NTNU
Norwegian University of
Science and Technology

Easing the Transition from Visual to Textual Programming

Håkon Gimnes Kaurel

Master of Science in Informatics
Submission date: December 2016
Supervisor: Hallvard Trætteberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Dedication

Throughout this project I have been faced with countless challenges, both technical and non-technical. I owe a large debt of gratitude to the people who have helped me overcome these challenges, and ideally I would name them all. Unfortunately this is not feasible, and I will have to name but a few of those who have helped me along the way.

First and foremost I am very grateful for the support offered by my supervisor, Hallvard Trætteberg. From the very beginning he has shown an exceptional interest in the project and a willingness to help that by far exceeded my expectations. Every questions posed to him has been responded to with a thorough and insightful response. Having him as a partner in discussions about everything from major design decision to minor technical intricacies has helped me overcome and even anticipate challenges, as well as developing my skills within the field of software development.

The second person I would like to thank is my friend and co-counder of Kodeklubben Trondheim, Lars Klingenberg. He has time and time again proved himself to be the the man to go to when there is help required and work to be done. During this project he helped me during the data collection process. I could not have asked for a better partner in this, nor any other of the countless projects that we have worked together on.

I would also like to thank my close friend, Elias Inderhaug, for his support throughout this last year. There are few people who have been more vital to keeping me motivated and enthusiastic about my work than him. In addition, he has helped me proofread large parts of this document, for which I am very grateful.

Last, but not least, I would like to thank my parents, Petter Kaurel and Berit Kaurel. During this last year, and the ones preceding it, they have provided me with unwavering support.

Abstract

In later years an effort to teach programming to children has been gaining traction. Programming is seen as an important skill for the future, and many countries are seeking to ensure that children are given the opportunity to learn programming at an early age. Visual programming languages are often used as a first introduction to programming, as they have proven to be able of lowering the bar of entry to programming. However, experience shows that moving from visual programming to textual programming is difficult. This thesis explores how this transition can be eased by creating a new programming environment specifically tailored for this purpose. Through the project a prototype system has been developed and evaluated. The system enables the user to write two-dimensional games in a domain specific language, and choosing between a visual and a textual syntax. The prototype seeks to isolate the transition from visual to textual programming, by ensuring that it can be made without worrying about semantic differences. The results indicate that this is a viable approach to solving the problem, however, further research is required.

Sammendrag

I senere år har det å lære bort programmering til barn vokst frem til å bli et viktig tema. Programmering blir regnet som en viktig ferdighet i fremtiden, og mange land ønsker å sørge for at barn blir gitt muligheten til å lære programmering i ung alder. Visuelle programmeringsspråk er ofte brukt som en første introduksjon til programmering, ettersom de har vist seg å gjøre det lettere å komme i gang for nybegynnere. Men erfaring viser at det å gå fra visuelle til tekstlige programmeringsspråk er vanskelig. Denne oppgaven utforsker hvordan denne overgangen kan gjøres enklere ved å lage et nytt utviklingsmiljø som er skreddersydd for dette formålet. I løpet av prosjektet har en prototype blitt utviklet og evaluert. Prototypen gjør det mulig å utvikle todimensjonale spill i et domenespesifikt programmeringsspråk, og velge mellom å bruke en visuell og en tekstlig syntaks. Prototypen er forsøker å isolere overgangen fra visuell til tekstlig programmering for å sørge for at den kan bli gjennomført uten at brukeren trenger å bry seg om semantiske forskjeller. Resultatene indikerer at dette er en god måte å løse problemet på, men mer forskning må til før noen definitive svar kan gis.

Table of Contents

List of Figures

Abbreviations

DSL	=	Domain Specific Language
GPL	=	General Purpose Language

Chapter 1

Introduction

1.1 Motivation

Information technology (IT) has transformed and will continue to transform many aspects of modern society. Personal computers have become an integral part of the public, private and professional lives of billions of people. We have become dependent on computers, not only to acquire information, but also in how we perform our jobs, how we manage our private and global economy, and even to the point of how we communicate with other people. So essential are computers in today's society that the ability to use computers has become a requisite to become a part of society. Look at British sociologist Thomas Humphrey Marhshall's definition, and one quickly realizes why information technology (IT) has assumed a secure place in what is considered today's civilized life. Marshall defines citizenship as a status that is bestowed on those who are full members of a community. As such, citizenship includes certain civil, political and social rights of membership, including "the right to share to the full in the social heritage and to live the life of a civilized being according to the standards prevailing in the society" (Marshall, 1950).

The term "digital citizenship" has thus been defined as the ability to participate in society online (Mossberger et al., 2007). According to the EU Commission, 90% of European jobs in careers such as engineering, accountancy, nursing, medicine, art, architecture, and

many more will require some level of digital skills. Every citizen needs to have at least basic digital skills in order to live, work, learn and participate in society (Commission, 2016). Every major technological innovation brings with it a demand for new skills, which is why Europe is faced with a paradox. Although millions of Europeans are currently without a job, companies have a hard time finding skilled digital technology experts. As a result, there could be up to 825,000 unfilled vacancies for ICT (Information and Communications technology) professionals by 2020 (Commission, 2016). Countries around the world are facing the same challenges related to an increasingly digital society, and many are working to make sure that digital skills find their place in their schools' curriculum. A digital skill is a broad term that covers just about any task that can be completed using a computer, from browsing the web to creating your own software. Deciding which of these skills are important in terms of participation in society is an important political issue in many countries. Should children and students be taught simply how to use digital technology, or should they understand how that technology works? Several countries around the world have started integrating computer programming into their education at lower and middle schools.

Computer programming is the activity of creating new software, and is closely related to understanding how a computer works. In 2013 I co-founded an organization called Kodeklubben Trondheim (The Codeclub Trondheim). Kodeklubben Trondheim provides free programming courses as an after school activity. The organization's vision is that every Norwegian child should be given the opportunity to try programming, and be given guidance when they struggle to master it.

1.2 Problem description

In response to the increasing demand for skilled software developers a plethora of new tools for teaching programming to children has emerged. There are new game development libraries, new programming languages, and new development environments being created. The tools aim to lower the bar of entry to programming, making it more accessible to children.

Many of the new programming languages replace the textual notation of traditional

programming languages with a visual one. These are referred to as visual programming languages (VPL). VPLs have the advantage that they are able to eliminate many mistakes that are commonly made by novice programmers.

However, visual programming languages are not nearly as frequently applied the software industry in general. There are some inherent disadvantages and limitations associated with visual notations as opposed to textual ones, which leads to the majority of software development being done using textual languages. There is therefore little indication that visual programming will become a large factor in professional software development any time soon. In other words, children who master visual programming will need to make a transition to textual programming at some point if they are ever to become professional developers.

Through teaching programming to children I have experienced that many children struggle to make the transition from visual to textual programming. Through this thesis I try to explore how this transition can be eased.

1.2.1 Kodeklubben Experiences

Scratch is one of the most successful tools for getting children started with programming. Scratch is a complete development environment created specifically for children. It consists of three parts, a visual programming language, a game development framework, and an integrated development environment (IDE).

Kodeklubben Trondheim has utilized Scratch as its entry level programming language since the beginning. The Scratch courses have been rather successful and compared to other courses, few of the participants drop out of the Scratch courses during a semester. The participants of our Scratch courses are usually able to master the language to the degree that they are able to solve simple tasks using the language. In summary, we are quite pleased with our Scratch courses.

Python is a very popular textual programming language. It is a very flexible language with a simple and concise syntax. In contrast to Scratch, Python is not specifically created for educational purposes. It is a general purpose programming language that is used both in academic circles and in the software industry.

Children who have completed the Scratch courses at Kodeklubben Trondheim have been recommended to try our Python courses. Despite Python being a relatively simple language, our Python courses have not been as successful as intended. The participants struggle to understand the concepts of the language, as well as the textual syntax. This affects the experience of programming, and a lot more participants drop out of the Python courses during a semester than the Scratch courses.

This transition from Scratch to Python will be used as the context throughout this thesis, as it is a concrete instance of the transition from visual to textual programming.

1.3 Objective and research questions

The research objective for this project is to explore how a development environment where visual programming is combined with textual programming should be designed in order to ease the transition from one to the other.

Research questions

1. How can a textual and a visual programming language be combined to ease the transition from visual to textual programming?

2. How can a development environment aid the transition from visual to textual programming?

The first research question is used to explore possible approaches to integrate visual and textual programming. There are many different approaches that could be taken. The VPL could be used simply to generate textual code, or the other way around. Another approach is to create a hybrid language that allows the users to freely combine visual and textual notation. It is also possible to create separate languages and allow automatic translation between the two.

The second research questions is used to explore additional tools that can be used to aid the transition from visual to textual programming. There are several development environments aimed at teaching programming to children, and their design decisions are taken into

consideration. However, there are special considerations to be made when constructing an environment suitable for the purposes of this thesis.

Chapter 2

Background

This thesis touches upon several important aspects of computer science. This chapter presents previous work that is deemed as relevant to this project, and explains some of the terminology used in the rest of this document.

2.1 Programming Languages

Language is the ability to acquire and use complex systems of communication. As opposed to human communication, programming is a simple form of communication between a computer and a programmer. Because the communication between the computer and a programmer is more rudimentary than the complex human communication, programming utilizes specialized languages called programming languages. Compared to natural languages programming languages are simple, strict, and unambiguous. Their only purpose is to provide a means of issuing instructions to computers. However, not all programming languages can be executed by computers directly. The subset of languages that can be directly interpreted and executed by computers are referred to as machine languages or machine code.

Machine languages are defined exclusively in terms of binary numbers. Machine language instructions tend to share a similar and simple structure. There is a binary number, called the opcode, that identifies the type of instruction, such as an addition or subtraction.

The opcode is proceeded by some additional binary numbers that are the parameters of the instruction, such as which numbers to add and where to store the results. These languages are generally optimized from the perspective of the computer, and not the programmer. Computers can easily interpret and execute binary code, but writing binary code is for most programmers unintuitive, tedious and error-prone. Programmers rarely create software by writing machine code, instead they rely on languages that can be automatically translated into machine code. These languages exist primarily to ease the human activity of writing software. The expressive power of a language A that can be translated into a language B cannot be greater than the expressive power of B. This means that programming languages that are translated into machine code, cannot express anything that cannot be expressed in machine code directly. Therefore, these languages do not provide any additional expressive power. Many of them are less expressive, and a select few are equally expressive as machine code. The majority of programming languages provide useful abstractions that help humans express their instructions in a more precise and less error-prone way.

2.1.1 Syntax

Every language, whether it be a programming or a natural language, has a syntax. Syntax defines the rules for what constitutes a legal sentence in a given language. Generally, syntax consist of a set of words and rules for how the words can be combined. The sentence ”White wrights write withering words where wanderers wash worn winter wagons” is syntactically correct English, despite being a completely nonsensical phrase. The words are all valid English words, and they are combined in a way that does not violate English grammar. The meaning of the sentence is of no concern to the syntax. The following quote is not syntactically correct, ”(..) the dark side I sense in you.”. This is a quote from Yoda, a character in the Star Wars universe. He is widely known for breaking the subject-verb-object rule of the English language. The subject of this sentence is ”I”, the verb is ”sense” and the object is ”the dark side”. As you can see, the order in which these words occur is not subject-verb-object, but rather object-subject-verb. This phrase is therefore a breach of English grammar, and not syntactically valid. In the context of programming languages, a syntactically valid program is one that can be executed. The program does not have to

do anything useful in order to be syntactically valid, it only needs to be interpretable as a program. The syntax of a programming language can aid in making the language more intuitive.

2.1.2 Semantics

The semantics of a language assigns meaning to the words and sentences of the language. Semantics define what can be expressed in a language, while the syntax defines how it can be expressed. It is the semantics of a programming language that decides what will happen when a syntactically valid program executes. There are many semantic constructs that are common to many programming languages, while their syntax may differ. The foreach loop is an example of a language construct that is common to many programming languages. The foreach loop lets the programmer specify a sequence of instructions that is to be executed once for each element in a list.

```
for(Car someCar : listOfCars) {
    // do someting to someCar
}
```

Figure 2.1: Iterating over a collection using a foreach loop in Java

```
foreach(Car someCar in listOfCars) {
    // do something to someCar
}
```

Figure 2.2: Iterating over a collection using a foreach loop in C#

Figure 2.1 and Figure 2.2 show how a list of cars, called listOfCars, can be iterated over using a foreach loop in Java and C#, respectively. The semantics of the foreach loop is the same in both languages, there is only a slight syntactical difference between the two. Each element in listOfCars will in turn be assigned to the variable called someCar and then the body of the loop will be executed, i.e. the sequence of code between the curly brackets.

The semantic concepts of programming languages are nothing more than convenient

abstractions, they provide useful solutions to common problems. Programming is not unlike a lot of other activities, in that the same problems repeatedly surface. Iterating over a collection of elements is an example of a problem that arises numerous times in most programming projects. There are many different ways of solving this problem in most languages, and using a foreach loop is just one example.

```
for(int i = 0; i < listOfCars.size(); i++) {
    Car someCar = listofCars.get(i);
    // do something to someCar
}
```

Figure 2.3: Iterating over a collection using a for loop

Figure 2.3 shows how the same problem can be solved using a for loop in stead of a foreach loop. This solution is more verbose, and contains more details, hence, more room for errors. Due to the frequency at which the problem occurs, both C# and Java have decided to include the foreach loop. The foreach loop does not enable the programmer to express anything new, it only allows the programmer to reuse an existing solution to a common problem.

[1]

Static and Dynamic Typing

All data stored in a computer is in reality just sequences of binary values. The only difference between a single character of text and an integer stored in a computer is how the data is treated. To make sure that programmers do not get their data mixed up programming languages usually provide what is referred to as a type system. Type systems keep track of what type of data any given sequence of binary values is.

There are different types of type systems and there is a fundamental distinction that needs to be made. Type systems can be put into two categories, static type systems and dynamic type systems. The difference between the two is that in a static type system the type of a variable can be determined from the source code alone, in other words the type

[1] There are some slight semantic differences between Figure 2.1 and Figure 2.3 in regard to modification of the list during execution of the loop. However, assuming the list listOfCars remains unchanged during execution the two examples will have identical behavior.

of all variables are known in advance of execution. In a dynamic type system the types of variables are determined at runtime, the same variable might even have different types at two given moments during execution.

In general dynamic type systems are more flexible than static type systems. However, greater flexibility also removes some of the guarantees that a static type system can provide. The decision of whether to use a static or dynamic type system is one that is very fundamental in programming language design.

2.2 General Purpose and Domain Specific Languages

Languages can be divided into two categories, domain specific languages (DSL) and general purpose languages (GPL). DSLs are designed for a specific domain to simplify the process of describing things within that domain. GPLs are designed to be versatile enough to be used in just about any context or domain. There are many examples of DSLs that are widely used in the software industry, such as HTML, CSS, and regular expressions. These languages are highly suited for their respective domains, however, they are generally not suited for other domains.

General purpose languages such as Java or C, could be used to describe the same things as you can in HTML, CSS or regular expressions. However, doing so would probably lead to less intuitive and more verbose code. The concepts available in general purpose languages are often quite abstract and general, which makes them applicable in many domains. Programming with a GPL often includes defining the concrete concepts of the problem domain in terms of the abstract concepts of the language. The concepts of a DSL are more concrete and more specific, and the developer can spend less time defining concepts and spend more time applying the concepts.

There are several reasons for developing and using domain specific languages. The users of DSLs are not necessarily professional software developers. DSLs can be used to enable domain experts to do basic programming within their domain. Due to the fact that the concepts of the language match the ones of the problem domain, the development process can be greatly simplified. Professional software developers usually use DSLs in order to increase productivity and avoid ”reinventing the wheel”, that is, spending time

solving problems that have already been solved.

The major drawback of relying on DSLs is that the programmer needs to learn a new language every time she starts working in a new domain. However, if the programmer is going to be working within the same domain for a longer period of time, then a DSL can improve productivity. This makes the decision of whether or not to use a DSL a trade off between time spent learning the language, and the increased productivity once mastered.

Programming in general purpose languages often requires a lot of abstract thought, which can be a challenging mental exercise, especially for children. Hence, creating educational DSLs with concrete and recognizable concepts is a quite common approach to teaching programming to children. However, abstract thought is something that is quite inherent to the programming activity, and avoiding it all together close to impossible. Educational DSLs often focus on a subset of common abstract language concepts, such as loops, and try to create a concrete context around them.

2.2.1 Textual and Visual Programming

The majority of programming languages used in the software industry are languages with a textual syntax. The syntax of these languages are defined in terms of sequences of characters, and programming becomes a matter of writing text. Visual programming languages (VPL) replace the textual representations of textual languages with visual components. Software developers using VPLs are not only writing text, or even writing text at all, but also connecting and combining visual components. VPLs have the advantage that they can create representations that are more intuitive than written words.

Visual programming has been around for several decades, yet VPLs only occupy a small area of the programming landscape. There has been done much research into the field of visual programming and some inherent challenges have been uncovered. Because text is a very concise method of expressing meaning, creating visual representations that are equally concise has proven to be a challenge. Therefore, many VPLs require a lot more space on screen than their textual counterparts. This is an issue as it can make it hard to get a proper overview of sufficiently large parts of the code being written. In addition, programming often relies on quite abstract concepts. Creating intuitive visual

representations for these abstract concepts is often very hard. Experience shows that VPLs are best suited as DSLs, in part due to the fact that the concepts of DSLs are generally more concrete than those of GPLs, which simplifies the process of creating good visual representations for them (Myers, 1986).

VPLs also have the advantage that they can make it harder for the developer to make syntax errors, or even entirely impossible. This is an aspect of visual programming that has made it a central part in modern educational programming languages. Syntax is a frequent source of errors for most novice programmers, and being able to completely rule these out can be very beneficial. This enables the novice programmer to focus on understanding semantics and learning problem solving, rather than spending time on understanding syntax.

2.3 Programming Languages for Children

Through the years many different programming languages have been developed to aid in teaching programming to children. These include everything from DSLs with visual syntax to GPLs with textual syntax.

2.3.1 Logo

Advocacy for computer programming as a worthwhile educational domain is nothing new. During the 1980s there was a similar effort in many countries around the globe. Large claims were made about the benefits of teaching computer programming to children, and quite a few tools were developed to aid the process of teaching programming.

Among the most historically important tools for teaching programming is the Logo programming language. Logo is an educational programming language that was developed as early as 1967. It is a multi-paradigm GPL with a textual syntax.

The creators of the Logo programming language also developed theories and guidelines for how programming should be taught to children. They boasted about the great benefits to teaching programming to children, including positive effects way outside the bounds of pure programming ability. Many of these claims have been shown to be ex-

aggerated, and their proposed approach to teaching programming to children is widely questioned today (Pea, 1983). However, Logo is still held as an important educational programming language.

Turtle Graphics

Despite being a general purpose language, Logo is often associated with what is called turtle graphics. Turtle graphics is a relatively intuitive, yet an arguably cumbersome, way of drawing graphics on a computer screen. The core of turtle graphics is the turtle, a simple shape that is drawn on the screen. The programmer can instruct the turtle to move around, drawing a line where ever it goes. Turtle graphics was included in the standard library of the Logo language, and hence provided a default way of doing basic graphics in the language. There was no need to install any third party tools or libraries in order to get started writing software with graphics, as it was shipped with the language and provided a simple and standardized way of doing it.

Turtle graphics has since been included in many other languages and is currently included in the standard distribution of the Python programming language. It has therefore also become a convenient way of doing simple and basic graphics programming in Python, and has been a central part of teaching the Python programming language to children in Norway.

2.3.2 Greenfoot

Greenfoot is a programming environment that seeks to teach object oriented programming. The core of the approach taken by Greenfoot is to provide the users with a simple game development framework that can be used to program the behavior of visual components drawn on a canvas. This is an approach that is very similar to that of Scratch, as programs are written by defining the behavior of visual elements on the screen. The interface provided by the framework also resembles that of turtle graphics.

As opposed to Scratch, Greenfoot lets the user choose between two different programming languages for interacting with the framework. There are two options, either to write programs using Java or a partially visual language called stride. Stride is a visual language,

however, it still relies on the user writing textual code as well. By doing so it is able to provide the user with some helpful guidance, but it is not able to provide the same guarantees that languages such as Scratch can provide. Stride can for instance not guarantee that syntax errors will not occur.

2.3.3 Combining Visual and Textual Programming

There have been previous attempts at combining visual and textual programming. (Erwig and Meyer, 1995) addresses the poor adoption of VPLs, despite years of research on the subject. This research points to the advantages and drawbacks to VPLs and proposes a hybrid visual programming language(HVPL) as a solution. An HVPL is a language that combines visual notation with textual notation. In addition, they propose a framework for HVPLs that can be used to integrate with conventional textual programming languages. To utilize the strength of VPLs, the fact that they are best suited as DSLs, the framework allows for specialized versions of the HVPL to be created.

2.3.4 Generating Textual Code from Visual Code

(Cheung et al., 2009) addresses many of the same issues as this project. The authors claim that there is an unfilled gap when it comes to programming environments for children and youth. They experienced that some students found VPLs such as Scratch to be too simple and too limiting. However, the same students found textual programming to be too hard. To solve this problem the authors present a text-enhanced graphical programming environment. The programming environment provides the user with a VPL that can be used to generate textual source code. There are many examples of tools that apply this technique, as it allows children to see the textual equivalent of their visual code, and it is also frequently used to enable visual programming in environments originally designed to textual languages.

2.4 Scratch

Scratch is a very popular tool for teaching programming to children by enabling them to create simple animations and games in 2D. It consists of three parts, a visual programming language, a game development framework, and an integrated development environment (IDE). The programming language is a DSL that precisely models the domain of the game development framework.

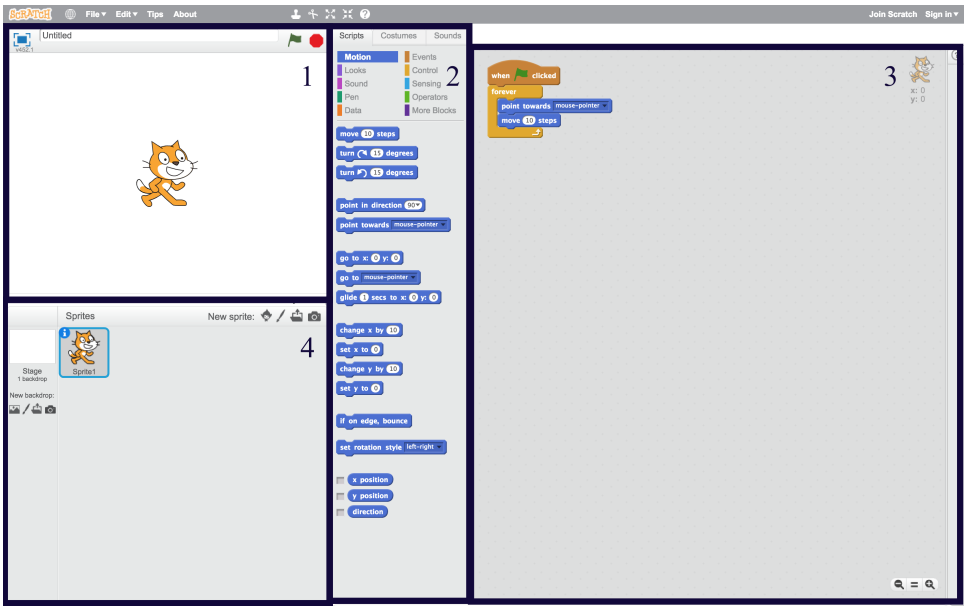


Figure 2.4: The Scratch IDE

Figure 2.4 shows what the Scratch IDE looks like. The section marked with the number 1 is the stage, which is canvas on which the games played. Section 4 is the overview of the sprites in the game, where sprites can be created or removed. Section 2 is the palette of statements and expressions. Finally, section 3 is the code editor, where blocks from the palette are assembled. The example code ensures that the sprite follows the mouse pointer.

The most fundamental concept in Scratch is that of the sprite. Creating programs in Scratch usually starts by defining the sprites that you need. A sprite is a visual element on the screen that can be moved around, rotated or otherwise transformed. Figure 2.4 shows an example of a sprite, namely the cat which can be seen on the stage.

The behavior of the sprite is defined by using the code editor on the right hand. To add behavior to a sprite, you need to find find the desired code blocks in the palette and drag them into the code editor where you assemble them. Figure 2.4 shows a simple program where the cat will follow the mouse pointer as soon as the green flag on the stage is clicked. This work flow of first defining visual elements and then defining their behavior is useful as it makes programming less abstract, as all code is associated with a visible component.

2.4.1 Syntax

The Scratch programming language is a VPL. Programs are created by connecting blocks representing statements and expressions. The shapes of the blocks indicate how they can be combined with each other, blocks with rounded edges can only be inserted into slots with rounded edges etc. The Scratch code editor enforces that blocks can only be combined in ways that are syntactically valid. If you were to attempt to connect two incompatible blocks, the editor would simply refuse to snap the two blocks together. This means that creating syntactically invalid programs in Scratch is impossible, which is a great advantage for users who are just beginning to learn how to write code.

Figure 2.5: Two attempts at creating an if-statement in Scratch. The one on the right remains disconnected due to incompatible blocks

Figure 2.5 shows how the Scratch editor avoids syntax errors. The example on the left shows a valid and fully connected code block in Scratch. On the right the green block is disconnected from the rest, due to the fact that it has rounded edges, as opposed to the diamond shaped slot. The code on the right is still syntactically valid. However, the green block representing the addition of 10 and 20 will be ignored, and the if-statement will use a default value for the empty slot.

2.4.2 Semantics

The semantics of the Scratch language are relatively unique. There are many of the concepts of the language that look and feel familiar to most programmers, however, there are some fundamental differences between Scratch and most other languages.

The Scratch programming language is an event-driven language. Event-driven programming is a programming paradigm where the flow and execution of the program is determined by events such as user input, sensor outputs, or incoming messages from other processes. Event handlers are a core concept in event-driven programming. An event handler is nothing more than a sequence of code that is to be executed every time a specific event occurs.

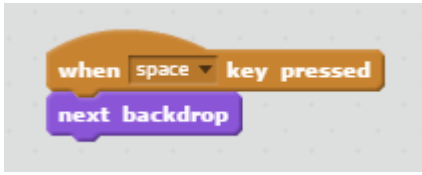


Figure 2.6: An event handler that changes to the next backdrop when the space bar is pressed

Figure 2.6 shows an example of an event handler in Scratch. The orange block on the top defines an event handler that will be executed each time the space bar is pressed. Blocks attached to it define what happens when the event occurs, in this case the background image of the stage will be changed.

Each running event handler in Scratch acts as a separate thread of execution. The actions performed by one event handler can seemingly happen at the same time as the actions performed by another event handler. However, this is not the quite the case. There will at most be one event handler running at any given moment, but they take turns doing work. The execution of event handlers is therefore concurrent, but not parallel (Maloney et al., 2010). This is equivalent to the situation you have when you are running multiple programs on a computer with a single processor core. The computer is seemingly doing multiple things at once, but in reality is only switching fast from one task to another.

There are many different issues that arise from concurrency in computer science, such as race conditions and deadlocks. Scratch tries to handle a lot of the issues related to

concurrency behind the scenes, rather than providing synchronization primitives for the programmer to use. Usually when programming with concurrency in mind one has no guarantees as to when a thread will be paused and another will be started. Scratch on the other hand provides a very clean and simple set of rules for when a thread will yield control to another. Threads in Scratch only ever yield control to another if it has run out of work i.e. terminates, decides to sleep for a given amount of time (sleep is a statement block in the language), or if it has completed an iteration of a loop. In addition, Scratch uses a simple round robin scheme for scheduling the threads. This means that the runtime will treat all the threads equally, and no thread will be given processing time twice before another gets its turn (Maloney et al., 2010).

The simple and predictable scheduling and yielding of control has a lot of powerful effects. Among them is the fact that every sequence of code blocks becomes an atomic operation, unless it includes a loop or a sleep statement. This removes a lot of the concerns related to race conditions. The round robin scheduling makes sure that all loops execute at the same pace, which is an uncommon luxury.

Scratch uses what is known as an actor based runtime model. Each sprite on the screen is also a logical component in the runtime called an actor. The actor has a set of event handlers that can modify the state of the sprite. The state of a sprite is a combination of its visual properties, such as size, rotation, position etc, and a set of private variables. Sprites can not modify each others' state directly, but can rely on message passing for coordination. There is a special actor that is not linked to a regular sprite, but rather to the background of the screen. This actor is referred to as the scene. The variables defined in the scope of the scene are available to all other actors directly, i.e. they are in the global scope.

2.5 Python

Python is a very popular textual programming language. It is a very flexible language with a simple and concise syntax. In contrast to Scratch, Python is not specifically created for educational purposes, nor is it a domain specific language. It is a general purpose programming language that is used both in academic circles and in the software industry.

As most GPLs, Python is not tightly coupled to any single library or framework, but rather supports working with a multitude of different frameworks and libraries. Python even provides simple mechanisms for interfacing with libraries implemented in other languages. The same thing goes for the development environment in general, Python is not bound to any one IDE.

2.5.1 Syntax

Python has a relatively minimalistic syntax compared to many other languages. It is intended to first and foremost be a concise and easily readable language.

```
for someCar in listOfCars:
    #do something to someCar
```

Figure 2.7: Iterating over a collection using a foreach loop in Python

Comparing Figure 2.7 to Figure 2.2 some major differences between the syntax of C# and Python can be seen. From a purely syntactical perspective the primary difference is that Python uses a lot less character to express the same logic. The parenthesis have simply been removed, and the braces enclosing the body of the loop have been replaced with a single colon, at least it seems that way at first glance. However, it is worth mentioning that there is something going on here that is not immediately apparent.

[2]

Whitespace aware syntax

Python uses what is called whitespace aware syntax. This means that whitespace, that is characters that are displayed as empty space, hold meaning to the syntax. Examples of whitespace characters are new lines, tabulations, and spaces. In Python all of these hold meaning to the syntax and are used to group statements into code blocks. Languages such as Java and C# use the curly brackets to denote the beginning and end of code blocks.

[2]The observant reader might have noticed that the expression ”Car someCar in listOfCars” from Figure 2.2 has been replaced with simply ”someCar in listOfCars”. The word ”Car” has been removed from the expression. This is partly due to a semantic difference between the two languages. Python is a dynamically typed language that allows elements of different types to be placed in the same list, as opposed to C# which requires some common type for the elements of the list.

2.5.2 Semantics

The semantics of Python are in many ways more traditional than those of Scratch, however, it sets itself apart from many other textual GPLs in many ways. Python is built around a principle of everything being an object. This is combined with a very flexible dynamic type system and built in types that work well with all objects.

As previously mentioned programming using a GPL often includes spending a lot of time defining the concepts of the problem domain in terms of the concepts of the GPL. Python seeks to minimize this effort by encouraging extensive use of flexible built in types as opposed to defining new types. It is often considered as idiomatic Python to use tuples as opposed to defining a new class and instantiating it.

Python also discourages the use of traditional and verbose control structures. The for loop is an example of a control structure that is very common, however, Python does not provide a for loop. In stead it provides a foreach loop in combination with useful built in generator functions, which can be used to produce the same effect. Figure 2.7 shows an example of such a foreach loop, that simulates the behaviour of traditional for loops.

Python also has a dynamic type system which is an advantage in terms of keeping code short and concise, as it does not require the programmer to define the type of a variable. This is convenient in many situation, and is often held a great advantage of using the language. However, there is a backside to this dynamic type system as well. Since types are determined at runtime it opens up for a class of errors that do not exist in most statically typed languages. Operators are often not applicable to all types of data, and applying an operator to data that it is not defined for will cause an error. Avoiding these problems is left to the programmer, which can be a source of many errors, especially for children.

Chapter 3

Methodology

This chapter describes the methodology applied in this project, and the reasoning behind choosing it.

This thesis seeks to explore how the transition from visual to textual programming can be eased, and seeks to find out how this can this can or should be done by combining the two. This is a problem that can be addressed from many angles, such a pedagogical science, psychology, or a technical perspective. Each of these perspectives can be assumed to be of equal worth, but exploring them all in full is far beyond the scope of this project. The natural approach for me, as a computer science student, was to explore the problem from a technical point of view. By looking at the existing tools available for teaching programming to children and examining their properties I could attempt to find the cause(s) of the issues related to making the transition from visual to textual programming, and then proceed to attempt to confirm that these were the actual source of the problems. The most straight forward way of confirming my assumptions was to create a new set of tools that would attempt to mitigate the issues by removing what was assumed to be their cause. The new tools could then be tested and evaluated by their ability to solve the identified issues.

This aligns well with the tenets of the design science methodology, which seeks to explore and solve problems by designing artifacts that attempt to solve them (March and Smith, 1995). For this reason design science was chosen as the primary research methodology for this project.

3.1 Design science

Design science is a research methodology that is commonly used in information systems (IS) research. Design science acknowledges the fact that technology is the result of intelligent design, rather than a naturally occurring phenomenon. This distinction is important as it can render the traditional research methods of natural science unfit for some design research projects. Traditional natural science seeks to understand and explain how a phenomenon works and to understand reality. However, when creating technology you usually start off with a problem that needs to be solved, rather than a phenomenon that needs to be explained. Design science acknowledges this distinction and provides some guidelines for how research projects of this nature should be conducted.

(March and Smith, 1995) describes design science in terms of two orthogonal dimensions, artifacts and design processes. The first dimension is concerned with what is actually designed through the research process. The products of design science research are referred to as artifacts. There are four different types of artifacts that can be produced, constructs, models, methods, and instantiations. These four constitute the first dimension of design science. The other dimension is that of the design processes, that is, the steps taken while doing the research. There are four design processes as well, build, evaluate, theorize, and justify.

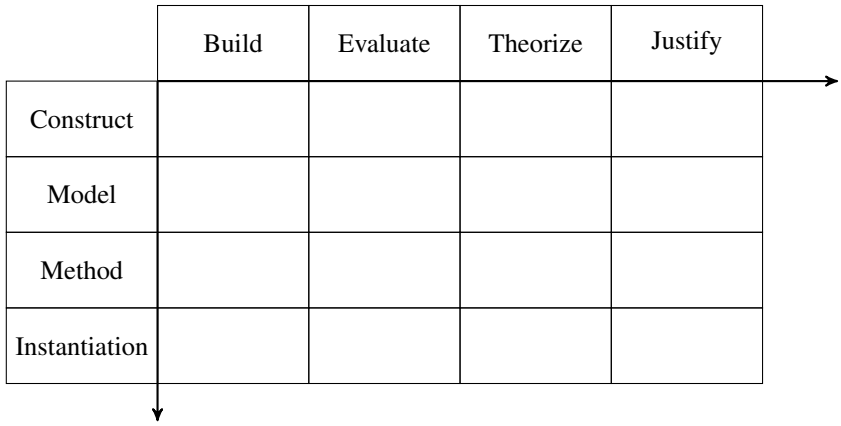


Figure 3.1: The two dimensions of design science according to (March and Smith, 1995).

Figure 3.1 shows the two dimensions of design science. Constructs are formal defini-

tions of concepts within a given domain. Most areas of science got its own set of concepts and expressions that are specific to the domain. In fact, these can be considered as domain specific languages(DSL), and creating a new construct can be viewed simply as defining and adding a new word to the DSL. Creating a new construct can enhance the language used and help reason about a set of problems within the domain.

Models are sets of propositions or statements that express relationships between constructs. A model can be viewed as a description of how things are, they propose that phenomena be understood in terms of certain concepts and the relationships among them.

Methods are sets of steps that can be taken to perform a task, such as an algorithm or a guide as to how to approach a problem.

Finally we have instantiations, which can be considered as the realizations of constructs, models and methods. Instantiations can typically be fully implemented and working IT systems.

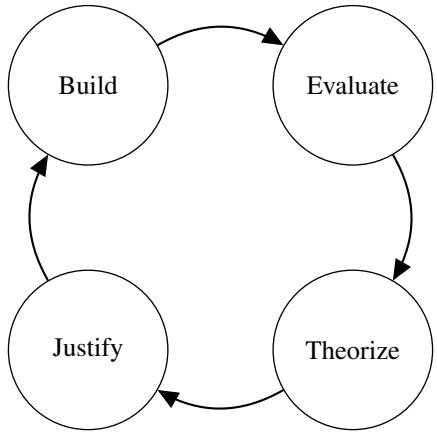


Figure 3.2: The iterative process of design science

The design processes, build, evaluate, theorize and justify are all important to the process of generating useful knowledge throughout a design science project. Design science is an iterative research methodology and all of the design processes mentioned are usually repeated multiple times, as shown in Figure 3.2 throughout the course of a project.

The first step is that of building an actual artifact. Building an artifact is done by first identifying a set of requirements for the artifact and then trying to create an artifact that

meets the given requirements. The initial requirements for an artifact can be derived exclusively from data collected from other projects. However, during the proceeding iterations of the cycle the requirements should be adapted to accommodate the knowledge gained through the other design processes, evaluate, theorize, and justify.

Evaluation is the next step after building. As previously mentioned, artifacts are built to solve problems. Evaluating an artifact usually involves putting the artifact in its intended environment and collecting data regarding which degree it solves the problem it was designed to solve.

After evaluation the researcher is left with data concerning how the artifact performed. It is important to determine why and how the artifact worked or did not work within its environment. This is what is referred to as the theorize step of design science. New theories are created in an attempt to explain the data collected. The theories should explain the characteristics of the artifact and its interaction with the environment that resulted in the observed performance. Finally, justification of the developed theories must be provided.

Design science has several strengths and weaknesses. Among the pitfalls of design science is ending up doing design as it is being done in the IT industry, rather than doing actual research. It is important to focus on the generation of knowledge, rather than just the creation of a functioning IT system. It is important that data collection and analysis is used to explain the reasoning and arguments behind the findings throughout a project.

Design science is a suitable choice of method for answering the research questions for this thesis, as answering them is hard to do without building and evaluating a prototype. The first research question addresses how visual and textual programming can be combined to ease the transition from visual to textual programming. Predicting the effects and qualities of a proposed system is no simple task without a working prototype. Due to the complexity of the problem it is hard to provide answers to this question without actually implementing a possible solution and evaluating it.

3.1.1 Data collection

Data was collected for this project by means of a focus group. The focus group is a qualitative data collection instrument that has been used in many different areas of research

throughout the years, especially marketing and human machine interaction. A focus group is defined as a moderated discussion among 6-12 people who discuss a topic under the direction of a moderator, whose role is to promote interaction and keep the discussion on the topic of interest (Stewart and Shamdasani, 2014). The questions in a focus group are open ended but at the same time carefully planned in advance.

During the initial planning of the project several other data collection instruments were considered, including both qualitative and quantitative instruments. There were many aspects to take into consideration when deciding on how to collect data for this project. The intended end users of the system are children and youths, and it was therefore natural to use them as the source for data and feedback. Using children in research can introduce additional challenges compared to using adults. Especially since the problem addressed in this thesis is a relatively complex one, that can be viewed from many different angles and that has many implications.

Another consideration is that the data collection instrument would have to be one that I as a researcher would be able to implement properly, and make sure that the data collected was of proper quality, rather than producing a lot of seemingly interesting data ridden with flaws. Despite being quite familiar with the problem domain of teaching programming to children, my formal background is strictly technical. Resisting the urge to go beyond the extent of my abilities and delve too far into psychology and pedagogical science has been a factor in deciding on which data collection instrument to use. With this in mind, it was decided that the project would be evaluated using techniques from user driven design, from which I had previous experience.

During an initial phase of the project it seemed appealing to evaluate the prototype by letting the intended end users, children currently making the transition from visual to textual programming, interact with the prototype directly. The prototype could then be evaluated by the children's ability to use the prototype to solve problems or simply gather data about their impressions of using the system. However, it is important to ensure that it is the core design decisions of the system, rather than the actual implementation that are being evaluated. Direct access to the prototype sets very strict requirements to the usability of the system, flaws such as bugs in the software might shift the children's focus

away from the core design decisions and leave the collected data polluted with what is essentially bug reports. For this reason giving children direct access to the prototype was ruled out, and it was decided that they should rather be shown examples of the system in use.

The developed prototype is a relatively complex system, due to the complexity of the prototype it would be naive to assume that I would be able to consider every relevant aspect of the system for evaluation. The data collection instrument chosen would therefore have to be suited for new ideas and reflections to be brought up during the evaluation of the prototype. In general, qualitative approaches are more suited for uncovering and exploring new ideas (Sofaer, 1999). Therefore it was decided to choose a qualitative approach rather than a quantitative one.

The focus group was chosen among the qualitative data collection instruments in an attempt to ensure that the participants would be comfortable enough to express their opinions and thoughts freely. Having a one on one interview with an adult can be intimidating for children, which might in turn affect how they respond to questions. In order to avoid this pitfall, the focus group was chosen as would allow the children to discuss the topics in a group of their peers.

3.1.2 Data analysis

Data from focus groups can be analyzed using many different qualitative data analysis techniques. For this thesis template analysis was chosen, as it provides a relatively structured and yet flexible way of analyzing the data. The first step in template analysis is to create an initial template by exploring the focus group transcripts, academic literature, the researcher's own experiences, anecdotal and informal evidence, and other exploratory research (King, 1998). Before conducting the data collection it was clear that there would be some apparent patterns in the data that could be used as an initial template for data analysis.

Chapter 4

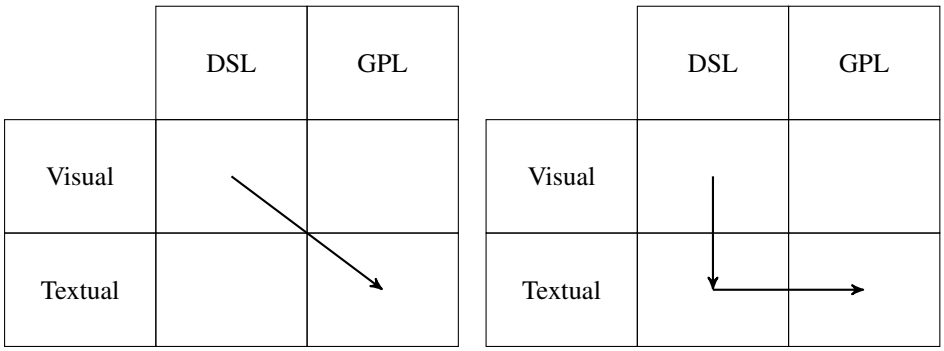
Artifact Design

This project follows the guidelines of design science, and this document goes through the steps of build, evaluate, theorize and justify. This chapter covers the first step of this process, namely the building of the artifacts. The designed artifacts are presented along with the reasoning behind their design, proceeded by a short description of the development process.

The research objective of this project is to ease the transition from visual to textual programming. Experience shows that this is a transition that many children struggle with and find to be too challenging, and hence stop working on their programming abilities. The learning curve is simply too steep for many children when trying to master textual programming, even after mastering visual programming (Cheung et al., 2009). It is important to note that this thesis does not attempt to find ways of getting children from visual to textual programming faster, it is an attempt at finding out how more children can survive the transition with their motivation intact.

The fundamental issue at hand is that the learning curve from visual to textual programming is too steep. There is one or more issues related to making this transition that causes this steep learning curve. During an early phase of this project a possible source of many issues was identified, namely the fact that going from visual to textual programming is often tightly coupled with another transition, the transition from using a DSL to using a GPL.

The VPLs used to teach programming to children are almost exclusively DSLs, while the textual languages that are being taught are GPLs. This is the case at Kodeklubben Trondheim where the children are first taught Scratch, a visual DSL, and then proceed to being taught Python, a textual GPL. The fundamental assumption posed by this thesis is that this attempt at making two transitions at once is a central cause to why the learning curve is too steep for many children. There are naturally some inherent challenges related to mastering textual syntax, but these are not addressed in isolation by most children. For this reason this project takes the approach of trying to mitigate the issues by creating an environment that is able to isolate these two transitions from each other. Due to the inherent limitations of both DSLs and VPLs this project does not argue that either transition should be ignored, but that they should be addressed one at a time.



(a) Route taken by many children today. **(b)** Route proposed by this thesis.

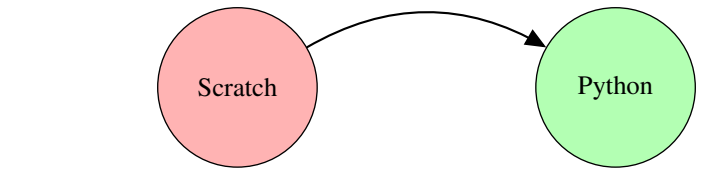
Figure 4.1: Two possible routes from visual DSLs to textual GPLs

Figure 4.1a shows the double transition that is attempted by many children, and Figure 4.1b shows the separation proposed by this thesis. The children should be able to move from using a visual DSL to using a textual DSL. After mastering a textual DSL they should move on to using a textual GPL.

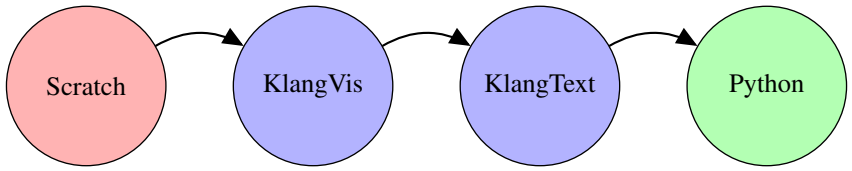
The first research question is related to how visual and textual programming can be combined in order to ease the transition from visual to textual programming. The approach taken in this thesis is to create a new DSL, an instantiation, that has both a visual and a textual syntax, a programming language that can be edited either in a visual editor or a traditional textual editor. This does not only isolate the transition from visual to textual

programming from the transition from DSL to GPL, but takes it one step further. This way the users can write software in two languages that semantically identical, it is not just that the visual and the textual languages are both DSLs, they are completely identical aside from the syntax. It eliminates the need to learn any new programming concepts when moving from one to the other, other than those related to syntax. For convenience the new programming language has been given a name, Klang, which will be used throughout the rest of this document. The two notations have been given their own names as well, the visual notation is referred to as KlangVis, while the textual notation is referred to as KlangText.

It is important to note that the artifacts are intended as an intermediate between the existing VPLs used for teaching programming to children, such as Scratch, and textual programming languages used in the software industry, such as Java or Python. The system is not intended as one that is suited as children's first introduction to programming, nor one that they should stick with for a long time. It is simply an additional stepping stone used to ease a specific part of the learning process. This means that the system is built on the assumption that its users have got previous experience using VPLs such as Scratch, and that they are moving towards mastering a textual GPL such as Python.



(a) Direct transition from Scratch to Python.



(b) Transition from Scratch to Python with Klang as an intermediate language.

Figure 4.2: The intended context for the Klang programming langauge.

Figure 4.2a shows the path taken by children at Kodeklubben Trondheim today, a direct transition from Scratch to Python. Figure 4.2b shows how KlangVis and KlangText are

intended to fit into the existing path. This thesis seeks to explore the general problem of moving from visual to textual programming, however, Scratch and Python have been used as concrete examples of such languages.

As can be seen from Figure 4.2 the introduction of Klang adds additional steps to the process. Each of these steps are taken into consideration in the design and will be examined in later chapters. The fundamental idea is that it should be easier for children to move from Scratch to KlangVis, then from KlangVis to KlangText, and finally from KlangText to Python, rather than going the direct route from Scratch to Python. This means that each of these steps should be a lot simpler than going the direct route, if not, the system is nothing but an inconvenience.

The second research question addresses how a development environment can support the transition from visual to textual programming. The approach taken by this thesis is to create a game development framework that tightly integrates with the Klang language, and that can utilize existing tooling for the drawing of visuals. This approach draws inspiration from other projects, such as Logo, Scratch, and Greenfoot, which all provide a default way of doing computer graphics. The Klang language and the development framework are tightly coupled in order to take advantage of the possibilities that come with creating a DSL rather than a GPL. The development framework seeks to provide similar features to those found in Scratch and provide some additional features in order to create an extra incentive to using the prototype system. Given the integration between the framework and the Klang language, the prototype can be considered as a DSL for creating visual two-dimensional games.

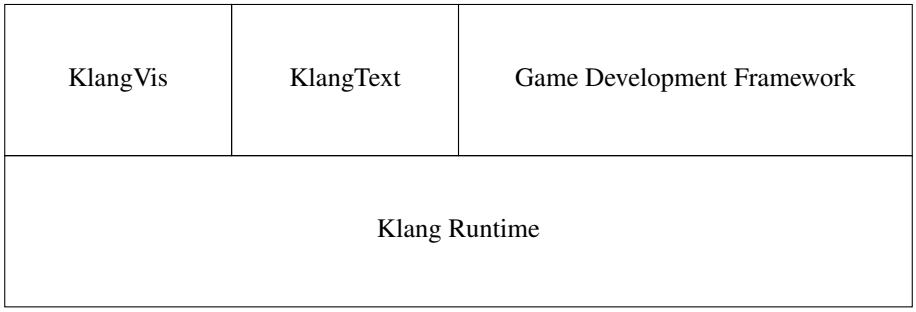


Figure 4.3: Overview of the artifacts developed.

Figure 4.3 shows the artifacts that make up the prototype system. The Klang programming language is linked to the game development framework, and provides two notations, KlangVis and KlangText. The following sections will describe each of these artifacts and the reasoning behind their design.

4.1 The Artifacts

The artifacts are all relatively tightly coupled. This has the consequence that the design of one artifact has an impact on the design of the other two. The artifacts all contribute towards the same goal, and need to work properly together. However, each of them need to work properly in isolation as well.

Transitions shown in Figure 4.2b have been used as guidelines for the design of the artifacts. The design of KlangVis takes into consideration both the transition from Scratch to KlangVis and the transition from KlangVis to KlangText, and the equivalent is the case for KlangText and its surrounding transitions.

4.1.1 Klang

The Klang programming language, like any other language, is defined in terms of syntax and semantics. The semantics of the language define what can be expressed in the language, while the syntax defines how it can be expressed. The most fundamental decisions when creating a programming language is often that of the semantics. As previously mentioned, is important that the design of the language is suited for all of the transitions from Figure 4.2b. It is important that the semantics of Klang cater to the right transitions. The transition from KlangText to languages such as Python, is a transition from a DSL to a GPL, and will involve major semantic differences either way. KlangVis and KlangText share the same semantics, therefore this transition is of little interest when defining the semantics of the language. However, it is important that the experience of moving from languages such as Scratch to KlangVis is relatively painless, both in terms of mastering syntax and semantics.

For these reasons the semantics of the Klang are relatively similar to Scratch. In addi-

tion, by borrowing concepts from Scratch the project is hopefully able to build further on the success of Scratch, rather than ending up reinventing the wheel. The language borrows concepts such as an actor based runtime model, and implements the same concurrency mechanisms. Programs are defined in terms of interacting actors that respond to occurring events and communicate with each other through message passing.

As in Scratch, the actors are all coupled with visual components that are defined by the game development framework. There are two different types of actor, a special actor referred to as the scene, and any number of sprite actors. The sprite actors are visual components that can be moved around on a canvas, while the scene is the canvas itself. Each actor is only able to modify its own state, such as moving around on the canvas or changing color etc.

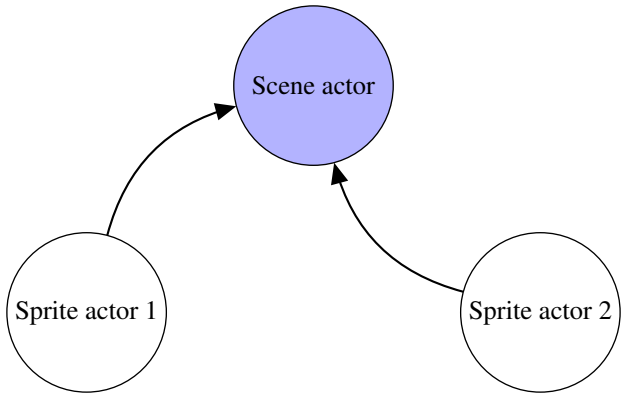


Figure 4.4: Variable scopes in Klang. Each actor has got a private variable scope, aside from the scene actor, which is treated as the global scope.

In addition to being able to modify the state of its visual representation, each actor can define a set of variables. The variables are private to each actor, aside from the variables of the scene actor that are placed in the global scope. Figure 4.4 shows an abstract representation of a program with two sprite actors, where the edges represent access to another sprite's variable scope.

The statements and expressions available in the language are mostly ones that are common to just about any imperative programming language. The language provides mechanisms such as if statements, loops, basic arithmetic operators etc. These are all

concepts that are found in most popular programming languages, and are nothing unique to this language. Klang does however provide some simplified versions of different loops, such as a forever loop, which is a loop that executes its body repeatedly until the program is stopped.

The actor based runtime model is thoroughly event driven, meaning that any sequence of instructions executed is a response to some event. There are many different types of events, such as the game starting, a sprite being clicked on, or a key being pressed. Code sequences executed in response to events are usually referred to as event handlers. The execution of event handlers in Klang is performed in a similar manner as they are in Scratch. In many programming environments event handlers need to be short lived as they are executed one at a time by the same thread of execution, and creating a long lived event handler would block the execution of other handlers.

However, Klang solves this issue by borrowing the concurrency model of Scratch. Scratch executes event handlers concurrently and provide synchronization mechanisms behind the scenes. Event handlers are executed concurrently, but not in parallel, meaning that there will only ever be one event handler running at a time, but their execution will be interleaved. The system uses non-preemptive scheduling, meaning that the event handlers are never forced to pause, but decide for themselves when they are to yield control to another. The only way to create a long lived event handler in Scratch is to use loops to repeat some task over and over, and the loops implemented in Scratch will yield control to another handler after completing an iteration (Maloney et al., 2010). This has the great benefit of enabling the user to use infinite loops to perform long running tasks, without having to think about either synchronization or blocking the execution of the program. By borrowing this concurrency model Klang is able to provide the same level of convenience.

A fundamental strength that languages such as Scratch has is that they make it impossible to create syntactically invalid programs, and there is no way of generating runtime errors. As opposed to Scratch, which is an exclusively visual language, Klang is both visual and textual. Therefore there is no way of guaranteeing that syntax errors will not occur, however, it has been designed to avoid any kind of runtime errors. In order to achieve this guarantee when it comes to runtime errors Klang is a statically typed language. This way,

errors such as applying an operator to data that it is not defined for will be caught in the code editor, rather than during execution. There are three data types available, boolean values, double precision floating point numbers, and strings. The language also provides thorough static analysis which allows for any possible source of errors to be caught ahead of execution.

Further details about the Klang language can be found in the appendix in the form of UML Diagrams.

KlangVis - visual syntax

The core of this project is related to the transition from KlangVis to KlangText, and especially the syntactical perspective of this transition. This project tries to ease exactly this transition, from visual notation to textual notation. However, as previously mentioned the transition from Scratch to KlangVis is also an important one in this regard. The visual syntax should feel familiar to the user when they first start using it, but there should also be some consistency between the visual and textual syntax in order to smooth the transition from one to the other.

KlangVis is built on top the visual editor presented in (Hungnes, 2016), and only modifies its defaults slightly. This existing editor defines the visual syntax of KlangVis. It allows for some customization out of the box which has been used to create a syntax which is relatively similar to that of both Scratch and KlangText.

Figure 4.5 shows an example Klang program using the KlangVis notation. The example program shown here defines behaviour for a sprite actor called Ball. Ball has got a single event handler, that is set to be triggered when a GameStartEvent occurs. The event handler contains a forever loop that repeats a call to a function named print, which takes a string parameter. The string parameter is the concatenation of two smaller strings ”Hello” and ”, world!”. In other words, this program will continuously print the string ”Hello, world!” to the console.

The guiding principle behind the design of the visual syntax has been to make it impossible for syntax errors to occur. There is no way of assembling blocks that are incompatible with each other, such as creating a block that tries to subtract two boolean values. The vi-

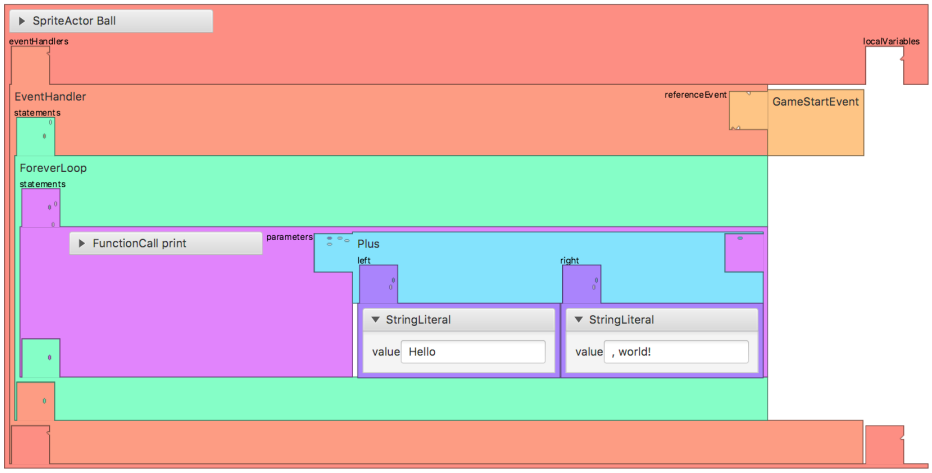


Figure 4.5: An example Klang program shown using the visual notation

sual editor relies heavily on static analysis to avoid any syntax errors, and will in stead of providing error messages simply refuse to connect incompatible blocks.

The visual syntax is designed to be relatively consistent with the textual syntax. Blocks that are assembled horizontally are mostly equivalent to textual code written on a single line, and blocks assembled vertically are equivalent to several lines of code. There are however a few exceptions here, due to some limitations in the visual editor that KlangVis is built on.

Despite some rudimentary customization of the editor, the visual syntax still relies heavily on the defaults of the original editor. There is definitely room for improvement in the visual syntax in terms of color coding, and the layout and sizing of blocks. This will be elaborated further on in later chapters.

KlangText - textual syntax

Just as the design of KlangVis, the design of KlangText takes multiple considerations into account. It is important that making the transition to KlangText from KlangVis should be as simple as possible. The textual syntax is designed in a way that borrows principles from high level textual languages such as Python. It is intended to be as simple and concise as possible, utilizing few keywords and few special characters.

KlangText is a whitespace aware language, meaning that it puts syntactic value into whitespace characters, that is, characters that are displayed simply as empty space in most editors. Examples of whitespace characters are the newline character, tabulation, and single spaces. Many traditional programming language simply ignore whitespace characters, which gives the programmer some additional freedom in terms of formatting their code using empty space. However, giving the users freedom to format code however they like is not always a good idea. Experience shows that many children do not care to properly format their code unless forced to do so, and therefore write mangled and hard to read code. KlangText forces the user to write relatively structured code, and can at the same time enforce formatting rules that make it easier to see the resemblance between KlangText and KlangVis.

As previously mentioned, KlangText draws inspiration from other languages, and especially Python. Python seeks to be an easily readable language, and often favours keywords over punctuation. This is also the case for KlangText. Among other things, it provides boolean operators such as ”and” and ”or”, as opposed to ”&&” and ”||”. This way it seeks to become more easily readable and less obscure.

```
sprite Ball
    when game starts
        forever
            print("Hello" + ", world!")
```

Figure 4.6: An example Klang program shown using the textual notation

Figure 4.6 shows the same program as Figure 4.5 using the textual notation as opposed to the visual one. The semantics are therefore naturally completely identical. The first line of the code defines a sprite actor called Ball, that has an event handler that is set to be triggered when the game starts. The event handler executes a forever loop that repeatedly concatenates the strings ”Hello” and ”, world!” and passes the resulting string to the function called print.

4.1.2 The game development framework

The game development framework is tightly integrated with the Klang programming langauge. It defines what which actions the actors can perform, such as moving around, rotating etc.

The game development framework is built on top of a existing UI framework, called JavaFX and an existing physics engine called JBox2d. The framework wraps around these to existing libraries and exposes functionality that is similar to that exposed by the Scratch framework. However, since it is built on top of a proper physics engine, it is also capable of doing more advanced simulations out of the box.

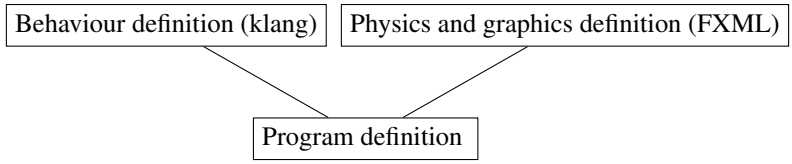


Figure 4.7: The composition of a Klang program.

The work flow of using prototype is aimed to be similar to that of Scratch. In Scratch actors are created simply as visual sprites on the screen, and then their behavior is defined by writing code. The prototype takes the same approach by first letting the user define the visual and physical properties of an actor, and the define its behavior in the Klang language. The prototype lets the user use FXML for defining the actors of the program. FXML is a language designed for creating UI for Java applications, however, this project extends FXML by also letting you define the physical properties of each visual element. Programs written using the prototype are therefore divided among two files, the FXML file and the Klang file, as shown in Figure 4.7. FXML is an existing technology, and there is existing tooling in place for generating FXML, such that you do not have to write FXML by hand.

This approach draws inspiration from other projects, such as Scratch, Logo, Greenfoot etc, which all provide a simple and out of the box way of doing computer graphics. In addition to computer graphics, this framework also provides a default mechanism for physics simulation, which hopefully provides an additional incentive for using the system.

4.2 Development process

The prototype presented in the previous section is a relatively complex piece of software and its implementation relies on many interconnecting parts. The development of such a system with several tightly coupled components is prone to mistakes, such as design decisions that seem optimal when looking at one part of the system, but fall short when the other parts are taken into consideration. It is therefore important to ensure that not too much time is spent looking at a single isolated component of the system, at least not without properly planning its implementation in advance.

Planning the implementation of a system in advance is often quite challenging, and unforeseen problems have a tendency to appear during implementation. This is especially true in situations where the developer is relatively unfamiliar with either the technology stack used or the type of system to be created, which has been the case for me during this project.

During this project these issues have been mitigated by combining thorough planning up front with an active policy of switching between working on each part of the system. The initial planning set out to uncover which features needed to be supported by the system, that is, which language constructs were needed, and the outline of the features provided by the development framework. It was decided early on that the system should be capable of doing many of the same things that Scratch does, in order to build further on the foundation laid down by Scratch.

However, Scratch is a vast system and implementing all of its functionality is far beyond the scope of this project. Therefore a selection has been made from the functionality available in Scratch. This was a two-step process, first a selection of Scratch programming exercises was chosen, proceeded by a process of deriving functional requirements based on what was needed to solve the exercises.

The selection of programming exercises contained tasks that were all relatively different and that had been used for a long time in the Kodeklubben Trondheim Scratch courses. Kodeklubben Trondheim is a part of a national initiative in Norway, that is simply called Kodeklubben. The programming exercises chosen were not exclusively used in Trondheim, but had also been used around the country. Despite the fact that I am quite

experienced when it comes to teaching programming to children using Scratch, there was a need for a second opinion when making the selection of exercises. It was important that they were representative of the most important features of Scratch. Therefore an expert on the matter and creator of many of the exercises used by Kodeklubben was contacted. Using his input I was able to make a selection of four exercises that were deemed varied enough to give an insight into the core functionality of Scratch.

The selected exercises were then analysed and the features needed to solve them were noted and written down as the initial functional requirements for the prototype. The requirements were then given a priority and categorized. The requirements that were given the highest priority were those concerned with responding to different types of events, and basic movement, such as translation and rotation.

Chapter 5

Artifact Evaluation

This chapter presents the details of evaluation the artifacts built during this project, including the planning and execution of a focus group session, a presentation of the results, as well as a discussion of their implications.

5.1 The Focus Group

Before initiating a data collection process it is important to have a clear understanding of what one seeks to uncover. There are several pitfalls that are common to data collection in design science, one of which is to gather data primarily about the prototype implementation and fail to shed light on the research questions. For this reason this project has had a clear focus on trying to gather data that is relevant to the research questions, such as the impact of each of the core design decisions. Aspects such as the effects and implications of having a single programming language with two notations would have to be explored. However, this thesis presents what is a single iteration of the design science cycle. This iteration should, in addition to providing preliminary answers to the research questions, provide useful data that can be used to prepare for future iterations. This includes implementation specifics details that can be used to improve the prototype.

5.1.1 Participants

The decision of choosing participants for a focus group is one of great importance and has a major impact on which data you receive and how that data should be treated. There were two primary options considered for this project, using domain experts, and using intended end users of the system. It was decided to use intended end users for this project.

The participants of the focus group were chosen based on their previous programming experience. It was important to ensure that the participants were in fact the intended users of the system, children making the transition from visual to textual programming. The children chosen were all participants of a Java programming course at Kodeklubben Trondheim. The Java course is the most advanced course that Kodeklubben Trondheim provides, meaning that its participants were likely to have both previous experience using Scratch, Python and Java, which meant that they would probably have insight into the problem addressed by this thesis. It should also be mentioned that the participants had previous experience using FXML, on which the game development framework is built.

In advance of the focus group session the parents of the children were contacted by email and asked for permission to use them in the research. None of the children were denied participation, however, some did not respond. The children that I had not gotten permission to use were naturally excluded from the focus group session.

5.1.2 Roles

Using a focus group requires that someone leads and moderates the discussion. It was natural for me to claim this responsibility for myself, as I had the most insight into problem addressed by this thesis and which aspects that I wanted to gain further insight into.

It was decided not to do any form of recording during the session, and rely solely on taking notes. If I were to attempt to both moderate the discussion and rely exclusively on my own notes, a lot of useful data would probably be lost before making it into the notes. For this reason one of my co-founders of Kodeklubben Trondheim was recruited for the purpose of taking additional notes. He was chosen as he had insight the problem addressed by this thesis, and would be able to differentiate between useful and less useful data.

5.1.3 Recording the Focus Group

Video or audio recording the event was considered, but was discarded as an option due to the fact that it might create a more tense environment and lead to children not wanting to attend, or not feeling comfortable enough to speak freely. Therefore it was decided to rely on taking notes. The notes that were taken during the focus group would include direct quotes of what was said by the participants, but also observations of their emotions, such as level of enthusiasm etc.

5.1.4 Time and Place

The focus group participants all attended the same programming course at Kodeklubben Trondheim. It was therefore both convenient and beneficial to the research to perform the focus group session during a regular course night. This way the children would hopefully feel more comfortable and familiar than they would have at some other location, and they did not need to change their schedules in any way.

In order to ensure that the children got a chance to know me and my assistant in advance of the focus group session it was also decided that we would step in as teachers for the Java course this night. This way we would be able to interact with the children and hopefully ensure that they would be comfortable speaking freely during the focus group session. This was an important aspect as the session was used to evaluate the artifacts, and for the data to be useful the participants would have to be comfortable enough to give both positive and negative feedback, and bring up new ideas.

5.1.5 Focus Group Plan

The general outline of the plan for the focus group session was as follows:

1. Warm-up questions
2. Discussion of the transition from visual to textual programming
3. Discussion of using a programming language with two notations as a solution to the problem

4. Interactive demonstration of the prototype

5. Discussion of the prototype

There were two levels of questions prepared, initial open ended questions and sets of optional and more specific follow up questions that would be used in situations where the discussions needed some extra help to get started. The questions can be found in the appendix.

Warm-up Questions

Despite being able to interact with the participants in advance of the focus group session, it was assumed that a couple of warm-up questions would be beneficial. Initiating the session would have to include rearranging the seating and would create a clear break from the regular events of the programming course. Ensuring that any tension built up during this process would be alleviated before moving onto the important questions was a priority.

For this reason a couple of warm-up questions were prepared. These were mostly simply yes-no questions, and served only to reaffirm some basic assumptions, lighten the mood, and make sure that everyone got used to speaking up in the group.

Discussing the Transition

This project addresses a problem that is of relatively complex nature, and it would be reasonable to assume that the participants could have fundamentally different views on the subject than my own. Therefore a set of questions were prepared that sought to explore the participants experiences of moving from visual to textual programming. These included questions that sought to explore whether or not they actually found the transition to be troublesome, and their reasons for thinking that it was or was not so.

Discussing the Approach

The artifacts developed throughout this project represents a specific approach to solving the problem at hand. Evaluating this approach in general was important to provide useful answers to the research questions.

For this reason a set of questions were prepared that explored the participants thoughts about the approach that this project has taken. The questions mostly revolved around their thought around having a programming language with one visual and one textual syntax. These questions were to be posed without informing the participants of the fact that this was an approach that I had taken, in an attempt at not guiding them into being overly positive. In advance of these questions the children were given an explanation of the difference between syntax and semantics, and shown examples of how the same semantic construct could be present in different languages with different syntax.

Demonstration

Naturally, the data collection process for this project would have to narrow in on the developed artifacts. It was decided that the participants would be given an interactive demonstration of the prototype system. Therefore a set of programming exercises were prepared that would be solved in collaboration between myself and the participants. The prepared exercises were such that the participants should be able to apply the same type of reasoning as in Scratch when solving them. I would take care of the direct interaction with the system, and write the code that they suggested.

The demonstration would serve two purposes, giving the children insight into the prototype, which would lay the groundwork for further discussion, but also seeing how capable they were of using the system to solve tasks, to which degree they were able to apply their experience from Scratch to solve tasks etc.

Discussing the Prototype

The demonstration would be followed up with a discussion of the prototype. This way flaws in the implementation could be uncovered, and additional light could be shed on the core design decisions. The questions sought to explore to which degree the system felt familiar and intuitive to them, and whether or not they were able to see resemblance between Klang and other languages. The final questions were concerned with room for improvement within the prototype, both in terms of the language and the programming environment in general.

5.2 Analysis

Template analysis was used as the primary technique for data analysis for this project. The data collected through the focus group was in the form of a collection of notes containing concrete quotes from the participants, paraphrases and general observations. The goal of the analysis was to uncover general trends in the data, such as which design decisions the participants were approving or disapproving of, to which degree they were able to understand the different aspects of the system etc.

The first phase of template analysis is to create a template of categories that the data is inserted into. The template created for this project was based on the set of transitions shown in Figure 4.2. These transitions from Scratch and all the way to Python are the underlying essence of this project, and they all need to be taken into consideration. It was therefore natural to group the data in this way, and then proceed to analyze the data in each category. However, the categories were not considered as a partitioning, and the same data points could be put into more than one category.

1. Scratch to KlangVis
2. KlangVis to KlangText
3. KlangText to Python

Proceeding the categorization each category was examined in isolation and the data was interpreted in the context of the core design decisions of the artifacts. The data assigned to each category was first skimmed through in order to get an initial impression of the data in each category, before moving on to more thorough analysis.

By skimming the data a couple of patterns emerged, these were patterns were used to do some rudimentary color coding of the data. This color coding was used as an indication as to which aspects were most important in the context of each category.

5.3 Results

This section will present the results found by analyzing the data. The results are presented in the context of the transitions shown in Figure 4.2.

5.3.1 From Scratch to KlangVis

The transition from Scratch to KlangVis is intended as the first step in the process from visual to textual programming. This transition is key to whether or not the users would be interested in even using the system. The data shows a couple of clear patterns.

The first of these patterns is that the participants find Scratch to be somehow childish and not to be considered as proper programming. They also refer to programming in Scratch as ”just assembling things” rather than actually creating something. They express a desire for more features to be available in languages such as Scratch, and find the idea of being able to do physics simulations very appealing. A simple demonstration of the physics capabilities of the Klang language was literally met with applause.

However, the participants found that Scratch was an easy language to learn. It did take some getting used to, but that was expected as it for most of them was their first programming language. They were unable to come up with anything in particular that they remembered as hard to learn in Scratch. They found both the syntax and semantics of the language to be relatively simple to grasp. They were also very aware of the fact that Scratch did not produce any error messages, and ”just worked”. Error messages in Python and Java were considered as hard to read and a source of much frustration.

The participants were easily able able to see the resemblance between the semantics of Klang and Scratch. The actor based runtime model seemed familiar to them and they recognized the concepts that were common to the two languages, such as sprites, event handlers, and loops. However, there was some criticism of the syntax of the visual notation, KlangVis. Despite being able to understand the code by looking at it thoroughly, they found that it was not properly organized. They requested changes such as a more understandable pattern in how the blocks were connected as they could not detect a proper pattern in how it was done. In addition, improvements of the visual representations of the blocks were requested, such as clearer edges around the blocks, and larger text labels on each block.

The participants were concerned with the UI of the visual editors of both Scratch and KlangVis. Both languages provide a palette of blocks that can be dragged into the code editor. They found that having to look for blocks in a palette was too cumbersome and

wanted to be able to search for the blocks they wanted. Scratch divides the blocks into categories in the palette, while KlangVis has only got a single flat list in the palette. The absence of categorization was pointed out as a major flaw in its design, as too much time had to be spent looking for the right block in the palette. It should be mentioned that the editor that KlangVis is built upon supports this kind of categorization, and that implementing such a feature in future versions of KlangVis should be trivial.

5.3.2 From KlangVis to KlangText

This transition is at the core of this thesis. This is were the actual transition from visual to textual programming takes place.

The participants did all express that the transition from visual to textual programming was a difficult one. They struggled with things such as remembering the exact syntax of different statements and expressions, and found that it was often the case that they remembered how a particular concept worked, but not how to write it.

The data shows that the children did not find the semantics of the Klang language to be particularly hard, and they were able to see the resemblance to Scratch. The observations made during the demonstration show that the children are able to apply the same reasoning was they would in Scratch to Klang when solving problems. Understanding the difference between syntax and semantics did not strike them as difficult, and understanding that a language could have two different notations was an easy concept to grasp.

Different ways of using a system such as Klang was also present in the data. The children discussed how they could use the palette from the visual language to find concepts that they did not remember how to write, and use it to generate the textual code that they wanted. There is a general optimism expressed towards a system with a visual and a textual notation present in the data, and the participants found that it would be a useful tool for mastering visual syntax. However, there was also an expressed desire for this kind of translation mechanism between existing languages, such as Scratch and Python. The participants pictured being able to generate Python code from Scratch blocks etc.

There was also expressed opinions that the syntax of KlangText was simple to read and understand. The children were unified in the opinion that the syntax of Python was

easier to get used to than that of Java. In general they wanted as few special characters as possible in the language, such as curly brackets etc.

5.3.3 From KlangText to Python

The transition from KlangText to existing textual languages such as Python is the final step of the process shown in figure 4.2b. It is a transition that is intended to occur after the actual transition from visual to textual programming, however, it holds importance as Klang is designed to be a stepping stone for other languages.

The data shows that the participants found that it was hard to learn new concepts in textual GPLs. It was not necessarily the semantics of the new constructs that were hard to understand, but rather remembering how they were written. The participants showed a desire for the generation of code to existing languages such as Python or Java. Particularly there is an expressed opinion that these languages are hard to get used to, and learning both new syntax and new concepts is hard.

Participants also found that it was harder to get used to the syntax of Java than Python. They found the syntax og KlangText, which is inspired by Python, to be quite simple and easy to read. They saw the commonalities between KlangText and Python and saw the system as a useful stepping stone.

5.4 Discussion

The answers provided by this thesis should be considered as preliminary, however, treated as such they are seemingly trustworthy. This project has completed what can be considered a single iteration of the design science cycle. There is a need for more iterations to be completed before any definitive answers can be provided.

The greatest limitation of the data is that it has not been collected on the basis of the users interacting with the prototype directly, but rather by having it demonstrated. There is a major difference between using a system and having it demonstrated. Despite the fact that the focus group participants did provide reasoning and participated in solving simple programming tasks using the system, they were guided while doing so. There is reason to

believe that giving end users access to the prototype and letting them use it solve problems on their own would uncover flaws in its design, or shed light on new parts of the problem domain.

The data collected and answers provided are therefore to be considered as preliminary, rather than definitive and should be treated as such. However, given its status as preliminary the data is trustworthy. There are several pitfalls associated with collecting data through the use of focus groups, especially when using children as the participants. One of the major pitfalls is to fail to create an environment where the participants feel free to express their opinions, where the participants end up either providing too shallow answers or answers that they assume would please the researcher. Before conducting data collection for this project a lot of effort was put into creating an environment where the participants would feel comfortable and able to express themselves. This effort was seemingly successful, and resulted in a lively discussion that included both positive and negative feedback. The participants did not fear giving negative feedback, which can clearly be seen from looking at the data concerning the visual syntax of KlangVis.

However, another possible pitfall of interview techniques, such as the focus group, is the elite bias. Elite bias is a term that describes the situation where the data has not collected on the basis of a representative selection of interview subjects, but rather favours an elite group, such as an above average well-informed group of people. The elite bias can lead to data that fails to capture the broader picture (Myers and Newman, 2007). The participants used in the data collection for this project were all participants who attended the Kodeklubben Trondheim Java course, which is our most advanced course. There is reason to believe that these children belong to an elite group of young programmers, and that the data fails to capture the needs of less skilled children.

5.4.1 Research question 1

The first research question posed by this thesis was ”How can a textual and a visual programming language be combined to ease the transition from visual to textual programming?”.

The general approach taken by this project was to create two programming languages

with common semantics, one visual and one textual, that enabled simple and direct translation between the two. The intent was to create an environment where visual and textual programming could be used interchangeably, and that could isolate the syntactical transition from a semantic transition. The language was built on the assumption that making the transition from visual to textual programming was made even more difficult as it was often coupled with a transition from DSL to GPL.

The results clearly indicate that this approach can be a viable solution to this problem. The core concept of having two notations for the same underlying language appeared to be easy for the children to grasp and the work flow of using the system was immediately understood. The children were able to picture useful ways of utilizing the system to ease the transition, by addressing what they found to be the core of the issue, which simply was to remember the textual notation for each language construct. The children pictured using the visual syntax as a fallback mechanism in cases where they did not remember a specific part of the textual notation.

The results also confirm that the transition from visual to textual programming is a challenging one, and that these are related to both syntax and semantics. The participants expressed that learning textual syntax was challenging, and even more so when combined with learning new semantic concepts. However, recognizing common semantic constructs between languages was deemed to be less of a challenge. This confirms my assumption that the transition from visual to textual programming should be kept separate from the transition from using a DSL to using a GPL.

The data also sheds some light on how a system such as this one should be designed. Klang heavily borrows semantics from the Scratch programming language, the reason for which is two-fold, minimizing the effort needed to move from Scratch to KlangVis, and making sure that the semantics of Klang are generally easy to grasp. The results show that in this respect the prototype was successful. The children easily recognized the concepts from Scratch in Klang, and were able to apply the same type of logic and reasoning that they had gotten used to in Scratch. Defining behaviour for visually represented actors was familiar ground, and hence easy to grasp.

There were issues related to the design of Klang as well, however, not related to se-

mantics. The design of the visual notation, KlangVis, was highly criticized. The children found the visual representations to be too similar to each other, their colors to be seemingly random, and the border around each one to be too diffuse. In addition, they struggled to see a clear pattern in how blocks were organized, some were connected horizontally, some vertically. These two aspects combined resulted in a visual syntax where blocks blended into each other, and the meaning of sequences of code became unclear.

The data also confirms that errors produced in programming languages is a large source of issues for children. Reading and interpreting error messages is hard, the complete lack of error messages in Scratch was something that the participants missed when using textual languages. Klang is not able avoid all errors, due to the fact that it is in part a textual language, which makes avoiding syntax errors close to impossible. However, avoiding runtime errors appears to be a good choice for programming languages aimed towards children.

The textual syntax received almost exclusively positive feedback. There was an expressed desire for a textual syntax that used as few special characters as possible, which was a guiding principle when developing KlangText. The children expressed that the syntax of Python was easier to master than that of Java, and that drawing inspiration from Python as opposed to Java was a good decision for this type of system. However, it should be mentioned that the children also expressed that they found the syntax of Java to be more structured and reliable over time.

In summary, the results confirm that the approach of creating a language with both a visual and textual syntax is a good approach to easing the transition from visual to textual programming. Isolating the transition from visual to textual programming from the transition from a DSL to GPL also seems beneficial. Borrowing semantics from languages such as Scratch also appears to be a good choice, as it makes it easier to transition to using the system. Errors should be avoided to the extent possible, and creating a visual notation that makes it impossible to create syntax errors is important. The visual notation should be thoroughly organized, using techniques such as color coding and categorization for code blocks.

5.4.2 Research question 2

The second research question for this project was ”How can a development environment aid the transition from visual to textual programming?”. This relates to how the programming environment can assist the transition from visual to textual programming, in a way that is not directly related to the programming language design.

The approach taken by this project was to create a game development framework that was tightly integrated with the language, and enabled the use of existing tooling for drawing graphics, as well as providing basic physics simulation capabilities. The intention was to provide a simple, yet powerful way of doing both graphics and physics.

The results clearly show that the focus group participants were enthusiastic about the possibility of being able to do basic physics simulations. Being able to physics simulations was expressed as something that they had previously wanted to do, however, they had not been able to. These capabilities clearly provide an additional incentive for using the system.

In addition to providing additional features to the system, the framework ensures a similar work flow in Klang as that of Scratch. Actors are first defined in terms of the visual and physical properties, then behavior is specified in the Klang language. This appears to be an intuitive way of programming for many children, possibly due to the fact that it makes programing less abstract. The code that is written is specifically linked to a visible entity that can be seen on the screen.

Chapter 6

Conclusion and Future Work

This chapter summarizes and reflects on the work done during this project, and presents the future work to be done.

6.1 Conclusion

Through this thesis I have explored the problem of children making the transition from visual to textual programming, and sought to find ways of easing this transition. The project has looked into other attempts at solving the same problem, and proposed a new solution. The solution has been implemented in the form of a working prototype. The prototype is a new programming language that is a DSL with two syntaxes, one visual and one textual. The programming language is tightly integrated with a game development framework that enables the users to write simple two-dimensional games. The prototype has been evaluated by arranging a focus group session where the intended end users have discussed the problem and my approach to solving it.

The main finding of this project is that the issues related to making the transition from visual to textual programming is that it is not only related to syntax, but also semantics. The visual programming languages used to teach programming to children are mostly DSLs, while the textual languages that they proceed to learning are often GPLs. This means that there are big semantic differences that the children need to get accustomed to

while at the same time trying to master textual syntax. The results indicate that this double transition is indeed an issue for young programmers.

The developed prototype tries to mitigate these issues by enabling the users to choose freely between visual and textual notation, without worrying about semantic differences. The results indicate that this is a viable solution to the problem. However, the answers provided by this thesis are to be treated as preliminary, and further research is needed in order to provide any definitive answers.

The main pitfall that this project fell into was that of not putting enough weight on the principle of ”just enough prototyping”. This means that one should not put more effort into a prototype than what is required to provide the answers that one seeks. During this project too much time has been spent developing parts of the prototype system that were never a central part of the artifact evaluation. The project would have benefited from a narrower scope on the prototype, which would have freed up time that could have been used to ensure better quality of the prototype and the data collection process.

6.2 Future Work

The answers provided in this thesis provide initial insight into the transition from visual to textual programming. However, the work presented by this thesis can only be considered as a single iteration of the design science cycle of build, evaluate, theorize, and justify. The work done lays a solid foundation for future research, and proceeding iterations.

The build phase of the next iteration should take into consideration the knowledge gained this far. The main drawback of the work done in this iteration is that the intended end users were never allowed to interact with the prototype directly. This was due to the prototype having usability issues that would have made direct interaction frustrating for the users, and would skew the results. These issues should be solved, and quality and robustness of the prototype should be ensured. This way the prototype can be evaluated in a more proper context and more definitive results can be provided.

There should also be steps taken to ensure that the chance of these types of issues surfacing again is minimized. There will be future changes to the design of the prototype, and its implementation should be able to withstand changes without introducing too many

new and undiscovered bugs. The current implementation is modular and easily adaptable to change, however, it has poor test coverage. In order to make sure that bugs introduced by new changes are captured early on, there should developed a thorough test suite that covers the core aspects of the language.

The data collected in this thesis is also a possible victim of the elite bias Myers and Newman (2007). Future iterations should take care to avoid this pitfall, and try to reaffirm that the results presented here can be reproduced. The prototype should be tested by children belonging to a broader group than the participants of the Kodeklubben Trondheim Java course.

The next iteration should also explore in deeper detail how a visual syntax should be designed. The results show that there is a lot of room for improvement in the visual syntax of KlangVis, the design decisions of other visual notations should be studied in greater detail and used to create a new and improved prototype.

Bibliography

Cheung, J. C., Ngai, G., Chan, S. C., Lau, W. W., 2009. Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students. In: ACM SIGCSE Bulletin. Vol. 41. ACM, pp. 276–280.

Commission, E., 2016. Grand coalition for digital jobs.
URL https://ec.europa.eu/digital-single-market/en/grand-coalition-digital-jobs

Erwig, M., Meyer, B., 1995. Heterogeneous visual languages-integrating visual and textual programming. In: Visual Languages, Proceedings., 11th IEEE International Symposium on. IEEE, pp. 318–325.

Hungnes, O., 2016. Jigsaw emf editor. Master's thesis, Norwegian University of Science and Technology.

King, N., 1998. Template analysis.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E., 2010. The scratch programming language and environment. ACM Transactions on Computing Education (TOCE) 10 (4), 16.

March, S. T., Smith, G. F., 1995. Design and natural science research on information technology. Decision support systems 15 (4), 251–266.

Marshall, T. H., 1950. Citizenship and social class. Vol. 11. Cambridge.

Mossberger, K., Tolbert, C. J., McNeal, R. S., 10 2007. Digital Citizenship: The Internet, Society, and Participation (MIT Press). The MIT Press.
URL http://amazon.com/o/ASIN/B001949SXG/

Myers, B. A., 1986. Visual programming, programming by example, and program visualization: a taxonomy. In: ACM SIGCHI Bulletin. Vol. 17. ACM, pp. 59–66.

Myers, M. D., Newman, M., 2007. The qualitative interview in is research: Examining the craft. Information and organization 17 (1), 2–26.

Pea, R. D., 1983. Logo programming and problem solving.[technical report no. 12.].

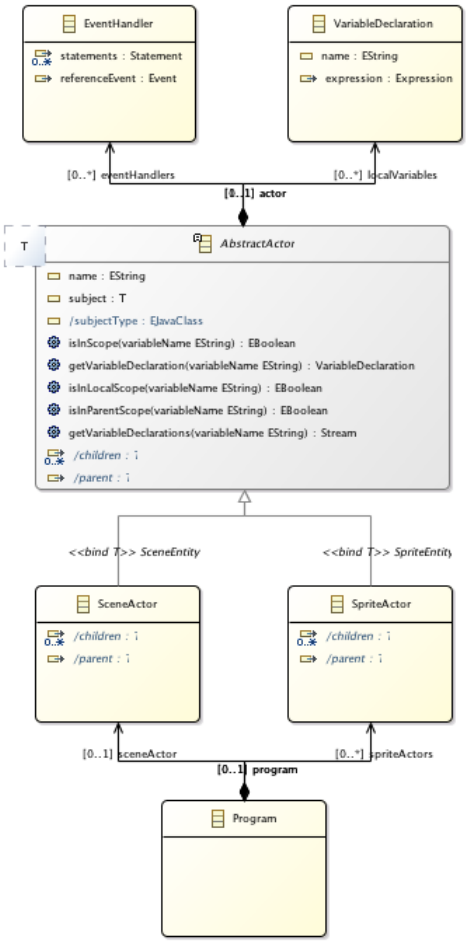
Sofaer, S., 1999. Qualitative methods: what are they and why use them? Health services research 34 (5 Pt 2), 1101.

Stewart, D. W., Shamdasani, P. N., 2014. Focus groups: Theory and practice. Vol. 20. Sage publications.

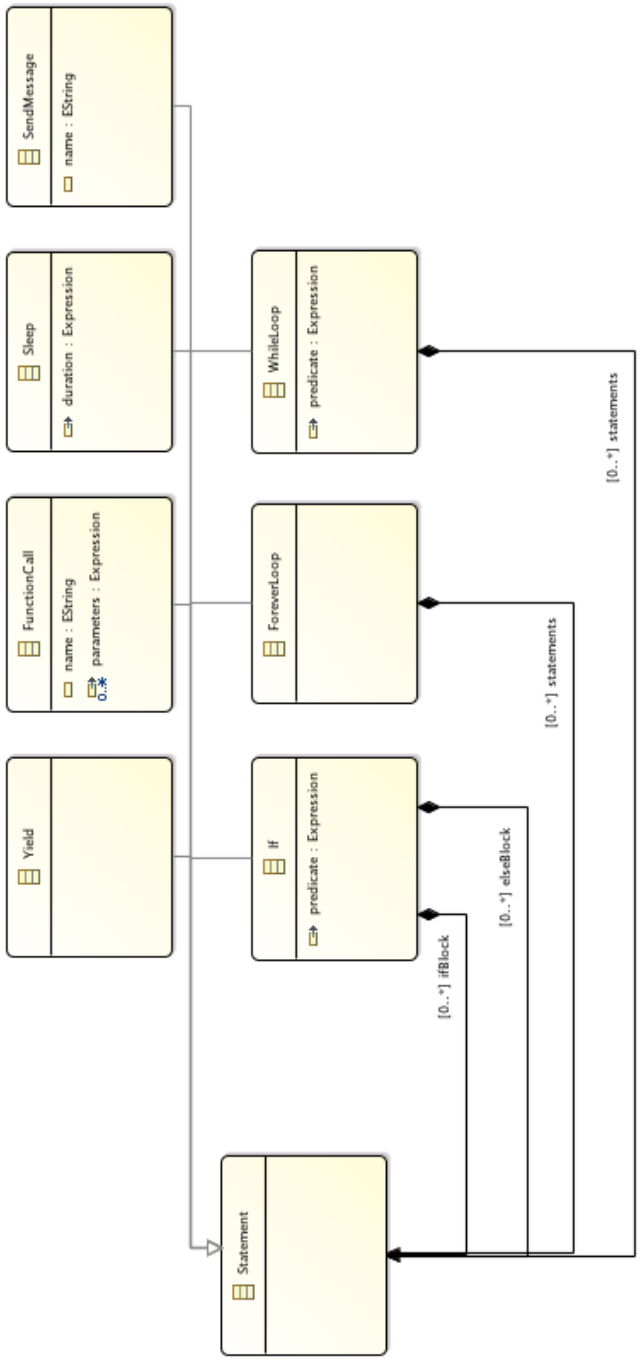
Appendix

6.2.1 Klang UML Diagrams

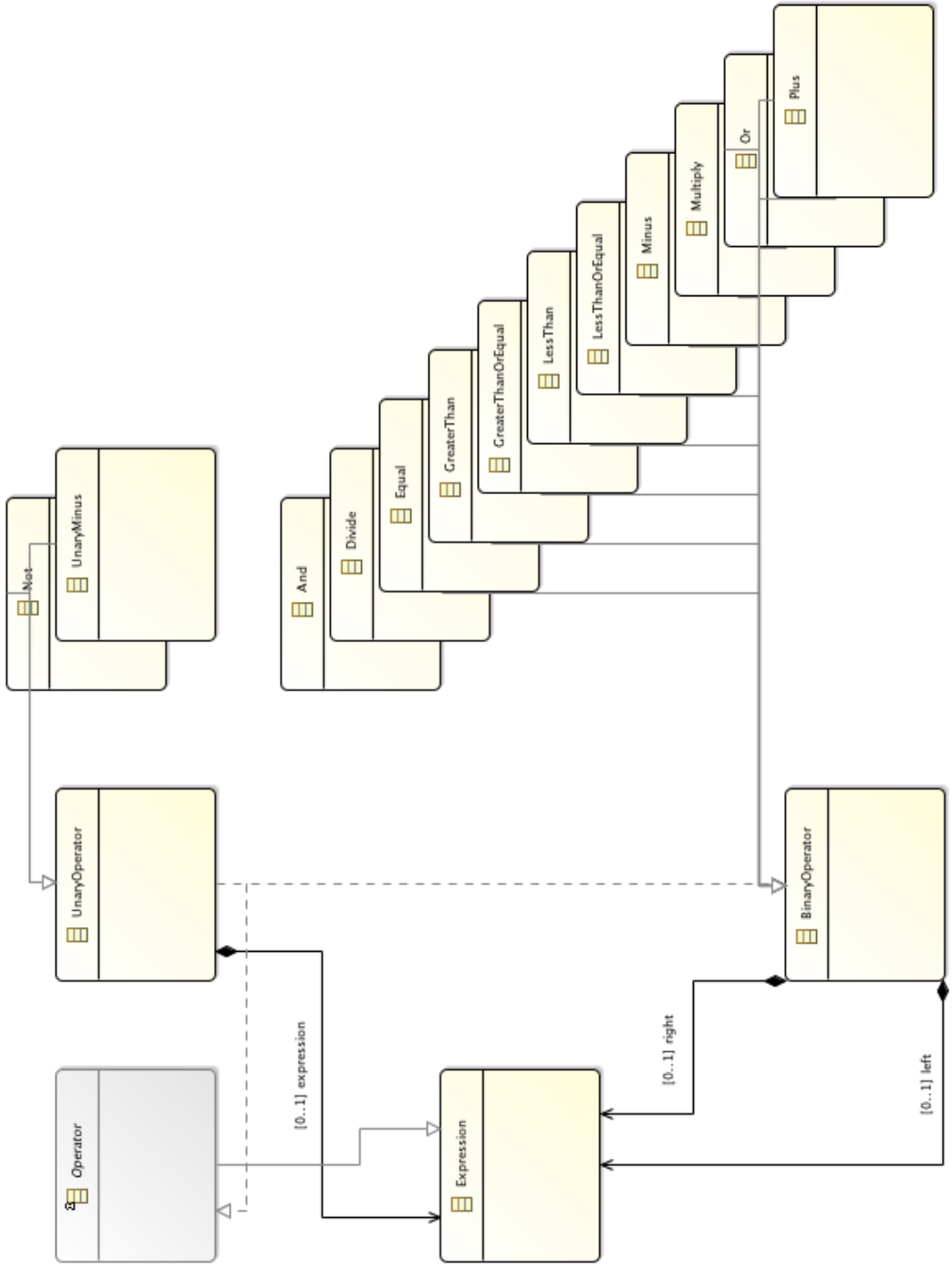
Language Core



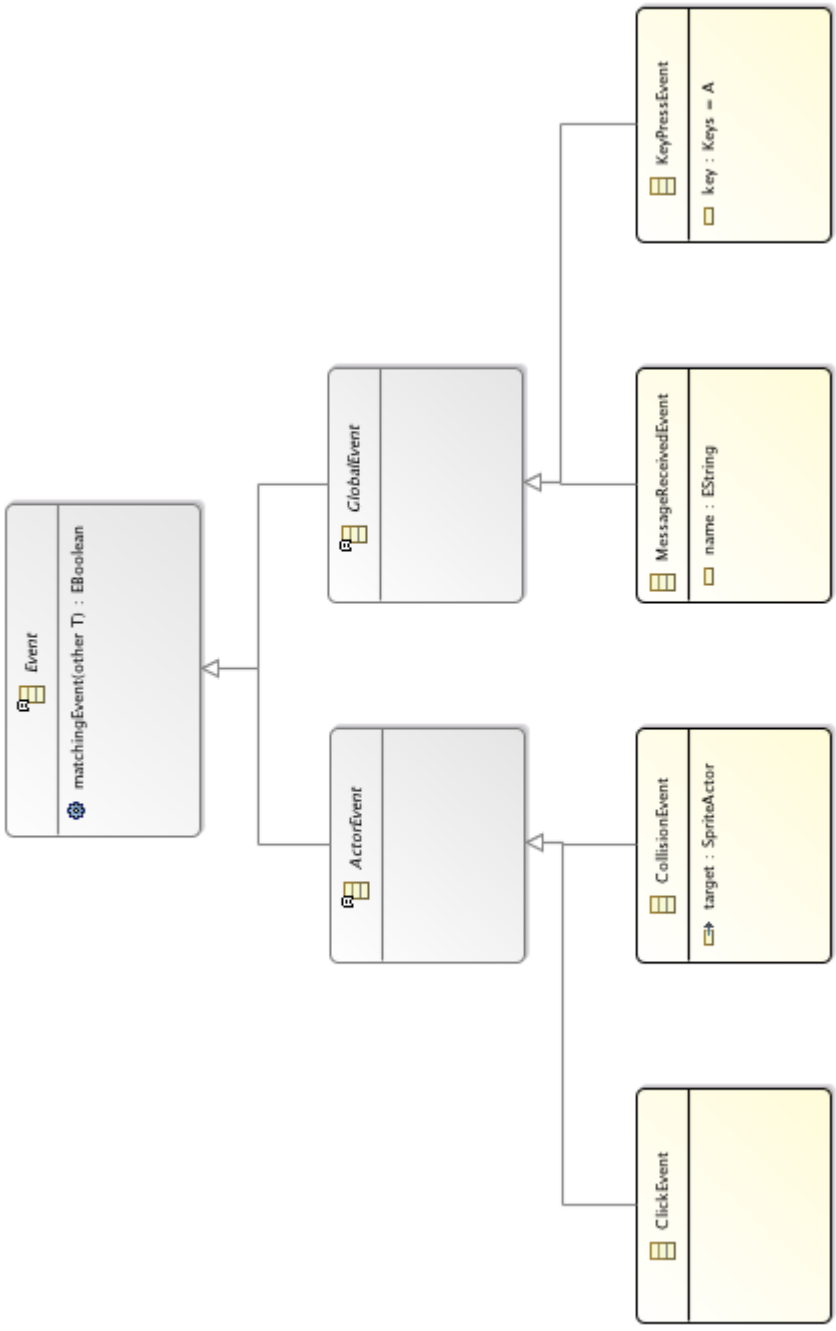
Statements


SendMessage
name : EString
Sleep
duration : Expression
FunctionCall
name : EString
parameters : Expression
0..*
Yield
WhileLoop
predicate : Expression
ForeverLoop
If
predicate : Expression
Statement
[0..*] statements
[0..*] statements
[0..*] elseBlock
[0..*] ifBlock

Expressions



Events


Event
matchingEvent(other T) : EBoolean
GlobalEvent
ActorEvent
KeyPressEvent
key : Keys = A
MessageReceivedEvent
name : EString
CollisionEvent
target : SpriteActor
ClickEvent

6.2.2 Focus Group Questions

1. How many of you have tried programming in Scratch?
2. How many of you have tried programming in Python?
3. How many of you have tried programming in Java?
4. How would you describe programming in Scratch?
   (a) How was it getting started using Scratch?
   (b) What did you find challenging in Scratch?
   (c) Is there anything you feel is missing from Scratch?
5. What was it like going from using Scratch to using text based languages such as Python or Java?
   (a) What were the greatest challenges when you were trying to get accustomed to Python or Java?
   (b) Is there anything from Scratch that you miss when you're using Python or Java?
   (c) Did you recognize concepts from Scratch in Python or Java, such as if-statements?
   (d) What was the greatest challenge when going from visual to textual programming?
6. How do you feel about Python's syntax versus Java's syntax?
7. Imagine being able to write textual code in Scratch. That is, you could decide whether or not to use visual blocks and textual code. What do you think that would be like?
   (a) Do you think it would make it easier to get used to writing textual code?
8. I've created a new programming language that is both visual and textual. That is, the same source file can be opened either as a text file, or it can be opened in a visual editor with blocks etc. How do you feel this kind of system could impact going from visual to textual programming?

9. What are your initial thoughts on this system (prototype)?
10. Is it easy to understand that the two languages are actually one and the same?
11. Do you have any suggestions as to how the visual language might be improved?
12. Do you have any suggestions as to how the textual language might be improved?
    (a) The syntax of the textual language resembles that of Python more than that of Java. What do you think about this decision?
13. Both languages resemble Scratch in one way. All the code that you write is linked to one specific visual sprite on the screen and is written in terms of ”when this happens, do this”. What are your thoughts on this way of writing programs?
14. What are your thoughts on being able to do simple physics simulations straight out of the box, such as this?
    (a) Is this something that you have wanted to add to your previous projects?
    (b) Do you think this could make programming more rewarding or exciting?
15. Any final thoughts?

6.2.3 Source Code

The source code can be found at `https://github.com/kwrl/masteroppgave`. The majority of the code found here has been produced by me, however, there are some exceptions. The game development framework, jbox2ddemo, has largely been developed by my supervisor, and can also be found at `https://github.com/hallvard/javafx`. The reason why the development framework is included in my own repository directly is that I have made changes to the source code that should not be merged back into the original repository. The visual editor used by Klang can be found at `https://github.com/hallvard/emfblocks`.