From Sketches to Functional Prototypes

Extending WireframeSketcher with Prototype
Generation

Fredrik Haugen Larsen

Master of Science in Informatics
Submission date: December 2014
Supervisor: Hallvard Trætteberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NTNU – Trondheim
Norwegian University of
Science and Technology

From Sketches to Functional Prototypes:

Extending WireframeSketcher with Prototype Generation

Fredrik Haugen Larsen

Master of Science in Informatics

Submission date: December 2014
Supervisor: Hallvard Trætteberg, Associate Professor, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description:

When developing user interfaces (UI) the designers often utilize prototypes of varying *fidelity* (how real and functional it is), e.g. sketches and more functional prototypes. In order to save time it would be practical if *low fidelity* prototypes, such as sketches, could be quickly developed to a prototype with higher fidelity and functionality.

There are several Eclipse based tools for prototyping among them WireframeSketcher for creating wireframe sketches. This project should explore how such a tool can be extended in order to turn a sketch into a functional prototype.

Assignment given:	January 15th 2014
Supervisor:	Hallvard Trætteberg, Associate Professor, IDI

Abstract

When designing graphical user interfaces (GUI) for software applications an important part of the process is prototyping. Initially layout and interaction is often more interesting than the actual look of the design itself. There are several tools available that allow users to create simplistic *low fidelity* sketches which are fast and easy to work with. The problem arrives when developers need to implement the design in order to create *functional prototypes* - prototypes that also include some of the functionality of the planned application. The underlying sketch-design data is normally not compatible with the tools used by the developers, so it gets discarded and the same design has to be recreated by the developers which affects costs and efficiency. This process works poorly.

In this thesis the prototyping tool WireframeSketcher has been extended with information that makes it possible to generate a functional prototype to a certain degree from a sketch. The solution is implemented using Eclipse and consists of two programs. One program handles the parsing of sketches and generation of data required for the functional prototype, while the other program is responsible for running the functional prototype itself. The solution utilizes simple graphical *notes* within WireframeSketcher in order to augment the sketch with information regarding functionality. This information is interpreted by the programs and the result is a functional prototype. After development a set of qualitative user-tests were conducted and they indicate that the result is a viable solution for turning sketches into functional prototypes.

Index terms— prototyping, functional prototype, wireframe, interaction design, sketch, design, javafx, code generation, fxml, xtend, emf, wireframesketcher, eclipse

Sammendrag

Når man designer grafiske brukergrensesnitt (GUI) for programvare-applikasjoner er prototyping en viktig del av prosessen. I starten er det mer interessant hvordan utforming er og hvordan interaksjon fungerer, enn hvordan det grafiske designet ser ut. Det finnes mange tilgjengelige verktøy som tillater brukere å lage forenklede lav-fidelitet (gjengivelse) skisser som er raske og enkle å arbeide med. Problemet oppstår når utviklere må implementere designet for å lage *funksjonelle prototyper* — prototyper som også inkluderer litt av funksjonaliteten til den planlagte applikasjonen. Den underliggende skisse-dataen er normalt ikke kompatibel med verktøyene som utviklerne bruker, så den forkastes og det samme designet må lages på nytt av utviklerne, noe som påvirker kostnad og effektivitet. Dette fungerer dårlig.

I denne oppgaven har prototypingsverktøyet WireframeSketcher blitt utvidet med informasjon som gjør det mulig å generere en funksjonell prototyp til en viss grad fra en skisse. Løsningen er implementert i Eclipse og består av to program. Det ene programmet håndterer tolkning av skisser og genererer data som er nødvendig for den funksjonelle prototypen, mens det andre programmet har som ansvar å håndtere utførelsen av den funksjonelle prototypen. Løsningen benytter enkle grafiske *notislapper* inne i WireframeSketcher for å utvide skissen med informasjon som beskriver funksjonalitet. Denne informasjonen tolkes av programmene og resultatet er en funksjonell prototyp. Etter utviklingen ble et sett av kvalitative brukertester gjennomført. Testene antyder at resultatet er en potensielt brukbar løsning for å gjøre skisser om til funksjonelle prototyper.

Contents

Preface

This master's thesis was written at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU). It is the culmination of research, study, empirical trial and error, prototyping, development, user-testing and discussions that took place during 2014. It has been a challenging and very interesting year, and I have learned a lot about myself in addition to the technical aspects of the project.

I would like to express my deepest gratitude towards my supervisor Hallvard Trætteberg. Due to the nature of this thesis, his knowledge and help has been invaluable. Over the course of the last year we have had a lot of meetings discussing both general subjects and subjects strictly related to the thesis. They have all been interesting and is partially responsible for making this a fun project to work on. Thank you for being patient and understanding.

I would also like to thank Magnus Jerre for assisting me during the user-testing process. I hope some of my work will prove useful for his own thesis. His presence and smalltalk at *Fiol*, the room where I have done most of my work, has been motivating. On that regard I would like to thank everyone who has been studying in the room. It is a great incentive to have students around you who are hard-working.

Thank you for the experience,

[illegible]

Trondheim, December 7^{th}, 2014

List of Figures

Chapter 1 Introduction

In a time when computer applications, be it desktop or mobile, are becoming more and more intertwined with our lives, the importance of prototyping is increasing. Because the number of computer users increase, it is now more difficult than before to develop solutions that conform to each specific target group. It is crucial for designers and developers to test the underlying ideas behind the application without spending unnecessary time and work on actual programming and implementation. This also helps to reduce development costs. Prototyping can be used to better customize the solution for the specific target group. It can be used together with an iterative process where revisions and refinements become results of testing and evaluation. Prototyping and the iterative process is an accepted way of working in the field. Traditionally the prototypes are of a *disposable* nature. They are created, used for testing and then thrown away. Most times this is because there are different classes of prototyping tools depending on how *functional* they are, and there is no practical way of transitioning between one class to another. The prototyping process could become more agile by reusing the underlying prototype data in order to make a smooth transition between each class.

1.1 Motivation

When the author attended the graphical design course IT3402 [2] at NTNU in 2013 the focus was on the process behind developing GUI prototypes. One of the experiences during the course was that even though one would traditionally use disposable paper prototypes, a lot of different digital tools are now available and they are of different complexity. Most of them utilize a simple WYSIWYG [1] architecture that enables designers to drag and drop the elements (e.g. buttons, labels) they want in their design. The sketches are also known as *wireframes*, due to the nature of their layout. There are even simpler tools that only support images, but most tools

[1] WYSIWYG stands for *What You See Is What You Get*. The finished layout should look very similar to what is displayed in the editor - preferably identical.

support some kind of navigation between the designs. The end product from such a prototyping tool is a series of static sketches that can be navigated amongst. It became obvious that there are several possible improvements related to current prototyping tools. The group experienced that when creating an interactive prototype that allowed users to navigate from one design to another it required a lot of *duplication*. If a design represented the screen state before there was e.g. text in a field, another duplicate of that design had to be made where the field was populated. Creating an interactive prototype that emulates a real program and presents the interaction design generated a lot of duplicates using this method. Although many prototyping tools support different levels of fidelity, the process of designing a GUI normally starts out as a low-fidelity design. The designers focus on layout placement and interaction. The graphical design itself i.e. how a button looks is not important this early on. When the solution meets the requirements many choose to implement a functional prototype, which implement strategically chosen parts of both the user interface and the functionality of the planned application. [3]

The problem is that even though the functional prototype use the same design created in the prototyping tool, it usually has to be implemented from scratch by developers. The underlying data of the design is discarded, because it is incompatible with the tools used by the developers. This forces the developers to *recreate* the design which is a duplication of effort. This disposable approach is time consuming and decremental to the development process. If a functional prototype could be implemented directly from a prototyping tool, the process would not require the duplication of effort associated with the design, it would be more efficient and it could also allow less technical users such as designers to perform the task. The goal is to be able to create something that is more functional, without loosing the strength of the approach which is simplicity. Could such a solution also be used to create a finished application? At least this thesis will explore ways to extend prototype sketches with functionality.

1.2 Research Questions

This thesis will explore how a wireframe sketch can be transformed into a functional prototype by extending a low fidelity sketching tool with information. There are various aspects of a functional prototype that should be explored as well. It is interesting and necessary to see how it would be structured for a modern target group. Before extending a sketching tool, some study of which parts of such a tool is important should be used as rationale. There are most likely multiple ways of implementing a solution, so the specific implementation itself is not that interesting, but it is interesting to see how a modern toolkit can be used in such a project. One of the advantages with low fidelity sketching tools is that they are simplistic by nature. It is important to retain this simplicity.

This can be concretized into the following research questions (RQ):

- *RQ 1: How can a low fidelity sketching tool be enriched with information that enables generation of a functional prototype.*
    - *RQ 1.1: How can functionality be added to a sketch without loosing the simplicity inherent in low fidelity sketching tools.*
    - *RQ 1.2: How can the added information be utilized together with a modern toolkit in order to realize a functional prototype.*

In order to answer the research questions it is helpful to look at how other tools work and which functionality they provide to the user (designer). Depending on the technical requirements of the chosen tools a state-of-the-art and modern toolkit should be used. A fitting research methodology should be used in order to help the development process.

1.3 Structure of the report

This report aims to present the rationale behind the various choices made to answer the research questions, and discusses the two programs made during this project. The programs and choices are evaluated against the research questions and against data from user-tests. The report has been written in such a way that it should be possible to jump directly to a specific chapter or section. Most sections refer to relevant information, but it is recommended that the reader look through the first chapters before continuing.

- **Chapter 2** discusses the rationale behind the chosen research methodology and how it was executed.
- **Chapter 3** presents general background information about the field. It gives an introduction to what prototyping is and how it is related to iterative design. It also covers prototyping tools and other related subjects that the reader should be familiar with before reading on.
- **Chapter 4** discusses the rationale behind and the technologies used to create the thesis programs.

The development process is divided among three chapters.

- **Chapter 5** discusses the intial development and includes implementation specific theory.

- **Chapter 6** presents the design related work.
- **Chapter 7** presents the technical choices and implementation. Both chapters 6 and 7 are separated into two main iterations. Figure 1.1 shows a matrix that depicts the way it will be presented.
- **Chapter 8** presents an analysis of typical JavaFX implementations compared to the *generated* implementation.
- **Chapter 9** evaluates the design based on the user-tests and it also evaluates the system based on the research questions. It suggests further development and include the conclusion.

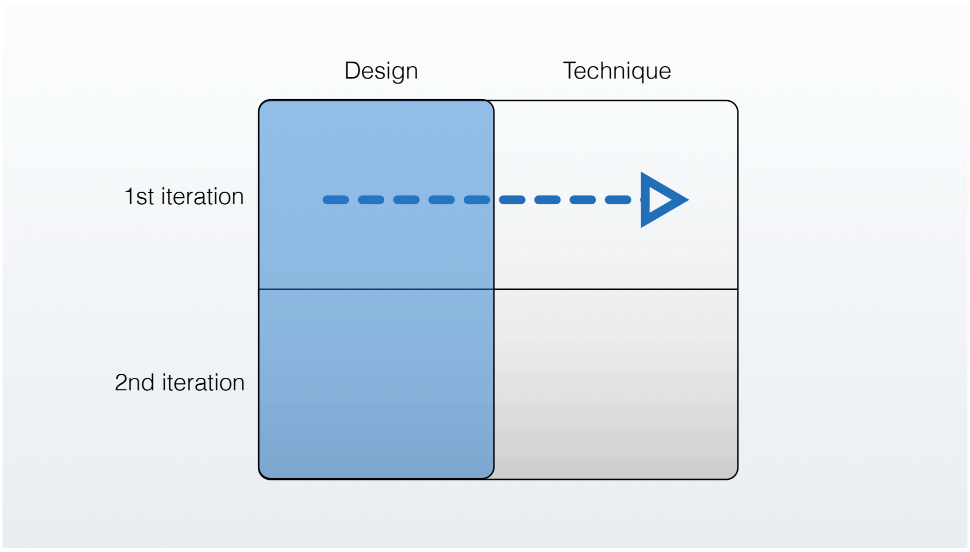


Figure 1.1: An overview of how the development process will be presented. First the design will be presented in both iterations, then the technique will be presented in the same manner.

Chapter 2

Methodology

2.1 Design science

The methodology used in the thesis is the *Design Science Research Method*. It is an outcome based research method which provide project guidelines. In the words of March & Smith: *whereas natural science tries to understand reality, design science attempts to create things that serve human purposes.* [4] Wieringa generalize this to also include attempts to improve *existing* things to serve human purposes better. [1] The generalized method is depicted in Figure 2.1 which outlines the various process steps. It revolves around a *regulative cycle* which starts by investigating a problem. The problem can be the outcome of a solution to an earlier problem. Then it specifies solution designs and validates them before implementing a selected design. The implementation can then be evaluated, which could be the beginning of a whole new turn trough the regulative cycle. [1]

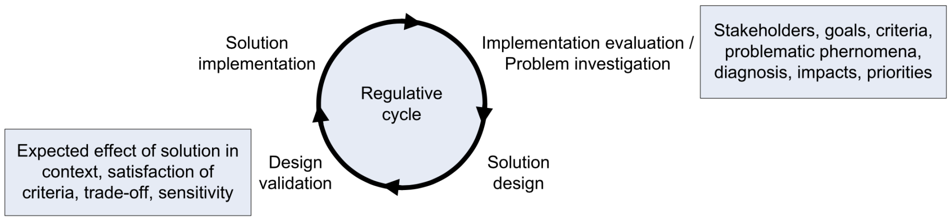

Solution implementation
Implementation evaluation / Problem investigation
Stakeholders, goals, criteria, problematic phernomena, diagnosis, impacts, priorities
Regulative cycle
Expected effect of solution in context, satisfaction of criteria, trade-off, sensitivity
Design validation
Solution design

Figure 2.1: The process of design science. Wieringa, 2009 [1]

Since a major part of this thesis consist of developing a prototype program, design science is a viable choice. It provides an empirical approach for solving the problem.

2.1.1 Using the method

The initial project description together with several discussions with the supervisor was a basis for understanding the problem. The experiences of attending the design course IT3402 [2] also made it easier to relate to the problem, but since it was quite technical and depended on several new concepts it took some time learn. Figure 2.2 present a simple way to decomposition the problem into what Wieringa refers to as *Knowledge questions* and *Practical problems.* [1] In the process it was no specific structure to how knowledge problems and practical problems were approached, other than what was discussed on the meetings. In the beginning the biggest challenge was answering knowledge questions trying to figure out what the problem was. It should be emphasized that there was not just *one* problem to decompose, but it started with one general problem. The technique was used for the subproblems as well.

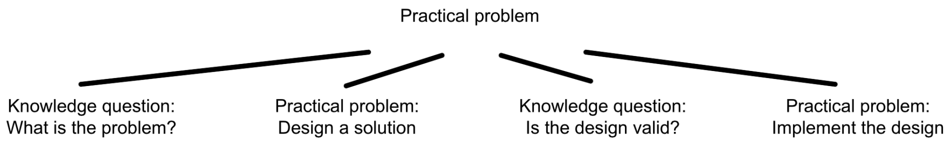

Practical problem
Knowledge question: What is the problem?
Practical problem: Design a solution
Knowledge question: Is the design valid?
Practical problem: Implement the design

Figure 2.2: A simple decomposition of a problem. Wieringa, 2009 [1]

Almost every week there was a meeting where both design related and technical issues were discussed. Due to the fact that the supervisor was the one who created the initial project description and that he had extensive background knowledge about the problem, the meetings became valuable for both understanding the problem and learning about new technologies.

Based on the problem several design solutions were discussed during the meetings. The solutions were typically of a general nature, but for specific problems the solutions could be specific as well. The supervisor would often provide information on how to solve small technical problems related to the specific technologies dictated by the project description. These problems were deemed *not important* by the supervisor, since they were outside the scope of interest. This allowed more time to be spent on solving the important problems.

Between each meeting it was attempted to implement one or more solutions, independently of the supervisor. In the beginning a lot of trial and error was required to do that. But the process of trying to implement a solution usually generated more ideas about the overall design. And the new ideas could be discussed on the next meeting together with an evaluation of the implementation. Many of the meetings revolved around problems from earlier meetings, but the meetings served as an important place for discussion. During the project two *main* iterations was defined based on pre or post a graphical input mechanism. The differentiation is made in order to easier summarize a lot of work into two distinct sections.

Evaluation

Choosing the design science methodology turned out to be a good choice. It's difficult to imagine how a different methodology could have worked out, but the processes described in the design science method felt natural and non-intrusive. The iterative process used to design, implement and evaluate the prototype was very helpful. In total the method was a good fit for the project.

Chapter 3 Background

3.1 Prototyping

The word prototype derives from the Greek *prototypon* meaning ”primitive form”. From *protos* - ”first” and *typos* -”impression”. [5] Thus, a prototype can be a first draft or proof of concept of a physical object e.g. bicycle, car, tool or software in the form of a computer program. Prototypes are not necessarily used to test all the aspects of a product at once. But rather to test one or a few specific aspects e.g. the design of a car door, or how a photo application should allow the user to add pictures into an album. This narrows the focus of development and allows for faster iterations, resulting in a product that is better and more thoroughly tested. Different fields of study prototype in different ways. It also depends on what materials they work with. Some materials, such as metals, are expensive and difficult to manipulate. Using plastics or other cheaper materials can speed up the process and reduce costs, which is the goal in any field. Nowadays the big hype when creating tactile prototypes is 3D printing. It enables rapid prototyping by allowing designers to transform the computer aided design (CAD) directly into a tactile prototype.

Figure 3.1 depicts an aerofoil with its nomenclature which can be modified to change the aerodynamic properties of the foil. When prototyping an aircraft, many would argue that aerodynamics is one of the most important factors. Prototyping the aerofoil separately from the rest of the aircraft, allows the engineers to find optimal aerodynamic properties for the foil before continuing. Only prototyping parts of a product can be generalized to most prototyping scenarios.

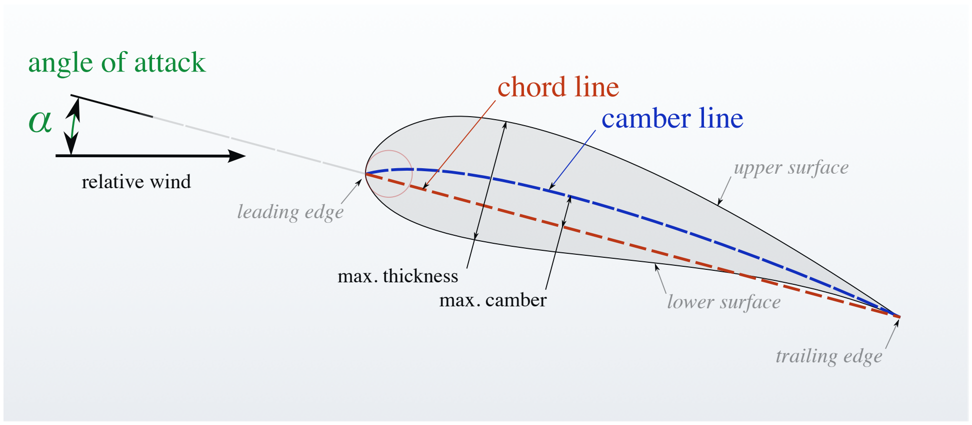


Figure 3.1: Prototyping in general usually focus on one or a few aspects of the product on which they prototype. When designing an aircraft, there are multiple properties of the aerofoil that can be changed to improve the aerodynamics.

Prototyping in general is a process which utilize prototypes in order to develop and refine a design. In software development there are several elements of a program that can be prototyped e.g. Software architecture, but prototyping the graphical user interface (GUI) is probably most common. It is usually performed using tactile materials like paper-mash and paper, or digitally using computers. The paper prototypes usually represent a program in low fidelity (i.e., sketched by hand), while computer prototypes increases fidelity toward the higher end. But there are no rules against having high fidelity paper prototypes and low fidelity computer prototypes. Interestingly enough, studies have shown that users respond more openly to lower fidelity designs. [6] Another proceeding advocates the use of low-fidelity prototypes because they are cheaper to create and easier to iterate. Using low-fidelity prototypes allows the designers to focus on *interaction design* and *information architecture* instead of the graphical design itself. This is an important aspect of GUI prototyping. When the designers create the early revisions, it is not how pretty or polished the design looks that they want to prototype, it is the layout structure and how a user would interact with the design that is interesting. The group found that both paper and computer prototypes perform equally well, but acknowledges the advantage of being able to automatically record and distribute the test electronically, if a computer is used. [7]

Figure 3.2 shows two GUI prototypes. The left hand-drawn design is in very low fidelity and the other is in slightly higher fidelity.

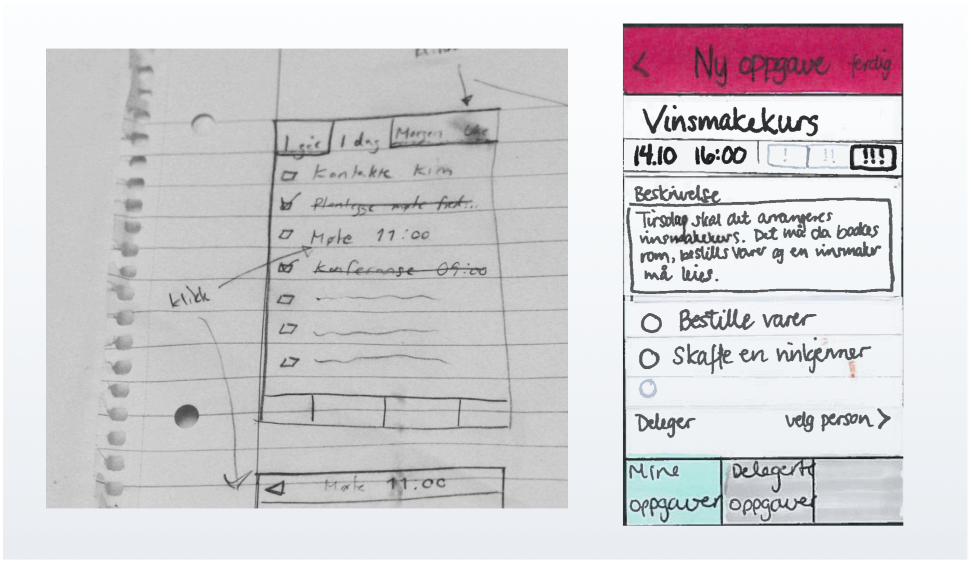


Figure 3.2: Two prototypes with different levels of fidelity. The left one is very low, while the right one is higher, but still not *high* fidelity.

In software development the process of GUI prototyping is very powerful, because it allows the developer to test the layout of the product without spending time on the actual programming. The design can be presented to a group of selected people who individually attend a usability test together with a person responsible for the test and an observer. [1]. When the results of the tests are evaluated, the design can be revised quickly without the need of any extensive programming. It also allows for graphical designers with little technical computer experience to create prototypes and test designs without having the skills required to program.

3.2 Iterative design

Prototyping is normally part of an iterative design cycle. A cycle consisting of *designing* (developing) the prototype, *testing* it using usability testing and *evaluating* the results from the test. Then refining the design according to the evaluation, before continuing with testing, evaluation etc. The goal is to have iterations that span over a relatively short time, e.g. two weeks, allowing for changes to the design to be tested and refined quickly.

[1] Usability testing is a process where test subjects interact with a prototype as if it was a finished product. Facilitators handles the behavior of the prototype to create the illusion of a functional program. It allows the developers to test their designs quickly and refine them if necessary.

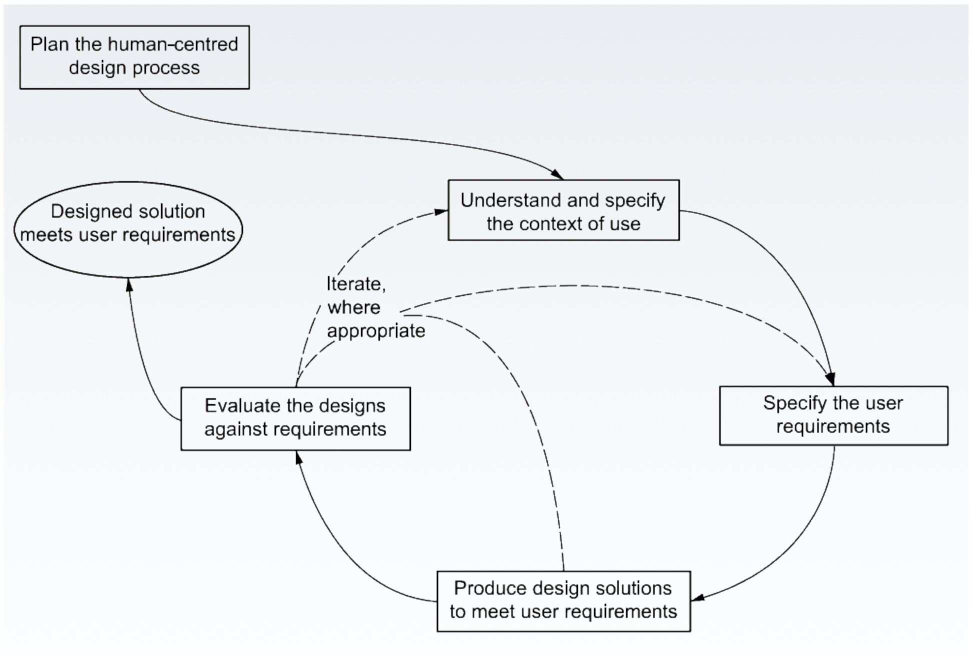

Plan the human-centred design process
Understand and specify the context of use
Designed solution meets user requirements
Iterate, where appropriate
Evaluate the designs against requirements
Specify the user requirements
Produce design solutions to meet user requirements

Figure 3.3: ISO 13407: Human centered design. An iteractive design cycle. The design can be redefined after each evaluation.

It is important to mention that the final solution is usually not the last solution to be found. In order to settle on one solution, the developers should try different alternatives. Normally a solution will be found that is acceptable, but the only way to find out if it is the best solution is to continue and compare the results. Figure 3.3 illustrates the process on an axis from *problem* to *solution*. Thus the final solution is normally found before the last solution.

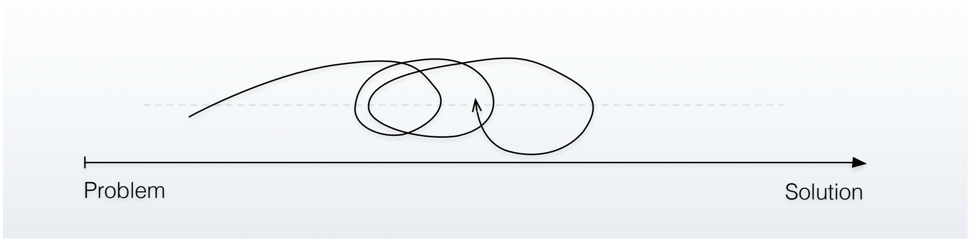


Figure 3.4: The typical iterative process usually settle on a solution that is found before the last.

3.2.1 Evolutionary prototypes

A full fledged prototype of a computer program or system normally ends up being a finished, working product. This process contain several design iterations that modify the prototype in sequence. The iterative process stops when the designed solution (prototype) meets the user requirements. In practice the prototype consist of several smaller prototypes. Like how prototyping an aircraft would include the prototyping process of the aerofoil in Figure 3.1. Not all prototyping processes result in a finished product though. E.g. if the process involves only prototyping the GUI of a computer program the result is not a finished, working product, but rather an important aspect of such a product. Traditionally when prototyping a GUI the programmers would write code for it. When the GUI meets the requirements the result is a functional GUI. That would not be the case if the GUI was prototyped in a prototyping tool.

3.3 Prototyping tools

The use of prototyping in a development process can help to reduce development cost and result in a better end product. While prototypes can be made both digitally on a computer and on actual paper, studies have shown that even with the same level of fidelity, users prefer computer prototypes. [8]

3.3.1 Functionality coverage

There are several software tools available that can help a developer realize a prototype. These tools can be placed along an axis from *design* to *construction*. The farther on the right hand side the tool is the more it covers the user interface functionality. In other words, a static image of a design would fall on the left side, and a program that matches the prototype functionality would fall on the right side. As a reference, *paper prototype* is placed on the left side and *JavaFX* on the right.

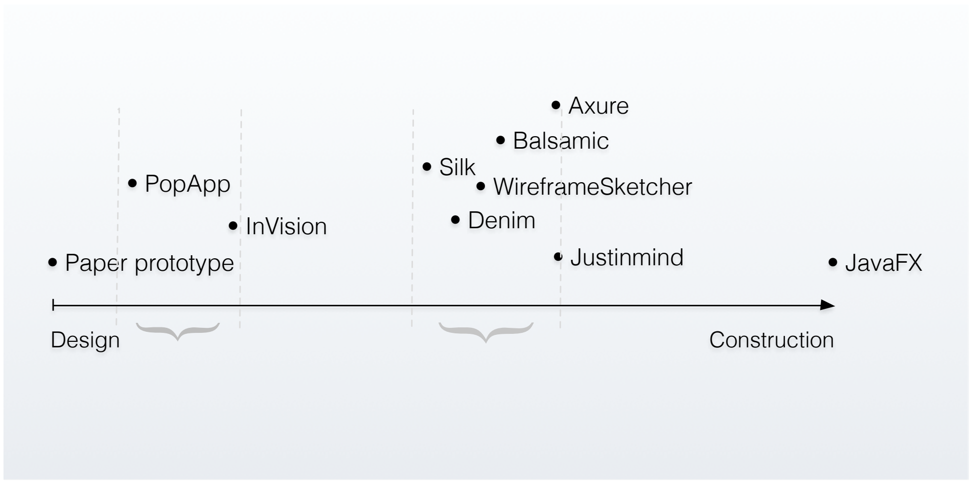


Figure 3.5: Tools placed according to how much of the interface functionality they cover. Notice that the tools are clustered inside the two regions. The y-axis is irrelevant.

In Figure 3.5 one can see that the tools are mostly clustered together in two regions. The voids represent regions where hypothetical tools could cover a different amount of the user interface functionality. Although this is just the authors opinion, one could argue that the reason for these voids are that adding just a little extra functionality coverage does not yield better results for the end user. The tools are either very simple, or require substantially more functionality in order to be useful. It is important to determine where on the axis the tool should be according to the target group. If it should cover more functionality it should be further to the right.

3.3.2 Different approaches

The feature set differ among the tools, and some are more advanced than others. Depending on what platform the developer is working with, some tools might be a better choice. E.g. PopApp supports only simple navigation among images which could be useful for sketching on paper, taking a picture of it and then navigate among those images. InVision implements swiping gestures, transitions and animations in order to better support smartphone application development. Justinmind and Axure RP also support state variables making it possible to navigate using conditional logic like user authentication, selected attributes, etc. Many of the tools have an overlapping feature set, but which tools fit the user best might be based on these features.

Here are some tools and their strengths.

- Tools that only display images and allow for navigation between different images e.g. PopApp[9] and InVision[10]. They are simple and require little to no explanation. The downside is limited functionality.
- Tools that allow for customized layouts using a WYSIWYG editor e.g. WireframeSketcher[11], Balsamiq[12], Justinmind[13] and Axure RP [14]. They allow the user to create a more realistic prototype but are a lot more complex and require some time to learn.

3.3.3 Program architecture

In programming

When programming an application with a graphical user interface there are usually several dimensions to be aware of. A normal implementation would contain a data model layer, a controller layer and a visual representation layer. The data model layer would represent and store all data required for the application. e.g. a *Person* class that can store the *name* of a person and his or her *Social Security number*. The visual layer (or view) would be the graphical interface between the user and the program. The view only updates its data representation when the model notifies that a change has been made. These changes are performed by the controller which get input from the user. This is normally referred to as the architectural software pattern *MVC* or Model-View-Controller and is depicted in Figure 3.6. MVC is a widely used and time tested software pattern.

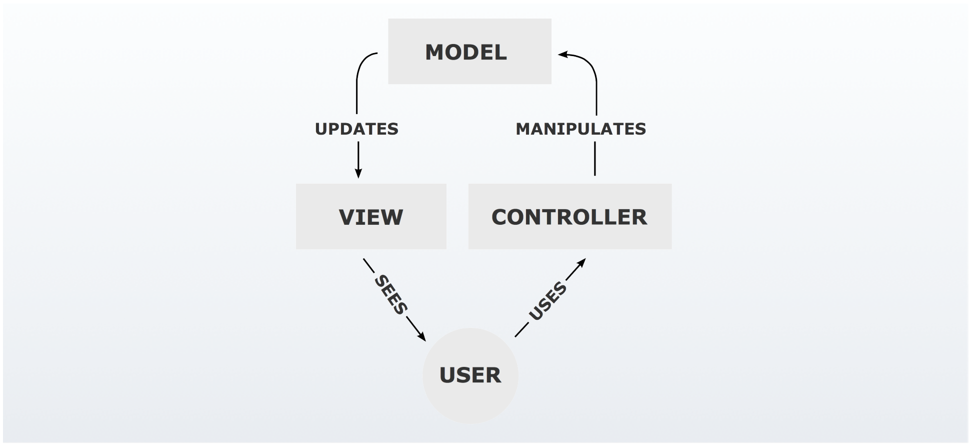


Figure 3.6: The Model-View-Controller software architectural pattern.

Typical prototyping tool approach

Not referring to the tool itself, but their prototypes - most prototyping tools do not conform to the Figure 3.6 above. They typically exist just as one *view* with a very simplistic navigation controller. Many would argue that the point of prototyping is to reduce the amount of coding (if any) to a minimum so that the new design can be tested as soon as possible and design iterations can become more frequent. Depending on which tools are used some of the dimensions above are still relevant. e.g. Prototypes in PopApp and InVision uses a view to convey the graphics to the user. A small controller handles the linking between other graphics. In e.g. Axure RP and Justinmind the prototypes also can have state variables. That is, the prototype can have a state e.g. *logged in* which could affect the workings of the prototype. These state variables would be stored in the prototype's data model.

With that in mind using the MVC pattern as a guide for the prototype's software architecture makes sense. Having a prototype tool with an extended controller functionality in addition to a data model could provide useful.

3.4 Model-driven Engineering

Model-driven Engineering (MDE) is a software development methodology that utilize abstract representations of a particular application domain, named *domain models*, instead of the more usual *code-driven* programming approach. [15] Model-driven engineering is centered around the model. The domain model describes entities with attributes and roles. It also identifies the relationships among all the entities within the scope of the problem domain. In e.g. UML, a class diagram is used to represent the domain model.

The Object Management Group has developed a set of standards called *Model-driven Architecture* (MDA), although unlike Eclipse EMF, described in section 3.6, it is not a concrete framework. [16]

3.4.1 Model-to-text transformation

When working with models the developer might want to manipulate the data in other ways than just modifying the model. This is particularly true when generating code (see section 3.4.2). In order to retrieve a textual representation of the model, a model transformation has to be performed. In the case where the output is text this transformation is called a *Model-to-text transformation* or *M2T* for short. It is a common transformation due to text being a ubiquitous format that is used by many tools as either source, intermediate or target format - or a combination of them all.

3.4.2 Code generation

There are different ways to generate code. Some tools can generate different class files from a model using M2T transformation in order to supplement or substitute the model. In MDE this is the most common way to generate code. Generic code can also be generated, and many programs do that. That could be HTML files based on a template and a set of input data e.g. a list of people. Or it could be program code based on system dependancies, or some other conditions. If the program should generate generic code, it is important that the developer use a programming language that support an efficient and powerful mechanism for code generation.

As an example, think about how the developer could generate an HTML file in the following format, using the input data `Alice, Bob, Mallory, Eve`.

Listing 3.1: HTML example

```
<html>
  <body>
    <p>Person #1: Alice<\p>
    <p>Person #2: Bob<\p>
    <p>Person #3: Mallory<\p>
    <p>Person #4: Eve<\p>
  <\body>
<\html>
```

It would be very cumbersome to generate that file using the following Java code. Just imagine how this approach would effect scalability.

Listing 3.2: Generating HTML using Java

```
public void generatePersonHTML(String[] input) {
  writeToFile("<html>\n\t<body>");
  for (int i = 0; i < input.length; i++){
    writeToFile("\n\t\t<p>Person #" + i+1 + ": "+ input[i] + "<\\p>");
  }
  writeToFile("\n\t<\\body>\n<\\html>");
}
```

Even if you are not familiar with Java, the problem lies within `writeToFile("<html>\n\t<body>")`. This approach requires the programmer to handle indentations and line breaks manually, resulting in very unreadable code which gets exponentially more difficult to read and modify once its size increases. A better approach would be to use a language that supports a mechanism to extract indentation and line breaks from the code itself, so the developer can focus on the overall program flow.

Acceleo is a code generator from the Eclipse Foundation that conforms to the MOFM2T standard for performing model-to-text transformations (see section 3.4.1), specified by the Object Management Group (OMG). It allows developers to use a model-driven approach to building applications. [17]

Using Acceleo templates the HTML code in Listing 3.1 above could be generated with indentations and line breaks in the code itself, making it much more readable and scalable. The example in Listing 3.3 uses demonstrates this using data from an EMF model. It is important to keep in mind that while generating HTML using Java could be done differently (i.e., using DOM objects [18]), the key is to have a mechanism that support generating code in a generic format e.g. Java/Xtend class files.

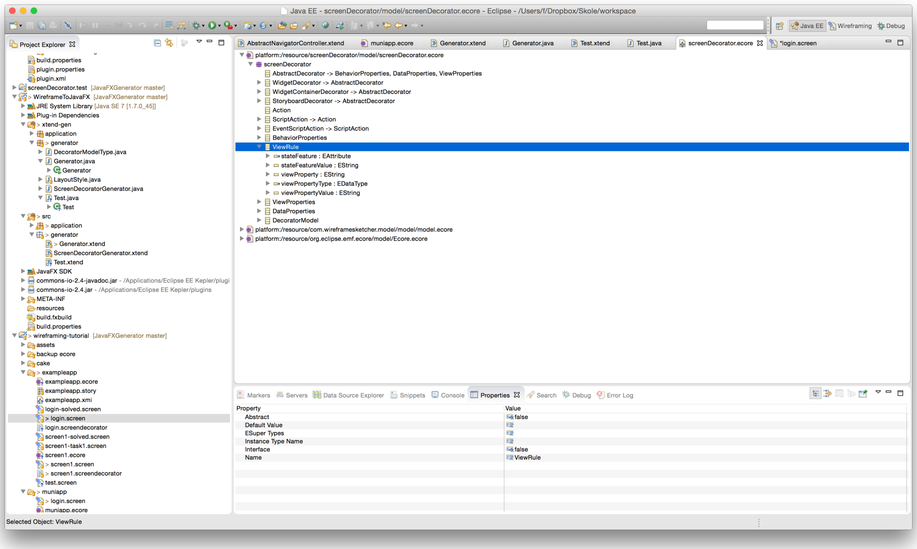
Listing 3.3: Generating HTML using Acceleo and an EMF model

```
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]
[template public generate(e : EClass)]
[file (e.name + '.html', false, 'UTF-8')]
<html>
  <body>
  [for (attribute : EAttribute | eClass.eAllAttributes)]
    <p>Person #[i + 1/]: [attribute.name/]
  [/for]
  </body>
</html>
[/file]
[/template]
```

Although it's easy for a computer to copy the same code to several files, it is still code and it should be reused if possible. Performance is always an issue and reducing the amount of code that needs to be generated improves that. When generating code for several files with minor differences, most of the underlying functionality is placed further up in the program e.g. using superclasses, in order to reduce the amount. The result is smaller files that are easier to read, maintain and debug if necessary.

3.5 Eclipse IDE

Eclipse is an open-source integrated development environment (IDE) [2]. It is written mostly in Java and comes with built-in Java development support. Eclipse uses a plug-in architecture revolved around a base workspace. Plug-ins provide all the functionality within and on top of the runtime system. This allows it to support different programming languages by means of different plug-ins. As of version 4.4 *Luna* the list is comprehensive, and include languages such as Ada, ABAP, C, C++, COBOL, Fortran, Haskell, JavaScript, Lasso, Natural, Perl, PHP, Prolog, Python, R, Ruby (including Ruby on Rails framework), Scala, Clojure, Groovy, Scheme, and Erlang. In addition to adding support for other programming languages, plug-ins can be used to work with typesetting languages e.g LaTeX [19] or other common unix programs such as Terminal. The plug-in based architecture makes Eclipse a very powerful platform. Because most of the code base in Eclipse is written in Java, Eclipse is available for Mac OS X, Windows, Linux and Solaris. [20] Figure 3.7 depicts and overview of the Eclipse GUI. Section 4.2 describes the GUI in more detail.



```
platform:/resource/screenDecorator/model/screenDecorator.ecore
  screenDecorator
    AbstractDecorator -> BehaviorProperties, DataProperties, ViewProperties
    WidgetDecorator -> AbstractDecorator
    WidgetContainerDecorator -> AbstractDecorator
    StoryboardDecorator -> AbstractDecorator
    Action
    ScriptAction -> Action
    EventScriptAction -> ScriptAction
    BehaviorProperties
    ViewRule
      stateFeature : EAttribute
      stateFeatureValue : EString
      viewProperty : EString
      viewPropertyType : EDataType
      viewPropertyValue : EString
    ViewProperties
    DataProperties
    DecoratorModel
platform:/resource/com.wireframesketcher.model/model/model.ecore
platform:/resource/org.eclipse.emf.ecore/model/Ecore.ecore
```

Property	Value
Abstract	false
Default Value	
ESuper Types	
Instance Type Name	
Interface	false
Name	ViewRule

Selected Object: ViewRule

Figure 3.7: An overview of Eclipse 4.4 *Luna*

[2] IDE - An **I**ntegrated **D**evelopment **E**nvironment is a software application that provides comprehensive facilities to computer programmers for software development.

3.6 Eclipse EMF

The Eclipse Modeling Framework (EMF) is an open source Java framework for modeling and it is one of the more known Model-driven Engineering initiatives [21]. It allows, among other things, automated generation of code from models. EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and editing of the model using a basic editor. Models can be specified using annotated Java, UML, XML documents or modeling tools, then imported into EMF.

Ecore is the *core* of EMF. It is a model used to express other models by defining their constructs. Ecore is actually its own *meta*model, as it is defined in terms of itself. EMF is based on two meta-models. The aforementioned *Ecore* and the *Genmodel*. While the Ecore consist of information regarding the defined classes, the Genmodel also provide information regarding code-generation, such as file and path information. The main purpose of the Genmodel is to control how the code should be generated.

In EMF the model describing e.g. a Person class would be referred to as a model, and what actually contains the name, age and similar attributes would be referred to as the *instance*. To store these instances EMF uses XMI (XML Metadata Interchange) by default, but it is a plug-able framework that allow other formats to be used. [22] Eclipse, and thus EMF include model-to-text transformations (see section 3.4.1) so it is easy for a program to access the model data. Many Eclipse based projects utilize EMF under the hood. One of them is the prototyping tool WireframeSketcher discussed in section 4.3.1.

Chapter 4 Criteria

4.1 Empirical data from GUI design course

In 2013 the design course IT3402 [2] had graphical designers working together with system developers with the intention of creating an GUI prototypes for fictional applications. The focus of the course was centered on the actual prototyping process, more than the end result. Each group chose their own tools for designing and for presenting their prototypes. And many chose different tools according to the different stages they were in the process. The combination of graphical designers and system developers in the course is very similar to how teams are organized in the industry. They both face many of the same obstacles during the prototyping process. In the reports provided by each group many have emphasized what tools they used, how they used them, and what made them a good (or bad) choice in the prototyping process. The reports are available digitally through the course lecturer. [2]

- *Group 10* used Adobe InDesign to make the design, and printed out tactile paper prototypes. They used images of the paper prototypes together with the application POPApp for their interactive prototypes. In retrospect they regretted the decision of using POPApp because of its linking limitations (It can only link from one design to another), and because a highlight appears if the user press on an unlinked region of the screen. Despite asking the subjects to think out loud, they sometimes forgot and just pushed on the screen. Because of the highlight, it was harder to deduce what the user was thinking and thus the value of the test diminished. A way to turn this off would have been helpful.

- *Group 9* did not test using tactile paper prototypes, instead they took images of the paper prototypes and presented them using POPApp. They found the lack of transitions/animation to be limiting and used Justinmind for the final interactive prototype. Illustrator was used to make the design.

- *Group 8* used Axure RP both for the design and the interactive prototype. Only fonts available on the test device would work, so standard fonts had to be used despite their vision.

- *Group 7* created their interactive prototype using FluidUI. They reported that slow load times and responsiveness was an issue and it worked against the illusion of a real program.

- *Group 5* drew their designs in Photoshop, and used Justinmind to create and present an interactive prototype. They felt that bad responsiveness and long load times worked against the illusion of a real program.

- *Group 4* made the design using Adobe Photoshop and created interactive prototypes with InVision. Each screen had to be made separately in Photoshop since InVision only present finished pictures. They reported that it was cumbersome to duplicate the designs for every minor change.

- *Group 3* used Adobe Illustrator for drawing, based on the members experience. They used Axure RP Pro (trial) for the interactive prototype. The designs were first made with Illustrator, then each component was isolated and exported into Axure RP where they were reassembled and the interactions programmed. The finished prototype was exported as HTML, which made it possible to test on virtually any device including smartphones. They reported that the process was very technically demanding, but the result was solid.

4.1.1 Analysis

What was in common for most groups was that the paper prototypes were as simple as possible and easy to refine, while the interactive prototype was a little more complex both in design and functionality. Based on the reports the total set of features that the groups reported lacking were choosing fonts, choosing to turn off highlighting indicators for links, animated transitions, a way of performing minor changes to a design without duplicating it and fast and responsive load times. Although the set does not include all the features that the groups enjoyed as well, it can be joined with the feature set of each tool in order to find what features are important to support in such a tool. It is important to point out that depending on what stage in the prototyping process the user is, it might not be desirable to support features such as animations and transitions because it can be beyond the purpose of the prototype. The requirements are different for tools used to create fast and simple prototypes and prototypes that refine and finish the GUI design with custom buttons, images and colors. The information from these reports can be used as a basis for how a prototype that allow the user to create functional prototypes should be developed.

4.2 Eclipse Workbench

As described in section 3.5 Eclipse is an integrated development environment (IDE) written in Java. It is based on a plug-in architecture. All the functionality in Eclipse is due to different plug-ins. The Eclipse GUI represents information using *Workbench Windows*. Each window contains one or more *perspectives*. Perspectives contain *views* and *editors*, and control the content of certain menus and tool bars. It is possible to have more than one window open at the same time, but it is essentially another running instance of Eclipse. The hierarchy is:

- Workbench
  - Window
    - Perspective
      - Editor
      - View

See Figure 4.1 for an overview with annotations.

4.2.1 Perspectives

In the workbench window, a perspective defines which set of views should be visible, and their layout. The job of the perspective is to provide functionality in order to accomplish a certain task or in order to work with a certain type of resource. One of two examples is the Java perspective that combine views which are common to use when editing Java source files, and the Java Debug perspective which use a different set of views and layout for improving debugging of a Java application. It depends on the task at hand which perspective is most practical to use. It is possible to develop additional perspectives that can be used via the *Window → Open perspective* menu-item. [23] [24]

4.2.2 Editors

Most perspectives in the workbench window consist of an *editor*. Each filetype can be associated with an editor so when the user opens e.g. a Java source file an appropriate Java editor is used. Or when an XML file is opened, an XML editor is used. The user can manually specify which editor to use when opening files, however, not all editors are compatible with every filetype. This can be helpful when there are several ways to view the data e.g. HTML code and the interpreted page. It is possible to have multiple editors open at once, but only one can be active at a given time. The main menu and toolbar follow the active editor and contain operations which can only be performed upon it. Visually, the editors are presented using tabs, just like the ones commonly used in todays web browsers. An asterisk beside its filename indicate that there are unsaved changes to the file. Eclipse perspectives are highly customizable and it is possible to view more than one editor at a time by dragging the tabs to various positions inside the perspective. The containing editor space will split into the current and the new editor. Although, only one editor can be selected and *active* at the same time. Each perspective share the same set of editors within the window. It is possible to develop additional editors that work with specific file extensions. This allows existing or new perspectives to handle different file types.

4.2.3 Views

In addition to an editor, a perspective usually contain one or more *views.* Views can provide alternative ways of displaying data and are used together with editors. An example of a view is the console output of a typical program, or the Project Explorer which display all the projects and files the user is working with. It is possible for a view to have its own menu and toolbar. By closing or adding views to a perspective, the user can change the layout and functionality. The view can register which elements are selected in the editor, so it can update its content depending on the specific element. It is possible to develop new views that represent/modify the data from the editor differently than existing ones. Technically it's quite simple, and it can be accessed via the *Windows* $\rightarrow$ *Show view* menu-item. Both views and editors can be added to existing perspectives extending the functionality.

4.2.4 Adding additional behavior

As described in section 3.5, Eclipse uses a plug-in architecture centered around a base workspace. All the functionality of Eclipse within and on top of the runtime system is provided using plug-ins. In addition to custom *Perspectives*, *Editors* and *Views* Eclipse support additional behavior through plug-ins that makes it possible to select a file in the Project Explorer and make it *perform* something e.g. a Java class file with an included main method can be *Run as Java Application.* This will compile and run the Java code. If the class file does not include a main method that plug-in will issue an error stating *"Selection does not contain a main type"* and return. It is possible for developers to implement their own plug-ins that behave differently. For instance a plugin that allows the user to right click on an image file and press *Convert to PNG.*

4.3 Tools

There are several tools for prototyping, but the premiss for this thesis is to see how an *Eclipse based tool* can be integrated in order to improve the efficiency of the

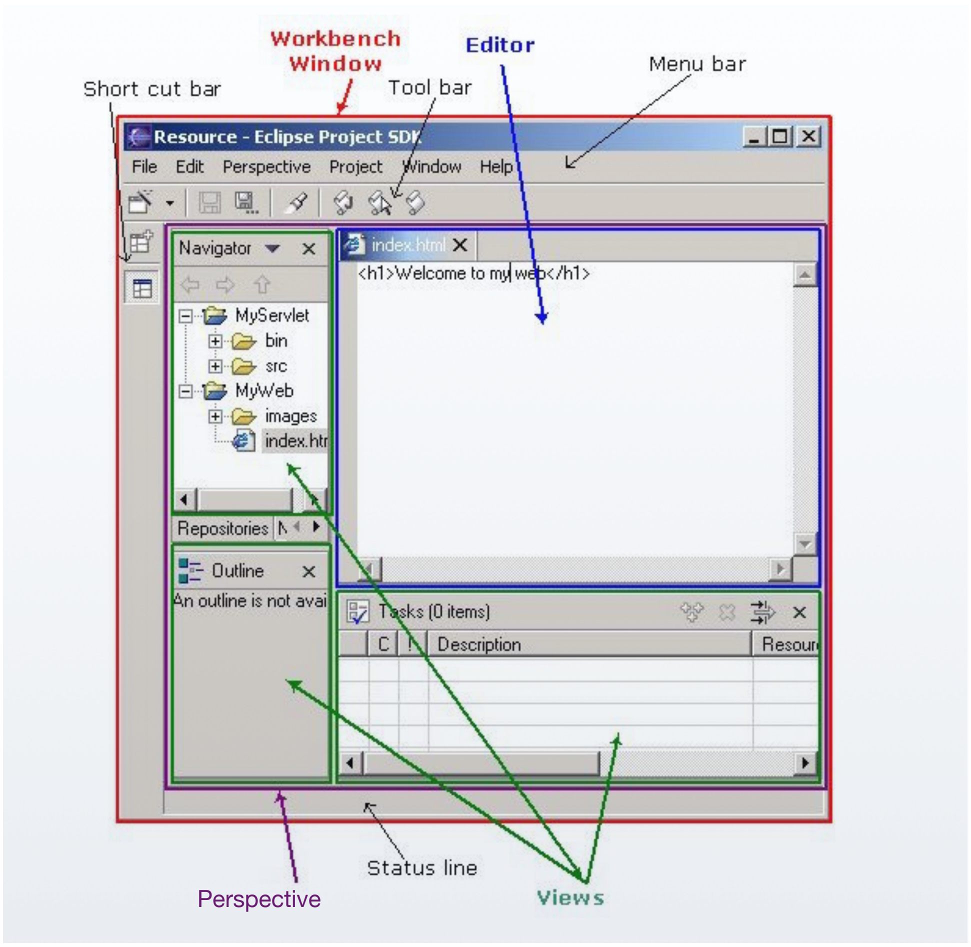


Figure 4.1: An overview of the Eclipse Workbench - Window, perspective, editor and view.

prototyping process. Thus, only prototyping tools based on Eclipse were considered. As part of the initial project description WireframeSketcher was mentioned as a prototyping tool candidate. It is available in an educational version, which made it affordable enough to be considered for use. It is Eclipse based and as such met the criteria. It was chosen as the tool to work with for this thesis.

4.3.1 WireframeSketcher

WireframeSketcher is a prototyping tool based on Eclipse that utilize EMF (see section 3.6) to represent its designs (sketches), and comes both as a plugin for

Eclipse and a standalone version. It is a *WYSIWYG* editor that support dragging and dropping *widgets* (e.g. buttons and labels) and allow for some widget types to link to other sketch files. Sketch files are known as *screens* and have the file extension `.screen`. In addition to screen files, it is also possible to add a storyboard where the user can get a visual representation of the linked screen files. The storyboard file extension is `.story`.

To reduce confusion the prototypes created in WireframeSketcher are hereinafter referred to as ***screens*** *or* ***sketches****, while* ***program*** *refer to the thesis prototype programs.* ***Functional prototype*** *refer to the end-product produced by the program.*

When the sketch is complete, the user can enter a presentation mode where clicking the linked widgets will navigate among the screens accordingly. It is not possible to have conditional jumps, conditional layout or any form of variables. The main purpose of WireframeSketcher is to create design mockups as quickly and easily as possible. Due to the fact that WireframeSketcher only describes a visual layout and nothing related to dynamics (except simple navigation), it's important that the program is optimized for that purpose. WireframeSkethcer comes with a *hand-written* and a *clean* theme.

Since WireframeSketcher is based on Eclipse, its GUI also conforms to the Eclipse Workspace layout depicted in Figure 4.1. The WireframeSketcher perspective include an active editor and several views. Screen files are edited in the *Screen Editor* and Story files are edited in the *Storyboard Editor*. Because the files are stored using XML they can alternatively be edited manually using the *XML Editor*.

As seen in Figure 4.2, in addition to the editor there are several views in the perspective. The position of the editor and views can easily be changed, but the default positions in the perspective are as follows: The *Project Explorer* used in most perspectives (e.g. Java) is located in the upper left corner and has another tabbed view below it for *Properties* and *Links*. The *properties* view depicted in Figure 4.3 shows options such as font face, font size and color for a text label; and solid, dotted and dashed lines for a horizontal divider. Notice that the properties pertain to the selected button widget *Week*. If the selected element supports linking, the link tab has a dialog for linking to another screen. It's important to emphasize that while widgets make up the content of the screen, it is possible to change the properties of the screen itself (e.g. Font Family and Sketched or Clean theme). Thus widgets are not the only elements that can be selected in the editor. Depending on what element is selected, the views show the appropriate information for that element. On the right hand side is the *Palette* view. It contains a library of widgets that can be dragged into the editor. It is possible to show all, or view them under different categories. There is also an *Outline* view not depicted in Figure 4.2. It simply lists

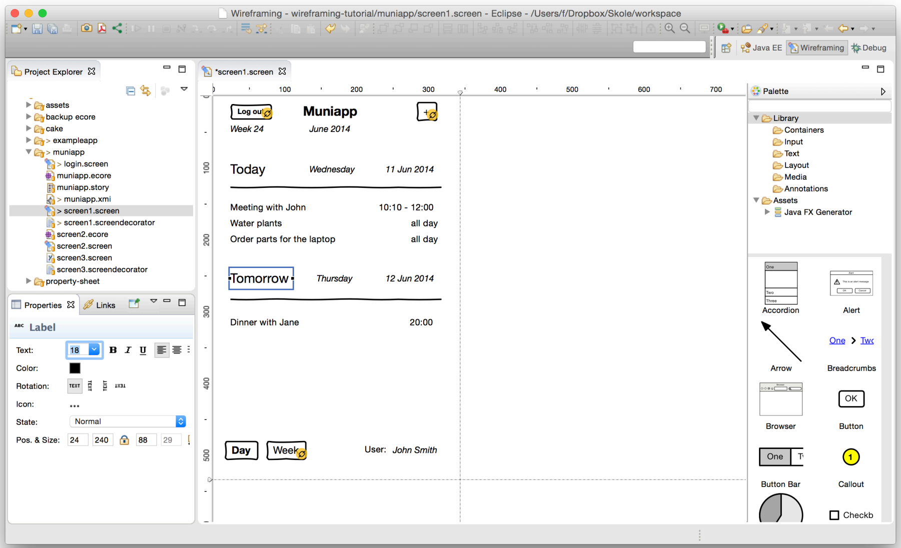


Figure 4.2: An overview of the WireframeSketcher GUI. *Project Explorer* in the upper left corner. Below that is the *Properties* and *Links* view. *Screen Editor* in the middle, and *Palette* on the right.

all widgets in the editor.

As an Eclipse developer it is possible to create your own *views* that supplement the existing ones. Section 4.2.3 outlines how this can be done. The goal is to develop an extension that follows the architecture that Eclipse and WireframeSketcher is built on. It has proven to work quite well in practice.

WireframeSketcher model

The WireframeSketcher EMF model contain all the data which is needed to represent screens in WireframeSketcher. It is beyond the scope of this description to go into too much detail, but the model includes classes for the different widget types and various *Support features* i.e a button has *Link support* and thus can link to another screen, and it also have *Font support* so it should be able to change font through the *Properties* view.

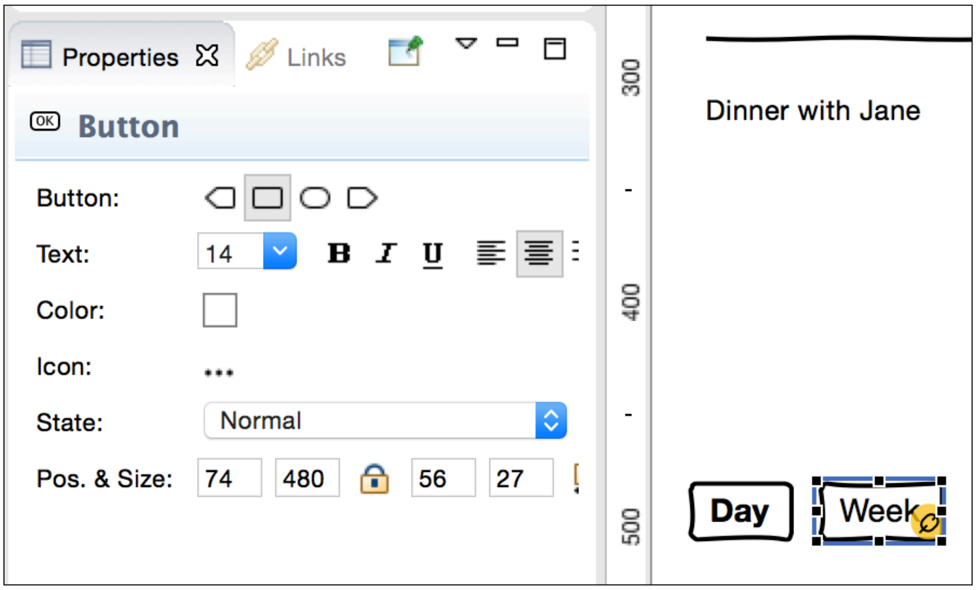


Figure 4.3: The *Properties View* inside the WireframeSketcher perspective. Notice that the properties pertain to the selected button widget *Week*.

4.3.2 JavaFX and FXML

In order to generate a functional prototype based on a screen, the prototype layout has to be represented using a format which allows generated code. In other words the functional prototype must be able to load its graphical markup from files which can be generated. This makes it possible for a program to parse the screen files and generate markup files which can be used by an User Interface library. Because Eclipse and EMF is Java based it makes sense to use Java or something that is compatible with Java.

JavaFX (as of Java 8 named JavaFX8 following the same numbering) is replacing Swing[1] as the Java client UI library. It is designed to provide a lightweight, hardware-accelerated Java UI platform for applications. As of JavaFX 2.2 the libraries are installed as part of Java SE, which increases availability. The GUI markup for JavaFX is stored in *FXML* files, which can be coded manually or generated using the Scene Builder editor. It also supports CSS for additional layout modifications like custom buttons, colors and event behavior. [25] JavaFX and FXML is the recommended approach by Oracle, which makes it more future proof. It is state-of-the-art and a logical choice for a UI library since it fits the criteria well.

[1] Swing was until JavaFX the primary Java GUI widget toolkit

4.4 Creating requirements

4.4.1 Target group

Before any functional requirements can be created, some decisions has to be made regarding functionality. A deciding factor for what functionality should be added is the target group. If the tool is made for very programming savvy developers it's reasonable to assume that it's functional to use full fledged scripting rather than something less expressive, yet simpler. The opposite is true if the target group are users with minimal programming experience. This decision affects how the program should be made. When designing a GUI designers are involved and it makes sense to use designers as the target group. Developers who are not involved in the GUI prototyping process will not work with such tools anyway. In order to allow for dynamics that are fairly easy to implement and use, the target group should be designers with some programming experience.

4.4.2 Deciding on functionality

All prototyping tools researched in this thesis (see section 3.3) support simple navigation. Some tools support more advanced functionality like variables and actions. Other tools might even support full fledged scripting. It is important to choose functionality that gives a lot back to the user of the tool without being too time consuming to implement. I.e. with little developer effort create something that result in substantially better user functionality. This can be explained using the analogy of a staircase where the effort of developing a features is represented by the depth of the step and user functionality is represented by the hight of the step. Figure 4.4 depicts the ideal scenario where it takes a small developer effort to implement a lot of user functionality. The opposite scenario is depicted in Figure 4.5 where a lot of developer effort result functionality that is not very useful for the user. Choosing functionality that fits these criteria is crucial in a prototype where time is limited. The implemented functionality should be useful for the user of the tool.

The target group for this program is slightly technical designers, rather than very technical developers. It is therefore important to add functionality that is just complex enough so that the designer can complete the task of building a functional prototype, without the program being too complex and hard to use. This is a very fine line, and it is difficult to choose what *just enough functionality* is. When researching other prototyping tools and comparing them to the experiences of the students in the course IT3402 2013 [2] the majority changed their tool because of a lack of functionality (see section 4.1). Using data from these reports it looks like the following functionality would be useful:

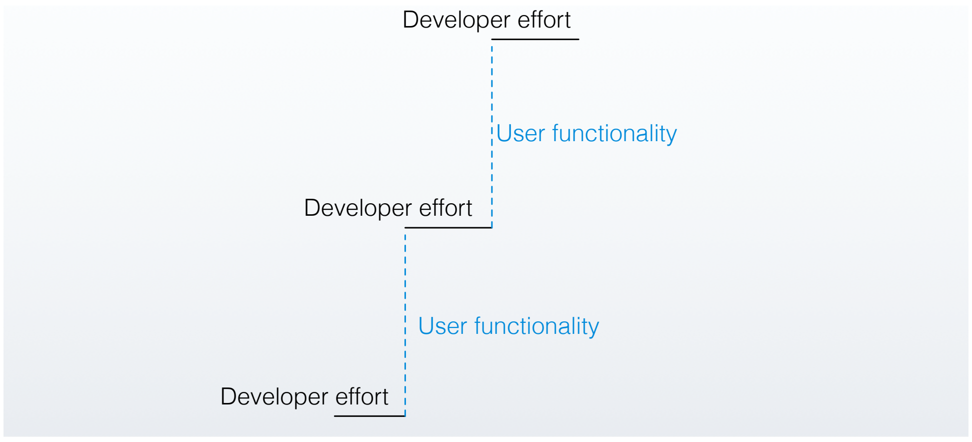


Figure 4.4: The ideal scenario where just a little developer effort result in a lot more user functionality.

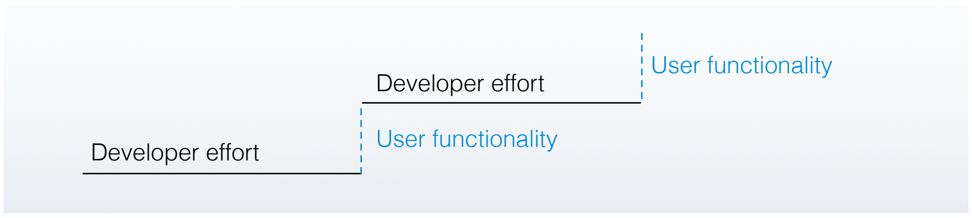


Figure 4.5: A non-ideal scenario where a lot of developer effort results in just a little more user functionality.

- Variables - *string*, *int*, *boolean.* e.g. *Int* can be used for counting login attempts.
- Action methods - e.g. set *username* to textfield value on button click
- Conditional jumps - e.g. jump only if *isLoggedIn* equals true
- Conditional layout - e.g. show text and color depending on variable value

Referring to Figure 4.4 it is the *authors* opinion that starting with a relatively small set of the most crucial functionality is the better approach. Also, although animations and transitions were wanted by most of the student groups, it will not be implemented because the effects are testable using other software, and it takes too much time to develop. Referring to Figure 3.5 the tool should be shifted to the right of the existing tools since it should cover more functionality.

After user testing this functionality could be altered to better accommodate the users. Referring to the MVC methodology in section 3.3.3 it makes sense to follow the same pattern for the functional prototypes produced by the program. The data variables can be represented by means of a data model, the visual information can be represented in a view, and the behavior can be placed in a controller.

In order to decide which functionality should be included, it is helpful to look at the typical requirements of a functional computer prototype e.g. an application for storing a secret message in a private user environment. It supports login and a way to store the message. It also includes actions that only allow the login to proceed if the username and password combination is correct. The program contains an error message that is only visible when the user fails to log in. A typical implementation could include the login credentials and message stored as string variables, along with a boolean for the state of authentication (e.g. isLoggedIn). Assuming the program follows the MVC pattern, the *model* would include *String* and *Boolean* variables. Visual markup code would be placed in the *view*, and the actions that change the variables and any layout conditionals would be integrated into the *controller*.

Using all this information the functional requirements for the program can be described.

4.5 The Functional Requirements

Since WireframeSketcher store its projects in *.screen* and *.story* files, the program should be able to load and parse them. Inside each screen file reside widgets. A JavaFX node is the runtime representation of the WireframeSketcher widget. The generator should generate JavaFX nodes for each widget that represent the same type e.g. Button or TextLabel and store them in an FXML file. So if the screen includes a button widget, the FXML should include a JavaFX button. When the program loads the FXML file it should display (close to) the same visual information as the original design.

In order to transform a static WireframeSketcher design into a functional prototype the program needs a mechanism for controlling certain behavior. As already discussed with MVC a *data model* is used to store data. The screen in WireframeSketcher represents the *view*, so the last component needed is a *controller*. The controller can control behavior. How the user should interact with the controller is part of the thesis to explore, but the controller should allow for navigation, statements and layout conditionals. The rationale behind it is that it must be possible to *navigate* from one screen to another. In order to support data variables it must be possible to change their values using *statements* e.g. `variableName = my string`. A mechanism for *conditional layout* e.g. `if isLoggedIn == true; show button` is required to change the layout depending on the variable values. Exactly how this is implemented is not important, only that the functionality is covered.

As discussed in section 4.4.2 above, the data types should include `String`, `Bool` and `Int` in order to allow for dynamic text, boolean conditionals and counters. Using semantics from general programming languages, variables should be able to exist in different scopes. It should be possible to have global data variables, variables per screen and variables per widget. They should follow a scope hierarchy where a variable declared in a more specific layer in the scope overrides a variable in a less specific layer. e.g. a screen layer variable named `hasSaved` should override a global layer variable with the same name. Similarly a widget layer variable with the same name should override the screen layer variable. Since the data model is an EMF model, the variable values are stored in an XMI instance of the model. In order to store and change the instance variables, the user should be able to add actions that set the value of a variable when triggered. e.g. when a button is pressed, the action should be able to set a variable to the value of a specified text field or similar text input widget. To reduce duplicates of screens where the only difference is a minor text or visibility change, support for conditional layout, or *styles*, should be added.

Styles should allow the user to modify the JavaFX-node representation of the widget, depending on variable values. e.g. the user should be able to change the visibility of a *Label* depending on the value of `isLoggedIn`. Since the program already must handle and parse EMF files (*screen*, *story*), the three dimensions *Data*, *Actions* and *Styles* should also be stored and manipulated using EMF. In order to make the program more convenient for the user, it should be possible to build the functional prototype by right clicking on a Storyboard file and pressing *Generate Functional Prototype* (see section 4.2.4). This makes the footprint of the program very small and easy to use.

4.6 Non-functional requirements

Similar to how WireframeSketcher is integrated with Eclipse, the program should integrate with the platform in such a way that the user can install it as a plugin. It should be possible to run the program easily by right clicking on a sketch file within Eclipse.

4.7 Exploring the dimensions

The three dimensions *Data*, *Actions* and *Styles* discussed in section 4.5 represent a possible way of constructing the program. It's important to emphasize that although there are multiple ways these can implemented, the purpose of this thesis is not to find the best implementation. The purpose is to look at how the dimensions function and how they can be used together in a design to realize a *screen.* One way to approximate that is to create and test several sketch examples and see how the dimensions work together.

Chapter 5

Initial development

Most of the initial work was researching existing tools and reading UI design reports, discussed in section 4.1, in order to figure out what was state-of-the-art and which features users were likely to want. This built a basis for which a design could be developed. It was decided that dividing the challenges into different subtasks would be advantageous. It would be easier to focus on a smaller problem at a time, and each subtask could stand as proof of concept without being part of a complex implementation. With this rationale the first thing that was explored was a simple *property list* depicted in Figure 5.1. The goal was to design a program that could interpret the sketch file and output a JavaFX representation of it.

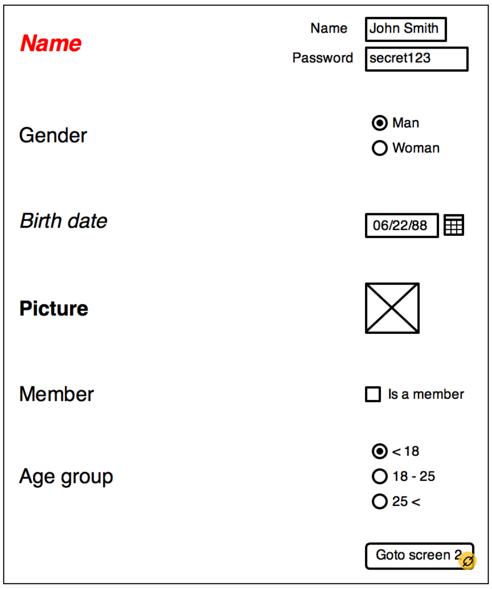


Figure 5.1: Starting to explore the design interpreting a *property list*.

5.1 Focusing on different aspects

There was several good reasons to start with a property list. The program had to interpret different widgets and output corresponding JavaFX types. But first the developer had to learn and map the different types. Using a property list made it easy to manually inspect the screen file XML to see each widget type and their attributes. In addition to exploring possible program design structures it acted as a proof of concept as to how widget attributes could be copied to the output. Font family, font size, font face, color, rotation, images and other attributes that are an important part of a sketch could be copied to an FXML file and presented using JavaFX.

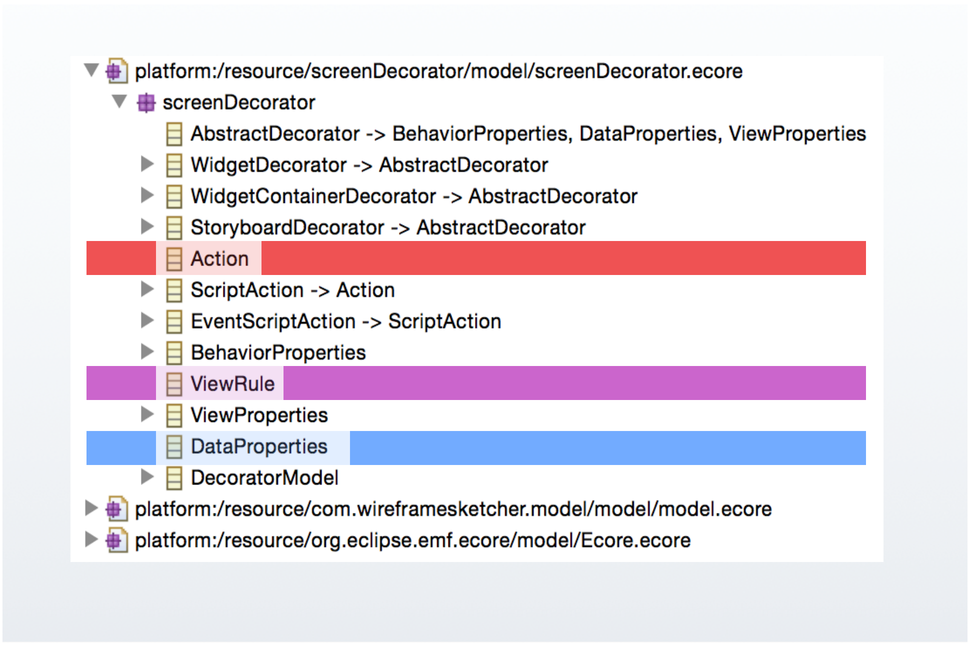
It was discussed early on that the program would be a proof of concept and should focus on different *aspects*. A successful representation of the sketched property list was one aspect which proved that it was possible to transform a sketch into a JavaFX program including the properties of each attribute. Other aspects was discussed during the iteration. This included *layout interpretation* where the program could interpret what layout to use based on the widget positions in the sketch. This could be useful to enhance the resizing capabilities of the functional prototype. Another aspect was *Model data* where authentic data could be automatically be used in place of e.g. list. So if the sketch included a list or something that resembles a list the program could use an actual list and populate it with generic data from the model. This would make the result look more authentic than using e.g. *John Smith* type of names or *Content here* for text fields. It also functions much in the same as the *Lorem Ipsum* text fill used in web design, due to the fact that users are distracted by repeating content. Using model data would speed up the process and remove the designer from the task of creating dummy data. The layout for the sketched prototype is obviously *sketched*, but the functional prototype which uses JavaFX utilize normally styled buttons and labels which are not sketched. Many prototyping tools support stencils for different layouts to emulate a specific platform device. e.g. an iPhone or Android phone layout. An interesting aspect to explore is how the program can apply various layouts either depending on the layout set in the sketch, or in another more generic manner. If a team chooses to use a prototyping tool because it supports layouts compatible with the platform they are sketching for, implementing such a feature might make the program a lot more attractive the end users. The last aspect is using a model to *decorate* the existing model in order to add new functionality. It was decided that this was is the most interesting aspect to focus on.

5.2 Decorating sketches with a decorator model

In model based programming it might be desirable to extend a model, but it is not always possible to modify the model itself, e.g. due to wrong permissions or closed source. One way to achieve the same effect is to create a *decorator model*. The decorator model is a standalone model whose purpose is to *decorate* another model e.g. adding new education attributes to a Person model, or adding completely new classes. In the case of this project a decorator model was used to add functionality to a sketch.

5.3 Screen decorator model

Using the decorator model technique, the program creates a decorator model instance for each screen. The models are created using the *screen decorator EMF model* depicted in Figure 5.2. It is modeled after the criteria discussed in section 4.5 and it contains several classes for adding functionality, among them *Action*, *ViewRule* and *DataProperties*. These are central components relating to the program. They are consistently color coded to make it easier to differentiate between them. There is no special reasoning behind the color choices other than that the colors should be easy to differentiate and hopefully retain some aesthetic properties. That results in the *Action* class in **red**, the *ViewRule* class in **violet** and *DataProperties* class in **blue**.



```
▼ platform:/resource/screenDecorator/model/screenDecorator.ecore
  ▼ screenDecorator
      AbstractDecorator -> BehaviorProperties, DataProperties, ViewProperties
    ▶ WidgetDecorator -> AbstractDecorator
    ▶ WidgetContainerDecorator -> AbstractDecorator
    ▶ StoryboardDecorator -> AbstractDecorator
      Action
    ▶ ScriptAction -> Action
    ▶ EventScriptAction -> ScriptAction
    ▶ BehaviorProperties
      ViewRule
    ▶ ViewProperties
      DataProperties
    ▶ DecoratorModel
▶ platform:/resource/com.wireframesketcher.model/model/model.ecore
▶ platform:/resource/org.eclipse.emf.ecore/model/Ecore.ecore
```

Figure 5.2: The screen decorator model used to create screen decorator models. Action in red and ViewRule in violet.

These colored classes corresponds to the dimensions discussed in section 4.5 and make up the three decorating elements:

- Data (corresponds to *DataProperties* in the model
- Actions (corresponds to *ScriptAction* in the model)
- Styles (corresponds to *ViewRules* in the model)

In order to create and manipulate the screen decorators there has to be an input mechanism in the GUI. Chapter 6 discusses the various designs for the input mechanism.

Chapter 6 Design

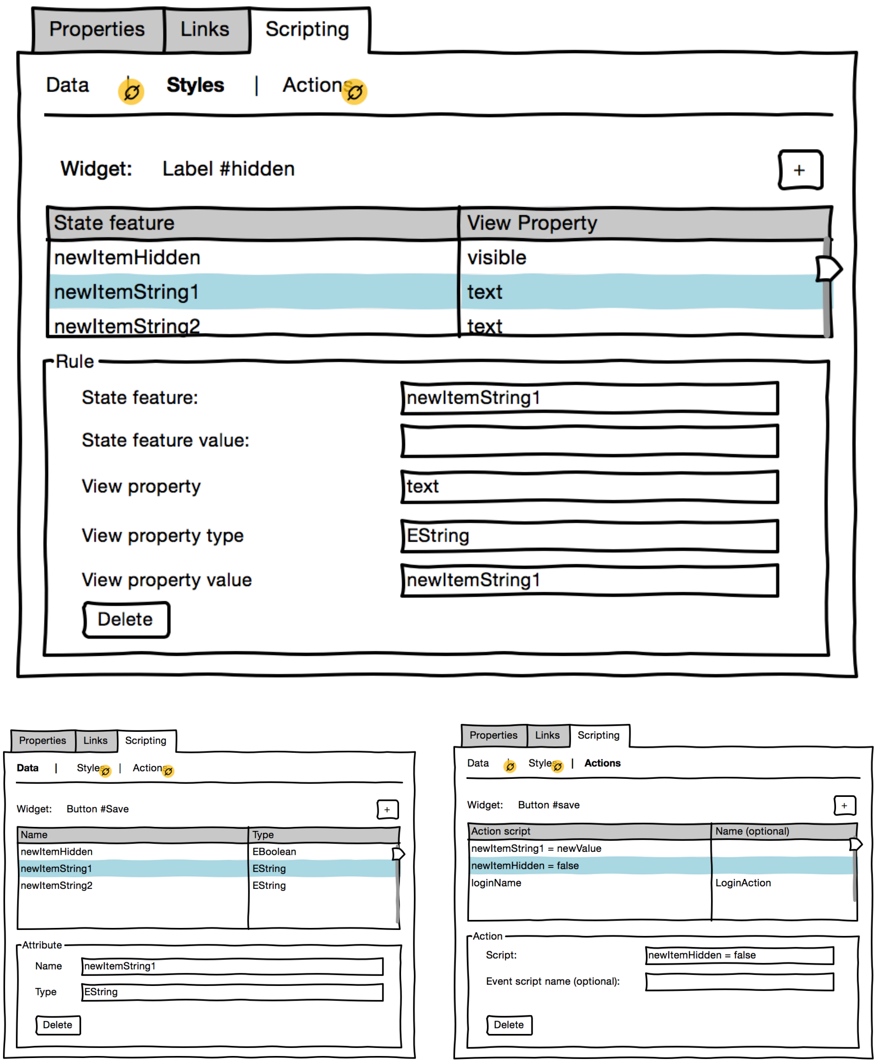
6.1 1st Iteration

As discussed in section 4.2.3 it is possible to construct new *views* that allow the user to manipulate underlying data. So the initial idea followed the Eclipse philosophy of using additional *views* to manipulate the screen decorators.

6.1.1 Editing screen decorators in views

Several ideas was discussed as to what information should be available and how the layout of the view should be. It made sense to design the view to conform to the `screenDecorator.ecore` model so that the user could modify the underlying data. Based on which screen file is active in the editor, the view should present the corresponding screen decorator model. Only essential attributes should be presented. Although the placement of the view wouldn't be fixed, it was initially designed to complement the *Properties* and *Links* view. A tabbed approach was used to save real-estate space. One tab for each of the decorating elements *Data*, *Styles* and *Actions*.

Figure 6.1 depicts a sketch of a possible design of the view.



Widget: Label #hidden

State feature	View Property
newItemHidden	visible
newItemString1	text
newItemString2	text

Rule

State feature:	newItemString1
State feature value:	
View property	text
View property type	EString
View property value	newItemString1

Widget: Button #Save

Name	Type
newItemHidden	EBoolean
newItemString1	EString
newItemString2	EString

Attribute

Name	newItemString1
Type	EString

Widget: Button #save

Action script	Name (optional)
newItemString1 = newValue	
newItemHidden = false	
loginName	LoginAction

Action

Script:	newItemHidden = false
Event script name (optional):	

Figure 6.1: A sketch of the tabbed view that could be used to manipulate the screen decorators. Notice the *Properties* and *Links* tabs that already exist in WireframeSketcher.

One of the downsides of using such an approach is that the user has to be more technically competent, and the GUI is also rather complex. It could be modified to assume `EString` for strings and `EInt` for numbers etc, but it is still not clear and it is difficult to get an overview of all the information at the same time. Implementing such a view is not very difficult, but deciding which attributes should be accessible from the view, and their placement inside the view itself is time consuming. In the second iteration this a new approach was implemented.

6.2 2nd Iteration

6.2.1 Representing screen decorators using custom widgets

In the second iteration the technical backend of the prototype was beginning to take shape. It was technically possible to create and manipulate screen decorator models, but no input mechanism was implemented. After a discussion with the supervisor, it was decided not to utilize the *view* approach imagined during in the first iteration. It was more time consuming to implement and a more unorthodox idea was put forth. Instead of a *view* the widgets themselves would serve as an input mechanism. After some research regarding the underlying WireframeSketcher model, it turned out that creating custom widgets was fast and easy. These widgets could then serve a special purpose when parsed by the generator. In order to connect the decorators to other widgets the idea to use the existing annotation *Arrow* widget was realized.

Three custom note widgets was used to represent the decorators. When parsed by the generator they have a special meaning. Since the only potentially ambiguous color is the yellow used by WireframeSketcher for a normal *Note*, the same color convention as presented in section 5.3 was used. *Actions* are represented by a **red note**, *Data* is represented by a **blue note**, and *Styles* are represented by a **violet note**.

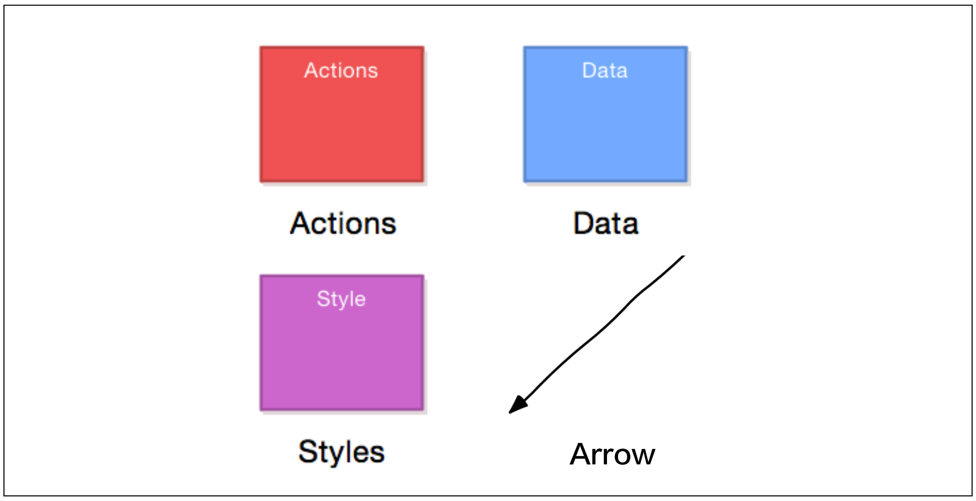


Figure 6.2: The screen decorator types in their respective color suit, with the widget-connecting arrow.

The scripting format

When the intention was to implement the input mechanism using a *view*, the relevant attributes had separate fields for its data as seen in Figure 6.1. When that solution was discarded and special notes was chosen, there had to be another way of manipulating the data in the model. A scripting format was chosen to do just that. The scripting format was based on several real programming and scripting languages. In order to facilitate a less technical approach, the format loaned features from natural languages like AppleScript and HyperTalk [26]. The goal was to make it easier for designers to understand the format, rather than developers who already know the *defacto* standards of using e.g. two equal signs to compare values.

During the project this format was never revised. It was decided that there was no basis to do so, and that it might be better to implement an existing language like JavaScript. In order to evaluate the effectiveness of the scripting format user-testing was required, so the scripting format will be evaluated in section 9.1.5.

Data dictates which data variables are to be declared e.g.

Listing 6.1: "Data"

```
String username
String password
Bool isLoggedIn
```

```
Int loginAttempts
```

The supported data types are *String*, *Bool* and *Int*. The way variables are declared is semantically very similar to *C* like languages which include Java. The type can be written in every capitalization e.g. `sTrInG` is allowed, however, the variable names are case sensitive. Meaning that `isLoggedIn` and `isloggedin` are two separate variables.

Actions dictates which variables should be set to which value when executed. A set of actions are connected to a specific widget using the *arrow*. They are similar to *statements* in most programming languages.

Listing 6.2: "Actions"

```
isLoggedIn = true
username = ${nameField}
password = ${passField}
```

The actions is used to set a *variable* to a *value*. In listing 6.2 the bool variable `isLoggedIn` is set to `true`. The special format `${`*`identifier`*`}` is used to refer to *input fields* like the ones used to enter username and password. The text field itself set the identifier by containing the text `=${`*`identifier`*`}` in the sketch. In the case of listing 6.2 it would be `=${nameField}`. *Identifiers* can be given to text areas as well. The format `${`*`variable`*`}` is also used for outputting variable values e.g. as the text for a label.

Styles dictates which properties should be set depending on the state of a variable. They are similar to *if statements* in most programming languages. Styles can be connected to a specific widget via an arrow, or it can affect the whole screen if not connect at all.

Listing 6.3: "Styles"

```
if isLoggedIn is true
set visible to true
if isLoggedIn is false
set visible to false
```

The format of *styles* is very influenced by natural languages. It includes a condition and a statement. `if isLoggedIn is true` - *If* the value of `isLoggedIn` is `true`, `set visible to true` - *Set* the property named *visible* to `false`. Listing 6.3 includes a style for both the true and false condition. *visible* is a property of the runtime application's representation of the widget - the node e.g. a button in the functional prototype. In other words in case of a button, if the *property* is `text` and the code is `set text to Hello World` then the text property of the button will be set to *Hello World* In Xtend this would equal the code `button.text = "Hello World"`. This mechanism allows the *property* to be any method available to the node. In addition to *text* and *visible* some useful properties are *disable* (hide the button) and*style* (set CSS style).

Using the scripting in listing 6.1, 6.2 and 6.3 above the decorating notes would look like Figure 6.3. The keyboard *app* inside the *Data* decorator indicates that the variables should be global for the whole application, and not only visible for the screen file it is defined in.

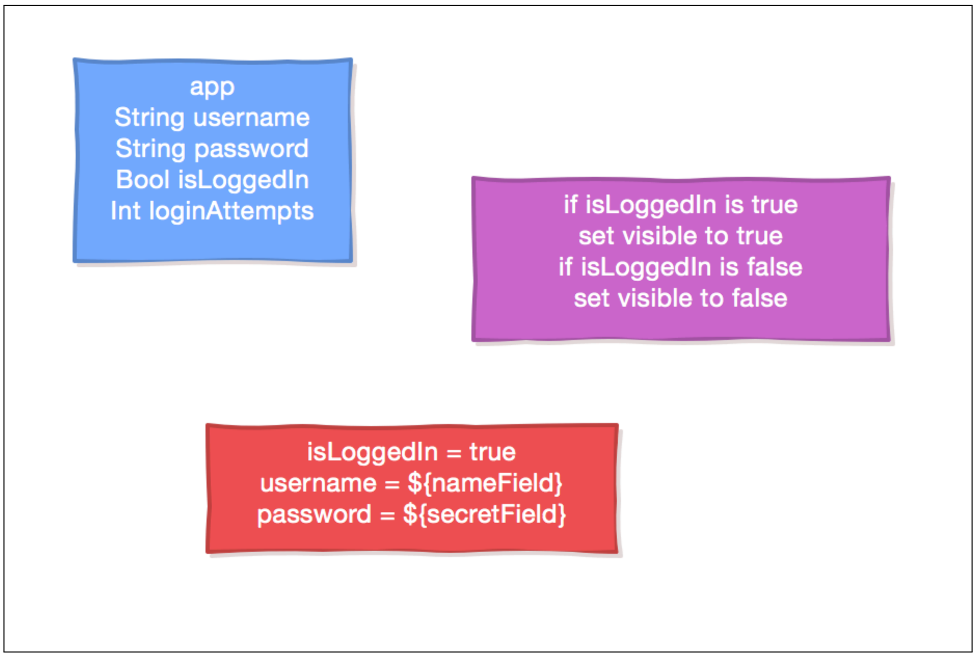


Figure 6.3: The screen decorator types with data from listing 6.1, 6.2 and 6.3

The decorators adds functionality when applied to a sketch and could be used as the example in Figure 6.4.

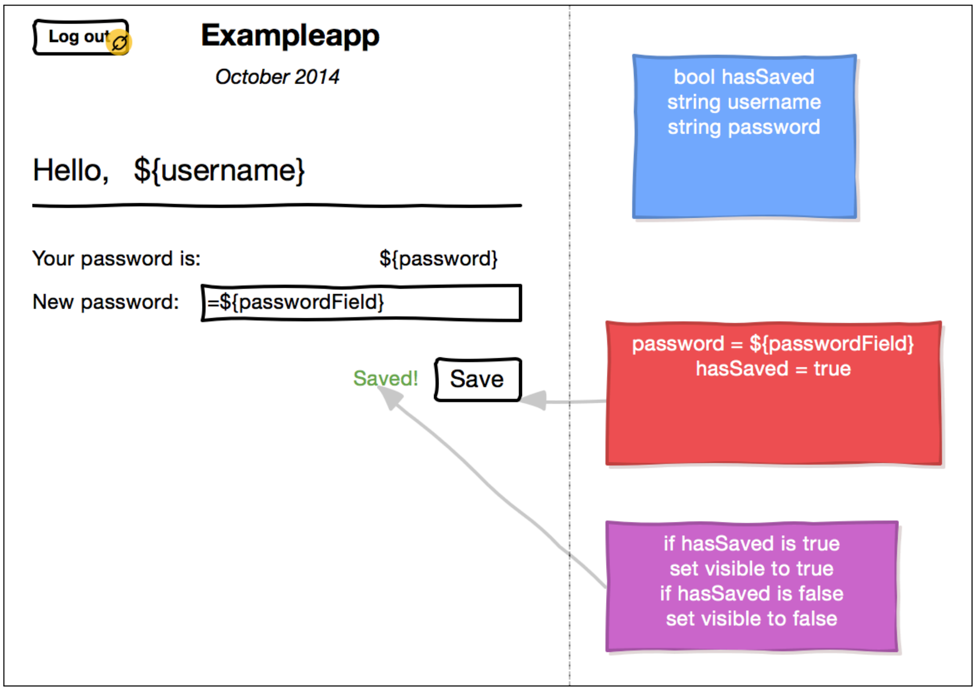


Figure 6.4: A sketch decorated with functionality.

Chapter 7
Technique

7.1 How development was done

The two programs developed for this thesis was written in the Xtend Programming Language using Eclipse EE *Luna*. It uses the WireframeSketcher EMF model (see section 4.3.1) in order to generate a JavaFX (see section 4.3.2) application that uses the wireframe designs for its layout. The result is a dynamic JavaFX representation of one or more static wireframe designs.

Tools

The following tools were used during development:

- Eclipse EE *Luna* with JavaFX and Xtend - as IDE.
- WireframeSketcher for designing mockups stored using EMF, and for designing new features for the program.
- SceneBuilder for creating early FXML drafts

7.1.1 SceneBuilder

SceneBuilder is a tool developed by Oracle which is designed to be a visual layout tool for quickly designing JavaFX applications. Is is a WYSIWYG editor where the user can drag and drop UI components to the work area, modify their properties and apply CSS. SceneBuilder generates FXML code in the background. The tool was of great help when learning how FXML works, and which properties combinations were available for each component. It was used as an educational and debugging tool during the development process. [27]

7.1.2 Xtend programming language

The official website [28] defines Xtend to be *"...a flexible and expressive dialect of Java, which compiles into readable Java 5 compatible source code."* It was chosen for the development because it is well integrated with Eclipse and EMF, and supports templates which makes generic code generation a lot more efficient and readable. It also reduces code by supporting better defaults, type inference, lambda expressions and extension methods.

Templates

Xtend supports *Templates* which makes it possible to generate code with whitespaces, newlines and indentation kept intact. This is one of the major reasons why Xtend was chosen as the programming language, along with the fact that Xtend is compiled to readable Java code, and thusly works together with Java in the same project. Xtend also support inline variable interpolation for fast and readable code using guillemets « » (angle quotes). These are seldom used in programming languages, which makes them a good choice. Continuing the HTML example from section 3.4.2 below is an Xtend implementation of the HTML code in Listing 3.1. It is a method named `generatePersonHTML` that takes one string array parameter `input` and returns a text string with newlines, indentations and spaces as presented in the body of the method. Notice how the `FOR` and `ENDFOR` is be placed inline with the code. `person` is the current element of `input`.

```
def static generatePersonHTML(String[] input) {
  writeToFile('''
  <html>
    <body>
    «FOR person : input»
      <p>Person #«input.indexOf(person) + 1» is «person»</p>
    «ENDFOR»
    </body>
  </html>''')
}
```

Here is an excerpt from the generator program where templates are used to generate an action method where required. The formatted string is stored in the variable `methodString` which is written to a class file. The example also illustrate how conditional template content is possible using `IF`/`ENDIF`:

```
methodString = '''
/* Generated */
@FXML
def handle«eventType»«widget.id»(«eventType» event) {
  «IF eventType == "MouseEvent"»
  if(event.button == MouseButton.PRIMARY){
  «ENDIF»
    val resource = getResourceForScreenFile(screenName + ".screen")
    var id = getPropertyForNode(event.source as Node, "id")
    if (id != null && id != PropertyResult.NO_SUCH_METHOD){
      resource.performActionForWidgetId(id as String)
    }
    resource.evaluateRules
  «IF eventType == "MouseEvent"»
  }
  «ENDIF»
}
'''
```

After variable interpolation, in the case where `eventType` is the string `"MouseEvent"`, the result is:

```
/* Generated */
@FXML
def handleMouseEvent37(MouseEvent event) {
  if(event.button == MouseButton.PRIMARY){
    val resource = getResourceForScreenFile(screenName + ".screen")
    var id = getPropertyForNode(event.source as Node, "id")
    if (id != null && id != PropertyResult.NO_SUCH_METHOD){
      resource.performActionForWidgetId(id as String)
    }
    resource.evaluateRules
  }
}
```

The generated code is human readable which makes debugging a whole lot easier and opens up the possibility of modifying the generated code as if it was written manually.

7.2 1st Iteration

Although researching other prototyping tools and designing was a big part of the first iteration, it was desirable to begin exploring the technical side early on. Starting to work with Eclipse EMF and programming test examples is a great way to get to know the tools and to generate ideas.

7.2.1 Initial architecture

Choosing an appropriate program architecture for the prototype can ease the development process. Based on the requirements described in section 4 it was logical to split the *Generator* and the *Runtime Application* into separate programs. One of the major reasons being that the two processes can work independently of each other, allowing for the generator to parse new sketches while the runtime application is executing. The runtime application can then update the newly parsed and generated files during program execution without restarting. Splitting the two processes also allow alternative runtime applications to utilize the generated files, or to use another generator without changing the runtime application.

When making the prototype work with property lists (see Figure 5.1) the architecture was fairly simple. The generator created FXML files of the WireframeSketcher screen files, and the Runtime Application loaded them into a JavaFX environment. This was an important step in mapping what functionality was needed in order to get a JavaFX version of the sketch. The Runtime Application cached the FXML file together with a MD5 hash of the file. This allowed the user to edit the sketch and run the generator while the application was running. If a change was made to the sketch and the generator was run, the runtime application loaded the new file and updated the cache. If the file was not edited it simply loaded the cache.

Figure 7.1 depicts the architecture.

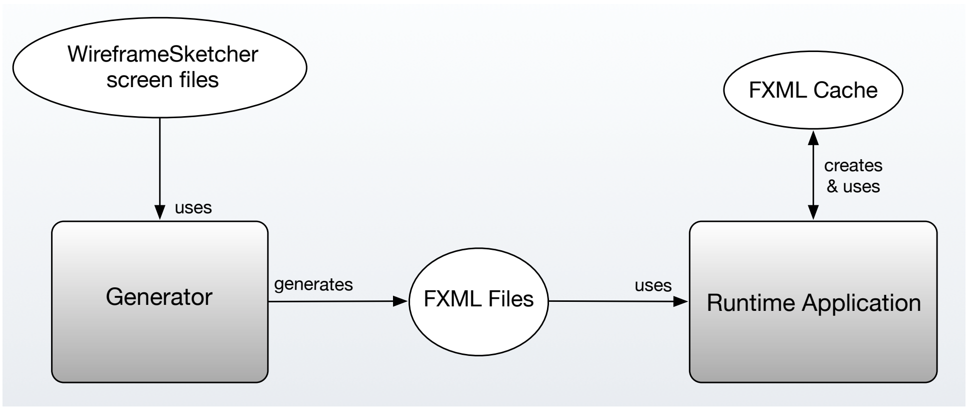


Figure 7.1: The initial architecture of the prototype.

7.2.2 Choosing JavaFX layout styles

One way to increase the functionality of a generator is to generate FXML in a layout depending on how the sketch is parsed. That is, the parser could interpret what layout is most appropriate for the sketched elements. An example is the property sheet where the generator could interpret labels on the left side and text fields, checkboxes or radio-buttons on the right side to be a property sheet. This would then enable the generator to group those elements inside a box, which could then float inside the window, enabling automatic resizing. This aspect was not explored much, because it is very narrow and time consuming. But two other layout styles was implemented: *AnchorPane* and *GridPane*

AnchorPane is a layout style where elements are *anchored* using absolute coordinates. That means that the x and y coordinates of the sketch widgets can be translated directly into JavaFX nodes with the same coordinates. A part for small differences in widget and node dimensions, the layout becomes identical. The down side is that resizing the window will either crop the elements out or introduce white space.

GridPane is on the other hand *not* an absolute layout style. Instead of using coordinates to position elements, it utilize a grid structure with rows and columns. Elements are positioned in the grid using row and column numbers, e.g. $(0, 0)$ would equal the upper left corner. $(2, 5)$ would equal the second row and the fifth column. The advantage of using grids is that resizing is handled automatically by the application in a way that allow elements themselves to be resized. The way the generator implements GridPane is in two main passes. First it registers the left and right side of each widget. For each widget it checks if there are any other widgets that start inside its boundaries, if so that's a new column. If there is only space between two widgets its adds a new column and sets the minimum width of that column to the space. This assures that the layout ends up looking like the sketch. The same process is done for the top and bottom sides of the widgets. Then they are positioned in the grid using the information about the sides.

After some experimentation it was decided that when sketching designs it was adequate to use AnchorPane since the elements already were placed where the user wanted them. Thus, AnchorPane became the default layout. Further development could enhance the parser to combine AnchorPane, GridPane and other layout styles in order to create even better resizing capabilities.

7.3 2nd Iteration

Most of the work in the second iteration was related to parsing the scripting format and handling the underlying screen decorator models in such a way that the program functioned properly. Figure 7.2 depicts an overview of the final program architecture together with its interaction with WireframeSketcher.

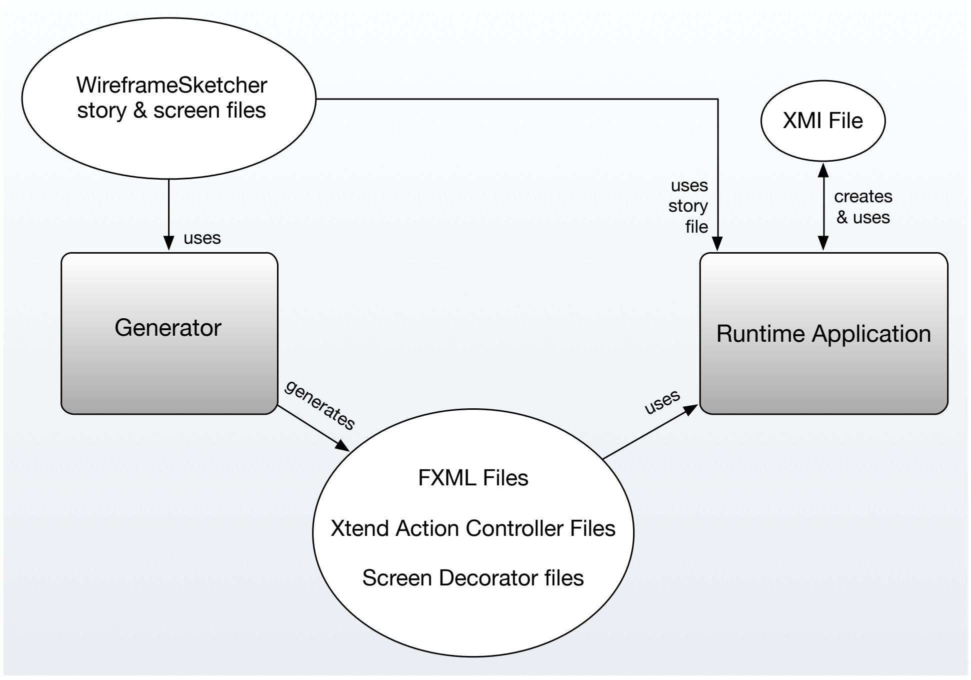


Figure 7.2: Overview of the prototype program architecture and the interaction with WireframeSketcher. The runtime application uses the story file to deduce which FXML file should be loaded first.

One requirement has to be met in order for the generator to work with WireframeSketcher. The Wireframesketcher project must utilize a *Storyboard* file (`.story`). This is required because all the Wireframesketcher screen files are deduced from this one file. This approach enables the user to have several *unused* screen files in the project, and also allow for screen files which are not in the same directory. In addition to FXML files for graphical markup and Xtend Action Controllers that map e.g. buttons to action IDs, the prototype utilize *screen decorator files.* They include the specific actions and markup layout rules. Screen decorator files are a result of the decorator model technique.

7.3.1 Prototyping process

The whole prototyping process from an idea to a functional prototype is as follows:

- **WireframeSketcher** The user creates one or several sketches in WireframeSketcher and save the files as normal. In order to demonstrate the features of this prototype the user also decorates the sketches (see section 5.2), although this is not technically required.
- **Generator** The generator loads the Story file and deduces which screen files should be parsed. For each screen file a corresponding FXML files is created. These are the JavaFX layout markup files.
- The generator create Xtend Action Controller files for each sketch file. These files will control ActionEvents and MouseEvent in the runtime application.
- The generator parses the special widgets who are used for the decorator technique and creates *ecore* files for the data variables and *screendecorator* files for the Actions and Styles (see section 5.2 for more information about the decorator models)
- **Runtime Application** The runtime application creates an XMI file with instances for each *ecore* files. It deduces which screen file should be loaded first by parsing the Story file. It then loads the screen file's corresponding FXML and Action Controller file. Before showing the newly loaded FXML file the runtime application evaluates the Actions and Styles found in the corresponding Screen Decorator file.
- The runtime applications reevaluates the rules each time an action is performed (e.g. Button pushed, mouse click on text) and stores any variable changes dictated by the Actions inside the XMI instance.

To see the inner workings of the process in greater detail, please refer to the source code in Appendix B.

Chapter 8 Analysis

8.1 Comparing manual and generated implementations

In the process of manually implementing a sketch using JavaFX, patterns begin to surface after a while. These patterns can be generalized in the process of automatically generating implementations. The following sections shows how a sketch made in WireframeSketcher could be implemented manually using JavaFX, and how the implementation compare to the generated solution developed during this thesis. To get the source code of the developed programs see Appendix B.

8.1.1 The *sketch*

This sketch example represents the *Login* section of an application. It is a minimalistic example, but it should convey the generalized technique that has been realized.

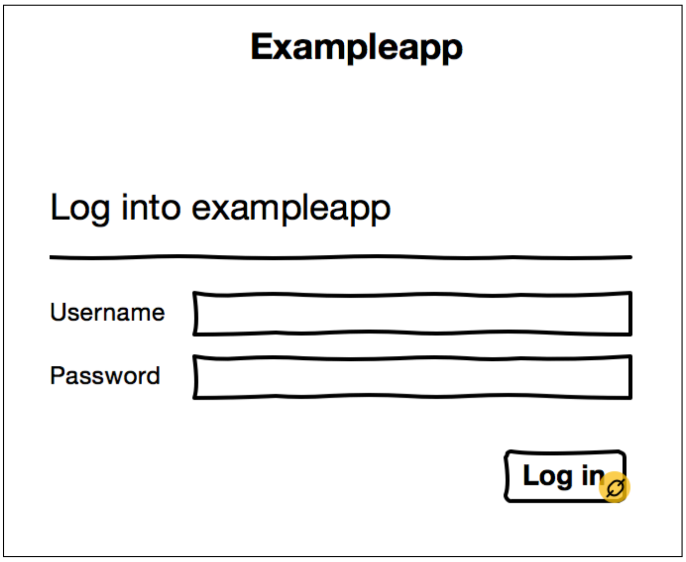


Figure 8.1: A WireframeSketcher *sketch* of a Login section.

One can imagine that the input fields store the username and password when the login button is pressed. Pressing the login button also navigates to another scene.

8.1.2 Implementing the sketch manually using JavaFX

When implementing the sketch using JavaFX the developer has to create the FXML file either by using SceneBuilder or by typing XML manually. The markup is then stored in `login.fxml` and the program is implemented in `AppController.xtend`. Notice that on line 6 of Listing 8.1 `login.fxml` the controller class is set to AppController (`fx:controller="application.AppController"`). When implementing a JavaFX program with FXML markup files, a controller include action methods used to handle e.g. button presses. These actions are referenced from the FXML file using `onAction="#methodName"`. The methods can reside in the AppController itself or in another class file. Following are three possible methods to structuring the program:

Method 1: Using the AppController itself as `fx:controller`. The FXML files references AppController through `fx:controller="application.AppController"`.

- 👍 Fast and easy to implement. Needs just one file.
- 👎 Poor scalability. For many FXML files the result is a huge AppController class file.

Method 2: Using a dedicated ActionController class to handle all action methods. The FXML files uses `fx:controller="application.ActionController"` to reference the ActionController.

👍 Slightly better scalability and improved readability.

👎 For large projects with many FXML files it results in poor scalability since the controller file becomes cluttered with several action methods.

Method 3: Creating a unique ActionController class file per FXML file in order to handle specific actions depending on which FXML file is loaded. When the AppController loads an FXML file it also sets `fx:controller` to the corresponding ActionController programmatically. A naming scheme convention e.g. ***`filename`***`ActionController.xtend` could be used in order to load the correct controller for ***`filename`***`.fxml`.

👍 Great scalability for a large amount of FXML files. Very clear which controllers and action methods correspond to which FXML files. Excellent readability.

👎 Results in many files for small projects.

Listings 8.1, 8.2 and 8.3 show *login.fxml*, *AppController.xtend* and *ActionController.xtend* respectively. They represent *Method 2* discussed above.

Package and *imports* are removed in order to save space and improve readability.

Listing 8.1: login.fxml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AnchorPane xmlns="http://javafx.com/javafx/8"
    xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="application.AppController" maxHeight="-Infinity"
    maxWidth="-Infinity" minHeight="528" minWidth="349">
    <children>
        <Label alignment="CENTER_LEFT" layoutX="125" layoutY="7"  rotate="0"
    text="Exampleapp" textAlignment="LEFT" textFill="black">
            <font>
                <Font name="System Bold" size="18"/>
            </font>
        </Label>
```

```
 9          <Button alignment="CENTER"  layoutX="253" layoutY="224"
      onAction="#handleActionLogin" style="-fx-base:white;" text="Log in"
      textAlignment="CENTER"/>
10          <Label alignment="CENTER_LEFT" layoutX="24" layoutY="88" rotate="0"
      text="Log into exampleapp" textAlignment="LEFT" textFill="black">
11              <font>
12                  <Font name="System Regular" size="18"/>
13              </font>
14          </Label>
15          <Separator layoutX="24" layoutY="124" prefWidth="294"/>
16          <Label alignment="CENTER_LEFT" layoutX="24" layoutY="144" rotate="0"
      text="Username"  textAlignment="LEFT" textFill="black">
17              <font>
18                  <Font name="System Regular" size="12"/>
19              </font>
20          </Label>
21          <Label alignment="CENTER_LEFT" layoutX="24" layoutY="176" rotate="0"
      text="Password" textAlignment="LEFT" textFill="black">
22              <font>
23                  <Font name="System Regular" size="12"/>
24              </font>
25          </Label>
26          <TextField alignment="CENTER_LEFT" layoutX="96" layoutY="144"
      fx:id="username" prefWidth="222" text=""/>
27          <TextField alignment="CENTER_LEFT" layoutX="96" layoutY="176"
      fx:id="password" prefWidth="222" text=""/>
28      </children>
29  </AnchorPane>
```

As seen on line 9 the *Log in* button references the controller action method `onAction="#handleActionLogin"` . JavaFX will look for it in the controller class set during runtime. The AppController loads *login.fxml* and sets the `fx:controller` to a new instance of *ActionController*. Note that the controller can easily be set using the FXML file by changing line 6. It only remains to prove as an example.

Listing 8.2: AppController.xtend

```
1  class AppController extends Application {
2     override start(Stage primaryStage) {
3        // Loading the markup from login.fxml
4        val fxmlLoader = new FXMLLoader(getClass().getResource("login.fxml"))
5        // Using the dedicated ActionController class
6        fxmlLoader.controller = new ActionController
7        try {
8           val root = fxmlLoader.load as Parent
```

```
         primaryStage.scene = new Scene(root)
         primaryStage.title = "Login"
         primaryStage.show

      } catch (IOException exception) {
         throw new RuntimeException(exception)
      }
   }
   def static void main(String[] args) {
      launch(args)
   }
}
```

ActionController includes the action method `handleActionLogin` that is called when the login button is pressed. It also includes two special variables annotated with `@FXML` which makes them accessible between the FXML file and the controller using `fx:id="username"` and `fx:id="password"` on line 27 and 28 in Listing 8.1. In this example the variables are only printed to console, but they could be stored to disk and another FXML file could be loaded and displayed.

Listing 8.3: ActionController.xtend

```
class ActionController {
   /* These variables are references from login.fxml using
    fx:id="variableName" */
   @FXML
   private TextField username
   @FXML
   private TextField password

   /* This is the action method "handleActionLogin" referenced from
    login.fxml using onAction="#handleActionLogin" */
   def handleActionLogin(ActionEvent event){
      println("Username is " + username.text + " and password is " +
    password.text)
      // Store information
      // Jump to another screen
   }
}
```

8.1.3 Generating the implementation

The generator works almost the same way as the example implementation in the previous section, but it follows *Method 3* for structuring the program. First it generates one FXML file for each sketch. Then it generates one unique ActionController per FXML file, that include action methods for that specific FXML. For the generator to succeed with generating an implementation it needs to decorate the sketch further using the decorators discussed in section 5.3.

The *Runtime Application* (*AppController*) loads the FXML file and sets the corresponding action controller class file. When e.g. buttons that reference the controller are pushed the specific method is called and rules are evaluated and actions for that button is performed.

Figure 8.2 depicts the same sketch, but with decorating elements added to it. Two variables are declared and `=${name}` and `=${secret}` act as identifiers for the input fields and serve the same purpose as `fx:id` in the previous example.

The resulting generated code is very similar to the manual example implementation. *login.fxml* ends up with a corresponding ActionController file named *ScreenNavigatorControllerLogin.xtend* after following the naming scheme convention `ScreenNavigatorController`*`filename`*`.xtend`. It contains the action methods. In addition to the AppController, ScreenNavigatorControllers and FXML files the generator also create screen decorator files. An *ecore* file for the data and a *screendecorator* file for the actions. This is explained in depth in section 5.3.

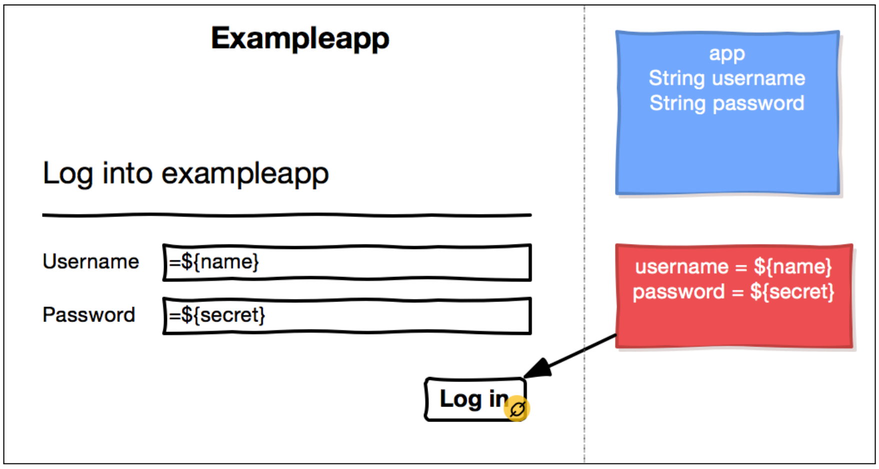


Figure 8.2: A WireframeSketcher *sketch* of a Login section decorated for the generator.

Chapter 9
Evaluation

9.1 User-testing the prototype

Developing a prototype that transforms a WireframeSketcher sketch into a functional prototype could yield different results. But without testing the result it is hard to evaluate the different aspects of the prototype and rate how well they work. In order to do this the prototype has to be *user tested*.

9.1.1 How testing was performed

The prototype was tested with 6 students attending Computer Science, Informatics and Industrial Design at NTNU. There was 3 designers, and 3 developers. They were all master students. Industrial Design had recently been divided into two sub-studies: *Product Design* and *Interaction Design*. The former is less technical study which focuses on the design itself, while the latter is more technical and include an obligatory programming course as part of the study. Two of the three designers attended product design and one attended interaction design. The developer versus designer partitioning was a conscious choice made to see how the tool works for less technical users compared to very technical ones. It represents a realistic target group as this is often the users who work in teams in the industry. The users of such a tool typically want it to be easy to use and fast. Those are the two most important aspects.

It was a qualitative test that presented tasks to the user and it was dialogue based as opposed to monologue based which is more common when testing GUI. The rationale behind using a qualitative test instead of a quantitative test is that the purpose was to observe what the user was thinking, what the user liked or disliked about the prototype and what parts of the program the user seemed to struggle with. It is very hard, if not impossible, to collect that kind of data using a quantitative test. Also if the test should be quantitative the number of test users would have to be increased. Much time and planning must have been put into creating a selfcontained test application. As of the time of testing it was very impractical to setup the prototype on a new computer by a test user.

The test consisted of 5 main tasks, and the tester who facilitated the test read them out loud and explained in more detail what the user should do, if the instructions were unclear for the user. Before the test itself began the user was *primed* about the prototyping process, the different approaches used to create prototypes, Eclipse and WireframeSketcher and the prototype itself. Much emphasize was given to separate the difference between WireframeSketcher and the prototype. A quiet observer took notes during the test. After priming was over the tester started with *Task 1*.

9.1.2 The tasks

- ✓ Task 1: Interpret how *muniapp* would behave if run through the generator.
- ✓ Task 2: Interpret how *exampleapp* would behave if run through the generator.
- ✓ Task 3: Implement a way to change the username in *exampleapp*
- ✓ Task 4a: In *exampleapp*, implement a way to store a textual note by using a TextArea and a Button.
    - Task 4b: Display the contents of the note somewhere in the window.
    - Task 4c: Display the contents of the note inside the TextArea itself.
- ✓ ~~Task 5: Be creative! Create anything you want. Explain what you are thinking. Do you feel there is a lack of functionality?~~

9.1.3 Challenges with the test

One of the big challenges of testing the prototype was that from the user's perspective, the prototype is integrated with WireframeSketcher. That means that when the user is asked to add functionality that require a new Button, the user has to remember where this is done in WireframeSketcher. The initial priming did decreased the need for intervention, but in order to observe things related to the prototype itself, the tester helped the user in such cases. Even though the tester intervened many of the comments the users made was strictly related to WireframeSketcher and not the prototype. The test itself was actually tested by a user, together with a facilitator and an observer, which evaluated *Task 5* to be potentially problematic. After the first real test *Task 5* was removed because the user unwittingly focused more on WireframeSketcher than on the prototype.

9.1.4 The sketch applications

Exampleapp

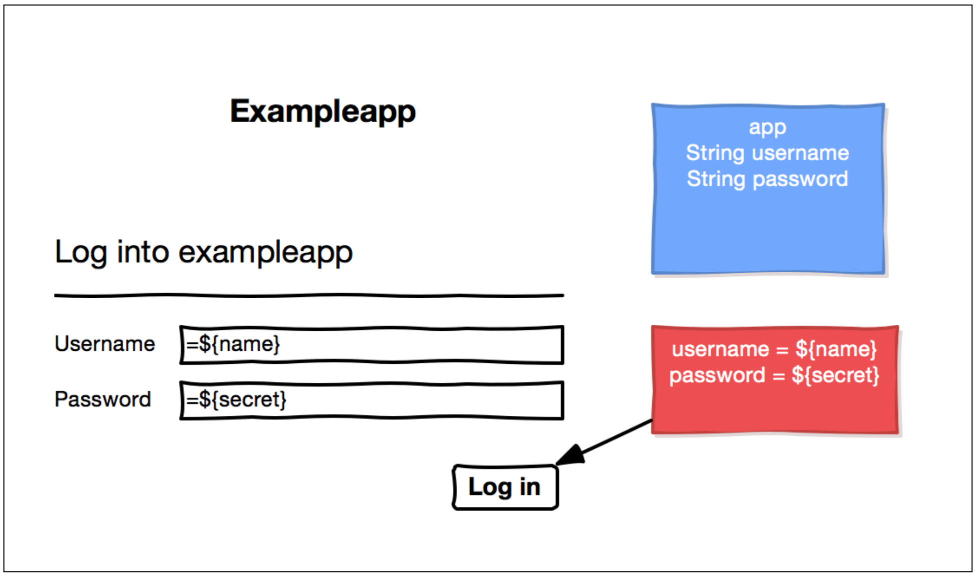


Figure 9.1: The Exampleapp sketch login section.

The sketch application *exampleapp* was created in order to provide a simple demonstration of the prototype and in order to test how a user might extend it with functionality. It was also used as part of the *priming* given to each user before the test. It includes a login section and a section for changing the password. The sketch is decorated with data, actions and styles. It also display how variable values can be displayed and how input fields can be created. The *Log in* button links to the section for changing password.

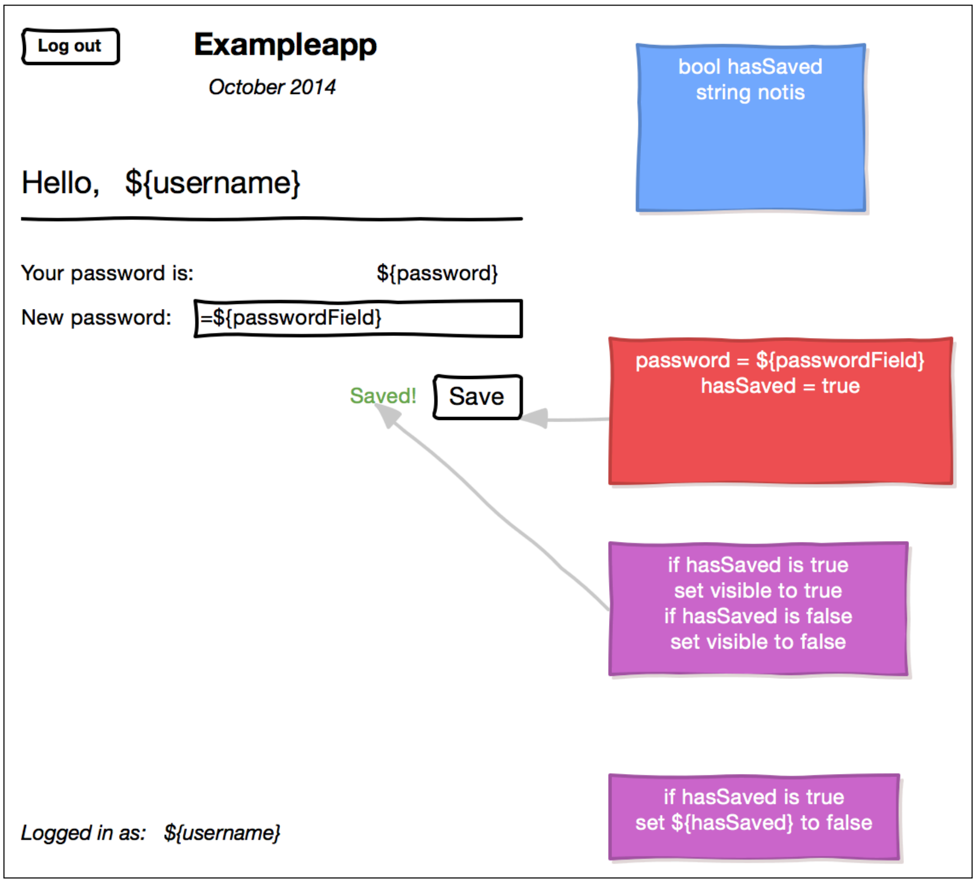


Figure 9.2: The Exampleapp sketch after logging in.

There are two styles in the sketch. One is linked to the label *Saved!* and one is not linked to anything. The former sets the label visible if the value of `hasSaved` is `true`, while the latter is evaluated every time something triggers an action e.g. a button is pressed. The result is that the label *Saved!* is visible until the user press *Log out* (or any other button or label).

Muniapp

The sketch application *muniapp* was created in order to demonstrate all the functionality of the prototype. It models a typical ”todo” application where the user adds tasks that needs to be done. It gives realistic examples related to the application, and some examples that just demonstrate functionality. The *muniapp* was used as part of the primer that each user was given before the test began. Muniapp consist of a login section, the main section and an *add new event* section. The login section is almost identical to Figure 9.1. The main section is depicted in Figure 9.3 and the *add new event* section is depicted in Figure 9.4.

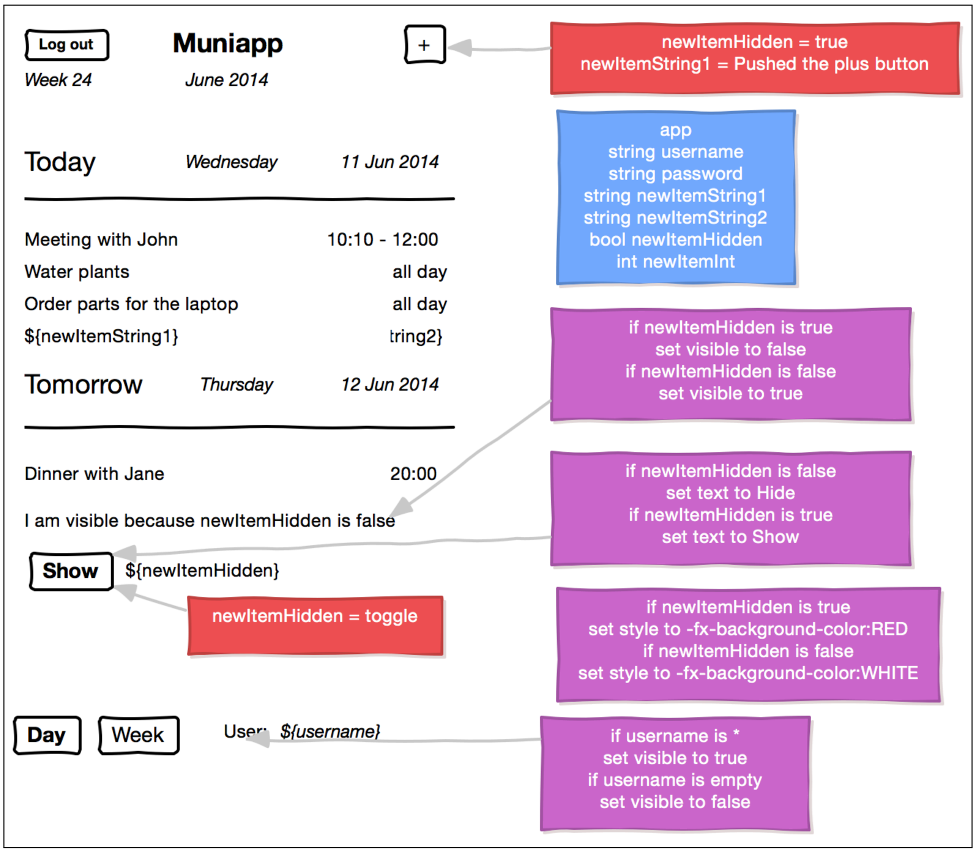


Figure 9.3: The Muniapp sketch after logging in.

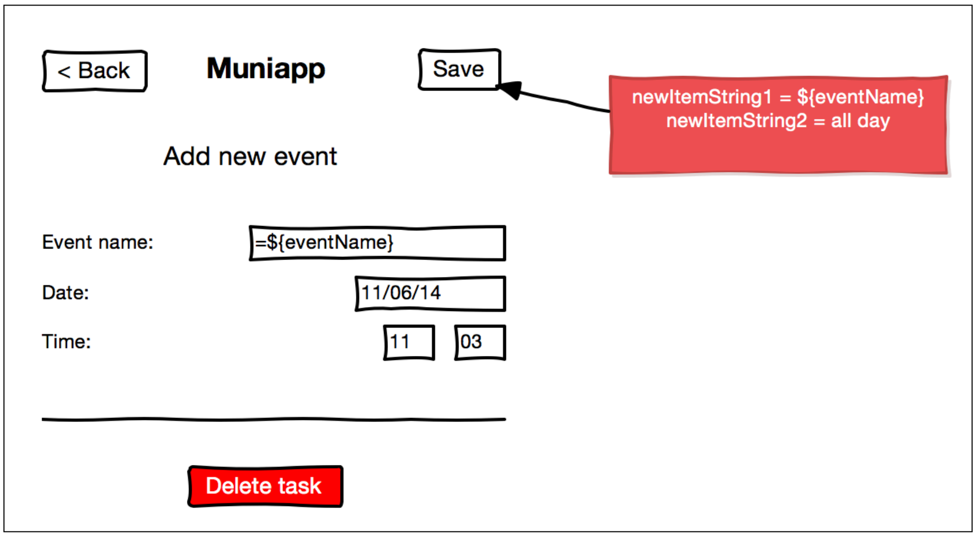


Figure 9.4: The Muniapp sketch for adding a new event.

The main purpose of Muniapp is to demonstrate all the features of the prototype and as such exist as a set of working examples that the user can learn from. e.g. `newItemHidden = toggle` is an action connected to the *Show* button. It toggles the boolean value between `true` and `false`. The bottom style connected to the label *User:* hides the label if the username is an empty string. For an in-depth explanation of the decorators see section 5.3.

9.1.5 Evaluating the user-tests

The evaluation of the user-tests tries to focus more on conceptual ideas, rather than design details. E.g. it is more interesting if scripting in general is a viable approach, and less interesting which syntax is used. It is more a discussion based on the observations rather than a point to point comparison between how the users solved each task. Detailed anonymous observational notes can be found in the Appendix A. In addition to the test itself, they record which year and program the students attend, and what relevant background they have. e.g. programming or GUI design course. It's important to note that all the *designers* had been exposed to at least a small amount of programming prior to testing and it should also be noted that the users were primed with information about the prototype before the test, so testing how intuitive certain implementations are might be inconclusive. The users discussed several of the implementation choices. Some of these choices have alternatives that might be better and will be discussed in section 9.2.

The graphical decorators

If this test indicates anything it is that representing data and scripts graphically together with the prototype is a good idea. All the users said that they liked how easy it was to get a clear view of what was happening. The semantics behind connecting decorators to widgets using arrows was well received, although many wanted the arrows to be created automatically and snap to the widget when dragged. This is a WireframeSketcher limitation, but underlines a good point.

The data decorator was probably the easiest to understand. Most users used app in order to declare global variables. Some gave feedback that they didn't understand the point with dividing variables into a scope hierarchy. It was pointed out that one of the strengths with the graphical decorators was that all the data and script were visible. But when global variables were declared in other sketch files, that was no longer true. The feedback regarding color coding the three decorators was positive. It helped the users differentiate between the decorators and their purpose. Two of the users who had previous experience with Axure RP said the prototype was easier to use, especially when knowing it came to dynamics such as actions etc. One user said it was more *programming based* than Axure RP.

The scripting format

As mentioned the users were primed with information regarding the prototype. This included the various scripting formats, although not in great detail. It was thus possible to ask questions about the formats. The users with heavier programming background found the format to be understandable, but more like a scripting language. The designers thought it was understandable after the primer, and some said they understood it after studying the examples. When asked about the format, no one had any suggestion as to how it could be improved. Some did mention that like most languages it requires some learning in the beginning, but ones that is over it was easy to use. One user wanted the actions to follow the defacto standard of declaring strings within double quotes e.g. `stringVar = "Several words"`. But then commented that the user would have to escape the quotes if he/she wanted to use them in the string. Most users wanted to be able to drag arrows from one style to several widgets. The formats `${variableName}` used to display the value of a variable was easy for most to understand. But `=${identifier}` used to identify an input field was a little hard to understand before examples was given. Also, only the most programmer savvy users understood the combination `=${identifier}${variableName}` meant to indication both an input field and displaying the variable value. It would seem that the prototype was easier to use for developers rather than designers, but one of the designers who had programming experience found it easy to use and stated that he *would like to develop apps this way.*

Summary

Using a qualitative and dialogue based user-test was a good choice for highlighting positive and negative parts of the program. Although there was only *six* users who performed the test, using a *qualitative* method resulted in some usable data. The graphical decorator seem to work as intended, and they seemingly provide a clear overview for the user. The users who had experience with other prototyping tools liked this graphical approach more than the dialogue approach in e.g. Axure RP. Two users also found the dynamic text feature to be helpful in cutting down on unnecessary duplicates. It might seem that the prototype is too *scripting based* for designers without programming background, but the for the indented user group of *technical designers* it seemingly fits quite well. The scripting format works ok, but further testing should be performed to see if it can be improved. Section 9.2 discusses possible changes based on the user tests.

9.2 Evaluating the design based on the user-tests

There are several design choices that could be implemented differently. One of the criteria for the program was to implement a scope hierarchy for variables, in order to have local and global variables. The rationale behind the choice was that a variable e.g. *isSaved* could have different meaning for a widget, a screen and globally. Several test users pointed out that they would rather be able to prefix the variables with the containing element. In other words if variable `var` was declared in `screen1` it could be referenced using `screen1.var`. This would also quickly show the user where the variable is declared. It may be unnecessary to have scopes at all. It might be the case that it increases the complexity of the program without yielding a better result. The drawback is that variables would have to be declared globally and as such variable names must be informative e.g. `screen1SaveButtonHasBeenPressed` for a `boolean`. If this is better or worse require more testing, but based on the user-tests scoping variables is not a very important feature. This might be a false-negative due to a flaw with the test itself.

As mentioned in section 6.2.1 the scripting format used is based on a mixture of common scripting languages and natural languages. There are similarities to AppleScript and semantics such as equals, which is often represented using two equal signs `==` has been replaced with `is`. One equal sign often used to set the value of a variable is replaced with `set` *property* `to` *value*. The goal has been to create a format that is more understandable for users with little to no programming background. However, this was not a priority during the development process. Several things should be mentioned regarding the user-test and the format. Most users found the format easy to understand after spending some time with it. It is always desirable to design a format that is *intuitively* easy to use. But some acquired knowledge might be required, and should not be a problem. A few of them commented on the repeating format, such as the *style* connected to the label *Saved!* in Figure 9.2. A final format should include else to reduce typing. One user pointed out that strings should be surrounded by double quotes *"string"*, but discussed the problems of having to escape the quote characters if they were to be used. It would seem that if the goal is to keep it simple, double quotes might be bad choice. One of the difficulties to creating a scripting format such as this is to satisfy the user requirements. If the user has never seen a scripting language before, a natural language based scripting language could prove effective. However, for technical developers the limiting format might be a cause of annoyance. To meet both user groups the program could allow the user to write a full fledged scripting language like JavaScript, Perl or Python. All users wanted to be able to connect one *style* to several widgets using multiple arrows. This should be possible to reduce duplicates. Arrows should be handled automatically when dragging from one decorator to a widget.

9.3 Evaluating the research questions

The research questions were as follows:

- *RQ 1: How can a low fidelity sketching tool be enriched with information that enables generation of a functional prototype.*
  - *RQ 1.1: How can functionality be added to a sketch without loosing the simplicity inherent in low fidelity sketching tools.*
  - *RQ 1.2: How can the added information be utilized together with a modern toolkit in order to realize a functional prototype.*

As seen throughout Chapter 6 and 7 the *Generator* and *Runtime Application* programs developed during this thesis answers the research questions. They represent a possible implementation that create functional prototypes from sketches made using WireframeSketcher by extending it with information in the form of special *notes*. In other words they *extend WireframeSketcher with prototype generation.* And the prototype functionality is added to sketches using the decorators discussed in section 6.2.1. Chapter 7 show how the modern toolkit JavaFX is used. It is state-of-the-art and utilize the generated layout - the FXML files - as markup. It is indicated by the user-testing in section 9.2 that the solution is viable and retain at least some of the simplistic nature of a low fidelity sketching tool, although it is difficult to say if it is simplistic enough. If a scripting format is considered easy by the user, the solution is likely simplistic enough. It is feasible to create normal application using this method, however, it would require further development.

9.4 Further development

Finishing incomplete implementations

Some of the criteria outlined in section 4.5 and 4.6 were not completely implemented. Alhtough it is not important with respect to the research questions it should be possible to install the program as a plug-in and the user should have the option to *Generate Function Prototype* when right clicking on a Storyboard file. Not much time is required to finish the implementations and it would increase the availability and usability of the program.

Looking at different aspects

During the development process of the prototype several ideas were discussed, but only a fraction of them were realized. As discussed in section 5.1 there are several *aspects* that can be of great interest for further development. The results from the user-tests indicates that the system has potential. It would be very interesting to see how the other aspects would improve the program.

Specifically a more expressive scripting format could be useful and allow for more functionality such as a simple natural-language based syntax for novice users and JavaScript for advanced users. The program could interpret which format is used automatically. Custom *note* widgets seem to work quite well, but the arrow widget used to connect each widget is cumbersome to use. It does not snap to the widgets and it must be added manually. A better solution is to be able to connect the two widgets by *dragging* from one to the other. It is most likely not be possible to implement such a feature as an extension, but it could be implemented in the tool itself. [1]

The *variable scope* hierarchy should be redesigned and tested for effectiveness. As a result of the user-tests it seems unnecessary to isolate variables depending on their declaration layer. A better solution might be to use global variables, or as another option declare local variables in each screen and allow referencing from other screens possible i.e. allow the variable `isLoggedIn` declared in screen1 to be accessed by screen2 using `screen1.isLoggedIn`. What solution works best for the end user requires more user testing. Using *model data* in order to manually write dummy data in lists etc. could be very useful. The prototype would automatically fill the desired elements with model data such as names, addresses and food ingredients. Not only would this save time for the user, it could also be combined with *real* model data.

[1] In an email correspondence with Peter Severin, the author of WireframeSketcher, it was stated that such a connector tool is planned.

Exploring different layout styles is also another interesting aspect. It might increase the usability and also allow users to provide a close-to-reality functional prototype that could be used in meetings with stakeholders, or in user-tests. There are two general sub-aspects of this. One is for the program to generate a functional prototype that use the same sketched layout (e.g. iPhone or Android) already available in WireframeSketcher. The result being a functional prototype that looks exactly like the sketch, not an approximation. The other sub-aspect is to include *hi fidelity* layouts of the same styles in order for the functional prototype to look like a native application. This latter aspect would diverge from the ideology where prototypes should be very simple and focus on layout position over actual graphical design, but it should be explored. The end result could be a full fledged working program.

9.5 Conclusion

The goal of this project was to look at a way to extend a low fidelity sketching tool in such a way that the sketches could be turned into a functional prototype. Different aspects, such as how the functionality could be added to the sketch, was an important part of the process. During the project several prototyping tools have been categorized, and together with the reported opinions of students taking the design course IT3402 [2] a set of features was chosen. There are several interesting aspects to study, but this project followed a *proof of concept* mentality and focused on aspects the was closely related to the research questions. The developed solution can easily be integrated with Eclipse and uses EMF and JavaFX to realize the prototype. It uses a graphical approach to extending WireframeSketcher designs with functionality, and the user-tests strongly indicate that this is a viable solution for making functional prototypes. The process of designing a more traditional tab based approach to adding functionality indicated that it was less clear and more complex than the graphical approach. The graphical approach seemed to retain some of the simplistic nature of typical sketching tools, but it is hard to conclude if it is simplistic enough, especially combined with scripting. In order for the solution to be practical for the end-user it would require more development, but it is the authors opinion that the research questions has been answered and that the report discusses and presents the interesting and relevant parts of the process.

References

[1] Roel Wieringa. Design science as nested problem solving. In *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, DESRIST '09, pages 8:1–8:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-408-9. doi:10.1145/1555619.1555630. URL http://doi.acm.org/10.1145/1555619.1555630.

[2] User interface design course. Online. URL https://sites.google.com/site/userinterfacedesigncourse/. Accessed may 12th 2014.

[3] Dirk Bäumer, Walter R Bischofberger, Horst Lichter, and Heinz Züllighoven. User interface prototyping—concepts, tools, and experience. In *Proceedings of the 18th international conference on Software engineering*, pages 532–541. IEEE Computer Society, 1996.

[4] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decis. Support Syst.*, 15(4):251–266, December 1995. ISSN 0167-9236. doi:10.1016/0167-9236(94)00041-2. URL http://dx.doi.org/10.1016/0167-9236(94)00041-2.

[5] Douglas Harper. Online etymology dictionary. Online, 2001-2014. URL http://www.etymonline.com/index.php?search=prototype&searchmode=none. Accessed april 29th 2014.

[6] Carolyn Snyder. *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces (Interactive Technologies).* Morgan Kaufmann, 1 edition, April 2003. ISBN 1558608702.

[7] Miriam Walker, Leila Takayama, and James A Landay. High-fidelity or low-fidelity, paper or computer? choosing attributes when testing web prototypes. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 46, pages 661–665. SAGE Publications, 2002.

[8] Reinhard Sefelin, Manfred Tscheligi, and Verena Giller. Paper prototyping - what is it good for?: A comparison of paper- and computer-based low-fidelity prototyping. In *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, pages 778–779, New York, NY, USA, 2003. ACM. ISBN 1-58113-637-4. doi:10.1145/765891.765986. URL http://doi.acm.org/10.1145/765891.765986.

[9] Popapp tool. Online. URL http://popapp.in. Accessed june 5th 2014.

[10] Invision prototyping tool. Online. URL http://invisionapp.com. Accessed june 4th 2014.

[11] Wireframesketcher tool. Online. URL http://wireframesketcher.com. Accessed februrary 11th 2014.

[12] Balsamiq prototyping tool. Online. URL http://balsamiq.com. Accessed july 12th 2014.

[13] Justinmind prototyping tool. Online. URL http://justinmind.com. Accessed june 6th 2014.

[14] Axure rapid prototyping tool. Online. URL http://www.axure.com. Accessed july 3rd 2014.

[15] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, feb 2006. doi:10.1109/MC.2006.58.

[16] Richard Soley et al. Model driven architecture. *OMG white paper*, 308:308, 2000.

[17] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. Acceleo user guide. 2006.

[18] Gavin Nicol, Lauren Wood, Mike Champion, and Steve Byrne. Document object model (dom) level 3 core specification. 2001.

[19] Texlipse - latex for eclipse. Online. URL http://texlipse.sourceforge.net. Accessed november 13th 2014.

[20] Eclipse - ide. Online, . URL http://www.eclipse.org/ide/. Accessed november 13th 2014.

[21] Eclipse modeling framework project (emf). Online. URL http://www.eclipse.org/modeling/emf/. Accessed may 13th 2014.

[22] Lars Vogel. Eclipse modeling framework (emf) - tutorial. Online, 2007-2013. URL http://www.vogella.com/tutorials/EclipseEMF/article.html. Revision 3.0, Accessed november 11th 2014.

[23] Eclipse workbench user guide. Online, . URL http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-2.htm&cp=0_2_1. Accessed november 13th 2014.

[24] Using perspectives in the eclipse ui. Online, . URL https://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html. Accessed november 13th 2014.

[25] Javafx frequently asked questions. Online. URL http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html. Accessed october 28th 2014.

[26] Matt Neuburg. *AppleScript: The Definitive Guide: Scripting and Automating Your Mac.* "O'Reilly Media, Inc.", 2006.

[27] Scenebuilder 2.0. Online. URL http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html. Accessed november 10th 2014.

[28] Xtend - modernizing java. Online. URL http://www.eclipse.org/xtend/. Accessed may 13th 2014.

Appendix A
Observational notes from User-testing

Below are the observational notes made during the dialogue based user-tests discussed in section 9.1. They have been anonymized in order to protect the identity of the test subjects.

Test person #1
Gender: Male
Background: Informatics, Msc (5th year)
Start time 12:35, end time 13:25

<u>Abbreviations/codes:</u>
WS - WireframeSketcher
FL - the prototype
TP - test person

Starts by explaining what WS is and how it works, including its features and limitations. Explains that what he has added of functionality and shows what it can do.

Asked if the user has understood the relevant difference between WS and FL, got an affirmative answer.

Task 1: Interpret what "toduka" does: 10 min

Some misconceptions about exactly what he should interpret. Was supposed to understand what blue, red, and purple note is used for, not what the meaning of the app is. Seems to understand what the different notations mean, fail to explain out loud what he's thinking, getting confirmation nod. Does not quite seem to understand the script notation. Suggested that it displays the password form vs plain text. He understood the second part of the task. The fact that a style that does not have an arrow wont affect the entire screen was perhaps not entirely intuitive.

Task 2: Interpret what "exmpleapp" does. Time: 6 min (started 45)

Explains what the purpose of the app is.Tester asks for an explanation of what and why things happen. Taking some time (not much) to get an overview of what is happening in the app. Some confusion over what the tester really is looking for. Perhaps because the TP have knowledge regarding variables and data? Thinking that it is check the box that are properly referred, but it is the variable *checked* that is referenced. Explains what toggle and bool have to do with the checkbox. Only *checked* variable that was a bit confusing, the rest was properly explained. Tester repeat now what the different things do. Tester understand in hindsight how things are connected.

Task 3: Implement the ability to modify the username in exampleapp, time: 9 min (started 51)
The tester explains that it is the same way as changing passwords. TP says he would have copied the boxes and just replaced the text contents of them. Needs help with mac -keyboard layout to find bracket characters. He believes he is finished and explains how it works. He is able to explain it without problems. It worked! The tester suggests to add a field that shows existing username. TP moves things down, told to fix the arrows. Set arrows right? Yes, after som thinking. It ran without problems.

Task 4a: In "exampleapp": Implement the ability to store a textual note using a

TextArea and a button. Time: 7 min at full 4 (started 1:00 p.m.)

Adds a TextArea and Button without problems. Then enters the variable reference in the TextArea. Adds a property. Copies the existing action box and fill in some stuff that is correct. Creates an arrow from the action box to the new button. Tester demonstrates that the text has actually been saved by showing XMI file, so it works!

Task 4b: Implement a way to display the note another place in the window.
TP starts with copying an existing label. Changed single variable. Went quickly. It worked!

Task 4c: Implement a wao to display the note in the TextArea
It is in a way already solved, since he has done it before in the test. Can he do it? Yes, no problem!

Task 5: free rein. Create anything you want! Explain what you think and what you respond to. time: (started 1:07 p.m.)
Skipped this task after directions from tester.

Qustions asked by the tester:

Q. What do you miss functionality?
A. The notes, wondering how they work. Commenting on what happens if an arrow pointing errors. Tester explains how the program works in the background. Tester shows XMI files. Explains how the decorators (red, purple) is generated. Remember that actions must be no spaces, it understood the tester. Who is it aimed at (tester ask)?

Suggestions: Did not quite understand what the tester wanted in the beginning. Did not realize that the boxes were something that was done, but thought that it was only a few pictures in the background. What is the difference with or without the keyboard *app* in the blue box, the tester asks. TP thinks that it is a layout for mobile applications. Tester explains that app does globally, no keyword makes them only visible for the screen. Says that the tester should prime the TP a bit more before starting. E.g. when things are running what a style without an arrow is. And when a button is pressed all the values in the screen are evaluated.

Was there anything that was uintuitivt relative to script format?. TP thinks it is the same as programming common. Believes that it is only the way it is when you learn programming language. Clumsy having to drag the arrows manually without snap.

Test person #2
Gender: Male
Background: Informatics, Msc (5th year), IT3402 2014.
Start time 11:03, end time 11:34

Abbreviations/codes:
WS - WireframeSketcher
FL - the prototype
TP - test person

Tester starts with explaning WS. Explains that decorating has been added by him. Explains what the generator is and how it works. Continues with explaining the purpose of the arrows. TP nods and generally says yes. Seems like he has a computer background? NewItemString, understand how it works also.

11:15
Task 1: Intepret what "exampleapp" does.
TP: Updates the strings in the app independant of the input. It has to do with login. The først screen was easy to understand. Log out gets you back, shows the username you logged in with. Changing password is what is intended.
Tester: How does this look when the program is running?
TP: Shows password in clear text? Write the passord in order to add a new password. It is a label. By pressing "Save" button the label "Saved" will appear. {*loggedInAs}* is a label, the variable navn is displayed when running.

Tester: Password and username are variables. What is passwordField? TP: The textbox
Tester: What does part two of it means? TP: It shows the password that is already stored. Tester: Quite right.

TP: Doesnt that styles have to point to something? Tester: No arrow means it affects the whole screen during runtime.

11:22:

Task 3: Implement a way to change the username in exampleapp
TP: Can one action point to several things? Copies the action and writes the same text as in username.
Tester: Why did you name is usernameField? Is it something unique about the name?
TP: I'm just following the standard as I see it. The TP understand that the name does not have to be special.

Tester: Can you explain what happens when "Save" is pressed?
TP: No state is saved, change username to what is in the box.
Tester: There are states, but in the next login the variables are overwritten.

Task 4: In "exampleapp": Implement a way to store a textual note by using TextArea and a Button.

TP: Adds a new TextArea and a Button, thinks that he needs a variable to store the result in. Adds a variable in the blue decorator. Gives the TextArea a variable. Shows the stored variable in the textarea in the same way that username does it. Creates an action. Asks the tester if you can get an arrow from the action decorator. He understands it is required. Finishes.
Tester: Quite right, apart from a few missing brackets in the syntax. Runs the program.
TP: Verifies that the program works. TextArea is cleared when "Save" is pressed.

11:31

Task 4b: Display a note in another place in the windows
TP: Need a new textarea, but approx. the same.
Tester: says it doesnt necessarily need a textarea to display it.
TP: Try to use Text from WS palette.

It did not work because Text is not implemented.
Tester changed Text with Label and it works, so the approach was correct.

Task 4c: Display the note in the TextArea itself
Alreay implemented in 4b.

Task 5: Make whatever you want
Skipped by the tester.

11:34

Tester: What funcitonality do you feel was lacking? Something you felt was badly implemented?
TP: It can be difficult for someone who hasn't programmed before. Although it seemed like scripting, not programming. Haven't scripted much before, but it was understandable.

Tester: Anything that should have been said in the priming?
TP: Many foreign words, that non-computer people would have a hard time with.

Tester: Was it obvious what the script was affecting?
TP: Yes, didn't use styles that much but it was ok.

Tester: Diffictult to find something for people who haven't programmed before.

TP: Works very nicely for prototypes. It gives extra features. More advanced functions than Axure (his opinion). Advanced scripting language in Axure. It can have variables.
TP: FL has a nice layout which is very clear. It displays all the information at once. This was easier to use than Axure. Axure is a little cumbersome.
Tester: It's possible to change the scripting language with JavaScript, but simple natural languages would be easier for non-programmers.

TP: Someone might try to display variables without dollar signs
Tester: It has to be a little gained knowledge. Without the dollar sign the text itself is displayed.

Tester: Anything else to add?
TP: Nothing special. The colored notes were nice. Easy to see what was going on.

Tester: Have you used any other prototyping tools?
TP: InVision and Axure RP. Axure have some of the same features but the scripting is complex. I dont like it.

Test person #3
Gender: Female
Background: Computer Science (5th year), IT3402 2013.
Start time 12:00, end time 12:33

Abbreviations/codes:
WS - WireframeSketcher
FL - the prototype
TP - test person

12:00

Tester explains how the program works. Both WS and FL. Shows exampleapp.
Tester: What does newItemString do?
TP: creates a new TextField?

12:08

Task 1: Intepret what "muniapp" does.

Tester: What do you thinkg is going to happend? Please explain on a "variable level"
TP: When logging in the variable that is previously stored is checked. Password will be several asterix' since it is *secret*
Tester: What is the blue box?
TP: It is variables for username and password, type string.

Tester: Red box?
TP: It is the thing that sends username and password when you log in.

Tester: In the red box, one of the elements is the username that is also in the blue box. What is this?
TP: It's the variable used in the textfield as well.

New screen
Tester: What is the blue box this time?
TP: Boolean variable that is stored. The textfield.

Tester: The red box?
TP: Action to change the password.
Tester: How does the modification occur? What does the variable become afterwards?
TP: It is set to the value of "new password" textfield. Not sure what the second half of the field is. [It's not clear what TP is thinking]

Tester: What is the purple box connected to?
TP: It's connected to "Saved". It will be visible when save is pressed, hidden otherwise.

Tester: The purple box without a box, what does it do?

TP: It resets the boolean isSaved so that the "Saved" label is hidden.

Task 2: Interpret what "exampleapp" does

TP: Asterix means that it can be anything. Toggle is that hide/show changes when you pres it. At least it changes the layout.

Tester: Intepret the big styles box without arrows
TP: I think it affects the color of the text that is displayed by the show button.

Tester: What is newItemHidden? is it a string, int
TP: It's a boolean

Tester: if there was an arrow between it and the text it would have change the textcolor. But since it has no arrow it affects the whole screen.

Task 3: Implement a way to change the username in exampleapp
Tester: You can copy/paster elements, or drag them from the right hand side.
TP: I'm thinking it's pretty similiar to changing the password. Copy paste ftw!
Mentions that the placement will be random.
TP: Can one action point to several elements or just one?
She is thinking of using an arrow from the existing action box.
Tester: Asks rhetorically: What do you need the arrow for?
She votes not to use it and says che can use the same action box.
TP: Adds the field in the action box in the same manner as password. Works.

Task 4a: In "examplapp": Implement a way of storing a textual note by using a TextArea and a Button
Tester: How you design it is irrelevant.
He shows where the WS elements can be found.
TP drags one TextArea and a button and ads a new action box. Uses the dollarsign notation in the textarea like prior examples.
TP: the color of the arrow is different.
Tester: No problem. What do you think happends when you press "Save"?
TP: Ah, I need to store the variable in the blue box.
TP plays with the program and sees that it works.

Task 4b: Display the note in a different place of the screen
TP copies the textfield with text "Hello: ${username}" and changes it with "Text: ${text}". Moves the existing save button, but forgot to move the arrow.
Tester: Do you think there's anything special with the format you're using now?
TP: I think it just has to be the same in the field as in the other places it's used.

Task 4c: Display the note inside a TextArea
did that in 4a already.

Task 5 dropped by tester

Tester: What did you felt was lacking of functionality? Any comments?
TP: I like that it's presented graphically instead of dialogues. It was very a very well

presented way to solve the problems. Clear layout.

Tester: Was there anything that you felt was nonintuitive?
TP: No, not really. Different colors for the boxes was nice. I liked the graphical style of adding scripting. The initial priming was important though.
TP: Very nice way of making prototypes. Instead of using photoshop!

<u>12:33.</u>

Test person #4
Gender: Male
Background: Industrical design, specialization Product design (4th year)
Start time 13:30, end time 14:00

Abbreviations/codes:
WS - WireframeSketcher
FL - the prototype
TP - test person

13:30

Tester explains what a prototyping applicaiton is. That is makes simple prototypes that doesnt focus on the graphical design, but more of the layout and flow. Explains what FL is and how it works. Compares WS with Adobe Illustrator. Only static pictures that doesnt do anything. Runs the FL program and shows how it works. He says he have some experience with simple programming. Explains what the different colored decorator boxes do.

Task 1: Intepret what "exampleapp" does

TP says that ${username} shows the value of the variable.
Tester explains what the app keyword means int the blue box.

Tester: What is the blu ebox for?
TP: Not sure, it contains variables.

Tester: If "Saved" button is pressed, what happens?
TP: hasSaved variable is set to *true*. Password is set to whatever is in the field.

Tester: If you look at "Hello ${username}".What do you think the format in the field for new password means?
TP: Not sure.

Tester: What is the "Saved" text?
TP: Something that pops up when you press save.
Tester: Can you tell that from the purple box?
TP: Yes, if it's true its visible, othervise not.

Task 3: Implement a way of changing the username in exampleapp

Tester: Copy paste is ok. Desing is not imporant.
TP: No quite sure what you want me to do.
TP copies the text of change password and creates a new button with a corresponding action box. Correct implementation apart from the unecessary action box (could have used the same). Adds an arrow from the action box. Copies the text element that hides/shows. Changes the variables in the textbox with the correct username.

Task 4a: In "examplapp" implement a way of storing a textual note

TP adds a TextArea and Button. Thinks about creating a new blue box, but realize that it's not necessary. Was thinking about where the username password variables were.
TP: I'm a little cought up in creating everything like the previous examples. One button and a separate style for each.
TP thinks he should write code that store the text of the textarea.

Tester: Is the variable *note* declared anywhere?
TP: No, it has to be added to the blue box.
TP adds the variable.

Task 4b: Display the note in a nother place of the window.
TP: Copies a text element. Not quite sure how it should be done.
Tester: Are there any other elements in the screen that be of help?
TP says Yes and points at "Hello ${username}". Writes ${note} in the text box. Wants to use visibility to say if tehre is anything visible or not.

Tester demonstrates that the program works.

Task 5 skipped by the tester.

Tester:Do you have any comments?
TP: I think it's something I could have used for developing apps.
Tester: Anything you would have liked to be different?
TP points to WS specific "bugs" like text completion that doesn't make sense for the prototype.
Tester? Anything else?
TP: What was the tool called?
Tester: WireframeSketcher

<u>14:00</u>

Test person #5
Gender: Male
Background: Industrical design, specialization Interaction design (4th year), IT3402 2014
Start time 14:00, end time 14:30

<u>Abbreviations/codes:</u>
WS - WireframeSketcher
FL - the prototype
TP - test person

<u>14:00</u>

Tester explains what a prototyping application is. That is makes simple prototypes that doesnt focus on the graphical design, but more of the layout and flow. Explains what FL is and how it works. Compares WS with Adobe Illustrator. Only static pictures that doesnt do anything. Runs the FL program and shows how it works. He says he have experience with programming. Explains what the different colored decorator boxes do.

Task 1: Intepret what "muniapp" does

TP: If the username is anything, set visible to true.
TP understand the purpose of the Arrow but he's not sure what the rules of "connecting" it is. Can it overlap or does it have to be exact?

TP: The action turns Show button on and off.

Tester explains what toggle means
TP: So it if you press on Show, the text becomes visible. What is the text variable?
Tester recognizes that the user is familiar with programming langauges.
Tester: It is the property on the button used by JavaFx.
TP didn't quite understand so Tester explains more.
TP: It would be more intuitive if the string was surrounded by quotes. Easier to understand, but then you have to escape if you want quotes. Hm.

Tester explains the reevaluation that occur each time something is pressed.

Task 2: Intepret what "exampleapp" does

TP: Declared variables in the blue box. Not sure what "app" means. By pressing "log in" the red boxes are performed and the variables are updated.
Tester: App means that variables are visible for the whole application, not just the screen.
Tester: What is visible if this is run. Ignoring the boxes.
TP: Username and password is displayed. Not quite ure what happens in "new password". Maybe it's stored and displayed at the same time. By pressing "Save", the red box is performed and the text by it will be visible. If it is not pressed, the text

will be invisible.
Tester: Explain the two variable elements of the text box
TP: It will show the variable and write the other.
Tester: What does the purple box do without an arrow?
TP: If the button is pressed, it will set the variable to false.
Tester: When is it called?
TP: everything time something is updated

Task 3: Implement a way to change the username in "examplapp"

TP copy pastes from the password textfield. Changes the text contents. Adds the statements required in the existing action box. Went fluently.

Task 4a. In "examplapp" Implement a way of storing a textual note using TextArea and Button
TP moves everything in the screen down a little to give extra room for a TextArea that he drags from the WS palette.
TP: I'm thinking of writing pretty much the same as with the username, but using my own variables.
TP uses the existing "Save" button and updates the corresponding action box with the new statement. Adds the variable *note* in the blue box.

The program runs and everything works.

TP: I hope the variables are stored, but I guess they will be overwritten with the new ones.
TP compares it to Axure.
TP: It's a little bit more programming oriented than Axure.

Task 4b and 4c is alreay done.

Task 5 is skipped by the tester.

Comments?

TP: If the arrows snapped it would be much better and easier to use.
Tester informs that WS is about to add that feature (according to the founder Petru Severin)

TP: The things that affect the screen should be able to minimize and group. The global once could be grouped. This could save space.

Test person #6
Gender: Male
Background: Industrical design (5th year), IT3402 2013
Start time 11:00, end time 11:30

<u>Abbreviations/codes:</u>
WS - WireframeSketcher
TP - test person

Tester explains how WireframeSketcher works by presenting an example (F5 Presentation Mode). Explains the prototype and what the differnet colored decorators are. Blue is data, red is action, purple is styles. Explains the purpose of the arrow. Explains the ${variable} format and the reasoning behind it. Tester explains the difference between the prototype and wireframesketcher.

Task 1 Intepret what "muniapp" does

Tester: What do you think it does? Newstring. What is the purpose of the dollar sign and brackets? What will be visible during runtime?
TP: I have no idea about the programming. Is it a code for something else that should be displayed?
Tester: Yes, it displayes the value of the variable

Tester: What happens when you press show?
TP: It changes the variable to true and shows the text

Tester: What is the purple box that points to the buttons?
TP: I think its the same as action boxes.
Tester explains the difference.

Tester: What happens in the login screen?
TP: It shows the username an dpassword when you press login.

Tester explains the equals sign = format =${variable} used a input fields.
TP: So it saves the input for further use
Tester shows the program generation and that it runs successfully.

Task 2: Interpret what "exampleapp" does

TP: It looks familiar. The text written in the the fields username and password is stored. By clicking login the action box is performed. Because it says app the variables global. [This was explained in the primer]

Tester: Start by explaining the blue box
TP: I'm assuming it is something that checks if the username is true or not
Tester: Just focus on the blue box itself
TP: I'm assumin the stored value is true or false since it says Boolean

Tester: What about "Hello ${username}"?
TP: It shows whatever is set to username. Unsure if it is displayed by clikcing save or

by login. Most likely login.

Tester: What abour the password field? What happens when you press "Save"?
TP: I don't quite understand what passwordField does since it's already stored a password. But it says new password so maybe it deletes the old and sets the new by pressing save.

Tester: What is username and password set to?
TP: It says "secret" so I'm guessing it shows the password using asterix'

Tester shows that the first part of the format is equal to the login
TP: It will display the password as code for the secret password?

Tester: It will be blank since it stores whatever is in the field. What does part two of the field mean?
TP: It is what already is stored in the password variable. It will display it.

Tester: What happens to hasSaved by pressing "Save"?
TP: It is set to true

Tester: What about the green text?
TP: By pressing "Save" the green text "Saved" will be visible if it is true and invisible if false.

Tester: What about the style box with no arrows?
TP: It will affect anything.

Tester: If you intepret the purple style box literallly word by word. What do you think it does?
TP: I'm assuming it is the message next to the button that is going to be set to false, but that doesnt make any sense. Maybe the text in New Password disappears. I'm not sure what the dollar sign is use for.
Tester explains what it means that it has no arrow.

Task 3: Implement a way of changing the username in "exampleapp"

TP discussed various complecated methods of implementing the task.
Tester: It can be visible all the time, no problem.

TP: I'll do it the same way as with the password change. I'm thinking there must be a place to input the text.
TP copies the password field. Thinking outloud it is clear that she wants to do it the same way as with password.

Tester: What do you think happens if you remove the last dollar sign part of the field?
TP: Then I don't think the text will appear when "Save" is pressed.
TP leaves the dollar sign part
TP: It's nice to show the username

Tester: What happends when you press "Save"?

TP: The username will be saved

Tester: Is there a connection between the save button and the username field you added?
TP: I want that yes

Tester: Right now you have a connection between password
TP: I'm thinking it also must be one for username.
TP tries to write hasSaved (duplicated) in the box.
Tester: No quite sure why you want to do that?
TP: I'm thinking only parts of the box is performed.
Tester explains that the whole box is performed.

Task 4a: In "exampleapp": Implement a way of storing a textual note by using a TextArea and a Button.

TP drags in a TextArea. Uses a variable named *text* in the TextArea. Makes a TextField with an input field format =${*textField*}
Makes a new button. TP says she is thinking that it needs an action for the button. Makes an arrow from the action to the button.

Tester: If you press "Save", what is *text* then.
TP: Hm
Tester: What is hasSaved? Technically, is it a String, Int?
TP: boolean
Tester: How do you know?
TP: It is depicted in the blue box.
Tester: Then what is *text*?
TP: It should be a String since it is inside the brackets
Tester: Do you remember that password was declared earlier?
TP: Oh, I have to declare *text* in the blue box.

Task 4b: Display the note another place in the window.

TP overcomplecates the task, but when Tester says it can be done easier she changes username with ${text} and it works.

<u>Comments</u>

TP: I felt "green" when it came to the programming. Maybe I would have understood more if I knew some more programming?
Tester: If you had to do it again, would you have understood more?
TP: Yes, definitely.

Tester: What do you think about making prototypes this way?
TP: Very nice and clear. Smart to have different colors on the boxes depending on what they do.

Tester: Anything you would have done differently if you were to make it?
TP: The purple box without an arrow was a little vague. Maybe have all the variables

in one box so they were all visible?

Tester: Did you think it was logical that
    ${username} displays the content of the variable?
TP: Yes

    That =${usernameRandomName} inputs to the identifer?
TP: Yes

    The combo? =${usernameRandomName}{username}
TP: Yes

After seeing what they mean it seems logical.

TP: I would have liked several arrows from one box. Less clutter.

Appendix B
Source code

For practical reasons the source code is available at GitHub. Please refer to the README, and look at the *wireframing-tutorial* project as an example.

Dependencies: **WireframeSketcher, Xtend, JavaFX, Apache Commons IO**

- Create a WireframeSketcher project.
- Add a *Screen*, and a *Storyboard*.
- From the storyboard add the screen.
- Create the sketch inside the screen file. Refer to the examples in *wireframing-tutorial*
- Edit Constants.xtend and ensure the correct directories.
- Run Generator.xtend
- Run AppController.xtend

Source code: `http://github.com/frel/WireframeToJavaFX`