype provides a novel core programming model [26] that makes use of an agile methodology to support incremental development of software applications. It tackles the previously mentioned issues combining a programming model based on a reactive dataflow paradigm and a live programming environment. The properties of a dataflow language combined with the live coding setting allow the developer to obtain immediate feedback on the effects of gradually introducing new programming elements in the system or redefining existing ones [26]. Moreover, it relies on a type system [26] that ensures static verification of operations, which, in turn, ensures the absence of runtime errors at each increment of the development process. The combination of those components allows for incremental construction and maintenance of software systems, capturing a verified and flexible style of agile programming [14].

The following sections detail each one of the high-level components behind this prototype's approach: the programming model and the development environment.

2.2 Programming Model

This section focuses on an essential high-level part of the *Live Programming* prototype: its programming model. We present a general overview of the concept behind it (the theoretical aspects and paradigm) and a description of each one of the semantical elements of the implemented programming language.

The programming model is supported by a dataflow reactive programming paradigm [16]. A reactive paradigm is oriented to the propagation of effects throughout data, caused by the update of a component of an application or program. In practice, several data dependencies amongst programming elements can be identified throughout a program. In this paradigm, the effects of a given modification are propagated throughout those elements and their dependent ones. Let us consider the expression $a = b + c$. Considering a typical imperative paradigm, at the moment the expression executes, the value of $b + c$ is assigned to a and the modification of both b or c variables later in the execution produce no modification in the value of a. However, in a reactive language, an implicit dependency is created between those elements and the updates issued on b or c are propagated into a; thus, modifying its value.

The previous description brings us to the conceptual notion of a dataflow [31] paradigm: a program is represented through a directed graph, where each node is an expression of the program's code such as an arithmetic or comparison operation and the arcs are dependencies between expressions. This provides us with a visual notation of flow dependencies throughout the program, which allows us to reason on the propagation of effects triggered by a given action. In the previous example, b and c would both have an arrow pointing towards the node with the value a, indicating a dependency between those elements and the direction in which the propagation of effects occur. One advantage of this model is that it supports parallelism in the execution [31] since the effects of

```
var counter = 0
def inc = action { counter := counter + 1 }
def reset = action { counter := 0 }
def counterPage =
    <div>
        <p>"Counter: " counter</p>
        <button doaction=(inc)>
            "Increment"
        </button>
        <button doaction=(reset)>
            "Reset"
        </button>
    </div>
```

Figure 2.1: Implementation of a simple counter.

modifications may propagate into different expressions at the same time, contributing to higher efficiency.

The dataflow reactive paradigm is the base on which the prototype's programming language operates. The language consists of three main elements [27]:

- state variables. Models the persistent layer of the application;
- pure data transformation expressions. Models the logic that queries data and presents web pages to users;
- actions. Imperative updates to the persistent layer of the application. In more practical terms, it consists of updates on the state variables.

Pure data transformation expressions may be dependent on other expressions or state variables. When an action is triggered, modifying a state variable (changing the persistent layer of the system), those effects are propagated throughout the flow-graph, updating the relevant pieces of data (expressions that are dependent on the variable in question). Moreover, the language allows a pure data transformation expression to model a web page. Essentially, this means that a web page can be seen as a logical expression that depends on particular persistent data (or similarly, the current state of its dynamic content depends on the value of state variables). When an action is triggered, updating the value of a state variable, the dynamic content of the web page that depends on that variable is updated almost immediately, ensuring that all named elements are kept up-to-date with respect to the application persistent state [27].

To provide an example of a simple application that one can implement with this language, let us consider a simple counter, whose value is possible to increase and reset. The application also provides an *html* view that displays the value of the counter to the user and allows him to change its state. The code fragment of figure 2.1 implements the example above.

The variable *counter* is defined with the keyword *var* (line 1), that specifies a state variable. Lines 2 and 3 define, respectively, an action that increments *counter* and an action that sets its value to zero. The following lines (lines 5 to 14) define a view that displays the value of *counter* and two buttons to execute each action.

The language also provides a style of typeful programming; it means that each increment throughout the development process is statically verified by the language's type system [32], ensuring the absence of type errors. Moreover, the programming system guarantees that the application evolves in a safe way by maintaining the same set of properties throughout the iterations of development, ensuring that established security and safety invariants are not broken at each update and that the system does not evolve to invalid states. The system also implements a scheduling discipline that ensures the absence of runtime errors and interference between execution and development [26], enabling constant feedback at the level of program results and invariant verification [26].

The whole execution model and the reactive properties inherent to it contribute to continuous and immediate feedback from the platform on the effects of modifications; therefore, being an essential element to understand in the context of this work.

2.3 Development Environment

The development environment consists of a live coding setting designed to reduce the feedback loop between code modifications and the perception of its effects [26]. The idea behind it is to enrich the task of programming by interactively providing a response at each of the developer's actions (such as writing statements as in textual editors or manipulation of visual elements). In more practical terms, these environments offer features such as, for instance, the evaluation of code statements as they are written (verifying their correctness or testing for potential violation of imposed restrictions, for example) or visual feedback on the modifications of a web page as its source code changes [34]. Moreover, this type of setting also allows for a program to execute while under modification, which means that, at a large scale, an application keeps providing its service to clients even when under updates.

In the specific context of the *Live Programming* system, the platform allows an incremental construction of applications. An increment may consist, for instance, in the modification of a given feature, the addition of new functionalities or the actualizations of the data schema of the application. The use of this type of setting allows for the effects of those modifications to be immediately visible to the developer, which promotes a much more efficient and flexible construction. As mentioned earlier, each actualization throughout the development process is statically verified, thus, ensuring the soundness of the system at all times [26]. To provide all of those desirable properties, the platform relies on a runtime system [32] to support the execution of applications.

The platform combines a runtime system (that executes on a remote web server) with an integrated development environment accessible through the client's browser. The IDE

allows programming the behaviour and the interface of a web application [32] using the earlier mentioned language semantics. The runtime system evaluates statements as they are written, executes them and updates the application's code and data. It also keeps a subscription list [32] containing information on the system's clients and on the web pages currently accessed by each client. Figure 2.2 illustrates the architecture of the system.

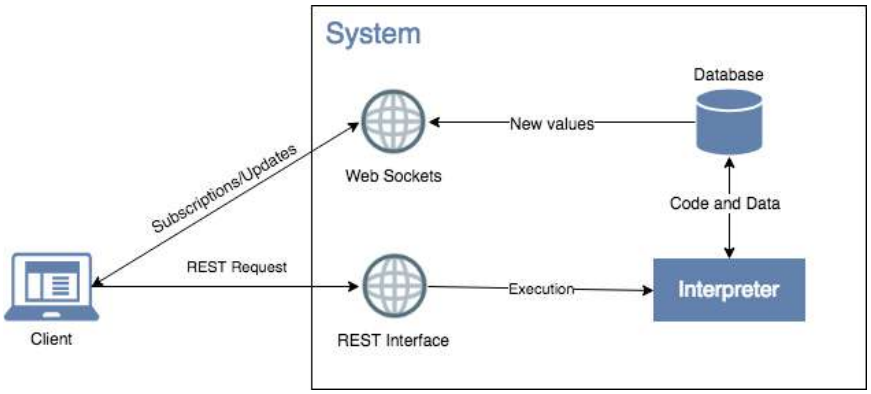


Figure 2.2: System's Architecture. Adapted from [32].

An application evolves by gradually introducing new programming elements and redefining existing ones. In order for a client to receive the most recent update on a given programming element, he initially subscribes that element. Every time the value or expression of an element changes, the clients currently subscribing it are notified with the most recent update via web sockets (the pages depending on that particular element are recalculated and displayed) [32]. The system provides a REST interface to handle requests initialized by the client. GET requests are used to obtain a value associated with a given element, such as a web page or a function. POST requests are used to add new code to the application (the modifications of the application are sent to the web server to be evaluated and executed). Finally, PUT requests are used to execute actions. Actions are operations that modify the state of the application, typically associated with events such as button clicks [32]. The central unit of the system is the *Interpreter*. Every time a code actualization takes place, the modifications are automatically sent to the web server. The *Interpreter* executes the written code relying on the type system of the language to statically verify it, ensuring a safe evolution of the application throughout development. The values of the names that depend on the modified element are sequentially recalculated until the effects caused by the update have propagated throughout the whole system. The *Database* component stores the code and the persistent data of the application. In more detail, it stores the structure of the current source code, the declared names and their corresponding list of dependent elements (as a means to propagate effects throughout the graph). Additionally, it stores, as well, the persistent data of the application.

The interaction amongst the described components combined with the reactive properties of the programming language allows for the flexible style of incremental and live programming [26] of software systems we aim to achieve.

Chapter 3

Design Concepts

In this chapter, we point out different concepts related to the design of graphical interfaces and provide a description of each. Those concepts are useful understanding as a means to comprehend the adopted approach. The first section provides a formal definition of the designer's role and details each entity that takes a part in an interaction process. In the subsequent section, we explain the criteria used to measure the quality of an interface; then, we present the techniques used to evolve the interface throughout the design process. In the final section, we describe different techniques of design evaluation.

3.1 Interaction and Conceptual Model

Computational systems and humans are both highly complex and quite distinct from each other; they function and communicate in quite dissimilar ways. Interaction consists of a translation process between the human user and a computational system, such that they are actually able to exchange information among themselves. Aiming towards an optimal translation between both entities will potentially lead to a much smoother and efficient execution of user tasks.

Understanding the concept of interaction and its relevance in this research allows us now to examine its related theoretical foundations in order to project an appropriate design. Good design is an act of communication between the designer and the user. Several high-level entities are involved in the interaction process: system, user, designer and interface. Each one has its own conceptual model of the system. Figure 3.1 shows an adaptation of Norman's framework, which depicts the relationships between these entities and their corresponding conceptual models.

The *System Model* expresses how the computational system actually works. In more detail, it represents how the system's components interact amongst each other to provide

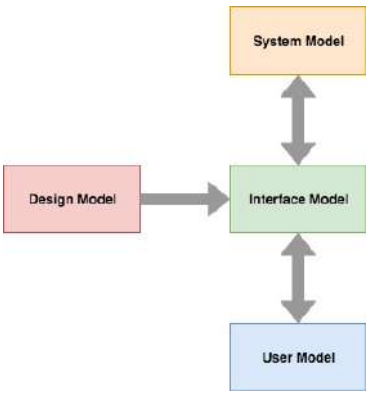


Figure 3.1: Norman's framework. Adapted from [37].

the required functionality. The *Interface Model* expresses how the system is presented to the exterior world (the interaction with the system). The *User Model* expresses how the user believes the system works, and the *Design Model* expresses how the designer expects users to perceive the system's interface. Resorting to the descriptions of these models, we now formally define the role of the designer: given a certain *System Model*, the designer has to choose a *Conceptual Model* in such a way that the relevant system's functional interactions are clearly transmitted to the user. In other words, he chooses a *Conceptual Model* which must be well communicated to the user through the *Interface Model*, supporting the creation of a correct *User Model*. In the sketching and prototyping [24] phases, the decisions regarding the design will influence how well is this *Conceptual Model* communicated to the user.

3.2 Usability and Design Principles

Before proceeding to the actual design process, one must consider distinct quality requirements, analyse the trade-offs amongst them and decide which ones should be assigned a higher level of importance. Defining those quality criteria for the interaction between the user and the platform may be a challenging task since quality is a subjective concept and consequently those requirements become difficult to measure with precision, as opposed to what happens with functional requirements. Usability [36] is an important concept to comprehend within the context of this work since it allows to actually measure desirable quality attributes instead of simply basing the evaluation of the interaction on the designer's subjective perceptions.

As stated in the ISO 9241-11 [9], the term usability can be defined as an "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". Usability can, therefore, be viewed as a multi-dimensional interface property that determines how well users can use the system's provided functionality. According to Nielsen [36], the usability of a system is measured according to five attributes:

- Learnability. This usability attribute measures the ease on which novice users (individuals who have never used the system before) learn to effectively interact with the interface.
- Efficiency. It is related to a second stage of the learning curve in which the user is considered an expert while interacting with the system. Efficiency refers to the maximum level of productivity a user can achieve after learning how the system actually works.
- Memorability. Casual users are users that use the system intermittently and not as frequently as experts, and, unlike novices, they have used the system in the past so they do not need to learn how it works from scratch; they only need to remember how to use it based on what they have previously learned when interacting with it [36]. If a user does not use an application for a large amount of time, then it is essential to remember how it works when using it again.
- Errors. An error can be defined as an action that does not accomplish a given goal. During the interaction, it is desirable that the user makes as few errors as possible. Therefore, it is important to ensure that the interface is not error-prone.
- Satisfaction. It measures the pleasure felt by a given user when interacting with the platform.

Another important usability attribute that is not considered by Nielsen, is the *effectiveness* [9] of the interface, that is defined as the accuracy and completeness with which users achieve specified goals.

Throughout the design process, it is essential to follow certain principles as a means to achieve predefined usability goals and assure an appropriate interface model. The following list contains each one of Donald Norman's design principles [37]:

- Affordances. Properties of the interface that suggest and determine how to interact with it.
- Mappings. Relationships between controls and their respective effects on the system. Those relationships should be clear to the user.
- Visibility. The relevant parts of a system's state and the available operations should be clearly visible to the user.
- Feedback. Each action the user executes should result in continuous and immediate feedback.
- Constraints. Only available actions should be visible. The design should limit the number of possible actions to the user.

The prioritization of usability attributes is made based on the information collected during the stages of the iterative design process, described in the following section.

3.3 User-Centered Approach

This section provides a detailed description of the followed design methodology, examining each phase of the processes' structure and identifying the important aspects of each one. Furthermore, we additionally reference the philosophy behind a User-Centered approach [25] and identify the evaluation categories to be applied later on.

3.3.1 Iterative Design

Software development methodologies are methodical approaches used as a means to structure and plan software design and implementation. As previously mentioned, one of the most popular software development methods is the *Waterfall* model [39]. This model consists of a structured sequence of steps, each one with a particular purpose in the development process: requirements specification, design, implementation, testing and integration and maintenance. Note that the terminology used in the model may be different depending on the author. As a means to understand the reason why this model is not suitable to achieve the specified goals of our work, let us consider the following: the only two stages in which the user's input is actually considered is during the phases of requirements specification and testing and integration. Moreover, the late detection of errors causes expensive rectifications, which may imply redefining the initially specified requirements and applying each phase of the process again. The first problem is that the process repeats itself every time an error is detected, translating to a high lack of flexibility of the development process. The second problem is that each iteration corresponds to a deployed version of the system, which means users can actually interact with the system in the presence of errors.

As the requirements of the system can not be completely specified in the initial activity, it is necessary a methodology that allows to iteratively evolve the design of the interface based on the user's continuous feedback. Those are the foundations for an iterative design process [25]. This process is summarized in figure 3.2.

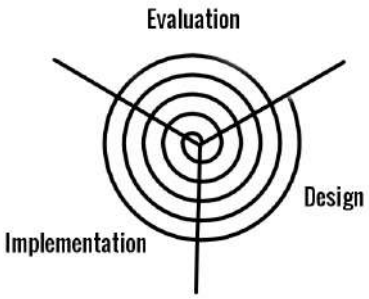


Figure 3.2: Iterative Design.

An iteration consists of a sequence of steps: design, implementation and evaluation. Each step attempts to tackle the problems identified in previous iterations by observing the interaction between users and the prototype under test and by receiving their insights on the opted design decisions. The iterative design is a fundamental element of a *User-Centered* design, in which the development process is based on the user's needs, abilities, social context and work environment and should make use of human skills and judgment [38] to create a well-designed system.

3.3.2 User and Task Analysis

Before the actual design process, a *User-Centered* approach consists of two essential information gathering activities: the *User* and *Task Analysis* [25].

The *User Analysis* activity consists of clearly identifying the target users of the system. Initially, one must consider all the possible types of target users the system is designed for, identifying the different user classes. It is possible to employ several techniques to collect information on potential target users, such as interviews or inquiries.

The *Task Analysis* activity determines the user's goals: identifies the tasks they perform on a daily basis and examines the reasons they do it, the knowledge they must have to execute those tasks and the tools they resort to [25]. The following list contains examples of different collecting information techniques:

- Direct observation. Consists of observing users performing tasks in, for instance, their work environment and encourage them to think aloud while doing it. Task-related questions may be asked to users in order to decompose them into smaller sub-tasks, to be later hierarchically structured

- Users interview. May consist of a structured interview, in which a plan is followed (formulation of specific questions to efficiently tackle all the identified problems), a non-structured interview (open talk between the interviewer and interviewee) or a semi-structured interview (starts with a plan of questions and ends up in an open talk)

- Participatory design. Designers include representative users of each user class in the design team in order for them to test the interface and offer their insights during the the design process

At this point, it is fundamental to consider existing related tools, in order to gather ideas concerning functionality and distinct interaction styles. The gathered information is then structured resorting to notations or diagrams as a means to describe the identified tasks, decompose them hierarchically and identify their goals (what is the task intended to achieve), preconditions (which state must the system be in before executing the task) and exceptions (what fails when the action is executed).

3.3.3 Sketching and Prototyping

The design process begins as the previously described activities finish. In the initial iterations of the process, designers are mostly focused on having several distinct ideas, explore each one of them individually, discuss their advantages and disadvantages and finally creating a concept for the interface. In this initial phase, it is appropriate to use a *Sketching* [24] technique as sketches are cheap and quick to create, thus, allowing to explore several ideas simultaneously. Furthermore, a sketch is encouraged to have minimal detail and be ambiguous enough so that it can be interpreted in distinct ways by users, thus promoting the exchange of insights within the team. Sketches change during the evaluation as a means to fully capture the essence of user's suggestions.

In subsequent iterations and after the main concept of the interface has been defined, the designer must focus on the global design. At this point, it is convenient to use *Prototyping* [25] techniques such as paper prototypes or other early prototyping tools such as *Mockingbird* [20] or *Justinmind* [4]. Paper prototypes are fast to build (which consequently leads to quicker feedback from the users) and simple to change. Usually, they consist of interactive paper mock-ups, in which the interface elements are represented with pieces of paper and its dynamic behaviour is simulated by the designer during tests (the interaction dynamics may be difficult to capture). During tests, the user is observed interacting with the prototype and the results must be registered and taken into account when redesigning the next iteration.

In later stages of the process, once defined the concept and the global design of the interface, the modifications and corrections become less general and aim to tackle specific usability problems or uncover errors during evaluation. The paper prototype can then be converted into a computational prototype. These prototypes capture the interaction dynamics and the intended appearance more closely than the previously mentioned methods, thus, the evaluation results must also determine if the overall disposition of controls and the use of fonts, colours and icons are appropriate. Moreover, at this point, the prototype should already have a high enough level of refinement so that it is comparable to the intended final product (for instance, having all the required functionalities implemented until a certain degree).

3.3.4 Evaluation

The evaluation of design aims to examine and test the system's functionalities, the impact the interface has on the user (if it satisfies the essential usability attributes) and identify specific problems related with the design [25]. The evaluation is classified into two main categories: *evaluation through expert analysis* and *evaluation through user participation* [25].

In *Evaluation through expert analysis* techniques, experts (such as designers) test the prototype in order to identify design issues that may cause difficulties throughout the interaction with the system, due to, for instance, the violation of cognitive principles. Our research focuses on two techniques of evaluation through expert analysis:

- Cognitive Walkthrough. Evaluators execute a sequence of actions in order to achieve a particular goal while searching for usability problems. For each action to be executed, the evaluator verifies if the action is actually available, if it is easily identified given its controller and if appropriate feedback is provided to the user
- Heuristic Evaluation. The interface is thoroughly inspected in order to detect potential usability problems. These issues are detected based on a well-known set of usability guidelines named *Usability Heuristics*. Each usability problem must then be assigned with a severity factor, the heuristic it violates and a possible solution for it. The list of detected problems is included in a report named *Heuristic Evaluation Report*, which is delivered to designers in order to attempt solving the identified issues in the next iteration

In an *evaluation through user participation*, the prototype is tested by the system's future users in order to detect potential usability problems. Our work focuses on a particular technique called *Formative Evaluation*, in which representative users from each *User Class* are selected and assigned a set of tasks to execute. An individual from the design team, called *Facilitator* explains the overall purpose of the system and assigns tasks to the users, and other members, called *Observers*, observe the interaction and collect notes in order for the user's behaviour to be recreated later.

To conclude, users may still be asked to fulfil a usability questionnaire, such as System Usability Scale (SUS) [18], in order to gather their opinions on the interaction.

3.4 Visual Notation

In the previous sections, we have examined general Human-Computer interaction concepts that are required to justify given decisions when designing a user interface. In this section, we explore a set of principles for designing cognitively effective notations for a visual language [35]. The purpose for this set of principles is to lead the design through a self-conscious process, in which the designer is aware the decisions made throughout design and development may positively or negatively influence the user's problem-solving capabilities when interacting with the platform. Furthermore, because the interaction relies on a *visual notation* to represent a system and its construction process, it is crucial to analyse and understand concepts behind the definition of these languages in Software Engineering.

Firstly, it is important to define the term *visual notation* (or *diagramming notation*): A *visual notation* consists of a set of symbols (the *visual vocabulary*), a set of rules that determine how these symbols communicate amongst each other (the *visual grammar*) and a meaning for each symbol and relationships between them (the *visual semantics*). The information carried by a visual symbol can be encoded by eight variables: *shape*, *brightness*, *size*, *orientation*, *colour*, *texture*, *horizontal* and *vertical positions*. The *visual vocabulary* and the *visual grammar* form the *visual syntax* of the language. A diagram consists of a set

of instances of symbols arranged according to the rules of the grammar. A symbol have associated meaning, that should to be quickly and easily perceived by users. This brings us to another important concept: the *cognitive effectiveness*. *Cognitive effectiveness* can be defined as the speed, ease and accuracy with which a representation can be processed by the human mind. Similarly to how usability attributes are used to evaluate how good the usability of the interaction is, the principles mentioned in this section are used to design an appropriate visual notation and measure how good its cognitive effectiveness is. The following list provides a brief description of each principle:

- Semiotic Clarity: A visual symbol should only have one semantic construct associated, and a semantic construct must be only associated with a visual symbol.
- Perceptual Discriminability: It measures the accuracy and precision with which one can distinguish symbols.
- Semantic Transparency: The appearance of a visual representation should provide cues for their actual semantics (for example, the use of icons instead of abstract shapes).
- Complexity Management: The notation should provide mechanisms to deal with the continuous increase of complexity.
- Cognitive Integration: The notation should provide mechanisms to support the integration of information from different diagrams (when multiple diagrams are used to represent a system).
- Visual Expressiveness: The visual symbols of the language should make use of the full domain of values on each visual variable used.
- Dual Coding: The notation should rely on textual descriptions as a means of providing additional information to the graphical elements.
- Graphic Economy: The number of graphical symbols should be cognitively manageable (to promote a simple language syntax).
- Cognitive Fit: The notation should provide different visual representations of information for different target users (the target audiences may be very different from each other).

The described principles are mentioned in later sections of this document. These principles are useful understanding as a means of justifying our decisions throughout the design and development process.

Chapter 4

Related Work

In this chapter, we introduce some tools related to the context of this thesis. Part of the referenced software incorporates reactive properties that are relevant understanding inside the scope of this work. We provide a categorization of these tools according to the particular properties they present. Throughout the chapter, we make reference to visual programming languages, interactive platforms and other types of tools we find crucial to be mentioned, such as, for instance, *Low-Code* [22] technologies.

4.1 Visual Manipulation

When discussing the evolution of graphical user interfaces throughout history, it is crucial to reference *Smalltalk* [30] development environment due to the major impact it had on the subject. *Smalltalk* became public in 1980 and consists of a dynamically typed object-oriented programming language and of an integrated set of tools to visually interact with the language features [30]. An important aspect to mention regarding the programming language is its dynamic properties: it allows for a fast and flexible development by reducing verbosity in the source code (supporting high-level constructs such as objects, that reduce the amount of code needed). Moreover, the *Smalltalk* environment supports *hot swapping*, which means it is possible to update the code of a program without the need to restart it (it preserves the state of the computation [34]). As aforementioned, *Smalltalk* is a dynamically typed language, which means that type errors can only be detected at runtime. The environment of *Smalltalk* offers a graphical interactive approach to object-oriented programming since it allows the user to visually explore and develop *Smalltalk* class descriptions. The main issue found in the interaction with the *Smalltalk* graphical user interface was caused by the architectural pattern used to visually represent programs: the *model-view-controller* [23] pattern. This pattern was initially used to

update the representation of programs without stopping their execution. However, it introduces a clear separation between the domain objects (objects that model the real world) and their visual representation on the screen [23]. Furthermore, the views of the pattern are static, not allowing the effects of modifications on the program to be immediately reflected on its behaviour, but only after re-executing it, which hinders interactive development [34].

Later on, and as an alternative to the previously described implementation, an open-source version of the *Smalltalk* environment appeared: *Squeak* [23]. *Squeak* includes an integrated development environment and a visual learning setting for children named *EToys*. *EToys* provides a graphical user interface through which the developer is able to program the behaviour of visual objects, specifying actions that directly manipulate them. It is based on a graphical user interface model named *Morphic* and the visual objects on the screen (such as windows or menus, for instance) are called *Morphs*. All the *Morph* objects have the same basic structure (same properties) and can be manipulated in the same way. When clicking on a *Morph* object, a set of icons (named *halos*) is displayed on the screen. Each *halo* provides a different manipulation action, such as, for example, resize the object or changing its colour. Figure 4.1 displays the *EToys* setting. The environment [10] provides access to a set of painting tools, to allow the user to draw and customize his own objects. The created objects can be moved freely around the screen through dragging and dropping. Each object is assigned a *Viewer*. A *Viewer* is a window that allows the user to change certain properties (such as the object's current position on the screen and rotation angle) and to execute different actions, such as, moving the object forward or rotate it. The system allows for grouping actions in order to create simple programs that visually manipulate the objects, providing a fast way to create simple animations.

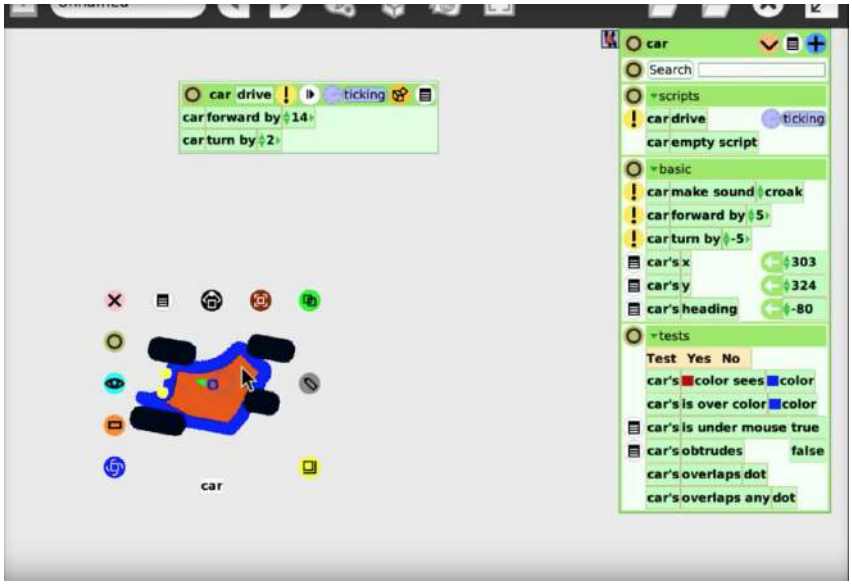


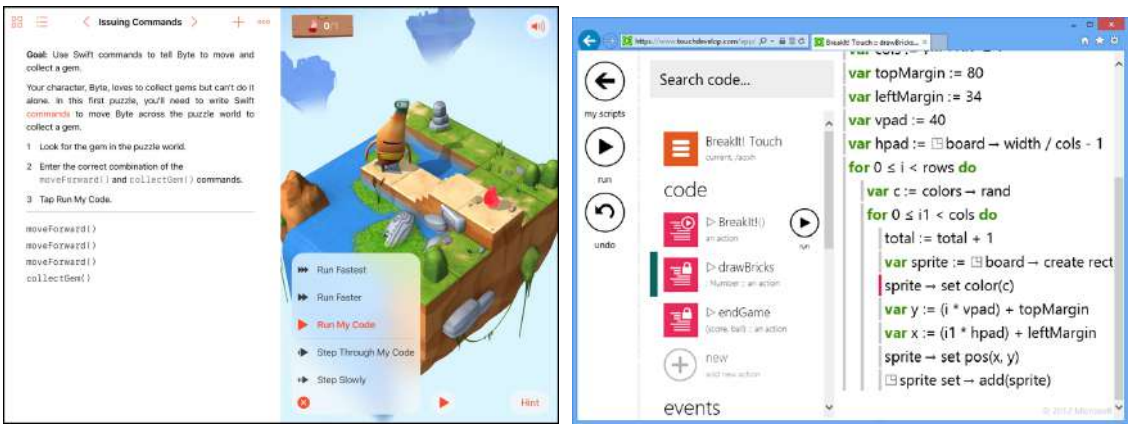
Figure 4.1: *EToys* Learning Environment.

In contrast with the *Smalltalk* environment, in *EToys* there is no separation between

the domain objects and their visual representation, therefore opening an opportunity to explore visual programming. *EToys* falls under the category of direct manipulation interface, which, according to Ben Shneiderman [41], is an interaction style that allows direct manipulation of objects through virtual actions that resemble physical ones [23].

Since the appearance of *Squeak* in 1996, there have been major technological advances in graphical user interfaces and interaction styles. *Swift Playgrounds*, released in 2016 by Apple, can be viewed as a modern variant of *Squeak*. Figure 4.2a displays the application's graphical user interface. *Swift Playgrounds* provides a powerful programming language and environment that allows individuals without programming knowledge to learn the language *Swift* in an interactive way. The graphical user interface is divided into two sections: on the right side of the screen, there is an interactive world where it is possible to observe the main character and a gem it must collect. On the left side of the screen, there is a sequence of instructions to guide the user throughout the problem to be solved [17]. At each level, the user must build small scripts by selecting different available actions, in order to guide the character towards the gem, while facing several obstacles along the way. Moreover, the application provides an interactive way to learn basic programming concepts such as loops or conditional statements. It is conceptually similar to Squeak since it consists of the selection of code fragments in order to directly manipulate visual objects on the screen.

Touch Develop [19], shown in figure 4.2b, is another example of these types of platforms. It is an environment and a typed structured programming language created by Microsoft, that allows users to build applications using exclusively the touch screen as the input device. Similarly to the previous tools, it has built-in primitives that can be selected in order to execute different actions. The user creates applications and games through scripts that perform several computing tasks. The effects of the execution of those scripts are visually displayed to the user.



(a) Swift Playgrounds (b) Touch Develop

Figure 4.2: Visual development environments.

BPMN or *Business Process Model and Notation* is a diagram editor that allows to easily

specify business processes based on flows of information (a technique called *flow-charting*). An example of interaction with the editor is displayed in figure 4.3.

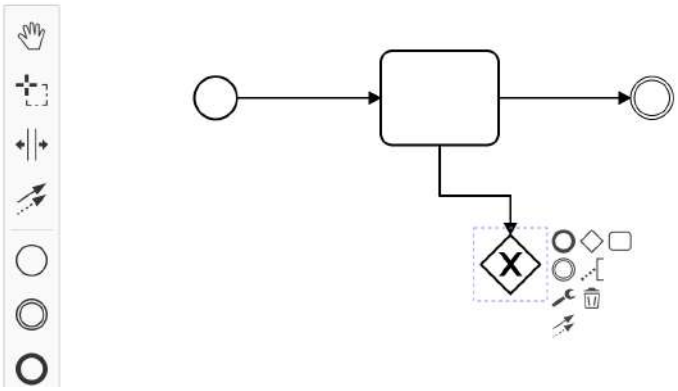


Figure 4.3: Business Process Model and Notation

This editor allows to quickly and effortlessly expand a business diagram. When selecting a business element, the editor displays a group of actions the user can choose from in order to add new elements (as it is possible to observe in the figure). It provides a good approach to diagram construction.

To summarize, the previously described tools are interactive platforms in which is possible to trigger motion on visual objects through manipulation of code snippets. The following list contains the tools that are mentioned in this section:

- Smalltalk (1980). It is an object-oriented programming language and environment that allows the user to save, manipulate, or retrieve data in a visual way [30].
- Squeak (1996). It is an open source implementation of the *Smalltalk* system that includes a learning environment for children called *EToys*. *EToys* allows the user to visually create his own objects and group sets of actions to program their motion around the screen.
- Swift Playgrounds (2016). It is an interactive platform developed by Apple, that allows users with no coding knowledge to learn the *Swift* programming language by choosing the appropriate sets of actions in order to complete different challenges.
- Touch Develop (2011). It is an environment and a typed structured programming language developed by Microsoft that allows building applications through the selection of built-in primitives, without the need of writing actual code.
- Business Process Model and Notation (2011). Editor to create and edit business diagrams based on *flow-charting*.

These tools are essential to explore because of the interaction mechanisms they implement. For instance, just like in the *EToys* environment, we also aim to provide manipulation of visual objects, ensuring that both the domain object (such as data or actions)

and their visual representation are not separate, but part of the same entity. Moreover, as in the original *Smalltalk* environment, we also aim to provide a style of *hot swapping* so that the effects of updates are immediately observable without the need to re-execute the system. However, unlike the previous tools, our interaction is intended to allow the development of actual full-stack applications that can be deployed into a web server and used by clients. Furthermore, the system we propose does not necessarily focus on the construction and execution of scripts that process the behaviour of applications, but rather in an incremental construction where the user is immediately aware of the effects of his actions.

4.2 Dataflow Incremental Programming

As mentioned throughout this document, the model of interaction for our platform must be designed taking into consideration important live coding properties. Therefore, it is crucial to reference and study existing software that provides such characteristics. This section now focuses on the IDE *Light Table* [5]. *Light Table* is an integrated development environment that has advantageous live programming characteristics. It can be seen as an advanced code editor due to the powerful features it includes. It applies concepts such as incremental and dataflow programming [32] to enable continuous feedback on the code written by the developer. More specifically, the editor supports a technique called *REPL* (or *read-eval-print loop*) [33], which allows for evaluation and display of results on code statements as the developer writes them. It is possible to visualize the execution of an application inside of the editor (such as, for instance, a web application) and that execution is kept synchronous with the written code. Moreover, as the developer introduces modifications in the code, it is possible to observe the effects of those modifications in real-time on the application's behaviour or appearance.

Another example of a software tool that takes advantage of dataflow properties to provide a live programming environment is *Circa* [29]. *Circa* is a dataflow-based programming language and environment that allows for incremental construction of applications. *Circa* programming environment supports two different development modes: visual and textual. Both modes are kept synchronous throughout development. It is possible to manipulate visual objects through *drag and drop* techniques and the effects of those modifications reflect in the produced code. Just like in CLAY programming model, it is possible to see a program as a dataflow graph, where each node is a code statement (called a term in *Circa*) and the arcs are dependencies between those statements. The language's dataflow-based programming model allows for a highly introspectable and understandable code: for a given term, it is always possible to trace upwards in the dataflow graph to check where its inputs came from. It is also possible to show the user how a given value was computed [29]. This ability to easily introspect the application's code can be quite advantageous in the debugging process. Moreover, the environment allows accessing filtered views of the produced code by selecting particular elements on visual

mode. Essentially, it means that the developer can access the set of terms responsible for the rendering of an element by simply clicking on it while on visual mode, thus proving the usefulness of a dataflow-based model when aiming to provide an easy introspection and understanding of code.

Despite the clear differences between the two referenced software tools, both aim at providing a style of dataflow incremental programming as a means to ensure the presence of desirable live coding and reactive properties throughout development. The following list summarizes the tools referenced in this section:

- Light Table (2012). It is an IDE that provides immediate feedback on the developer's actions. It supports *REPL*, which means that code statements are evaluated as they are written
- Circa (2013). It is a reactive dataflow language that supports an incremental construction of applications and allows for flexible code inspection. It provides a visual mode and textual mode that are kept synchronous throughout the development

Just like the referenced tools, our platform must provide a style of dataflow incremental construction. Applications are intended to be built by small increments and it may resort to a *REPL*-based technique to provide immediate evaluation of the user's actions. Furthermore, our system is expected, as well, to allow the construction of applications in small increments and without disruptions. However, there are differences to consider between our system and the referenced tools: although it provides a flexible visualization of code updates, in *Light Table*, applications are still built resorting to traditional computer programming. *Circa* allows updates through visual manipulation of objects, but it focuses on programs that are intended to execute locally [32] and not in a web server to be used by several clients.

4.3 Low-Code Software

We design our model of interaction with the intent to provide the developer with an interactive approach to build software applications through direct manipulation of visual entities, without the need of writing actual code. The type of environment we aim to design and build is conceptually similar to the ones provided by *Low-Code* [22] technologies. A *Low-Code* environment allows the creation of application software through a user graphical interface instead of traditional computer programming [12]. Such platforms allow building entire operational applications resorting to minimal hand-written code and focusing on the interaction with the user. *Low-Code* solutions also offer an increase in efficiency during development, since they provide tools that allow to skip the re-implementation of common patterns and optimise the set up of libraries, APIs and third-party infrastructures, thus enabling the developer to focus on the real purpose of the software in production [22]. A typical *Low-Code* environment allows for visually

defining the *User Interface* for the application, to model its business processes and to define its data and logic models.

An example of a *Low-Code* environment is the one developed by the portuguese enterprise, *OutSystems* [22]. The *Low-Code* environment provided by *OutSystems* allows defining the components of a web application and expressing the interactions amongst them. The developer defines the data model for his application visually, creating the relevant database tables and defining their attributes. Upon deployment, the physical tables are updated according to the current information. Moreover, the platform allows to easily build the user interface, resorting to *drag and drop* techniques to define buttons, links and widgets to display particular pieces of information. It is also possible to define the logic workflows by visually specifying sequences of actions (such as, for instance, the update of a particular table with the addition of new entry) upon the triggering of a given event. The deployment is completed with a single click and the application stays accessible through a web browser.

Mendix [11] is another example of a *Low-Code* platform that promotes an agile approach in the development of software applications. *Mendix* has two major components: the *Web Modeller* and the *Desktop Modeller*. *Web Modeller* is a collaborative web-based application development environment that allows for both business analysts (focused on the solution) and the IT developers (focused on the required technology) to collaborate on the process of development. It allows, as well, for citizen developers [21] to quickly change and build applications without any software development experience. The *Desktop Modeller* is an integrated development environment that is locally installed and allows to develop both mobile and web applications whilst offline. Just like the *OutSystems* platform, *Mendix* provides a work set to build entire functional full-stack applications. It allows to visually express the following components: data model, logic model, presentation and security. The data model can be easily built and understood resorting to *Unified Modelling Language* (or UML) notations to express the application's entities, their attributes and associations amongst them. *Mendix* puts considerable effort in providing reusability of components, so the environment supports, as well, the inheritance of entities to avoid data duplication. Similarly to *OutSystems*, the platform allows to visually add complex business logic through the use of flows. It supports two distinct types of flows: micro-flows, that runs on the server-side and expresses, for example, database transactions, and nano-flows, which runs on the client-side and are useful for offline decisions. Figure 4.4 provides an example of a micro-flow in the *Mendix* platform. The graphical user interface can be built through dragging and dropping of dialogues, buttons and other widgets. Furthermore, the platform provides security features to handle threats such as SQL injections and cross-site scripting. It also supports a role-based access control model to allow developers to define what pages, logic and data a user have access given his role on the system.

Mendix also includes an integrated version control system to allow the collaboration amongst developers and access to the complete history of the project updates. A great

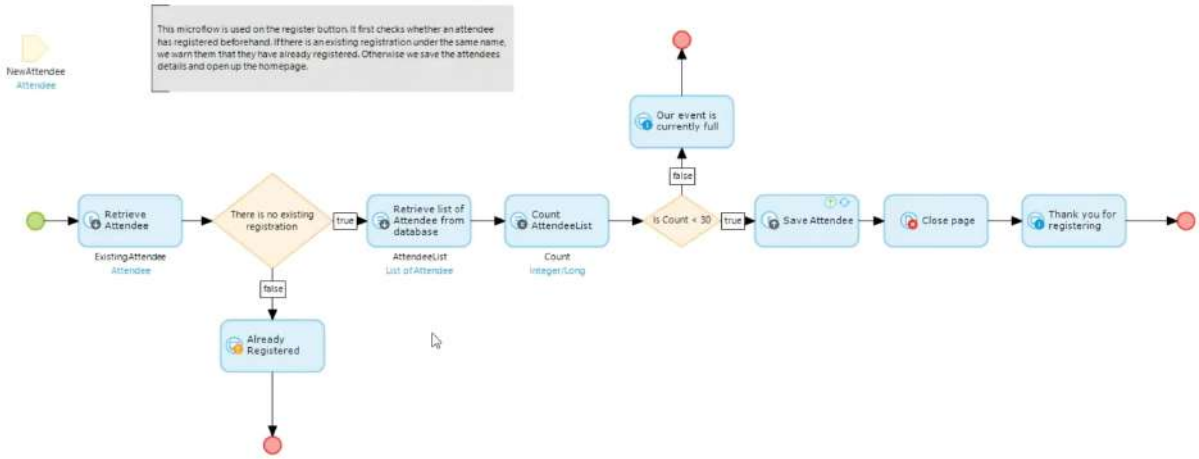


Figure 4.4: Example of a microflow in Mendix [11].

advantage of this platform is its community: developers can share components (such as sub-flows, modules or widgets) on the *Mendix* store, helping others solving recurrent design problems. To finish, the *Mendix* platform is designed to centre the user's needs throughout the design and development processes. It promotes continuous collaboration between stakeholders and developers through a built-in feedback loop integrated into the development environment. Stakeholders can specify new requirements and transmit them to developers through a shared portal. Developers, in turn, can share prototypes with users in order to get real feedback and iterate through different designs.

Both of the previously described tools consists of powerful *Low-Code* systems that allow developing modern, cross-platform enterprise mobile and web applications by visually defining user interfaces, workflows and data models. This chapter includes initial research on the general concept and two examples of existing *Low-Code* technologies. However, during the initial analysis of the design process, both interaction models must be thoroughly explored and understood in order to gather useful insights towards an appropriate design approach.

Although this type of software is conceptually similar to what we aim to achieve, we strive to avoid a visual representation of data and functionality based on UML diagrams and notations. Instead, we aim at providing a more intuitive visual representation of the system resorting to real-life metaphors, and modifications in the internal state of the system must be somehow displayed to the user. Moreover, these tools allow a style of construction based on the *code-compile-deploy* cycle: the effects of updates on the code structure may only be perceived when the program is deployed, instead of providing a constant and immediate feedback on each of developer's actions as we intend to achieve.

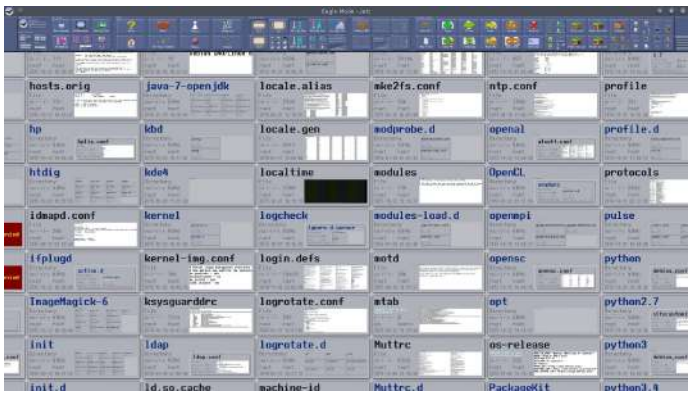


Figure 4.5: *Eagle Mode* graphical environment [6].

4.4 Other Tools

We now describe other tools that follow creative approaches to UI design and data visualization.

Eagle Mode [6] is a file manager that provides a style of GUI based on zooming called *Zooming User Interface*. A ZUI [7] is a graphical environment where the user can change the scale of the environment by simply zooming it, in order to increase or decrease the details of the view. *Eagle Mode* allows the user to access directories and files by zooming into them. Figure 4.5 displays a view of the graphical environment.

Each section represents a directory that is accessed when zooming into it. However, in few situations it may be difficult to precisely determine in which folder the user is at a given time.

This interaction and visualization method may provide useful insights on the way the components of our language are accessed or in how the user navigates through the system's abstraction levels. *Eager Reader* [3] provides another good example of a ZUI, in which the contents of articles can be quickly accessed by zooming them (with a mouse or a trackpad).

Another tool is called *Mercury* [8]. *Mercury* presents an interesting alternative to the standard *Desktop* metaphor, allowing the user to specify his intentions through flows of actions. The atomic unit of *Mercury* is a *Module*. A *Module* is a group of actions based on user's intent. A sequence of *modules* is called a *Flow* and a contextual group of different *flows* required to fulfill an intent is called a *Space*. Figure 4.6 displays the architecture of *Mercury*.

If a user specifies a goal, such as, for example, review his mail inbox, *Mercury* populates a space automatically with flows containing unread messages. It then responds fluidly to the intent without pushing random information (outside the current context) into the screen. *Mercury* is an interesting case because it takes advantage of contrast to bring clarity wherever needed, while obfuscating the rest. Furthermore, it provides an innovative model of interaction that allows to effortlessly navigate throughout *modules*

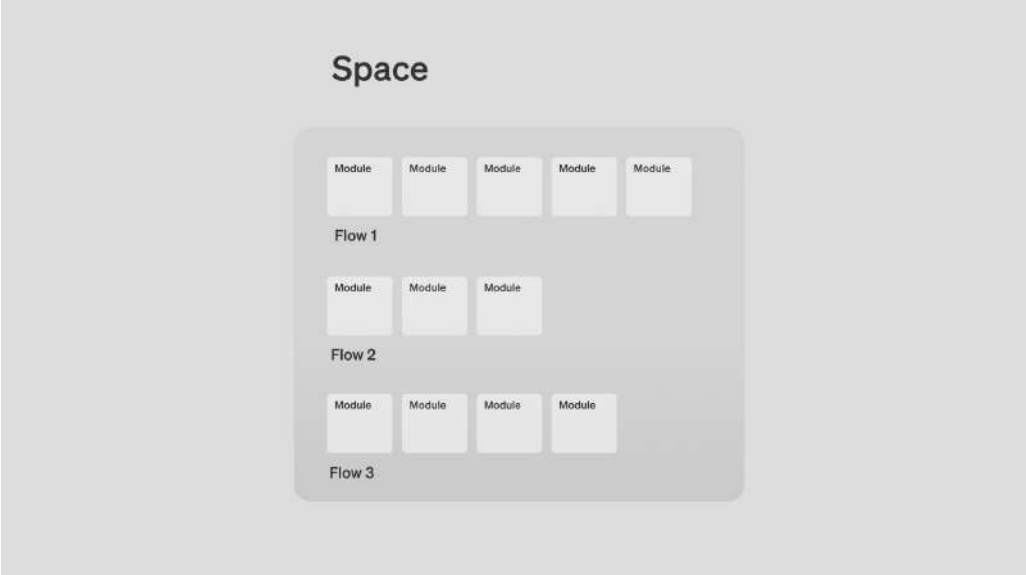


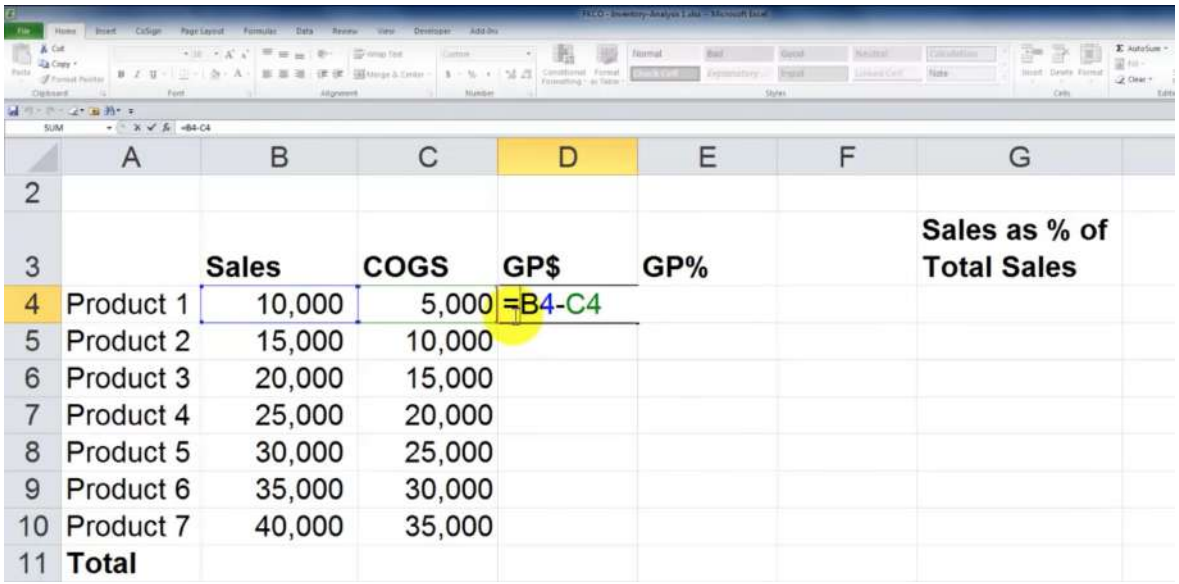
Figure 4.6: Mercury's Architecture.

and *flows* (through a touchable user interface). This can be useful when exploring a way of navigating throughout contextual groups of information in our system.

Another tool that provides an interesting navigation flow is *Prezi* [15]. *Prezi* is an online tool to create presentations and it allows for a good flow between containers of information.

Focusing now on the reactive properties of our system, a very popular tool that provides an interesting case study is *Microsoft Excel* [13]. *Excel* is a spreadsheet (organized by rows and columns) that allows for calculations, graphics and even programming. An extremely powerful feature of *Excel* allows the calculation of multiple cells at once by defining an expression. Figure 4.7 displays an example of this feature.

In the example, the user specifies that the values of column D are calculated by subtracting the value of column C to the value of column B. When the user presses the *Enter* key, column D is populated. When changing a value in column B or C the effect of that modification is propagated to the corresponding value in column D, updating it (the values of column D are dependent on the values of columns B and C). This provides the same reactive behaviour of our system: programming elements may depend on others (we can specify the behaviour of a given element by providing an expression like the one from the example above), and the effects of changing a given element are propagated to elements that are dependent on it. In *Excel*, however, the existing dependencies between elements are not visible. In our system, we want these relationships to be clearly visible and perceived by the user.



	A	B	C	D	E	F	G
2							
3		Sales	COGS	GP$	GP%		Sales as % of Total Sales
4	Product 1	10,000	5,000	=B4-C4			
5	Product 2	15,000	10,000				
6	Product 3	20,000	15,000				
7	Product 4	25,000	20,000				
8	Product 5	30,000	25,000				
9	Product 6	35,000	30,000				
10	Product 7	40,000	35,000				
11	Total						

Figure 4.7: Microsoft Excel Example.

Chapter 5

Methodology

We now describe some of the activities that took place throughout design. We provide a set of system tasks and initial sketches. We also do mention the used technologies in the conception of our prototype.

5.1 Design

In this section, we describe some of the design methods used throughout the conception of the system. As mentioned in section 1.3, we analyse three *User Classes*. The initial goal was to direct the design towards experienced users (users of *type A*) and inexperienced users that could acquire basic software developing skills through the interaction with the user interface (users of *type C*). However, we figured that this would be a non-achievable goal, given the amount of time available for the development of our prototype and the specific needs of each target user group. With this, the design is directed only to users of *type A* and the expansion of the design to inexperienced users is part of future developments.

We begin to idealize our construction model thinking about the inherent properties of the dataflow programming paradigm [16], described in section 2.2. Following this approach, we intuitively know that a program is visually represented by an oriented graph (or diagram), where nodes map to programming elements and the arcs map to dependencies between those elements. Considering this, a diagram expands with the gradual introduction of new programming elements, and, therefore, the set of tasks the system provides consist, mainly, in the addition of new symbols and establishment of new relationships. As it is later explained in section 6.1, each one of these symbols encapsulates *Live Programming* logic, so as the diagram evolves, so does the actual program.

The following step consists of identifying the set of programming elements (or building blocks) we require in order to target the construction process of simple web applications. A very basic and important concept in programming is the *Collection*, which allows to store data elements. In our prototype, this is achieved through the programming element *Table*. A *Table* represents a piece of state that stores entries with the same structure (all entries respect the same data model). We then identify a group of classical operations that modify the state of a *Table*: the *Insert*, the *Delete*, the *Update* and the *Clear* operations. These operations are programming elements that do not have state but modify state. From here, we continue to expand the language and we identify two new operations that do not modify the state of a *Table* but return a logical transformation on it:

- *Size*: consists of an operation that iterates over the entries of a *Table*, increments a value for each entry and returns the result of the sum;
- *Map*: consists of an operation that iterates over the entries of a *Table*, applies a transformation on each entry and returns a collection containing the new set of entries.

We now identify another programming element that represents the same concept of integer variables: the *Number*. A *Number* simply stores an integer value. Then we must have a way of updating the value of a *Number* variable. This is achieved through the programming element *Assignment*. We identify, as well, the programming element *Numeric Transformation*, that performs a logical transformation on a *Number*. The difference between an *Assignment* and a *Numeric Transformation* is that an *Assignment* actually updates the value (changes the state) and the *Numeric Transformation* computes a new value that depends on the value of the *Number*.

So far we defined elements that allows to establish the logic of a web application (the backend of the application), but we do not have a way of interacting with it. We now define a programming element called *Html View*. An *Html View* is an expression that may depend on other elements to display data or change the state of the application. The idea is that each programming element has an *html* representation (for example, a *Table* is translated to an *html* table), and this representation is displayed when the view depends on the element.

According to the previous description, we can identify the following set of tasks:

1. Add a new *Number* variable;
2. Add an *Assignment* that assigns a single value to a *Number* variable;
3. Add an *Assignment* that updates an existing variable to a value provided by the user;
4. Add a *Numeric Expression* that depends on a given existing variable;

5. Add a new *Table* with the following data model: *Name: string*, *Age: number*;
6. Add a *Delete* that removes each entry of a table where the value of *Name* is the same as one provided by input;
7. Add an *Insert* that adds an entry (*Name*, *Age*) provided by the user;
8. Add a new *Map* that sums 1 to the field *Age* in every entry the table;
9. Add an empty *Html View*;
10. Display a list of names on a *Html View*;

The emphasized elements correspond to visual elements that have an associated *Live Programming* definition. It is important to mention these elements do not have a direct association to *Live Programming* (these concepts do not exist in the original prototype). We defined them at the beginning of the design process, in order to identify some possible case studies. Thus, it is possible to add more elements. Figure 5.1 sketches the construction process of a simple list of tasks. It consists of a table of entries, an operation that inserts items into it and a web page that displays the existing tasks and a form to add new ones.

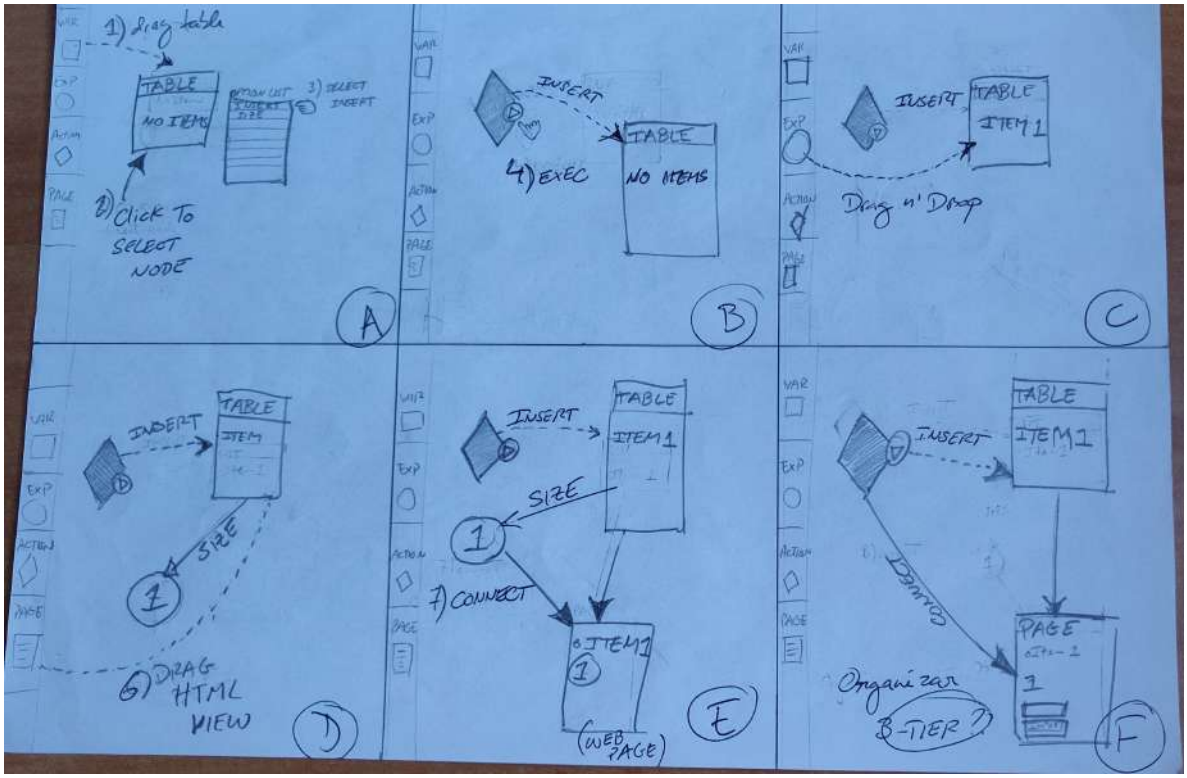


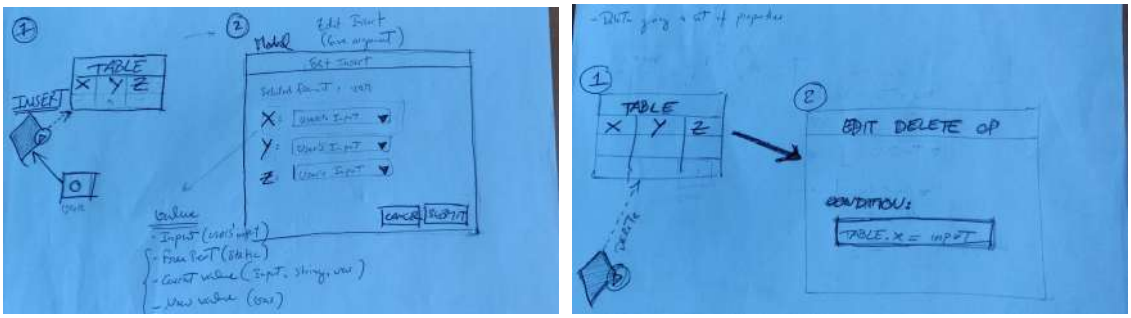
Figure 5.1: Construction of a Task List application.

Here, the visual objects are added to the canvas by dragging them from the palette on the left onto the canvas. When dropping an element, a new instance is added to the

system. The user drags and drops a table to store entries and clicks the element to access a list of options. Selecting one of these options adds a new element to the canvas (as it happens in the *Bpmn* editor or in the learning environment of *Squeak*, described in chapter 4).

The sketch of figure 5.1 and others were analysed by some representatives users (users of *type A*). Some users mentioned that the symbols could be difficult to distinguish in the diagram. It was considered it was due to lack of colour. Other users mentioned the problem of diagram complexity: its complexity could quickly escalate with the size of the application. An approach to handle this issue is the use of *view modes*, described in section 6.5.

As mentioned before, a visual object is associated to a piece of logic. Therefore, when creating a new visual object, the user must provide input data to the system (for example, when creating a table, one must specify its name and data model). The previous sketch does not illustrate this component of the interaction. Figure 5.2 displays two examples on how to specify the body of actions.



(a) Specification of an *Insert* action

(b) Specification of a *Delete* action

Figure 5.2: Data input interaction examples.

When adding an action that inserts an entry on a table (figure 5.2a), the body of the insertion must first be defined by the user. The environment opens a modal that allows to specify a value for each attribute of the table's data model. Figure 5.2b sketches the specification of an action that removes an entry from a table.

5.2 Used Technologies

The implementation of our prototype consists of a *TypeScript* setting that resorts to the D3 *JavaScript*'s framework [1] to render the visual objects on the screen. D3 is a framework for data visualization that allows to bind data to the *Dom* and display it through visual objects.

We use the *TypeScript* framework in order to clearly define each programming element and inheritance relationships between them.

5.3 Focus Group

As previously mentioned, an interesting part of our work consists of proposing different approaches to program visualization (for example, ways of organizing and displaying the components of a program on a diagram). In order to complement the research on interaction techniques and to collect new insights on the subject, we decided to invite a few members of our department (Departamento de Informática da FCT/UNL) to be part of a *Focus Group*. A *Focus Group* is a group interview, in which its members debate ideas concerning a particular matter. It encourages an exchange of insights that provides a deeper understanding of the study being carried. The meetings are guided by a special member, called the *facilitator*.

The main purpose of our meeting was to define different methods to view the structure and state of a program within the context of our visual language and graphical environment.

The first stage of preparation was contacting potential members. After all replies were received, there was seven confirmations from members of the department (five Master's students, one Phd student and one post-doctorate). At that time, three of these members were involved in project CLAY as well.

After this initial stage, a work plan was defined for the meeting. Initially, there was a small briefing to explain the purpose of the work and provide an abstract overview of the system. It is important to mention there was a working prototype at that time and we had planned a small demonstration. However, the plan was to test it at the end of the meeting, in order to avoid restricting different ideas. Instead, the approach for the beginning of the meeting was to encourage members to propose interaction strategies and techniques.

The members were challenged, before and after presenting the general overview of our system, to reason about concrete case studies (the construction of small web applications) to analyse the problem in more detail.

In the end, members answered a set of questions with the intent to detect and address possible problems in the interaction.

The following list defines the events initially planned for the meeting and specifies important points to be mentioned:

1. Briefing
    - Development of a User Interface that allows its users to build web applications through the manipulation of visual elements
    - A user must be able to build each layer of the application (data, logic and presentation) and establish all the relevant connections amongst components
    - Idealize and implement convenient ways to display the components of the program
2. Open discussion

3. Simple case studies
    - Counter
    - Task List
4. Open discussion
5. Prototype's overview
    - Nodes represent programming elements
    - Links represent dependencies between programming elements
    - The actualization of nodes results in the propagation of effects throughout the graph
6. Open-ended questions
    - A user is asked to develop an application that allows to add and remove items from a list. Consider the following user stories: *add a list to the application*, *add an action that inserts an item in the list*, *add an action that removes items from the list*, *add an item* and *add a web page that displays the list to the user*
    - How to represent each programming element in terms of colour and shape?
    - In which ways can it be interesting to visualize the program and the data, considering expert and novice users?
    - How to communicate to the user that a connection between programming elements is allowed?

In the beginning, members were given a paper sheet to sketch possible ideas. Due to the input provided by members, the discussion evolved to related but different topics than the ones originally planned. The subjects discussed throughout the meeting were different than the ones defined during preparation. The first topic to be discussed was the system's target users. The design of the user interface could take very dissimilar directions depending on the target population. The first contribution was defining the relevant *User Classes* of the system (described in section 1.3) and discuss the implications of directing the development towards each of the different user classes.

It was suggested to direct the design to users of type B and base the approach on a low-code solution (one member suggested exploring the *OutSystems* platform and complete a few tutorials). This approach would consist of integrating the concepts of *Live Programming* with *low-code* software. The construction of applications would be guided from the presentation view (web page) and the data model would be hidden from the user (since users of type B do not want to know implementation details). Members suggested it could be interesting to project the interaction for both users of type B and C (while learning the user interface, the design would be directed to users of type B, and when

acquiring solid knowledge solving problems, the design would be directed to users of type C).

The last minutes of the meeting were used to make a brief demonstration of the prototype. The demo consisted of building a small list of tasks. The main concern of the participants was the same as in sketching phase (described in section 5.1): the diagram's complexity quickly escalates with the size of the application. Participants mentioned the user interface would benefit from a module mechanism to allow for an effortless navigation in the diagram.

Members were very engaging and useful suggestions and insights were exchanged throughout the meeting. Although the original prototype was not deeply discussed, we collected information on other approaches and we analysed them afterwards.

The week that followed the meeting was used to explore alternatives to the original design. As suggested, we explored the *OutSystems* platform and completed a set of *low-code* tutorials. As a result, some ideas were documented and sketches were produced to illustrate them. Figure 5.3 shows a sequence of sketches that illustrates the process of building a small list of task application.

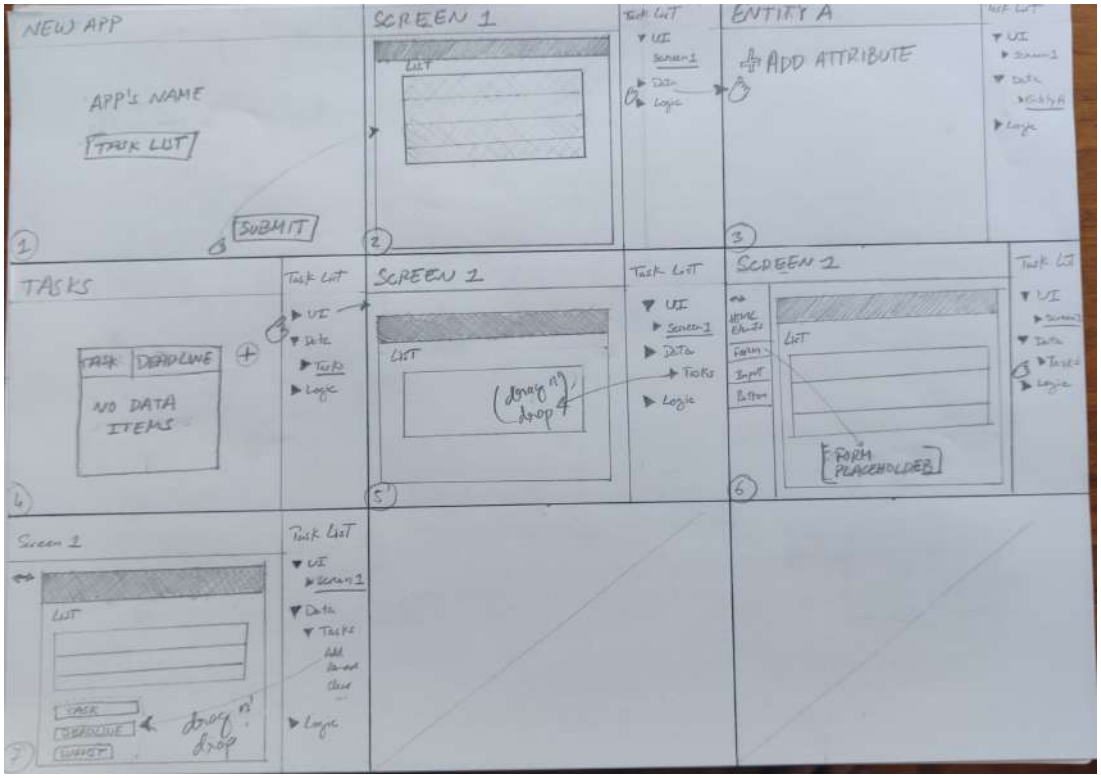


Figure 5.3: Presentation-centered approach.

The focus of this approach is improving the integration between the user interface and the data model of a web application. The construction process is managed from the presentation layer (the component that is visible to the user). There are three main sections: the *UI*, the data model and the logic. The *UI* is associated to a set of screens

and each screen is associated to a set of widgets. A widget may or not be connected to an event: in figure 5.3, the entity *Tasks* is dragged onto the widget *List*, specifying that *Tasks* must be loaded from the database and display the results through the *List*.

A database table provides a set of predefined operations (for example, *Add*, *Remove*, *Update*, etc...). In the example, a user drags the widget *Form* onto the screen (it is still not visible until an event is dragged onto it). After, he drags *Add* onto the *Form*, so that a user can actually add items to the table.

The problem with this approach is that it resembles a lot to a simplified version of the *OutSystems* software, that, from our perspective, handles these tasks very well. After few days of analysis, this idea was discarded. Despite that, the meeting was useful to gather some initial external feedback on our prototype and sketch other ideas.

Chapter 6

Prototype

We now describe the fundamentals that support the design and development of our prototype and model. We first provide an overview of the architecture and few examples of possible interactions. In the following section, we define the structure of our visual language. After, we point the relevant aspects about our graphical environment and we finish the chapter with some sketched program visualization techniques.

6.1 Overview

Our prototype combines a graphical environment (the user interface) and a visual language to support the development of web applications. The client issues requests and receives updated values from the *Live Programming* server, that are immediately displayed in the environment. When the client initiates communication, the environment generates a REST request to the server. After the server updates, the results are sent and the client receives them via web socket. The server processes requests in the same way as in the original architecture of *Live Programming*, described in detail in chapter 2.

The original client consists of a live code editor. The user specifies the behaviour of web applications by writing actual programs with the language defined by the prototype, as exemplified in the *counter* application of figure 2.1. Let us suppose the user defines a new variable, *a*, and assigns it the value 0. To achieve such feat, the user writes the following line of code in the editor:

```
var a = 0
```

This statement must be evaluated in the server, so the following request is issued:

```
POST "var a = 0"
```

```
POST "def aTimesTwo = a * 2"
POST "def inc_a = action { a := a + 1 }"
POST "do inc_a"
```

Figure 6.1: *Live Programming* request sequence example.

If the server's *Interpreter* validates the code statement, the server adds the code to the program (storing it in the database) and it becomes visible in the editor. In the same way, the user may want to define an expression that multiplies *a* by two and increment the value of *a* afterwards. This intent corresponds to the requests in figure 6.1.

The first request defines an expression that multiplies *a* by two. The second one defines an action that increments *a*. The third request executes the action previously defined. Once issued the requests, the server evaluates the statements and the editor displays the definition of the new action and the most recent values of *a* and *aTimesTwo* (1 and 2, respectively).

Through our user interface, these requests are formulated and issued when adding new and manipulating existing visual symbols on the screen, instead of writing the actual code. Therefore, a visual symbol is always associated with a definition statement, such as the ones presented. Adding a new symbol issues a POST request that carries its definition (the statement in the body of the request starts with the keywords *var* or *def*). As we will see, when interacting with some of the existing symbols, the client can also issue requests that update the state of the application (the statement starts with the keyword *do*).

Adding new symbols through our graphical environment, translates in gradually adding new lines of code to the structure, which corresponds to an incremental construction of the application. As the diagram expands, so does the declarative program stored in the server.

Figure 6.2 shows an example of a visual symbol. When adding it to the canvas, the environment prompts the user to enter a name and an initial value for the element (in this case, the number 0). The request is issued and the server adds the statement *var a = 0* to the program (stored in database). The client then receives the update via web socket and renders the value 0 inside the symbol (as it is possible to observe in the figure).

When selecting a symbol, a list of options is displayed next to it. This list provides all the transformations the system allows to perform on the element according to its value. We can think in an analogy with *Abstract Data Types*, since its behaviour is defined by its value and allowed operations. The same can be done when thinking in terms of object-oriented languages, in which an object provides methods that can either modify its state or access its internal information. Selecting an operation from the list corresponds to a new increment and step further on the implementation. Providing only the strictly available operations ensures the system is less error-prone.

In this specific case, there are two operations we can perform on an integer: an *Assignment*, that updates the element's value, and a *Numeric Transformation*, that creates

Figure 6.2: Example of a Visual Symbol.

a logical expression whose value depends on the element's value. On the left side of the element there are two symbols: the small *arrow* icon allows to relate nodes in the diagram (when clicking on the icon, the user is allowed to drag a new line to connect this element to another one in the canvas). The *garbage* icon removes the element from the system. It is important to mention that a given element may only be removed if there are no other elements that depend on it (if the node has no outer arcs). If the user attempts to remove an element from the diagram in these circumstances, an error message is displayed.

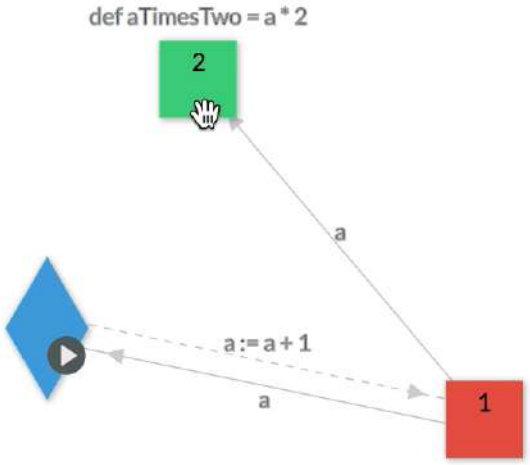


Figure 6.3: Example of a program's visual representation.

Figure 6.3 shows the state of the diagram after issuing the requests of figure 6.1. The differences between the symbols in the figure will be later described. The user initially selects the *Numeric Transformation* option. This results in the first request of figure 6.1. The client receives the response from the server and the new symbol is rendered (represented by the green square in the figure), initially with the value 0. Notice that the

```
1    var a = 0
2    def aTimesTwo = a * 2
3    def inc_a = action { a := a + 1 }
```

Figure 6.4: Example of a program 1.

implicit dependency between *a* and *aTimesTwo* is explicitly represented by the continuous arc (the target symbol depends on the source symbol). The user then selects the *Assignment* option, issuing the second POST request in 6.1 (the blue diamond is rendered in the canvas). The dashed arc between *inc_a* and *a* expresses that *inc_a* can issue an imperative update on *a*. Because the action requires the value of *a* to update its value, a second dependency is created (*a* is the source element and *inc_a* is the target element). At this point, the program of figure 6.4 is defined and stored.

As it is possible to observe in the previous figure (fig. 6.3), when hovering the cursor over a node, the *Live Programming* definition of the element is displayed to the user. Considering we design our system around users with, at least, basic programming knowledge, this information is useful when reasoning about and understanding the several dependencies that relate the elements throughout the diagram. Firstly, our intuition was having this definition constantly attached to the corresponding node as a label. This solution, however, would imply "flooding"the diagram with unnecessary information that could make it harder to perceive, lowering the *cognitive effectiveness* [35] of the user. Instead, we decided to keep this information hidden and only displayed when requested.

The last step is to increment *a*. This goal is achieved when clicking on the *play* button in the action. This issues the third and last request of figure 6.1. The action executes in the server and updates the value of *a* to 1. The assignment to variable *a* triggers the effects' propagation and, consequently, the value of *aTimesTwo* changes to 2. Through our environment, the user is actually able to observe the propagation of effects throughout the graph, as the affected nodes and relevant arcs are highlighted.

By analysing the previous example, we know, intuitively, that it is possible to translate a declarative program to a directed graph, where the nodes correspond to programming elements, and the arcs correspond to dependencies between those.

It is also important to mention that these elements may require input data from the user (for example, an action may require arguments to compute a value). When this happens, the environment prompts the user to enter values for each one of the required arguments.

The previous example describes the construction process of a simple application. However, it is only possible to interact with the program through the graphical environment. What we aim to achieve is the interaction with the application through a web page. This means each defined element must have a corresponding *html* representation.

Let us consider the simple program of figure 6.5. The program defines two elements: an integer *a* with value 0 (line 1) and a an action that sums an argument, *inNum*, with *a*

```
1    var a = 0
2    def sum_a = (inNum) => action { a := a + inNum }
```

Figure 6.5: Example of a program 2.

and assigns it to *a* (line 2). The element defined in the first line consists simply of a piece of information, therefore it is translated to the following markup:

```
<div class="single-value" id="a">
    <label>"a"</label>
    <div>"0"</div>
</div>
```

The element defined in line 2 consists of an action that receives one argument. Therefore, it requires one input field and one button that triggers the execution. It is translated into the following markup:

```
<div class="action-form">
    <form>
        <div class="in">
            <label for="inNum">"inNum"</label>
            <input id="inNum" type="number"/>
        </div>
        <button class="do-action-button"
            doaction=( action { a := #inNum + a } )>
            "sum_a"
        </button>
    </form>
</div>
```

The environment allows to effortlessly add a *html* view to the diagram. Figure 6.6 displays the structure of the graph after connecting both elements to the view.

As it is possible to observe in the figure, both markups are joined together inside the view. The input field receives the argument for the action, *inNum*, and the button is associated to an event that executes it.

There are several occasions where the system requires input data from the user. This happens in situations where the user creates a variable and must assign it a value, has to specify the body of an expression or has to provide arguments for the execution of a given action. When adding a new element and providing its specification, the user may reference existing programming elements (already defined in the program), provide input values, or reference table attributes. Our environment contains a *parser* that analyses the input provided by the user. When submitting the specification and receiving a positive

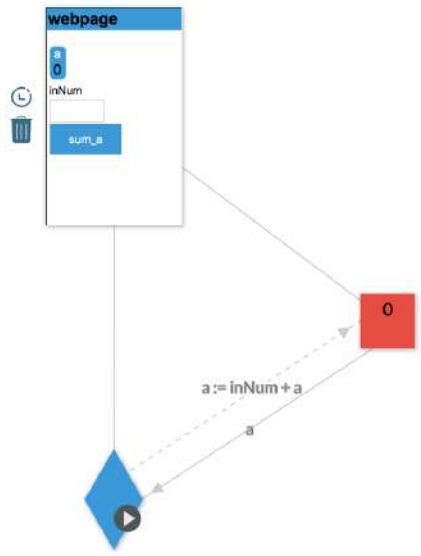


Figure 6.6: Example of a program containing a *Html View*.

response from the server, the *parser* searches for references to existing elements and the environment renders the corresponding dependencies. Figure 6.7 illustrates an example.

Initially, the graph defines a program with three integer variables: x, y and z, each one assigned with the value 5 (fig. 6.7a). The user wants to add a logical expression that computes the sum of all elements. Through the modal, the user provides a name for the new element, *elemSum*, and the intended expression, $x + y + z$ (fig. 6.7b). After submitting, the following line is added to the program:

```
def elemSum = x + y + z
```

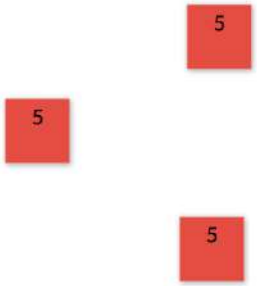
The *parser* analyses the expression and the corresponding dependencies are rendered (as it is possible to observe in figure 6.7c).

When defining an expression, the user may also specify input values. As an example, let us consider, once again, the program of figure 6.5. The definition of *sum_a* (line 2) receives an argument, *inNum*. When creating the action, the action defined in line 2 is generated by the following specification:

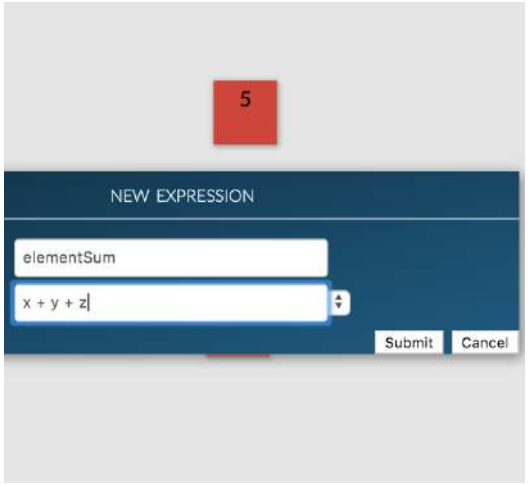
```
a + inNum
```

When evaluating the specification, the *parser* processes *inNum* as an input value because it recognizes the prefix *in*. When executing *sum_a* through the user interface, the environment prompts the user to provide a value for *inNum*. When generating *sum_a*'s *html* representation, the environment generates one input field (that corresponds to the only argument).

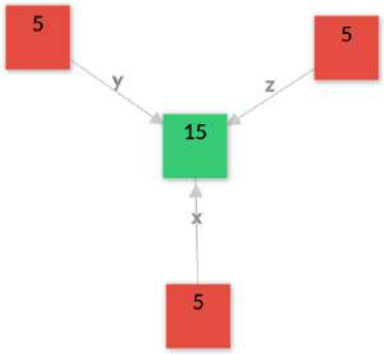
There are situations where table attributes may need to be referenced in the specification. A case where this happens recurrently is the *map* operation. A *map* function consists



(a) First: unrelated variables in the canvas.



(b) Second: specification of a new expression.

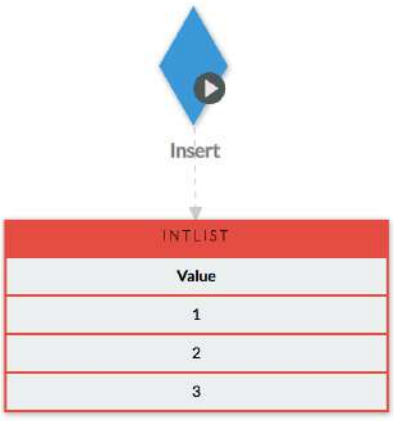


(c) Third: visual representation of the new expression.

Figure 6.7: Rendering of program's dependencies.

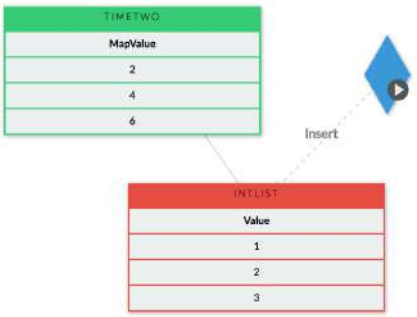
of iterating over the entries of a table and apply a modification on each one. We show an example of this in Figure 6.8.

The application illustrated in figure 6.8 consists of *IntList*, a single attribute table, whose attribute, *Value*, stores integers. If the user wants to define an operation that multiplies each of table's values by two, the user should select the *Map* operation from the list of options and provide a name (*TimesTwo*) and specification (*IntList.Value * 2*) for the function. This step is illustrated in figure 6.8b. The *parser* detects the existence of an attribute and the environment establishes a dependency between *IntList* and *TimesTwo*, since the *map* depends on the actual table of integers (fig. 6.8c).



(a) First: integer collection and insert operation.

(b) Second: specification of a map that multiplies each entry by two.



(c) Third: visual representation of the map.

Figure 6.8: Rendering of a *Map*.

6.2 Visual Language

The second component of our prototype is the visual language. The visual language maps *Live Programming* statements to visual symbols. In this section, we initially define its vocabulary and justify some design decisions. We then specify the rules of the language (how the elements relate) and describe the meaning of some semantic constructs.

6.2.1 Visual Vocabulary

The *visual vocabulary* corresponds to the domain of visual symbols the language operates with. Our *visual vocabulary* consists of two groups of symbols: nodes and arcs.

A node visually represents a programming element; in the context of our system, a programming element corresponds to a *Live Programming* definition (a code statement starting with *var* or *def*). Therefore, a node must visually carry information on the programming element itself, in order to help the user perceive its purpose in the system.

One initial issue to tackle is that programming elements may represent distinct constructs with very different purposes (integer variables, tables, text, assignments, etc...), making it necessary to distinguish them visually. Our first intuition was to follow an icon based approach to represent programming elements (each node of the graph is represented with an icon). However, this first intuition is flawed; taking into account that we may want to integrate new programming elements, it is unsustainable to design a new icon each time a new element is added to the visual language.

As described in chapter 2, the programming model covers three important concepts that may be involved in the definition of an element:

- state variables, if it is part of the application's state (defined with *var*);
- pure data transformation expressions, if it produces some sort of logical transformation (defined with *def*);
- actions, if it can trigger a state update on other elements (defined with *action*).

Let us name these concepts *categories*. The *category* is essential because it represents the general responsibility of a programming element. Therefore, information on the *category* must be carried by the node. However, these concepts have semantic meanings that are difficult to visually represent. For this reason, we use different properties of the visual element to transmit information to the user. For instance, we use the shape of the node to display its values and the colour to convey its *category*. Figure 6.9 displays three nodes, each one representing a programming element from different *categories*.

(a) Number (b) Action (c) Map

Figure 6.9: Examples of visual symbols.

For now, let us focus on the color of a node. There is not a logical cognitive association between a specific colour and the meaning of the *category*, therefore the colour that assigned to each one is not essential, as long as these are clearly distinguishable amongst each other. State variables are represented with the colour red (figure 6.9a), actions are represented with the colour blue (figure 6.9b) and pure data transformation expressions are represented with the colour green (figure 6.9c).

Let us now consider the complete domain of programming elements covered by our visual language:

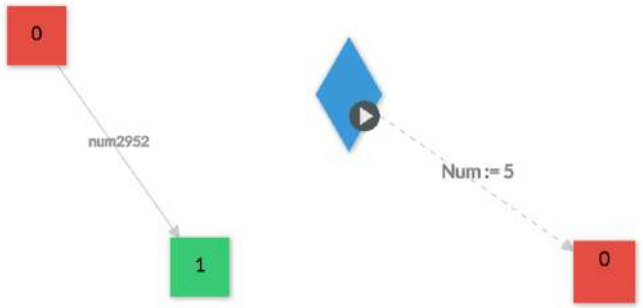
- State Variables
  - Number. Defines an integer variable;
  - Text. Defines a string variable;
  - Table. Defines a table of entries;
- Pure Data Transformation Expressions
  - Numeric Expression. Defines a logical numeric transformation on an integer;
  - *Html* View. Defines an expression that displays content;
  - Size. Defines an operation that retrieves the size of a table;
  - Map. Defines an operation that performs a transformation on each entry of a table;
- Actions
  - Assignment. Defines an action that assigns a value or expression to an integer.
  - Insert. Defines an action that inserts a table entry.
  - Delete. Defines an action that removes a set of table entries.
  - Clear. Defines an action that clears a table.
  - Update. Defines an action that updates a set of table entries.
  - Edit Text. Defines an action that sets a text field.
  - Action Sequence. Defines a sequence of actions.

As mentioned before, each one of the previous programming elements has a piece of logic matching to a definition written in *Live Programming* code. The semantic meaning of each one of these elements is represented by their current value (for example, the *Map* of figure 6.9c represents a logical transformation on each entry of a *Table*, and thus having the appearance of a table). Actions do not have value and, therefore, no content. An action carries only the logic to update an element, and its semantic meaning derives from the usage context (the element it affects, or the node it is connected to). Furthermore, an arc between an action and the element it affects has a complementary textual notation that describes it. We often use textual notations to complement the representation of elements, following the *Dual Coding* principle, described in section 3.4. Actions do not have a value to display, thus being possible to assign them a different shape and colour from the other *categories*, making them highly distinguishable in the diagram and contributing to a higher *visual expressiveness* (two *visual variables* are modified).

The second group of visual symbols are arcs. Arcs visually represent dependencies between programming elements. A dependency relates two programming elements: the *source* and the *target*. Dependencies are classified as *regular dependencies* or *modification*

dependencies. In *regular dependencies*, the *target* depends on the value of the *source* to compute its own value, or, in some cases, use it to compute a third element's value (if it is an action). In *modification dependencies*, the *source* can trigger a state update on the target.

Figure 6.10a displays an example of a *regular dependency*. These are represented by a continuous line. The value carried by the *target* (*value = 1*) depends on the value carried by the *source* (*value = 0*). Figure 6.10b illustrates an example of a *modification dependency*. These are represented with a dashed line. Upon the execution of the *action*, the value 5 is assigned to the *target* (*Num := 5*).



(a) Regular Dependency (b) Modification Dependency

Figure 6.10: Types of dependencies.

6.2.2 Visual Grammar

We now define the *visual grammar* of the language. A *visual grammar* is a set of rules that specify how to connect and organize programming elements. The following list contains a set of high-level rules that restrict the dependency domain of our language and define how elements can be related based on their *category*.

a) State Variable → Pure Data Transformation Expression

b) State Variable → Action

c) Pure Data Transformation Expression → Pure Data Transformation Expression

d) Pure Data Transformation Expression → Action

e) Action → State Variable

Let us examine the previous notation: considering a dependency in the language, the element at the origin of the arrow (the left-side operand) denotes the *source* and the element at the end of the arrow (the right-side operand) denotes the *target*. For example, rule *a* denotes that the value carried by a pure data transformation expression can depend on the value carried by a state variable.

When an action, A, is the *target* of the dependency (rules b and d), A requires the value(s) carried by the source(s) to compute a third node's value (the *target* of the *modification dependency*).

Rule e denotes that a state variable, S, can only be the *target* in a relationship, *rel*, if an action is the *source* of *rel* (values held by state variables are never dependent on any other elements of the program). Similarly, an action, A, can only be the *source* of *rel* if a state variable, S, is the *target* in *rel* (actions can only modify state variables).

Rule e defines a *modification dependency* and the remaining rules define *regular dependencies*.

The previous list defines a set of high-level rules that restrict the language's dependency domain. However, these rules are not enough to precisely define the relationships the language supports. The establishment of these rules is solely based on the semantic meaning of the *category* and not on the programming element itself. When analysing the set of programming elements of the language, not every defined relationship is semantically valid. As an example, let us consider a *Number*, n, and an *Insert*, i: in the previously defined domain, the rule $i \rightarrow n$ is acceptable. However, an integer does not provide methods of insertion (it is not a collection). Therefore, the semantics are invalid and the dependency should not be allowed. Whenever a user attempts to establish an invalid dependency, the colour of the *target* changes to gray, informing him that that particular relationship is not allowed. An example is displayed in figure 6.11.

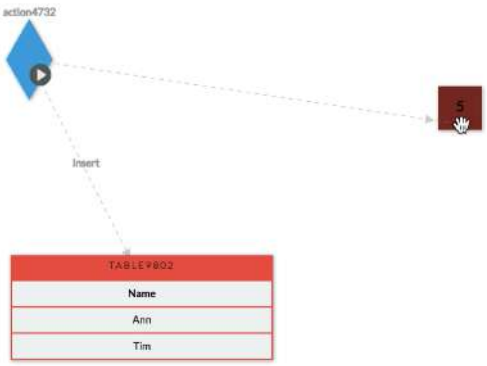


Figure 6.11: Example of an invalid dependency.

Given the domain of programming elements of the language, we now define a new set of rules that precisely define the dependency domain:

a) Number → Numeric Transformation

b) Table → Map, Size

c) Assignment → Number

d) Edit Text → Text

e) Insert, Delete, Update, Clear → Table

f) Numeric Transformation → Numeric Transformation

g) Map → Size

h) Number, Text, Table, Numeric Transformation → Insert, Delete, Update

i) Number, Numeric Transformation → Assignment

j) Text → Edit Text

k) * → Html Expression

Let us consider the following generic rule, *r*: *x*, *y*, *z* → *m*, *n*, *o*. *r* denotes that a dependency where the *source* is *x*, *y* or *z* and the *target* is *m*, *n* or *o* is valid. Let us examine a subset of previous rules: rule *a* denotes that a *Numeric Transformation* depends on a *Number*. A *Numeric Transformation* is an expression that depends on other numeric variables in the program. For example, it may be an expression that multiplies a *Number*, *n*, by two (*f* = *n* * *2*). This example is represented in figure 6.12, where *n* carries the value 4.

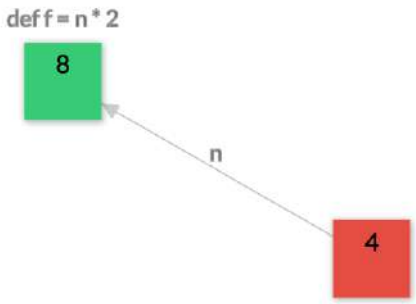


Figure 6.12: Dependency between a *Number* and a *Numeric Transformation*.

Rule *e* declares that the state of a *Table* variable may be modified by the *Insert*, *Delete*, *Update* or *Clear* actions: an *Insert* action adds new entries to the table (it may be a declared variable or data provided by the user); a *Delete* action removes table entries that verify a given condition; an *Update* action updates entries that verify a given condition; a *Clear* action removes all the entries from the table.

Rule *k* states that if a dependency, *rel*, has a *Html view* as *target*, then *rel* is always valid. The symbol * represents the full domain of programming elements. This means that any node is interpretable as *html* markup and displayed to the user. It is important to mention that the full domain does also include the *Html View* itself. This allows for

a hierarchical construction of the web page (a *html* view may depend on other views to display data to the user).

6.3 Graphical Environment

The graphical environment provides the tools and features to build web applications. It consists of a user interface that communicates with the *Live Programming* server. Its structure is designed to only contain the strictly necessary user interface components, so it can be easily perceived by its users. It has three main components:

1. the canvas section: it is responsible for hosting the construction of applications. It is the largest section in the user interface;

2. the left-side menu: it allows for adding new programming elements and modules to the system. It contains a search bar to find elements by keyword;

3. the right-side menu (or the detail section): provides details and information on a selected programming element.

Figure 6.13 displays the user interface. The canvas is the section where the user manipulates the diagram by dragging elements and establishing new dependencies. Through the left-side menu, the user adds new programming elements to the system: it consists of a *pallete* of elements, each one visually represented with a label (that identifies its semantic meaning) and a symbol (that allows the user to establish a faster visual association between its meaning and representation). When selecting a symbol from the menu, the user adds a new instance of that element to the system (a new symbol appears in the canvas and it is added to the diagram). From this section, it is not possible to add actions to the system: both state variables and pure data transformation expressions may actually be part of the diagram without any input and output dependencies (disconnected from the remaining nodes). State variables are never dependent on other elements and expressions may or may not be. Actions can not exist in the diagram when not connected to any other elements. Therefore, the environment only allows to add an action after selecting the element it is supposed to update. This way, we take into consideration the *constraints* principle [37], described in section 3.2.

The detail section provides additional information on a currently selected node. The corresponding code statement and actual value are displayed in this area. As an example, if a *html* view is selected, this section shows the contents of the page.

Similarly to standard code editors, our environment contains a search bar that allows for the quick search of particular elements in the canvas. Each programming element is associated with a set of keywords. The search mechanism identifies all the nodes tagged with a given keyword and highlights them to the user. In the example of figure 6.14, the user types the keyword *State* in the search box and, as a result, all the state variables

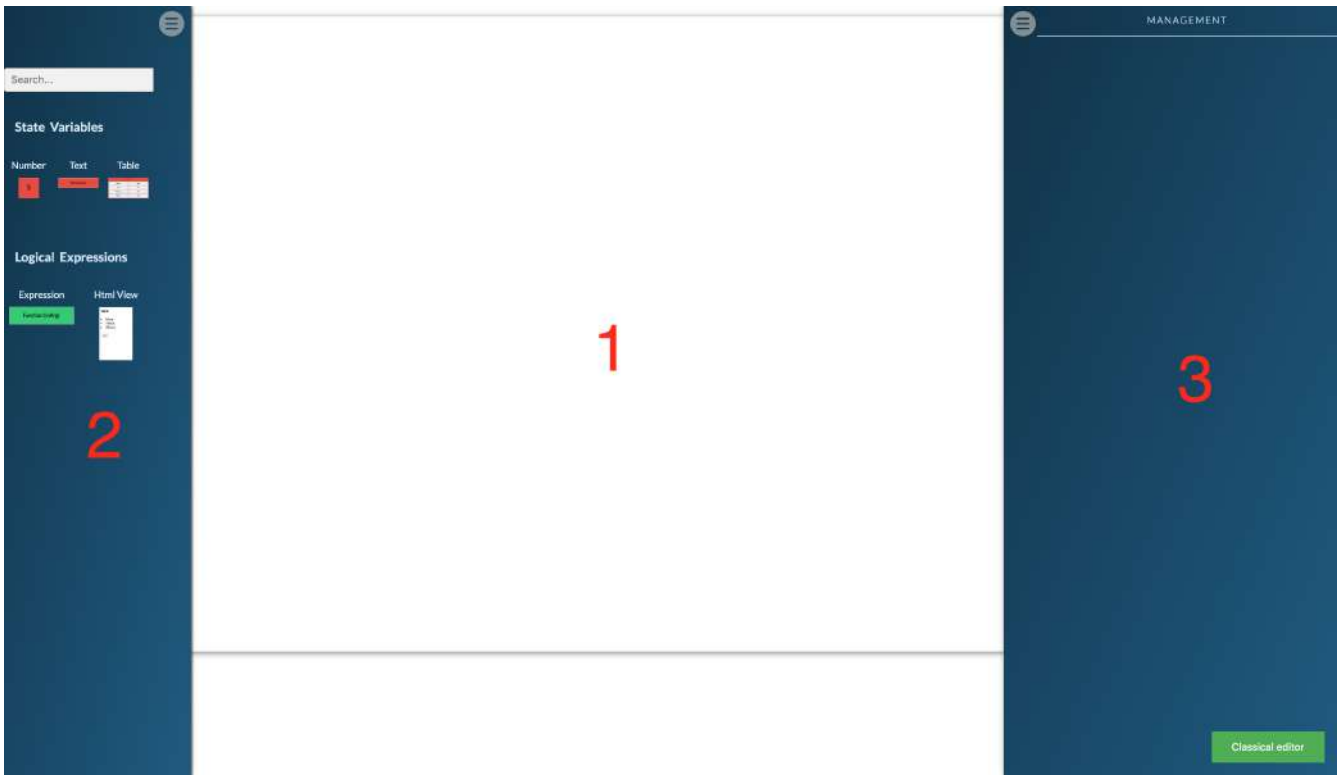


Figure 6.13: Structure of the workspace.

are highlighted. Other examples of keywords are *Action*, *Insert*, *Expression*, *Table*, *Web* or *Page*.

Throughout the development of a web application, there are several occasions where the system requires input data from the user. This happens when, for example, the user creates a variable and needs to assign it an initial value, has to specify the body of an expression, or provide arguments for the execution of an action. Each time the system requires data from the user, the environment renders a modal. As an example, Let us consider the modal rendered in figure 6.15.

A sketched version of this modal is provided in section 5.1. This modal requests data for a new *Insert* action. The table has three attributes: *Name*, *Age* and *Country*. For each, the modal includes an *input* field and a *select list*. The purpose is to allow the user to freely write a specification (in the *input* field) while providing him a list of variables to help him achieve that.

6.4 Development Details

We now explore important modelling decisions made throughout the conception of our prototype. Our goal is to provide a more technical overview of the development process and a greater understanding of our visual language's internals. Due to the complexity of the overall model, we analyse two sub-models separately to better comprehend the

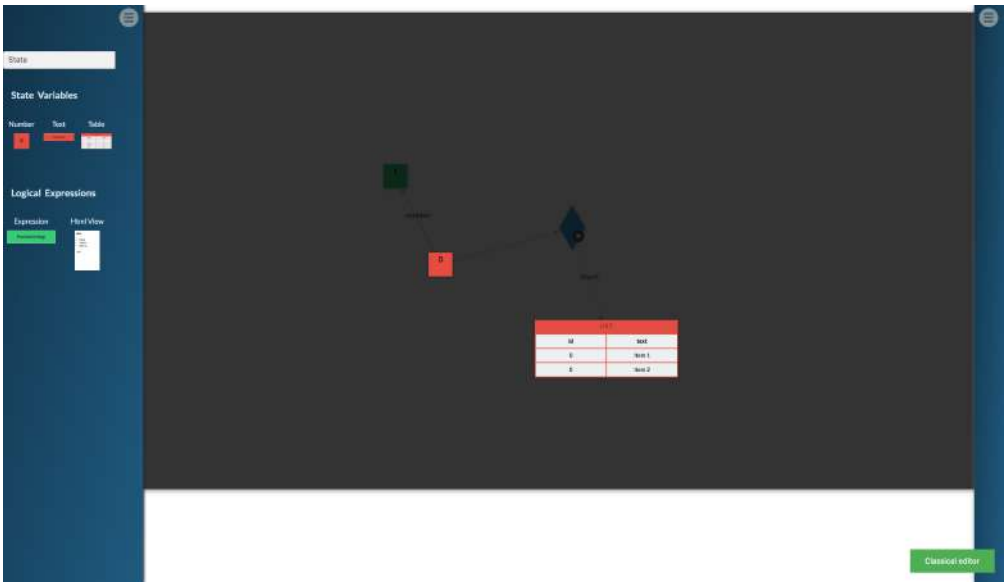


Figure 6.14: *Keyword-Search* feature example.

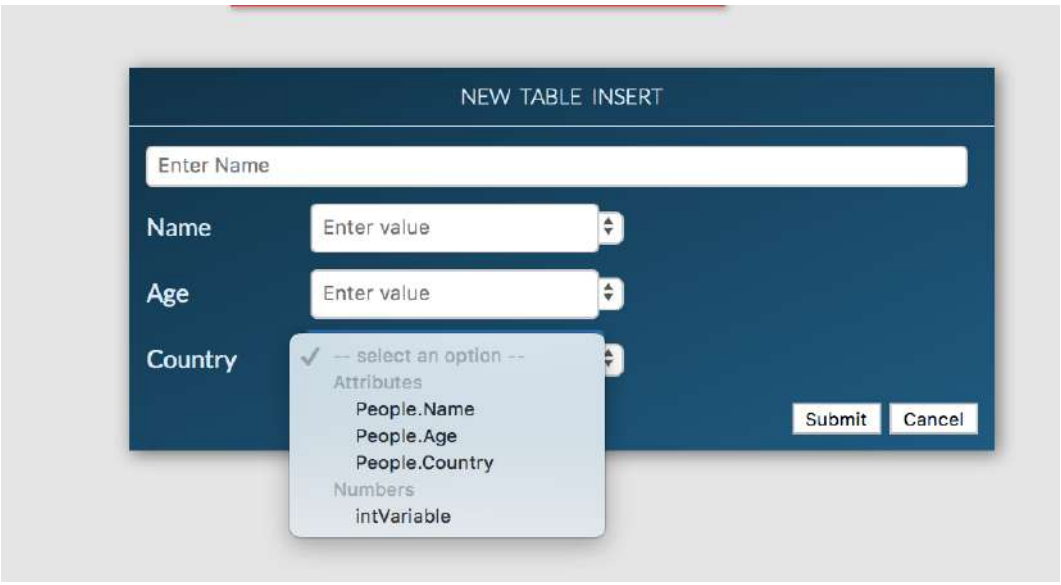


Figure 6.15: *Insert* specification modal.

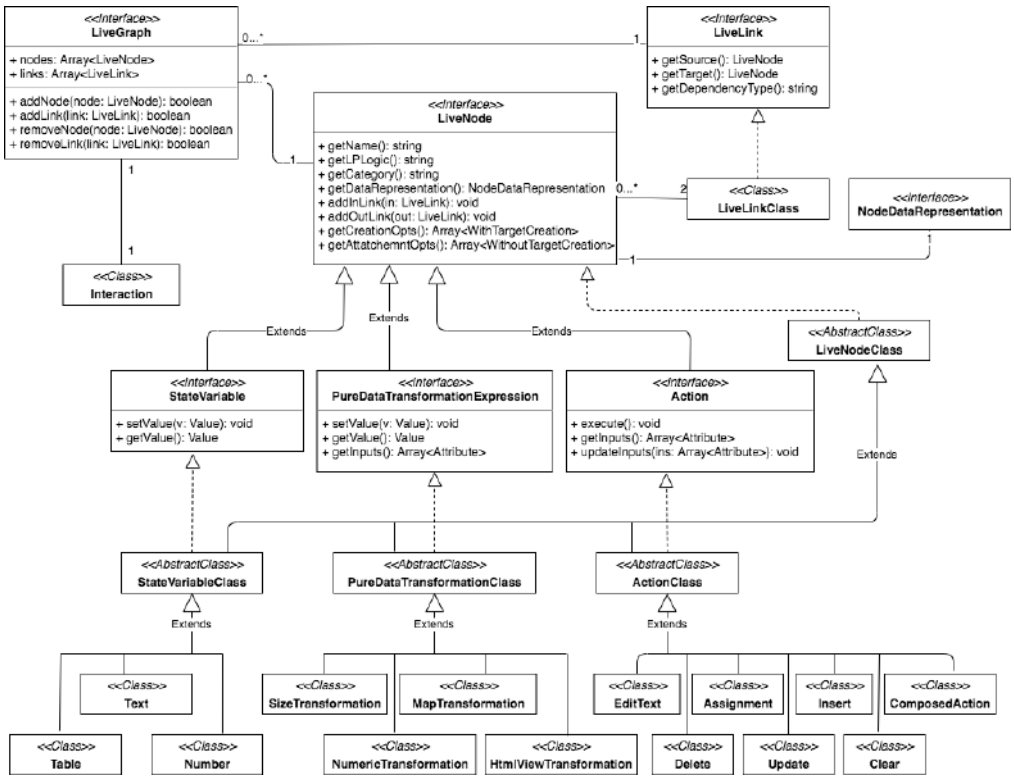


Figure 6.16: System language class diagram.

structure of the language.

The first sub-model, illustrated in figure 6.16 UML class diagram, denotes the class hierarchy that defines a programming element and how does it relate to other components in the system.

The *LiveNode* interface models the general behaviour of a node in the system: the *addInLink* and *addOutLink* methods allow to attach new entering and exiting dependencies to the node, respectively; *getCategory* provides the *category* of the element (mentioned in section 6.2); *getLPLogic* provides the *LiveProgramming* definition associated with this particular node; and *getDataRepresentation* returns an instance of *NodeDataRepresentation*, responsible for encapsulating the logic to render the node's content to the graph (such as dimensions, shape, content or color). *LiveNode* has three extending sub-classes, each one representing a particular *category*. Programming elements are sub-classes of *categories*.

The *LiveLink* interface relates two *LiveNode* instances in a dependency relationship (the *source* and the *target*), and provides a method, *getDependencyType*, that informs the system if it is a *regular* or *modification* dependency.

The *LiveGraph* type consists of two data structures: one that stores *LiveNode* instances and one that stores *LiveLink* instances. This type provides methods to add or remove elements from both structures.

Lastly, the *Interaction* class binds the described type structure to the *D3* framework: it is responsible for binding the data stored in the *LiveGraph* instance to its corresponding

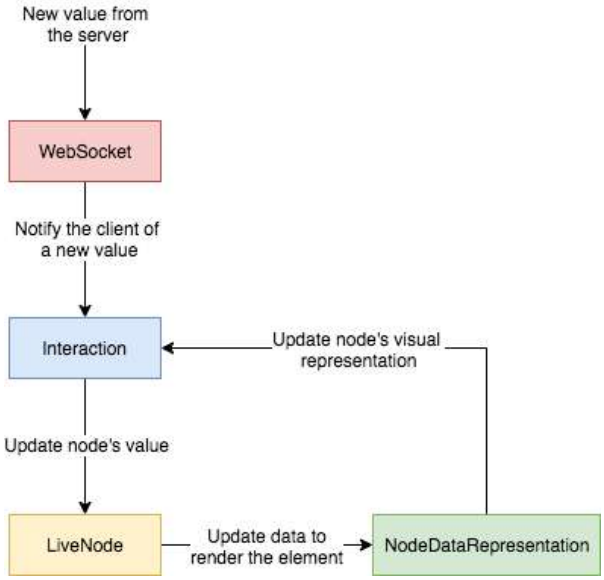


Figure 6.17: System language class diagram.

visual representation, rendering the visual objects to the screen. The scheme of figure 6.17 illustrates the process of updating the visual representation of a node.

When a value is updated on the server, the client is notified of that update via a web-socket. The new value is passed to the *Interaction* instance, where the graph structure is traversed in order to target the correct node. When the element is found, its value is updated and that update is communicated to its data visualization instance (of type *NodeDataRepresentation*). To finish the process, the *Interaction* instance is notified and it updates the visual representation of the node in the graph.

The second sub-model, illustrated by the class diagram in figure 6.18, structures the creation of new dependencies. It is important to mention that the *LiveNode* type is the same type from the diagram illustrated in figure 6.16. We display only the relevant methods in the context of this representation and omit the remaining ones for the sake of simplicity.

Each *LiveNode* instance has a collection of *RelationshipFactory* objects as one of its fields. An object of this type allows to create dependencies of a given kind (for example, *OptionAssign* allows to create assignment dependencies between *Number* and *Assignment* elements).

As seen in previous sections, it is possible to add a dependency between two nodes when both of them exist (when the user drags a line on the canvas to connect both nodes) or before one of the nodes exist (when the user accesses a node's option list and selects one of them, a new node and dependency are created). These two kinds of dependencies are added by the *attach* method of *WithoutTargetCreation* and the *create* method of *WithTargetCreation*, respectively. Both methods consist of issuing a request to the server and rendering the relevant arcs in the canvas when obtaining the response.

The collection of *RelationshipFactory* instances assigned to each node depends on the

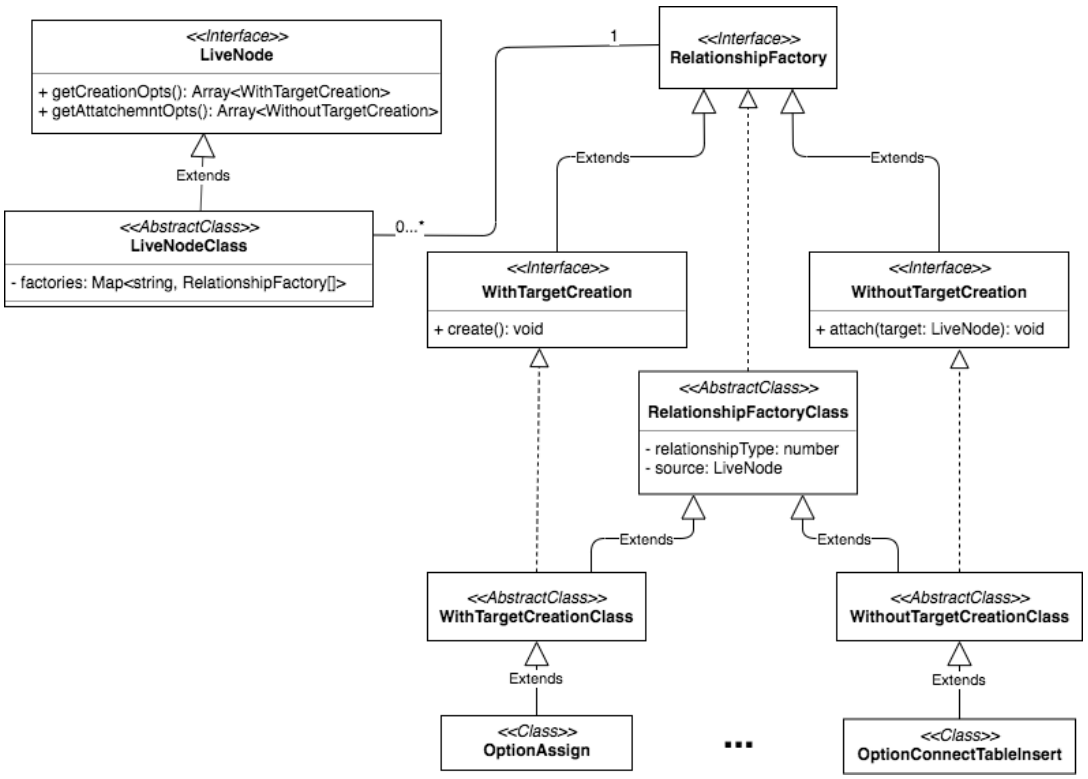


Figure 6.18: Creation of dependencies class diagram.

actual programming element. For example, if the node is a *Number*, then it must have a factory that allows to produce assignment dependencies. The collection for each element gets assigned upon the element's creation, and its generation depends on a static data structure that stores the rules of the language. While traversing the rules structure, the new element stores the factory instances that allow it to produce valid relationships.

6.5 View Modes

As we described in the previous sections, our implementation supports a subset of features of the *Live Programming* prototype. We now present some interaction techniques we idealized beyond the implementation of our system.

The primary purpose of this prototype is to allow for a visual and incremental construction of software. However, an important aspect of this research and crucial component of the interaction are *view modes*. *View modes* are merely ways to visualize an application's current state and construction process. When thinking about *view modes*, an interesting question to ask is: how can elements of the program be rearranged and displayed in order to let the user know what is happening in the background? An important aspect of a *view mode* is the organization of the program's structure. However, it may also serve another important purpose, such as helping the user to effortlessly navigate throughout the diagram.

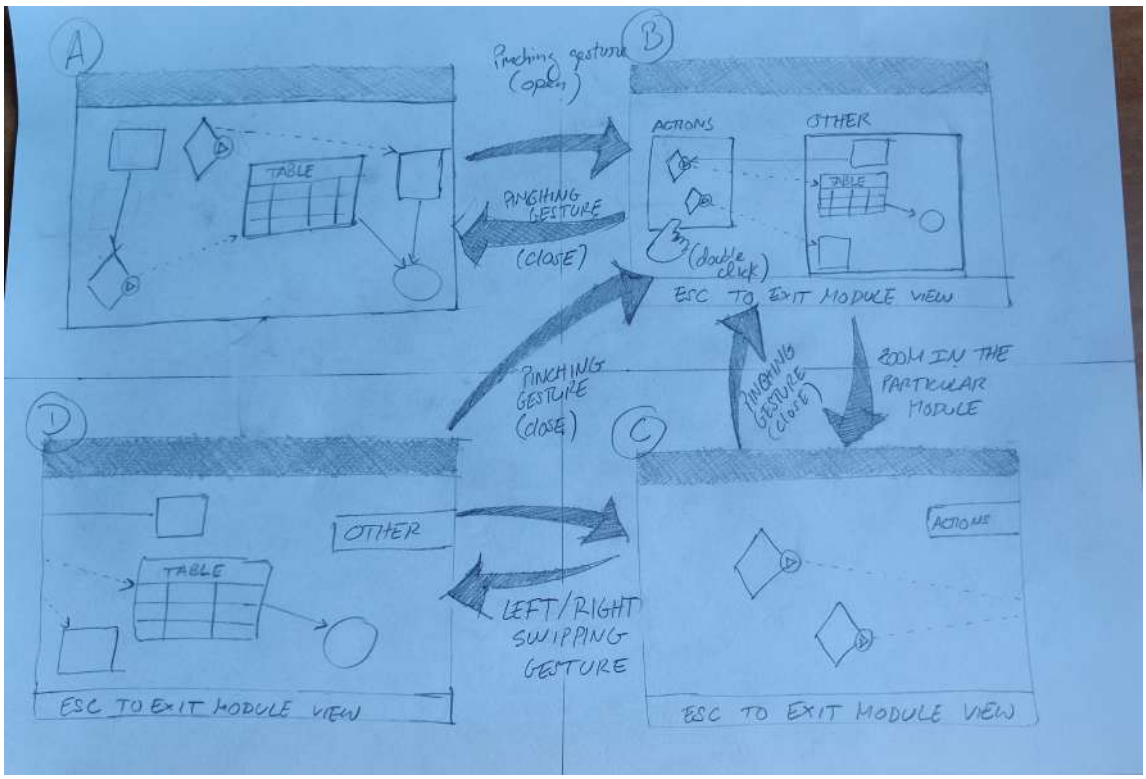


Figure 6.19: *Module view* sketching.

A convenient way of reorganizing the elements of a program is to group them by modules. The number of elements and relationships in the program may quickly escalate and the diagram may become overwhelmingly difficult to understand. Grouping elements by modules is convenient because it allows to abstract the view of the program. Figure 6.19 displays a sketched idea of a *module view*. The idea projected by this sequence of sketches implies the existence of two distinct view modes: a *global view* and a *module view*. The *global view* is shown in *sketch A* (top-left side), in which the system exhibits its lowest abstraction level. In the next step, the user produces a *pinching* gesture with his fingers (imagining a *touch-screen*), and, as a result, the diagram *zooms out* and the nodes are grouped into previously created modules (as it is possible to observe in *sketch B*, the nodes are separated into two distinct groups). At this point, the user is in *module view*, thus he has no visual information regarding the contents of each module, only about the way they relate themselves inside the system. The user then selects a given module to access its contents (the elements of that particular module are *zoomed in*). In *sketch C* (bottom-right side) the user views solely that particular set of elements (the relevant elements are displayed throughout the canvas while the remaining ones are hidden). The user can now produce *swiping* gestures to the right or left to navigate through the module collection (as shown in *sketch D*). The user can, as well, produce a single *pinching* gesture at any time to return to *module view* (*sketch B*) or two *pinching* gestures to return to *global view* (*sketch A*).

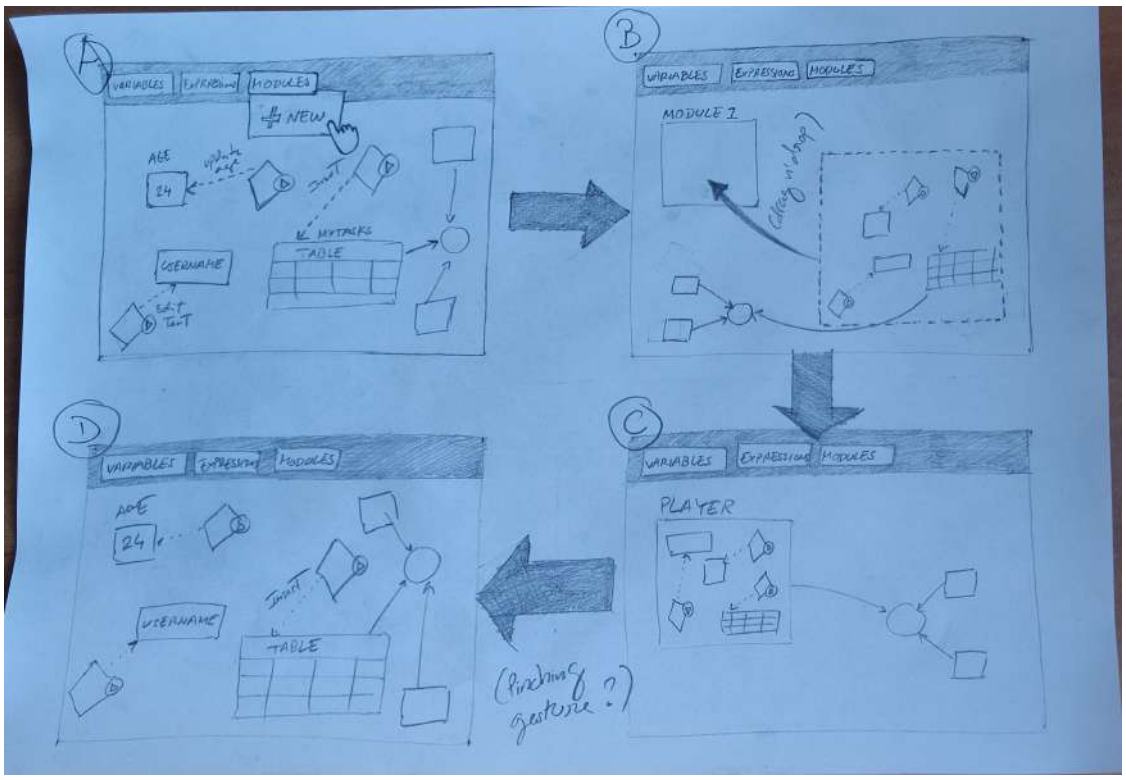


Figure 6.20: Addition of a module in *module view*.

The *module view* provides an efficient way of navigation and a good high-level view of the diagram. Furthermore, the user is intended to create his own modules and add elements to them (instead of using predefined modules). Therefore, it is important to define how are modules added to the system and how are elements associated to them. To complement the previous sketched interaction, figure 6.20 illustrates a step sequence through which a new module may integrate the system. Just as before, in *sketch A* the user has *global view* of the diagram. He selects the option *Modules* and then *New*. As a result, the system enters in *module view* (*sketch B*) and an empty box appears in the canvas (labeled with the default name, *Module1*). The user may now group a set of nodes (using his fingers or mouse) and associate them with the recently created module. At this point, he drags the whole set into the empty box. All the elements are now associated with a module (*sketch C*). Once again, the user may return to *global view* (*sketch D*) by producing a *pinching* gesture on the screen (or, for example, press the *Esc* button on a keyboard).

Module view offers users freedom to build their own library and organize the structure of the system in ways they may find appropriate. However, in certain situations, he may find useful to visualize the program in predefined patterns. For example, a very commonly used architectural pattern in software engineering is the *3-Tier* model. This model separates an application in three layers: the top-layer, or the *presentation tier*, that consists in the user interface (the way the information is displayed to the user); the middle-layer, or the *application tier*, that consists in the functional business logic; and

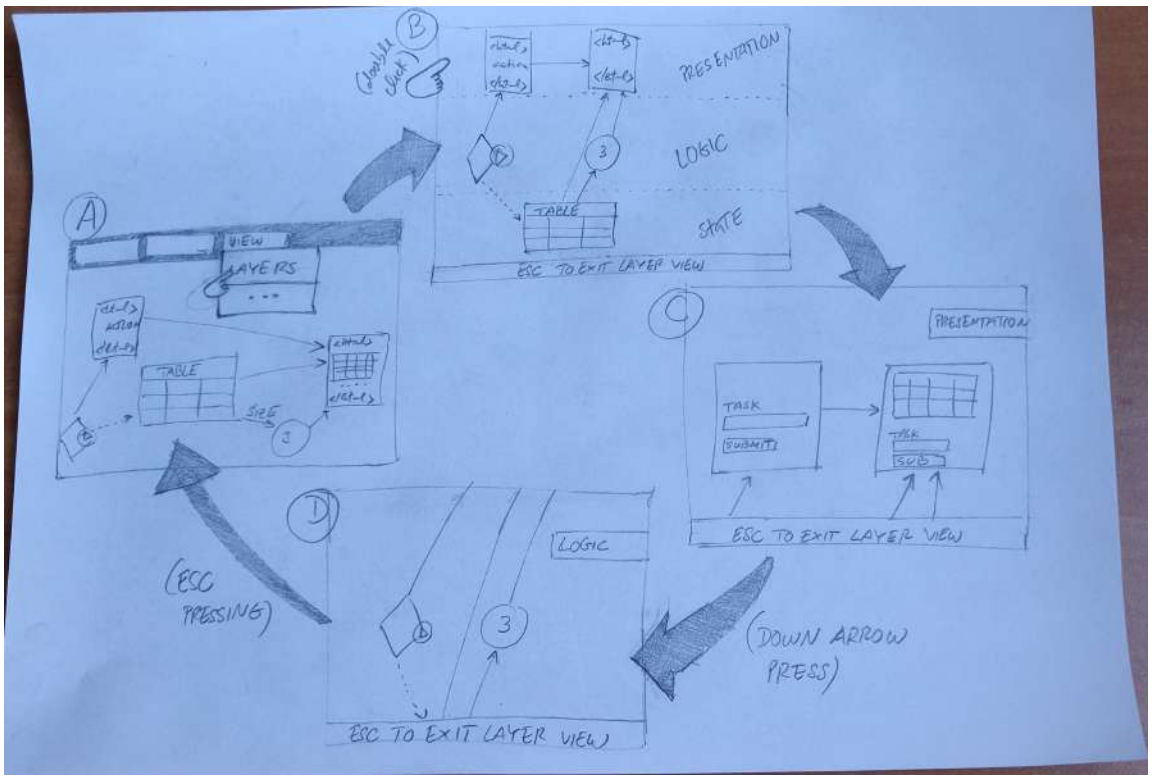


Figure 6.21: *3-Tier* View Mode.

in the bottom-layer, or the *data tier*, that consists in the state of the application. In our prototype, this would consist in reorganizing the programming elements throughout the canvas according to their semantic meaning: every state variable would be placed in the bottom-layer (representing the state of the application); *Html views* would be placed in the top-layer (representing the user interface); and, finally, the remaining elements (actions and the other logical expressions) would be placed in the middle-layer (representing the functional business logic). Figure 6.21 illustrates this idea. As before, the system is, initially, in *global view* (*sketch A*). The user selects the option *View*, (that opens a list of *view modes*), and then *Layers*. At this point, the system is in *3-tier view* (*sketch B*) and the user can select a layer to *zoom in* its contents. The user navigates to the *presentation tier* (in *sketch C*) and to the *application tier* (in *sketch D*). Just as in *module view*, the user can easily and quickly navigate through tiers, using *swiping* gestures (on a touchscreen) or arrow key pressing (on a keyboard). The user may return to *3-tier view* by doing a single *pinching* gesture and to *global view* by doing two *pinching* gestures. When considering architectural patterns, it is easy to think about other ways in which a program can be viewed (for example, the *Model-View-Controller* pattern). This approach may be useful to experienced developers, who are usually well familiarized with the use of these patterns.

Both *module view* and *3-Tier view* decompose the original diagram into smaller sub-diagrams that hold different parts of the application. These mechanisms help the design supporting the principle of *complexity management*, since the original complexity

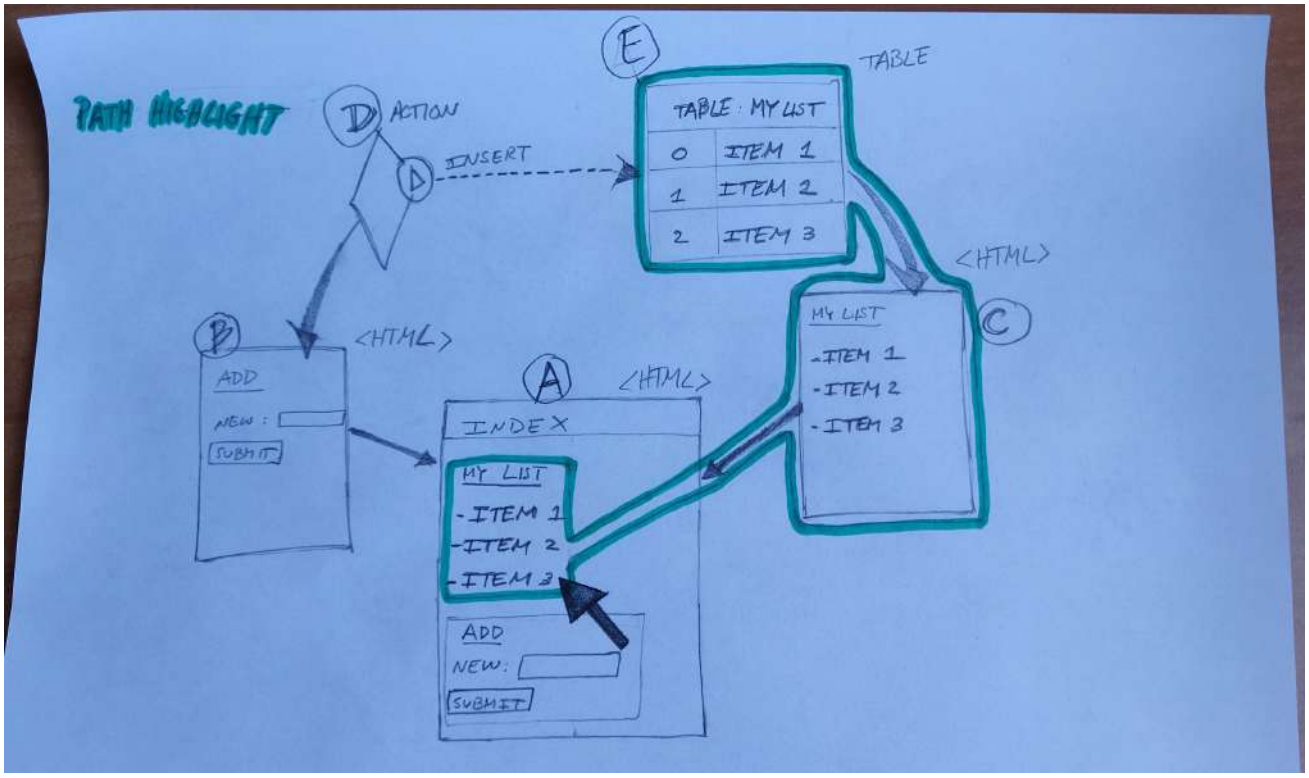


Figure 6.22: *Dependency Trail* sketch.

decreases.

Important features of these environments are debugging tools. While building an application, it is essential that developers understand how data pieces flow through components. This topic was discussed throughout research. In the context of our prototype, a feature such as this one, provides the user with a partial view of the diagram and helps him perceive the flux of data throughout a subset of elements in the program (similarly to *Circa*, described in chapter 4). This feature is named *dependency trail*. A *dependency trail* is a path in the diagram (a subset of dependencies) that helps the user understand what data and logic are involved in the rendering of a given *html* view. Figure 6.22 illustrates this idea. In the represented scenario, there is a *Table* (labeled with *E*) and an *Insert* action that stores items in it (labeled with *D*). There are three *html* views: the first one displays the stored items in a *html* list (labeled with *C*); the second one displays a *html* form to add new items (labeled with *B*); and the third one (labeled with *A*) joins both *C* and *B* in the same view. There is a clear path of dependencies between the *html* list in the main view (*A*) and the table of entries (*E*). When the user hovers the cursor over the *html* list in *A*, the *dependency trail* is highlighted, helping him find the data source of that particular component. Just like in *Circa*, this mechanism allows for a better understanding of information flows between elements by providing partial views of the diagram.

Chapter 7

Case Studies

In this chapter, we describe a set of case studies to provide a deeper understanding of the software construction process resorting to our graphical user interface.

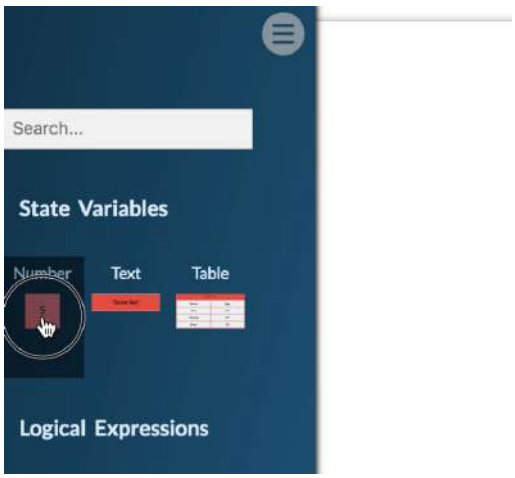
7.1 Counter

The first case study describes the construction process of a simple counter application. It consists of an integer variable which is possible to both increment and reset. Figure 7.1 illustrates the initial stage of the construction which consists of the addition of the variable *Counter* to the system. The user chooses *Number* from the left-side menu (fig. 7.1a) and assigns it the value 0 (fig. 7.1b).

The following stage of the process consists of adding an action to the system that increments *Counter* by one. First, users double-click *Counter* and select *Assignment* from the list of options (7.1c). Next, they set the new *Assignment*'s name to *incCounter* and its expression to *Counter := Counter + 1* (fig. 7.2a). Notice there is a *regular dependency* where the *source* is *Counter* and the *target* is *incCounter* (fig. 7.2b). This is because *incCounter* requires the current value of *Counter* to compute its next value. Figure 7.2c shows the state of the system after adding the action that resets *Counter* (*Counter := 0*).

The last stage of the process (figure 7.3) consists of adding an *html* view that displays the value of *Counter* and allows for the user to interact with the application. The user simply adds a new *html* view to the system (7.3a) and connects *Counter* and both actions to it (figures 7.3b and 7.3c). As result, all *html* representations are joined under the same view.

The previous construction process generates the code depicted in figure 7.4. As it is possible to see, the generated code is similar to the code of figure 2.1.



(a) First: user selects *Number* to add a new instance.

(b) Second: user assigns it the name *Counter* and an initial value of 0.

(c) Third: the variable is displayed on the canvas.

Figure 7.1: *Counter* application construction 1.

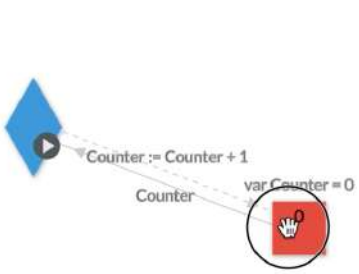
7.2 Task List

The case study that follows is slightly more complex than the previous one. It consists of a *html* view that displays a list of tasks and a form to add new tasks to it. Furthermore, the application must allow the user to delete tasks by id.

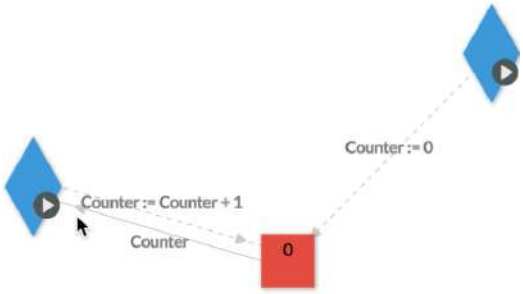
The first stage of the process is the definition of the state of the application (figure 7.5). We first add a *Table* variable to store tasks. Each entry of the table has a unique id (number) and a textual description of the task (string). We assign it the name *TaskList* (fig. 7.5a). Secondly, we must have a way of generating a unique id each time a new entry is added. For this, we add a *Number* element with the name *idGen*, followed by an action that increments its value (figures 7.5b and 7.5c).

We have successfully defined the state of the application. We must now define an action that inserts an entry in the table *TaskList*. Let us take a look at figure 7.6. As it

(a) First: user specifies the body of a new *Assignment* instance.



(b) Second: the *Assignment* to increment *Counter* is added.



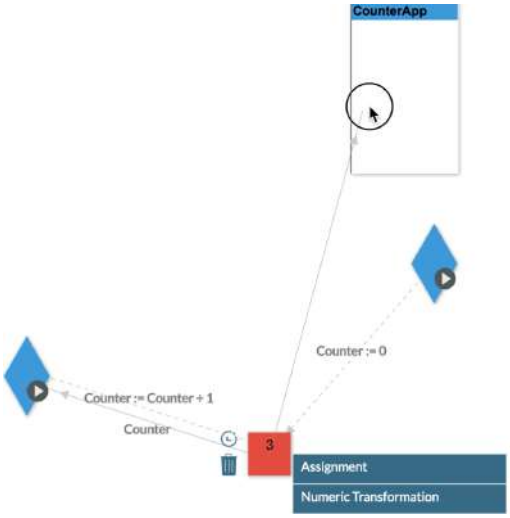
(c) Third: the *Assignment* to reset *Counter* is added.

Figure 7.2: *Counter* application construction 2.

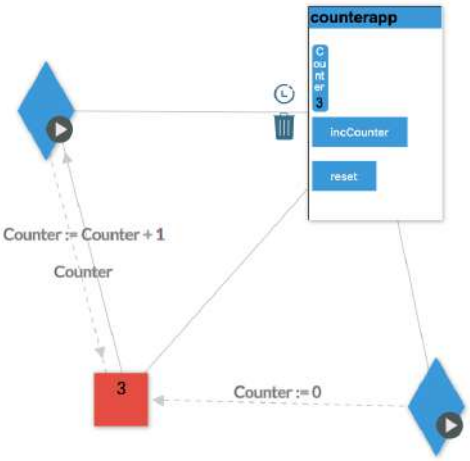
is possible to observe in figure 7.6a, we specify the id as *idGen* and the description as an input value, *inDesc* (the prefix *in* informs the parser this value must be provided by the user at runtime). After submitting the form, we can observe that the new insertion operation depends on the value of *idGen* to add new entries to the table. We now have an action that inserts an entry and another that increments *idGen*. However, we aim for these two actions to execute sequentially. With this purpose, we manually connect them together in order to create a single action that has the desired effects on the state of the system (fig. 7.6b). In figure 7.6c, we have two modification dependencies with same source (the new generated action).

The next step of the construction process consists of adding the feature to delete a task by id, as displayed in figure 7.7. After selecting *Delete* from the table's option list, the user provides a name for the action (for example, *removeTask*) and provides the expression *inId == TaskList.id* (figure 7.7a). Figure 7.7b shows the state of the system after adding the deletion operation.

(a) First: user selects *Html view* to add a new instance.



(b) Second: user manually connects *Counter* to the view.



(c) Third: user connects both actions to the view.

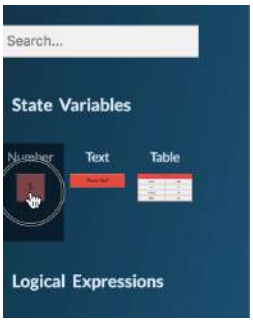
Figure 7.3: *Counter* application construction 3.

The last step is displayed in figure 7.8 and consists in the construction of the client-side so the end user can visualize the data and interact with the application. For each entry in our table, we must provide the user a button to delete it. To be able to attach an action to each entry of our task collection, the user creates a new *Map* (fig. 7.8a) and he sets three attributes: the id of the task, *TaskList.id*; the description of the task, *TaskList.desc*; and an action that removes it, *removeTask TaskList.id* (the name of the deletion operation followed by the argument). After submitting the form, the new *Map* (named *displayTasks*) is added to the diagram (figure 7.8b). Lastly, the user adds the *html* view to the diagram and connects the *Map* and the insertion operation to it (fig. 7.8c). This results in the render of *displayTasks* (that allows to visualize data and remove entries) and the form (that allows to add new tasks). The described process generates the code of figure 7.9.

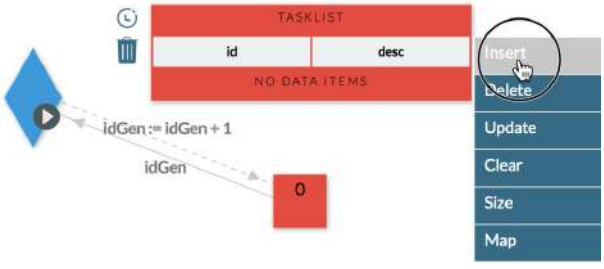
```
1     var Counter = 0
2     def incCounter = action {Counter := Counter + 1}
3     def reset = action {Counter := 0}
4
5     def counterapp =
6         <div class="template1" id="counterapp">
7             <h3 class="main-title">"counterapp"</h3>
8             <div class="single-value" id="Counter">
9                 <label>"Counter"</label>
10                <div>Counter</div>
11            </div>
12            <div class="action-form">
13                <form>
14                    <button class="do-action-button"
15                        doaction=( action { Counter := Counter + 1 } )>
16                            "incCounter"
17                    </button>
18                </form>
19            </div>
20            <div class="action-form">
21                <form>
22                    <button class="do-action-button"
23                        doaction=( action { Counter := 0 } )> "reset"
24                    </button>
25                </form>
26            </div>
27        </div>
```

Figure 7.4: *Counter* generated program.

(a) First: user specifies the structure of the new *Table*.



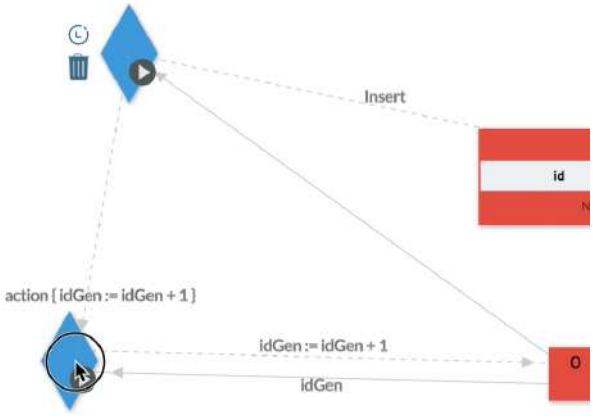
(b) Second: user selects *Number* to generate unique ids.



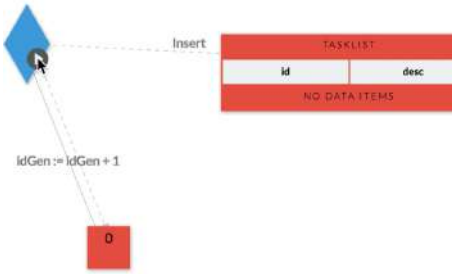
(c) Third: User adds an action to increment *idGen*.

Figure 7.5: *Task List* application construction 1.

(a) First: user specifies the structure of the *Insert*.



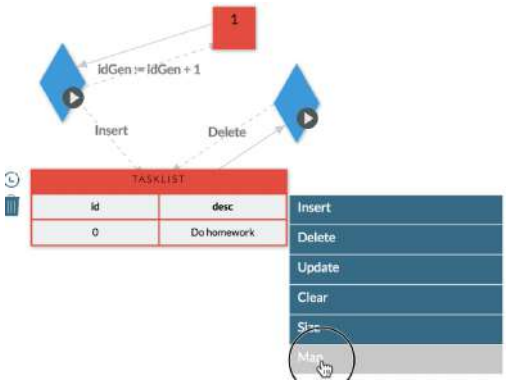
(b) Second: user manually connects together the insert and increment actions.



(c) Third: the composed action is generated and rendered in the canvas.

Figure 7.6: *Task List* application construction 2.

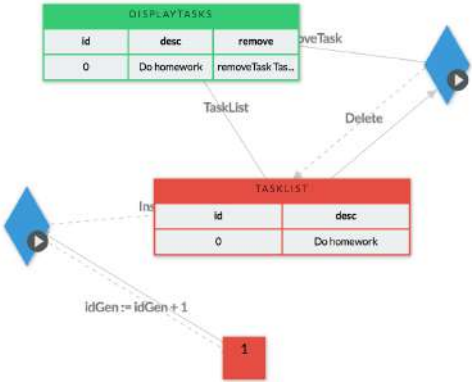
(a) First: user specifies the structure of the *Delete*.



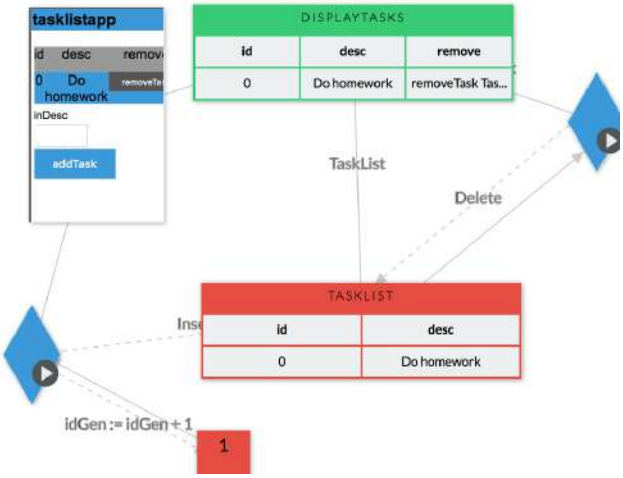
(b) Second: state of the system after adding the deletion operation.

Figure 7.7: *Task List* application construction 3.

(a) First: user specifies the structure of the new *Map*.



(b) Second: the *Map*, *displayTasks*, is successfully rendered.



(c) Third: The *Html view* is added to the diagram.

Figure 7.8: *Task List* application construction 4.

```
table TaskList { id: number, desc: string }

var idGen = 0

def addTask = (inDesc) => action {
    insert {id: idGen, desc: inDesc} into TaskList,
    idGen := idGen + 1 }

def removeTask = (inId) => action {
    delete r in TaskList where inId == r.id }

def displayTasks = map(r in TaskList)
    {id: r.id, desc: r.desc, remove: removeTask(r.id)}

def tasklistapp =
    <div class="template1" id="tasklistapp">
        <h3 class="main-title">"tasklistapp"</h3>
        <div id="displayTasks" class="collection">
            <div class="table">
                <div class="table-header">
                    <div class="attribute-label">"id"</div>
                    <div class="attribute-label">"desc"</div>
                    <div class="attribute-label">"remove"</div>
                </div>
                <div class="table-body">
                    (map (e in displayTasks)
                    <div class="item">
                        <div class="attribute">(e.id)</div>
                        <div class="attribute">(e.desc)</div>
                        <div class="attribute">
                            <button class="table-button"
                                doaction = (e.remove)>
                                    "removeTask"
                            </button>
                        </div>
                    </div>)
                </div>
            </div>
        </div>
        <div class="action-form">
            <form>
                <div class="in">
                    <label for="inDesc">"inDesc"</label>
                    <input id="inDesc" type="string"/>
                </div>
                <button class="do-action-button" doaction=( action {
                    insert {id: idGen, desc: #inDesc} into TaskList,
                    idGen := idGen + 1 })>
                        "addTask"
                </button>
            </form>
        </div>
    </div>
```

Figure 7.9: *Task List* generated program.

Chapter 8

Evaluation

In this chapter, we describe the methodology used for evaluating our computer prototype and we present the results gathered during evaluation sessions and from user experience questionnaires.

8.1 Objectives

As mentioned in section 1.2, the main motivation behind the idealization and conception of our visual construction model, is to substantially increase the developers' efficiency when building web applications, in comparison to the amount of time it would take to actually program the same application using the *Live Programming*'s original text editor. In addition to this, the development is intended to occur through a simple user interface, designed to decrease the amount of prior information the user would need in order to interact with the system.

Considering this, we aim our evaluation process towards measurement of efficiency and effectiveness of use, usefulness, innovation, learnability and user satisfaction.

8.2 Participants and Evaluation Method

Initially, we had to consider strategies for our experimental evaluation. For our purposes, we followed a *formative evaluation* [25] (described in section 3.3), in which a subset of target users is chosen to interact with the prototype.

Eight participants took part in the experimental evaluation of our prototype. The participants, all members of the Informatics Department of FCT-UNL, are of different genders (seven male participants and one female participant) and aged between 22 and 28 years old (an age average of approximately 25 years old). All of the participants were

users of type A (all of them have programming skills and have, at least, basic notions of web development, as described in section 1.3). In the beginning of each test, we briefly described the purpose of the work to the participant. After, we described each one of the activities:

- Activity 1: the participant is asked to build a simple *Task List* application with a well defined set of features. To complete the activity, participants have at their disposal a step-by-step guide to help them. In this activity, we evaluate the user's performance by measuring the time required for constructing each feature. The sequence of steps is the following:

  1. Create a *Table* variable to store tasks (with the name *TaskList*, with an id, *id*, and description, *desc*)
  2. Add a *Size* to keep track of the number of entries in the table (with the name *taskListSize*)
  3. Add an *Insert* action to add entries to the table (with the name *addTask*, *id* set to *taskListSize* and *desc* set to *inDesc*)
  4. Add a *Delete* action to remove entries by id (with the name *removeTask* and a condition *TaskList.id* == *inId*)
  5. Add a *Map* expression to display tasks (with the name *DisplayTasks*, *id* set to *TaskList.id*, *desc* set to *TaskList.desc* and *remove* set to *removeTask TaskList.id*)
  6. Create a *Html View* to interact with the application
  7. Manually connect both the *DisplayTasks* and *addTask* to the new view

- Activity 2: the participant must build a *Counter* application without any support, that is, users must rely on what they were able to learn in the previous activity.

Activity 1 attempts to measure the effectiveness of use and efficiency on the completion of each proposed step while *Activity 2* attempts to measure the learnability of the interface (if the user understands the basic concepts of the system and can find his way around the interface after the first interaction).

After the completion of both activities, we asked the participants to respond to a short usability questionnaire, in order to gather some insights on the ease of use, usefulness, innovation and overall satisfaction.

8.3 Results

We now present the results gathered throughout the evaluation sessions. In the context of our evaluation results, there is an initial aspect we consider important to mention: the most accurate way to measure our results would be to compare the amount of time

Tasks / Participants	P1	P2	P3	P4	P5	P6	P7	P8
Task 1	17.89	114.94	20.21	20.71	32.61	44.22	13.25	11.49
Task 2	35.04	22.46	59.07	5.22	51.43	17.14	12.47	22.67
Task 3	58.00	56.64	153.43	22.26	131.19	112.22	99.08	50.27
Task 4	50.00	42.64	47.01	29.45	33.33	28.12	35.98	52.51
Task 5	35.64	106.60	128.46	42.53	120.84	100.58	131.02	107.24
Task 6	10.73	13.60	11.36	18.96	10.21	27.56	12.02	10.87
Task 7	25.67	35.85	109.51	105.22	45.36	101.37	22.99	25.09
Total (in sec)	232.97	392.73	529.05	244.35	424.97	431.21	326.81	280.14
Total (in min)	≈ 4 min	≈ 7 min	≈ 9min	≈ 4 min	≈ 7 min	≈ 7 min	≈ 5 min	≈ 5 min

Table 8.1: Measured times for *Activity 1*

used to implement an application through the *Live Programming* original textual editor with the time used to build the same application through our graphical environment. For this to be accomplished, the participant would actually need to implement the full application during the session. The first problem with this approach would be the large amount of time required from each participant. Furthermore, in order to gather accurate results, each participant would have to know how to program using the prototype's language and editor (know the concept of the system well and be comfortable with the language's specific syntax). This would be practically impossible to accomplish given the time constraints and the availability of each participant.

As a means to circumvent this issue in the best possible way, we measured the time for the construction of each feature through our environment during *Activity 1*, in order to get a closer approximation in terms of increased efficiency/productivity, in comparison to the amount of generated code by our system.

After the completion of each activity, the generated code was displayed to each participant. After a brief analysis of the code, they were asked if our system had the potential to highly increase their efficiency and productivity throughout the task of web development, in comparison to traditional programming. 100% of participants answered positively. Two participants mentioned it would be quite of an useful tool for them, because it abstracts the hurdles of web development. Table 8.1 provides the measured times (in seconds) for each one of the tasks of *Activity 1*, according to the participant.

As it is possible to observe in table 8.1, the times measured for each participant tend to have a small deviation from each other when considering a particular task. In rare situations, however, some values may present very high deviations from the norm. This is because some participants went for what was more intuitive for them in given situations, instead of following the step-by-step guide given at the beginning.

The last row of the table provides the amount of time (in minutes) it took to each participant to build the whole application and successfully finish the activity (an average of 6 minutes, approximately). In terms of *effectiveness*, the results were clearly positive (in both activities), since all participants were able to successfully complete each proposed task (completion rate of 100%). Furthermore, we did not observe any severe interaction problem throughout the tests. Let us consider the following: our visual language, as

any other language, contains inner concepts and constructs that must be previously understood in order to efficiently use it for specific purposes (for example, programming elements, relationships and semantics constructs). When learning a new programming language (such as Java or Python), documentation is extremely useful (one participant even suggested that the language would hugely benefit from documentation). Considering this, plus the fact of being the very first time for each participant, we consider these results to be extremely positive.

At the end of each test, the participant was asked to respond to a usability questionnaire as means to provide his/her input on the system they had just used. The questions and gathered statistics are shown in figure 8.1. Most of the answers are rated resorting to the *Likert* scale, where 1 stands for "strongly disagree"and 5 stands for "strongly agree". The questions are mainly directed towards the measurement of the complexity, usefulness, ease of use, efficiency and overall satisfaction. In terms of complexity, half of participants respond the system is very little complex and that requires an average amount of prior information in order to be used. More than half of participants (a percentage of 62.5%), state the elements of the interface are very well organized and that it is easy to learn the system. In general terms, all the participants agreed the system is quite easy to use (50% of participants assigned the grade 4 as the other half assigned the grade 5). Participants state that there is a high increase in terms of efficiency in comparison to traditional coding (75% assigned it a grade 4 as 25% assigned it a grade 5). All participants think our system represents an innovative alternative to web development. To conclude, all participants were generally satisfied with our system (grade 4 by 62.5% and grade 5 by 37.5% of the population).

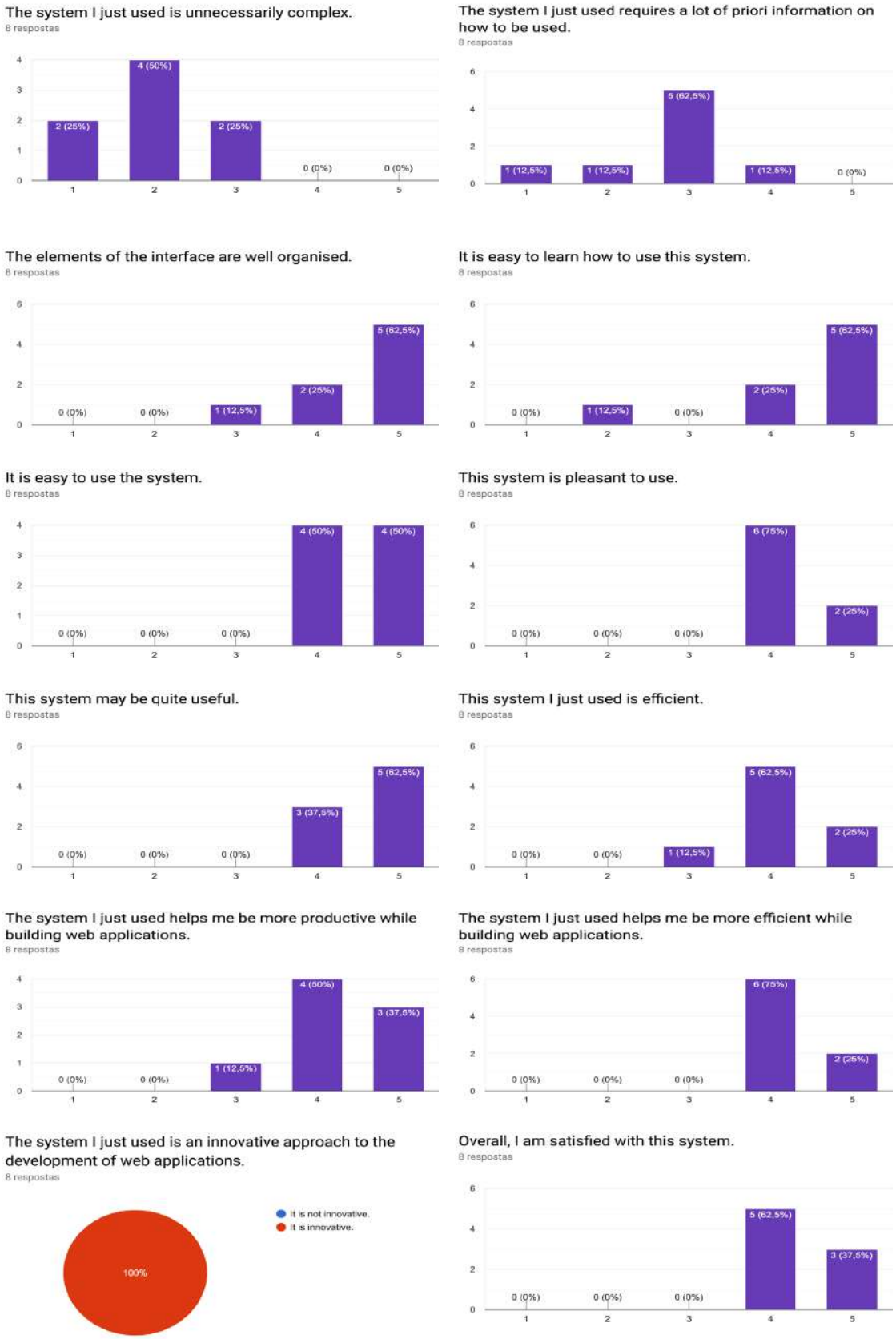


Figure 8.1: Usability Questionnaire

Chapter 9

Conclusion

We now present the final remarks and future developments of our dissertation.

9.1 Final Remarks

9.1.1 Contraints and Limitations

When presented, the project consisted of a very high-level overview of what to produce and deliver. There was an initial struggle to refine a concrete set of features and functionalities the prototype should offer its users and what it should look like. We produced several sketches of possible interactions at an early development stage, but those sketches were not added to this document due to time and space constraints; most sketches are not trivial to understand without a proper explanation.

From a technical perspective, due to the limitations of the *D3* javascript framework, it was not possible to implement the *view modes* described in chapter 6. The main objective of *D3* is to provide a good data visualization framework, and, therefore, when it comes to interacting with diagrams, *D3* has some limitations and its API is poorly documented.

9.1.2 Conclusion

As originally intended, our tool is able to generate full programs through the manipulation of visual elements on the screen, without relying in computer code. It allows to quickly build a functional web application and interact with it. Our visual construction model is implemented to easily integrate new programming elements, which makes it easier to scale in the future. Furthermore, we conceived and sketched a set of interaction techniques to allow for convenient visual representations of programs and their construction processes.

Regarding the *Focus Group*, the information we gathered throughout the meeting was not the one that was originally intended. Despite that, it was useful to gather the first external feedback on our computational prototype and to document other interaction ideas.

The results gathered throughout evaluation and the provided feedback were very positive, especially in terms of usefulness and efficiency, as it was originally intended. Participants mention the prototype, even covering a small fraction of the *Live Programming* features, is already very useful, as it enables saving a huge amount of time when developing web applications. Participants also say that it has an enormous potential to grow and manage larger and more complex applications. Furthermore, they mention the interaction does not present a high level of complexity when considering it can generate a huge amount of code. To conclude, we were able to observe a natural curiosity and excitement from participants while testing the prototype.

In spite of the appearance of some obstacles that hindered the development process, we consider that we achieved the main goal of this thesis. Furthermore, we consider that this project has an enormous potential to grow and we know we have offered the first contribution to an extremely useful and powerful software.

9.2 Future Work

Considering the computer prototype we developed, the first future development is allowing the user to interact with the system through a *touchscreen*. Providing a touchable interface allows the user add new elements and form new relationships much more intuitively, and therefore, granting the user a better experience.

A second future development is the implementation of *view modes*, described and sketched in Chapter 6. These provide mechanisms to effortlessly navigate throughout the diagram and reorganize nodes in such ways the user can easily perceive the program's structure. We reach an interesting result if we combine *view modes* with a touchable user interface. As an example, it is interesting to imagine a scenario where the user navigates throughout the modules of a web application with *swiping* gestures or throughout the abstraction views of the system with *pinching* gestures.

Another interesting feature to integrate with the current prototype is a *Html* editor. It is important to provide the user with the freedom of manipulating the *Dom* elements as he may find appropriate. Furthermore, the editor should provide a set of *css* templates to change the styles of the views.

Lastly, we consider it important to extend the visual language with new symbols and rules. To each existing element, there may be other possible transformations (for example, the system may integrate an operation that creates a sequence of integers that follows a specified number variable).

Online resources

[1] *D3.js - Data-Driven Documents*. Accessed: 2019-12-13. URL: https://d3js.org/.

[2] M. Domingues. *About the Prototype - Live Programming*. Accessed: 2019-12-13. URL: http://live-programming.herokuapp.com/about.

[3] *Eagre Reader*. Accessed: 2019-12-13. URL: https://eagrereader.appspot.com/.

[4] *Free prototyping tool for web and mobile apps*. Accessed: 2019-12-13. URL: https://www.justinmind.com/.

[5] C. Granger. *Light Table*. Accessed: 2019-12-13. URL: http://lighttable.com/.

[6] O. Hamann. *Eagle Mode*. Accessed: 2019-12-13. URL: http://eaglemode.sourceforge.net/emvideo.html.

[7] O. Hamann. *Zooming user interface*. Accessed: 2019-12-13. URL: https://en.wikipedia.org/wiki/Zooming_user_interface.

[8] *Introducing Mercury OS*. Accessed: 2019-12-13. URL: https://uxdesign.cc/introducing-mercury-os-f4de45a04289.

[9] *ISO 9241-11:2018(en)*. Accessed: 2019-12-13. URL: https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:ed-2:v1:en.

[10] lennypitt. *Etoys tutorial 1*. Accessed: 2019-12-13. URL: https://www.youtube.com/watch?v=0h_2OTVzQZk&t=12s.

[11] *Low-code Application Development Platform - Build Apps Fast and Efficiently Mendix*. Accessed: 2019-12-13. URL: https://www.mendix.com/.

[12] *Low-code development platform*. Accessed: 2019-12-13. URL: https://en.wikipedia.org/wiki/Low-code_development_platform.

[13] *Microsoft-Excel*. Accessed: 2019-12-13. URL: https://en.wikipedia.org/wiki/Microsoft_Excel.

[14] NOVA-LINCS. *CLAY*. Accessed: 2019-12-13. URL: http://ctp.di.fct.unl.pt/CLAY/.

[15] *Presentation Software - Online Presentation Tools - Prezi*. Accessed: 2019-12-13. URL: http://bit.ly/36IQGaL.

[16] *Reactive programming*. Accessed: 2019-12-13. URL: https://en.wikipedia.org/wiki/Reactive_programming.

[17] *Swift Playgrounds*. Accessed: 2019-12-13. URL: https://www.apple.com/swift/playgrounds/.

[18] *System usability scale*. Accessed: 2019-12-13. URL: https://en.wikipedia.org/wiki/System_usability_scale.

[19] *TouchDevelop*. Accessed: 2019-12-13. URL: https://www.microsoft.com/en-us/research/project/touchdevelop/.

[20] *Website wireframes: Mockingbird*. Accessed: 2019-12-13. URL: https://gomockingbird.com/home.

[21] *What is a Citizen Developer*. Accessed: 2019-12-13. URL: https://www.techopedia.com/definition/30968/citizen-developer.

[22] *What is Low-Code?* Accessed: 2019-12-13. URL: https://www.outsystems.com/blog/what-is-low-code.html.

Bibliography

[23] 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: ACM, 2007, pp. 101–120. ISBN: 978-1-60558-084-5. DOI: 10.1145/1352678.1352685. URL: http://doi.acm.org/10.1145/1352678.1352685.

[24] B. Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design*. First. Morgan Kaufmann, 2007.

[25] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-computer Interaction*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN: 0-13-437211-5.

[26] *Type Safe Evolution of Live Systems*. Pittsburgh, 2015.

[27] M. Domingues and J. C. Seco. *Type-Safe Evolution of Data-Centric Applications*. Tech. rep. Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 2014.

[28] M. B. Domingues and J. C. Seco. "LiveWeb - Core Language for Web Applications." In: *Proceedings of InForum 2010*. Universidade do Minho, Sept. 2010.

[29] 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.

[30] A. Goldberg. *SMALLTALK-80: The Interactive Programming Environment*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN: 0-201-11372-4.

[31] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. "Advances in Dataflow Programming Languages." In: *ACM Comput. Surv.* 36.1 (Mar. 2004), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/1013208.1013209. URL: http://doi.acm.org/10.1145/1013208.1013209.

[32] J. Mateus, M. Domingues, and J. Costa Seco. "Sistema de Runtime para uma Linguagem Web Reativa." English. In: *INForum 2015 - Actas do 7º Simpósio de Informática*. Sem PDF. 2015.

[33] S. McDirmid. "Usable Live Programming." In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 53–62. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509585. URL: http://doi.acm.org/10.1145/2509578.2509585.

[34] I. Messias. *Graphical user interface for a functional live programming environment*. Tech. rep. 2018.

[35] D. Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering." In: *IEEE Trans. Softw. Eng.* 35.6 (Nov. 2009), pp. 756–779. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.67. URL: https://doi.org/10.1109/TSE.2009.67.

[36] J. Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 0125184050.

[37] D. A. Norman. *The Design of Everyday Things*. New York, NY, USA: Basic Books, Inc., 2002. ISBN: 9780465067107.

[38] J. Preece, Y. Rogers, and H. Sharp. *Interaction Design: Beyond Human-Computer Interaction*. 4th ed. Hoboken, NJ: Wiley, 2015. ISBN: 978-1-119-02075-2.

[39] W. W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques." In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. Monterey, California, USA: IEEE Computer Society Press, 1987, pp. 328–338. ISBN: 0-89791-216-0. URL: http://dl.acm.org/citation.cfm?id=41765.41801.

[40] S. Sachdeva. "Agile Methodologies." In: ().

[41] B. Shneiderman. "Human-computer Interaction." In: ed. by R. M. Baecker and W. A. S. Buxton. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987. Chap. Direct Manipulation: A Step Beyond Programming Languages, pp. 461–467. ISBN: 0-934613-24-9. URL: http://dl.acm.org/citation.cfm?id=58076.58115.