

BACHELOR THESIS
COMPUTING SCIENCE



RADBOUD UNIVERSITY

Digital flowchart maker

How do we aid the learning of programming?

Author:

Frank Gerlings
s4384873

First supervisor/assessor:

J.E.W. (Sjaak) Smetsers
S.Smetsers@cs.ru.nl

Second supervisor/assessor:

MSc T.J. (Tim) Steenvoorden
T.Steenvoorden@cs.ru.nl

December 12, 2018

Abstract

Learning the basics of programming can be hard. Worldwide efforts try to teach people programming but there still is an ongoing shortage of programmers. This thesis tries to make it easier to understand the basic concept of control flow.

We will start by identifying a way to separate programming into algorithmic thinking and coding by using flowcharts and argue that making flowcharts can be made easier by constructing them digitally. We will ask ourselves what a flowchart creating tool should look like, delve into three different designs for such tools and finally choose one and work out a tool around it.

To see how well our tool supports the process of learning how to program, we will test our tool by observing and interviewing students while they use it. In the results we will see that adding and deleting nodes and choosing specific node types goes fairly well. However, we will see that editing content of nodes is hindering the students. These problems seem to originate from technical limitations, rather than design flaws.

Acknowledgements

First off, I would like to thank Sjaak Smetsers and Tim Steenvoorden, my research supervisors, for their ceaseless help throughout the project. Even though the project took longer than expected, they never hesitated to show me how to do research and write a thesis.

Also, I would like to thank Renske Smetsers-Weeda, the students and in extent the Montessori College for testing the tool and giving me the valuable feedback that you are about to read.

Another group of people that I owe my thanks to are all the people that worked with me on a daily basis at the university. For their occasional welcome distraction and their listening ear.

And finally, I want to thank my mom and dad for being there for me.

Contents

1	Introduction	3
2	Background	4
2.1	The Problem: Learning to Program	4
2.2	The solution: Flowcharts	5
2.3	Digital Flowcharts	5
2.4	Drawing Flowcharts	6
2.5	Nielsen's Heuristics: Guidelines to Usability	7
3	Tool development	10
3.1	Requirements	10
3.2	The programming language	11
3.2.1	JavaScript	11
3.2.2	Elm	12
3.2.3	Conclusion	12
3.3	Designing Flowchart Manipulation	12
3.3.1	Selection bar	13
3.3.2	Drag and drop	13
3.3.3	Overlay buttons	14
3.3.4	Conclusion	15
3.4	Implementation	15
3.4.1	Model	15
3.4.2	Update	17
3.4.3	View	19
3.4.4	Technical limitations	19
3.4.5	Other functionalities	21
4	Methodology	22
4.1	Research question	22
4.2	Measuring environment	22
4.3	Empiric variables	24
5	Results	26

6	Discussion	29
6.1	Interpretation	29
6.1.1	Add and delete nodes	29
6.1.2	Choose node type	30
6.1.3	Edit content	31
6.2	Remarks	33
7	Conclusions	34
7.1	Related Work	34
7.2	Future work	35
8	Appendix	39
8.1	Interview questions	39
8.2	Interview transcriptions	39
8.2.1	Group 1	39
8.2.2	Group 2 and 3	41
8.3	Video quotes	44
8.3.1	Pre-test: Group 2	44
8.3.2	Pre-test: Group 3	44
8.3.3	Main test: Group 2	44

Chapter 1

Introduction

A study by Smetsers and Smetsers[1] developed a course called *Algoritmisch Denken*, where students need to develop and implement algorithms in a Java¹ environment. During the course algorithmic assignments are posed to students. The problem solving of these assignments is divided in two parts: first the students work out a solution on paper by drawing a flowchart. Next, they convert the flowchart to Java-code to see if their answer is correct. The idea behind this was to separate the semantics and the syntax of the algorithm, in order to reduce the cognitive load placed on students, more on this later. However, they found some trouble in letting the students create flowcharts analogous, which is why in this thesis we will search for a way to improve that by building a tool that lets the students create flowcharts. To put it directly, we will try to answer the next question:

What does a digital flowchart creating tool looks like, such that it supports learning to program the best?

To this end we will look at different designs, explore its implementation and finally test it and draw conclusions. The full code of the tool can be found online².

¹Java, Oracle <https://www.java.com/nl/>

²Flowchart tool - Frank Gerlings https://gitlab.science.ru.nl/gerlings/Thesis_Sources/tree/master/flowcharttool

Chapter 2

Background

First we will look into what programming is and why it is hard. Next, we will find an answer by using flowcharts, after which we will elaborate on the technical details. Then we will see why we will build these flowcharts digitally rather than on paper. Lastly we will see that there are certain criteria, better known as heuristics, that lead to a successful digital flowchart maker.

2.1 The Problem: Learning to Program

Programming is the confluence of algorithmic thinking and coding. This confluence makes programming hard and tends to lead to confusion for novice programmers[2]. The two main struggles are the mathematical challenge that algorithmic thinking poses and the syntactical knowledge that is needed for coding.

The mathematical challenge of designing an algorithm is larger than normal with programming because students are also facing syntactical errors from the coding. These errors put students in an act-first mode: they neglects the algorithm and dive straight into syntactical issues. While doing so, students tend to forget what the exact problem is they're trying to solve[3]. This leads to a lot of logical errors, that is, errors in the algorithm.

The syntactical knowledge that is needed to code in a certain programming language can be difficult too. For starters, code shows all nitty gritty details that are needed for the algorithm to work. Take for example the data structure that is used or specific libraries needed to make something work. These are details that obfuscate the logical errors and hinder when trying to get a quick overview of the program flow[1]. Secondly, using a concrete programming language might shape one's understanding of programming[4]. This is disadvantageous when a course tries to teach algorithmic thinking, rather than a specific language.

2.2 The solution: Flowcharts

Luckily, there is a way to counter these problems. For this we will use flowcharts. A flowchart is a diagram that visually represents a computer program in a step-by-step progression using conventional symbols¹. We will put text in these symbols that describe what should happen at that step. For an example, look at figure 2.1. We should note that this figure describes an algorithm in a concrete manner, rather than the abstract way that we will describe later. Later on we will look further into the conventional symbols that we will use for these flowcharts. Now we will discuss how flowcharts are going to solve earlier posed problems.

To begin with, if students make a flowchart before starting to program directly they are not confronted with the syntactic details. This prevents them from entering an act-first mode. Now they can first figure out how the algorithm needs to be solved and learn the bigger scope of the project[1].

Another advantage is that the way that we are going to use flowcharts does not enforce concrete details. Thus students are not directly imposed to think of technical details. This is why we can easier see the logical thinking steps and their corresponding problems before diving into the technical details[4].

Also, flowcharts serve as 'road maps', or overviews. They can swiftly communicate the program flow. If we compare flowcharts for example to another unplugged programming method such as using natural language, we see that this is a major advantage[5].

A last advantage of flowcharts is that they abstract away from language specific constructs. This way we can see more easily what algorithmic steps the student does wrong and does not impose a specific mindset.

2.3 Digital Flowcharts

In this section we will see what kinds of problems that drawing flowcharts on paper introduces and how a digital flowchart maker could solve these.

The first kind of problem that working on paper introduces has to do with syntax. Flowcharts try to focus on the algorithmic side of programming so it is easier to implement the syntactical part later. However, flowcharts have a syntax of their own, this is the set of conventional symbols we mentioned earlier. We see this being a problem when we create flowcharts on paper. Students tend to confuse the shapes of the different elements and make mistakes in the syntactical structure of the flowchart[4].

One way that a digital flowchart maker solves this is by visually representing all conventional symbols. Students now don't need to remember the

¹Merriam-Webster Online Dictionary: <https://www.merriam-webster.com/dictionary/>

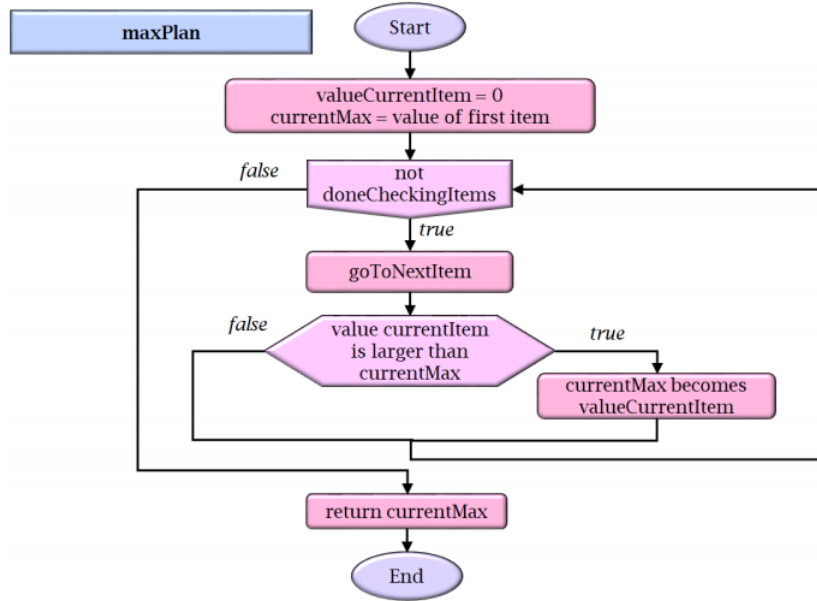


Figure 2.1: Example of a Smetsters and Smetsters flowchart

shape of the one they want to use, they only have to select the correct shape. Another way in which a digital tool helps with this syntax problem, is by limiting the possible ways to create flowcharts, ensuring that all flowcharts created have the correct structure. For instance, having an if-statement with more than two branches or a flowchart with multiple ending nodes can be made impossible.

A second problem has to do with the readability of the flowchart. Drawing a flowchart by hand and adjusting it later makes it cluttered really fast. Especially when you consider the creation of the flowchart as a learning process which needs reiteration[2]. Working digitally solves this problem.

One last problem that working on paper introduces is the limited shareability. Digital flowchart makers allow for digital copies that can easily be spread in the ICT-network of the school. This allows a student, their partner students and their teacher to look at it and change it whenever they want. It also makes it possible to have assignment deadlines outside of regular hours, creating flexibility for the teacher.

2.4 Drawing Flowcharts

Now, we will look further into the drawing of flowcharts as described in the Smetsters and Smetsters paper[1]. An example illustration can be found in figure 2.1.

A flowchart consists of different nodes connected with arrows. It is pos-

sible to traverse through the scheme by executing the step described in a node and following an arrow to a new node. There are different kinds of nodes on which we will now elaborate.

A first node is the Start-node. It is the point where the algorithm starts and is unique. This node is by convention placed at the top of a flowchart. There is also exactly one End-node, by convention placed at the bottom of the flowchart. Further, to execute an elementary step, such as the assignment of a variable, we use a Statement node.

A more intricate node is the if-node, which has a condition. From the node there are two arrows going to other nodes, one annotated with *true* and the other with *false*. Should the condition in the node be met, the user must follow the *true*-arrow to the next node. We call this the *true*-branch of the if-statement. Should the condition not be met, the user must follow the *false*-arrow. Eventually, these two branches converge again.

Lastly, there are two other kinds of nodes, being the while- and forEach-nodes. They also work with a condition and a *true*- and *false*-branch. Again, you follow the *true*-arrow should the condition be met and the *false*-arrow otherwise. The difference with the if-statement, however, is that *true*-branch of the while- and forEach-statements eventually loops back to themselves. The difference between the while- and forEach- statement is that the forEach-statement works on a specific data structure, lists, whereas the while-statement is less specific and can work on other data structures as well.

2.5 Nielsen's Heuristics: Guidelines to Usability

The last part of the background will be used to outline Nielsen's heuristics^[6]². Heuristics are broad guidelines for trial-and-error methods³, in this case they will guide us get good usability for our tool. We will list the heuristics in two parts. The first list will help us make a distinction between different designs of our tool. We will use these heuristics in section 3.3. The heuristics in the second list can be applied to all of them and therefore will not be used further in this work.

The heuristics that we will use to distinguish different designs are:

- **Visibility of system status**

An application should always keep the user informed on its internal state. This needs to be in reasonable time when a user is confronted with background processes that need time to load.

In our case users should always be aware what the flowchart is that will be executed. We will not implement time-costly background processes.

²10 Usability Heuristics for User Interface Design - <https://www.nngroup.com/articles/ten-usability-heuristics/>

³Merriam-Webster Online Dictionary: <https://www.merriam-webster.com/dictionary/>

- **Recognition rather than recall**

Users should not have to remember anything. Objects in play should be visible and instructions on how to use them should be easily retrievable or unnecessary.

This means for us that users should not need to remember the syntax of a flowchart and that adding, removing and editing nodes can be done without further instruction.

- **Aesthetic and minimalist design**

No irrelevant information should be shown, since this information takes away attention from relevant pieces of information. Features that are barely used should not take a disproportionate amount of space and user attention.

When we look at our tool, we can see that it should represent our requirements but prioritise these based on how often they are used. For example, adding nodes should be easier and therefore more prominent in the design than downloading flowcharts.

The other heuristics can be applied to all designs. We will list them and argument why and how will fulfil these:

- **Consistency and standards**

The same words or actions should have the same implications. If applicable, they should comply to general application standards.

All figures, buttons and text boxes should be consistent with each other. A button that adds a node should look the same as a button with the same behaviour in a different place. Also, these buttons should meet the global standards, they need to look like add buttons on other sites⁴.

- **Match between system and the real world**

The words used in the application should be familiar to the user, following conventions and following a logical order.

Within our tool all jargon used is explained to the students by the course that uses our tool. Therefore users should be familiar with the syntax and terminology.

- **Error prevention**

Check for error-prone input and try to let the user correct possible mistakes. This is applicable to for example date, since they need to follow a specific format.

⁴An example website that sets these standards is *Best practices for buttons* - Longo, Luca <https://uxplanet.org/best-practices-for-buttons-b7048479d440>

The tool explicitly allows for fuzzy input in order to give the user as much freedom as possible. No input is being tested, because no input is further used by the application.

- **Help and documentation**

If necessary, extra information needs to be easy to find, centred around the task at hand and concise.

For our tool we will search for a design that is sufficiently intuitive so it will not need extra documentation. Even though this is an important heuristic, we will not discuss it since the tasks performed with our tool are sufficiently simple that they do not need further documentation.

- **Help users recognise, diagnose and recover from errors**

Error messages should be clear, expressive, not contain unrecognisable codes and suggest a solution.

Later in this work, during the choice of our programming language^{3.2}, we will see that run time errors can only occur if they are explicitly programmed in. We will choose to not have run time errors, but instead handle the exceptional state in a predefined way.

- **Flexibility and efficiency of use**

Expert users can use accelerators, that is, functions that tailor frequent actions, speed up interaction and are unseen by first-time users. This way the application can serve first-time users and long-term users.

Our tool will not incorporate accelerators due to time constraints.

- **User control and freedom**

Users should be able to undo and redo every step. Other functionalities should always provide a *cancel*-button, should users start using the functionality by accident.

Our tool will not incorporate undo and redo functionalities due to time constraints.

Chapter 3

Tool development

In this chapter we will first establish the requirements, then discuss what programming language we will use. Next we will look into the design. Once we settled on a design, we will look into the details of implementing the tool.

3.1 Requirements

In this section we will outline all requirements that the tool needs to fulfil. We will do so in order of importance.

The first and foremost requirement of the tool is that the tool can create flowcharts as described by the Smetsers paper, explained in section 2.4. We will do so in a user friendly manner. To check if the tool is user friendly we will use Nielsen heuristics, described in section 2.5. The answer to this requirement will be formulated in two parts. First we will design the tool in section 3.3 and then we will implement it in section 3.4.

The second requirement of the tool is that it needs to be a web application. At the start of this research a preliminary prototype was developed in Java, so not as a web application. This has the drawback that it had to be installed on every school computer, whereas a web application would be accessible by default from every computer with internet access. Section 3.2 will be centred around finding the appropriate programming language to do so.

In this paragraph we will discuss some less prominent functionalities that the tool needs to have. A first example of this is the ability to formulate the state of an algorithm before and after the it's execution. In the Smetsers paper they call it the pre- and the postcondition. Another functionality is the possibility to save a flowchart with the intention to work on it at a later time on a different computer or to hand it in with the teacher. A last functionality is the ability to convert the flowchart into Java-code, to help bridge the gap between the problem solving in the flowchart and the actual coding. The implementation of these functionalities will be discussed in the

last section of this chapter, 3.4.5.

3.2 The programming language

In this section we will motivate our programming language choice. The tool needs to be a web application, what strongly determines our programming language. In web development there is one go-to language for web applications, JavaScript[7]. We will now match JavaScript to the programming language Elm¹, a functional programming language that compiles to HTML, CSS and JavaScript². We will motivate why we picked the latter.

Keep in mind that we want to construct a single page web application that is easy to maintain.

3.2.1 JavaScript

JavaScript started in 1995 as hastily constructed dynamic, easy-to-learn web language. Nowadays it is an established language for building dynamic web pages, having all needed features in extensive libraries.[7] However, there are two aspects that are more negative than positive in regard to our project.

The first one is that JavaScript gives freedom to the users on how to use it, leading it to be used with different paradigms, having grown broad and complex over the years. Libraries are not forced to use semantic versioning correctly, meaning there must not necessarily be a clear distinction between backwards compatible library updates and updates that might break your current project build. This means that libraries could push changes that break your build even though their version number says that it won't. Also, projects in JavaScript can have very different project architectures because there has never been one universal standard, this makes projects less clear to other programmers and therefore less maintainable.

The second aspect is JavaScript's loosely typing[8]. Again, this makes programming very flexible and gives freedom to the programmer. However, this also enables the programmer to make code unreadable but still functioning, demanding a great discipline of the programmer to produce organised code. Aside from that, loosely typing makes refactoring code hard, since it can result in ambiguous code. These effects can make writing easy maintainable code harder.

All in all, JavaScript code is very flexible and easy to program with, but demands quite a time investment to master due to its extensiveness and pitfalls. Also maintenance is hard, since there are no strict coding standards enforced and refactoring might lead to ambiguous code.

¹Elm - E. Czaplicki <http://elm-lang.org/>

²Based on The Why and When of Choosing Elm - O. Hanhinen <http://ohanhi.com/why-and-when-of-choosing-elm.html>

3.2.2 Elm

Now let's look at Elm. Elm is relatively new, originating from a master thesis[9] and built for robustness and performance. It doesn't hold back when it has the chance to enforce certain programming habits onto the programmer. It limits the programmers freedom in a programmer friendly way. We will look at the two aspects that bothered us in JavaScript.

First off, because Elm is new and originates from an academic environment[10] its programming standards have been very well thought through³. We see this in several ways. To begin with, the Elm package manager enforces semantic versioning, meaning that your builds cannot break if you managed your dependencies right. Also, the project architecture in Elm projects is unified. This is because Elm forces you to use a Model-Update-View architecture, which we will explain in section 3.4. This unification of build schemes makes maintenance a lot easier.

Secondly, Elm uses type inference⁴. This means that the compiler must know what type every object is during compile time. It allows the compiler to detect some mistakes beforehand and report these to the programmer. To infer types the programmer needs to explicitly state what type all variables and functions have. It therefore limits flexibility and ease of use for the programmer, but once the program compiles it is guaranteed to run without inconsistencies, ambiguities or runtime errors.

3.2.3 Conclusion

With Elm enforcing higher code quality, it ensures easier maintainability and the possibility to extend it later on without giving in on functionality. We therefore think that Elm fits our personal wishes better than a JavaScript framework.

3.3 Designing Flowchart Manipulation

To fulfil the first requirement we will discuss three different designs for flowchart manipulation and search for the most user friendly one. The three designs are *Selection bar*, *Drag and drop* and *Overlay buttons*. For each of them we will describe the interface and match it to Nielsen's heuristics, which are listed in section 2.5. Finally we will compare the three with each other and conclude on a final design.

³Design Guidelines - E. Czaplicki <http://package.elm-lang.org/help/design-guidelines>

⁴Compilers as Assistants - E. Czaplicki <http://elm-lang.org/blog/compilers-as-assistants>

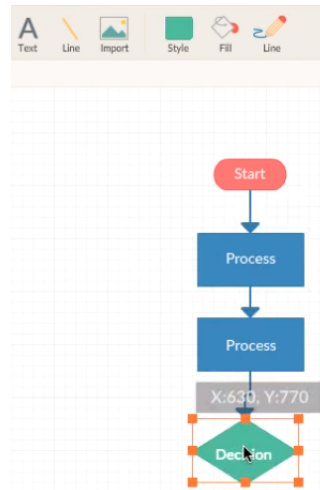


Figure 3.1: An example of a selection bar from Creately. At the top we see the selection bar and at the bottom a flowchart that is being built.

3.3.1 Selection bar

The first design that we will discuss is the selection bar. An example of a selection bar interface can be seen in figure 3.1, which is built using Creately⁵. It has a bar of buttons and each button changes the way you interact with the flowchart. For example, clicking the "Add node above"-button in the selection bar and then clicking a node will add a node above said node. The flowchart also contains a node that allows the user to insert nodes in the tree, the Empty-node. Selecting for example the "Insert If"-button and clicking an Empty-node will change the Empty-node into an If-node.

One of the upsides of this design is that the user always has a clear view of the current state of the program. There is just one flowchart and by ignoring the empty-nodes you will always have a syntactically correct one. Also, all nodes are given inside the bar so the user doesn't have to recall anything.

There is one downside to this design. Nielsen urges to use a minimalist design. However, the selection bar at the top is always present, even when the user is finished with the flowchart.

3.3.2 Drag and drop

Another possible design is the drag and drop interface. This is illustrated in 3.2, which is built using Scratch⁶. In this design the components of a flowchart are available at the side of the flowchart. They can be moved into

⁵Cinergix Pty. Ltd. - <https://creately.com/diagram-type/flowchart>

⁶Lifelong Kindergarten Group, MIT Media Lab - <https://scratch.mit.edu/>



Figure 3.2: Example code of Scratch, which uses drag and drop flowchart manipulation. Left we see the components and right the building area. Note that Scratch does not work with a Start- and End-node.

a building area, and snapped onto existing components. There are always exactly one Start- and End-node which are always connected to each other. All components must be put in between them. These together with the Start- and End-node form the flowchart.

One advantage is that the system status is always clearly visible, the flowchart is between the Start- and End-nodes. Furthermore all possible new flowchart components are visible, so the user won't have to rely on recalling them.

However, there is a disadvantage to this design too. The area with new components always shows all components, even though the user might be done with it's flowchart. This conflicts with the heuristic of a minimalist design.

3.3.3 Overlay buttons

The last design for flowchart manipulation that we look into are overlay buttons. An example is shown in figure 3.3, which is built using the final version of the tool that we will build in this thesis⁷. Upon hovering over a node several buttons will fade in. These buttons allow you to insert and delete nodes. An inserted node is an Empty-node, which has buttons in it for all possible insertable nodes. Clicking one of these buttons will change the Empty-node in the corresponding insertable node.

Just like the other designs, this one has a clear visibility of the system state and removes the necessity for the user to recall flowchart components. The biggest advantage of this design however, is it's minimalist design. Only buttons that are relevant at that moment are visible. Once the user has finished his flowchart and he isn't hovering over a node, there are no buttons at all. This is ideal, since the user is done building the flowchart and therefore doesn't need buttons for flowchart manipulation anymore. At the same time,

⁷Flowchart tool <http://course.cs.ru.nl/greenfoot/flowchart/flowcharttool.html>

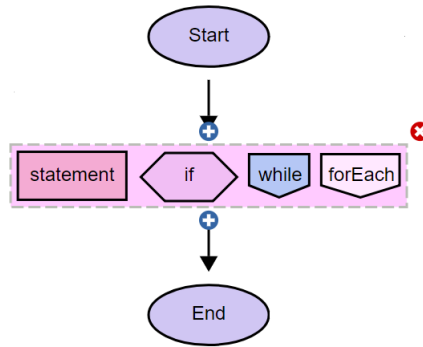


Figure 3.3: An example flowchart using the overlay buttons menu on an Empty-node.

all necessary buttons are provided. This is because we use the location of the mouse to determine what the user wants to do. Is it on a node, than he probably wants to delete this node or insert a node above or below it. The user probably wouldn't be searching for a way to delete a node at another location.

3.3.4 Conclusion

In conclusion we can see that the overlay buttons design is superior over the others due to its extremely minimalist design. This is the design that we will build during the implementation

3.4 Implementation

In this section we will look at the implementation of the tool. First we will look at the implementation of the core parts of the tool, by talking about its model, then its update and finally its view. Next we will discuss the technical limitations that the implementation has. At the end of the section we will look at the non-core functionalities of the tool.

3.4.1 Model

In a model-update-view architecture, the model describes all information contained by the program. In this section we will elaborate on the data type that forms our flowchart, the *Tree* data type. Below we can see the associated code for reference. Note that a comment in Elm is a line that starts with ‘—’ and is followed by recursive text.

```

type alias Id      = Int
type alias Content = String

```

```

type alias Tree =
{
  id : Id
  , basicTree : BasicTree
}

type BasicTree
-- keyword      restOfFC
= Start        Tree
-- keyword
| End
-- keyword      restOfFC
| Empty        Tree
-- keyword
| Void
-- keyword      text      restOfFC
| Statement    Content    Tree
-- keyword      text      falseBranch trueBranch restOfFC
| If           Content    Tree        Tree        Tree
-- keyword      text      trueChild   restOfFC
| While        Content    Tree        Tree
-- keyword      text      trueChild   restOfFC
| ForEach      Content    Tree        Tree

```

We should clarify that this *Tree* data type is a derivative of the tree data structure⁸. This means that almost every node contains a *Tree* inside it, being the rest of the flowchart. In this work we will use *Tree* to denote the following child node and all of its successors, whereas we say child node if we only mean the directly succeeding node of our current node.

To start with we have two type aliases, *Id* and *Content*. These are synonyms for types that already exist. They help us identify the meaning of the variable and help the compiler distinguish regular *Int*'s from *Id*'s and regular *Strings* from *Contents*.

Next we have the type alias *Tree* which is synonym for a couple of *Id* and *BasicTree*. We will discuss the latter in the next paragraph. We made *Tree* because every node has an *Id* and we did not want to repeat *Id* as attribute for all node-types. Rather, we want to make explicit that every node has an *Id*.

The last piece of code describes the type *BasicTree*. Contrary to type aliases this describes a whole new type, it is not simply composed of already known types. A newly defined type has different instances and to make a distinction between them we begin every instance with a keyword. After the keyword we give all attributes that belong to that instance. Note that all instances are preceded by a comment describing the semantics of the attributes.

For example, we see that *BasicTree* has an instance named *Start* which has only one attribute, a *Tree*. This *Tree* is the remaining part of the

⁸Tree - Wikipedia [https://en.wikipedia.org/wiki/Tree_\(data_structure\)](https://en.wikipedia.org/wiki/Tree_(data_structure))

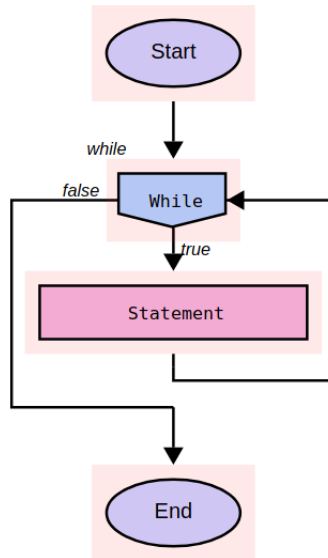


Figure 3.4: Statement has an invisible and immutable child, Void

flowchart, so the following node and all of its successors. Another example is the instance *If* which has four attributes. The first is a *Content*, it is the text that holds the condition of the *If*. The second and third attribute are the *Tree*'s that contain the *false* and *true* branch respectively. The last attribute is the *Tree* that follows after the *false* and *true* branches merge.

However, this gives us a problem at the end of a branch, since all nodes that can be inserted in a branch have an attribute that wants to contain the rest of the tree. The question now is what will be the last child of a branch.

Technically, we could end every branch with an instance of *End*. However, we do not want every branch to have a visible *End* node, since defined our flowcharts differently in section 2.4. So to solve this problem we use an auxiliary node, *Void*. *Void* is invisible, cannot directly be mutated by the user and is used to end separate branches. So for example in figure 3.4 the *Start* node has *While* as child and *While* has *End* as direct child. *While* is special in the sense that it has an extra child in it's loop, a *Statement*. Now *Statement* per construction needs a child too. This is where we use *Void*. *Void* is injected in the tree structure as last child of a branch. So here *Void* is the child of *Statement*.

3.4.2 Update

In the model-update-view architecture the update function takes care that the model is renewed every time new information comes in. We will now take a closer look at the types that make the flowchart manipulation possible and the actions taken upon receiving a so called message. Below another piece

of code for reference. Note again that ‘—’ denotes a comment that explains the semantics of the line below it.

```

type Msg
  — keyword          updatedText  currentNode
  = UpdateContent Content      Id
  — keyword          newNodeType  currentNode
  | FillEmpty      FillEmpty    Id
  — keyword          actionNeeded currentNode
  | ChangeTree     ChangeTree    Id

type FillEmpty
  = AddStatement
  | AddIf
  | AddWhile
  | AddForEach

type ChangeTree
  = NewAbove
  | NewBelow
  | NewTrue
  | NewFalse
  | Delete

```

To begin with, Elm obligates the use of the *Msg* type, or in normal words, the message type. The message type lists all possible ways in which the program can update itself. Every time something is triggered in the view it can send a message, which will then arrive at our update function. We will not explicitly give the update function here, but we will describe its behaviour.

When our update function receives an *UpdateContent* message it searches the *Tree* for the node with the according *Id*. Once found it will replace the current *Content* by the new given *Content*.

A *FillEmpty* message will result in the update function searching for the node with the given *Id*. This should lead to an *Empty* node which we will then replace by either a *Statement*, *If*, *While* or *ForEach* node. To make a difference between these we introduce a new type that we will call *FillEmpty*, named just like the keyword of this message, and consists of these four options.

Lastly we will discuss the *ChangeTree* message. This message is used to add or delete nodes. As with the *FillEmpty* message there are different options to consider that we will list in a different data type, which is called *ChangeTree*. Again, named just like the keyword of the message type. These options are adding a node directly above or below another node or deleting the node itself. On top of that, an *If* node has two branches that it can add nodes to, a *true*- and *false*-branch.

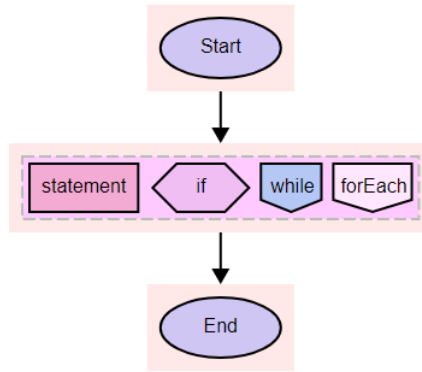


Figure 3.5: The initial flowchart

3.4.3 View

The last part of the architecture that we will discuss is the view. The view is the part that we designed to use overlay buttons in section 3.3.3 and is illustrated in figure 3.5. The elements it draws are described in the model section and the messages it releases are described in the update section, view does not introduce new types or type aliases. So in this section we will not look at those, but at how we draw the model and send messages.

To start with, the *Tree* elements will be drawn using Scalable Vector Graphics⁹. Within the *Tree* elements that have a *Content* we put HTML-textboxes to make them editable. Typing in these textboxes fires *Update-Content* messages with the new text, updating the content of the textbox.

An *Empty* node contains all four insertable nodes, as seen in figure 3.5. Clicking one of these will send a corresponding *FillEmpty* message and change the *Empty* node into the clicked node.

As for the overlay buttons, every node has a rectangular hitbox behind it that registers when a mouse enters and leaves. When a mouse hovers in three buttons appear, two add-buttons and one delete-button, as can be seen in figure 3.6. The add-button is blue and has a white cross and is used to add *Empty* nodes. The delete-button is red with a white cross in the top right corner and will remove the underlying node. Some nodes such as the *If* node have extra add-buttons at the sides of their condition box. All of these buttons work by sending the according *ChangeTree* messages to the update function.

3.4.4 Technical limitations

In this section we will discuss some technical difficulties that could influence the usability of the tool, but have nothing to with the design.

⁹Scalable Vector Graphics (SVG) 1.1 (Second Edition) - World Wide Web Consortium <https://www.w3.org/TR/2011/REC-SVG11-20110816/>

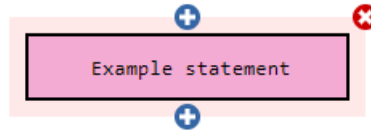


Figure 3.6: An overlay menu on a statement node

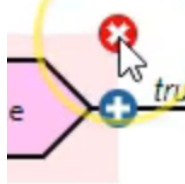


Figure 3.7: Only the bottom left quarter of the delete-button is within the hitbox

First off, the hitbox' outlining is used for further calculations. This is why the arrows between nodes stop at the hitbox rather than the node itself, as can be seen in figure 3.5. Making the hitbox invisible would create the inconsistency that arrows not always connect to boxes, so we gave it a discrete colour. Also, the overlay buttons are drawn based on the outlining of the hitboxes rather than the underlying node, as can be seen in figure 3.6. This leads to only a quarter of the delete-button being inside the hitbox, as illustrated in figure 3.7. Upon leaving the hitbox, the overlay menu would disappear. Thus hovering your mouse into one of the three outer quarters of the button would lead to the disappearance of the button.

A second limitation is that the tool has a character limit, as seen in figure 3.8. The line count received from HTML-textboxes is inaccurate due to invisible characters being counted. This leads to the SVG-shapes scaling inaccurate with the HTML-textboxes. In length however, they do scale correctly. This lead us to create a character limit of 22 characters on the *Content* of a *Statement* node. The *If*, *While* and *For-each* nodes have a limit of 25 characters. When this limit is reached, no more input can be given.

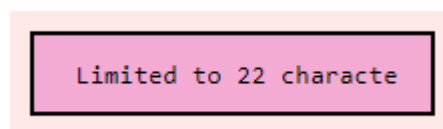


Figure 3.8: There is a character limit of 22 tokens for Statement-nodes

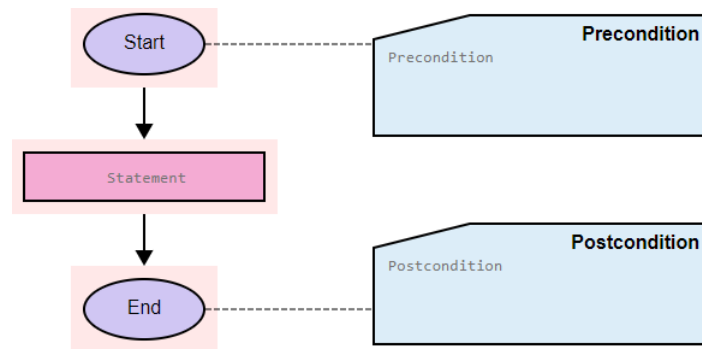


Figure 3.9: The pre- and postconditions

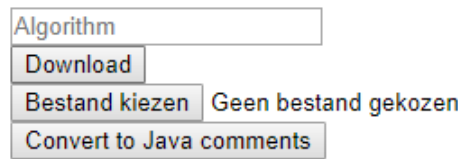


Figure 3.10: Menu presenting extra features

3.4.5 Other functionalities

In this section we will look deeper into peripheral functions of the tool. These are the possibility of pre- and postconditions, local save files and a conversion method to Java comments.

For starters, part of the course "Algorithmisch Denken" was to formulate the beginning and ending state of the algorithm in the pre- and postconditions. As such we implemented text boxes accordingly, as can be seen in 3.9. These are connected with the Start and End node, since they give the states of these nodes.

The requirements to save files and convert a flowchart to Java-comments is fulfilled by adding an extra menu, given in figure 3.10. The first box allows the user to enter a name for the algorithm. The button below it allows for a download of the flowchart, followed by a button to upload a local save. The last button lets the user convert the flowchart to Java comments

Up- and downloading is implemented by transforming the tree into a JSON-file[11]. This JSON-file is then downloaded, assuming the name of the algorithm as name with a .flow extension. This .flow extension is necessary to evade the school's built in spam filter since it marks all .json files as malicious files.

Our last feature is the possibility of converting a flowchart to comments. Upon clicking the accessory button, a prompt will come up with a commented text version of the flowchart. This text file can be pasted directly into a Java file without resulting in errors.

Chapter 4

Methodology

In this chapter we will look into the specifics of what we are going to research and how we are going to do that. We will first formulate our research question and sub-questions. Next we will outline the environment in which we will do the research. In the last section we will look at how to measure the sub-questions by choosing suitable theoretical and empiric variables.

4.1 Research question

In this thesis we are looking for the best way to create a tool that restrains the user from making syntactically incorrect flowcharts. On the other hand we do not want to limit students capacity to express their algorithmic thinking. In other words:

What does a digital flowchart creating tool looks like, such that it supports learning to program the best?

To answer this question we formulate three sub-questions to which we will always refer to by number. Is it clear for users how to...

1. add or delete a node?
2. choose a node type from an *Empty*-node?
3. edit the content of a node?

4.2 Measuring environment

In this section we will look at the environment of the research. To test the tool we will use two measuring moments on the same participants. A pre-test at the 20th of November, 2017, and a main test one week later. We will first look into the participants group and the exercises that they will do.

Then we will look at the measuring methods during the pre-test and the main test.

We will test the tool in a class that follows the course *Algoritmisch Denken*¹ which is outlined in [1]. This course is given to a class of eight fifth year high school students. According to Faulkner [12] a group of 8 participants in a usability research is sufficient to find at least 75% of the mistakes. Students will not be expected to have programming knowledge prior to the start of the course in September. For this research the students will be divided in three groups, the same groups that they worked in during earlier courses. These will be groups made by the students themselves. The research will be done during courses when the course is already running for two months on a weekly basis. During the research the students will get new theory on algorithmic thinking that they will have to implement using flowcharts.

At the start of the course of the pre-test, the students will have a few minutes to get accustomed to the tool, without explicit instructions on what to do. After that they will have to solve an exercise that made them use all flowchart manipulations. During the course of the main test the students will have to build another two flowcharts.

During the pre-test we want to find out if the testing tools operate correctly. Also we will be able to slightly adjust the tool after the pre-test, should we see design flaws. In that sense, the pre-test allows us to reiterate our work. Data generation in this pre-test will be done in three ways, through overt observation and by capturing screen activity and filming the reactions of three groups of students with a webcam. This way we can experience the results firsthand and also have material to delve into afterwards.

The main test will be used to generate more and richer results. It will therefore not only have earlier mentioned observations on two groups, but will also include semi-structured interviews at the end of the class. Due to time restrictions, there will only be two of these interviews: one with group 1 and one with group 2 and group 3 together. The interviews will be recorded and transcribed.

The screen activity of the pre-test and main test will be coded using Atlas.ti². The interviews will not contain direct quantifiable information about the users interacting with the tool and will therefore not be coded. Instead, we will use quotes of the interviews in the discussion, chapter 6, when a student explicitly refers to an occurrence, so we better understand their point of view.

¹“Algoritmisch denken” course - Smetsers, Sjaak and Smetsers-Weeda, Renske <http://course.cs.ru.nl/greenfoot/>

²Atlas.ti - Scientific Software Development GmbH <https://atlasti.com/>

Table 4.1: The sub-questions with their according theoretical variables, empiric variables and the empiric variable abbreviations

	Theo var	Abbreviation	Empiric variable
1	Affordance	ClickAdd	Clicking Add-button, rather than dragging
		ClickDelete	Clicking Delete-button, rather than dragging
		Misclick	Misclick due to buttons disappearing
	Meaning	CorrectButton	Clicked on button in Empty-node, rather than dragging
2	Affordance	ClickNodeType	Using the correct button
	Meaning	CorrectNodeType	Used the correct node type
3	Affordance	TypeText	Typing text, rather than leaving the text box empty
	Ease of use	RemovePH	Typing after removing the placeholder text during the pre-test
		NoRemovePH	Typing without removing the placeholder text during the main test
		CharLimit	Stopped typing due to character limit
		TextBoxBlur	Stopped typing due to by text box blur

4.3 Empiric variables

In this section we will look further into the sub-questions that we pose in order to answer our research question. For each sub-question we will give theoretical variables and for each of these we will give empiric variables with abbreviations. For a complete overview, see table 4.1.

The first sub-question concerns adding and deleting nodes. For starters, we will use *affordance* as a theoretical variable. *Affordance*³ is the situation in which an objects characteristics imply it's functionality and use. In our case this means that we want our buttons to be clicked on and not for example dragged. Specifically we will check here if the user perceives the add and delete buttons as clickable. In our next empiric variable we will count the times that a user tries to click a button but the menu disappears

³Usability First: Glossary - Foraker Labs <http://www.usabilityfirst.com/glossary/affordance/>

due to the mouse leaving the hit box.

The second theoretical variable that we will use is *meaning*⁴, that is, does the user intend to do the thing that happens. Here we will check if the user is aware of the meaning of the buttons. We count how often a user uses the correct button. We also measure this by looking if the user tries to undo its actions, after clicking a button. If so, we can say that the user did not intend the action to happen.

The second sub-question that we treat concerns the use of the *Empty*-node. To measure this we will use the amount of clicks on the buttons inside our *Empty*-node. Every time a user tries to interact in a different way with these buttons, we will count the variable as failed. The second theoretical variable that we will use is the *meaning*. For the use of the *Empty*-node this means that when the user clicks on the miniature *Statement*-node he should get a *Statement*-node, rather than for example a *While*-node. When a user clicks on a node and directly deletes it in order to replace it by another node, we see this as a failure towards this empiric variable.

The last sub-question is about editing the content of the node. Again we will test the *affordance* by counting the times that users try to write in the text box. The second theoretical variable that we will use is *ease of use*, that is, to what extent the text boxes hinder the thinking process of the user. We will do so by counting the times that a user doesn't remove the placeholder text before typing during the pre-test, but instead starts typing in the middle of this text and then remove the placeholder text afterwards. We will also count the times during the main test that users tried to delete the placeholder text while it already disappeared out of itself. Another thing that we will count is the times that users are limited by the character limit, by counting the times that they are typing but no extra input is shown due to the character limit. The last empiric variable belonging to *ease of use* will count the times that users try to type but the program loses focus of the text box and thus no typed text will be appear on screen.

Note that we make a difference between empiric variables that can succeed or fail and variables that simply occur. For the empiric variables that can succeed or fail we will make a distinction between the first time that something is done and later times. Should an empiric variable only lead to a fail the first time, we know that the user accommodates his behaviour without too much effort. This helps us prioritise design issues.

⁴Merriam-Webster Online Dictionary: <https://www.merriam-webster.com/dictionary/>

Chapter 5

Results

In this chapter we will objectively describe the results of our research, given in three different tables. We will first outline the empiric variables that can succeed or fail split out over two tables, one for the pre-test and one for the main test. The third table that we will discuss displays the occurrences of the empiric variables that cannot fail or succeed.

We must remark that a part of the data got lost during the main test. This is because the program that records the screen and films the students had to be closed in a specific way in order to save the recordings. One group however did not save the recordings during the main test. This is why there is one group less in the main test then in the pre-test.

We will now discuss the empiric variables taken during the pre-test that can succeed or fail, shown in table 5.1. Since we are mostly interested in the times an empiric variable failed, we have listed these. For reference we also show the amount of total occurrences. So for example, we see that *ClickDelete*, belonging to research sub-question 1, has failed once when one of the three groups used it the first time, but never failed after that. This means that during the pre-test one group tried to interact with a delete-button, not by clicking on it, but by doing something else. Also, we can see that *ClickDelete* happened 22 times, thus in total 21 successes. This implies that during the pre-test the delete-button was clicked 21 times and only once had been interacted with in another way.

Note that the sum of the total occurrences of *ClickAdd* and *ClickDelete* should be equal to the total occurrences of *CorrectButton*. However, during the pre-test the students were allowed to get used to the tool. In this period they clicked buttons without having a concrete intention with it. Therefore we did not register *CorrectButton* and *CorrectNodeType* during this trial period. After this trial however, we did.

The empiric variables that either succeed or fail during the main test are given in table 5.2. It is similar to the table with the pre-test results in that it only shows fails and total occurrences. So take for example *NoRemovePH*,

Table 5.1: Empiric variables during the pre-test that succeed or fail, discriminating first occurrence

	Empiric variable	First time fail	Later fail	<i>Total occurrences</i>
1	ClickAdd	0	1	32
	ClickDelete	1	0	22
	CorrectButton	0	0	44
2	ClickNodeType	1	1	48
	CorrectNodeType	0	0	42
3	TypeText	0	0	39
	RemovePH	1	0	39

Table 5.2: Empiric variables during the main test that succeed or fail, discriminating first occurrence

	Empiric variable	First time fail	Later fail	<i>Total occurrences</i>
1	ClickAdd	0	1	11
	ClickDelete	0	0	3
	CorrectButton	0	0	14
2	ClickNodeType	0	0	19
	CorrectNodeType	0	0	19
3	TypeText	0	0	18
	NoRemovePH	1	1	18

the empiric variable that measures how often someone tried to remove a placeholder text. It failed once the first time, out of the two groups that were observed, and failed a second time later on. In total there were 18 occurrences where the placeholder could be removed of which only 2 where users tried to do so.

Note that the main test did not include time to get used to the tool, so the total occurrences have their logical relations. For example, the sum of total occurrences of *ClickAdd* and *ClickDelete* is equal to the total occurrences of *CorrectButton*.

The last table that we will discuss, shows all empiric variables that cannot specifically succeed or fail and is given in table 5.3. Here we split the table up in pre-test and main test and over groups. So we can see for example that during the pre-test group 2 had no occurrences of *Misclick*, but did have 8 occurrences of *TextBoxBlur*.

Table 5.3: Empiric variables during the pre-test and the main test, split over groups

	Empiric variable	Pre-test			Main test	
		Group 1	Group 2	Group 3	Group 1	Group 2
1	Misclick	3	0	4	2	1
3	CharLimit	1	2	3	3	3
	TextBoxBlur	3	8	7	8	0

Chapter 6

Discussion

This chapter will interpret the results. We will do so by going over all three sub-questions and discussing every empiric variable. After we discussed all empiric variables of one sub-question, we will try to formulate an answer to it. At the end of this chapter we will mark a possible critic to this research.

6.1 Interpretation

6.1.1 Add and delete nodes

For the *ClickAdd* empiric variable, we see that there are no first time fails. This implies that students perceive the add-button as something clickable. The two later occurrences of fail are due to an add-button missing. These cases are illustrated in figure 6.1. In the first case, 6.1a, a student just deleted the *false*-child of an *if*-node. At the spot where the node disappeared, he expects to find something to undo what he has done. Later he finds out that the *if*-condition box allows him to add a child in the *false*-branch. The second case, 6.1b, involves an *if*-node that is nested in a *while*-node. Students were working step by step from top to bottom to fill the body of the while. However, upon adding an *if*-node, they could not add a child directly underneath it. Contrary to the other case, there is no undiscovered possibility to add a node at this location. The solution of the students was to recreate the flowchart with an extra *empty*-node below the *if*-node before instantiating it.

We have found that *ClickDelete* always results in success, except for one of the first times, during the pre-test. One group tried to drag the button when they wanted to delete an *empty*-node, therewith selecting all the contents of the *empty*-node. We can conclude from this that their initial thought was that this was a draggable shape, rather than a button. Also there are two noteworthy events associated with the fact that deleting for example an *if*-node will also delete both its branches. A first is that a student once deleted a valuable child upon deleting a *while*-node. The second is that

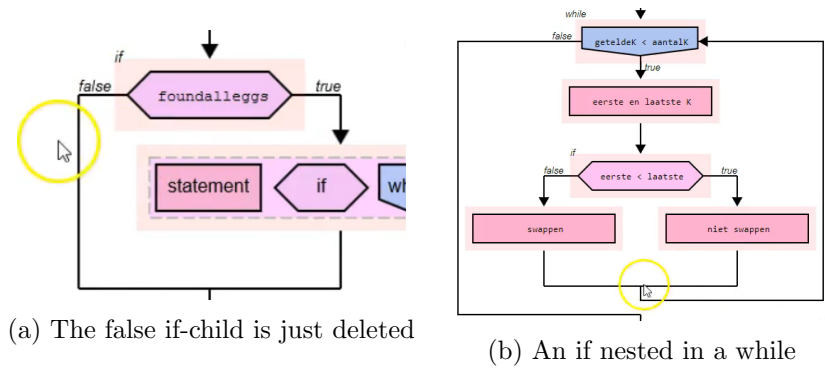


Figure 6.1: Missing options to add nodes at the cursors location

one group wanted to delete their flowchart, consisting of nested *if*-nodes and did so by first deleting the child nodes and finishing with the parent node. This took them five clicks, whereas deleting the top-level *if*-node first would have accomplished that in one click.

The *Misclick* empiric variable did not occur with one group during the pre-test. The other two groups had a few occurrences. The times it did occur, did not lead them to openly comment about it during the test. There was however one occurrence where the students gave up clicking on a button when they tried to delete a node, saying: “*That one can’t go away.*”¹ and moving on to do something else. In the interviews the students commented “...*the crosses tend to disappear when you try to click on them.*”²

The last empiric variable that we will discuss for this sub-question is *CorrectButton*. The results show that no one ever added or deleted a button without intending to do so, so its functioning seems to be clear.

To conclude, we see that the *affordance* of the add- and delete-buttons is clear in general. The most problematic results are misclicks, caused by the technical limitations, outlined in 3.4.4. The *meaning* of the buttons seems to be clear too. All in all we can say that adding and deleting nodes is clear for the users.

6.1.2 Choose node type

ClickNodeType only failed twice, once during a first time in the pre-test and once later on during the pre-test with another group. In both cases the students tried to drag an element in the *empty*-node outside of it. However, the large amounts of successful occurrences show that students correct their behaviour after mistaking once.

¹Original: “*Die kan niet weg.*”

²Original: “...*alleen die kruisjes enzo als je probeert er op te klikken dan gaat het meestal weer weg.*”

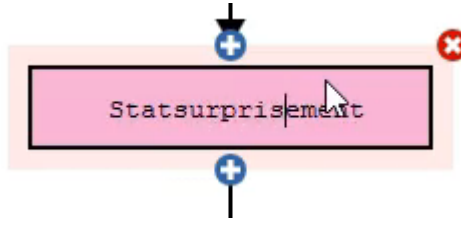


Figure 6.2: The placeholder text is expected to disappear

The *CorrectNodeType* variable never failed and has relatively high frequencies compared to the other empiric variables. We can therefore conclude that the link between the drawings of the nodes and their study material is good.

In conclusion we can see that clicking on node types can be a bit confusing in the beginning, since students seem to feel the need to drag the buttons. However, they change their behaviour swiftly. We can say that the functionality of choosing node types seems to be clear to all students.

6.1.3 Edit content

The *TypeText* empiric variable is always successful. So we could say that it was always clear for users that text boxes are for typing text.

During the pre-test we counted the *RemovePH* variable. One group forgot to remove the placeholder text the first time as shown in figure 6.2. All other times were successful. It should be noted that one group reused parts of the placeholder text and another replaced it with the dutch variant.

In the main test we counted *NoRemovePH*. One group tried to remove the placeholder text twice, once during the first time interacting with the text box and once later. None of the students said they missed the placeholder text. Together with the fact that students always seemed to delete the placeholder text, we can conclude that removing it was a good idea.

The next empiric variable that we will discuss is *CharLimit*. During the pre-test one group reported it as a bug upon first encountering it. “*We’ve got a problem. It doesn’t grow when the text doesn’t fit.*”³ In the main test most groups got real struggles with the character limit. They shortened their answers, making them less clear. For example “Swap smallest and” got changed into “Swap 1 and small”⁴ and “take first card and la” got changed into “first and last K”⁵. The latter took 40 seconds for the students to change, since they could not come up with a shorter wording. The students commented

³Original: “*We hebben een probleem. Hij wordt niet groter als de tekst er niet meer op past.*”

⁴Original: “Wissel kleinste kaar” ← “Wissel 1 en klein”

⁵Original: “pak eerste kaart en la” ← “eerste en laaste K”

“*You almost can’t write anything in it and that is really annoying.*”⁶ In the interviews all students agree that the character limit is annoying, saying things like “*... and also the length of the text cannot be bigger than a certain amount of letters. I think that is really annoying because some statements are a little bit longer than others and then it is not small enough for the box.*”⁷.

TextBoxBlur has had quite some occurrences. All groups had trouble with this, commenting “*OK just leave it, it is good.*”⁸. One group in particular had a lot of trouble with this during the pre-test. An important thing to note is that once a text field blurs, all keyboard input is interpreted directly by the browser as shortcuts. The group that had trouble with the blurring issue was also the only group to use the Firefox browser⁹, whereas the others used the Chrome browser¹⁰. In Firefox the *backspace*-key is used to go back to the previous page. Doing so will unload the tool and all information in it, effectively deleting all work done. It should be noted that this happened during the pre-test. As seen in figure 6.3 all nodes still had default text in them. So when the students made a new *while*-node the first thing they did was delete the contents by repeatedly pressing *backspace*. However, upon pressing *backspace* the text box and hit box shrink. So as seen in 6.3b there is a point where the cursor itself stays still, but still leaves the hitbox because the hitbox shrinks, thus blurring the text field. The students do not notice the blur, press *backspace* once more and leave the page, deleting all their progress. It was technically impossible to remove this bug before the main test, so instead we urged all groups to use Chrome.

All in all we can say that the *affordance* was clear, since all students immediately started typing in the text boxes. The *ease of use* however is questionable. Even though removing the placeholder text was a good move, the character limit and the text box blur hindered the students too much to call the text boxes a good and clear functionality.

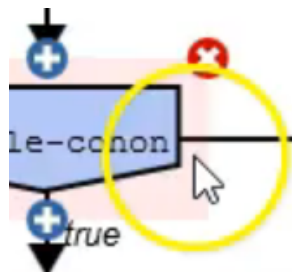
⁶Original: “*Je kunt er echt bijna niks in schrijven en dat is echt heel irritant.*”

⁷Original: “*... en ook mag de lengte van de tekst kan niet groter zijn dan een bepaald aantal letters. Dat vind ik ook irritant want sommige statements zijn iets langer dan anderen en dan is het niet kort genoeg voor het blokje.*”

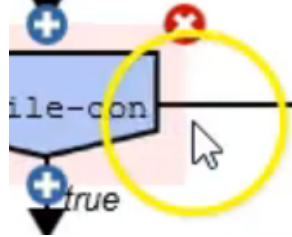
⁸Original: “*OK laat maar, het is goed*”

⁹The Mozilla Firefox browser, the Mozilla Foundation - <https://www.mozilla.org/nl/firefox/new/>

¹⁰The Google Chrome Browser, Google LLC - <https://www.google.com/chrome/>



(a) Right before the blur



(b) The text field is now blurred

Figure 6.3: The events leading up to an intended screen closure

6.2 Remarks

One could argue that one of the writers of the paper that this work is based on[1] is also involved in the research itself, namely by being the teacher of the class. However, this research concerned the usability of the tool, whereas the possible conflict of interest is situated on the level of the course method. Therefore we do not mark this as a problem.

Chapter 7

Conclusions

We looked into the best way to design a flowchart tool through a design and creation method. To this end we got acquainted with basic flowchart terminology and Nielsen's heuristics for a good design. Next we have set up requirements for the tool, chosen a programming language accordingly and then went over three designs of flowchart manipulation. Using Nielsen's heuristics we choose one. After that we looked into the technical specifics of the tool, namely how it concretely works, what the limitations are and what other functionalities it encompasses. Next we formulated a research question and split it up over three sub-questions to which we assigned empiric variables. We then discussed the results of the testing and interpreted these. It showed that the add- and delete-buttons work good, except for the technical limitations, and that choosing node types also works well. Most problems were encountered with the editing of the content where the character limit and the sudden text box blurs hindered user experiences. All in all we have explored the ways to design and build a flowchart creating tool. Our final result seems only to be hindered by technical limitations, rather than its design.

7.1 Related Work

One tool that was used before to draw flowcharts is the web application draw.io¹. Draw.io has all benefits over working on paper that we noted in section 2.3. Handwriting for instance is no longer a limiting factor, things can be quickly reordered and flowcharts can now be saved as pdf-file and sent to the teacher through the schools digital environment.

However, draw.io has some drawbacks. The main one being that students have to do a lot of actions to accomplish simple things. For example, to start constructing a flowchart a student has to create a *Start*- and *End*-node. To do this, they have to select a shape, drag it to the right position, create a

¹draw.io - JGraph Ltd. <https://www.draw.io/>

text box within the shape and type the word "Start" or "End" in it. Finally, they need to connect these shapes with arrows, unless they still want to add a statement in the middle. Also the student still has to choose the right shape to use for said node out of a large array of shapes. In other words, he is still forced to know the syntax of flowcharts by heart, structure isn't coerced and it takes a lot of tedious work to visualise a small thinking step. All of this is a logical consequence of the fact that draw.io is designed for general purpose drawing, rather than specifically drawing flowcharts.

This tool has in fact been used in the course *Algoritmisch denken*, so it is in fact the predecessor of our tool. During the interviews the students commented on their previous tool with the following "[The new tool is] *Nicer than the other tools we had for drawing. With them it took very long to find the right thing and to drag it into the right place.*"². We therefore look for other options.

Alternative tools similar to ours are Flowgorithm³, Hour of Code⁴, Scratch⁵ and Lego Mindstorms[13]. Contrary to draw.io, these are specifically built to create flowcharts. Also, they can directly execute the built flowchart. Expanding on that, they can execute step-wise, while showing the current value of all parameters and the position of the execution in the flowchart. Lastly, the flowchart can be converted to multiple kinds of functioning programming code. All of this can be done because the text in the flowchart has a correct syntax and can be interpreted by the compiler.

This brings us to our main disadvantage: Flowgorithm is too zoomed in for the *Algoritmisch denken* course. The reason we would like to use flowcharts is because they function as an abstract sheet that can be used to keep overview while programming. The idea while creating such flowchart is that you do not have to worry about implementation details such as boundaries and syntax. In Flowgorithm however you do need to worry about these things, otherwise the flowchart wouldn't be able to compile. Aside from that there are also studies that question these tools and see little change in the students motivation[14] or programming skill[15]. We therefore went for our own take on the matter and created our own tool.

7.2 Future work

The main focus of this work consisted of the interaction between a student and this tool in order to build flowcharts. There are two ways to continue on this work.

²Original: "[De nieuwe tool is] *Fijner dan die andere tooltjes die we hadden om te tekenen. Daar duurde het heel lang voordat je het goede had gevonden en dan zeg maar in de goede plek had gesleept.*"

³Flowgorithm - Devin Cook <http://www.flowgorithm.org/>

⁴Hour of Code - Code.org <https://hourofcode.com/>

⁵Lifelong Kindergarten Group - MIT Media Lab <https://scratch.mit.edu/>

First off, one could directly work further on this thesis by improving on this interaction. This work has shown several technical problems that hinder the interaction, such as the focus on text boxes, that could be resolved. Another way is to implement new features such as an undo- and redo-button or keyboard shortcuts allowing the user for greater control, freedom and flexibility.

Secondly, one could look more at improving the learning of the student through other means than directly building flowcharts. One example of this is the introduction of notes. Students could use these to note the program state or to show the teacher which parts they find difficult. Also, the teacher could use notes to make corrections for the students.

Bibliography

- [1] R. Smetsers-Weeda and S. Smetsers, “Problem solving and algorithmic development with flowcharts,” in *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, ser. WiPSCE ’17. New York, NY, USA: ACM, 2017, pp. 25–34. [Online]. Available: <http://doi.acm.org/10.1145/3137065.3137080>
- [2] A. Robins, J. Rountree, and N. Rountree, “Learning and teaching programming: A review and discussion,” *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003. [Online]. Available: <http://www.tandfonline.com/doi/abs/10.1076/csed.13.2.137.14200>
- [3] J. Chetty and D. van der Westhuizen, “Towards a pedagogical design for teaching novice programmers: design-based research as an empirical determinant for success,” in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. ACM, 2015, pp. 5–12.
- [4] E. Rahimi, E. Barendsen, and I. Henze, “Identifying students’ misconceptions on basic algorithmic concepts through flowchart analysis,” in *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*. Springer, 2017, pp. 155–168.
- [5] S. Chen and S. Morris, “Iconic programming for flowcharts, Java, Turing, etc,” in *ACM SIGCSE Bulletin*, vol. 37, no. 3. ACM, 2005, pp. 104–107.
- [6] J. Nielsen, “Enhancing the explanatory power of usability heuristics,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’94. New York, NY, USA: ACM, 1994, pp. 152–158. [Online]. Available: <http://doi.acm.org/10.1145/191666.191729>
- [7] B. Eich, “JavaScript at ten years,” *SIGPLAN Not.*, vol. 40, no. 9, pp. 129–129, Sep. 2005. [Online]. Available: <http://doi.acm.org/10.1145/1090189.1086382>
- [8] D. Crockford, *JavaScript: The Good Parts: The Good Parts*. ” O’Reilly Media, Inc.”, 2008.

- [9] E. Czaplicki, “Elm: Concurrent FRP for functional GUIs,” *Senior thesis, Harvard University*, 2012.
- [10] E. Czaplicki and S. Chong, “Asynchronous functional reactive programming for GUIs,” *SIGPLAN Not.*, vol. 48, no. 6, pp. 411–422, Jun. 2013. [Online]. Available: <http://doi.acm.org/10.1145/2499370.2462161>
- [11] I. E. T. Force, “The JavaScript Object Notation (JSON) data interchange format,” <https://tools.ietf.org/html/rfc7159>, accessed: 2018-09-10.
- [12] L. Faulkner, “Beyond the five-user assumption: Benefits of increased sample sizes in usability testing,” *Behavior Research Methods, Instruments, & Computers*, vol. 35, no. 3, pp. 379–383, 2003.
- [13] B. Bagnall, *Core LEGO MINDSTORMS Programming*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.
- [14] F. KALELIOĞLU and Y. Gülbahar, “The effects of teaching programming via scratch on problem solving skills: A discussion from learners’ perspective.” *Informatics in Education*, vol. 13, no. 1, 2014.
- [15] W. I. McWhorter and B. C. O’Connor, “Do LEGO Mindstorms motivate students in CS1?” *SIGCSE Bull.*, vol. 41, no. 1, pp. 438–442, Mar. 2009. [Online]. Available: <http://doi.acm.org/10.1145/1539024.1509019>

Chapter 8

Appendix

8.1 Interview questions

We used a semi-structured interviewing technique. The initial questions are given below, follow-up questions can be found in the transcripts further up in this appendix. Since the interviews were in Dutch, the questions are also formulated in Dutch.

- Wat vinden jullie van de werking van het programma?
- Wat voor tool gebruikten jullie vorig jaar?
- Zijn er dingen in de tool waar jullie je aan storen?
- Wat vinden jullie er van als je elementen kan slepen?
- Zou je liever dit gebruiken of op papier werken?
- Is het *character* limit storend?

8.2 Interview transcriptions

8.2.1 Group 1

00:00

Researcher: Ok. Allereerst, wat vinden jullie er van?

Student A: Van wat?

Researcher: Heel die tool, heel dat klikprogramma.

Student B: Ik vond het wel handig, alleen die kruisjes enzo als je probeert er op te klikken dan gaat het meestal weer weg.

Researcher: Ja dat klopt. Daar ga ik iets aan doen. Verder?

Student C: Ik vind het overzichtelijk en handig en prima te gebruiken.

Student A: Nou ik vind het wel prima.

00:37

Researcher: Want ik ben wel wat dingen tegen gekomen, waar dat je je misschien een beetje aan zou kunnen storen. We noemen het bugs, visuele bugs. Dat tekst ergens doorheen loopt zoals hier. Dit is ondertussen gefixt, maar dit bijvoorbeeld nog niet, dat het kruisje half wegvalt. Of dat hier zo'n ifbox veel groter is dan het pijltje of dan de [interrupted]

0:55

Student C: en ook mag de lengte van de tekst, kan niet groter zijn dan een bepaald aantal letters. Dat vind ik ook irritant want sommige statements zijn iets langer dan anderen en dan is het niet lang genoeg voor het blokje.

Researcher: Klopt. Wat vinden jullie hier van? [Researcher shows "171120 Group 3 #1 [2:23, 160]"] Die dingen geven een beetje een lelijke overlap, dat er net iets weg valt. Is dat storend, is dat helemaal niet storend, helemaal prima?

Student B: Ja, dat vind ik niet heel erg storend.

Student A: Wat bedoel je dan precies?

Researcher: Dat hier dat lijntje niet helemaal uitkomt en normaal gesproken heb je dat hier nog lijntjes die mooi netjes aan de zijkant uitsteken en dat je er "true" of "false" hebt, alleen dat zie je hier helemaal niet, want daar valt dat ding helemaal overheen.

Student A: Nou ja, ik vind het niet heel erg denk ik. Het was me ook niet opgevallen.

Researcher: Oh, ideaal.

01:50

Researcher: Dan, de character limit, dat je niet verder kan typen soms in een blokje. Is dat heel storend?

Student B: Ja, soms is het nog een paar letters ofzo en dan denk je waarom kan dit er niet bij. Dan heb je toch wel veel meer nodig.

Student C: Terwijl als je op hoog niveau denkt, dan wil je langere zinnen maken en dat is dan lastiger.

Researcher: Eh... We doen dat eigenlijk, deels omdat je dan wordt gedwongen om met kleine zinnen te werken, kleine elementaire stukjes.

[Student D is a student that was interviewed 5 minutes earlier, and intervenes this interview]

Student D: Ik heb nog een suggestie dat als je op pijltje-terug klikt, dat ie dan iets vraagt van "Weet je zeker dat je de pagina wil verlaten?" want we klikten hem net weg en toen waren we alles kwijt.

Researcher: Ja dat is een goede, dat is een heel goede.

02:40

Researcher: Verder, werken jullie graag op papier of vinden jullie de tool echt iets van meerwaarde hebben?

Student C: Ik heb een lelijk handschrift, dus ik vind het fijn om op de computer te werken. Dat is wel handig.

Researcher: Maar als je later iets in de informatica gaat doen ofzo, dan ga je niet nog even naar deze webpagina toe om nog gauw even iets in elkaar

te steken?

Student C: Is op zich wel handig.

Researcher: Ik ga hem nog wat bijschaven. Hey heel erg bedankt!

8.2.2 Group 2 and 3

00:00

Researcher: Ja, wat vinden jullie er van? Eerste open vraag. Vind je het een beetje fijn?

Student A: Ik vind het een prima tooltje, alleen ja een paar vervelende puntjes waar ik hem dan aan herinner.

Student B: Fijner dan die andere tooltjes die we hadden om te tekenen.
[instemmende ja]

Student B: Daar duurde het heel lang voordat je het goede had gevonden en dan zeg maar in de goede plek had gesleept.

00:30

Researcher: Wat voor tool hadden jullie vorig jaar?

Student A: Je had allemaal figuurtjes gewoon. En dat was niet gewoon dat je aan kan klikken wat wat is, maar dan had je gewoon een rondje en een driehoekje...

Researcher: Dan moet je nog zelf onthouden welk vormpje waar bij hoort?

Student A: Dat ook nog en dan moet je ze nog naar de goede plek slepen en pijlen trekken. Ja, dat was wel minder, maar dit werkt een stuk beter.

00:54

Researcher: Er zijn wat bugs, zoals bijvoorbeeld als je in een while loop zit dat je geen plus-knopje er onder staat. Die zijn jullie al tegen gekomen en jullie bij een if in een while. Sorry, daar moet inderdaad een knopje zijn.

Verder heb ik... Zijn er nog andere dingen waar jullie je aan storen? De opmaak, de, de..

01:19

Student B: Ja, je kan niet zo veel typen in zo'n vakje

Student C: Ja, en het is daardoor ook stel je moet heel veel variabelen definiëren. Stel je moet er 5 definiëren. 5 van die grote vakken dat is meteen echt heel [wordt onderbroken]

Student A: dat je er meerdere onder elkaar kan zetten ook.

Student C: Ook al is het niet hetzelfde ding. Het is meer van links naar rechts, maar ook gewoon meer regels.

Researcher: Dat houdt het wat compacter ja. Maar tegelijkertijd dwingt het formaat een beetje dat jullie alles in kleine elementaire stapjes moeten doen.

Student C: Ja, maar dat is een beetje te erg denk ik.

Researcher: Te erg ja.

Student C: Natuurlijk hoeft je geen hele volle zinnen uit gaan schrijven enzo, maar dat snap je denk ik zelf ook wel.

Student A: Misschien "dit was de eerste klasmethode" en dan de volgende

keer mag je wat meer tekst geven. Dat ze eerst leren dat ze compact moeten schrijven, want we zijn nu wel een beetje...

Student C: Ja, dat helpt ja.

Researcher: Eerst mensen compact leren schrijven. Twee versies van de tool maken met zo'n klein vinkje rechtsboven.

Student A: Twee versies ja, haha.

02:28

Researcher: Hebben jullie de pre- en de postconditie nog gebruikt ergens tussendoor?

Student B: Ja één keer.

Student D: Één keer, maar niet echt.

Researcher: Daar zit ook nog een leuke bug in, die hebben jullie nog niet opgemerkt zo te zien.

Student B: Nee.

Researcher: Je kunt hier ook typen in zo'n vakje, dat is cool. [text disappears on overflow]

02:52

Researcher: Heeft een van jullie ooit... Wat vinden jullie er van als je elementen kan slepen? Dat je gewoon even dit stukje er uit kan slepen.

Student A: Dat je het even apart kan zetten en dan later er in. Dat is handiger misschien, want nu hadden we dat plusje, toen hadden we uiteindelijk uitgevonden dat er ook nog een textblokje er onder kreeg als je bij het object er boven weer op plusje drukt en dan weer gewoon, een while-loop was het volgens mij, neer zette en moet je daarna weer alles overtypen.

Student C: Ja dus dat je zeg maar we hadden dan... [interrupted]

Student A: ... dat plusje dat in het midden onderin wat.. [interrupted, Student A en Student C are talking simultaneously]

Student C: Voor de if een statement gezet en daar voor weer een if en dan alles...

Researcher: ... overtypen.

Student C: Dan werkt het wel, maar dat is wel een omweg.

Student B: Als je dan een stapje bent vergeten dat je dan weer terug kan eventueel, is misschien handig.

Student C: Of uberhaupt dat je ook niet per se alleen apart kan zetten, maar ook gewoon kunt verslepen binnen het stroomdiagram. Dat lijkt me sowieso handig. Dan hoeft je eigenlijk nooit meer iets over te typen.

04:02

Researcher: Dan heb ik nog... De vraag: ik heb van de vorige keer een screenshot gemaakt. Dat deze tekst hier doorheen liep. Dat is nu gefixt.

[Researcher shows "171120 Group 1 #1 [1:07, 640]"]

Researcher: Maar sommige dingen zijn niet gefixt. Zoals hier

[Researcher shows "171120 Group 3 #1 [2:23, 160]"]

Researcher: dit kruisje valt half weg, waarom weet ik zelf ook nog niet helemaal, maar hij valt half weg.

04:25

Researcher: Dat was bij jullie trouwens [points at group 3] ook bij jullie was, ja hier staat de debugger open. Als hier zo'n if bijvoorbeeld veel breder is getypt, dan komt hij over de lijntjes. Stoort jullie dat of maakt het eigenlijk helemaal geen bal uit?

Student D: Nee

Student C: Nee, is niet echt belangrijk

Researcher: Het idee is duidelijk, dus alles is top?

[students mumble agreeingly]

Student D: Hier, het kruisje, als het lastig te klikken is, dan kan het irritant zijn, omdat de lijntjes niet zo perfect lopen.

Student A: Dat is voor mensen met zware OCD.

Researcher: Welkom bij informatica.

05:04

Researcher: Dan heb ik nog een knopje er bij verzonnen afgelopen week, omdat ik die hitbox niet kon fixen dacht ik: ik ga jullie afleiden met andere mooie knopjes. Heeft één van jullie dat knopje al gebruikt? Wat stond er op dat knopje?

[silence]

Researcher: OK, ik zou zeggen, probeer hem straks even uit en kijk of dat het bevalt, want ik merkte dat jullie te veel overtypen enzo, dus dat is naar. Test hem even uit voor me en zeg "ooh dit is stuk", dan kan ik weer vooruit. Vind ik fijn.

05:32

Researcher: Verder nog één laatste vraag: zou je liever dit gebruiken of op papier werken?

Student A: Ik vind dit wel overzichtelijk

[students mumble agreeingly]

Student C: En je hoeft niet meer te gummen enzo, dus het gaat wel sneller

05:48

Student B: Ik vind het wel fijn dat je nog iets makkelijk er iets bij kan schrijven, misschien is dat ook nog wel een goede toevoeging.

Researcher: Wat voor dingen zou je er bij willen schrijven? Want dit [interrupted]

Student B: Misschien als je echt grotere stroomdiagrammen maakt, dat je dan er bij kan zetten van "dit doet dit" en "hier moet ik nog even naar kijken" of "denk hier bij aan verwijzing naar een ander stroomdiagram dat je hebt gemaakt"

Student A: Je hebt ook bij de eerste en de laatste een tekstblokje er naast staan waar je waarden in kan geven. Misschien kun je dan een plusje aan de rechterkant maken dat zo'n blokje er bij geeft

Researcher: Oh dat ziet er wel gelikt uit. Ja dat is een goede, hier had ik nog niet over nagedacht. Ja, cool, bedankt!

8.3 Video quotes

Only the relevant quotes are transcribed and given with a time indication.

8.3.1 Pre-test: Group 2

4:25 “We hebben een probleem. Hij wordt niet groter als de tekst er niet meer oppast.”

19:45 “Als je backspace doet dan doet ie dat [wijst naar computerscherm]. Dan doet ie een pagina terug. Alleen dat doet hij ook als je gewoon wel in het vakje zit.”

25:28 “Kan je dat niet slepen ofzo?” “Nee.”

25:40 “Oh, knippen en plakken werkt hier ook niet.”

8.3.2 Pre-test: Group 3

6:45 “OK laat maar, het is goed”

8.3.3 Main test: Group 2

18:00 “Die kan niet weg.”