



NTNU – Trondheim
Norwegian University of
Science and Technology

Populating a prototype with external data

Magnus Jerre

Master of Science in Computer Science

Submission date: June 2015

Supervisor: Hallvard Trætteberg, IDI

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

Abstract

Creating user interface (UI) prototypes is an important step when developing new software. Many applications, especially data intensive applications, end up displaying some form of data. A UI prototype for such an application can either display dummy data such as "Lorem ipsum..." or real data, such as the title for a specific movie. Populating the prototype with real data can be done by either manually entering it, or by referencing an external source. Manually entering and modifying data for multiple scenarios can be a tedious and selection biased process. Being able to reference an external data source can speed up the process as well as illustrate how the underlying data suits the designed UI.

This thesis looks into how a prototype can be populated with external data by annotating an existing wireframe prototype and turning it into a runnable program. The developed tool builds on the work of Fredrik Larsen by incorporating functionality for reading and binding data from a persistent XMI file to the UI prototype. It expands on Fredrik Larsen's proposed annotation method by adding three new constructs, namely *context*, *assignment* and *view component*, each having their own area of functionality.

The implemented tool was tested using three scenarios of different complexity to verify whether the proposed data binding method was viable, which the results indicated it was. The *view component* proved to be an especially valuable asset in order to provide modifiability and reuse within the wireframe prototype.

Sammendrag

Utvikling av bruker grensesnitt (UI) prototyper er en viktig del av å utvikle ny programvare. Mange applikasjoner, spesielt data intensive applikasjoner, ender opp med å vise en eller annen form for data. En UI prototype for den slags applikasjoner kan enten vise liksom data som “Lorem ipsum...” eller virkelig data som tittelen på en spesifikk film. En prototype kan populeres med data på to måter, enten ved å manuelt skrive inn dataene, eller ved å referere data fra en ekstern kilde. Å manuelt skrive inn og modifisere data for flere scenarier kan være en både tidkrevende og partisk utvelgelsesprosess. Ved å kunne referere til en ekstern datakilde kan prosessen utføres raskere og illustrere hvordan den underliggende dataen passer til det designede bruker grensesnittet.

Denne oppgaven ser på hvordan en prototype kan populeres med ekstern data ved å annotere en eksisterende trådrammeprototype og transformere det til et kjørbart program. Det utviklede verktøyet bygger på arbeidet til Fredrik Larsen ved å inkludere funksjonalitet for lesing og binding av data fra en persistent XMI-fil til UI prototypen. Programmet utvider Fredrik Larsens foreslåtte annotasjonsmetode ved å legge til tre nye typer, nemlig *context*, *assignment* og *view component*, som har hvert sitt funksjonsområde.

Det implementerte verktøyet ble testet ved hjelp av tre scenarier av varierende kompleksitet for å kunne verifisere om den foreslåtte data-bindingsmetode var levedyktig, noe resultatene tydet på at den var. *View component* viste seg å være en spesielt verdifull ressurs når det gjaldt modifiserbarhet og gjenbruk i trådrammeprototypene.

Preface

This is the master's thesis for the final subject "TDT4900 - Datateknologi, masteroppgave" written in the spring of 2015. The thesis and project was written and conducted by Magnus Jerre with the assistance of supervisor Hallvard Trætteberg.

I would like to thank Hallvard Trætteberg for valuable feedback for both the report and the project itself. His expertise with EMF and Eclipse helped me quickly learn new things both directly related to and not so directly related to the thesis. I would also like to express my gratitude towards Fredrik Larsen who was kind enough to explain to me how the software he had written worked, thus making my life easier when it came to building on it. Finally I would like to thank my family and girlfriend for moral support during this project.



Trondheim, June 14th 2015

Problem Description

The thesis will look into how a software prototype can be populated with data from an external data source. By allowing the software prototype to reference an external data source, possible issues with the planned design in combination with the data can be exposed. An example of such an issue is a name which is too long for the text field it will fill.

In order to populate the prototype with external data an appropriate method for correctly referencing the relevant data will be designed. Once designed it will be implemented and tested in order to validate the design's suitability for referencing the external data from the software prototype.

Assignment Start: January 18th 2015

Assignment Delivery: June 14th 2015

Supervisor: Hallvard Trætteberg

Contents

Abstract	i
Sammendrag	iii
Preface	v
Problem Description	vii
1 Introduction	1
1.1 Motivation	1
1.2 Goals and Research Questions	3
1.3 Approach	3
1.4 Thesis Structure	4
2 Background	7
2.1 What Are Prototoypes	7
2.1.1 Protype Processes	7
2.1.2 Prototype Implementations	7
2.1.3 Prototype Fidelity	8
2.1.4 What Kind of Prototype Does This Thesis Create?	9
2.2 Exisiting Tools	9
2.2.1 Pen and Paper	10
2.2.2 Photoshop	10
2.2.3 Wireframesketcher	11
2.2.4 Fredrik Larsen's Work	12
2.2.5 Axure	12
2.2.6 Programming	13
2.3 Fredrik Larsen's Annotation Method	14
2.4 Model Driven Engineering	14
2.4.1 What are Data Models?	15
2.4.2 Eclipse Modeling Framework - A Modeling Tool	16

2.5	Implementations of Meta Models and Instance Models	17
2.5.1	Relational Database Models	17
2.5.2	Object Oriented Models	18
2.5.3	Document Oriented Models	20
3	Analysis and Requirements	23
3.1	Requirements	23
3.1.1	Binding Data to View Elements	23
3.1.2	Binding Data to Simple Lists	25
3.1.3	Binding Data to Complex Lists	25
3.1.4	Filtering Data	26
3.1.5	Selecting Data	28
3.1.6	Requirements Summary	28
3.2	Proposed Solution	32
3.2.1	Using Data Sources	32
3.2.2	Binding Data to Simple View Elements	33
3.2.3	Creating and Using View Components	34
3.2.4	Binding Data to Lists	35
4	Results	41
4.1	Choices	41
4.1.1	Underlying Technology	41
4.1.2	Navigational Language Chosen	42
4.1.3	Scope - Focus on Data Retrieval	42
4.1.4	Decorators to Use	43
4.1.5	Handling Type	45
4.2	Implementation	45
4.2.1	Implementation Iterations	45
4.2.2	Final Implementation	46
4.3	Testing the Implemented Software	48
4.3.1	Binding Data to Simple View Elements	49
4.3.2	Binding Data to Complex View Elements	52
5	Discussion	63
5.1	Test Results Discussion	63
5.2	Areas of Use	66
5.3	How Can Real Data Enhance a Prototype	66
5.4	Data Model Complexity and Prototype Creation	67
5.5	Does the Method Scale	69
5.6	Expanding The Functionality	69
5.7	New Method for Developing Software	70

5.8 Reusing the generated Code to Create Software	70
6 Conclusion	73
7 Future Work	75
Appendices	77
Using the Prototyping Tool	79

List of Figures

1.1	Prototype without data vs with data	2
1.2	Design and creation reasearch time distribution	4
2.1	Illustration of verticality	8
2.2	An exmample paper prototype	10
2.3	A Wireframesketcher UI prototype using real photos	11
2.4	Fredrik Larsen’s work	13
2.5	Relationship between data model, view model and view	16
2.6	Relational database example	18
2.7	Object oriented model example	19
3.1	Simple view element bindings	24
3.2	Simple list bindings	25
3.3	Complex list bindings	27
3.4	Search bindings	29
3.5	Filter bindings	30
3.6	Annotating data sources	33
3.7	Binding data values directly to view elements	34
3.8	Binding data to complex view elements	36
3.9	Binding data to simple lists	38
3.10	Binding data to complex lists	39
4.1	Context decorators	43
4.2	Assignment decorators	44
4.3	View component decorator	44
4.4	Sample showing the pattern for a generated ecore file	48
4.5	IMDb class diagram	49
4.6	Soundcloud class diagram	50
4.7	Real version and wireframe version	51
4.8	The annotated screen file	52
4.9	Real version and wireframe version	53

4.10	Sample screenshot containing multiple reoccurring elements . . .	54
4.11	Corresponding wireframe design for figure 4.10	55
4.12	The view component annotation definitions	56
4.13	The annotated wireframe prototype excluding the definition of the view components	57
4.14	Providing a group of view elements with the "list" property name	57
4.15	Ecore file for the generated "Top 10" application	58
4.16	JavaFX application for the "Top 10" application	59
4.17	A profile page for a Soundcloud user	60
4.18	The wireframe prototype of the Soundcloud profile page . . .	60
4.19	The annotated screen for the Soundcloud profile page	61
4.20	The view component annotations for the Soundcloud profile page	61
4.21	The generated.ecore file for the annotated Soundcloud proto- types	62
4.22	JavaFX application for the Soundcloud profile page	62
5.1	Poorly designed model	68

Chapter 1

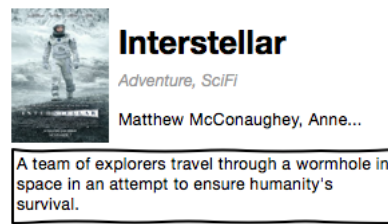
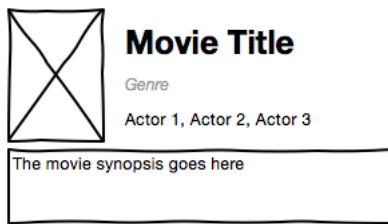
Introduction

1.1 Motivation

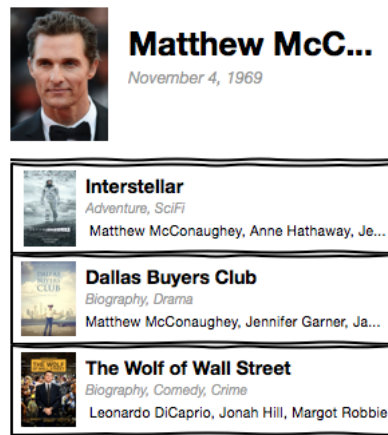
An important part in developing software applications is creating prototypes that illustrate both the intended design and functionality of the final application. Initially, a visual illustration of the application is created by designers later followed by implementation of a subset of the requested features done by software developers.

Many software products end up displaying some kind of stored data. A user interface prototype can either display dummy data such as "Lorem Ipsum" or real data such as the title for a specific movie. Using real data can ensure more constructive user feedback than by simply using dummy data [11]. Figure 1.1 illustrates the difference between using dummy data and real data. The example nicely demonstrates how the fidelity of the prototype is increased by populating it with real data. For projects where the client already has large amounts of data available, being able to use their data might help improve the prototype's usefulness.

Populating a prototype with data can be done in two ways, either manually inputting the data, or referencing an external data source. Manually inputting large amounts of data is time consuming and can result in selection bias by only selecting data that fit nicely with the prototype, an example of this would be only selecting movies with short titles. This way missing edge cases, such as a string of text that is too long, and how they should be handled can quickly happen. Enabling the software developers who will take over the prototype development to quickly reference an external data source, can help overcome these issues.



(a) A movie prototype using dummy data (b) A movie prototype using real data



(c) An actor prototype using dummy data (d) An actor prototype using real data

Figure 1.1

1.2 Goals and Research Questions

Goal Explore how a wireframe prototype can be populated with real data from an external source by providing the prototype with a little extra information.

Research question 1 What kind of input must be provided in order to specify which data goes where?

Research question 2 How can real data enhance a prototype?

1.3 Approach

This thesis is based on doing "Design and Creation" research for information systems as defined in [6]. When doing design and creation research some IT related artefacts are created, in this case the result is what's called an Instantiation artefact, a working system that demonstrates the method for populating a prototype with data. The system will be a 'proof of concept' prototype rather than a finished product for use by the public.

The design and creation process is an iterative one, consisting of five steps: awareness, suggestion, development, evaluation and conclusion. The idea is that by iteratively cycling through each step one will gain a better understanding for the next cycle.

Awareness is the recognition and definition of a problem.

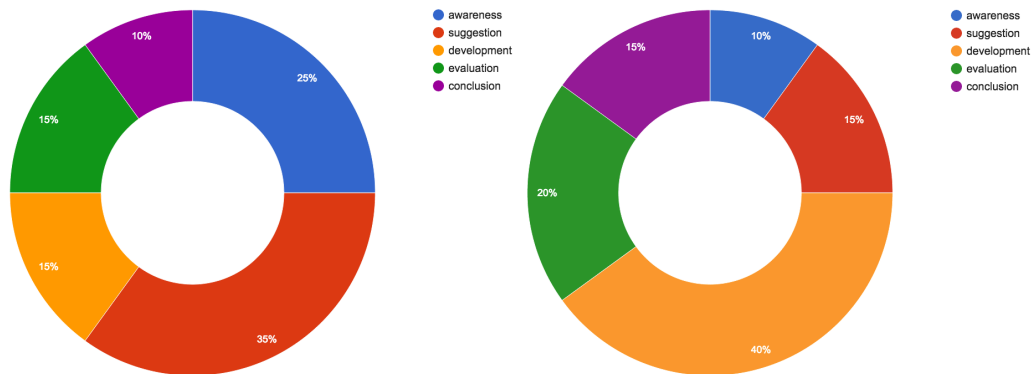
Suggestion is the process of moving from recognizing the problem to proposing possible solutions to them.

Development is the process of actually developing the system proposed in the suggestion process.

Evaluation is the process of examining and assessing the developed artefact.

Conclusion is the process of summing up the results from the entire process.

The research process will however not strictly follow the design and creation cycle at all times. In the starting phase, more time and effort will be spent on thoroughly understanding the problem at hand as well as finding ways to solve the problems. Some effort will go into developing a provisional simple version of a subset of the system in order to better understand how



(a) Chart illustrating the relative amount of time early on in the project (b) Chart illustrating the relative amount of time later on in the project

Figure 1.2

everything should be stitched together. This way, some alternatives that might prove unsuitable during implementation can be discovered earlier on.

In the later phases, more of the focus will be shifted towards actually developing the application as well as evaluating the product rather than the awareness and suggestion steps. This is natural as in the later phases, some of the suggestions will have been evaluated as either suitable or not.

Figures 1.2a and 1.2b illustrate how time will be spent in each stage relatively in the early and later phases respectively.

1.4 Thesis Structure

This master's thesis is split into a total of seven chapters, the first being the introductory chapter providing information regarding the motivation behind the thesis as well as the research methodology used.

Chapter 2 - Background covers the ground the reader should be familiar with before reading the rest of the thesis. It provides information regarding prototypes, and data models, as well as information regarding the work of Fredrik Larsen which this thesis builds on.

Chapter 3 - Analysis and Requirements presents examples of how existing applications display data and elicits requirements for the application to be based on the examples. It also proposes solutions on how wireframe prototypes can be annotated in order to populate it with data.

Chapter 4 - Results presents the choices that have been made regarding implementation and scope, as well as information on how the application is implemented and tested in order to be able to evaluate it.

Chapter 5 - Discussion provides answers to the research questions, evaluates the test results, discusses how well the chosen method will fare against models and prototypes of different complexities and the application's potential.

Chapter 6 - Conclusion summarises the results and implications thereof.

Chapter 7 - Future Work provides insight into what kind of features that can or should be implemented at a later time.

Chapter 2

Background

2.1 What Are Prototypes

This section introduces the concept of prototypes, their goals as well as defining the kind of prototype this thesis will be.

A UI prototype illustrates the intended design and functionality, and helps capture requirements for a system to be. Depending on the kind of study, they can even be used to determine whether a software system should be developed at all [5]. In short, prototypes are used as a means to acquire knowledge related to the development of software.

2.1.1 Prototype Processes

Various prototypes are used with different processes in mind. These processes can be separated into exploratory prototyping, experimental prototyping and evolutionary prototyping [5]. *Exploratory prototyping* is used to examine and understand the problem at hand often through the development of several design options. *Experimental prototyping* is the process of creating provisional versions of the product in order to gain more knowledge for the next prototype iteration. *Evolutionary prototyping* on the other hand is performed by iteratively working on and improving an existing prototype. An evolutionary prototype can end up being the final product if it meets all the requirements that the finished product should.

2.1.2 Prototype Implementations

The implementation of prototypes distinguishes between two classifications, namely horizontal and vertical prototyping [5].

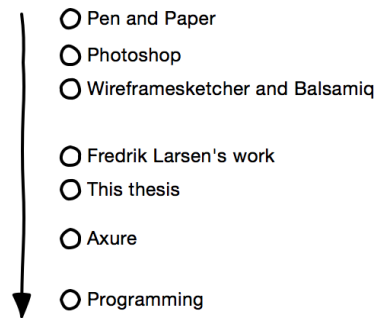


Figure 2.1: Illustration of verticality

Horizontal prototypes focus on building a specific layer, for instance the user interface (UI) layer or functional layer such as database transactions. A horizontal UI prototype focuses on covering a wide range of the necessary UIs for the application to be.

Vertical prototypes focus on building the necessary layers of a specific part of the software to be, for instance implementing both the user interface as well as database transactions to get a functional prototype.

By implementing some of the horizontal elements of the prototype vertically, functionality for parts of the system can be made complete. Implementing all of the horizontal elements vertically can result in a complete version of the final application. Figure 2.1 illustrates how a handful of prototyping tools relate to each other when it comes to verticality. The top of the stack focuses only on a single layer of the development process, namely the UI. Further down the stack, more layers are used, such as linking and data source connection. At the bottom, a fully developed part of the system is done using the actual programming framework for the final product.

2.1.3 Prototype Fidelity

When working with UI prototypes it's normal to create prototypes with different levels of fidelity. A prototype's level of fidelity is judged according to how complete a user perceives it to be rather than its similarity to the actual application. The fidelity levels is often separate into two, namely low-fidelity prototypes and high-fidelity prototypes [9].

Low-fidelity prototypes are simple drafts of the intended layout for the software to be, providing little to no user interaction. The goal with low fidelity prototypes is to establish a logical process flow and elicit knowledge

regarding requirements rather than defining the exact look of the software. Demonstration of the workflow is often done by someone skilled at operating the prototype rather than having the users interact with it, in addition the demonstration is carefully scripted in order to convey a story.

High-fidelity prototypes are in contrast to low-fidelity prototypes fully interactive and faithfully represent the software to be. The core functionality of the UI is represented in high-fidelity prototypes, they may however not be implemented fully vertically instead providing predefined results for some of the prototype's features.

2.1.4 What Kind of Prototype Does This Thesis Create?

This thesis builds on the work of Fredrik Larsen by adding a new layer of functionality, namely data binding. Fredrik Larsen's work is an extension to Wireframesketcher and resulted in an application that is itself a prototype. By adding a new layer of functionality to the tool it's brought one step closer to being an actual product rather than just a prototype. Repeating this process of expanding the prototype's functionality can, in the end, result in a finished product. The prototype process used can therefore be seen as an evolutionary one.

As illustrated in figure 2.1, Fredrik Larsen's work resulted in a tool that goes a step further than Wireframesketcher by adding a layer of programming to it. As stated this thesis builds on his work, since a new layer of functionality is added to the tool, namely data binding, it will naturally be a little deeper vertically.

The fidelity of a prototype is largely determined by the tool used to create the prototype. Some tools only allow crude illustrations, close to drawings, others provide functionality for creating a look that can exactly represent the final design. Wireframesketcher lands somewhere in the middle of simple drawings and sketches and the final design. However, the generation and use of JavaFX moves the fidelity from medium-low to high.

2.2 Existing Tools

This section describes different tools that can be used to create prototypes as well as their ability to display real data. The fidelity of the tools range from low to high, starting with the lowest fidelity tool, then moving on to the higher fidelity tools.

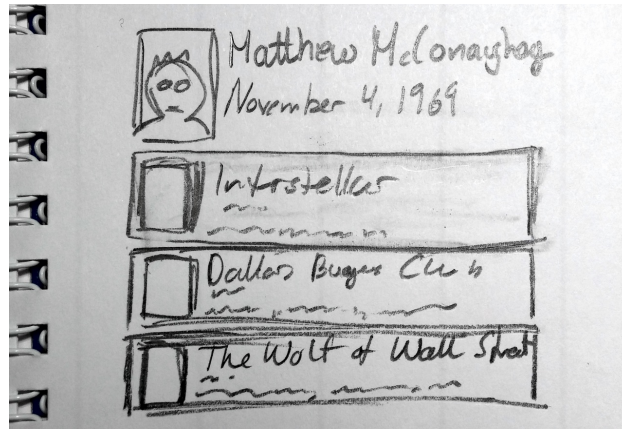


Figure 2.2: An example paper prototype

2.2.1 Pen and Paper

Creating UI prototypes using pen and paper is a fast, simple and cheap prototyping method [9]. The result is typically low fidelity and static prototypes such as the one illustrated in figure 2.2. These are mostly used to convey the intended overall layout for the application to be. Testing using paper prototypes is often done by placing the relevant pieces of paper representing the current state of the application in front of the tester. In order to move from one state to another, the tester notifies the facilitator of the intended action, the facilitator then replaces the existing piece of paper with new pieces of paper representing the new state. The test scenario is typically very scripted, making the facilitator's management job easier. This is by all means a manual process and is usually done at the early stages of prototype development.

Prototyping using pen and paper is a manual and non-digital process and therefore provides no mechanism for binding data from an external source to the prototypes, this process must therefore be manually handled.

2.2.2 Photoshop

Stepping up from the analog world of pen and paper and moving on to the digital world of computers it's possible to create UI prototypes of higher fidelity using drawing or photo editing tools such as Photoshop¹. Photoshop allows the user to create all the UI components digitally making the prototype to look a lot better than with sketches using pen and paper. These kinds of tools do however not provide any additional form of functionality such as

¹Photoshop can be found at <http://www.adobe.com/no/products/photoshop.html>

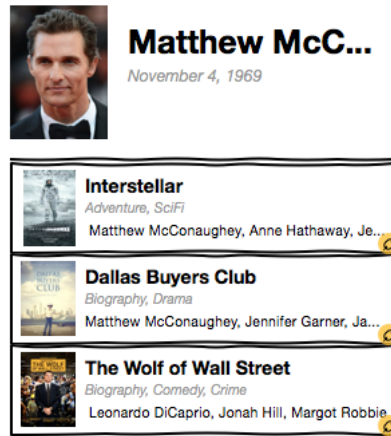


Figure 2.3: A Wireframesketcher UI prototype using real photos

point and click to move on to the next state. Performing test scenarios can be done by using printouts in the same manner as with pen and paper.

Photoshop is designed to edit photos, not to create prototypes with advanced functionality. Naturally it provides no functionality for binding external data to its view elements other than manually entering it.

2.2.3 Wireframesketcher

Wireframesketcher is a piece of software designed for creating so called Wireframe prototypes for UIs². A wireframe prototype is an illustration of the design, much like creating a prototype using pen and paper, rather than the exact design solution. Wireframesketcher is built around using pre-fabricated template models to boost productivity and therefore supports a various set of view elements, such as windows, web browsers, buttons and so on. It does however support definition of custom widgets and allows for real photos to be used with the prototype. Figure 2.3 illustrates a prototype created using Wireframesketcher.

Each state the prototype should be able to be in is modeled inside separate screen files. By creating links from one screen to another, it's possible to dynamically navigate between states, making test scenarios feel more like a real workflow than testing using printouts. Figure 2.3 illustrates how three links are represented using yellow circles. It should be noted that each state is static, meaning that the only form of user input allowed is clicking on links in order to move to a different state. There are no features to support text input or to add additional functionality to the wireframe prototype.

²Wireframesketcher can be found at <http://www.wireframesketcher.com/>

Due to Wireframesketcher's static nature there is no way to bind data to the view elements other than manually entering it.

2.2.4 Fredrik Larsen's Work

Fredrik Larsen's work [4] builds on Wireframesketcher by adding more functionality to the prototype through generating a runnable JavaFX application based on a wireframe model. The JavaFX application enables user input handling making the prototype much more functional than a simple Wireframesketcher prototype. For the generated application to understand how it should react to user input, the developers must annotate the different view elements using Wireframesketcher. Features such as hiding text or displaying text input from the user can be achieved using a combination of decorator elements.

The decorator elements needed to annotate the wireframe model correctly are separated into three areas of responsibility.

The Data Decorator which is colored blue, is responsible for defining the different variables that can be used by the runnable JavaFX application.

The Style Decorator which is colored thistle purple, is responsible for defining how the view elements should be displayed, such as changing or hiding text displayed based on the state of a data variable.

The Action Decorator which is colored red, is responsible for defining what should happen when actions such as pressing buttons occur.

Figure 2.4a illustrates a Wireframesketcher screen file annotated in order to define the functionality for the JavaFX application in figure 2.4b.

As with basic Wireframesketcher, no method for binding data from an external source to the prototype exists other than manually entering the necessary data.

2.2.5 Axure

Axure is another tool that can greatly increase the fidelity of the prototypes created with it³. Fredrik Larsen's work provides functionality for simple interaction with the JavaFX prototype through variables, Axure takes this interaction a step further and provides additional features such as describing the timing of interaction events. The two products differ however in terms of

³Axure can be found at <http://www.axure.com/>

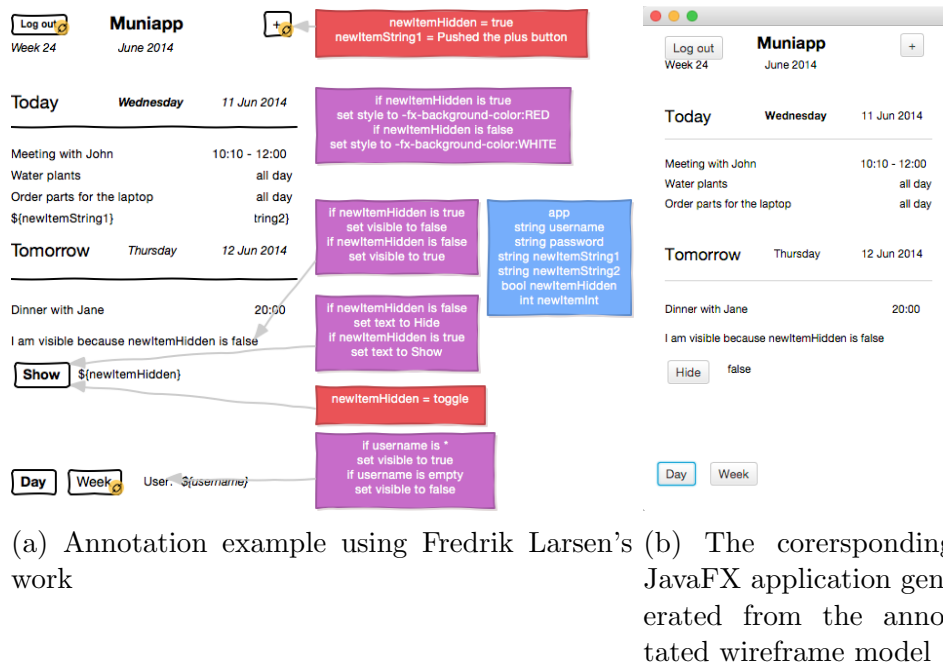


Figure 2.4

usage area and the results they produce. While Axure is focused on allowing the designers to build the prototype from the ground up, Fredrik Larsen's work is based on being able to annotate an existing wireframe model and turning it into a runnable program. The output between the two also differ in that Axure produces HTML and JavaScript code for demonstration using web browsers, Fredrik Larsen's work on the other hand produces ecore models, fxml and java code which can be used as a basis for later implementation.

Axure does provide the use of data stored using Excel by copying the data cells into Axure. However, data provided in other formats, such as relational databases or XML-documents is not supported. Using large amounts of complex data with Axure doesn't seem to be a viable option.

2.2.6 Programming

There is one obvious way of creating prototypes not yet discussed, that is through programming. There exists several tools that allow developers to create UIs using drag and drop, making the process of building the UI pretty fast and separate from coding⁴. The tools then generate the underlying

⁴A list of graphical UI builders can be found here https://en.wikipedia.org/wiki/Graphical_user_interface_builder

files necessary for the view which must then be complemented with manual programming in order to provide any sort of interaction features. Android applications can be developed using such a process. A tool for designing the UI using drag and drop is first used to create the corresponding XML-files describing the view. After the view is created the developers must manually code the functionality it should provide.

Creating high fidelity prototypes through programming comes at the cost of being a lot more time consuming than creating lower fidelity prototypes using say Wireframesketcher. It should be noted that when creating prototypes this way a lot of the finer functionalities can be replaced by dummy functions that simply produce predefined results for the test scenarios.

There are a couple of advantages in creating prototypes using programming. One advantage with programming the prototypes is that the code can be reused for the final implementation, another is the fact that the programming framework's whole toolset is available. This means that the developers will have complete control over how to access and format the data needed by the prototype. This control however comes at the cost of possibly unnecessarily high complexity and quite long development times.

2.3 Fredrik Larsen's Annotation Method

This section introduces the annotation method Fredrik Larsen proposed in his master's thesis.

During his work on the master's thesis, Fredrik Larsen came to the conclusion that an appropriate method for annotating the different view elements is by using annotation boxes, called decorators, and arrows to decorate each view element rather than providing a separate annotation view. By annotating the view elements this way it's possible to annotate view elements that themselves don't provide any method for further description.

Since this thesis will build on Fredrik Larsen's work, and therefore his software, it's natural to continue with his decorator method for annotating the view elements. His annotation method provides a solid foundation for further expanding the functionality provided through the use of decorators.

2.4 Model Driven Engineering

Model driven engineering (MDE) is a software development methodology based on defining models for a particular problem domain to solve the task at hand [10]. It raises the level of abstraction away from language specifics such

as Java and C# to domain specific languages (DSL). The models created as part of MDE are used as the basis for later automated model code generation. An example of a software package based on MDE is the Eclipse Modeling Framework (EMF) which will be discussed later.

2.4.1 What are Data Models?

This section delves into the concept of data models, what they are, what types of models exist and different model implementations.

A data model is a meaningful representation of a concept or real world entity, based on structuring data elements and their relations [15]. Take for instance a person. A person has a name, a birthdate, a height, eye color, hair color and so on. All of these features are attributes that describe a person. In order to represent the concept of a person, one would define a person model containing the relevant features that help describe a specific person. A very simple person model can for instance only contain the name of the person, whereas a more complex person model might contain the person's name and the person's parents. The latter example would be a model having properties that themselves are represented with the same model.

Meta Model and Instance Model

The simple definition of models can be made more complex by separating the model into a meta model and an instance model. The two concepts are tightly connected which will be illustrated shortly.

The meta model describes what kind of information is necessary and how it relates to other information in order to represent some concept. Continuing with the person example, the meta model is the definition of what information is needed to represent a person. The person meta model can for instance require information about the person's name, birthdate, and parents. It does however not contain any information related to a particular person.

The instance model describes a particular instance that satisfies the rules given by its meta model. Using the person meta model from the previous paragraph, an instance of the meta model can be a person named "Matthew McConaughey" with birthdate "November 5, 1968" and parents "James" and "Mary".

Data Model vs View Model

When working with applications that present data in some way, it's often the case that the information presented doesn't exactly match the structure of

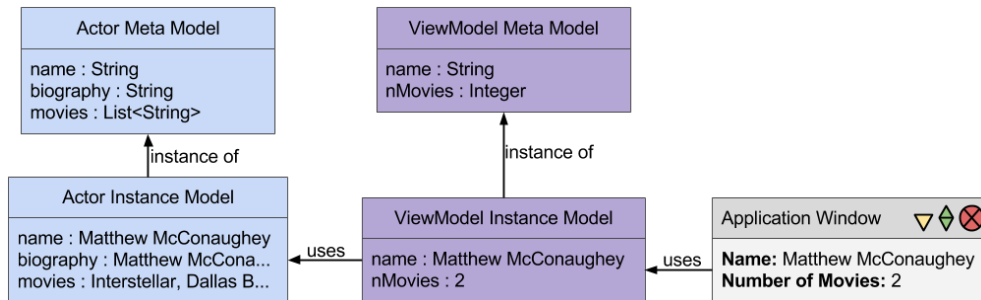


Figure 2.5: Illustration of the relationship between a data model, a view model and a view

the underlying data model. Figure 2.5 is an example of such a mismatch. The application window on the right hand side displays the name of an actor and the number of movies the actor has starred in. The yellow boxes represent the underlying data model, in this case an actor is associated with a name, a biography and a list of movies. The application however will only display the name of the actor (which exists directly in the underlying data model) and the number of movies he has starred in (which must be inferred from the movies property for the actor). The purple boxes represent the view model for the application and only contains data that is relevant for the view.

Utilising view models is a way to simplify the view's implementation. Since the view model contains all the data that the view needs, all the view has to do is to display this data. Figure 2.5 illustrates this nicely by displaying the number of movies the actor has starred in. Since the view model contains this value directly, the view won't have to calculate the number of movies, only display the value provided by the view model instance. The view model can thus contain data that doesn't exist or explicitly exist in the underlying data model.

2.4.2 Eclipse Modeling Framework - A Modeling Tool

The Eclipse Modeling Framework (EMF) is Eclipse's implementation of a model driven engineering tool [1]. EMF distinguishes between the meta model called ecore, and the instance model which can be persisted as an XMI file or used as a runtime instance. The ecore meta model is later used as part of a process to generate the necessary code for the domain model. The model can be edited at any time and code regenerated to facilitate changes in the specifications.

EMF uses XMI files to persist instances for the ecore meta model. These XMI files are read by EMF during runtime and turned into instances in the

object oriented format, allowing programming statements to be executed on them.

EMF provides built in support not only for its native ecore and XMI, but also XSD and XML which is a popular format for storing models and meta models. Tools for converting other types of meta models and instance models into XSD and XML exist, making EMF usable for several types of models.

2.5 Implementations of Meta Models and Instance Models

The following section will look into some implementations of the meta model and instance model pair and how to query such models using query languages.

2.5.1 Relational Database Models

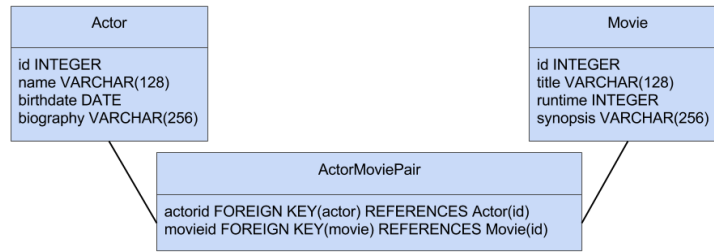
A relational database is a structured collection of data persistently stored using tables [8]. Each table usually represents some model or concept. In relational databases, tables can have references to each other in order to reuse data instead of storing several copies.

The relational database schema is the meta model for a relational database containing the definition of the different tables and how they relate to each other. Figure 2.6a illustrates a schema for a simple movie database. In order to have a many to many relation between entries in the Actor and Movie table a third table, called ActorMoviePair, is created.

As stated above, the instance data for a relational database is stored in tables as illustrated in figure 2.6b. By performing what is called a join, it's possible to retrieve the correct combination of data when the desired data is spread out over multiple tables.

Structured Query Language

The query language used for querying into relational databases is Structured Query Language (SQL). It enables the users to insert, query, update and delete data from the database using queries [8]. In order to do something with any kind of data, an operation must be specified in addition to where the operation should be performed. The following query returns a table containing only the titles for all the movies that the actor 'Matthew McConaughey' has starred in. The simple example gets quite complex since two joins are needed in order to get the desired output.



(a) Meta model represented using a relational database schema

Actor			
id	name	birthdate	biography
1	Matthew McConaughey	1969-11-04	Matthew Mcconaughey was bo...
2	Anne Hathaway	1982-11-12	Anne Hathaway was born in Br...

Movie			
id	title	runtime	synopsis
3	Interstellar	169	A team of explorers travel throu...
4	Dallas Buyers Club	117	In 1985 Dallas, electrician and h...

ActorMoviePair	
actorid	movieid
1	3
1	4
2	3

(b) Instance model for the meta model

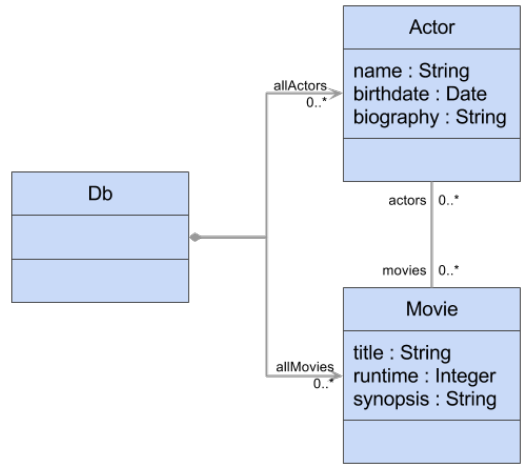
Figure 2.6

```

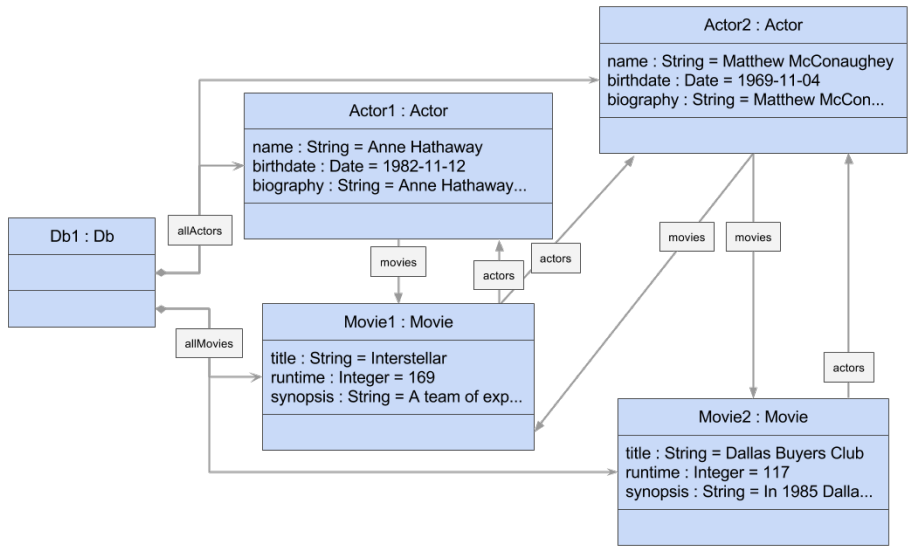
SELECT title FROM Movies
JOIN ActorMoviePair ON ActorMoviePair.movieid=Movies.id
JOIN Actors ON Actors.id=ActorMoviePair.actorid AND
Actors.name='Matthew McConaughey';
  
```

2.5.2 Object Oriented Models

Object oriented modeling is based on representing data using concepts and entities in the form of objects. The instance models are typically only part of a runtime instance, and therefore only exist in-memory, not as a persistent entity. Figure 2.7a illustrates how the concepts of movie and actor can be modeled as objects using the UML notation. In contrast to relational databases, there is no need for a separate entity containing relations between a pair of movie and actor, instead, direct links between the two entities are used. Figure 2.7b represents the runtime instance model for the corresponding meta model. A runtime instance model can be serialised into various document formats in order to persist or interchange data.



(a) Meta model representation for objects



(b) Instance model representation for objects

Figure 2.7

Java and other programming languages

If the data model is available during runtime as in-memory objects, a natural way of navigating it is using an object oriented language, such as Java. With Java 8, Lambdas have been introduced making filtering a lot simpler than before [2]. The following statement retrieves the movie titles that "Matthew McConaughey" has starred in.

```
List<String> titles = new ArrayList<String>();
db.allActors.stream()
    .filter(a -> a.name.equals("Matthew McConaughey"))
    .findFirst().get().movies.forEach(m -> titles.add(m.title));
```

Other languages, such as JavaScript also provide methods for filtering lists. The obvious power in using a programming language like Java is that it provides a powerful set of features, this power however comes at the cost of verbosity.

Object Constraint Language

The Object Constraint Language (OCL), was not designed with the intention to be a query language, but rather a language for describing rules and constraints that apply to Meta Object Facilities and validating them [7]. However, it supports functionality for retrieving specific objects and information regarding their type. As the name implies, it's based on describing objects and is therefore quite similar to object oriented languages, such as Java, when it comes to querying. The following statement retrieves the movie titles that "Matthew McConaughey" has starred in.

```
allActors->select(name = 'Matthew McConaughey')->at(1).movies.title
```

EMF provides an implementation of OCL that works in two ways, one is to use OCL statements directly on the XMI document model inside Eclipse, the other is to use it as a parser for use with Java code. Since it's split into these two different modules, it's possible to perform the queries directly on the XMI model and quickly verify their correctness, then use the queries inside the prototype.

2.5.3 Document Oriented Models

Document oriented models store models as persistent documents, such as XML or JSON. Documents such as XML follow a strict structure, and are human readable [12]. Storing data in documents allows for the data to be easily interchanged with several systems. The data can either be created

directly inside the document or a runtime model can be serialised into it. Various document formats exists, among them are XSD an XML, JSON-Schema and JSON, and ECORE and XMI. Below, an example using XSD and XML will be shown. The XSD [14] document describes the meta model for the corresponding XML document containing an instance model. A XML document is stored as a tree structure with a single root node being the entry point for the document.

Listing 2.1: The XSD schema for the meta model

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
  <xs:element name="db">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="allMovies" type="movieListTypeContainment"/>
        <xs:element name="allActors" type="actorListTypeContainment"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="movieListTypeContainment">
    <xs:sequence>
      <xs:element name="movie" type="movietype" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="actorListTypeContainment">
    <xs:sequence>
      <xs:element name="actor" type="actortype" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="actortype">
    <xs:sequence>
      <xs:element name="actorID" type="xs:ID"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="birthdate" type="xs:date"/>
      <xs:element name="biography" type="xs:string"/>
      <xs:element name="movies" type="movieListTypeRef"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="movietype">
    <xs:sequence>
      <xs:element name="movieID" type="xs:ID"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="runtime" type="xs:integer"/>
      <xs:element name="synopsis" type="xs:string"/>
      <xs:element name="actors" type="actorListTypeRef"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="movieListTypeRef">
    <xs:sequence>
      <xs:element name="movieIDREF" type="xs:IDREF" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="actorListTypeRef">
    <xs:sequence>
      <xs:element name="actorIDREF" type="xs:IDREF" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing 2.2: The corresponding XML instance model

```

<?xml version="1.0"?>
<db xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="db.xsd">
  <allMovies>
    <movie>
      <movieID>m1</movieID>
      <title>Interstellar</title>
      <runtime>169</runtime>
      <synopsis>A team of explorers travel throu...</synopsis>
      <actors>
        <actorIDREF>a1</actorIDREF>
        <actorIDREF>a2</actorIDREF>
      </actors>
    </movie>
    <movie>
      <movieID>m2</movieID>
      <title>Dallas Buyers Club</title>
      <runtime>117</runtime>
      <synopsis>In 1985 Dallas, electrician and h...</synopsis>
      <actors>
        <actorIDREF>a1</actorIDREF>
      </actors>
    </movie>
  </allMovies>
  <allActors>
    <actor>
      <actorID>a1</actorID>
      <name>Matthew McConaughey</name>
      <birthdate>1969-11-04</birthdate>
      <biography>Matthew McConaughey was born...</biography>
      <movies>
        <movieIDREF>m1</movieIDREF>
        <movieIDREF>m2</movieIDREF>
      </movies>
    </actor>
    <actor>
      <actorID>a2</actorID>
      <name>Anne Hathaway</name>
      <birthdate>1982-11-12</birthdate>
      <biography>Anne Hathaway was born in...</biography>
      <movies>
        <movieIDREF>m1</movieIDREF>
      </movies>
    </actor>
  </allActors>
</db>

```

XML Path Language

XML Path Language (XPath) [13] is a query language aimed at navigating a XML document. Since the XML documents are structured as sorted trees navigation is done from a root down to the different child elements. The statements however do not need to start at the root as it's possible to select nodes based on their type rather than their relationships to each other. The following statement retrieves the movie titles that "Matthew McConaughey" has starred in. In order to retrieve the actor nodes for a specific movie it's necessary to perform manual checks against each actor, making the retrieval of linked nodes cumbersome.

```

/db/allMovies/movie
  [(actors/actorIDREF)=(//actor[name='Matthew McConaughey']/actorID)]
  /title

```

Chapter 3

Analysis and Requirements

3.1 Requirements

By looking at existing applications and websites based on displaying data in different ways, it will be easier to elicit the prototype tool's requirements. The following sub sections will shed some light on what view elements are bound to data and how they relate.

3.1.1 Binding Data to View Elements

In order to populate a prototype with data, it will be necessary to provide a way of binding the external data to each view element responsible for displaying said data. Figure 3.1a is a screenshot taken from IMDb.¹ It displays different data for the movie "Interstellar", such as its poster, title and synopsis. Figure 3.1b illustrates which of the view elements must be bound to external data by enveloping them with a blue-dashed box.

From this example it's clear that a way of binding data to specific view elements is necessary, otherwise nothing of interest would be displayed inside each of the blue-dashed boxes. Figure 3.1a is just a small cut-out from the original screenshot, many more elements are actually bound to data. An efficient way to bind each view element to data will therefore be necessary.

Binding external data to view elements requires the prototype to know where the data comes from. Being able to specify the location of the data source is therefore necessary.

¹IMDb is an acronym for International Movie Database and can be found at www.imdb.com



Interstellar (2014) 43

11 | 169 min | Adventure, Sci-Fi | 7 November 2014 (Norway)

Your rating: ★★★★★★★★ -/10
8.7 Ratings: **8.7/10** from **631,961** users Metascore: **74/100**
 Reviews: **2,327** user | **656** critic | **46** from Metacritic.com

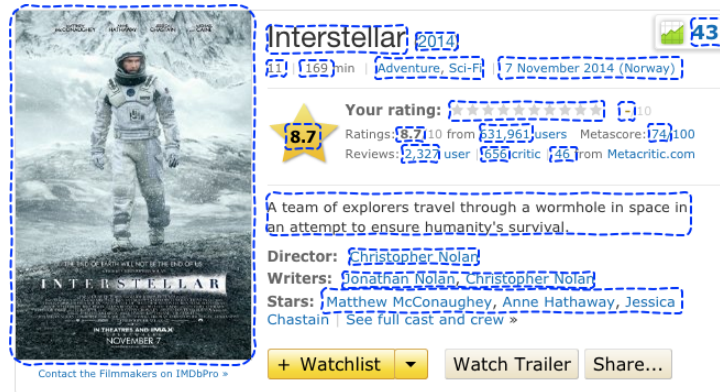
A team of explorers travel through a wormhole in space in an attempt to ensure humanity's survival.

Director: Christopher Nolan
Writers: Jonathan Nolan, Christopher Nolan
Stars: Matthew McConaughey, Anne Hathaway, Jessica Chastain | See full cast and crew »

[+ Watchlist](#) [Watch Trailer](#) [Share...](#)

Contact the Filmmakers on IMDbPro »

(a) Detailed information about a specific actor



Interstellar (2014) 43

11 | 169 min | Adventure, Sci-Fi | 7 November 2014 (Norway)

Your rating: ★★★★★★★★ -/10
8.7 Ratings: **8.7/10** from **631,961** users Metascore: **74/100**
 Reviews: **2,327** user | **656** critic | **46** from Metacritic.com

A team of explorers travel through a wormhole in space in an attempt to ensure humanity's survival.

Director: Christopher Nolan
Writers: Jonathan Nolan, Christopher Nolan
Stars: Matthew McConaughey, Anne Hathaway, Jessica Chastain | See full cast and crew »

[+ Watchlist](#) [Watch Trailer](#) [Share...](#)

Contact the Filmmakers on IMDbPro »

(b) Highlighting the different view elements that must be binded to data

Figure 3.1

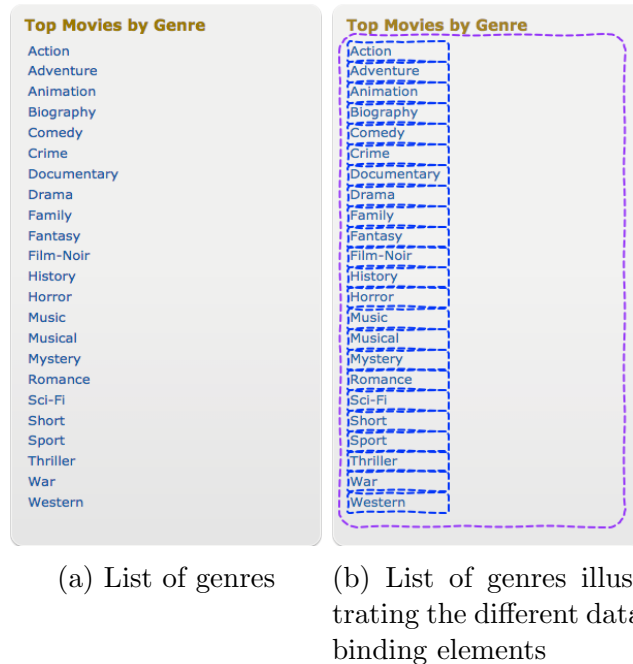


Figure 3.2

3.1.2 Binding Data to Simple Lists

The previous section described the need to be able to bind the data to each of the view elements on-screen. The following section looks into binding data to simple lists. Figure 3.2a shows a list of popular movie genres, taken from IMDb. Each element is simple, meaning that each element in the list only displays a single value, in this case a string. Figure 3.2b illustrates the bindings. As in the previous section, each blue-dashed box represents a single view element bound to a single data value. The purple-dashed box illustrates that the entire collection of view elements is a list.

Utilising the fact that the data is a collection of values can help save time assigning the values to the list. Manually binding each view element to their own data values can be a tedious task, especially if we are dealing with a large collection. Binding the list view element to the entire collection and let the prototype automatically bind each view element to the collection element will make the process less tedious.

3.1.3 Binding Data to Complex Lists

This section will further expand on the examples from the two previous sections by looking at complex lists. Complex lists are lists where each list

element will display more than a single data value, as illustrated in figure 3.3a.

Each of the lists are marked and numbered in figure 3.3b using purple boxes. The figure displays a total of six lists, four of which display their bounded data in unique ways. Lists 2 through 4 display their data in the same manner.

Even though lists 2 through 4 and list 6 have the same kind of input, namely a collection of tracks, their visual output is different. The same is true for list 1 and 5, their input is a collection of playlists. Lists 2 through 4 display a track art, track number, name of the user publicising the track, track name and the number of playbacks all in one line. List 6 however spreads this information out on several lines and includes additional information such as the number of likes, reposts and comments for the specific track.

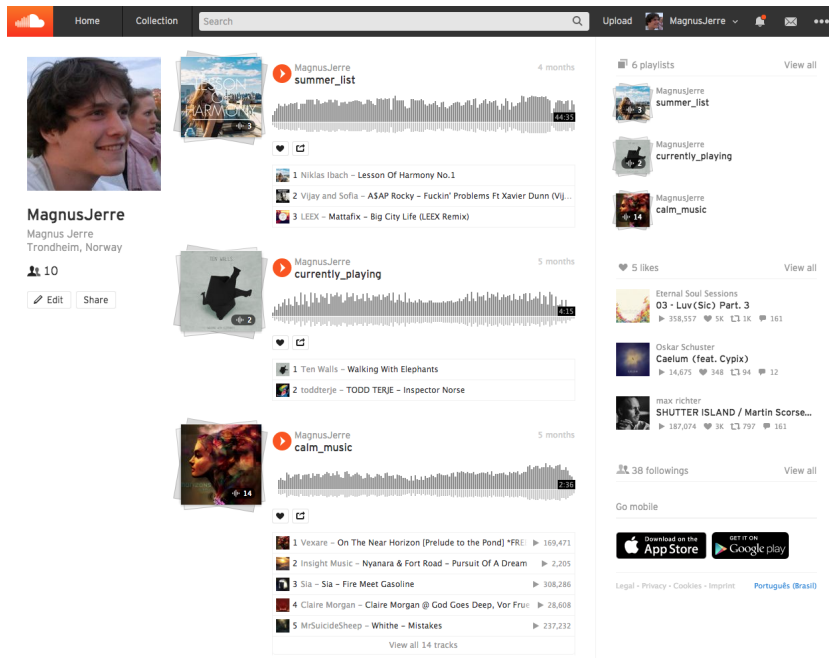
This provides a clear indication there is not only one single way of displaying the data for a given input.

Another thing to note are the orange-dashed boxes inside each of the lists. After the first iteration of implementation it became clear that repeating the data binding step for the same kind of group layout was inefficient. The need to reuse reoccurring groups of view elements was therefore deemed necessary. These reoccurring groups of view elements are marked with orange-dashed boxes in figure 3.3b and are called view components. The blue-dashed boxes inside the view components still represent a single piece of data bound to a single view element. Borrowing from object-oriented programming, each orange-dashed box can represent a class, or meta model, of how the list should treat its input data. These view components need not be limited to lists only, thereby maximizing their reuse. Both the information regarding the profile as well as the information regarding the currently playing song could potentially be seen as view components that should be reuseable in other parts of the application.

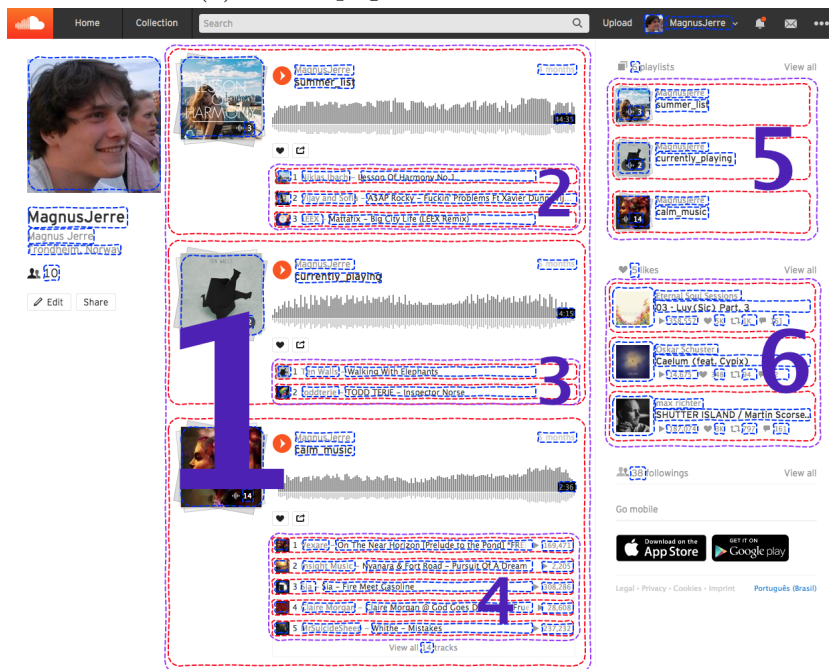
3.1.4 Filtering Data

Another popular website feature is being able to search or filter the data displayed. Figure 3.4a shows the search function available on IMDb. Searching for the movie "Interstellar" results in the dropdown list containing only elements relevant for the search query "Interstellar". Figure 3.5b shows a different method for filtering the data. Instead of allowing the user to input text, the "search criteria" is predefined.

Other than their way of defining the search filter, the two examples also behave differently in that IMDb only applies one filter, the search filter, while



(a) Profile page on soundcloud.com



(b) Highlighting the different view elements that must be bound to data. Also illustrates the need for lists and reusing view elements.

Figure 3.3

the Amazon² example allows several filters to be used at the same time, thus further refining the filtering results. Figure 3.5b shows three different lists of filters. Each filtering list allows several filters to be applied at the same time to either constrain or widen the result, depending on how the filtering is performed.

Looking at figure 3.5a, there are three filter categories, namely "International Shipping", "Operating System" and "Monitor Display Size". If only one filter from each category is applied, then performing filtering iteratively works well since the results should satisfy all the constraints. However, if more than one filter from a certain category is to be applied, this will not work because not all of the filters selected are supposed to be satisfied at the same time. For instance, filtering for "Mac OS X" and "Linux" is supposed to result in a collection containing elements that satisfy either "Mac OS X" or "Linux" or both. Applying each filter iteratively however will result in a collection of elements that must satisfy both "Mac OS X" and "Linux".

For filtering to be performed correctly by the prototype, it must therefore be possible to somehow specify which filter combinations will widen the result and which will constrain them.

3.1.5 Selecting Data

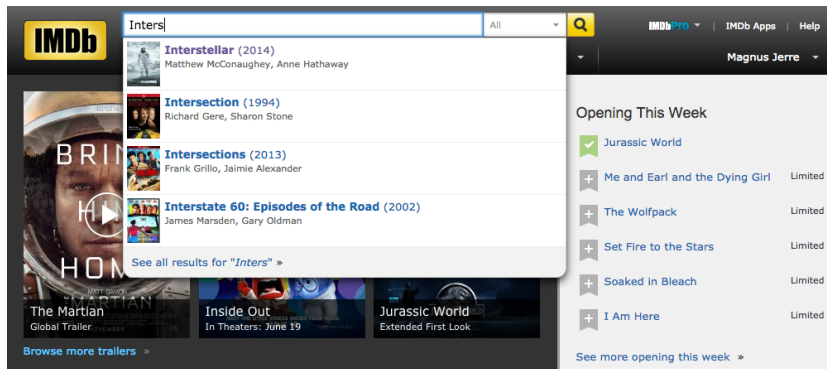
When data is presented in a list, it will often only display a sort of summary for each list element. Figures 3.3a and 3.4a illustrate this. In order to see the detailed information for a specific element it must be selected. Selecting the movie "Interstellar" from the search drop down list in figure 3.4a will send the user to a different page containing more information regarding that particular movie, such as the information displayed in figure 3.1a.

In order to implement this kind of functionality, it's necessary to allow the developers to somehow specify how or where view elements should look for the selected data.

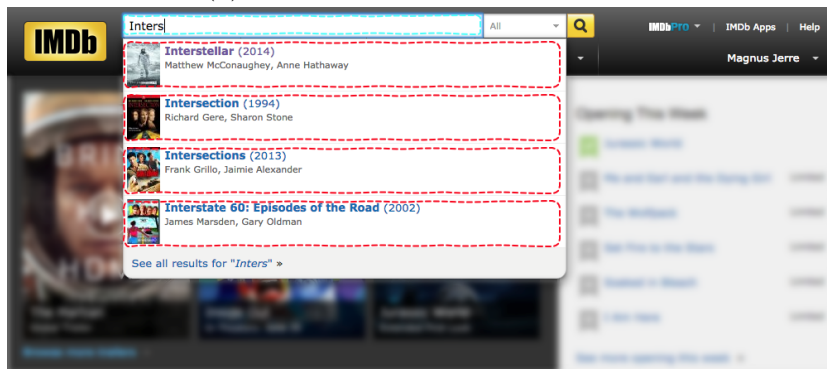
3.1.6 Requirements Summary

This section lists the requirements for the prototype tool to be developed related to what kind of functionality should be incorporated. The first list includes the requirements that are considered strictly necessary in order to increase the usefulness of using external data. The second list includes the requirements that are considered features nice to have, given that the time left to incorporate them is sufficient.

²Amazon is a website selling goods online and can be found at www.amazon.com

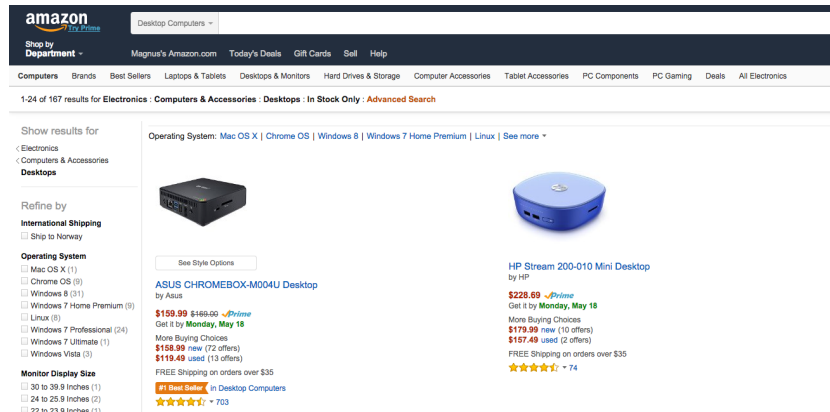


(a) Using the search bar on IMDb

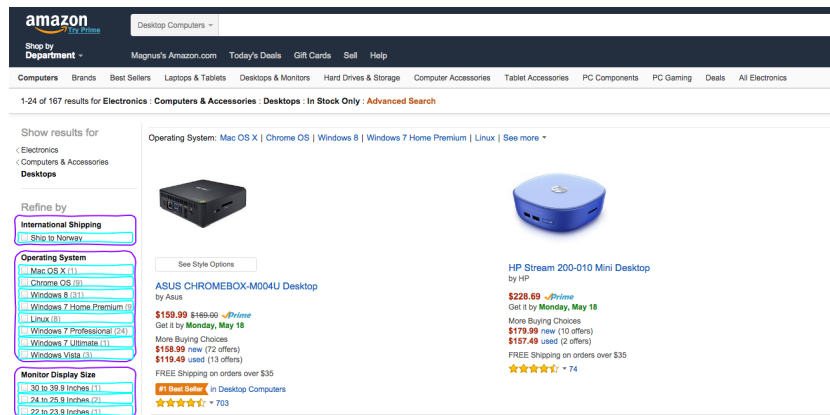


(b) The elements in the list are bound to the data resulting from the search query. The background has been blurred to make it easier to understand what is interesting in this screen.

Figure 3.4



(a) The left hand pane shows different filtering options for different categories.



(b) The left hand pane illustrates the different binding possibilities for the different categories.

Figure 3.5

The requirements in the first list are chosen because they are needed in order to create a prototype that can at least display the data for a given test scenario, such as: "You want to see more information for the movie Interstellar". By constraining the test case, functionality that is dependent on user input, such as search criteria, need not be implemented because it's expected that the tester will work towards completing the given test.

The requirements that are nice to have are features that help make the prototype feel even more real and don't constrain the test case in the same way. The above test case can be rephrased to: "You want to see more information for your favorite movie". The user can then see more information regarding a movie of his/her choice, rather than a predefined movie. Enabling the tester to affect the prototype output can actually help with cases not thought of, such as what to do if the movie doesn't exist or is misspelled.

The following features are deemed necessary

Using Data Sources The system should be able to use at least one external data source in order to have some data to populate the prototype with

Binding Data to Simple View Elements In order for the system to actually display any data from the external data source, a way of binding the data to the basic view elements is necessary.

Creating View Components To make the prototype development more modularised and efficient, enabling the creation and reuse of view components, i.e a grouping or collection of basic view elements, is necessary. View components can be especially practical when it comes to displaying complex lists.

Binding Data to Lists As the examples have shown, lists are heavily used when it comes to displaying lots of data.

The following features are deemed nice to have

Filtering Data Allowing the user to filter data will help make the prototype feel even more real. This is a feature that is also popular in many applications that use data. However, since test cases can constrain what to search for, it's not strictly necessary to incorporate this kind of feature.

Selecting Data A lot of the program flow is based on selecting various "things", such as a single movie from a list of movies. Therefore, enabling the prototype to display elements based on user input is a valid

feature to require. However, as with filtering data, the test cases can be constrained in such a way that it's not strictly necessary to incorporate this kind of feature.

3.2 Proposed Solution

This section will look into some solutions to binding external data to view elements in the prototype. In order to keep the examples clean, new and simple UI prototypes will be created rather than using the screenshots from the previous section. The examples will all be based on creating a prototype for a simple movie application that satisfy the requirements deemed necessary in section 3.1. The solutions proposed in this section will only target the method for binding data to view elements, not how the underlying code should be implemented.

To annotate the different view elements Fredrik Larsen's decorator method will be used. The decorator model concept helps annotate view elements that themselves don't provide any way of describing them. In addition it's a very easy to understand method providing a clean look at which elements are annotated and how.

The examples will use OCL for querying data since the language is fairly easy to understand. This doesn't mean that OCL must or even should be used for binding data to the view elements.

3.2.1 Using Data Sources

In order to have some data to populate the prototype with, it's necessary to provide the location for the data source. The simplest case is when only one source of data is necessary, as shown in 3.6a. The yellow decorator simply means that the data stored in the location given as input should be used as the data source. Since only one data model will be used, the context can be inferred when navigating the data.

A problem arises when multiple data sources will be used. Since the data sources are not separated by an identifier, such as a variable name, the context for the data bindings can't necessarily be inferred. Figure 3.6b illustrates a way to enable multiple data models. Each of the data sources are bound to their own variables, making it easy for data bindings to choose which data model should be used as source for binding data to view elements.

Forcing the data source to be stored as a variable will make all contexts behave and usable in the same manner, and help in making the context used clear.

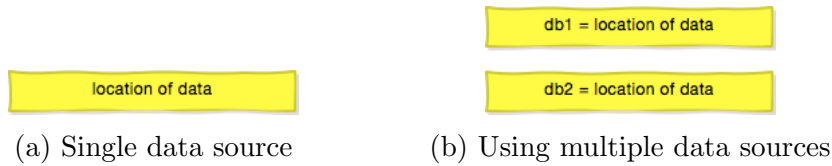


Figure 3.6

3.2.2 Binding Data to Simple View Elements

The previous section only looked at how data sources can be referenced in the prototype, not how to bind data to view elements. This section discusses the data binding step by building on the contexts defined in the previous section, as well as expanding the functionality of the contexts mentioned.

The following example illustrates a simple application displaying a little information regarding the actor "Matthew McConaughey", using data from the data source. Figure 3.7a shows the desired output for the application. Figure 3.7b illustrates how the prototype can be annotated in order to achieve the desired output. In addition to the yellow decorator defined in the previous section, purple decorators are added for this prototype. Each purple decorator contains a statement and has an arrow pointing from it to a view element. Naturally, the idea is that the view element will be bound to the data result from the given statement. Conceptually, the arrow represents the equal sign and the view element represents the variable name for the data result, therefore there is no need to include in the statement a variable name and an equals sign.

The example is a very simple one, yet the following sub-statement is repeated three times for the elements to be working with the same actor data.

```
db.actors->select(name = 'Matthew McConaughey')->at(1)
```

Adding more elements using information from the same actor will result in even more repetitions of the sub-statement. If we wanted to change the actor displayed in the output we would have to manually replace the name of the actor for all of the purple assignment boxes. This can be tedious, in addition, when there are a lot of assignments we might miss updating one or more of them. In order to overcome this issue it would be great to be able to store the actor in a variable which could easily be used as a starting point for the assignments.

Figure 3.7c illustrates how the actor instance can be stored in a variable by allowing the yellow decorators to not only contain the main data model instance, but also sub-instances. The prototype is easier to read and under-

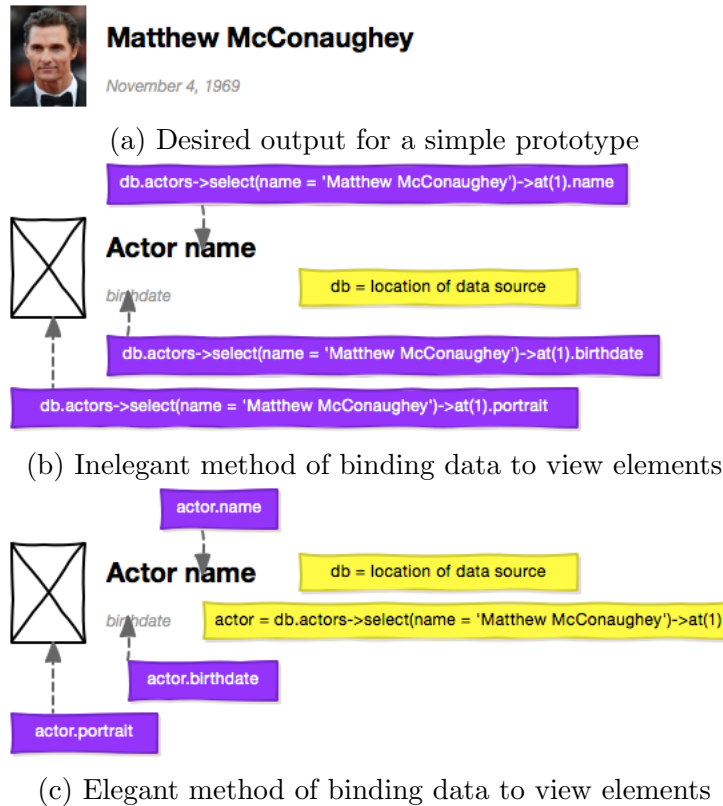


Figure 3.7

stand as it's now, it's also quicker and less prone to errors when changing the actor displayed in the output.

3.2.3 Creating and Using View Components

This section provides a look at how view components can be created and used in order to make the prototype more modular.

Looking at different websites it becomes clear that many of the elements on-screen have the same layout. This is very typical for lists and other manners of displaying a collection of data, figure 3.3a shows an example of this.

Figure 3.8a shows a simple example where two elements clearly use the same kind of layout, namely a portrait to the left, a name on the top and a birthdate on the bottom. The prototype needed to create this output can be made using the currently proposed solution for binding data to view elements. Figure 3.8b illustrates this. From the figure it becomes clear that the same type of assignments are repeated twice for each kind of view element. Had

the example included more information for the actor and actress even more of the assignments must be duplicated. Adding more actors or actresses to be displayed would result in each kind of assignment being duplicated even more. This clearly doesn't scale well.

In order to overcome the issue of repeating the same kind of assignments, one can lend a principle from the object oriented paradigm, namely classes. By defining the group of view elements and their data bindings as a class, or *view component*, one would only need to provide the data instance that satisfies the constraints for the view component, the properties will then automatically bind to the correct view elements.

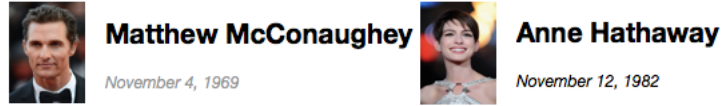
Figure 3.8c illustrates how a view component can be defined. A new type of annotation box has been added, the orange one. The orange decorator is responsible for defining the name of the view component, much like defining the name of a class, so that it can be referenced other places in the prototype. The view component assignments look a little different from the ones in the previous sections, they don't include a context variable name for the data instance to work with. That is because the assignments will automatically work with the data instance, or context, provided by the view component.

Now that the the view component has been defined, the prototype must be able to use it. Figure 3.8d illustrates how to do just that. The number of assignments has been reduced from six to two, making the prototype look cleaner. The assignments now consist of two lines, one being the data value the other being the name of the view component to use. An important thing to note is that the assignments now try to bind complex data to a group of view components, whereas in the previous example the assignments simple data values were bound to simple view elements. Since there is not necessarily any obvious way of binding complex data to view elements, the complex data must be split into simple data assignable to simple view elements. What happens if the complex data uses a view component, then one or more of the assignments for the view component results in complex data? A view component should be able to have assignments that use other view components. As long as there is a base case view component having only simple assignemnts, the view components can be handled in a recursive manner.

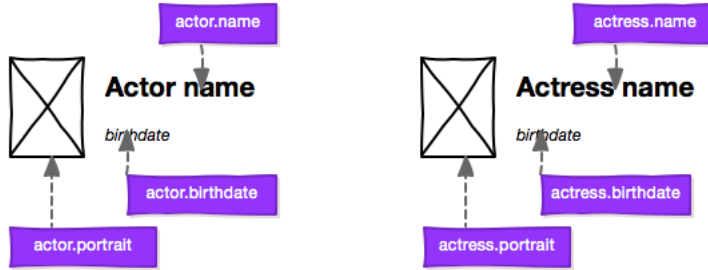
3.2.4 Binding Data to Lists

The power of using an existing data source is easily illustrated by displaying a big chunk of it using lists. Ideally, the developers should be able to display lots of data using lists through little effort. This section will look at how a list can bind data to each of its elements.

Figure 3.9a shows an example of a list containing simple view elements,

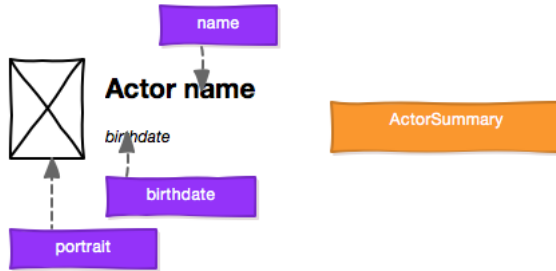


(a) Desired output for a simple prototype

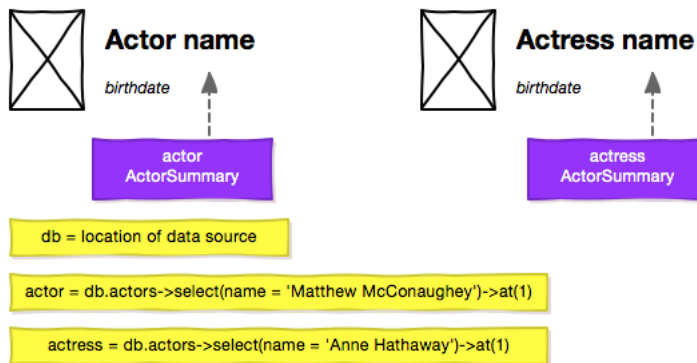


```
db = location of data source
actor = db.actors->select(name = 'Matthew McConaughey')->at(1)
actress = db.actors->select(name = 'Anne Hathaway')->at(1)
```

(b) Inelegant method of binding data to view elements



(c) Defining a view component



(d) Elegant method of binding data to view elements

Figure 3.8

namely strings. One way to bind data to each of the view elements in the list would be to try and make assignments for each element in the list, illustrated in figure 3.9b. This method might be fine for hand-picking a small number of data from a large data set, it would however be very slow and not at all scaleable for larger lists. It will also result in a very cluttered prototype that might be difficult to read.

Figure 3.10a shows a different example of a list. Unlike the previous example, this one uses complex elements to display more information about each data element. Using a similar annotation method as presented for the simple list yields something like figure 3.10b. For each list element there exists three assignments, one for the name, one for the birthdate and one for the portrait. Each of the assignments naturally point to the view element it should bind the data to. The example contains three list elements all of which have view elements that can have data directly bound to them using assignments. However, for the remaining data elements that don't fit inside the list, there is no logical way of binding the data to the correct view element. This differs from the case for simple lists, where all elements can just point to the list itself even though there is not enough visual space in the list. In addition to being a very inefficient means of binding data to lists, it won't work for all kinds of lists, making the method unsuitable.

One possible solution is to assign the entire data list to the visual list, then having the software be responsible for binding each of the data values to the correct view elements. Figure 3.9c illustrates how this can be achieved. The data result from the assignment is a collection of names, each of which can be directly bound to a simple view element due to the fact that the data values themselves are simple. For complex data values, the tale will be quite similar. By utilising the notion of view components, it's possible to specify that each of the elements in the list will display data using a specific view component. Figure 3.10c illustrates how the list can be annotated to facilitate the use of the view component defined in figure 3.8c.

In addition to being an efficient way of binding data to lists, it's simple to understand, easy to swap out the data to display and doesn't clutter the prototype more than necessary. One might argue that hand-picking the elements for the list is harder, but that is entirely dependent on the language used to query the data. OCL for instance provides many features for selecting only a subset of data.

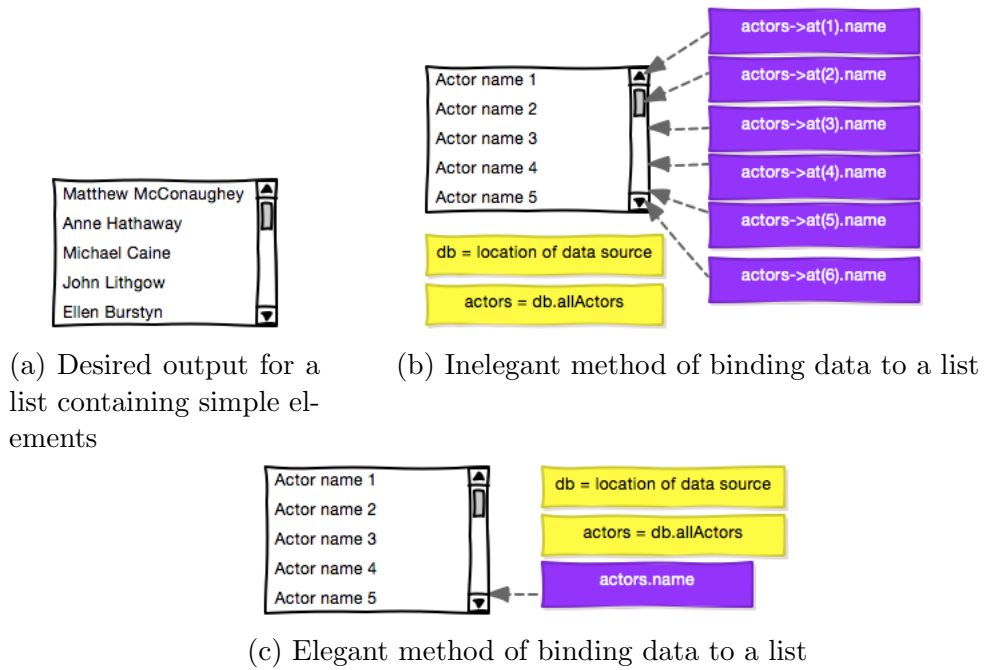
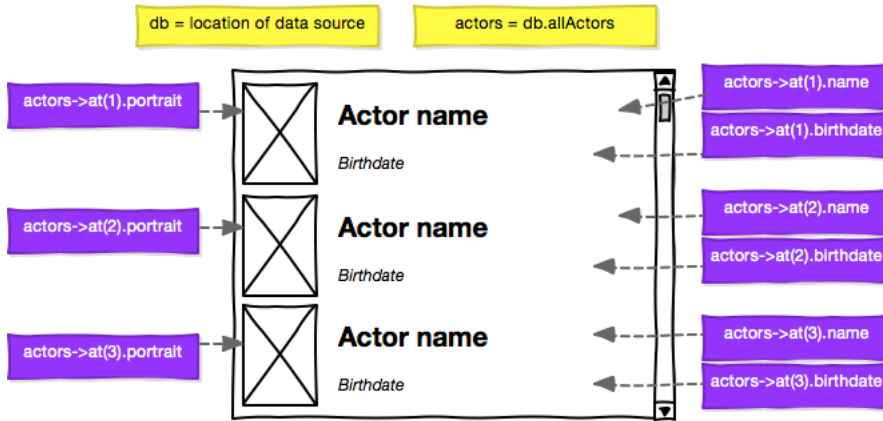


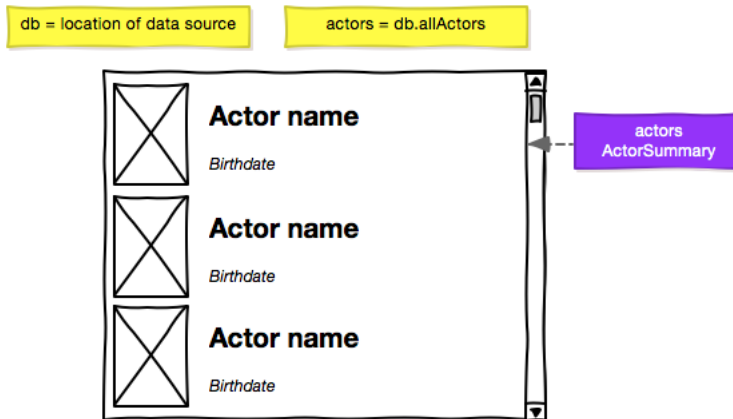
Figure 3.9



(a) Desired output for a list containing complex elements



(b) Inelegant method of binding data to a complex list



(c) Elegant method of binding data to a complex list

Figure 3.10

Chapter 4

Results

4.1 Choices

This section will go through the choices regarding technology, features and scope for this thesis.

4.1.1 Underlying Technology

The main underlying technologies for the prototype software to be developed is largely determined from what Fredrik Larsen used, which is a combination of Wireframesketcher, EMF and JavaFX as well as programming language called Xtend.

EMF will help with dynamically creating the necessary view models and instances for the prototypes. In addition, EMF is based on using ecore and XMI files for storing the meta model and instance model respectively. Since many sources of information can be converted to ecore and XMI, either directly or through successive conversions, using EMF's ecore and XMI format as the source for data to populate the prototype with is a fair choice.

JavaFX will naturally be used since this is the technology Fredrik Larsen chose for the runtime application. Switching it out with something else would mean to start entirely from scratch, rendering Fredrik Larsen's code unusable for this thesis. Due to time constraints, starting from scratch is not an option.

Xtend provides a lot of syntactic sugar for the Java programming language, it's however resource intensive. The computer that will be used to develop the prototyping tool is not sufficiently powerful to efficiently handle Xtend for software development, therefore normal Java code will be used instead.

4.1.2 Navigational Language Chosen

Four different query languages were introduced in section 2.5. Each of these languages work best with data models that are available in their respective domains. As EMF is the chosen framework for developing the software, using the models native to emf, namely ecore and XMI, is a natural choice.

SQL is aimed at querying into relational databases, however an XMI file is not a relational database. SQL is therefore not suitable for this application.

XMI is a variant of XML, therefore using a navigation language, such as XPath, based on navigating XML documents can be a good alternative. Retrieving linked nodes is however cumbersome compared to other languages, such as Java and OCL.

Since ecore- and XMI files can be loaded into a runnable Java program as in-memory objects, using languages based on navigating dynamic instances can be a great idea. Using a language such as Java will therefore be intuitive, it is however necessary to provide a sort of mapping from the java statement to the instances since methods like getters will not directly work without providing the instances' class definition or using reflection.

An alternative to writing Java code is to use OCL for navigating the data. EMF provides an OCL parser that allows both the expected type as well as the actual value from a query to be retrieved. Using the OCL parser will spare a lot of work related to creating a parser from scratch. Another advantage in using OCL is that it's simple to make queries that selects only a subset of elements that meet some constraint.

Due to the existing OCL parser within EMF and that it's a language fairly easy to learn, OCL will be the language used for binding data to view elements.

4.1.3 Scope - Focus on Data Retrieval

In order to make this project manageable within the allotted time it's necessary to limit the scope. There are four operations that can be performed on persistent data, namely create, read, update and delete, shortened CRUD [3]. Three of the operations are related to modifying the persistent data, this functionality is however not strictly necessary for conducting usability tests with the prototype. Allowing a tester to create, update or delete data can be simulated by constraining the test case to expect a certain input, then use another data model containing the data that should be inserted or modified. This thesis will therefore focus on reading data only.

The implemented prototyping tool will be further further limited to not support filtering specified by a user during testing due to time constraints.

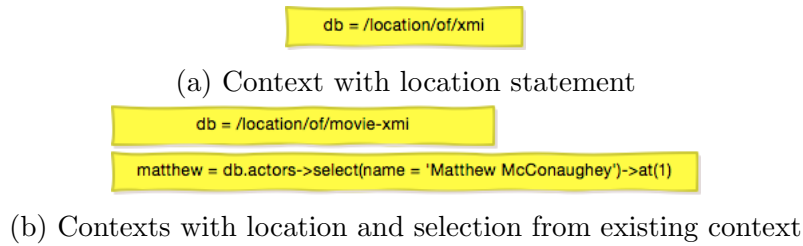


Figure 4.1

4.1.4 Decorators to Use

An important aspect of annotating using decorators is which kind of decorators are necessary to provide an intuitive way of annotating the view elements. Fredrik Larsen created three different decorators, namely data, action and style. Each have their own area of functionality, however, these areas don't perfectly fit the needed functionality for populating a wireframe prototype with data. This thesis will therefore propose three additional decorators to use, namely context, assignment and view component.

Context Decorator

The context decorator will be responsible for providing the main contexts for a specific screen file. A context contains a variable name and an assignment statement, and is colored yellow. The variable name is necessary in order to be able to access the correct content when binding data to a view element. A context statement can either be the location of an XMI file, starting with a slash, or a selection from an existing context. Figures 4.1a and 4.1b illustrate this.

The context decorator is a lot like Fredrik Larsen's data decorator with regards to functionality. It has been chosen to be a separate part however because in addition to storing a value, it's also responsible for storing the statement producing that value. Also, it will be easier to implement, which is an important factor due to the time constraints.

Assignment Decorator

The assignment decorator is responsible for actually binding data to a view element. Unlike the context decorator, the assignment decorator won't contain a variable name because the value it produces is not expected to be used by other elements in the view. Ordinarily, an assignment will be a one-liner, however, an assignment can produce a complex value, such as an object, and

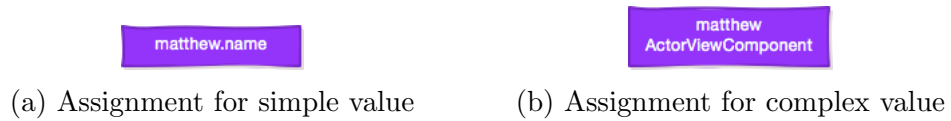


Figure 4.2



Figure 4.3: A view component named "ActorViewComponent" that expects a value of type "Actor"

must be able to correctly handle this. A second, optional line in the assignment decorator defining which view component the complex result should be associated with must be used.

Figures 4.2a and 4.2b illustrate both simple and complex assignment annotation boxes using the contexts defined in figure 4.1b. The complex assignment in figure 4.2b uses a view component named "ActorViewComponent". In order to actually assign the value produced from the purple assignment box, an arrow must be drawn from it to the view element it binds the data to.

View Component Decorator

The view component decorator is responsible for defining which group of view elements are part of a view component and the assignments it contains. The decorator contains a name for the view component as well as the name of the type it expects as input. Initially, the view component didn't require type declaration, however, at later development iterations it became clear that this was necessary. As with the assignment boxes, an arrow must be drawn to the view elements it references. Unlike the assignment boxes, no value is actually produced from the view component decorator.

An assignment using a view component passes its value to the view component which then delegates to its sub-assignments the values that should be bound to the correct view elements. Figure 4.3 illustrates what a view component annotation box can look like. The first argument is the name of the view component, "ActorViewComponent", the second argument is the name of the expected type, "Actor".

4.1.5 Handling Type

Each data value in a data model is of a specific type, be it Integer, String, Object or some other custom type. How they should be handled by the prototype depends on the what type it's dealing with.

During the first two development iterations, type declaration was not enforced in order to make everything as general as possible. However, after a discussion with the supervisor it became clear that enforcing type declaration was necessary.

When it comes to assignments that directly use a context, it is possible to automatically infer the type it produces by using the built in OCL parser. Therefore, forcing these kinds of assignments to explicitly declare a type is unnecessary. However, an assignment that is part a view component can't be directly interpreted by the OCL parser since it doesn't know what type to start parsing the statement on. One way to solve this problem is to enforce type declaration for assignments, after all, the developer must have some idea of what the expected type for the assignment is. Alternatively, type declaration can be enforced for the view component owning the assignment. This way the OCL parser can be used to infer the type for the assignment given the type of the view component, and the assignments can be kept as simple as possible.

Forcing the view component to declare a type means that it can't necessarily be reused by values of different type having the same fields, they have to be of the same type or super type.

The choice to force the view components to declare their type rather than the assignments was made because the number of assignments that will be used per screen will be higher than the number of view component definitions. Forcing each assignment to declare a type will be much more cumbersome than forcing its view component definition to declare its type.

4.2 Implementation

This section describes the different implementation iterations as well as the key points to how the prototype is implemented.

4.2.1 Implementation Iterations

The implementation was done through several iterations. The first iteration focused on enabling simple data binding by using contexts and simple assignments. Each assignment were stored using general types, such as an Object and EObject rather than the more specific String or Integer types, leaving

the handling of type to the code responsible for populating the runnable JavaFX application.

During the second iteration view components were implemented, again in a general way. The view components did in other words not expect any specific type, other than that it had to be an EObject, which is a general object used in EMF. Making it general meant that assignments of different types could use the same view component as long as they had the necessary fields.

Making everything general seemed like a good idea initially, but after demonstrating the results to the supervisor, it became clear that the generated view models were hard to understand. Another issue with the generalization was that the code needed to run and populate the JavaFX application got very complex, making further support for lists and complex lists a lot harder than it needed to be. After some discussion with the supervisor, the decision to make the view models less general was made.

The third iteration focused on modifying the existing code to adhere to a less generalised view model. Since each decorator needs to provide its expected type, the view component's implementation was changed to require information on its expected type, thereby allowing its assignments to interpret their expected types as well. The result was a view model that was a lot easier to understand than the previous implementation.

During the fourth iteration support for both simple and advanced lists were added. After the shift towards a more specialised view model, implementing lists was quite straight forward.

The fifth and final iteration was mostly focused on making the code easier to understand and fixing bugs. Since there was some development time left, a feature not deemed necessary was quickly implemented, namely selection. The selection works by providing Fredrik Larsen's Action decorator with a variable name and type, then storing it in a special selection model.

4.2.2 Final Implementation

The prototype is implemented with focus on separating between the data model and view model for the prototype, as well as separating between the meta model and instance model.

Wireframesketcher is built around creating what is called screens. Each screen is its own file and represents one screenshot. Each screen that displays any kind of external data will have two different types of models associated with them, one being the data model which is the source of the prototype's data and can be shared between multiple screens, the other being the view

model which represents the data the screen actually needs which is unique for each screen.

The data model is defined external to the prototype and is therefore not part of the implementation. The view model on the other hand is generated based on the screen files. A generated ecore file contains at least one class having the same name as the screen file, and optionally one class for each view component defined.

Main Ecore Class The main class for the ecore file contains one field for each context element, and one field for each assignment not part of a view component. Each of these fields are given the appropriate type and contains extra information associated with them in the form of details. Each context will be given a name based on its variable name and have exactly one detail associated with it. Depending on whether the context contains the location of the XMI file, the detail will be named "xmiLocation" and contain the actual location of the XMI file or it will be named "ocl" and contain the statement given as part of the context. Figure 4.4 illustrates the pattern for context fields.

Each assignment is given a name based on its statement and contain either two or three details depending on whether it uses a view component or not. Every assignment contains at least two details, one named "ocl" containing the actual statement for the assignment, the other name "layoutId" containing the actual id value for the view element the assignment points to. If an assignment uses a view component, the detail "useComponent" contains the name of the view component the assignment uses. Figure 4.4 illustrates the pattern for assignment fields, both as part of the main screen and as part of a view component.

The fields are ordered in a way such that their statements can be executed in a top-down order, making instance population easier.

View Component Classes Each of the view components for a screen will be its own class containing only fields that are its assignments. Unlike a main class the view component class has one detail associated with it named "expectedType" which contains the name of the expected type for the view component. The assignments have the exact same type of details as the assignments part of the main class. Figure 4.4 shows the pattern for a view component.

Selection To accommodate selection, a separate model named *selection-Model* is generated. This model is shared by the entire generated prototype,

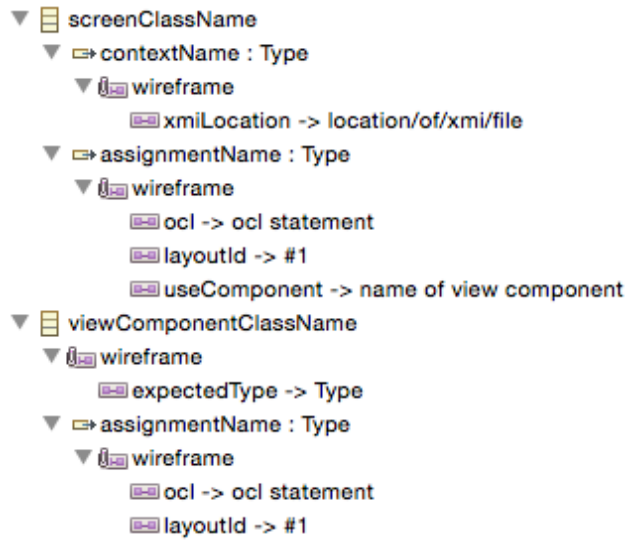


Figure 4.4: Sample showing the pattern for a generated ecore file

and contains the name for all selections, their types, as well as their details regarding each view element that can affect their value. The selection model is made accessible by contexts that contain the value `#selectionModel`.

The generated ecore meta models for the screens is later used at runtime to create instances of them. To populate the instances with the correct data, the runnable prototype need only perform the statements for each of the fields in its class.

4.3 Testing the Implemented Software

In order to be able to verify how well the proposed method and implemented solution works it must be tried out. The screenshots from section 3.1 will be used as the basis for creating design prototypes in Wireframesketcher since these are real world applications. The prototypes will then be annotated according to the proposed method. The resulting JavaFX prototype will not look exactly like the reference screenshots due to the prototyping tool's lack of styling options.

The tests are separated into three different scenarios, all based on the requirements deemed necessary. The selection requirement is not part of the test scenarios since it is part of requirements deemed nice to have. The first and simplest scenario is based on displaying information regarding a special movie. The second scenario displays the top 10 list of 2014 from IMDb and illustrates how view components work. The third scenario, illustrated using

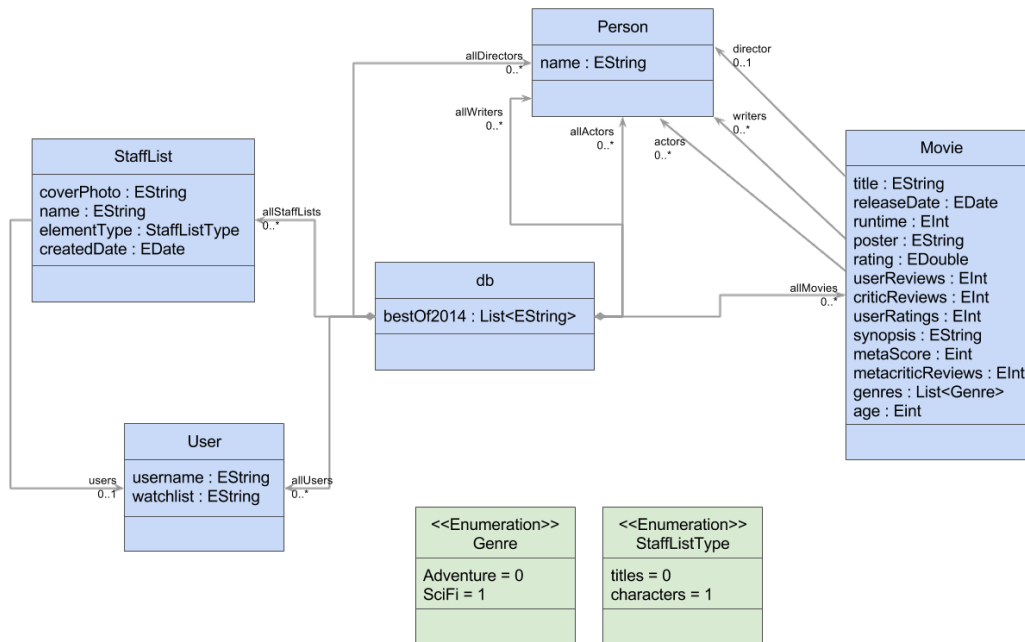


Figure 4.5: The class diagram for the IMDb experiments

Soundcloud¹, is even more complex than the second scenario in that the elements part of view components themselves use view components.

For the prototype to display any external data, two different data models were created, one for the IMDb scenarios and another for the Soundcloud scenario. Both models were created based on what kind of data is displayed, they do however not represent the scenarios' view model, but rather possible versions of the underlying data model. Figures 4.5 and 4.6 illustrates the IMDb and Soundcloud data models respectively.

4.3.1 Binding Data to Simple View Elements

The first scenario illustrates how simple data binding works. The detailed view for the movie "Interstellar", figure 4.7a, is the basis for the designed wireframe prototype shown in figure 4.7b. By correctly annotating the wireframe prototype a JavaFX application resembling the source is be generated.

Since the prototype doesn't display the same kind of information multiple times there is no need for using view components. The necessary annotations are illustrated in figure 4.8. The three assignments `movie.genres`, `movie.writers.name` and `movie.actors.name` produce lists but point to

¹Soundcloud is a website for sharing and discovering music. It can be found at www.soundcloud.com

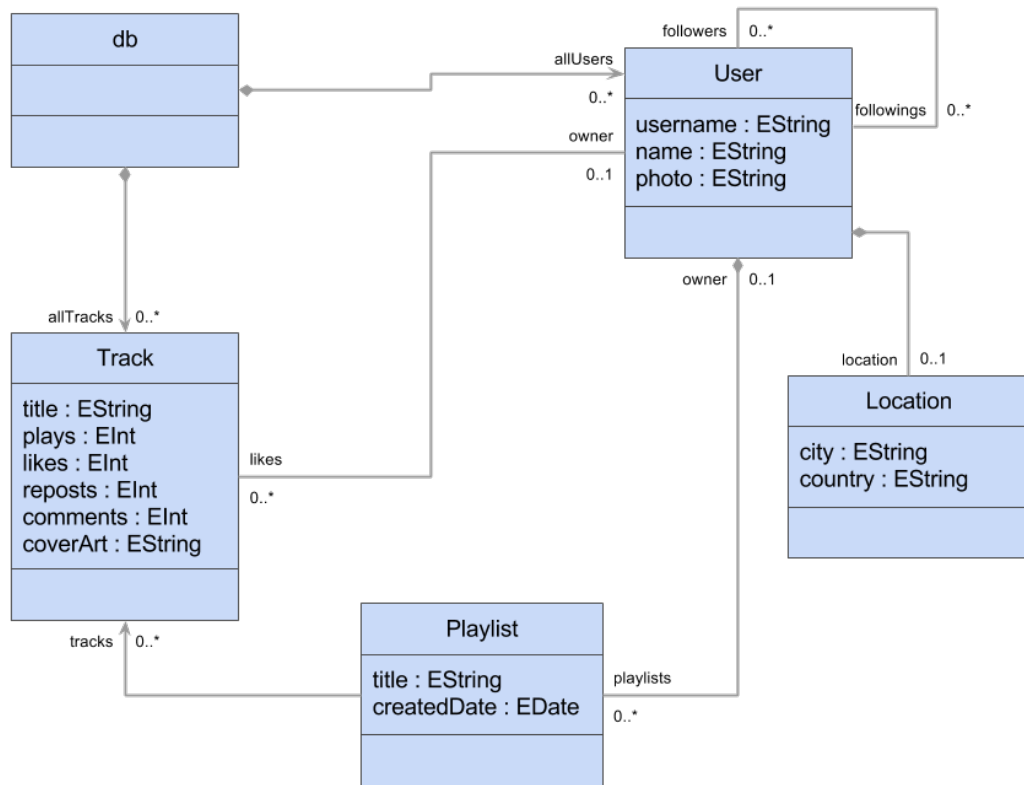
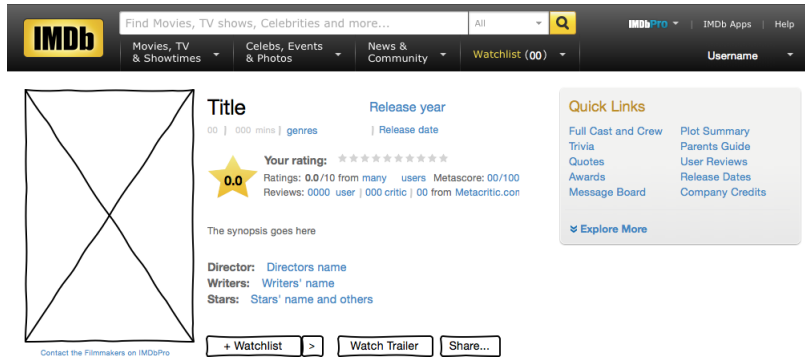


Figure 4.6: The class diagram for the Soundcloud experiment



(a) Details for the movie Interstellar



(b) Corresponding wireframe design for figure 4.7a

Figure 4.7

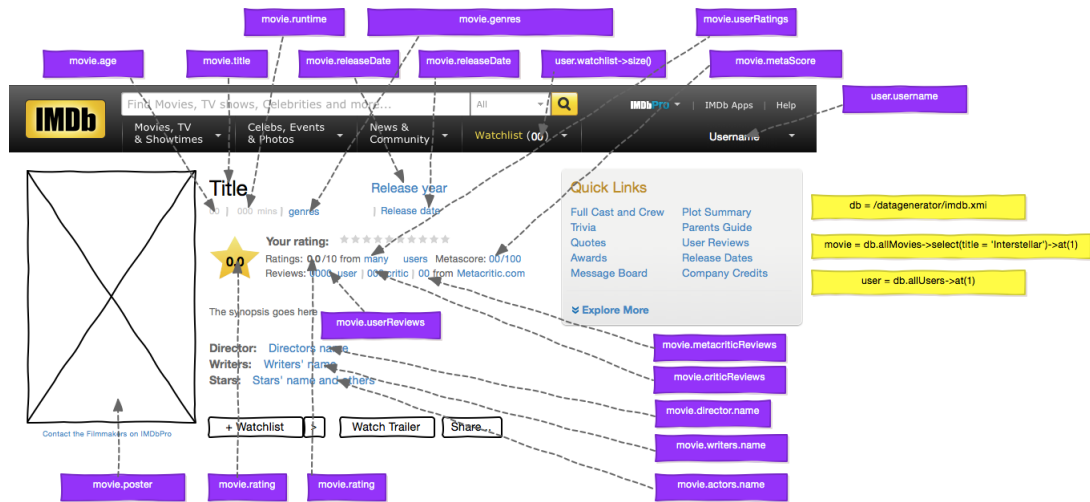


Figure 4.8: The annotated screen file

labels. To handle this, the application was implemented in such a way that lists bound to labels will concatenate each elements' `toString()` representation separated by a comma.

After the wireframe prototype has been correctly annotated, the generation program is run to produce the necessary fxml and ecore files. Figure 4.9a shows what the resulting ecore-file looks like. Some of the assignments and contexts have been expanded to show the details they contain. The resulting JavaFX application is shown in figure 4.9b.

4.3.2 Binding Data to Complex View Elements

Both the second and third scenario illustrate how binding with complex, reoccurring elements works. The simpler of the two scenarios is related to displaying a list of the top 10 movies of 2014 along with a couple of other user generated lists. The more complex one displays lists inside lists as well as multiple lists taking the same kind of data as input but displaying it differently.

Top 10 Movies of 2014 Scenario

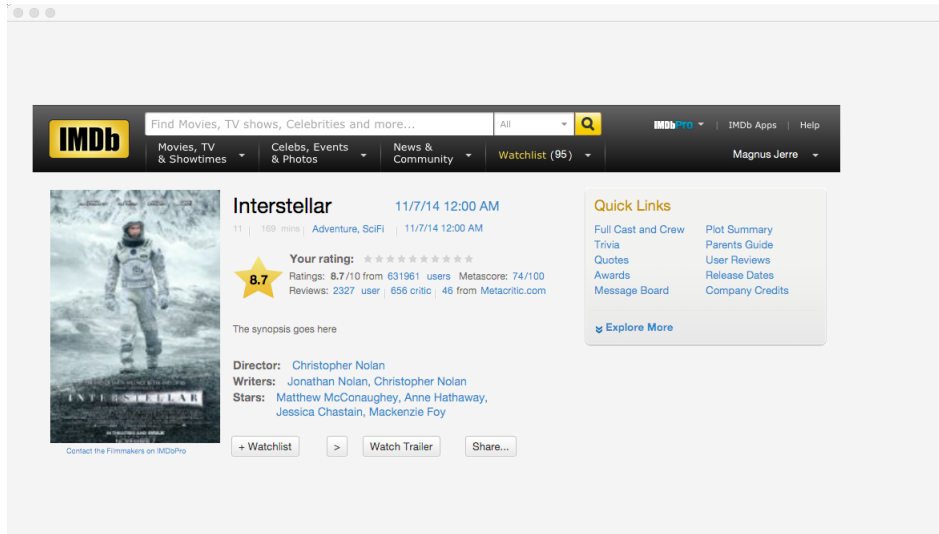
The following scenario is based on IMDb's list of the top 10 movies of 2014. Figure 4.10 is a screenshot taken from their website, the corresponding wireframe prototype created in Wireframesketcher is shown in figure 4.11. All the list elements have been replaced by placeholder view elements that await real data when annotated and generated into a runnable prototype.

```

platform/resource/WireframeToJavaFX/src/generated/moviedetail.ecore
├── moviedetail
│   ├── moviedetail
│   │   ├── db : db
│   │   │   ├── wireframe
│   │   │   │   ├── xmlLocation -> /Users/Magnus/Master/Workspace_final/MasterThesisCodeFinal/WireframeToJavaFX-master/JavaFX Test/src/datagenerator/imdb.xml
│   │   │   ├── movie : Movie
│   │   │   │   ├── wireframe
│   │   │   │   │   ├── ocl -> db.allMovies->select(title = 'Interstellar')->at(1)
│   │   │   ├── user : User
│   │   │   │   ├── aMoviePoster1 : EString
│   │   │   │   │   ├── wireframe
│   │   │   │   │   │   ├── ocl -> movie.poster
│   │   │   │   │   │   ├── layoutId -> #13
│   │   │   │   ├── aUserUsername2 : EString
│   │   │   │   ├── aUserWatchlist3 : EIntegerObject
│   │   │   │   ├── aMovieTitle4 : EString
│   │   │   │   ├── aMovieReleaseDate5 : EDate
│   │   │   │   ├── aMovieAge6 : EIntegerObject
│   │   │   │   ├── aMovieRuntime7 : EIntegerObject
│   │   │   │   ├── aMovieReleaseDate8 : EDate
│   │   │   │   ├── aMovieRating9 : EDoubleObject
│   │   │   │   ├── aMovieRating10 : EDoubleObject
│   │   │   │   ├── aMovieDirectorName11 : EString
│   │   │   │   ├── aMovieMetacriticReviews12 : EIntegerObject
│   │   │   │   ├── aMovieCriticReviews13 : EIntegerObject
│   │   │   │   ├── aMovieUserReviews14 : EIntegerObject
│   │   │   │   ├── aMovieMetaScore15 : EIntegerObject
│   │   │   │   ├── aMovieGenres16 : Genre
│   │   │   │   ├── aMovieActorsName17 : EString
│   │   │   │   ├── aMovieUserRatings18 : EIntegerObject
│   │   │   │   ├── aMovieWritersName19 : EString
└── platform/resource/WireframeToJavaFX/src/datagenerator/imdb.ecore

```

(a) Ecore file for the generated JavaFX prototype



(b) The final JavaFX program

Figure 4.9

IMDb Find Movies, TV shows, Celebrities and more... All

Movies, TV & Showtimes | Celebs, Events & Photos | News & Community | Watchlist (95) | Magnus Jerre

BEST OF 2014

Top 10 Movies of 2014

by IMDb-Editors created 6 months ago | last updated - 5 months ago

These are the highest rated movies of 2014. The ratings are tabulated from user ratings during 2014 and then ranked based similarly to the IMDb Top 250 movies.

- 1. Interstellar (2014)**
 ★★★★★★ 8.7/10
 A team of explorers travel through a wormhole in space in an attempt to ensure humanity's survival. (169 mins.)
 Director: Christopher Nolan
 Stars: Matthew McConaughey, Anne Hathaway, Jessica Chastain, Mackenzie Foy
[Add to Watchlist](#)
- 2. Boyhood (2014)**
 ★★★★★★ 8.1/10
 The life of Mason, from early childhood to his arrival at college. (165 mins.)
 Director: Richard Linklater
 Stars: Ellar Coltrane, Patricia Arquette, Ethan Hawke, Elijah Smith
[Add to Watchlist](#)
- 3. Gone Girl (2014)**
 ★★★★★★ 8.2/10
 With his wife's disappearance having become the focus of an intense media circus, a man sees the spotlight turned on him when it's suspected that he may not be innocent. (149 mins.)
 Director: David Fincher
 Stars: Ben Affleck, Rosamund Pike, Neil Patrick Harris, Tyler Perry
[Add to Watchlist](#)
- 4. Guardians of the Galaxy (2014)**
 ★★★★★★ 8.1/10
 A group of intergalactic criminals are forced to work together to stop a fanatical warrior from taking control of the universe. (121 mins.)
 Director: James Gunn
 Stars: Chris Pratt, Vin Diesel, Bradley Cooper, Zoe Saldana
[Add to Watchlist](#)

Best of 2014

- IMDb Countdown
- In Memoriam
- Top 10 Stars
- Top 100 Stars
- Top 10 Breakout Stars
- Top 10 Movers and Shakers
- Top Movies
- Top TV Shows
- Top 25 Box Office
- Col Needham's Top 10 Films
- Keith Simanton's Top 10 Films
- Ray Subers' Top 10 Films
- Top User-Rated TV Shows
- Top 10 Most-Viewed Trailers
- Melanie McFarland's Top 10 TV Episodes
- Top 10 Box Office Success Stories
- Top 10 Character Deaths

IMDb Staff Lists

- Best Kids Movies of 2014**
a list of 10 titles by EricGreene created 5 months ago
- thobilly's Best of 2014**
a list of 25 titles by thobilly created 5 months ago
- 2014: Best TV and Movies for Kids**
a list of 10 titles by VideoJordan created 6 months ago
- Best of 2014**
a list of 11 titles by buraczyn created 5 months ago
- Best Characters of 2014 (IMHO)**
a list of 8 characters by lindsay-421-467019 created 5 months ago
- My top 15 2014 Movies**
a list of 15 titles by linazep created 5 months ago
- scottmo's top 10 of 2014**
a list of 10 titles by scottmo-1046 created 6 months ago

Figure 4.10: Sample screenshot containing multiple reoccurring elements

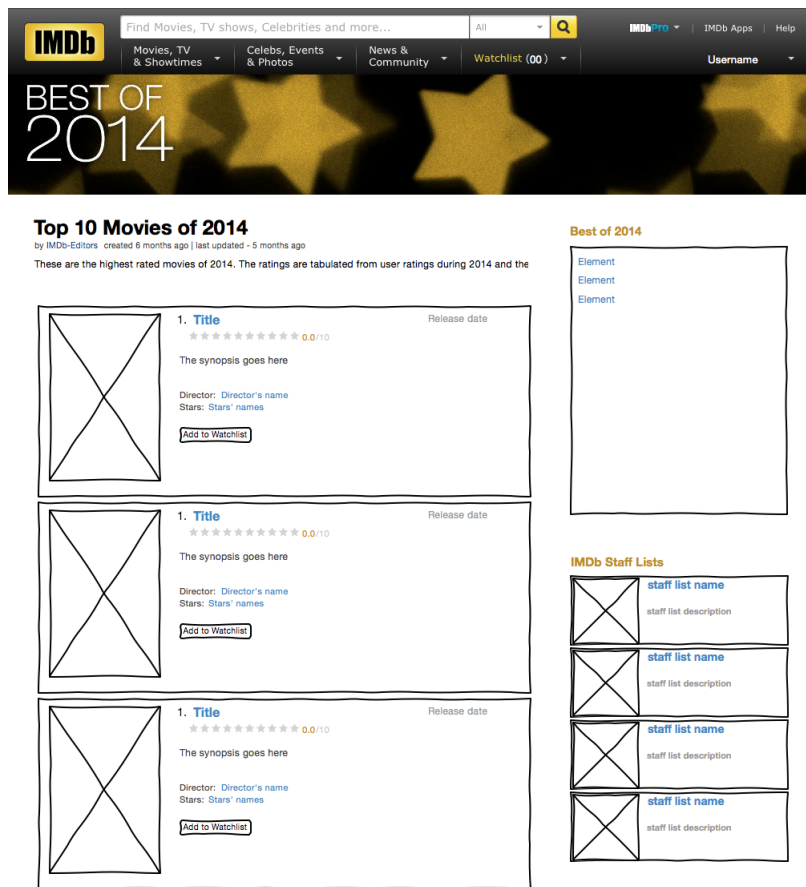


Figure 4.11: Corresponding wireframe design for figure 4.10

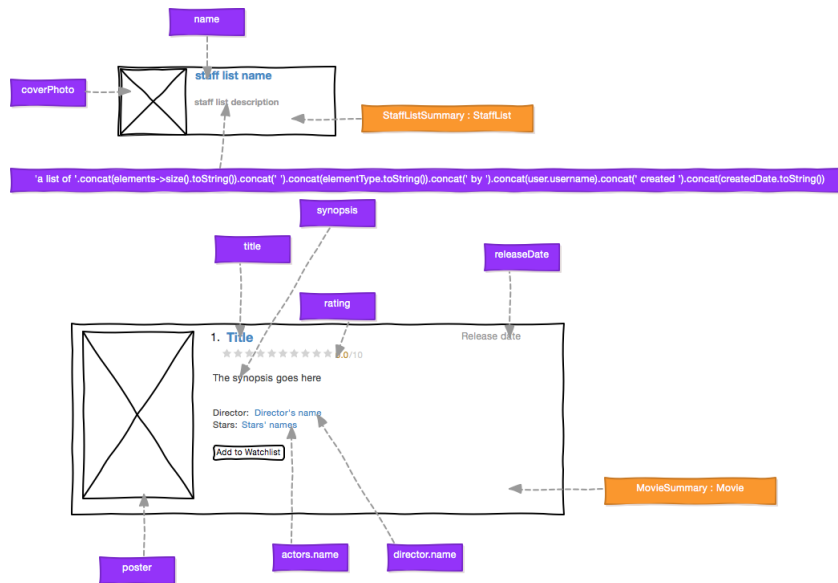


Figure 4.12: The view component annotation definitions

Unlike the first scenario, this one requires the use of view components to correctly and efficiently bind data to the different view elements. The first step to creating view components is to group each view element part of a single view component together and placing a duplicate of them at a different location in the screen file. An example of the necessary duplicate view groups along with their annotations is shown in figure 4.12. The view component decorator need only point somewhere inside the group of view elements to be associated with the view group. Each of the assignments part of a view component must point exactly to the view elements they will bind to.

Each of the lists in figure 4.13 have been annotated with an assignment pointing to its list view group. For the application to understand that the group of elements it's working with is actually a list, the group must be given a property name of "list" using Eclipse. Otherwise the application expects a single view component to be displayed instead of a list. Figure 4.14 illustrates how a group of view elements is given the property name "list" using Eclipse.

The resulting ecore file for the prototype contains three classes, the class named "top10" which is the screen that will be displayed and the classes "MovieSummary" and "StaffListSummary" which represent the definitions for the two types of view components the screen will use. Each of the view components will also have separate fxml files generated for them. When running the generated prototype, a JavaFX application like figure 4.16 is

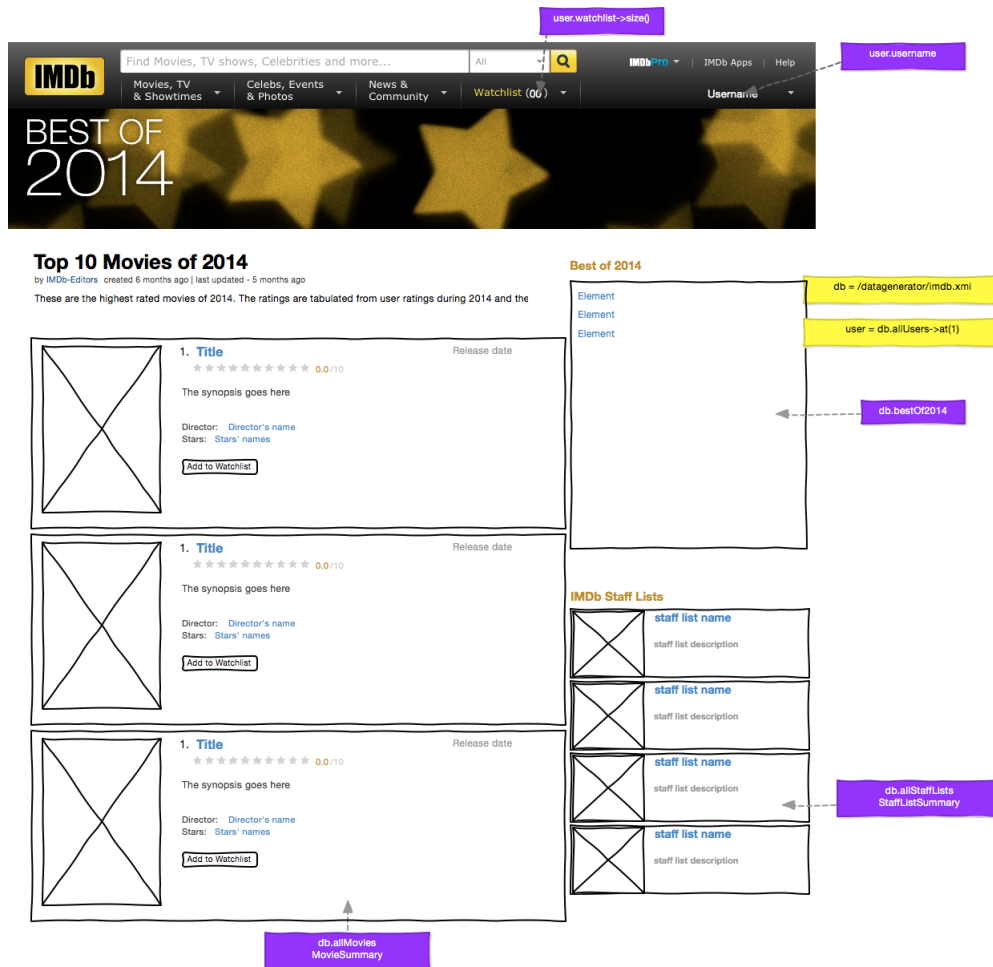


Figure 4.13: The annotated wireframe prototype excluding the definition of the view components

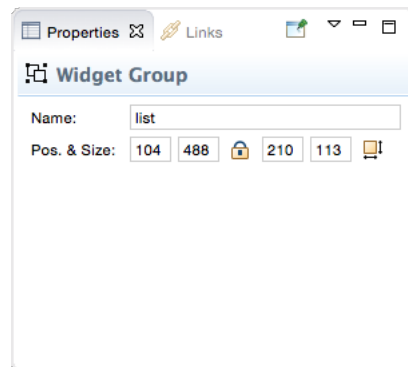


Figure 4.14: Providing a group of view elements with the "list" property name



Figure 4.15: Ecore file for the generated "Top 10" application

displayed.

Soundcloud Profile Page Scenario

The last scenario is based on using the profile page for a Soundcloud user as illustrated in figure 4.17. Soundcloud's profile page is a lot more complicated than the "Top 10" list from the previous scenario. The main list in the middle have list elements that temselves contain lists. In addition, there are multiple ways to display lists of playlists and tracks each of which must be defined. The corresponding wireframe prototype for the Soundcloud profile page is shown in figure 4.18.

All steps from the previous scenario must be taken here as well. Each of the view elements part of a single view component is grouped together then duplicated and moved to a different location in the screen file. Each of the list elements in the main screen and inside the view components are grouped and given the property name "list" using Eclipse. The annotated screen and view components can be seen in figures 4.19 and 4.20.

The generated ecore file resembles the one in the previous example. A total of five classes are generated, the first being the class for the main screen namely "soundcloud", the other four each represent their own view component. As in the previous example, fxml files for each of the four view components are generated. The runnable JavaFX application is shown in figure 4.22.

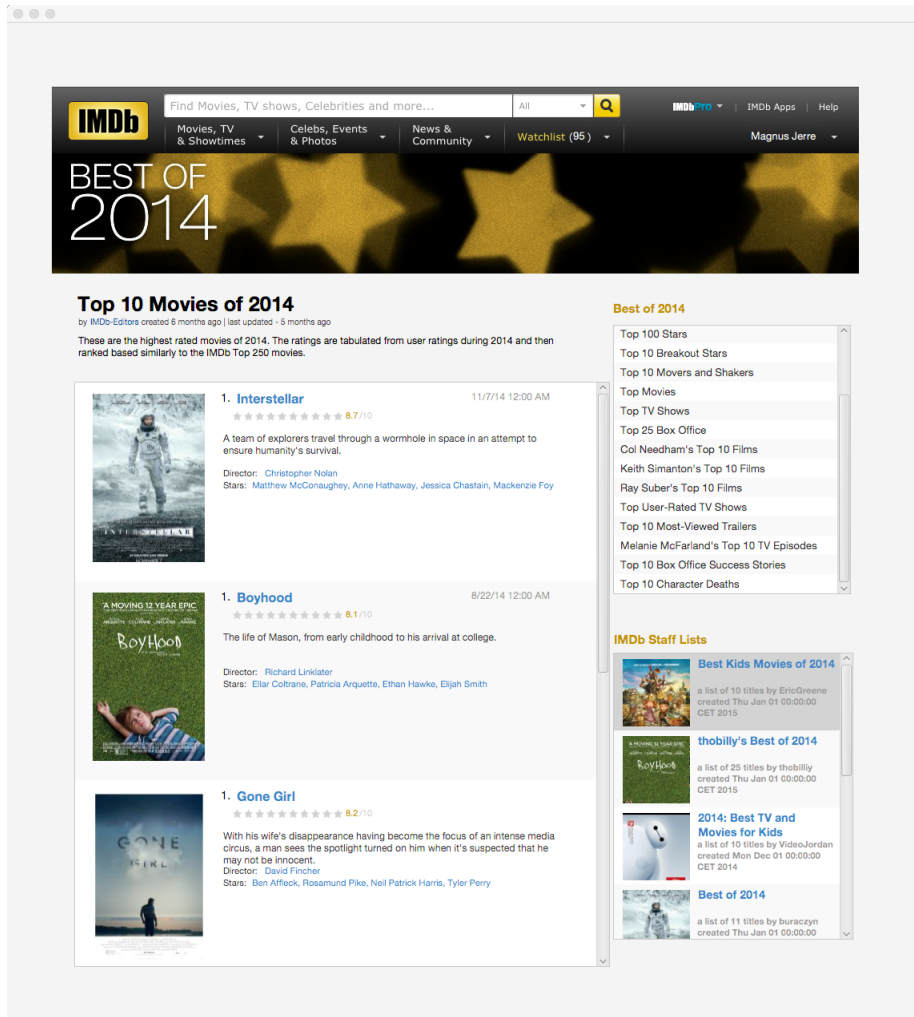


Figure 4.16: JavaFX application for the "Top 10" application

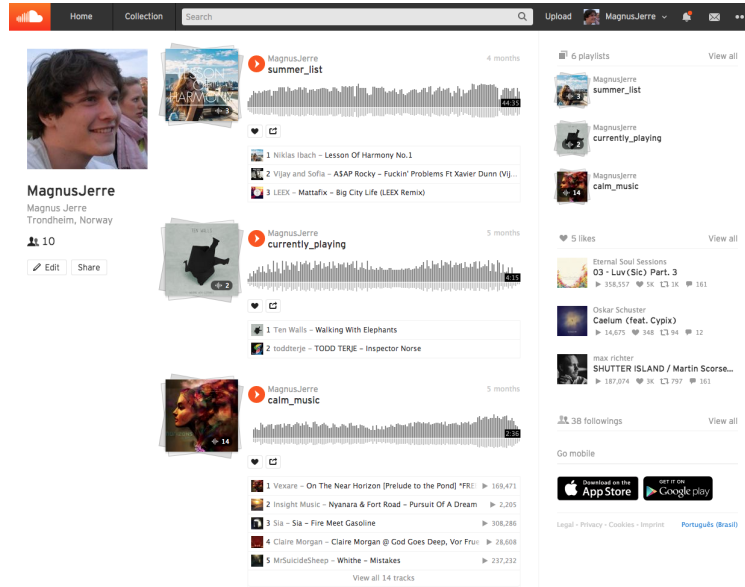


Figure 4.17: A profile page for a Soundcloud user

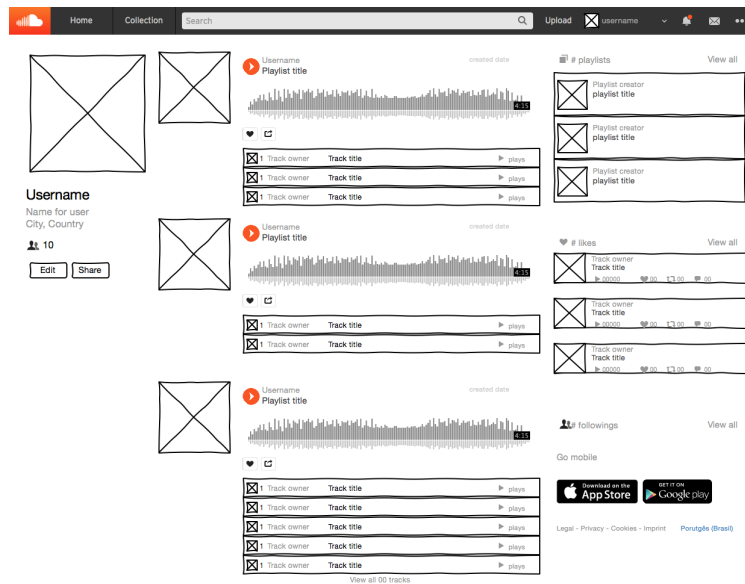


Figure 4.18: The wireframe prototype of the Soundcloud profile page

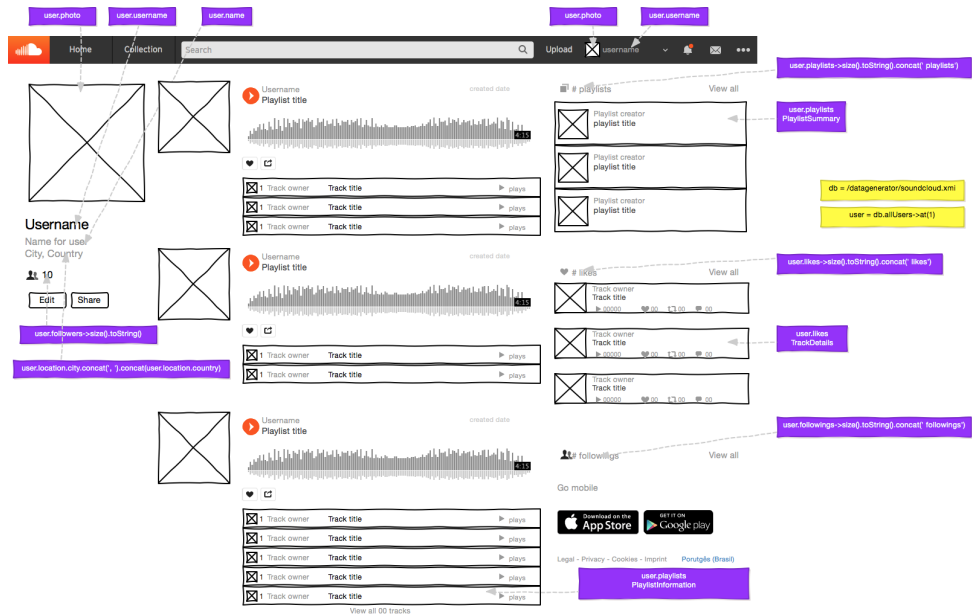


Figure 4.19: The annotated screen for the Soundcloud profile page

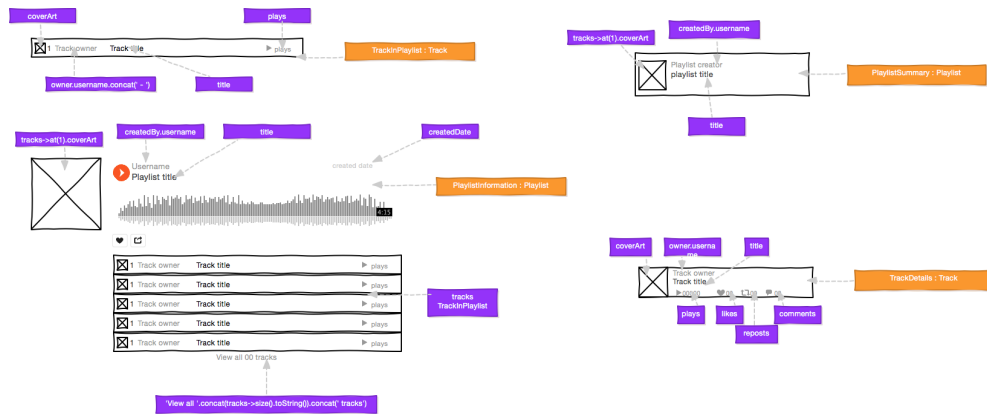


Figure 4.20: The view component annotations for the Soundcloud profile page

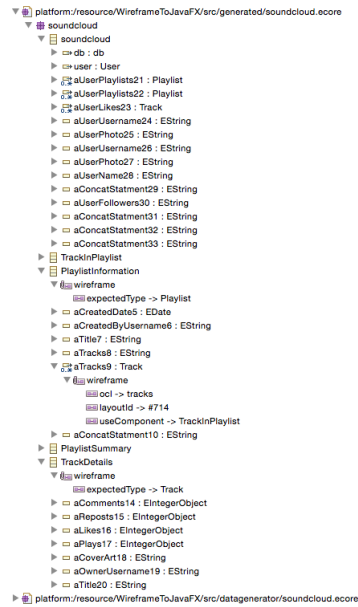


Figure 4.21: The generated ecore file for the annotated Soundcloud prototypes

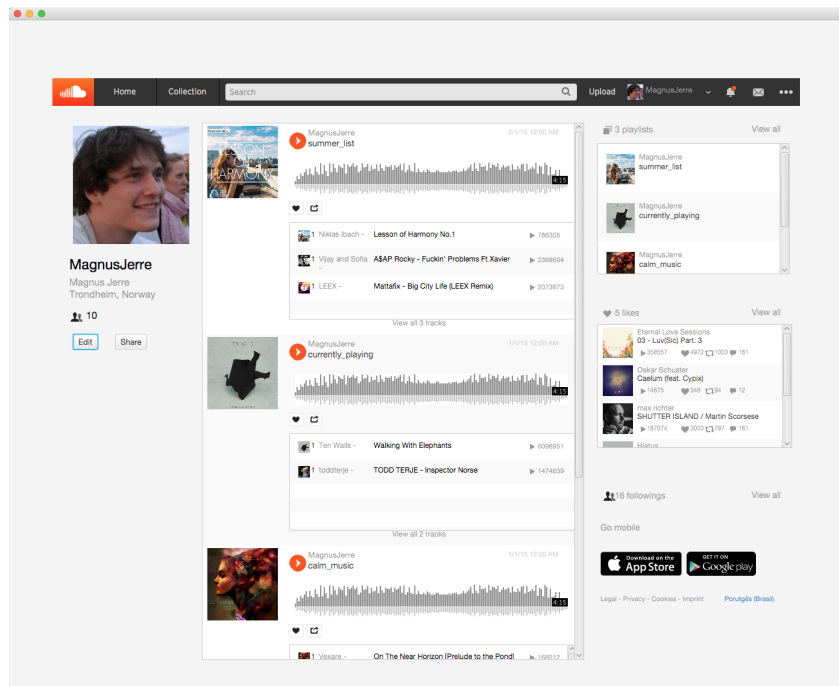


Figure 4.22: JavaFX application for the Soundcloud profile page

Chapter 5

Discussion

5.1 Test Results Discussion

The three scenarios illustrate both the extents and limits with populating the prototypes with data. All three scenarios are data intensive, therefore their fidelity is greatly increased when populated with real data. The most tedious part of developing the prototypes was creating the wireframe model for each of the applications as well as their data models. Both of these artefacts are assumed to already exist when the time to annotate the wireframe prototype comes which is why that phase will not be part of the evaluation of the test results.

Correctly annotating the wireframe model and grouping the view components is both fast and straight forward. The first scenario only contains simple view elements that can be directly bound to data, totaling at 19 assignments and 3 contexts. Drawing the arrows from the assignments is very easy for the larger and isolated view elements as there is no question exactly which view element they point to. Unfortunately this is not the case for smaller and more densely located view elements. Since the labels for each small text element is significantly larger than the actual text inside it, some of the labels will partly overlap making the arrow heads sometimes point to the wrong view elements. The center part of the screen containing the rating score and related elements borders on being too small for annotation using arrows. The height of the labels is defined by Wireframesketcher and can't be altered, therefore using this application for prototypes that will have a lot of small text elements close to each other is not guaranteed to work correctly.

Formatting Data Correctly

Most of the view elements can display the data they are given in the way intended by the developer since they are already in the correct format, such as the ratings and the movie title. A problem arises when the data is not in the correct format such as the release date for the movie. Next to the title the release year should be displayed, the date is however not stored as a year only, it is stored and represented as "11/7/14 12:00 AM", which is clearly not the format intended. One way to overcome this problem is to create a custom definition of a date which stores the date as separate year, month and day, thereby making it easy to access the year value only. It should be noted that the application does what it's supposed to do, the assignment correctly retrieves the data for the view element using an OCL statement. OCL however isn't a programming language designed for formatting output, this must be left to some other kind of language and input. Another formatting difference can be seen for the number of reviews, the source formats the number as "2,327" while the prototype displays it as "2327". Adding support for text formatting will help with the fidelity but is not necessarily a part of binding data to view elements. The advantage with displaying the data as is, is that it illustrates which parts of the application must format the provided data.

View Components Reduces the Amount of Work

The second scenario displays more data than the first one, yet it needs fewer assignments, only five for the actual screen and another ten assignments for the view components. This is due the introduction and use of view components. Using view components can greatly reduce the amount of assignments needed as long as the view elements can be grouped into view components and is reused, otherwise the total number of assignments will increase. This is not to say that it is definitely a bad idea to use view components even when they won't be reused since the main screen will become less cluttered. The rating part of the first scenario could for instance be extracted into a view component, reducing the number of assignments cluttering the main screen.

View Component Implementation Issues

Even though the introduction of view components is beneficial, the way they are implemented should have been done a little differently. Currently each view component can only be used by the screen it's defined inside of, thereby reducing the efficiency with which they can be used. The choice to make them

local to the screen was made due to ease of implementation, it does however seem like this was a bad choice. For a complete prototype application, some view components might be reused in several screens. Having to redefine the same kind of view component several times seems counter intuitive. In addition, each view component will have an fxml file generated for it. Instead of having just one fxml file for a view component, each view component that should be the same view component will have a separate fxml file generated for it. This is not exactly optimal if one would want to use the generated files as a basis for the final implementation.

Little Need to Modify the Wireframe Model

The third scenario displays even more data than the second one as well as displaying it in several ways. The total number of assignments for the entire screen and view components has been bumped up to 33, far exceeding the first two scenarios. Still, the amount of time spent annotating the different view elements and view components is quite low compared to the time it took to create the wireframe models. Even though the screen is quite complex with several lists and view components, not a lot of changes needed to be made to the wireframe model to accommodate data binding. Initially, each view element in the wireframe model is not part of any group, however to use view components, each view element part of a single view component must be grouped together. This process is fairly quick in Wireframesketcher, all that needs to be done is selecting the correct elements, right click and select "group". Once the view elements have been correctly grouped, one of each kind of view group is duplicated and moved someplace else for view component annotation.

Each part of the screen that is supposed to be a list must also be grouped and given the property name "list" using Eclipse. Again, the grouping process is straight forward. Since the application treats each list with a specific view component provided through the assignment, it doesn't matter what kind of elements are inside the list group. A list group can in other words contain nothing and still be populated with view components, the only thing the list group will affect is its physical size on screen. This kind of decoupling makes it fairly fast to switch out the view components that should be displayed in the list. A downside to this solution can be seen in figure 4.22. Each list inside the major list has a fixed size instead of snapping to the smallest size possible, this is one of the downsides of not providing the possibility to further describe how the list should behave. In addition, each element part of a list must use the same view component, forcing all elements follow the same exact design. The problem with adding such functionality is that the

prototype creation borders closer and closer to complete programming.

Other List Functionality

An issue that can go by fairly unnoticed is the numbering of list elements. Looking at figures 4.16 and 4.22, some of the elements part of the same list have the exact same element number, the number 1. Looking at the sources this is not how it should be, element number one should display the number "1", element number two should display number "2" and so on. This number is however not part of the underlying data, rather a part of how the underlying data is sorted. What this means is that the movie "Interstellar" doesn't contain a data field for its position inside the top 10 list, this number must instead be calculated based on its position inside the list. Storing its position as a data field makes little sense as the movie can be part of several lists and also have its position changed, having to update this position each time something changes makes very little sense. Adding support for this kind of functionality is not strictly related to binding data from an external data source, it would however be a feature that is nice to have as it is something that can be very useful for lists that rank its data in some way.

5.2 Areas of Use

This thesis has been focused on enabling a prototype to use data from an external source, much like an actual software application would. Naturally, the tool developed is suitable for creating prototypes reliant on displaying data in a static manner such as the examples presented in section 3.1. As long as the data is available in the correct format, whether it already exists or must be created, the tool should work well.

The tool will however fall short when it comes to creating prototypes for dynamic applications reliant on data since the implemented functionality is focused towards static screens rather than dynamic ones. Using the prototyping tool to prototype non-intensive data applications, such as Photoshop or Word will not prove fruitful as the data features of the tool won't be used.

5.3 How Can Real Data Enhance a Prototype

This thesis has provided a look into how a prototype can be populated with data, but so far not much discussion has been made on how using real data can be useful.

Using real data with a prototype can be handled in two ways, one is to manually enter all relevant data, the other is to populate the prototype with data using something like this thesis' proposed method. The problem with manually entering data is that it's time consuming and that the format of the displayed data might automatically be correct because it is made by a person, rather than being read from an actual data source. A major advantage with binding data to a prototype is that the format of the data will be dependent on the underlying data, exposing possible issues with the data format. An example of this would be that the text string can be too long for a textfield and should therefore be cut at some point with trailing dots, or a long name where the middle name should only be represented by the first letter and a dot. Even something as simple as a date being stored as a string rather than a date, can prove problematic when working with data view elements, the "Top 10 Movies of 2014" test scenario illustrated this.

Exposing issues like the ones mentioned will also help with understanding how suitable the underlying data model is for the intended design, and thereby how to overcome possible issues in using the data model with the prototype. If the underlying data model is designed from scratch with the design prototype in mind, making sure their formats work together will be easier.

If the prototype is developed for a customer, being able to use the customer's data can help them become more engaged in the development as the product will be more directly relatable, rather than displaying some generic data. Having a more engaged customer can help with communication, especially surrounding features wanted, and in the end result in a product better suited for their needs.

5.4 Data Model Complexity and Prototype Creation

This section discusses how the complexity of the model can affect the prototype creation stage.

A very simple data model will not need complex statements in order to correctly navigate the data model, the movie example used throughout this thesis is an example of just that. The statements will be short and easy to read, making the annotation boxes small and thereby keeping the screen file relatively free from clutter.

A more complex or poorly optimized model can however result in long and complex statements that can be hard to understand. Figure 5.1 illustrates

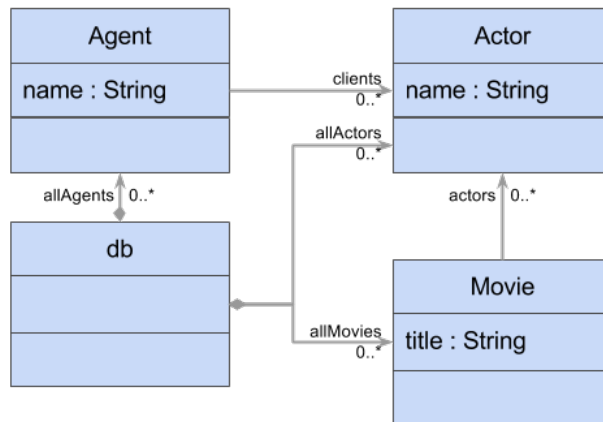


Figure 5.1: Poorly designed model

a poorly designed model having only unidirectional relationships with the actor. In order to get a collection of all the movies that agent number one has had clients star in, one would end up with the following statement:

```
allMovies->select(
  actors->asBag()->intersection(allAgents->at(1).clients)->size() > 0)
```

The statement is kind of difficult to understand as it stands by itself. It's even more difficult to actually create a valid and correct statement. If there had been a bidirectional relationship between the actor and movie objects, the statement would be a lot simpler and look like this:

```
allAgents->at(1).clients.movies
```

Even if the underlying data model is complex or poorly optimized, the screen file can still be kept quite clean since the annotation boxes can be resized without the text content being altered. Therefore using complex data models will mostly affect the complexity of the statements. Making sure the model is suitable and that a suitable navigation language for the specific model used is paramount in order to make the statements as easy to create and read as possible.

Working with complex models might not be possible to avoid, therefore providing a more dynamic development environment can help make the work easier. By allowing the decorators to display their statement results upon hovering or providing a clickable way of selecting the desired data rather than manually writing statements can help ease the work flow. Alternatively generating a new and more suitable data model based on the original one might be a better solution.

5.5 Does the Method Scale

This section discusses how the proposed method for populating a prototype with data scales in terms of large and advanced screens.

As illustrated in the examples from section 3.1, a lot of the elements displayed on-screen share the same kind of layout. Because of these similar layouts the need for something that enabled reuse of view components was deemed necessary. By reusing view components, large and complex screens can be kept fairly clutter free, especially when the same view components is reused several times. Even if all the groups of view elements use a unique layout, it's possible to create view components for each type of layout in order to minimize the number of assignments that are directly on top of the relevant screen area.

The way the software is currently implemented, means that each view component created is only usable by the screen it's defined in. This however battles reuse and will make the method scale less well than what it should. It would probably have been better to allow view components to be reused across screens, making the total number of existing view components lower than what is currently the case. This way unnecessary duplicate versions of the same view component can be avoided.

5.6 Expanding The Functionality

Due to limited time, some features that would be nice to have in order to make the prototyping tool more feature rich were set aside. As stated in section 4.1.3, only the read operation would be implemented, in addition, features such as filtering were omitted. Adding support for these features will undoubtedly increase the prototype's fidelity if the alternative is to not provide any such features for the usability tests. The same goes for enabling modification of the underlying data during usability tests. Having to manually create variations of the underlying data to accomodate any predefined modification scenarios can be quite time consuming. Time spent creating these data variations might be better spent elsewhere in the prototyping or even development process. Therefore, adding functionality for both filtering and modification can be justified.

An important question to ask is when should one stop adding new features to the tool. As more and more functionality is added to the tool, the prototype gets closer and closer to a complete version of the actual product. Providing extra functionality however comes at the cost of increased complexity and time spent on developing the prototype rather than implementing

the actual application. Another issue with expanding the feature set is how these features should be utilised. Using the current annotation method to incorporate new features can lead to screen-files overflowing with decorators thereby negating the readability they provide.

5.7 New Method for Developing Software

This section will discuss how the proposed method for populating a prototype with data in combination with Fredrik Larsen's work could be used as a new method for developing simple software applications.

The method for populating a prototype with data proposed in this thesis provides a rapid and robust way of building runnable prototypes. Combining Fredrik Larsen's implemented functionality with the implemented data binding functionality results in a simple piece of software. Starting by creating the application's interface then annotating it with functionality and data is a fairly simple process to understand. This process can be seen as a form of programming using gui elements. The threshold for developing a simple application this way is much lower than having to manually code up all the gui-elements as well as code the functionality and data binding routines.

Even though developing a standalone application this way is fast, the functionality currently implemented is not powerful enough to provide the developer with full control over the layout and actions taken. Adding support for modifying the existing data source and filtering through search will vastly improve the range of data dependent applications that can be developed this way.

Writing a new piece of software based on the same principles as those proposed by this thesis and Fredrik Larsen can be used as the base for a new software development method. Important aspects for such a piece of software would be to provide a more dynamic development experience than what is currently available using a combination of Wireframesketcher, Eclipse, EMF and JavaFX.

5.8 Reusing the generated Code to Create Software

This section discusses how the prototype's generated code can be used as part of the final software implementation.

The code generated as part of the prototype follows an intuitive structure which can be used as a starting point for any further implementation of the

prototype. As stated in section 4.2, each of the screens will have a generated ecore meta model for it, containing all relevant elements for the screen. In addition, an fxml file for each screens' layout is generated. These two types of files can be used as either an indication of necessary fields or as a source for any later implementation.

Even though the generated code is well structured, it is not currently optimized for human readability. The variable names for each assignment field follow a very simple pattern which is not necessarily unique enough, in addition all variable names have a trailing number, making the names a little hard to read and not great for later manual code creation. Allowing the developers to specify a variable name for each of the assignments can help overcome this issue. The fxml-files are not currently suited to be used as the source for any later implementation because they are created in such a way that it's easy to programmatically add fxml elements where they are supposed to be instead of actually containing the fxml elements. In order to base any further development directly on the generated code, the two problems adressed should be handled.

Chapter 6

Conclusion

The main goal of this thesis was to explore how a wireframe prototype can be populated with real data by adding a little extra information to the prototype. This goal has been achieved through looking at how existing applications display their data, leading to the definition of requirements and a proposed method for describing what data should be displayed where.

The method for describing where data will be displayed builds on Fredrik Larsen's idea of using decorators containing a description of what should take effect. Three new decorators were developed, namely *assignment*, *context* and *view component*, in order to provide a foundation for populating the prototype with data. A key feature was to allow the developers to define a template for how a set of data is displayed using the view component decorator, which very much resembles how object oriented languages allow the definition of classes.

The application developed closely followed the planned method for populating the prototype with data. Its implementation is based on using EMF to create and store the relevant annotation data. Each Wireframesketcher screen being populated with data have a view model generated for it that is used when running the prototype. This view model is populated during runtime using the underlying data model.

The advantages in using real data are many, among them is the increased fidelity the prototype will have. However, the power of populating the prototype with data is most evident when it's done by referencing the data rather than manually inputting and possibly selecting optimal data to display. Issues such as the underlying data's format and its compatibility with the planned application can in this way be exposed.

Chapter 7

Future Work

Given more time, at least two extra features would be implemented, namely filtering and modification of data. Even though both features can be circumvented by constraining test cases, they can prove valuable in order to speed up the prototype development time compared to making new data models or screen files.

Displaying the exact underlying data works fairly well at this point, however not all data is displayed as intended by the designers. An example of this is visible in figure 4.16 where the date displayed should only display the year and the numbers should have commas for every third digit. Expanding the functionality for the existing Style decorator created by Fredrik Larsen could be a possible route to go in order to achieve this.

Developing prototypes using the prototyping tool created is currently a three-step process. The first step is to annotate the wireframe model and save it. The second is to start the file generation process followed by manually refreshing the folder containing the generated files. The third and final step is to run the generated prototype. Cutting the number of steps down by forcing Eclipse to automatically generate the necessary files and refresh the correct folder when saving the annotated wireframe model will make the process a lot cleaner and much more efficient.

Appendices

Using the Prototyping Tool

This section provides a short explanation on how to use the code created as part of the thesis.

Getting the Code From Github

The code created as part of this thesis can be found on the following address <https://github.com/magnusjerre/MasterThesisCodeFinal>. The commit representing the final version for the thesis is named *Updated README*, was committed on June 13, 2015 and has the following sha:
2ecd8ffde9ac7aaa1cfc6aa7c830a3d965cddc61

Using the Prototype Tool

In order to use the prototyping tool with test scenarios from section 4.3, the following three steps must be completed:

1. Locate the package named *generator*, then the file named *Generator.xtend*, right click and select run as Java application.
2. Refresh the package named *data generated* and verify that the new files generated are visible in Eclipse.
3. Locate the package named *application*, then the file named *AppController.xtend*, right click and select run as Java application.

In order to change which Wireframesketcher project to generate a prototype for, locate the package named *application*, then open the file named *Constants.xtend* and change the value of *SUB_PROJECT_NAME* to the name of the Wireframesketcher project to generate code for.

Bibliography

- [1] Marcelo Paternostro Ed Merks Dave Steinberg, Frank Budinsky. *EMF Eclipse Modeling Framework*. Addison-Wesley, 2008.
- [2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java language specification java se 8 edition. Technical report, Oracle America, 2015.
- [3] Cory Janssen. Create, retrieve, update and delete (crud). <http://www.techopedia.com/definition/25949/create-retrieve-update-and-delete-crud>. [Online; accessed 2015-06-11].
- [4] Fredrik Haugen Larsen. From sketches to functional prototypes. Master's thesis, Norwegian University of Science and Technology, 2014.
- [5] Horst Lichter, Matthias Schneider-Hufschmidt, and Heinz Züllighoven. Prototyping in industrial software projects—bridging the gap between theory and practice. In *Proceedings of the 15th International Conference on Software Engineering, ICSE '93*, pages 221–229, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [6] Briony J Oates. *Researching Information Systems and Computing*. Sage Publications Ltd., 2006.
- [7] Object Management Group. *Object Constraint Language Version 2.4*, 2014.
- [8] ORACLE. A relational database overview. <https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html>. [Online; accessed 2015-05-22].
- [9] Jim Rudd, Ken Stern, and Scott Isensee. Low vs. high-fidelity prototyping debate. *interactions*, 3(1):76–85, January 1996.
- [10] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 2006.

-
- [11] Carolyn Snyder. *Paper prototyping, The fast and easy way to design and refine user interfaces*. Morgan Kaufmann Publishers, 2003.
- [12] W3C. Extensible markup language (xml) 1.0 (fifth edition). <http://www.w3.org/TR/REC-xml/#sec-origin-goals>. [Online; accessed 2015-05-28].
- [13] W3C. Xml path language (xpath) 3.0. <http://www.w3.org/TR/xpath-30/>. [Online; accessed 2015-05-28].
- [14] W3C. Xml schema part 1: Structures second edition. <http://www.w3.org/TR/xmlschema-2/>. [Online; accessed 2015-05-28].
- [15] Wikipedia. Data model. https://en.wikipedia.org/?title=Data_model. [Online; accessed 2015-06-13].