Rent It

Software Development in Large Teams with International Collaboration,

Second-Year Project,
Bachelor in Software Development,
IT University of Copenhagen

Group 12
Jakob Melnyk, jmel@itu.dk
Frederik Lysgaard, frly@itu.dk
Ulrik Flænø Damm, ulfd@itu.dk
Niklas Hansen, nikl@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

May 23rd, 2012

Contents

1 Preface

This report is the result of a project on the bachelor in Software Development at the IT-University of Copenhagen spanning from the 2nd of February 2012 to the 23rd of May 2012.
The project was given on the fourth semester and corresponds to 15 ECTS-credits.

The project ("Software Development in Large Teams with International Collaboration") is described as centered on developing and implementing a rental service for digital media. The goal is to develop a client-server solution backed by a relational database system.
Students work in the teams of four to six people and collaborate and negotiate with students from Singapore Management University (SMU) in English.
This specific project was done by five ITU students who collaborated with three SMU students.

The challenges of this projects were:

- Dealing with cultural gaps and time differences in communication with team members from Singapore.
- Sharing a balanced amount of information to have successful collaboration.
- Reaching agreements that both ITU and SMU students found satisfactory.
- Conducting constructive feedback on fellow students work.
- Constructing a client/server solution in C#.

Whenever we reference the Bibliography (found on page 40), we use the signature [1] to indicate such. [1] refers to the link to a blog describing "The Way of Testivus".

1.1 Scrum tracker, version control and service info

We use Git (at GitHub) as our version control system.

In addition, we use a free tool called Pivotal Tracker to track our user stories for the Scrum[1] development model we have chosen to use.

We have set up a server to let us publish a test copy of our service (and database). We use this service to automate tests of the client and the service interface.

git https://github.com/itu-bswu/RentIt

PivotalTracker pivotaltracker.com/projects/492063

Service address http://rentit.itu.dk/RentIt12/Services/Service.svc

Test Service address http://rentit.dk:9000/Services/Service.svc

Release database http://rentit.itu.dk (username: RentIt12Db — password: Zaq12wsx)

Test database rentit.dk (username: RentIT.dk — password: Vand22kanon)

Premade user username="Smith" password="userPassword"

Premade content provider username="Universal" password="contentProvider

[1] Further detailed in our Collaboration chapter on page 10.

2 Project overview

In this chapter we discuss our interpretation of the project "Software Development in Large Teams with International Collaboration", what we feel are the important parts of the project, the must-haves of the final product and the assumptions we make going into our requirement specification.

2.1 Problem analysis

Not too many years ago, media rental of physical media was a lucrative business[1]. The last couple of years have been hard on companies making their business in physical media rental[3]. This is, at least partially, due to the increasing popularity of companies like Netflix[4] making it easier to rent media digitally, thus enabling users to do it from home and not spend time going to the actual shops.

To create a media rental service that would be seen as interesting (if not competitive), it has to be:

- Easy to use[2]
- Price competitive[3]
- (Optionally) Offer an expanded array of services compared to other services.

A wide array of media rental services already exist for books, movies/films, music and other media, so there are many sources to draw inspiration from. In addition, media rental services do not necessarily have to be run by private companies. Some institutions (like libraries[4]) offer similar services for citizens.
A service does not necessarily have to focus on one kind of media (like Netflix), as evidenced by Apple's iTunes Store[7].

In addition to subjects concerning normal users of the service, the project description also mentions administrators. Administrators can upload, delete and edit movie information on the service. Because these types of administrator users provide content, we have decided to refer to them as *Content Provider*s. We make this distinction because we have another user type called *Admin*s[5].

2.2 Assumptions and decisions

In order to narrow down the focus and requirements for our system, we make some assumptions and decisions in addition to the points raised in the problem analysis.

2.2.1 Choosing a service type

As described in the problem analysis, the existing types of media rental services can be narrowed down to a) free public library rental type and b) paid media rental. They can be very similar (depending on development choices) and both types present some security issues (user information, credit cards, etc.).

[1] Blockbuster LLC[2] is an example of a successful company in the media rental industry.
[2] Piracy is a major concern, and if the service does not provide something that is (at least) just as easy to use, people would rather be inclined to download illegally rather than pay for content from a service[5].
[3] Rarely a problem with piracy, but if one service provides the same amount of media, support and access, price is certainly a factor.
[4] Roskilde Bibliotek is an example of a danish library providing similar functionality[6].
[5] *Admin*s are explained in further detail in our Design chapter (page 14).

We decided to develop a paid media rental system, as we felt it had more possibilities (such as payment models) in terms of functionality that could be implemented. While adding payment options is not necessarily a core requirement, we felt we should design our system with payment options in mind.

2.2.2 Choosing a media type

At first we wanted to make a streaming service for TV shows. This could involve paying for a single episode of a TV show or for a full season.
After doing some research on what streaming would involve (compared to just downloading and saving a file), we changed the way we let users access our content. Instead of doing streaming, we decided to just let users download movie files and store them on their system.

In addition we changed our media type. While we felt it could have been more interesting to do TV shows (compared to other types of media), we decided to pick a slightly less complex system and instead focus on designing the service to offer movie rentals.
We decided on this less complex system, because we wanted to make a compromise with the SMU students[6] and still something that had a close relation with TV shows, but simpler.

2.2.3 Digital Rights Management

Digital Rights Management (DRM) is a concept media rental systems should take into consideration. While we may limit how long users have active rentals on the service, there is a technical challenge in making sure users cannot view the downloaded files after rentals have expired. DRM is beyond the immediate scope of our project, but we include it as an optional goal, because we feel it is an interesting functionality to have.

2.2.4 Author rights

Author rights is another concept to consider. When we give `Content Providers` the rights to upload movie files, they may be able to abuse this by uploading files they do not have author rights to. We do not consider this a central focus point in our system, but in order to ensure a great quality service in a broader perspective, some sort of validation of uploaded material should be considered.

[6] Described in our Collaboration chapter on page 10.

3 Requirements

This chapter describes the requirements (and optional features) for our system and project. We have translated the required features and some of the optional features into use cases. We create tests[1] and workflows for our system from the use cases, so that we can document and make sure that our system fulfills our requirements. The use cases are described on page 8.

3.1 Required system features

The project descriptions lists a number of requirements for the design and implementation of the service and client.

- The service must use a SQL server database.
- The system must run in multi-user environments.
- The service must be implemented in C# using Windows Communication Foundation(WCF).
- The client must enable users to access, administrate, upload and download media.
- The graphical user interface of the client must be implemented in C# using Windows Forms, ASP or Windows Presentation Foundation(WPF).

Core Features The features listed below are the core features of our system. For us to deliver an acceptable system, we feel these features must be implemented, both on the service and the client, and must be thoroughly tested.

- User
  - Create a new user account.
  - Login.
  - Rent media.
  - Edit profile.
  - Download media.
  - View a list of all movies.
- Content provider
  - Login
  - Upload media.
  - Edit uploaded media.
  - Delete media.

3.2 Optional system features

Section 3.1 described the core features of our system. We consider those features the "bare bones" of our system. In addition to the core features, we have a number of optional features.
Some of these features involve bigger design decisions than others. This means we may decide not to pick up a "High priority" feature before a "Medium" priority feature, due to time constraints or other other reasons. The optional features we have decided to implement are in **bold**.

[1] Further described in our Testing chapter on page 32.

- High priority
  - **Searching for movies**.
  - **View movielists with different sorting**.
  - **Movie release dates**.
  - **Logout**.
  - **View rental history.**.
- Medium priority
  - **Movie editions (SD, HD, Director's Cut, etc.)**.
  - Implement cost for rentals.
  - Let users rate/review media.
  - Store information (Service-end) for analytical and statistical work[2].
- Low priority
  - Stream media from the browser (no download necessary).
  - Social network integration.
  - Allow users to buy products instead of renting them.
  - Instructions and/or tooltips.
  - Age ratings.
  - User banning.
  - Trailers.
  - "Featured" movies.

High Our high priority features are largely quality of life, yet almost core features. Users are used to being able to search for what they want, so we feel this should be one of the first things we do beyond the core.
Additionally we feel that release dates gives the user more information and more ways to sort the movie lists.

Medium The medium priority features are expansions of our core features.
The implementation of rental costs is the most interesting one, yet also what we feel is the most advanced of the options. Because we want our system to handle paid media rental[3], rental costs should be a priority. On the other hand, we feel that if we go for implementing rental costs, we should also design payment options, GUI for the payment and more.
In contrast, the other options are much on the same scale, but do not necessarily have the same amount of extra design time.

Low Low priority features are nice to have but not necessary.
Integrating social networks and streaming media in the browser are cool features, but we risk taking development time away from the core features to add optional features that do not really add much to the system as a whole.
Instructions, manuals, online help and tooltips are nice usability features and they may improve the product as a whole, but if we take development time away from the core features to create these usability features, we may end up with a manual for buggy/non-functional software.

[2]We have implemented a distinction between a users current rentals and rental history.
[3]Discussed in section 2.2.1 on page 4.

3.3 Additional project requirements

In addition to our required and optional features, we have a number of project requirements to make sure we deliver a good product.

- Collaboration
  - Use a version control system.
  - Document design decisions.
  - Use an iterative development strategy.
  - Feature freeze May 8th[4].
  - Code freeze May 19th.
- Quality Assurance[5] (QA)
  - Use cases must be covered by tests[6].
  - Test code coverage must be thorough[7].
    - Minimum overall coverage of the service project: 50%
    - Goal for overall coverage of service: 80%
    - Critical sections of service: 85%
  - All code must be documented[8].
  - User interface must be usability tested[9].

3.4 Use cases for the system

These are the use cases for our system. The use cases only include requirements that we have decided to implement.

User management

- A user wants to create a new account.
- A user wants to login.
- A user wants to edit his profile.
- A user wants to logout.

Browsing media

- A user wants to view a list of all offered movies.
- A user wants to browse movies by their release date.
- A user wants to search for a specific movie title.
- A user wants to view all movies of a specific genre.

[4] After this date, no new "features" can be added.
[5] The requirements listed here are what we feel need to be successfully covered so that we can say we deliver a well tested product.
[6] Functionality must be tested on both the service and the client if possible.
[7] See chapter 8 Testing for reasoning behind numbers.
[8] XML headers for classes, fields, constructors and methods. Additional comments if deemed necessary.
[9] How we do usability testing is described in 8.1.3 on page 33.

Media rental

- A user wants to rent a specific movie edition.
- A user wants to view all of his previous rentals.
- A user wants to view his current rentals.
- A user wants to download a current rental,

Content management

- A content provider wants to register and upload a movie.
- A content provider wants to register a movie.
- A content provider wants to upload an edition to an already registered movie.
- A content provider wants to edit information about a movie.
- A content provider wants to delete a movie.

4 Collaboration

This chapter will focus on how we worked together as a group, how we worked with the SMU team, what problems we ran into and what we could have done differently.

4.1 Collaboration

When we began the project, we decided to use the agile form for development known as "SCRUM". SCRUM is a development form where the team works in "sprints". Sprints are a period of time in which the team is supposed to work on certain features. These features are called "user stories" and they consist of a name, a short description and an estimation of how much time it will take to complete the story. Each user story descirbes a feature that the team is supposed to develop. These user stories can be prioritised by the product owner, assuming that the product owner wants a certain feature finished before another.

SCRUM also contains daily meetings/stand-up meetings. In these meetings, every team member stands up and, one after the other, tells the rest of the team what they have been working on since the last meeting, what they intend to work on until next meeting and if anything can prevent them from getting it done. These meetings gives the team a good overview of what has been finished, and what is left to finish. Furthermore, it gives the team an opportunity to discuss problems that have been encountered, and how to solve these problems.

Another important part of SCRUM are the retrospectives. At the end of every sprint, the whole team meets, and each teammember writes some points that went well, and some points that did not go well during the spring. These points are all gathered, after which they are discussed by the team. After the discussion, the team decides on which points to improve during the next sprint. This allows the team to improve over time. This should lead to a better project, both in terms of finished product, but also in terms of how the team works together.

4.2 ITU group structure

Having decided upon using SCRUM for our project, we had to distribute the roles that we felt were important to our work. This meant that we had the following 4 roles to distribute:

Product Owner The Product Owner is the person/company who has ordered the product. Normally, this would be a specific person or company, but in our case that isn't entirely true. Technically, our lector is the one who has "ordered" the product, which normally would make him the product owner. However, we also had the Singaporeans to take into account. Considering that they should use the product as well, they also had a say in how the product was supposed to work.

In the end, we decided that the Product Owner should be a combination of our lector and our Team Leader. The reason behind this is that they are the ones who decides whether the product lives up to the demands or not.

Team Leader The Team Leader has some responsebilities. His job is to make sure that the team finishes their tasks on time, make sure that the team shows up on the agreed time, make sure that the team is functioning well, etc. The Team Leader is basically responsible for the whole team. If the Product Owner has a problem with regards to the team, he should contact the Team Leader and let the Team Leader take care of it.

In our team, we had two candidates for the Team Leader role; Niklas Hansen and Jakob Melnyk. After some talking, Jakob Melnyk decided to let Niklas Hansen get the Team Leader role, as it turned out that it was not something he was very interested in.

SCRUM Master The SCRUM Master is responsible for ensuring that the SCRUM process is used as intended. A key part of his role is to keep the team focused on the user stories, and make sure that the team is not distracted by outside influences. In our group, we also decided to let the SCRUM Master control the daily meetings, the sprint plannings and the retrospectives.

For our SCRUM Master role, we decided upon Frederik Lysgaard, as he was the one who showed the most interest in the role.

QA Responsible The QA Responsible is responsible for looking at stories that have been finished, and making sure that they work as intended. The QA responsible is the one who decides if a user story has been finished or not. This is done by running tests that have been written for the story and making sure that the tests covers the user story well enough.

The QA role was given to Jakob Melnyk because he is very good at making sure that things work as are they are supposed to.

The second thing we had to do with regards to roles, was to decide whether we wanted the roles to rotate on a certain basis, or let people keep their roles until the end of the project. After some discussion, we decided to keep the roles static. Had we changed them every so often, it would cause confusion for both us and the SMU team, which is why we decided to keep the roles static.

4.2.1 Meetings

As mentioned in the descruption of SCRUM, meetings are an important part of SCRUM. This meant that we had a lot of focus on meetings, which will be discussed in this section.

In the beginning, we decided to meet on Mondays from 12.00 to 14.00, along with Tuesdays and Thursdays from 12.00 to 16.00. After finishing our other subjects, we decided to meet every day, except Friday and Sunday, from 10.00-16.00. When we met, we would start off with a stand-up meeting. during the meeting, we told each other what we had been working on since the last meeting, and what we intended to work on until the next meeting. After the stand-up meeting, we would begin working on our tasks.

Stand-up meetings weren't the only kind of meeting we had. Every second Tuesday we would hold a retrospective meeting, in which we would discuss how the sprint had gone, and what we should improve on. Using these meetings, we kept improving our work, both with regards to how we worked, but also with regards to the quality of our product.

The last type of meetings we used, were the sprint planning meetings. On these meetings we would look at new stories, estimate them and prioritise them, according to which stories our product owner wanted to have finished first. We would then assign the stories to team members, after which we would working on them. The sprint planning meetings were also used to look at our status, see how many stories we had completed in the previous sprint, and how many that were left. The ones left from previous sprints would be prioritised higher in the next sprint, so we could have them finished and begin working on new ones.

4.3 SMU cooperation

Working in teams on the ITU side wasn't the only form for teamwork that was to be done in this project. It was planned that we should collaborate with a group from the Singapore Management University during the project. This meant that another "dimension" was added to the project, as we suddenly were to communicate with people whom we had never met, and whose skills we knew nothing about. It turned out to be more of a challenge than expected, as we learned during the course.

We were introduced to the SMU team on the 6th of March. During this meeting, we agreed on using Google+ Hangout (a video conference tool) as our method for communicating during meetings. For communication that did not relate to the meetings, we agreed on using email, as it is an efficient tool for communication. It also has the advantage that everything that is sent back and forth is documented, and thus can be looked at at a later date.

4.3.1 Meetings

Our meetings with the SMU team were scheduled to be every Thursday around 13.00-15.00. We had a total of 5 meetings with the SMU team, in which we would update each other on how we were doing. After the update, we would talk about what our plans were until the next meeting. The meetings were also used for sharing ideas about what both groups wanted the service to be able to do.

4.3.2 Conflicts

Working with the SMU team was quite a new experience for us, as no one from our team had worked with a team from that far away before. The only expetations we had, were from what we had been told during the lectures. Therefore, we had hoped that working with the SMU team would be relatively painless, and it looked like that was the case at the beginning. But as time went by, we ran into different problems, which will be discussed here.

Mood Changes

The use of Google+ Hangout as a video conference tool improved our communication during the meetings. Unfortunately, it did not mean that we were able to predict some of the "mood changes" that the Singaporeans had. During our meetings we would agreed with them on something, and the next day we would receive a mail saying "can we do it this way instead? ", with their suggestion usually being something completely opposite of what we had agreed on. Whenever this happened, we would end up having an email conversation with the Singaporeans, and in the end we would come to a solution that both sides would agree on.

Wrong API

Another problem we encountered, was that they had spent some time looking at an API, before they sent us an email with questions about it. When we received the questions, we had no idea what they were talking about. Somehow, they had managed to find an API that wasn't ours, and they had been looking at that one instead of ours. This lead to confusion and a mail conversation, but in the end we managed make them look at the correct API.

Misconception

This is not to say that they created all the problems. On our end there was some misconception with regards to what they were capable of, when it came to programming. We thought that they were about our level, but it turned out that they weren't. This meant that some things were not done correctly on their end, which resulted in extra work for both ends.

Behind schedule

We also had some problems on our side, which impacted their side. For example we had some database issues at the beginning of the project, which prevented us from dooing much for a week. The database issue put us behind schedule, and because of that, the Singaporeans were put behind schedule.

Error reporting

The fact that the Singaporeans were put behind schedule turned out to be a major problem, as they were rather slow to report when they encountered problems. Towards the end of their schedule, they were unable to make our service work for them, and they didn't tell us until they were approaching their own deadline. We managed to solve the issue, though it could have been handled a lot better and faster if we had received their report earlier.

4.3.3 What we could have done differently

The problems mentioned in section 4.3.2 can be said to have happened because of one problem: Bad communication. Not only did we not have enough communication between the team teams, but our communication was not very clear. Because our communication lacked clarity at times, we had a few cases of complete misunderstandings, which took a lot of extra communication to figure out.

As stated at the beginning of this chapter, we used Scrum on the ITU side, and from our second meeting with the SMU team, we thought they were using Scrum as well. However, during the process, it felt like they were using another form for Scrum than we were, if they were even using Scrum at all. They wanted to implement the full service at once, instead of working on it in incremental steps.

If we had spent some more time talking with the SMU team at the beginning of the project, we would have been able to figure out how exactly they were running their part of the project, and we could have explained how what development model we were using. This way we would know what to expect from each other, and we would have an easier time figuring out how to help each other when needed.

While clear and verbose communication is, in our opinion, to be preferred, more communication could also have helped resolve the issues with bad communication. The group as a whole did not have communicate that much with the SMU team. It was mainly our Team Leader (Niklas Hansen) and our QA Responsible (Jakob Melnyk) who took care of the communication. This meant that some things were lost when other team members had some input on a subject, due to human errors.

We should, over the whole course, have been more adamant about receiving regular updates from them. Sometimes we would hear nothing from them and when they actually replied to status updates, we did not get much insight into what their status actually was. Had we been more adamant about receiving updates, we could have handeled a lot of problems earlier, easier and faster. But that is an experience we can take with us for our next project.

5 Design

5.1 Database

A good designed database can help out a lot during development. If adding foreign keys and unique constraints, the integrity of the data in the database will be high, and we can trust that data when implementing our service. We do not have to worry about whether or not a movie still exists, when we receive a movie ID from another table.

This is why we decided to put a lot of effort into our database, as we knew we would benefit from it in the long run.

5.1.1 Analysis

Decisions

When we started designing our data model, we had to decide on how much data we want to contain in our databases. We could go all-in IMDB-style, and keep information about actors and cast of each individual movie. Or we could do the exact opposite, and only keep the relevant data for the user to identify a movie.

We decided not to include actors, as IMDB provides this functionality brilliantly for free. Of course it would be nice to have these information, so that our users would not have to use two different services, to get the job done. But this is not crucial for the service to work properly, so we decided to just include enough information for the users to identify the movies, and if they need more information than that, they can use IMDB.

We decided to focus on simplicity and feature completeness in general, instead of adding a lot of half-done feature and/or untested functionality. In our experience it is better to have a program with limited functionality that does what it is designed to do quite well, instead of having a lot of features, which are not finished and are not properly tested. This of course meant a very simple data model, that would evolve over time, to only fit those requirements we ended up implementing properly.

ER-model

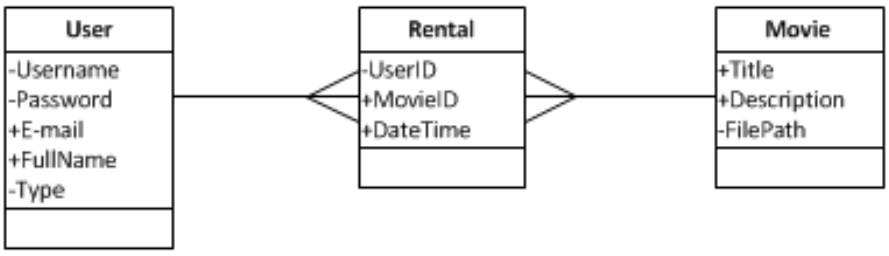

Figure 5.1: Initial datamodel

Figure 5.1 displays our simple initial data model. This captures the basic information about users, such as username, password, email address and the user's full name. We also have a field called type, representing whether a given user is a normal user, a content provider or a system administrator.

To begin with we also only wanted to capture the basic information about the movies, like title, description, genre and the filepath. To finish it off, we added a table for capturing movie rentals. This basically was a junction table, with references to the user renting the movie, the movie being rented and the time of the rental.

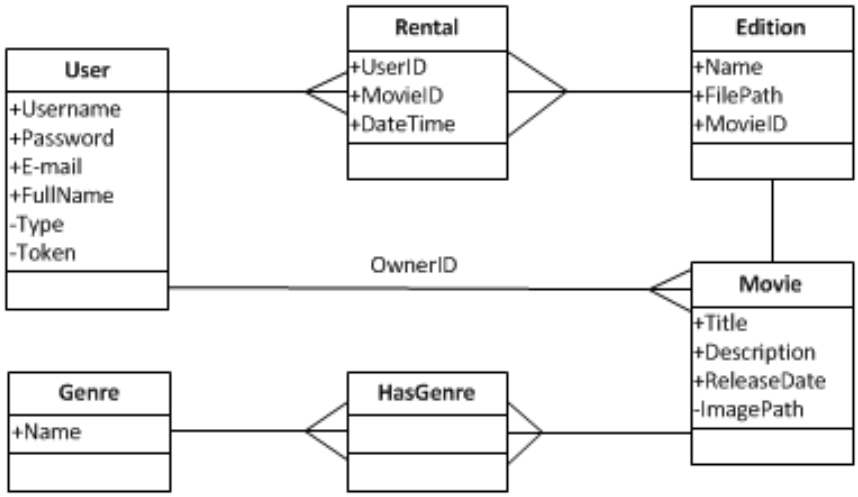

Figure 5.2: Final datamodel

Since we were using scrum and were working in sprints, our data model was constantly evolving. Figure 5.2 shows our final data model. We only added elements to the code and data model when it was needed. This of course means that our data model changed quite a lot over the time of the project, where we could probably have designed it up front and let it be throughout the project. But since it can be hard to figure out what features we actually wanted to implement two months into the future, we did it this way.

The most important changes are the Genre and Edition tables. The problem with genres beforehand, was that in order to add multiple genres to a movie, they must all be added to the same string, split by a predefined delimiter. Besides that, it was complicated to add and remove genres, and genres couldn't be re-used. That's why we separated it from the Movie table, and joined them by a junction table called HasGenre.

We decided to make it possible to add several editions of a movie (like SD, HD, Director's Cut and so on), and these should have their own files. That's why we made the Edition table and moved the filepath from Movie to Edition. We also changed the Rental table to reference a specific Edition instead of a Movie.

SMU involvement

The SMU team did not impact our data model a lot directly, as the database is an element behind the scenes of the service, and the data model does not directly affect a client team (such as SMU), as long as everything works as it should. But to be safe we did send our data model to them, for them to review and give ideas.

One of the suggestions, that we actually ended up implementing, was the release date. They suggested adding a release year, and after a little talk back and forth, we ended up expanding the idea. We ended up adding a release date to movies, and decided that movies with a release date in the future could not be rented before that date, giving greater flexibility to content providers.

They also suggested adding a price to movies and adding an end date for rentals. These suggestions were put in the product backlog, to be implemented if we got the time.

5.1.2 Tables

User

user_id The user's unique ID number.

username The user's unique username - used for login.

password The user's password - hashed and salted.

email The user's email address.

full_name Full name of the user.

type User, Content provider or System administrator.

token Unique session token generated at login and cleared at logout.

The User table contains data about our users. To provide some security to the passwords, we salt and hash (with the SHA512 algorithm) the passwords before putting it into the database. This means that if the database is hacked, then, unless they know the salt, they will still have a hard time figuring out the password of the users. The value in the password field will be useless to them, so they only find out the usernames. In order to figure out the passwords, they will need to try and login to our service with all possible passwords in a bruteforce attack. The problem comes up if they get access to our codebase, as they will also get access to the salt. This will make them able to bruteforce the passwords locally (and not using the login feature on our service), which will be much faster.

The "Type" field is an integer (since SQL Server 2008 does not have enum support) between 1 and 3. A value of 1 means that a given user is "just" a normal user, where a value of 2 means that the user is a content provider. A value of 3 indicates a system administrator.

The "token" field is a session token. We did not look much into the different WCF bindings, but we found a binding with streaming support, but with no session support. So we created our own sessions, by generating a session key upon login. This session token is then stored in the "token" field, and is cleared upon logout. This session token has to be provided at every service call (that requires the user to be logged in).

Movie

movie_id The movie's unique ID number.

title Title of the movie.

description A more or less detailed description of the movie.

owner_id Reference to the user creating the movie.

release_date The release date when the movie is available for rental.

The Movie table is quite straight forward. A movie has a title and a description to help identify the movie. The "owner_id" is a reference to the User table, to a content provider that created that movie. The release date is quite important. We took a decision to view all movies (also the ones not yet released) to the user, but it is only possible to rent a movie that has been released, and a movie has only been released if the release date has been set and is a time and date before the current time and date.

Edition

edition_id The edition's unique ID number.

movie_id Reference to the movie the entry is an edition of.

name Name of the edition.

file_path Relative file path to the video file.

An edition belongs a movie, and is referenced by the "movie_id" field. An edition can only belong to one movie, as it wouldn't make much sense to share editions between movies, as an edition has its own file path, and these are not shared between movies. The file path is a relative path, based from the movie root folder. Currently it is only a file name, but it is called "file_path" if this was to be changed later on.

Genre

genre_id The genre's unique ID number.

name The name of the genre.

We changed this to be its own table, when we discovered that a movie very easily can be of several genres, and instead of putting all these together in one string with a delimiter in between, we decided to move it to its own table. It also makes it easier to re-use genres, which will make it easier when searching for a specific genre, as it is quite easy to misspell a genre. So when the most genres already have been added, and the system suggests genres for a given movie upon creation, genres are re-used, which makes it easier to browse genres.

HasGenre

hasgenre_id Unique ID.

movie_id Reference to a movie in the Movie table.

genre_id Reference to a genre in the Genre table.

Junction table used to associate genres with movies. Contains references to a specific movie (by the "movie_id" field) and a reference to a specific genre (by the "genre_id" field). This is a many-to-many relationship, as a genre can belong to multiple movies (which is the reason we did it this way), and a movie can easily have multiple genres.

Rental

rental_id The unique ID number of the rental.

user_id Reference to a user in the User table.

edition_id Reference to an edition in the Edition table.

time Time of rental.

This is probably the most important database in a rental system: the tracking of rentals. This has been made quite simple. We capture the user renting the movie edition, the movie edition being rented and the time of rental. As our rental period is 7 days, we don't need to save the end time of a rental, but if we were to make it more dynamic, this would be a field to add.

5.1.3 Entity Framework

We used Entity Framework version 4.1 as an ORM (Object-Relational Mapping), which is Microsoft's ORM to compete with NHibernate and the like. With Entity Framework it is possible to query data with LINQ (the so-called LINQ-to-Entities), but where Entity Framework really shines, is that it focuses on code over configuration. There isn't any need to do a lot of configuration to get it working.

There are four different ways of getting started with Entity Framework, where the two last ones mentioned below are more or less the same approach, but in a different order:

Model-First Drag and drop. You get a visual editor where you can create new boxes (which will end up as tables and entities) and put lines between them, to model how the different entities interact with each other. An XML-file is created automatically, and the database and entities are created from this.

Database-First The other way round compared to Model-First. You start with creating the database, and then Entity Framework creates the XML file from the database. Then, just like Model-First, entities are created.

Code-First With code-first you start out from the code. You create your own entities, which are a lot simpler than the generated entities in the two previous modes. These are called POCO-classes, which means "Plain Old CLR Objects", referring to the simplicity of these classes. When the POCO entities are created, you get Entity Framework to create the database for you.

Code-First The final way of doing it is also called Code-First, but you start out with the database. Code-First does not refer to what component you start out with, but it means that you are "code-centric". Just like Database-First you start out with creating the database, and then use a simple tool for Visual Studio to generate the POCO entities from this database. This is also nicknamed "Code-Second".

We chose to use Code-Second, as we really wanted to get the simple POCO entities. The reason being that we wanted to send these entities back and forth over the service to transport data, but without any trace of Entity Framework. The reason we chose Code-Second over Code-First, is that we have more control over the database with Code-Second. Entity Framework does not add any indices or unique constraints by itself, so to get these, we had to create the database ourselves.

This has proved to be a bit more difficult than first expected, as no-one in our team had experience with SQL Server 2008, which is different from MySQL in certain areas. If we used Model-First, anyone in the team could have changed the data model, where as with Code-Second either the entire group had to read up on SQL Server and T-SQL or one team member should be responsible for the database part all by himself. We chose to pick one person as the database responsible, to make sure the rest of the group could continue working on the service in the meantime. This meant that only that person could alter the data model, but we did get complete control over our data model and database, which we felt was a big advantage.

5.2 Service

5.2.1 Analysis

Our service has two layers: the service layer and the logic layer. This is important when we have a public API that other developers can call. We do not want other people direct access to all our logic, but only what we expose as an interface. Our service layer is that interface. It is a wrapper around our logic layer, that validates input before calling an actual method. In our logic class, we may have methods that fails on invalid input, by throwing an exception, or worse, unexpected behavior. That is fine for us to use, as we have scenario tests to ensure that we are not breaking internally. It is a bigger problem when having a public API, since we ca not tell people not to call certain methods in certain ways. We have to validate all input from a client, and that is why we have the service layer on top of our logic layer. The service methods has

a standardized way of telling the client, that something went wrong, and will use this, if it detects invalid input. If the input is valid, any error will be a bug in the logic layer, which should not happen if we have enough scenario tests to cover all common use cases.

The logic layer also handles the connection to the database, which is something that should never be exposed to anyone, but should be part of the private code. Having a security leak, where a 3rd party could get access to our database in any way could be catastrophic, since they could either destroy data to lay down the service, manipulate data for their own interests, or get access to hidden information, such as unreleased movies. That is why the database connectivity should be at least a few layers of abstraction away from anything that is publicly exposed, and that is what we are doing: the database connection itself is handled by Entity Framework, and while we use entity framework directly in the logic layer, we have some abstraction in the layer itself, which means that we mostly does not use it directly anyways. And this layer is still wrapped by the service layer, so that gives a lot of abstraction towards the database, and it should be safe.

Our tests are also split up into scenario tests for the logic layer, and service tests for the service layer. The scenario tests should check that the logic methods are working, and returning expected output. The scenario tests, on the other hand, does not necessarily test wether or not a service level method returns the correct output - it just checks that a correct input returns any output and no error, while invalid input returns no output and an error. The service level tests should not worry about correct output, since the scenario tests should check this, and we do not want to write all tests twice.

User Types

A system user should have a type, which specifies wether the user is just a normal user, or some kind of admin user. A normal user is just someone who has signed up for the service through the signup form, and is able to browse and rent movies. The admin users does not have the ability to rent movies, but are controlling the content in the system. We chose this way to have the same method calls for normal users and admin users, keeping a clean and consistent API.

There are two kind of admin users: content providers and system admins. The content providers is those who own and manage movies. This could be movie studios, e.g. Universal, or independent movie makers. When logged in as a content provider, you are allowed access to editing and deleting your own movies, as well as registering and uploading new ones. System admins are users, who can manage all content in the system. They can create content providers, and they can view, edit and delete movies. They are also able to ban users, who in some way misuse the system. They ca not rent movies, and they ca not add new movies, but they are a way for the owners of the system to manage content, without having to modify the database directly.

Movie Editions

Typically, movies exists in more editions than just one. A movie can have a directors cut edition, extra material versions, etc., and they can be in both SD, 720p HD and 1080p full HD. Instead of having each version as a new movie in the database, we wanted to group them as editions of the same movie. This is not only to un-clutter movie listings and search results, but also to make it easier for both users and content providers.

We have a special table in the database for editions, and when the user rents a movie, it is actually an edition they rent. On the user interface side, when a user wants to rent a movie, they should be shown all editions of the movie, and be able to choose which one they want. When a content provider is adding a new movie, they first register the movie in the database, without actually uploading any movie data, and they are then able to add editions to the movie. Many movies will have only one edition, but many never movies should be available in both HD and non-HD, which should be uploaded as two different editions. Each edition will

have one corresponding movie file uploaded, which is the one the user will receive when renting that edition of the movie.

Rentals

When a user rents a movie, they will have the ability to download the movie file for as long as the rental period lasts. We decided to have a standard 7 day download period, and if the user wants the movie after that period, he will have to rent it again. There is no payment in the system, but we would have liked to extent the system to make a content provider able to set a price for renting a movie, and possibly also for buying the movie. Other features that could be implemented is discounts on multiple purchases and discount periods.

Having a end date for a rental and allowing content providers to customize prices on movies was some of the ideas suggested by the SMU team, that made it into our feature considerations.

5.2.2 Interface

Our public API should cover 4 different topics: user management (login, logout, sign up), content browsing (get movies, search, get movies in genre, etc.), rental management (rent movie, download movie, view rentals), and content management (register movie, upload edition, edit movie information, delete movie). That's why we have 4 interfaces: IUserManagement, IContentBrowsing, IRentalManagement, and IContentManagement. These four interfaces contains service methods for everything you need.

When designing the API, we had the goals that we wanted it to be simple, easy to figure out, easy to use, small and precise. That's why we decided to have a unified style for all methods: a boolean return value, which is false if an error occurred, a string token as the first parameter, and any return objects as either out or ref parameters. This gives us a unified API, where you don't need to learn how every method works, but are able to use it without much friction.

Testing

Other than scenario tests, we have a lot of service level tests. Since the service interface is only a thin wrapper around the actual logic classes, and almost only validates the input, these tests focuses a lot on trying to call the service methods with invalid input. For each method in the service interface, there is one tests that checks that a valid input produces an output and no error. It doesn't necessarily check wether or not the result is correct - that is up to the scenario tests, since the returned value is just the result of a logic-level method call

SMU involvement

When collaborating with the SMU team, they pushed to get a interface quick, so the first interface wasn't as well designed as we wanted it to be. It wasn't consistent and it was difficult to extent. An example of this was when we wanted to change the way genres worked: there was no way to do this without breaking the interface. That's why we chose to make the new interface, which improved the old one in every way.

We did exchange ideas with the SMU team about what features should be in the interface, which has been discussed earlier, and some of these has made it even to the new API.

5.2.3 Error handling

We started by throwing exceptions through the service if an error occurred. Down the line we realized that it probably was not a good idea to do it like that. Windows Communication Foundation (WCF) only throws FaultException - even if we throw another exception. The only setting we could tweak, was whether or

not the original exception was included in the FaultException, as an inner exception. This obviously is not easy for the client applications to handle, as they would have to catch FaultExceptions and validate which exception it really was, by looking at the inner exception.

The reason is that it is best practice to inform client application of errors in another way. At this time of the project when we realized that, our system was already built on exceptions. With other, more important tasks to do, we only had time for a simple change.

When we re-designed our service interface (see 5.2.2), we used boolean return values for most of the service methods. These would return true for success, and false if any errors occurred (like invalid input). If we were to return anything else, we would use `out` or `ref` parameters.

An even better solution, which we would have implemented if we had the time or had thought about from the start, would be to use enum return types, together with the `out` or `ref` parameters for returning data. The problem with only true/false, is that the client application has no way of knowing what went wrong - just that something went wrong. By creating enums we could pass more information about what went wrong to the client application.

For the SignUp service method, the enum could contain the following values:

Success Signup successful.

UsernameInUse A user with the specified username already exists.

InvalidEmail Email specified is invalid.

InvalidPassword The password specified is invalid (not long enough or empty).

Error An unknown error occurred.

This would provide a lot more information for the client applications, and it would give the client applications to recover and try again.

5.3 Client

This section covers the design decisions we made regarding the client implementation and Graphical User Interface (GUI) design.

5.3.1 Analysis

We looked into what architectures were commonly used and suggested when developing a client with a `GUI` front-end and a `WCF`-service backend. We have seen (and used) the `Model-View-Control` (`MVC`[1]) before this project, and we felt it fit nicely with having a model (the service), a view (`GUI`) and a controller to make it all work.
While looking further into `MVC` and how we could apply to our client, we found the `Model-View-Viewmodel` (MVVM[2]) architecture pattern. The `MVVM` pattern is based largely on `MVC`, but is targeted at modern `UI` development platforms (such as HTML5, Windows Presentation Foundation and Silverlight).

Because we decided to use Windows Presentation Foundation (`WPF`)[3], we decided that trying out the `MVVM`-pattern was a good idea. `MVVM` offers a complete seperation of the model (in our case the `WCF` service) and `GUI`. The viewmodels serve as translators (and sometimes logic functionality, depending on implementation).

[1] Wikipedia description of Model-View-Control[9]
[2] References on Wikipedia [10] and MSDN [11]
[3] Decision described in section 6.2 on page 26.

5.3.2 Our version of the MVVM Architecture

Having chosen the MVVM architecture, we decided to implement[4] our own version of it (instead of using frameworks set up to use MVVM). It gave us more control over what we wanted to do with the architecture, as well as letting us make any modifications we want to.

The biggest change we have made to the usual architecture[5] is our interpretation of models. Instead of having models be the actual database, we use it as a communicator with the service. In the MSDN blog post on the MVVM architecture [11] and how it can be used with WPF and a WCF service, the model is described as being the actual service.

In our approach, we design the models as being seperate classes with an interface that the view models can use. This completely seperates the viewmodels from the service calls (meaning the model could actually be anything, as long as it implements the same interface). We will still use classes and objects from the service reference in the view models, but to change the data/service the model accesses, one could simply use the model to translate the types into the types from the service reference. This allows a modular approach to the system.

Because we have a modular approach to the model-viewmodel relationship, we feel the view-viewmodel relationship should be modular as well. Because of this, the implementation of the view models could change vastly without having any effect on the views (except if it changes interface).

5.3.3 Graphical User Interface

Designing the GUI we had two different approachs. One where we opened a new window each time the user would access a new functionality, and one where we had one window in which all the functionallities would be shown. The design team tried out both options and found that having multiple windows to show the functionallities in was to clumsy and would disturb the users more than help them. We therefore went with the one window solution, since it felt naturally and we could represent new functionallities without disturbing or confusing the user. With the one window design we had each functionallity page consist of three areas: In the top we had shared buttons for the user type, in the middel we represented the page content associated with the functionallity and in the bottom we had the buttons associated with the page content. See figure 5.3 on page 22 for graphical representation.

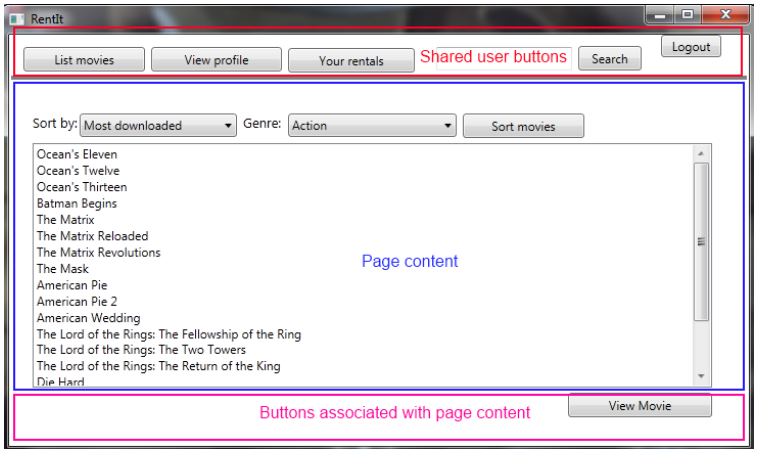

Figure 5.3: Graphical representation of the overall GUI design

[4] How we implement this is described in section 6.2.1 on page 26.
[5] `View(GUI)-Model(database/datafiles)-ViewModel(translator)`.

Usability

When we began designing our GUI, we felt that it was important it was user friendly. Therefore, we choose to make usability tests (see 8.1.3), since they always will grant some degree of usability if performed correctly. In total we conducted two usability tests.
From the first of the tests, the feedback told us that we didn't have enough user confirmation in the GUI. We discussed this in the design team and came up with a solution, which added dialog and confirmation boxses to a lot of our buttons which contained a save function (see figure 5.4on page 23).

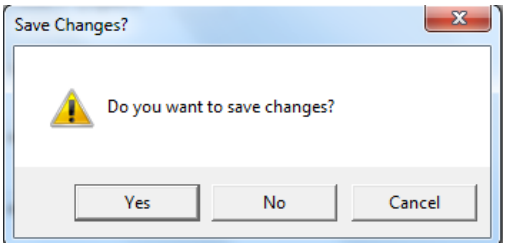

Figure 5.4: Confirmation box

In the second usability test, we went from testing on a papermockup of our client, to testing on our client prototype, which incorporated the design changes from the first test. In this test, we received no feedback concerning the lack of conformation in the client, which meant that we had sovled the problem from test one. We did however get feedback on the navigation of the client. Our test users found it hard to navigate to the correct pages during tests. They said that either they had to navigate through too many pages to get to the correct one or the buttons were named ambiguously. For example, one of the users thought that the view movie button would play the selected movie when clicked.

Unfortunately, the second usability test was carried out too late in the process, which meant that we didn't have time to incorporate changes to the client. If we had had time, we would have done the following: We would have revised the naming of our buttons, such that there would be no confusion with regards to their functionality. In addition, we would makes changes to our "menu bar" at the top of the client, such that the user would allways have more pages to navigate to. This would make unnecessary navigation through other pages redundant.

6 Implementation

This chapter describes the implementation of the service and the client.

6.1 Service

6.1.1 Architecture implementation

We have two parts of our service implementation: the service layer and the logic layer. The service layer is our public API, while the logic layer is our private, inner mechanics. Their purpose is clearly defined in their design: the service layer is completely build around the API we have designed, while the logic layer is designed to be compatible with Entity Framework.

Our API is divided into four main categories: user management, content browsing, rental management and content management. For each of these, we have a class in the service, which implements all public methods to do what we want the user to be able to. The classes are `UserManagement`, `ContentBrowsing`, `RentalManagement` and `ContentManagement`.

The logic layer is not based around the interface, but based on being compatible with Entity Framework. The core classes in this layer is our entities. They each corresponds to a table in our database, and is mapped to these, so that instances of these classes can represent data in our database. Along with being used as database structures, they also contains the logic associated with each entity. These are defined as either class- or instance methods. These methods implements both the CRUD functionality we want per entity, but also general functionality that relates to it, without actually utilizing a specific instance. An example of this is the search method on the `Movie` class, which is a static method, but returns a collection of movies, that matched the search criteria.

Each entity have an All property defined. This is a way we use to abstract away from Entity Framework. Instead of manually getting a context to get an enumerator of all instances in the database, we just use this property. It's both easier to use, and it's more secure, since it adds abstraction.

To communicate with the database, we have the `RentItContext` object, which is an Entity Framework object context. This class has a set of all out entities, which is mapped to the tables in the database, so that it will pull the data from there, when we're iterating through. This class is the way we handle reads and writes to the database. The intended way to use it is to create a new context each time you want to access the database in some way. Unfortunately, this conflicted with the way we had chosen to do POCO classes, because the same object received from two different contexts weren't compatible. This caused a lot of trouble, so we ended up with instead having a shared singleton instance, which is lazily initialized first time it's accessed, and will be preserved for the next call you want to make through it. If you want to reload it, we made a method for disposing the context, and a new one would be initialized next time the context was accessed. This solved a lot of the problems we had, because the same context was now used all the time. The only problem was that a multithreaded service would still access the same context, and multiple requests being processed at the same time would cause trouble. That's why we store the context in a thread local storage. This makes sure that a context will only exist in a single thread, and if another thread tries to access it, a new one will be initialized to run in parallel.

Issues, workarounds and fixes

One of the issues we kept working on throughout the project, was how to use Entity Framework properly. Initially we created a new context every time we needed data from the database. The problem with this approach, is that if we pass entity instances around (internal in the service), we cannot use an instance from one context in another. This meant that we had to find the object again in the new context, before we could use it.

We later found out that the best practice for ASP.NET websites was to use a single context for the entire request, and dispose it after the response was sent to the client. Since our service behaves the same way as a website, we decided to implement this approach. This proved to be a lot better, as we no longer had to create a new context all the time, and we did not have to find the same object in a new context all the time. The only time we had to "synchronize" an object with the context, was with the objects we received as parameters in the service layer. We did this synchronization in the service layer, which meant that we throughout the logic layer could trust all objects. This also meant that it would make sense to change our logic layer (which was primarily static methods in the entity classes) to instance methods, which provided a much nicer, more consistent and cleaner design throughout the logic layer.

Another problem with Entity Framework was whether or not to use lazy loading. With lazy loading all one-to-many and many-to-many relations was only loaded when it was requested. To do this, it created a proxy for the entity class behind the scenes. The problem with this was that these objects were passed to the client, where they failed due to no connection to the database through Entity Framework.

We fixed this by disabling lazy loading and proxy creation. But disabling lazy loading does not enable eager loading. There is not such thing as an eager loading setting to enable, which meant that all of a sudden our one-to-many and many-to-many relations were not loaded at all. We had to manually load them through calling Include() method in our LINQ-to-Entities queries. This was not very nice, and it was very easy to forget this, which lead to problems very hard to debug.

We solved this by creating a static property on each entity class, called All. This called all necessary Include() to get it to properly get all information for that specific entity. By using this as a base for our LINQ-to-Entities queries (by using `Movie.All` instead of `DbContext.Movies` as base) we were ensured to always have all relations available. This also means that we always load all data, even when we do not need it. But we think this is an acceptable sacrifice, as we did get code that was a lot nicer and cleaner, and less error-prone (as we would not forget to Include() what we needed).

Another difficulty we experienced, was with WCF. We used `MessageContract` (instead of `DataContract`) for our `RemoteFileStream`, as we gained more control over the SOAP message that way, and because we could not get it to function properly with `DataContract`. But we discovered that when using a class with `MessageContract` attribute as a parameter, no other types of parameters could be used. That meant that we had to move token and `Edition` (to identify which edition to download, or to pass name and movie id for uploading) to `RemoteFileStream`. Another quirk was that when using a class with `MessageContract` attribute as a parameter, we either had to return nothing or return something of the same type as the input parameter. This was also true for the other way around (using a class with `MessageContract` attribute as return type).

Because of that we changed `RentalManagement.DownloadFile` to take a `RemoteFileStream` as parameter (to identity the user and the movie edition to download) and return a `RemoteFileStream` with the stream. We also changed `ContentManagement.UploadEdition` to take a `RemoteFileStream` as parameter and changed the return type to `void`. This meant a slightly inconsistent service interface, but the service interface was to begin with quite consistent, and we agreed that these small inconsistencies were acceptable, to get it working.

6.2 Client

This section describes the implementation of our client, the issues/bugs in the code, the fixes and workarounds we used to circumvent those issues and how we handled errors.

6.2.1 Architecture implementation

Our implementation of the client is seperated into two parts. The implementation of the `MVVM` architectural pattern and the implementation of the `WPF` framework. The full overview of the different

Model-View-ViewModel

Our intention is for the client to follow a simple dependency flow such as described by figure 6.1 on page 26. The `View` (`GUI`) should only request information from the `ViewModel` and should never be called by anything but classes from the GUI namespace. Similarly the `ViewModel` should only be called by their respective `View` classes[1].

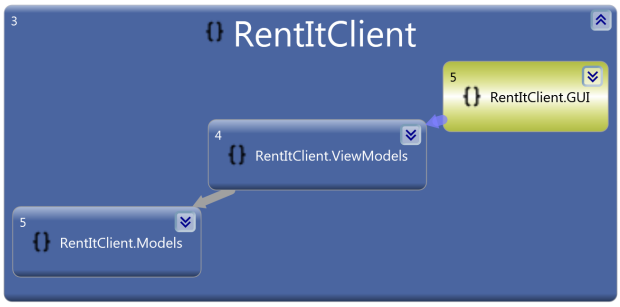

Figure 6.1: Namespace dependency in the client

At the "bottom" level of our architecture, the `Model` classes should only be called by eachother and the `Viewmodels` (although no `Viewmodel` should ever call the `ServiceClient` class). The `Model` should never make any calls to the `ViewModels` and `Views`. The actual data source that the `Model` classes use should not be of consequence to the `ViewModel` or the `GUI`.

Static classes The reason we use static classes and methods for the `ViewModel` and `Model` is due to the actual application being a singleton in itself, but the way we use `WPF`[2] means that we do not have access to static objects in the `GUI`.

Upload/download Our upload and download implementation is heavily inspired by "Streaming upload/-download files over HTTP" on [12].

Windows Presentation Foundation

Our implementation of the `View` is done in `WPF`. It involves a single `MainWindow` which is essentially a `WCF NavigationWindow`. It opens up a `LoginPage` inside the window on startup. Whenever we change to a new page (for example on when a user completes login), we use the `NavigationService` of the window to open a new page.
The way we navigate using the navigation service means we have to create new `Page` objects every time we

[1] This is described in more detail in D.1.1 on page 51.
[2] See section 6.2.1 for our implementation of the `GUI` using `WPF`.

want to change to a new page. Because of this, we placed all of our "page initialisation" in the constructors of our pages. The full overview of the dependencies of the `View` part of our client can be on D.3 found in section D.1.1 on page 51.

Issues, workarounds and fixes

Bugs We have a number bugs in our client - some of them are related to issues from communicating with the service.

- View Profile does not display all fields.
- Upload and download of movies rarely succeeds due to stability issues.
- Some `null` references (output from the service) are not caught at the Model level, and because of this, the client can crash when it is supposed to show a different error message.
- Client does a forced shutdown if any method with a token returns false. If this is due to some kind of authentication issue, we could retry login instead of just closing right away.
- User is not logged out when he closes the client with clicking log out.
- Sorting by newest rarely works.
- Sometimes editions do not display (possibly a bug on the service end).
- Edit movie does not work.

Workarounds We also have a few odd ways of dealing with some issues we have run in to.

- We use code from [13] to make sure that messageboxes use the right language for the buttons (for example "Yes" should remain "Yes" even on a danish system).
- `GenreChecked` and `GenreCheckedList` are a workaround to avoid having to design our completely own XAML element for the `CPRegisterMoviePage` and `CPEditMoviePage`.

6.2.2 Error handling

We handle errors in two different ways in the client.

Bad input to server If the server returns false (meaning bad input), in all cases, except at login, we tell the user of the client that an authentication error has occured. We do this because a lot of the method calls to the service contain nothing but the token as input, but they all return the same error (false in the return of the method).

Communication/faultexceptions If any of our method calls result in an exception, we tell the user "An error has occured..." and that we will shut down the client. Because the service should only throw exceptions on "bad input", any time where the service throws an exception, we assume that something went wrong and we cannot recover from it.

Improvements If an implementation of the fault enums[3] was done on the service, it would make it a lot easier to show a better error description to users and give the developer(s) of the client more options in terms of "error recovery".

[3] Described in section 5.2.3 on page 20.

7 Manual

7.1 Client

7.1.1 Navigating the client

The following section will be a short manual on how to use the client when trying to perform two standard tasks: to rent a movie from the service, and to upload a moive to the service.

Renting a movie

The first thing you see when you start the client is a login page that requires your username and password, if your not allready a user you can press the signup button which will navigate you to registration page where you can signup for the service, when this is done you will be returned to the login page. If you login as a user a new page will open see figure **??** on page 28

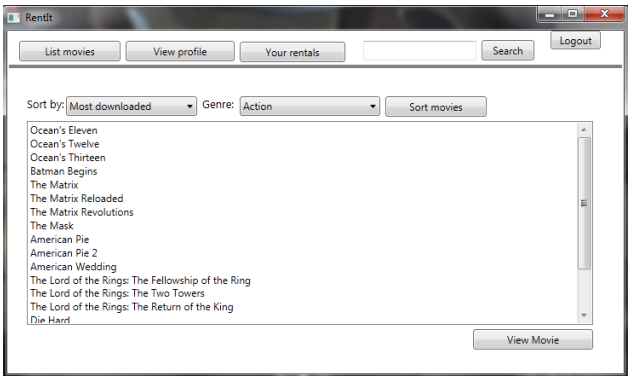

Figure 7.1: List of movies on service

and a diffrent if you are an content provider but more on that later.
When logged in as a user you want to rent a movie lets say "Batman begins". You now have two options either to sort your movie list after action genre or you can search after it by typing in the name in the search field and hitting the search button.
If you use the search option you will then be navigated to a new page see where the result of the search will be shown like in figure **??** you can then choose "Batman begins" and press the view movie button which in turn then will open this page, see figure **??** on page 29

where you will be able to see information about the movie, and also be able to choose which edition to rent. You then choose a edition you want to rent, in this case we will choose the HD 1080p edition of "Batman begins" we then select it in the list and click select edition, this will promt us to a page similar to the view movie page, the only exception being that we can now press "Rent edition" which will then add the movie to your account for seven days and then navigate to the download page where you can press the "Download movie". Here you wil be able to download the movie and choose a filepath to save it in. See figure 7.3 on page 29

When done you can then press the logout button to logout and return to the login page or you can press the close button (X) which will then log you out and close down the application completly.

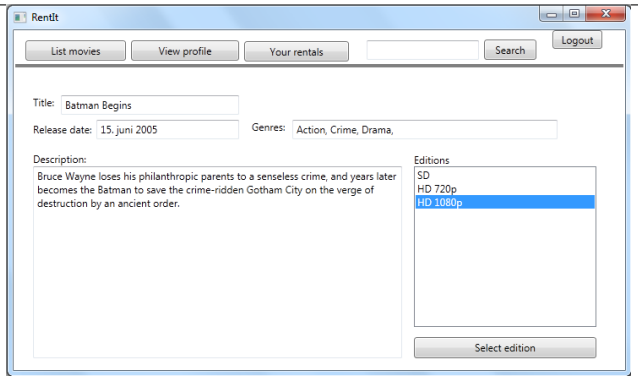

Figure 7.2: View movie

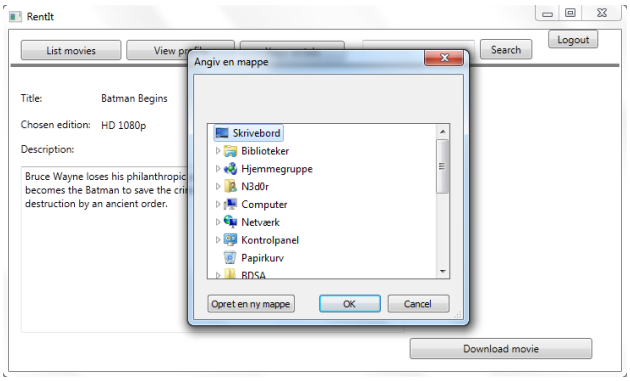

Figure 7.3: Download movie

Uploading a movie

In the case where you login as a contentprovider, you get a startscreen where you can see a list of your uploaded movies, see figure **??** on page 29

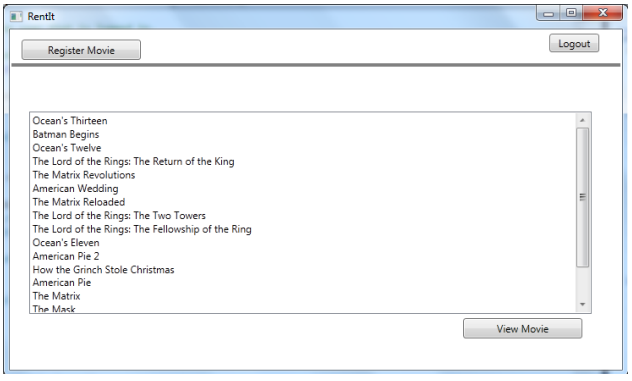

Figure 7.4: List of uploaded movies

you also have the ability to register new movies. To register a new movie you press the ” Register movie” button this will then open this image, see figure 7.5 on page 30

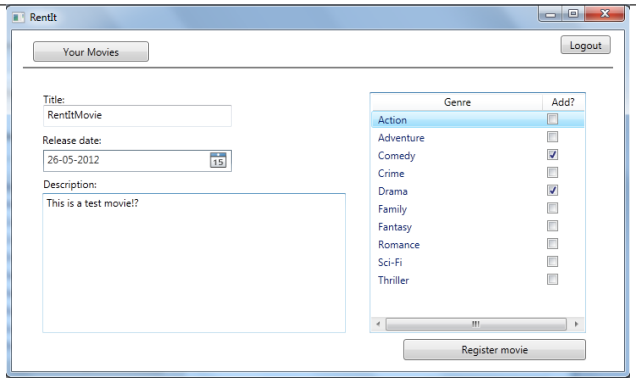

Figure 7.5: Register movie

here you can give the movie a title, release date, genres and a short description. You can then register the information by pressing "Register movie", when doing so a dialogbox will apear, see figure 7.6 on page 30

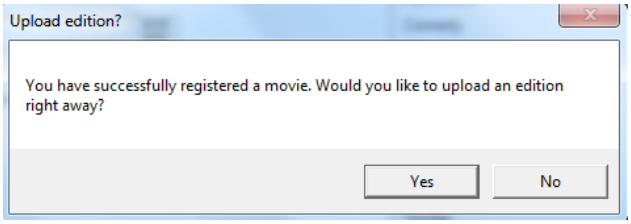

Figure 7.6: Upload edition popup box

prompting you if you want to upload editions right away aswell.
If you press yes, the window will then navigate to this page, see figure 7.7 on page 30

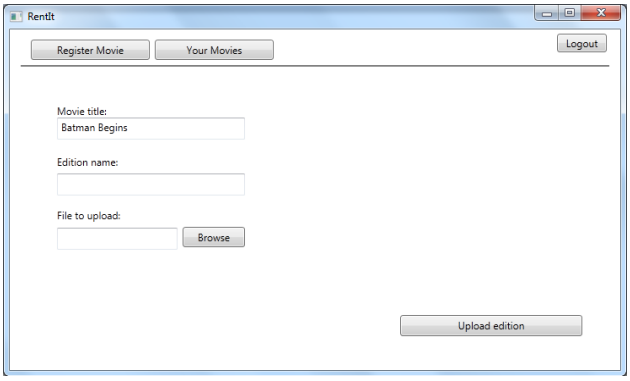

Figure 7.7: Upload edition

which allows you to upload an edition of the movie with a name and a file, when you press the "Upload edition" button, the edition will be uploaded to the service and you will be returned to your start screen.

7.2 Service

7.2.1 Using the API

The following section is a short manual on how to use the service API in custom client applications. It will follow the same two tasks as those in the client manual, only focusing on the service calls.

All methods in the API have the same format. The return type is always a boolean, which is true if the input was valid, but false if the input was invalid. All objects to be sent to the client is sent through either an out or ref parameter.

All methods are places in 4 different classes: `ContentBrowsing`, `UserManagement`, `RentalManagement`, and `ContentManagement`.

7.2.2 Renting a movie

The first step to using the service is always either log in or sign up for a new user. The `Login` method in `UserManagement` takes a username and password, and sends back the logged in user object. The `SignUp` method takes a referenced `User` object, where at least username, password and email is filled out. The returned user object for both methods will contain a token property, which will be used in all subsequent method calls.

After login, there is several ways to get movies, all using the `ContentBrowsing` class. The `GetMovies` method can get all movies, newest movies, and most downloaded movies in either all or a specified genre. The genres can be retrieved with the `GetGenres` method. Alternatively, you can search for movie titles with the `Search` method.

To rent a movie, the `RentMovie` method in the `RentalManagement` class is used. This takes a movie edition. The list of editions for a movie can be retrieved with the `GetMovieInformation` method in the `ContentBrowsing` class. When a movie edition has been rented, it can be downloaded with the `DownloadFile` method.

7.2.3 Uploading a movie

When you log in as a content provider, you get the ability to upload, edit and remove movies through the `ContentManagement` class. The `RegisterMovie` method registers a new movie in the system. It takes a referenced movie instance, which has to at least have a title. A release date can also be set, and if it's a day in the future, the movie won't be visible in the system before then. A movie can afterwards be edited and removed using the `EditMovie` and `DeleteMovie` method.

After a movie has been registered, versions of it can be uploaded with the `UploadEdition`, and afterwards be removed with the `DeleteEdition` method.

8 Testing

In this chapter we discuss our testing strategy, our results using said strategy and what we feel we could have done to improve our testing overall.

8.1 Strategy

Our testing strategy consists of different kinds of tests and a tight integration with how user stories work in the Scrum development strategy we are using. We have three different kinds of software tests: Scenario tests, Service tests and Graphical User Interface (GUI) tests. We also have usability tests to ensure we have decent interface in terms of usability.

Almost all functionality in our RentIt system is implemented by user stories. A user story is not accepted until tests of the functionality pass and the tests have been reviewed by the team member responsible for QA. The team member responsible for QA is Jakob Melnyk. If he is the one who created the test, a different teammember takes care of the QA for that test.

In our opinion, this ensures an acceptable level of peer review during development of the system. Between the feature freeze and code freeze dates, everyone reviews both the code and the tests, so that we get a more thorough peer review.

8.1.1 Code test types

We have different testing categories for testing different levels of the code of the system. The different levels we have defined are scenario, service and end-user levels. This section describes the kind of tests done at the different levels[1].

Scenario-level Mainly tests designed to cover a specific user story.

Service-level Mainly consist of testing the connection to the service and the different service contracts.

End-User-level Test functionality of the GUI for the end-user.

Scenario-level tests

We have used scenario tests to test features in our project. Features such as editing a user profile or renting a movie are tested on the classes containing the logic for editing the database and filtering information. This is done to seperate the logic from the Service class itself, such that the service can be exchanged without affecting the logic much (if at all). This also enables us to test the logic seperately from the service itself.

Most of our automated tests are scenario tests[2]. We have chosen to put our focus on scenario tests, because scenarios are integral to the way, we have developed our service. Most features are implemented by making one or two test methods to cover the necessary implementation of the feature [3]. Our scenario test list (with results) can be found in the appendix on page 55.

[1] We originally had method/unit testing as well, but we decided the way we have structured our system does not make it easy to do standard unit testing.

[2] Our automated tests number 66 scenario tests, 31 service tests and 13 GUI tests.

[3] Design and architecture is more thoroughly covered in chapter 5 Design.

Service-level tests

Our service levels tests do not cover as much of the API as our scenario tests and are not as thorough in testing the functionality. They are mostly intended to test for connection problems, bindings issues, data contracts and error handling. Our service test list (with results) can be found in the appendix on page 57.

Graphical user interface tests

Testing that the service works is not the only important thing to do. It is also important to make sure that the graphical user interface works, because otherwise the user will not be able to use the service for anything. We split the testing of the graphical user interface into two parts: Automatic tests and manual tests. They are described below.

Automatic GUI tests The automatic GUI tests have been made through the use of the Coded UI Tests in Visual Studio 2010. When one creates such a test, one can record all the actions that are taken using a program and save these. When a test has been recorded, it can be run any amount of times and it will take the same actions every time. Using this we made automatic tests for all the basic features, such as signing in, logging in, searching for movies, etc. They are listed in the appendix under E.1.3 on page 58.

Manual GUI test While we could automate some tests, there were others that were not worth automating. Automating things like upload/download would be too hard, as file directories change from computer to computer. This meant that we had to test certain functionalities manually, simply by opening the client and going through the necessary steps. They are listed in the appendix under E.2 on page 60.

8.1.2 Regression tests

Every time we run in to an error or a bug in the system, our strategy is to create a test that covers the scenario the error presented itself in. The test is intended to fail the first time it is run (to verify the bug exists), then when the issue has been fixed, the test passes. These tests are only meant to cover the specific bug they were designed for, but because they are already designed (to check if the bug has been fixed), they are kept as regression tests to ensure the bug or error is not reintroduced later. All of our automated tests can serve as regression tests, as they do the same thing - ensure that functionality is not broken by later changes.

8.1.3 Usability tests

When designing a user interface you have to take into account that not all users is equally proficient in navigating IT systems, therefore we have to design a interface which is easy to use. To accomplish this we did a couple of usability tests. Usability tests, is a testing technique which focuses on the usability of a user interface, this is measured in non-functional requirements. For usability testing you need a mockup to test against, you then make a list of usability goals[4] if these goals is fulfilled then you have the user interface that you wanted. For the usability test itself you make a list of scenarios that the your user shall go through [5], while performing the scenarios, the user is asked to think aloud, such that the overseer of the test can take notes on how to improve the system.

The way we went about doing our usability tests, was to first set down as a team and create some paper mockups, which we found user friendly and had high ease-of-use. We then made some usability goals which if fulfilled, would ensure us that our interfaced was indeed user friendly and had a high ease-of-use. With these we made our first usability test on the paper mockups, where we had two test users go through our usability scenarios, when they where done we then assessed how they compared to the usability goals.

[4]See E.3.1 for our usability goals
[5]See E.3.1 for our usability scenarios

For the second usability test we created a digital version of our paper mockups, but this time we added some dialog and confirmationboxses to ensure that the user didn't feel that their changes would go unsaved. Besides that we change abit of the design but without deviating to much from the paper mockups. When then conducted the test the same way as we did with the first, but this time on the digital version.

8.1.4 Code coverage

It is rarely an effective strategy to cover every combination of paths through the code, because it is very time-consuming (both in creating tests and actually running those tests). Instead it is important to test enough to cover a lot of code and functionaliy without crossing the line of where it starts being redundant to do so[1]. With that in mind, we have decided to use a twofold approach to our tests. We write tests to cover different inputs and scenarios for our features, then assess our code coverage. If the coverage has not reached our goal, we think up new tests to increase the coverage procentages.

Our requirements for code coverage are as follows:

- Minimum overall coverage is 50% of service.
- Goal is to have 80% overall coverage of service.
- Workflows: All use cases must be covered.

Minimum overall While a 50% code coverage may seem like a low number, we have it at this level because we do not plan on writing scenario tests for the the `Services` namespace. This means a lot of the precondition checking will not be covered, and so a lot of statements will not be covered. In addition the 50% requirement is only an absolute minimum so we aim to do better than 50%.

Overall goal The overall coverage perecentage goal for the service is a lot higher than the minimum coverage. This is because, in an optimal scenario, we will test the `Services` namespace more thoroughly and cover more a lot of statemets in it. While a true optimal coverage percentage would be 100%, this is rarely feasible (as mentioned in 8.1.4 in a real-life environment, as it is often costly to do so. As our software is not "mission critical"[6] (especially not since we have yet to implement any payment options).

Workflows In addtion to covering functionality on the service, we also need to cover "workflows" in the client. The easiest way for us to check if we have fulfilled the requirements of the project is to simply do a workflow that incorporates the use cases described in 3.4 on page 8. Our list in the appendix on page 60 describes which use cases are covered by which manual test.

8.1.5 Configurations

We use a different service and database for running our tests. The aim is to have a working version of the service on both the production/release address as well as the test address. Having seperate databases lets us reset data on the test database without having to fiddle with data on the production service.
The bad thing about doing this is that we have to make sure that the two versions are kept up to date. Else we cannot be certain about the quality of the production service from looking at the tests.

8.2 Test results

We do not have 100% test pass in our automatic tests. This is disappointing, but due to delays our system was not anywhere near being finished before very close to the deadline, so we ended up not being able to fix all of the bugs (both in code and tests). We have 3 scenario tests that fail and 16 service tests that fail.

[6] In this context, mission critical means any software that can cost lifes or a lot of money in case of a failure.

We have 6 automated GUI tests that fail and 3 manual GUI tests that fail. In total, 28 tests fail out of 111 total tests.

8.2.1 Code coverage

- Service[7] — 58%
  - RentItService.Services — 0%
  - RentItService.Exceptions — 17%
  - RentItService.Library — 61%
  - RentItService.Entities — 75%
  - RentItService — 100%
  - RentItService.Mapping— 100%

All in all we feel our values are quite acceptable.A coverage of 0% in the `Services` namespace was to be expected, as we did not design scenario tests for that namespace. We could have done more to increase code coverage, but because we also do service-level tests, we felt we should put our focus elsewhere.
All the exceptions inherit from the base `Exception` class and as such have constructors we do not use. We could artificially pump up the coverage of the `Exceptions` namespace by making tests for it, but we felt it unnecessary. The library has a low 61% because we do not test upload and/or download in the scenario tests. Because we have some bugs in both editions and rental, we have low coverage in some of the classes in `Entities`.

Our goal was a 80% overall coverage of the service, which we are a bit ways away from achieving, but the numbers could be reached through testing of the service code and fixing the problems in Edition and Rental.

8.2.2 Results of GUI workflows

Some of our automatic tests for the GUI workflows fail because of weird bug outs in the framework we use. The ones that fail (and the non automated GUI tests we have) have been done manually. During testing we found that some of our use cases (and thus requirements, see 3.4 on page 8) are not fulfilled as some of our tests fail.
The requirements we do not cover with successful tests of the use cases are: "User - Download media", "Content Provider - Upload Media" & "Content Provider - Edit Media".

8.3 Reflection on test strategy

We are pretty happy with the way our strategy turned out. It forces us to develop our service in modules and forces us to make our client able to perform the use cases from our requirements. Because we designed tests quite early (tests were made for each user story every time it was finished), we were forced to change them when we made major changes to the service logic during refactoring. This was both good and bad, as we could not use our code coverage tools for a while.

8.3.1 Improving our testing strategy

In order to cover more code with tests, we could have used an automated unit test generation framework such as Pex[18]. To really use frameworks such as Pex to their full potential, we would need to design our system in such a way that the logic of the service is not only seperated from the classes representing the endpoints (like we have done), but also seperated the database communication logic from the service logic. We could also "artificially" create more hand-written tests to cover the cases described in 8.2.1.

[7] Full coverage table can be found in section E.4 on page 64.

We should probably have defined some clearly critical sections for testing (such as service logic enabling the core requirements). This would have made us focus on making specific features work first before designing more elaborate ones.

9 Conclusion and reflection

In this chapter we summarize the different faults in our project (both code and in our collaboration), what we would have done differently and how we feel about the overall result of the project.

9.1 Collaboration

Looking back on the project as a whole, we made several mistakes internally in the group when it come to collaboration.

Distribution of workload In the early part of the project we were not able to properly distribute the workload, so we ended having two people working on quite a lot more than the rest of the group. Because we did not deal with this issue early on, it did not change until quite late in the project.
The three that had not been as big a part of the project had to be brought into the loop concerning the plans that the two ambitious people had for the project. Due to this we had to spend a lot of time getting everyone up to date and then distributing the workload for the last couple of weeks. Even with a much improved distribution of workload, we still did not have anywhere near an optimal distribution.
As the two ambitious group members were the people with the "know-how" in the plans for the project, a lot of the main work defaulted back to the and the rest of the group was given tasks that mostly featured testing.

Began working on the client too late Because of the issues with our distribution of the workload, we felt we had enough to do just in the service itself. This meant we hardly gave the client a thought until after we finished working with the SMU group. This meant we discovered a lot of issues with the bindings, interface etc. quite late and so it got a bit stressful in terms of fixing them in time for the code freeze.

Client/service team seperation After having finished working with SMU we decided to "split" the team into two groups: one working on the client and one working on the service. Two people were assigned to the service and the rest was assigned to the client. This split proved to be a wrong decision, as three people were too many to have working on the client and two people on the service proved to be too few.
We should have put three people on the service and only two on the client. In addition we could have had great benefit from having team members swap between client and service, as it would have given us more insight as a team.

9.1.1 SMU collaboration

As we point out in our Collaboration chapter, we made several mistakes in our communication with the SMU group during development.

Our biggest mistake The biggest mistake we made in our communication with SMU was when we thought everything was going well. Throughout the early stages of the co-development with SMU we got a lot of questions and error reports. Suddenly we stopped getting error reports and thus we thought everything was fine.
It was not. Instead they had been trying to fix the problems with keeping in contact with us, so we had no idea of the errors in our service. When we finally received a big bug report from them, we did not have a lot of time to fix the issue before they needed our service work for their hand-in.

Bad communication As a general mistake we made in our collaboration with SMU was our communication with them. We should have been more clear and verbose in our communication in order to make sure we were not being misunderstood by the SMU. Because we did not do this, we ended spending a lot of time clarifying issues days later. One time we ended figuring out that we had misunderstood what they have said a little over a week later.

9.2 Issues and potential fixes in code

As mentioned previously in the report, we have a number of errors/bugs in our client. This is a quick summary of the previously mentioned errors.

Service issues

- Download/upload does not work properly.
- Signup fails if a username is already in use.
- A content provider can currently only see his own movies.
- Optional parameters on `ContentBrowsing.GetMovie`() does not work.
- Enum types should be used for error returns on method calls to provide a detailed error description.

Client issues

- View Profile page does not display all the fields about a user.
- Upload and download of movies does not work.
- Some null references from the service are not caught properly and thus we do not show the correct error message on all crashes.
- Client does a forced shutdown if an input token is incorrect - should consider doing a "retry login" feature.
- User is not logged out when closing the service.
- Sorting by release date does not work.
- Edit movie does not work.

9.2.1 Issues we would prioritise

If we were to develop more on this project, we would prioritise making sure upload and download work so that our client and service actually lives up to the core requirements. Any further development time would be spent on implementing the error enums, so that the interface would enable other developers to have an easier time developing their client.

9.3 Summary

Looking at our project as a whole, we have had a lot of issues both in implementing the requirements and also in terms of successful collaboration as a group and with the SMU team. We had too much focus on making a very good service interface and reliable backbone that we almost forgot our core requirements. Because a digital media rental would not make a lot of sense without download functionality (and upload for the content providers), we feel it is a major issue that we do not meet the upload/download core requirement.

On the bright side we feel that followed our "project requirements" about development strategy, quality assurance and documentation of our design decisions.

9.3.1 Major learning points of this project

The most important point we have learned during this project is that we cannot be sure that everything is just going smoothly when it comes to international and/or intercultural team work. You need to have some sort of channel to keep track of the status of each group. Our weekly updates felt like too few in a project of our size.

Another very important point we have learned is that we really need to make sure that we distribute the workload properly. It is stressful for the the ones taking the big load and the group members that are not "in the loop" do not learn much about the project.

Bibliography

[1] Blog on "The Way of Testivus": http://www.artima.com/weblogs/viewpost.jsp?thread=203994 (Checked on 22nd of April, 2012)

[2] Wikipedia entry on Blockbuster LLC: http://en.wikipedia.org/wiki/Blockbuster_LLC (Checked on 9th of May, 2012)

[3] Blockbuster revenue loss: http://www.thewrap.com/movies/column-post/blockbuster-announces-q2-results-gets-debt-extension-20105 (Checked on 9th of May, 2012)

[4] Wikipedia entry on Netflix: http://en.wikipedia.org/wiki/Netflix (Checked on 9th of May, 2012)

[5] Interview with Gabe Newell (co-founder and managing director of Valve Cooperation): http://www.tcs.cam.ac.uk/story_type/site_trail_story/interview-gabe-newell/ (Checked on 9th of May, 2012)

[6] Roskilde library website: http://www.roskildebib.dk/ (Checked on 9th of May, 2012)

[7] Wikipedia entry on ITunes store: http://en.wikipedia.org/wiki/ITunes_Store (Checked on 9th of May, 2012)

[8] MSDN website: http://msdn.microsoft.com/en-us/

[9] Wikipedia on Model-View-Control: http://en.wikipedia.org/wiki/Model-View-Controller (Checked on 20th of May, 2012)

[10] Wikipedia on Model-View-ViewModel: http://en.wikipedia.org/wiki/Model_View_ViewModel (Checked on 20th of May, 2012)

[11] MSDN entry on WPF and MVVM pattern: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx (Checked on 20th of May, 2012)

[12] Inspiration for upload and download functionality: http://www.codeproject.com/Articles/166763/WCF-Streaming-Upload-Download-Files-Over-HTTP (Checked on 20th of May, 2012)

[13] Box for changed button text in message boxes: http://www.codeproject.com/Articles/18399/Localizing-System-MessageBox (Checked on 20th of May, 2012)

[14] User Interface Design - a Software Engineering Perspective, ISBN-13 978-0-321-18143-5, Paperback - 2004, 1st Edition, Pearson Education (Us)

[15] Using exceptions in WCF: http://www.olegsych.com/2008/07/simplifying-wcf-using-exceptions-as-faults/ (Checked on 20th of May, 2012)

[16] Entity Framework on MSDN: http://msdn.microsoft.com/en-us/data/aa937723 (Checked on 20th of May, 2012)

[17] Windows Communication Foundation http://msdn.microsoft.com/en-us/library/ms735119%28v=vs.90%29.aspx (Checked on 20th of May, 2012)

[18] Pex and Moles: http://research.microsoft.com/en-us/projects/pex/ (Checked on 20th of May, 2012)

Appendices

A Who did what?

This appendix describes who did what parts of the project. It is split into two sections - code and report.

A.1 Code

Who did what in the code part of the project. Includes writing code, tests, database setup and maintenence.

Frederik Lysgaard	
	GUI design
	Usability testing
Jacob Grooss	
	GUI tests
	Scenario tests (a few)
Jakob Melnyk	
	Client(Code-behind GUI)
	Client(Logic and architecture)
	Scenario tests (a few)
Niklas Hansen	
	Database
	Entity Framework
	Service
	Test Platform (Base test classes and data loader)
	Scenario tests
Ulrik Damm	
	Search(Levenshtein)
	Service
	Service tests
	Scenario tests (a few)

A.2 Report

Describes who wrote what in the report.

Frederik Lysgaard	
	Design (Client/GUI)
	Design (Usability)
	Manual (Client)
	Testing (Usability)
Jacob Grooss	
	Collaboration
	Testing (GUI)
	Test Appendix
Jakob Melnyk	
	Preface

	Project overview
	Requirements
	Design (Client/Architecture)
	Implementation(Client)
	Testing (Except GUI and Usability)
	Conclusion
	Report design (Latex preamble, document structure, etc.)
	Written Review
	Use cases
	Test Appendix
	Test running (Pass/fail + code coverage)
Niklas Hansen	
	Design (Database)
	Design (Entity Framework)
	Design (Error handling)
	Implementation (Issues)
	Service
Ulrik Damm	
	Implementation (Search)
	Manual (Service)
	Test Appendix
	Service

B Written Review

B.1 Layout

Even though the first draft is only ten pages, it feels odd not to have a table of contents (ToC). A ToC would have made it easier to navigate the document.
Section/subsection titles were not very distinct from the standard text. Either some form of numbers or letters could be used to show that a new section begins. Italics could also be considered.
The description of the data model and the web-service felt quite clustered and had no real distinction between when one ended and the other began. Again sectioning could alleviate this issue.
Bullet points (or some other "fancy" representation) of the methods in the web-service description would have been good. It was not very clear what was a method and what was not.
The communication section could have been improved by splitting it into subsections. There are three or four subjects discussed in the section and each of them could have had their own subsection.

B.2 Content

The ER-model notation is mentioned (and Søren Lauesens book referred to) in the communication section, but not in the data model section. It probably should be, as the data model section lacked an explanation of how to read the ER-models.
The test report section could have been improved by showing (and explaining) a template, then giving an example of an actual report log entry. Another option could be to just explain the report log snippet you have actually included, instead of just leaving it in there with no explanation.
The text has a lot of poorly argued for decisions. One of particular note is "Revised ER_model. If your smart you will notice that there is no distinction between admins and normal users, but just use your imagination and trust us n this one."
Another example of poor argumentation: "The way we have been testing out system is far from the most optimal way, but it is the only way to do it." A tool such as Pex is mentioned, but it would be nice if there were a more detailed explanation as to why your system is designed in such a way that only manual testing is possible. It seems weird because it is possible to make tests that do exactly what you are doing in your system. They can just do it automatically. Spamming could be avoided by just being sensible about how often/when you test.
Grammar and spelling errors: We assume this is because of no proofreading being done, but this should be a priority come the final hand-in. One example is the label for the revised data model as quoted previously. Just that label has two spelling errors.
The use cases could use some elaboration (in terms of describing each use case), but the illustration works quite well, and you could possibly just explain the more complicated ones (what does it mean to update a movie?).
You have a good problem introduction (besides the spelling and grammar problems).
Description is a bit lacking on the data models, but the way you present them is quite good.

C SMU meeting logs

C.1 Meeting 1

Date: 6th March 2012
Venue: Video conference
Denmark Time: 9:30 – 9:45
Present: Niklas, Frederik, Jacob, Ulrik, Amritpal, Leonard, Satoshi

Topics covered:

- Introduction
- Communication
  - Skype
  - GoogleHangout+
- Sharing of Information
  - Wiki
  - Facebook group
- Next meeting time
  - Thursday, 3pm Denmark Time
- Expectations
  - As good as we can
  - Good communication

Task to be completed before next meeting:

- Read project scope

Immediate Goals

- Formulate requirements

Next meeting
Date: 8th March 2012
Venue: VoIP
Denmark Time: 15:00 – 17:00

Agenda:

- Roles and Responsibilities
- Project requirements
- Confirmation of from of contact

C.2 Meeting 2

Date: 8th March 2012
Venue: VoIP
Denmark Time: 11:00 – 11:45
Present: Niklas, Frederik, Jacob, Ulrik, Jakob, Amritpal, Leonard, Satoshi

Topics covered:

- Requirements gathering
- Use Case development
- Scope of Project
  - Movie: By ITU
  - Music: By SMU

Task to be completed before next meeting:

- Scope of Movie Rental by ITU side
  - Requirements
  - Use Case
- Scope of Movie Rental by ITU side
  - Requirements
  - Use Case

Next meeting
Date: 15th March 2012
Venue: VoIP
Denmark Time: 13:00 - 14:00

Agenda:

- Refining of Requirements and Use Cases
- Web services
- UI design
- Coding

C.3 Meeting 3

Date: 15th March 2012
Venue: VoIP
Denmark Time: 13:00 - 14:00
Present: Niklas, Frederik, Jacob, Ulrik, Jakob, Amritpal, Leonard, Satoshi

Topics covered:

- Sprint 1
  - Started on Tuesday 13 March 2012
  - End 27 March 2012
  - Sprint planning in progress
- Sprint 2
  - Start 28 March 2012
  - End somewhere between 5th-8th March 2012
- Project Specific
  - Database to be used – MS SQL Server
  - ITU will create backend web services and application with basic UI
  - SMU will modify UI and implement some client-side functionalities
  - No shopping cart will be implemented
    - Instead each user will have a profile that has a list of rented items
    - Each user has to create an account and renting of videos will be done in single "transactions" updating user profiles automatically with a list of rented items
    - This will be done client-side by SMU
  - Users: 3 types of users
    - Admin
      - Has admin rights on other 2 users
    - Content Provider
      - Can upload cannot rent
    - Public User
      - Can rent cannot upload
  - Testing
    - Will be done during each sprint together with functionality development
    - SMU will also conduct testing upon receipt of solution from ITU and provide more documentation for report
  - Sharing on facebook functionality – SMU will implement Client Side
  - Facebook login – not a priority
- Project Backlog
  - ITU to provide a project backlog (without ranking) to get both SMU and ITU updated on functionalities to be implemented
- Metric
  - Project Burndown Chart
    - Weekly updates

Task to be completed before next meeting:

- Stories from ITU
- ITU to provide a basic solution with some functionalities and a simple interface for SMU
  - Deployment instructions
  - Acess to database

Next meeting
Date: 22nd March 2012
Venue: VoIP
Denmark Time: 13:00 - 14:00

Agenda:

- Review solutions and stories for remainder of sprint
- De-conflict any issues
- Discuss advanced features
- Review overall schedule

C.4 Meeting 4

Date: 27th March 2012
Venue: VoIP
Denmark Time: 11:15 - 11:20
Present: Niklas, Frederik, Jacob, Ulrik, Jakob, Amritpal, Leonard, Satoshi

Topics covered:

- Sprint 1 Recap
  - Web services are somewhat done
  - SMU team will send list of questions regarding various methods and variables of the web services to ITU for clarification
  - Web service reference address needed to be communicated
- Sprint 2
  - Started
  - Stories of ITU needs to be uploaded to wiki
  - Will end early late next week
- Project Specific
  - Database just re-established on servers
- Project Backlog
  - Needs to be uploaded onto wiki by ITU
- Metric
  - Project Burndown Chart
    - Needs to be uploaded onto wiki by ITU

Task to be completed before next meeting:

- Stories from ITU
- Wed reference address
- ITU to reply to SMU's email

Next meeting
Date: 29th March 2012
Venue: VoIP
Denmark Time: 13:00 - 14:00

Agenda:

- Review solutions and stories for remainder of sprint
- De-conflict any issues
- Review overall schedule

C.5 Meeting 5

Date: 29th March 2012
Venue: VoIP
Denmark Time: 15:45 - 16:45
Present: Niklas, Frederik, Jacob, Ulrik, Jakob, Amritpal, Leonard, Satoshi

Topics covered:

- Web Services
  - Token
    - Checks if these guys is an admin or publisher
  - Register new user
    - Class instances
  - Upload and download of movies
    - Name of file
    - Length of stream
    - 2gb space limit
- Sprint 2
  - Movie ownership
  - Retrieve entire list of movie

Task to be completed before next meeting:

- Web Services API

Next meeting
Date: 5th April 2012
Venue: VoIP
Denmark Time: 13:00 - 14:00

Agenda:

- Review solutions and stories for remainder of sprint
- De-conflict any issues
- Review overall schedule

D System Diagrams

This appendix covers the class diagrams and dependecies for the system.

D.1 Class diagrams

D.1.1 Client

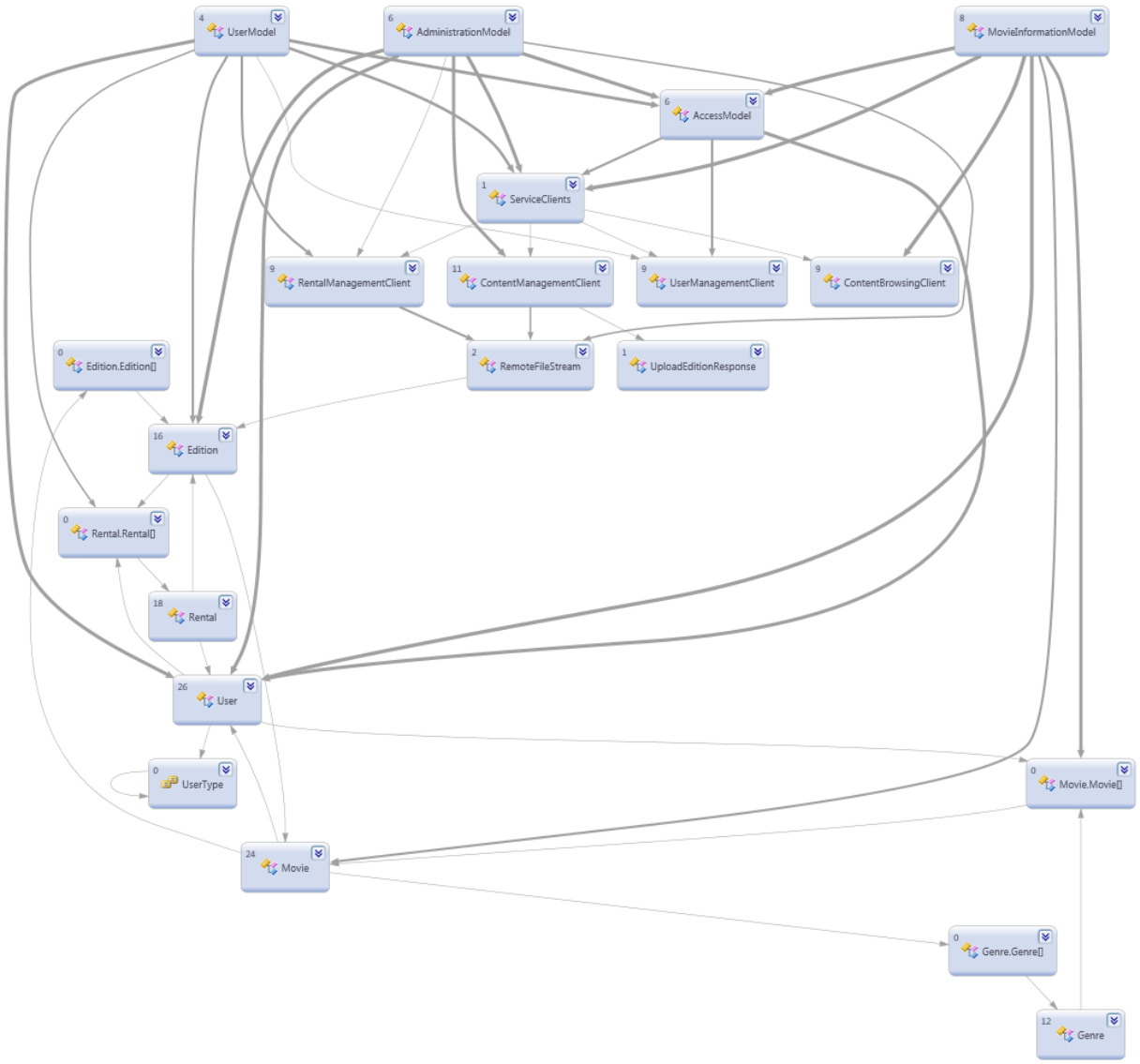

Figure D.1: The Model dependencies.

Figure D.2: The Model-Viewmodel relationship dependencies.

D.2 Service API

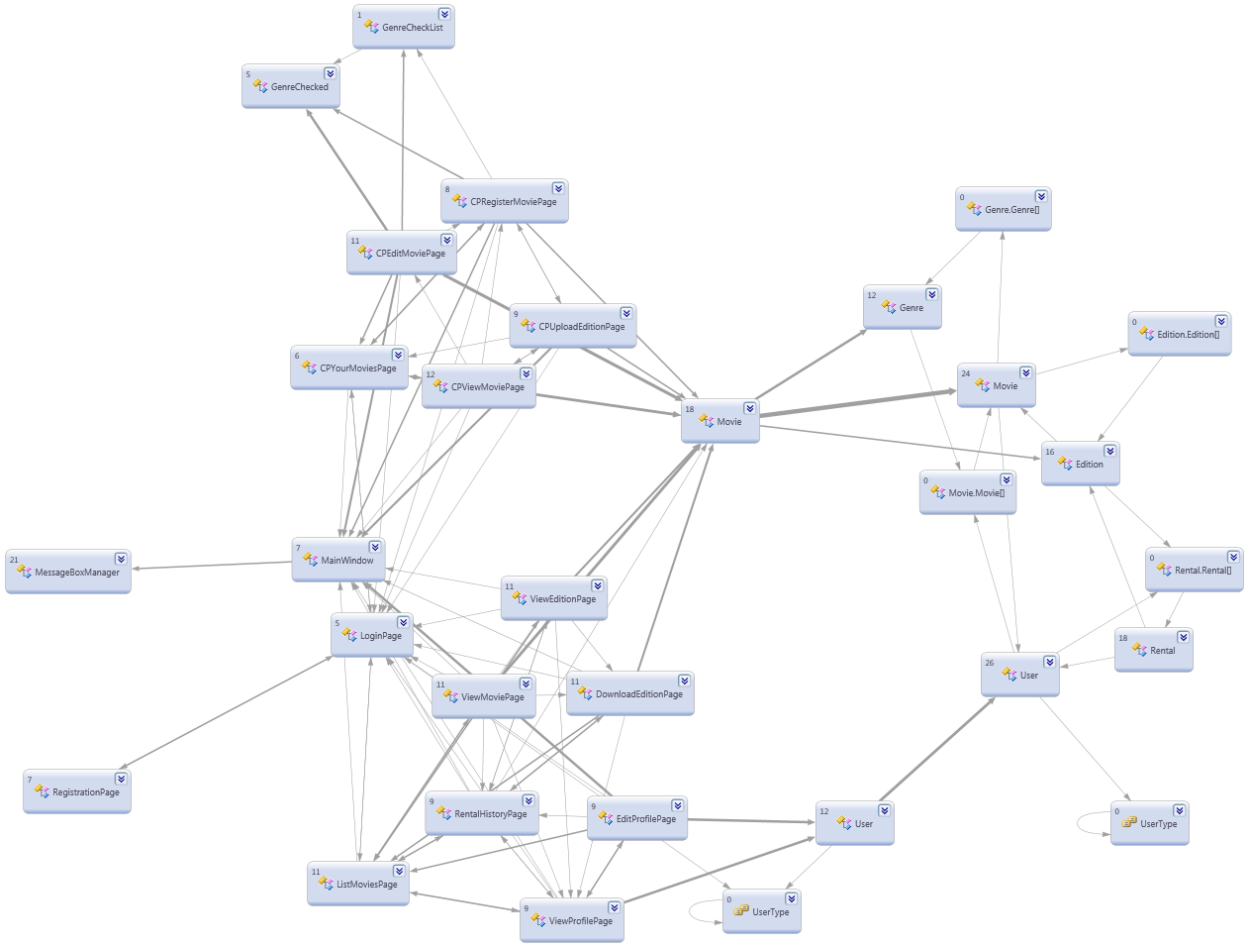

Figure D.3: The View dependencies.

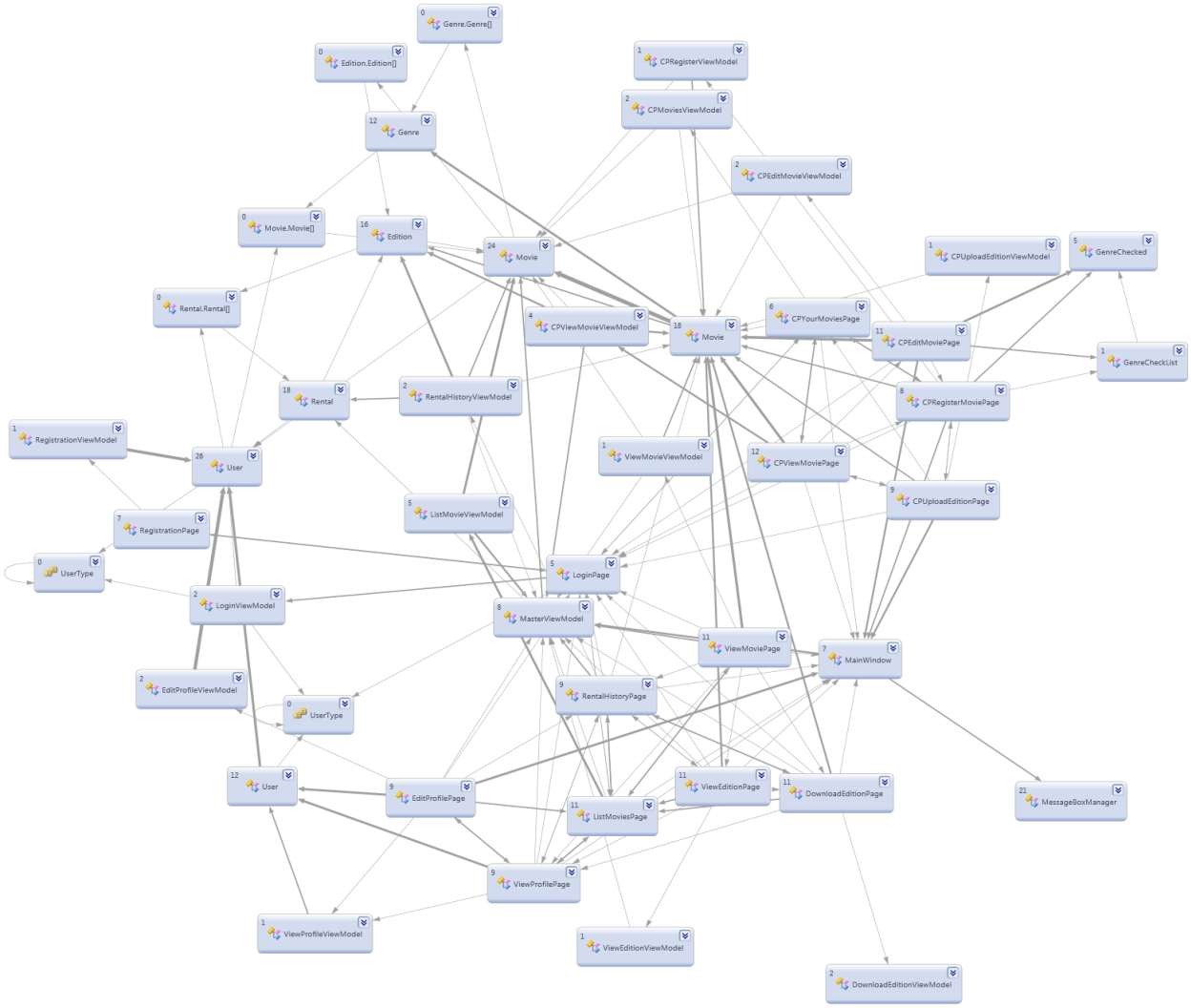

Figure D.4: The View-Viewmodel relationship dependencies.

E Test results

This appendix describes the results of all our tests. Our handwritten, automated tests are arranged in tables in section E.1. Our usability tests are described in section E.3.

E.1 Automated test results

This section shows the result of running our automated tests the day following the code freeze.

E.1.1 Scenario tests

Test title	Test Purpose	Result
AddGenreTest	Verify that you can add a genre to a movie	Pass
AddOne	Verify that if I add a new movie, the first movie to be returned is the new one	Pass
AddOneInFuture	Verify that movies with a release date in the future, will not appear in the newest movies	Pass
AddOneWithoutReleaseDate	Verify that movies without a release date, will not appear in the newest movies	Pass
AdminRentalHistory	Verify that a admin has no rental history	Pass
BrowseKnownGenreTest	verify that when the user browses for a specific known genre, all movies with that genre gets returned	Pass
ContentproviderRentalHistory	Verify that a contentProvider has no rental history	Pass
DeleteMovieEdition	Verify that it is possible to delete a movie edition	Pass
DeleteMovieEditionFromOther Provider	Verify that it is not possible to delete a movie edition, from another content provider	Pass
DeleteMovieFromOtherProvider	Verify that it is not possible to delete a movie that belongs to another content provider	Pass
DeleteMovieTest	Verify that deletion of a movie is possible	Pass
EditMovieFromOtherProvider	Verify that it is not possible to edit a movie uploaded by another content publisher	Pass
EditMovieInformationInvalid MovieIdTypeTest	Verify that the method throws the correct exception when called by an account with an insufficient user type	Pass
EditMovieInformationInvalid UserTypeTest	Verify that the method throws the correct exception when called by an account with an insufficient user type	Pass
EditMovieInformationValidTest	Verify that the method changes the values of the movie	Pass
EditMoviePartOfInfo	Verify that it is possible to only update part of the information about a movie	Pass
EditMoviePartOfInfoInvalid Values	Verify that a field is only updated, if the new value is valid	Pass
EditMoviePartOfInfoMixed ValidInvalid	Verify that fields with new valid values will be updated, even when other fields will not be updated, because of invalid values	Pass
EditPartOfProfileInfoTest	Verify that it is possible to edit only part of a user's profile	Pass
EditPartOfProfileOnlyPassword Test	Verify that it is possible to edit a user's password, and nothing else	Pass

EditProfileTest	Verify that it is possible to edit a user profile	Pass
GetAllGenresTest	verify that all genres in the database gets returned by GetAllGenres	Pass
GetAllMovies	Verify that All work as intended	Pass
GetCurrentRentalsTest	Verify that the user only gets current rentals and that they all belong to him	Fail
GetMovieInformationInvalid MovieIdTest	Verify that the method returns null when called with a movie ID that doesn't corrospond to a movie in the database	Pass
GetMovieInformationValidTest	Verify that the method returns the correct data	Pass
GetUnreleasedMovieInfoFuture Release	Verify that even though editions have been added to a movie, they will not appear / be passed to the clients, if the movie is not released	Pass
GetUnreleasedMovieInfoNo ReleaseDate	Verify that even though editions have been added to a movie, they will not appear / be passed to the clients, if the movie is not released	Pass
InsufficientAccessDeleteMovieTest	Verify that only Content Providers can delete movies	Pass
InvalidValuesRegisterMovieTest	Verify that it is not possible to use invalid values in the method	Pass
Limiting	Verify that it is possible to limit the amount of movies returned	Pass
LoginWithDifferentUsernameAnd Password	Verify that even though a user with a given username exists, and a user with a given password exists, login will fail if those two users are not the same	Pass
LoginWithExistingUser	Verify that login is possible when using the right username and password	Pass
LoginWithNonExistingUser	Verify it is not possible to login, when no users with the given username and password exists	Pass
LoginWithWrongPassword	Verify that even though a user with a given username exists, login will fail if the password is wrong	Pass
LoginWithWrongUsername	Verify that even though a user with a given password exists, login will fail if the username is wrong	Pass
LogoutValidToken	Verify that it is possible to logout, when specifying a valid token	Pass
MostDownloadedMultipleEditions	Verify that even though the rentals are split between multiple editions, the right movies in the right order is still returned	Pass
MostDownloadedWithRentals	Verify that when trying to get the most downloaded movies, the right movies are returned in the right order	Pass
MultipleRentalHistory	Verify that a user wtih serveral movies in rental history and with multiple instance of the same movie will return the correct list	Fail
NotAUserRentMovieTest	Verify that only users can rent movies	Pass
NullTokenRegisterMovieTest	Verify that it is not possible to call the method with a null token	Pass
RegisterMovieTest	Verify that a content provider is able to register movies in the database	Pass
RemoveGenreTest	verify that you can remove a genre from a movie	Pass

RentalHistoryNoRentals	Verify that you will get a empty list from a user with no rental history	Pass
RentalHistoryTest	Verify that it is possible to retreive list of the user rental history	Pass
RentalOfMovieWithFutureRelease	Verify it is not possible to rent a movie with a release date in the future	Pass
RentalOfMovieWithoutRelease Date	Verify it is not possible to rent a movie without a release date	Pass
RentMovieTest	Verify that it is possible to rent a movie	Pass
SearchBadSpelling	verify that search results includes movies with spelling errors in the title	Pass
SearchDifferentCase	verify that a movie is returned when a user searches for the title but with incorrect case	Pass
SearchExcactTitle	Verify that a movie gets returned when the user searches for its exact title	Pass
SearchLimit	verify that putting a limit on the search results actually limits the number of returned movies	Pass
SearchMoreTokens	verify that a movie will be returned, even if tokens not in the name is part of the search string	Pass
SearchOrder	verify that the exact movie title match will be ordered before a partly match	Pass
SearchPartlyTitle	verify that a movie is returned when a user searches for a part of the title	Pass
SearchTokenMatchCountOrder	verify that the search results is ordered by the number of token matches	Pass
SearchVeryBadSpelling	verify that the search result does not include words that are too badly spelled	Pass
SearchWithoutResult	verify that an empty collection is returned when the user searches for a title not in the database	Pass
SignUpWithEmptyEmail	Verify that email has to be set	Pass
SignUpWithEmptyPassword	Verify that password has to be set	Pass
SignUpWithEmptyUsername	Verify that username has to be set	Pass
SignUpWithExistingUsername	Verify that it is not possible to signup with a username that is already in use	Pass
SignUpWithInvalidInfo	Verify that type is automatically set to user, token is reset and ID is auto-generated when trying to set those settings to invalid values	Pass
SignUpWithValidInfo	Verify that it is possible to sign up	Pass
WrongUserTypeRegisterMovie Test	Verify that it is not possible to use the method as a user of type User	Pass

E.1.2 Service tests

Test title	Test Purpose	Result
AllGenresValidServiceTest	Verify that GetGenres return some genres	Pass
AllGenresWihtoutTokenService Test	Verify that without a token, GetGenres doesn't return anything	Fail
DeleteEditionValidServiceTest	Verify that you can delete a movie edition	Fail

DeleteMovieValidServiceTest	Verify that you can delete a movie	Fail
EditMovieInsufficientRightsServiceTest	Verify that normal users cannot edit movies	Pass
EditMovieValidServiceTest	Verify that you can edit movies	Fail
EditUserNullServiceTest	Verify that EditUser fails on invalid input	Pass
EditUserValidServiceTest	Verfiy that you can edit a user	Fail
GetAllMoviesGenreServiceTest	Verify that movies can be limited to a specific genre	Fail
GetAllMoviesLimitServiceTest	Verify that you can limit the number of movies returned	Fail
GetAllMoviesMostDownloadedServiceTest	Verify that movies can be sorted by number of rentals	Fail
GetAllMoviesNewestServiceTest	Verify that movies can be sorted by newest	Fail
GetAllMoviesValidServiceTest	Verify that movies are returned on valid input to GetMovies	Fail
GetAllMoviesWithoutTokenServiceTest	Verify that a token is needed to browse movies	Pass
GetMovieInformationUnknownMovieServiceTest	Verify that GetMovieInformation doesn't return anything when an unknown movie is referred to	Fail
GetMovieInformationValidServiceTest	Verify that GetMovieInformation gets information about a movie	Pass
GetMovieInformationWithoutToken ServiceTest	Vecrify that without a token, GetMovieInformation doesn't return anything	Pass
GetRentalsNullServiceTest	Verify that GetRentals fail without a user toke	Pass
GetRentalsValidServiceTest	Verify that you can get a users rentals	Fail
LoginValidServiceTest	Verify that you can log in	Pass
LoginWrongPasswordServiceTest	Verify that the user won't get logged in, if using a wrong password	Fail
LogoutValidServiceTest	Verify that you can log out	Pass
RegisterMovieInsufficientRightsServiceTest	Verify that normal users cannot register movies	Pass
RegisterMovieValidServiceTest	Verify that you can register a movie	Fail
RentMovieContentProviderServiceTest	Verify that content providers cannot rent movies	Fail
RentMovieValidServiceTest	Verify that users can rent movies	Fail
SearchValidServiceTest	Verifies that movies are returned from a search	Fail
SearchWithoutQueryServiceTest	Verify that nothing is returned for a null query	Pass
SearchWithoutTokenServiceTest	Verifies that nothing is returned without a user token	Pass
SignupMissingInfoServiceTest	Verify that you cannot create a user without basic information	Pass
SignupValidServiceTest	Verify that you can create a new user	Pass

E.1.3 Automated Graphical User Interface tests

Test title	Test Purpose	Result
GuiTest01CreateNewUser	See if it's possible to create a new user account	Pass
GuiTest02UserLogin	See if it's possible for the user to log in	Pass
GuiTest03UserEditUser Information	See if it's possible for the user to edit his/her information	Fail
GuiTest04UserLogout	See if it's possible for the user to log out	Fail
GuiTest05UserViewAllMovies	See if it's possible for the user to view all the offered movies	Pass

GuiTest06UserViewMoviesBy Newest	See if it's possible for the user to sort all movies with the newest movie first	Fail
GuiTest07UserSearchForMovie	See if it'sp ossible for the user to search for a movie	Pass
GuiTest08UserViewMoviesBy Genre	See if it's possible for the user to see all movies from a specific genre	Fail
GuiTest09UserRentSpecific MovieEdition	See if it's possible for the user to rent a specific movie edition	Pass
GuiTest10UserRentAndView Rentals	See if it's possible for the user to rent a movie and find it in his/her rental list	Fail
GuiTest12CPLoginLogout	See if the content provider can log in and log out	Fail
GuiTest13CPRegisterMovie	See if the content provider can register a movie	Pass

E.1.4 Failed automatic tests

This section describes which of our automatic tests that fail and what error they fail with.

Test title	Fail reason
GetCurrentRentalsTest	Threw exception: "ArgumentNullException: Value cannot be null. Parameter name: String"
MultipleRentalHistory	Threw exception: "System.InvalidOperationException: There is already an open DataReader associated with this Command which must be closed first.
DeleteEditionValidServiceTest	Threw exception: "System.ServiceModel.Fault-Exception"
DeleteMovieValidServiceTest	"AssertIsTrue failed. DeleteMovie failed"
EditMovieValidServiceTest	Assert.AreEqual failed. Expected: <New title>. Actual:<Ocean's Eleven>. Movie title wasn't changed
EditUserValidServiceTest	System.ServiceModel.Dispatcher.NetDispatcherFault-Exception: The formatter threw an exception while trying to deserialize the message: There was an error while trying to deserialize parameter
GetMovieInformationUnknownMovie-ServiceTest	Assert.IsFalse failed. GetMovieInformation didn't fail
GetAllMoviesGenreServiceTest	System.ServiceModel.Dispatcher.NetDispatcherFault-Exception: The formatter threw an exception while trying to deserialize the message: There was an error while trying to deserialize parameter
GetAllMoviesLimitServiceTest	Assert.IsTrue failed. Result is false
GetAllMoviesMostDownloadedService-Test	Assert.IsTrue failed. Result is false
GetAllMoviesNewestServiceTest	System.ServiceModel.Dispatcher.NetDispatcherFault-Exception: The formatter threw an exception while trying to deserialize the message: There was an error while trying to deserialize parameter
GetAllMoviesValidServiceTest	System.ServiceModel.Dispatcher.NetDispatcherFault-Exception: The formatter threw an exception while trying to deserialize the message: There was an error while trying to deserialize parameter
GetRentalsValidServiceTest	Assert.IsTrue failed. GetRentals failed
LoginWrongPasswordServiceTest	Assert.IsFalse failed. Login didn't fail.

RegisterMovieValidServiceTest	System.ServiceModel.Dispatcher.NetDispatcherFault-Exception: The formatter threw an exception while trying to deserialize the message: There was an error while trying to deserialize parameter
RentMovieContentProviderServiceTest	System.ServiceModel.Dispatcher.NetDispatcherFault-Exception: The formatter threw an exception while trying to deserialize the message: There was an error while trying to deserialize parameter
RentMovieValidServiceTest	Assert.AreNotEqual failed. Expected any value except:<0>. Actual: <0>. rentals is null
SearchValidServiceTest	Assert.IsTrue failed. Result is false
GuiTest03UserEditUserInformation	Could not find 'Close application' control
GuiTest04UserLogout	Could not find 'Close application' control
GuiTest06UserViewMoviesByNewest	Could not find 'MovieListBox' control
GuiTest08UserViewMoviesByGenre	Could not find 'Close application' control
GuiTest10UserRentAndViewRentals	Could not find 'Close application' control
GuiTest12CPLoginLogout	Could not find 'Close application' control

E.2 Manual GUI test

Test 3 - User, edit user profile (GuiTest03UserEditUserInformation).
Tests whether it's possible for the user to edit his/her information or not.
This test covers the requirement "User - Edit profile".

1. Login as the user "Smith"
2. Navigate to the Edit Profile Page
3. Change Full Name to "Neo Smith"
4. Click the "Save changes" button
5. See that the full name now is "Neo Smith"
6. Close the window

This test passes.

Test 4 - User, logout as user (GuiTest04UserLogout).
Tests if the user can log out from the service.
This test covers the requirement "Optional - High priority - Logout".

1. Login as the user "Smith"
2. Click the "Logout" at the top right of the screen
3. Close the window

This test passes.

Test 6 - User, view list of all movies sorted by release date (GuiTest06UserViewMoviesByNewest).
Tests if the user can see all movies sorted by their release date.
This test covers the requirement "User - View a list of all movies" and "Optional - High priority - View movielists with different sorting".

1. Login as the user "Smith"
2. Navigate to the View Movie List Page
3. Sort by Newest and All
4. View Movie for Ocean's Eleven
5. Click the "List Movies" button
6. View Movie for The Matrix
7. Assert that the release date is earlier than Ocean's Eleven
8. Close the window

This test passes.

Test 8 - User, view all movies of a specific genre (GuiTest08UserViewMoviesByGenre).
Tests if the user can see all movies of a certain genre.
This test covers the requirement "Optional - High priority - View movielists with different sorting".

1. Login as the user "Smith"
2. Navigate to the View Movie List Page
3. Sort movies by Newest and Sci-fi
4. Assert that The Matrix, The Matrix Reloaded and The Matrix Revolution are in the list
5. Close the window

This test passes.

Test 10 - User, view current rentals (GuiTest10UserRentAndViewRentals).
Tests if the user can see his/her current rentals.
This test covers the requirement "Optional - High priority - View rental history".

1. Login as the user "Smith"
2. Rent the movie "The Lord of the Rings - The Fellowship of the Ring - SD"
3. Click the "Your Rentals" movie
4. Assert that "The Lord of the Rings - The Fellowship of the Ring - SD" is in the list
5. Close the window

This test passes.

Test 11 - User, download current rentals.
Tests if the user can download a movie that he/she has rented.
This test covers the requirement "User - Download media".

1. Login as the user "Smith"
2. Click the "Your rentals" button
3. Find a movie in the list
4. Click the "View movie" button
5. Click the "Download movie" button
6. Check that the movie is saved to the chosen directory
7. Close the window

This test does not pass. The client crashes.

Test 12 - CP, login & logout (GuiTest12CPLoginLogout).
Tests if the content provider can log in and log out.
This test covers the requirement "Content Provider - Login" and "Optional - High priority - Logout".

1. Login as Universal (test content provider)
2. Assert that the "Logout" button exists
3. Click the "Logout" button
4. Assert that the Welcome screen is shown

This test passes.

Test 13 - CP, register and upload movie.
Tests if the content provider register a movie and upload a file.
This test covers the requirement "Content Provider - Upload media", "Optional - High priority - Movie release dates" and "Optional - Medium priority - Movie editions".

1. Login as Universal (test content provider)
2. Click the "Register movie" button
3. Fill the information and click the "Register movie" button
4. Click the "Upload Movies" button
5. Fill the textboxes
6. Click the "Upload movie" button

This test does not pass. The file is never uploaded to the server.

Test 15 - CP, upload new edition to already registered movie.
Tests if the content provider can upload an edition to a movie that has already been registered.
This test covers the requirement "Content Provider - Upload media" and "Optional - Medium priority - Movie editions".

1. Login as Universal (test content provider)
2. Select a movie
3. Click the "Upload new edition"
4. Browse for a file
5. Click the "Upload" button

This test does not pass. The file is never uploaded to the server.

Test 16 - CP, edit information about a movie.
Tests if the content provider can edit the information about a movie.
This test covers the requirement "Content Provider - Edit uploaded media".

1. Login as test content provider
2. Click the "Your Movies" button
3. Find the movie that is to be edited
4. Fill out the information and click the "Save changes" button

This test does not pass. The client crashes when one tries to save the changes.

Test 17 - CP, delete movie.
Tests if the content provider can delete a movie that he/she has registered.
This test covers the requirement "Content Provider - Delete media".

1. Login as Universal (test content provider)
2. Click the "Your Movies" button
3. Find the movie that is to be deleted
4. Click the "View Movie" button
5. Click the "Delete movie" button

This test passes.

E.3 Usability tests

This section describes the results of our usability tests. What influence the usability tests had on our graphical user interface and how we performed them is described in the report[1].

For the first usability test we got the feedback that even though our interface was easy to navigate through, we lacked user conformation all our test users felt uncertain that their actions was saved in the database.

For the second usability test we got no comments regarding the missing feeling of confirmation, but we did get the feedback that some of our paths was to obscured and not very intuitive. Generally the feedback told us that we needed to do something about our navigation and of our and how it was prestended to the user.

E.3.1 Usability goals and scenarios

List of usability goals:

- The user should be able to finish all given tasks within a time periode of 45 seconds.
- The user should be able to maneuver the client without need to ask the tester qustions.
- The user should be positive of the design.
- The user should be able to recommend the service to his or her friends.

List of usability scenarios:

- Your have heard of this new movie rental service and you would like to sign up for it.
- You would like to rent "Batman the Begining" from the service.
- You would like to see what movies are most popular at the moment.
- You have gotten a new email and would like to change your profile so it uses your new email.
- As a movie company employee, you would like to upload some movies to the service.
- You have uploaded a movie with the wrong title - change it
- As an admin for the service you've seen some companies upload explicit material to service, delete those companies from the service.
- You would like to see a list of all the users who are using the service.

[1]The influence is described in section 5.3.3 on page 23 and how we performed the tests is described in 8.1.3 on page 33

E.4 Code coverage results

- Service — 58%
  - RentItService.Services — 0%
  - RentItService.Exceptions — 17%
    - InsufficientRightsException — 17%
    - NoMovieFoundException — 25%
    - NotAUserException — 25%
    - UnknownGenreException — 0%
    - UsernameInUseException — 25%
    - UserNotFoundException — 0%
  - RentItService.Library — 61%
    - FileRequest — 0%
    - RemoteFileStream — 0%
    - StringDifference — 100%
  - RentItService.Entities — 75%
    - Edition — 38%
    - Genre — 88%
    - Movie — 71%
    - Rental — 50%
    - User — 90%
  - RentItService — 100%
    - RentItContext — 100%
  - RentItService.Mapping— 100%
    - EditionMap — 100%
    - GenreMap — 100%
    - MovieMap — 100%
    - RentalMap — 100%
    - UserMap — 100%

F Original Use Cases

The use cases here are the first list of use cases we agreed on with SMU. These later changed into what can be found in our Requirements chapter in section 3.4 on page 8.

F.1 User account management

- A user is signing up for the service.
- A user log in to the service.
- A user edits his/her personal information.

F.2 Media browsing

- A user is browsing the newest added movies.
- A user is browsing the most downloaded movies.
- A user is browsing movies by genre.
- A user is browsing all movies and sorts them by their name or genre.
- A user is browsing all movies he/her previously rented.
- A user is searching for at movie by its name.

F.3 Media rental

- A user is renting a movie.
- A user is viewing information about a movie.

F.4 Content management

- A content manager uploads a new movie, and enters information about that movie.
- A content manager edits information for a movie.
- A content manager deletes a movie.

F.5 System management (?)

- A system manager browses all the content managers.
- A system manager creates a new content manager.
- A system manager deletes a content manager.
- A system manager browses all the users.
- A system manager deletes a user.

G GUI images

This appendix contains the full array of GUI images that we have - both the early hand-drawn sketches for usability tests and the actual GUI prototype. The images may be placed a bit odd, so refer to two sections to see where each image belongs.

G.1 Hand-drawn sketches

The figures G.1 to G.17 are part of our hand-drawn sketches for the GUI.

Figure G.1: Admins view user

Figure G.2: Admins contentprovider list

Figure G.3: Admins list of users

Figure G.4: Admins welcome screen

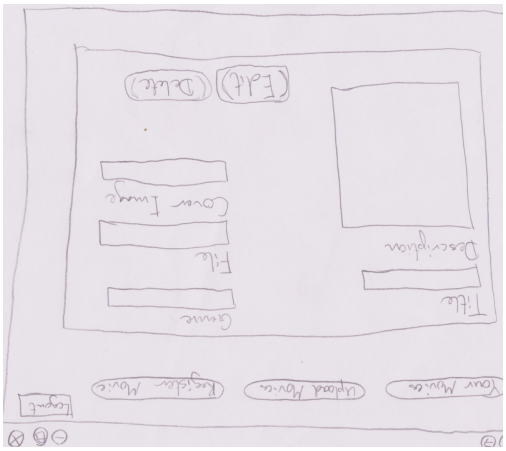

Figure G.5: Content provider view movie

Figure G.6: Content providers list of uploads

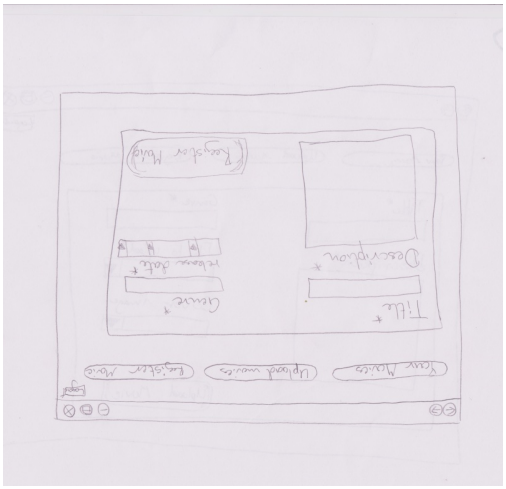

Figure G.7: Content providers register movie screen

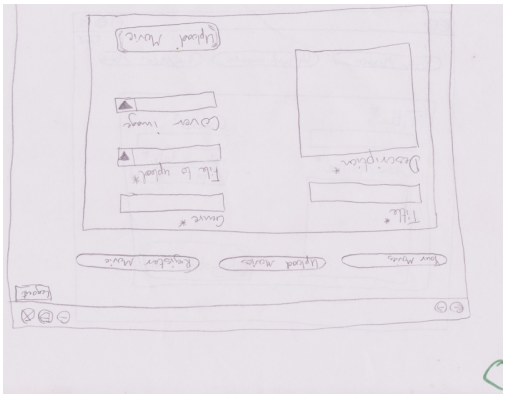

Figure G.8: Content providers upload movie screen

Figure G.9: Content providers welcome screen

Figure G.10: Users edit profile screen

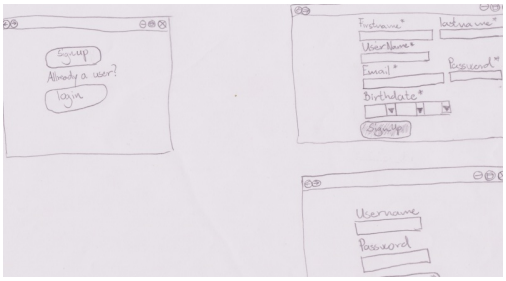

Figure G.11: The signup and login page

Figure G.12: List of the most downloaded movies on the service

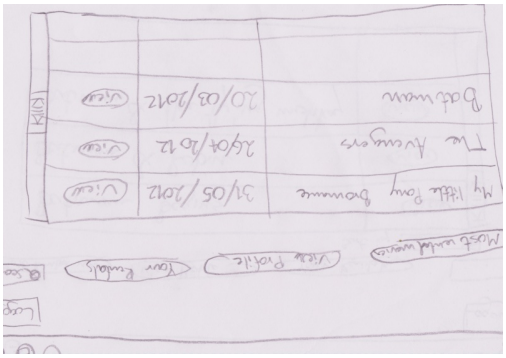

Figure G.13: List of your rental history

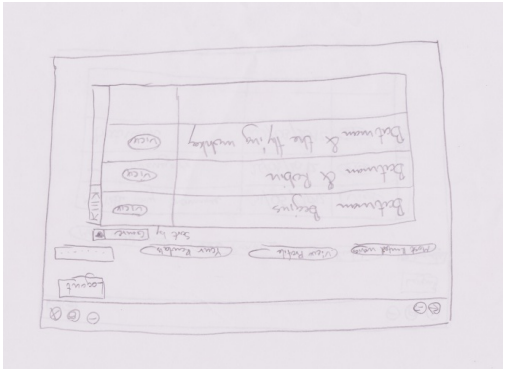

Figure G.14: Result of a search on batman begins

Figure G.15: Users welcome page

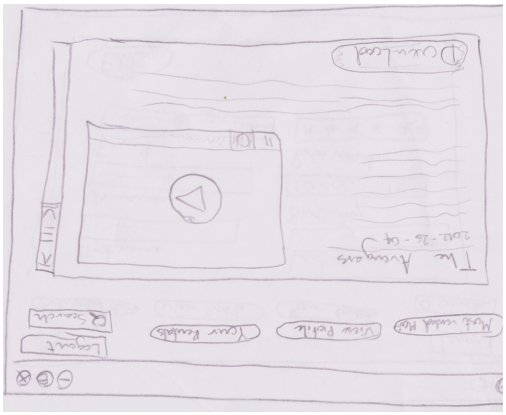

Figure G.16: View the details of a movie

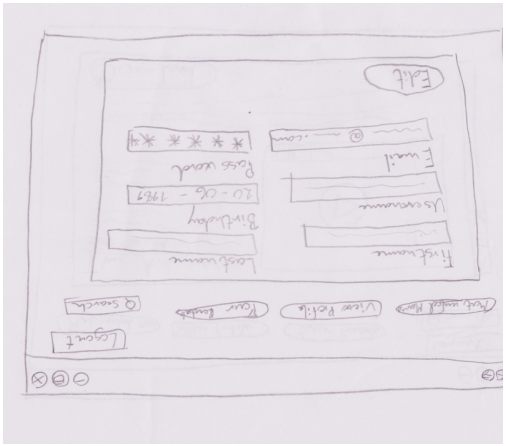

Figure G.17: View profile information

G.2 GUI prototype

The figures G.18 to G.33 are part of our prototype of the GUI.

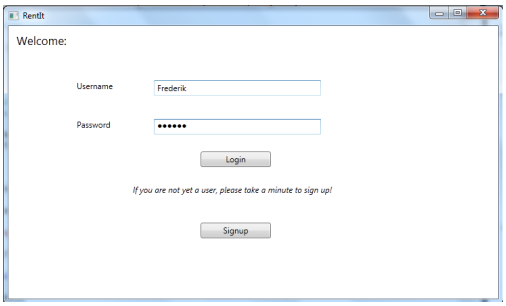

Figure G.18: The main login screen

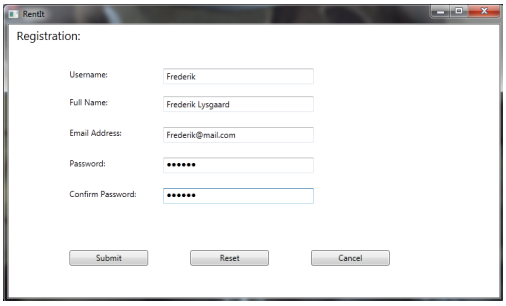

Figure G.19: The screen that is used for creating a new user account

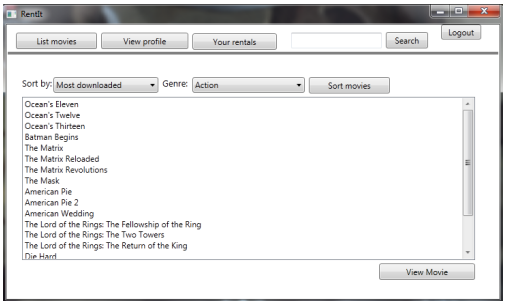

Figure G.20: The list of all the movies that the service has

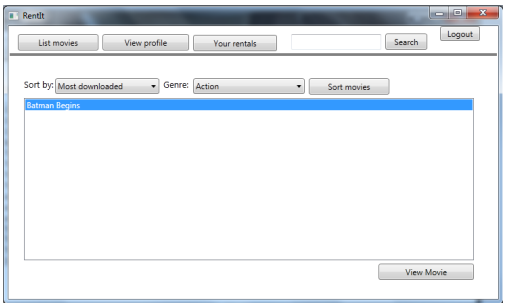

Figure G.21: The screen that shows the search results

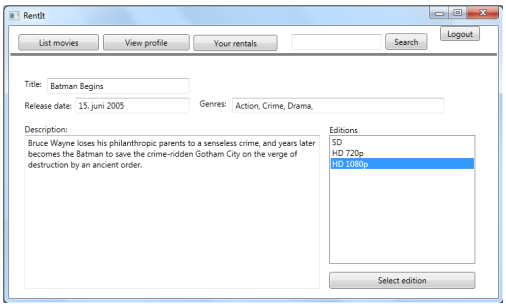

Figure G.22: The screen that shows the information about a movie

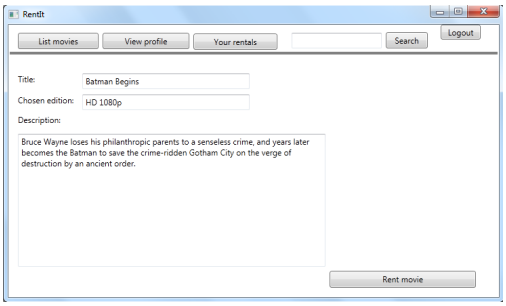

Figure G.23: The screen that lets the user rent an edition

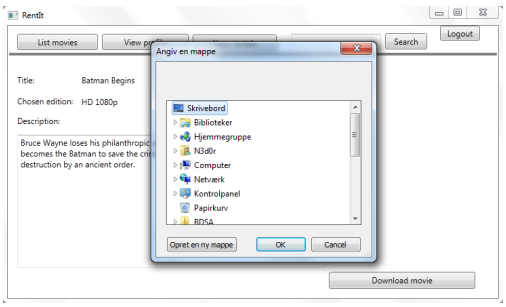

Figure G.24: The screen that lets the user download a movie he/she has rented

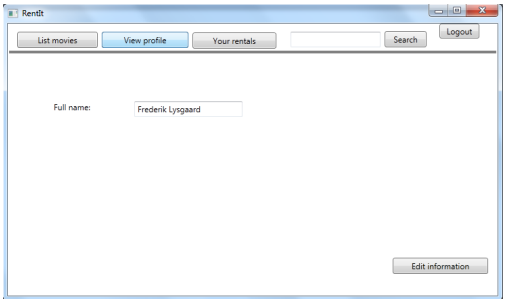

Figure G.25: The screen that lets the user view his/her profile information

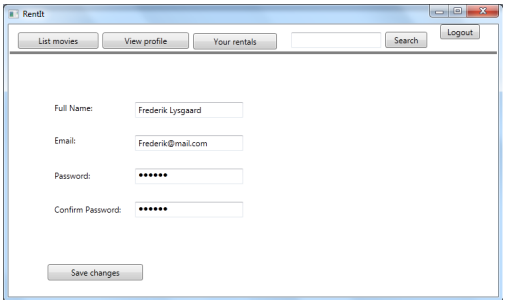

Figure G.26: The screen that lets the user edit his/her profile

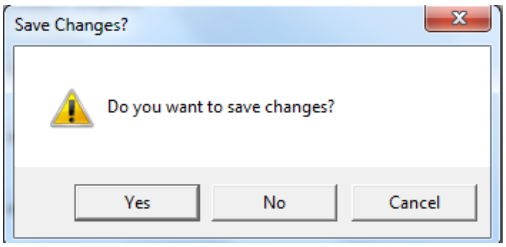

Figure G.27: The confirmationbox that asks if the user wants to save the changes

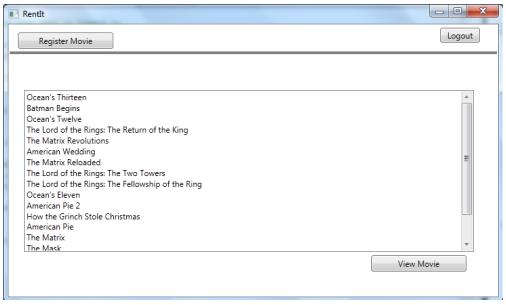

Figure G.28: The screen that lets the content provider see all the movies that he/she has registered

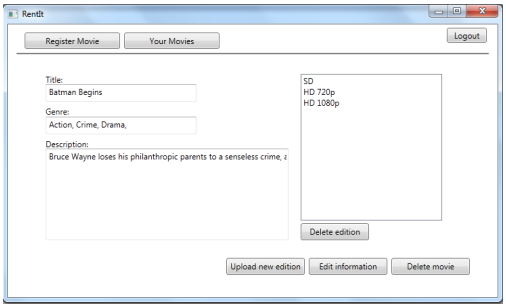

Figure G.29: The screen that lets the content provider see the information about a movie he/she has registered

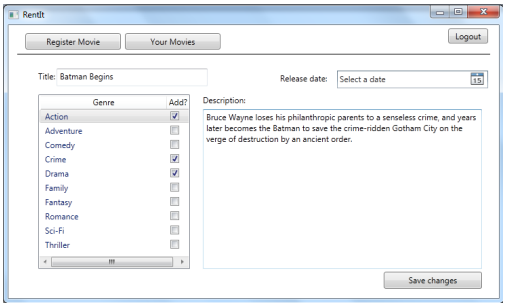

Figure G.30: The screen that lets the content provider edit a movie that he/she has registered

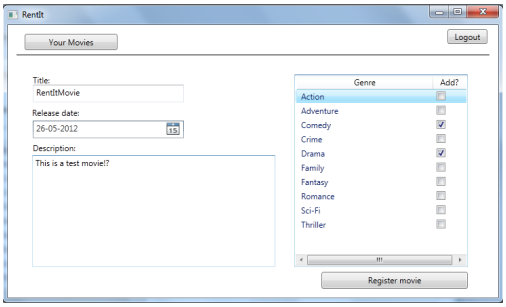

Figure G.31: The screen that lets the content provider register a new movie

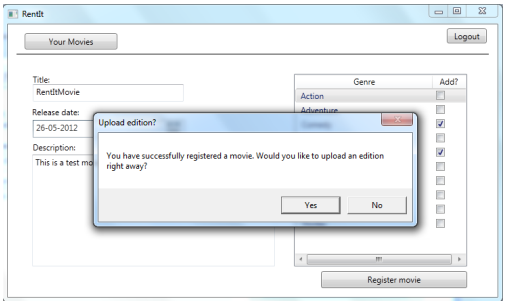

Figure G.32: The popup box that shows up when the content provider has registered a movie

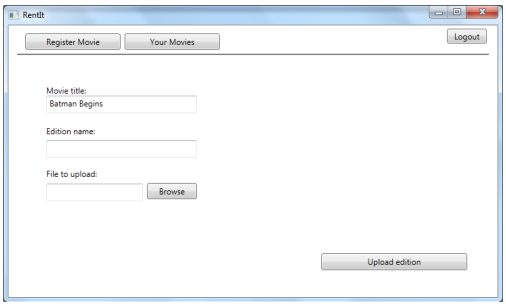

Figure G.33: Content provider can upload an edition from here.

H F# Handins

This chapter of the appendix contains our handins for the F# assignments. To make sure they could fit on the pages, we had to break up some of the lines in the code. We do feel that we have been able to make it look decent, however. The original .fs files have also been included on the DVD including with the report.

H.1 F# Handins - Frederik Lysgaard

H.1.1 HandIn 1

```
(* Student name: Frederik Roden Lysgaard
   Mail: Frly@itu.dk *)
module Module1

//Opgave 1
let sqr x = x*x

//Opgave 2
let pow x n = System.Math.Pow(x, n)

//Opgave 3
let dup x : string = x+x

//Opgave 4
let rec dupn (s : string)  (x : int) = if x = 1 then s else s + dupn s (x-1)

//Opgave 5
let timediff (hh1, mm1) (hh2, mm2) = (hh2*60 + mm2) - (hh1*60 + mm1)

//Opgave 6
let minutes (hh, mm) = timediff (00, 00) (hh, mm)
```

H.1.2 HandIn 2

```
(* Student name: Frederik Roden Lysgaard
   Mail: Frly@itu.dk *)
module Handin2

//Opgave 7
let downTo (n : int) = if n < 1 then [] else [n .. -1 .. 1]

let downTo2 (n :int) =
    match n with
    | n when n < 1 -> []
    | _ -> [n .. -1 .. 1]

//Opgave 8
```

```
let rec removeEven (list :int list) =
    match list with
    | [] -> []
    | [x] -> [x]
    | a :: b :: rest -> a :: removeEven rest


//Opgave 9
let rec combinePair (list :int list) : (int*int) list =
    match list with
    | [] -> []
    | [x] -> []
    | a :: b :: rest -> (a, b) :: combinePair rest

//Opgave 10
let explode (s :string) : char list = List.ofArray (s.ToCharArray())

let rec explode2 (s :string) : char list =
    match s with
    | s when s.Length < 1 -> []
    | _ -> [s.[0]] @ explode2 (s.Substring 1)

//Opgave 11
let implode (s :char list) : string =
        List.foldBack (fun elem acc -> string (elem) + string(acc)) s ""

let implodeRev (s :char list) : string =
        List.fold (fun elem acc -> string (acc) + string(elem)) "" s

//Opgave 12
let toUpper s =
        implode (List.map System.Char.ToUpper (explode s))

let toUpper1 =
        explode >> List.map System.Char.ToUpper >> implode

let toUpper2 s :string =
        explode s |> (implode << List.map System.Char.ToUpper)

//Opgave 13
let palindrome (s :string) =
        (explode s |> List.map System.Char.ToUpper |> implodeRev) = toUpper s

//Opgave 14
let rec ack (m, n) =
    match (m, n) with
    | (m, n) when m < 0 || n < 0 -> failwith "The Ackermann
                        function is defined for non-negativ numbers only"
    | (m, n) when m = 0 -> n + 1
    | (m, n) when n = 0 -> ack (m - 1, 1)
    | _ -> ack (m - 1, ack (m, n - 1))
```

```
//Opgave 15
let time f =
  let start = System.DateTime.Now in
  let res = f () in
  let finish = System.DateTime.Now in
  (res, finish - start)

let timeArg1 f a = time (fun () -> f a)
```

H.1.3 HandIn 3

H.1.4 HandIn 4 & 5

H.2 F# Handins - Jacob Claudius Grooss

H.2.1 HandIn 1

```
module HandIn1
//Exercise 1
let sqr x = x * x

//Exercise 2
let pow x n = System.Math.Pow(x, n)

//Exercise 3
let dup (s:string) = s + s

//Exercise 4
let rec dupn (s:string, x) = if x = 0 then s else s + dupn(s, x - 1)

//Exercise 5
let timediff (hh1, mm1)(hh2, mm2) = (hh2 * 60 + mm2) - (hh1 * 60 + mm1)

//Exercise 6
let minutes (hh, mm) = timediff (00,00)(hh,mm)
```

H.2.2 HandIn 2

```
//Exercise 7
let rec downTo n = if n < 1 then [] else n :: downTo(n - 1)

let rec downTo2 n =
    match n with
                | n when n < 1 -> []
        | 1 -> [1]
        | _ -> n :: downTo(n-1)
```

```
//Exercise 8
let rec removeEven (xs: int list) =
    match xs with
        | [] -> []
        | [xs] -> [xs]
        | xs :: xy :: rs -> xs :: removeEven(rs)

//Exercise 9
let rec combinePair (xs: int list) =
    match xs with
        | [] -> []
        | [xs] -> []
        | xs :: xy :: rs -> xs :: combinePair(rs)

//Exercise 10
let explode (s:string) =
    s.ToCharArray() |> List.ofArray

let rec explode2 (s:string) =
    match s with
        | s when s.Length < 1 -> []
        | _ -> s.[0] :: explode2 (s.Substring 1)

//Exercise 11
let implode (s:char list) =
    List.foldBack (fun str ch -> string(str) + string(ch)) s ""

let implodeRev (s:char list) =
    List.fold (fun str ch -> string(ch) + string(str)) "" s

//Exercise 12
let toUpper (s:string) =
    implode (List.map (fun x -> System.Char.ToUpper x) (explode s))

let toUpper1 (s:string) =
    explode >> (List.map (System.Char.ToUpper)) >> implode

let toUpper2 (s:string) =
    explode s |> (implode << List.map System.Char.ToUpper)

//Exercise 13
let palindrome (s:string) =
    (explode s |> implodeRev |> toUpper) = toUpper s

//Exercise 14
let rec ack (m, n) =
    match (m, n) with
        | (m, n) when m < 0 || n < 0 -> failwith "The Ackermann function
                                    is defined for non negative numbers only."
        | (m, n) when m = 0 -> n + 1
        | (m, n) when n = 0 -> ack (m - 1, 1)
```

```
        | (m, n) -> ack (m-1, ack(m, n-1))

//Exercise 15
let time f =
    let start = System.DateTime.Now in
    let res = f () in
    let finish = System.DateTime.Now in
    (res, finish - start);

let timeArg1 f a = time(fun () -> f(a))
```

H.2.3 HandIn 3

```
type 'a BinTree =
    | Node of 'a * 'a BinTree * 'a BinTree
    | Leaf;;

let intBinTree = Node(43, Node(25, Node(56, Leaf, Leaf), Leaf),
Node(562, Leaf, Node(78, Leaf, Leaf)));;

//Exercise 16
let rec inOrder tree =
    match tree with
    | Leaf -> []
    | Node (n, treeL, treeR) ->
    inOrder treeL @ n :: inOrder treeR;;

//Exercise 17
let rec mapInOrder (funct: 'a -> 'b) tree =
    match tree with
    | Leaf -> Leaf
    | Node (n, treeL, treeR) ->
        let left = mapInOrder funct treeL
        let value = funct n
        let right = mapInOrder funct treeR
        Node (value, left, right);;

(* They traverse the tree in different orders,
which can give different results. *)

//Exercise 18
//Doesn't have the right signature, but this was the closest I could get
//to it while getting the correct result
let rec foldInOrder funct acc tree =
    match tree with
    | Leaf -> acc
    | Node (root, treeL, treeR) ->
    funct (foldInOrder funct acc treeL) root (foldInOrder funct acc treeR);;

let func left root right = left + root + right;;
```

```
let seed = 1;;

let testingFol = foldInOrder func seed intBinTree;;

//Exercise 19 / 21 / 22
type expr =
    | Const of int
    | If of expr * expr * expr
    | Bind of string * expr * expr
    | Var of string
    | Prim of string * expr * expr

let rec evaluate expr (dict:
System.Collections.Generic.Dictionary<string, expr>) =
    match expr with
    | Const(i) ->
        i
    | If(expr1, expr2, expr3) ->
        if ((evaluate expr1 dict) > 0 || (evaluate expr1 dict) < 0)
        then (evaluate expr2 dict) else (evaluate expr3 dict)
    | Bind(var, value, expr1) ->
        dict.Add(var, value)
        evaluate expr1 dict
    | Var(text) when dict.ContainsKey(text) ->
        evaluate (dict.Item text) dict
    | Var(text) ->
        failwithf "Unknown variable '\%s'" text
    | Prim("-", expr1, expr2) ->
        evaluate expr1 dict - evaluate expr2 dict
    | Prim("+", expr1, expr2) ->
        evaluate expr1 dict + evaluate expr2 dict
    | Prim("max", expr1, expr2) ->
        (List.max [evaluate expr1 dict; evaluate expr2 dict])
    | Prim("min", expr1, expr2) ->
        (List.min [evaluate expr1 dict; evaluate expr2 dict])
    | Prim("=", expr1, expr2) ->
        if (evaluate expr1 dict).Equals(evaluate expr2 dict) then 1 else 0
    | Prim(opr, _, _) ->
        (printfn "Operation '\%s' not supported" opr; 0);;

let eval expr =
    evaluate expr (new System.Collections.Generic.Dictionary<string, expr>());;

//Exercise 20
let testingMinus = eval (Prim("-", Const 10, Const 5));;
let testingPlus = eval (Prim("+", Const 10, Const 5));;
let testingMx = eval (Prim("max", Const 10, Const 5));;
let testingMin = eval (Prim("min", Const 10, Const 5));;
let testingEquals = eval (Prim("=", Const 10, Const 5));;

let testingIf1 = eval (If(Const 3, Const 20, Const 18));;
```

```
let testingIf2 = eval (If(Const 0, Const 20, Const 18));;

//Exercise 23
let testingBindVal1 = eval (Bind("troll", Const 20,
Bind("anti-troll", Const 42, Bind("super-troll", Var "troll",
Var "anti-troll"))));;
let testingBindVal2 = eval (Bind("what",
Bind("happens", Const 20, Var "happens"), Var "what"));;
let testingBindVal3 = eval (Bind("lol", Const 1337, Var "lol"));;
let testingVarFail1 = eval (Var("troll"));;
let testingVarFail2 = eval (Bind("fail", Const 117, Var "troll"));;
```

H.2.4 HandIn 4 & 5

H.3 F# Handins - Jakob Melnyk

H.3.1 HandIn 1

```
module Module1

// Exercise 1
let sqr x = x*x

// Exercise 2
let pow x n = System.Math.Pow(x, n)

// Exercise 3
let dup s : string = s + s

// Exercise 4
let rec dupn (s:string) x =
        if x>=1 then (if x = 1 then s else s + dupn s (x-1)) else ""

// Exercise 5
let timediff (hh1, mm1)(hh2, mm2) = (hh2*60 + mm2)-(hh1*60 + mm1)

// Exercise 6
let minutes (hh, mm) = timediff(00, 00)(hh, mm)
```

H.3.2 HandIn 2

```
module Module2

// Exercise 7
let rec downTo x =
        if x < 1 then [] else (if x = 1 then [x] else x :: downTo (x - 1))

let rec downTo2 x =
    match x with
```

```
    | x when x < 1 -> []
    | 1 -> [1]
    | _ -> x :: downTo2 (x - 1)

// Exercise 8
let rec removeEven (x:int list) =
    match x with
    | [] -> []
    | [xs] -> [xs]
    | xs :: ys :: zs -> xs :: removeEven zs

// Exercise 9
let rec combinePair (x:int list) : (int * int) list =
    match x with
    | [] -> []
    | [xs] -> []
    | xs :: ys :: zs -> (xs, ys) :: combinePair zs

// Exercise 10
let explode (s:string) = List.ofArray (s.ToCharArray())

let rec explode2 (s:string) : char list =
    match s with
    | s when s.Length < 1  -> []
    | _ -> s.[0] :: explode2 (s.Substring 1)

// Exercise 11
let implode (cl:char list) : string =
                List.foldBack (fun elem acc -> string(elem) + string(acc) ) cl ""

let implodeRev (cl:char list) : string =
                List.fold (fun elem acc -> string(acc) + string(elem) ) "" cl

// Exercise 12
let toUpper (s:string) = implode (List.map System.Char.ToUpper (explode s))

let toUpper1 = explode >> List.map System.Char.ToUpper >> implode

let toUpper2 (s:string) = explode s |> (implode << List.map System.Char.ToUpper)

// Exercise 13
let palindrome (s:string) = (explode s |> implodeRev |> toUpper) = toUpper s

// Exercise 14
let rec ack (m, n) =
    match (m, n) with
    | (m, n) when m < 0 || n < 0 -> failwith "The Ackermann function
                                  is defined for non negative numbers only."
    | (m, n) when m = 0 -> n + 1
    | (m, n) when n = 0 -> ack (m - 1, 1)
    | (m, n) -> ack(m - 1, ack (m, n - 1))
```

```
// Exercise 15
let time f =
    let start = System.DateTime.Now in
    let res = f () in
    let finish = System.DateTime.Now in
    (res, finish - start)

let timeArg1 f a = time(fun () -> f(a))
```

H.3.3 HandIn 3

```
module FSharpHandIn3

type 'a BinTree =
    Leaf
    |   Node of 'a * 'a BinTree * 'a BinTree

let intBinTree =
    Node(
        43,
        Node(25, Node(56,Leaf, Leaf), Leaf),
        Node(562, Leaf, Node(78, Leaf, Leaf))
        )

// Exercise 16
let rec inOrder tree =
    match tree with
    Leaf -> []
    |   Node(n, treeL, treeR) -> inOrder treeL @ [n] @ inOrder treeR

// Exercise 17
let rec mapInOrder (f:'a -> 'b) (tree:'a BinTree) : 'b BinTree =
    match tree with
    Leaf -> Leaf
    |   Node(n, treeL, treeR) ->
            let left = mapInOrder f treeL
            let root = f(n)
            let right = mapInOrder f treeR
            Node(root, left, right)
(*Example:
The result tree should always be the same, as the function should
access all the elements no matter what.
The reason the individual nodes may not contain the same information
could be that the function depends on the order in which
the elements are accessed.*)

// Exercise 18
let rec foldInOrder f a t =
    match t with
    | Leaf -> a
```

```
    | Node(x, leftTree, rightTree) ->
    let left = foldInOrder f a leftTree in
    foldInOrder f (f x left) rightTree

// Exercise 19 & 21 & 22
type expr =
    | Const of int
    | If of expr * expr * expr
    | Bind of string * expr * expr
    | Var of string
    | Prim of string * expr * expr

let rec evalN expr (d:System.Collections.Generic.Dictionary<string, expr>) =
    match expr with
    | Const i -> i
    | Prim("-", expr1, expr2) ->
        evalN expr1 d - evalN expr2 d
    | Prim("+", expr1, expr2) ->
        evalN expr1 d + evalN expr2 d
    | Prim("max", expr1, expr2) ->
        List.max [evalN expr1 d; evalN expr2 d]
    | Prim("min", expr1, expr2) ->
        List.min [evalN expr1 d; evalN expr2 d]
    | Prim("=", expr1, expr2) ->
        if evalN expr1 d = evalN expr2 d then 1 else 0
    | If(expr1, expr2, expr3) ->
        if evalN expr1 d <> 0 then evalN expr2 d else evalN expr3 d
    | Bind(var, value, expr1) ->
        d.Add(var, value)
        evalN expr1 d
    | Var(name) when d.ContainsKey(name) ->
        evalN (d.[name]) d
    | Var(name) ->
        failwithf "Unknown variable '%s'" name
    | Prim(opr, _, _) ->
        (printfn "Operation %s not supported" opr; 0)

let eval expr =
    evalN expr (new System.Collections.Generic.Dictionary<string, expr>())

// Exercise 20
let testMinus =
    eval (Prim("-",Const(20),Const(30))) // Expected result = -10
let testPlus =
    eval (Prim("+",Const(20),Const(30))) // Expected result = 50
let testMax =
    eval (Prim("max",Const(20),Const(30))) // Expected result = 30
let testMin =
    eval (Prim("min",Const(20),Const(30))) // Expected result = 20
let testEqualFalse =
    eval (Prim("=",Const(20),Const(30))) // Expected result = 0
```

```
let testEqualTrue =
    eval (Prim("=",Const(20),Const(20))) // Expected result = 1

// Exercise 23
let testBindOne = // Expected result = 57
    eval (Bind("p", Prim("+", Const(13), Const(29)), Prim("+", Var("p"), Const(15))))
let testBindTwo = // Expected result = -16
    eval (Bind("x", Prim("-", Const(13), Const(29)), Prim("+", Var("x"), Const(15))))
let testBindThree = // Expected result = 97
    eval (Bind("x", Const(97), Bind("y", Const(3), Prim("max", Var("x"), Var("y")))))
let testBindFour = // Expected result = 0
    eval (Bind("x", Const(97), Bind("y", Const(3), Prim("=", Var("x"), Var("y")))))
let testBindFive =  // Fail case
    eval (Bind("x", Prim("+", Const(13), Const(29)), Prim("+", Var("y"), Const(15))))
```

H.3.4 HandIn 4 & 5

```
(* Define four functions that given a card with return one of the
four clothes on the card. *)
let findBot (card:card) = card.bot
let findTop (card:card) = card.top
let findLeft (card:card) = card.left
let findRight (card:card) = card.right

(* Define a function that as a string returns a pretty print of the
clothes. *)
let pp_clothes clothes =
  match clothes with
    | RED_JACKET      -> "RED_JACKET     "
    | RED_TROUSERS    -> "RED_TROUSERS   "
    | GREEN_JACKET    -> "GREEN_JACKET   "
    | GREEN_TROUSERS -> "GREEN_TROUSERS"
    | BLUE_JACKET     -> "BLUE_JACKET    "
    | BLUE_TROUSERS   -> "BLUE_TROUSERS  "
    | BROWN_JACKET    -> "BROWN_JACKET   "
    | BROWN_TROUSERS -> "BROWN_TROUSERS"

(* Split list [x1,...,xN] in the lists [x1,...,xn-1] and [xn,...,xN] *)
(* where n >= 0 and n < N .                                          *)
(* Fx: splitNth (0,[1;2]) gives ([], [1; 2])                         *)
(*     splitNth (1,[1;2]) gives ([1], [2])                           *)
(*     splitNth (2,[1;2]) gives ([1; 2], [])                         *)
(*     splitNth (3,[1;2]) should die : 3 outside range of list       *)
(*     splitNth (-1,[1;2]) should die : -1 outside range of list     *)
let rec splitNth (n, xs) =
    match (n, xs) with
    | (0, xs) -> ([], xs)
    | (n, x::xs) -> let (a, b) = splitNth (n - 1, xs);
                    (x :: a, b)
    | _ -> failwith "%s outside range of list" n
```

```
(* IT IS REQUIRED THAT THE OUTPUT MATCHES THE EXAMPLES
BELOW EXACTLY!!!    *)
(* Please use the helper functions above: pp_horizontal,
pp_vertical etc. *)
(* PrettyPrint all cards - colNo says number of columns on the board. *)
(* colNo is defined to be 4 above.                                    *)
(* A few examples of output below.                                    *)
let rec pp_cards (cards:card list) =
    match cards with
    | [] -> ()
    | cards when cards.Length <= 4 -> pp_row cards
    | _ -> let (a, b) = splitNth (4, cards);
           pp_row a; pp_cards b

(* matchClothes: the valid combinations of clothe.  *)
(* There are 8 valid combinations                   *)
let matchClothes clothe1 clothe2 =
  match (clothe1,clothe2) with
    | (RED_JACKET, RED_TROUSERS) -> true
    | (RED_TROUSERS, RED_JACKET) -> true
    | (GREEN_JACKET, GREEN_TROUSERS) -> true
    | (GREEN_TROUSERS, GREEN_JACKET) -> true
    | (BLUE_JACKET, BLUE_TROUSERS) -> true
    | (BLUE_TROUSERS, BLUE_JACKET) -> true
    | (BROWN_JACKET, BROWN_TROUSERS) -> true
    | (BROWN_TROUSERS, BROWN_JACKET) -> true
    | _ -> false

(* matchBot: Given a coordinate, match that card with the card
immediately below. *)
(* Notice, cards at the bottom row fulfils this automatically.
      *)
let matchBot (row, col, cards) card =
    if row > 0 then
        let botCard = List.nth cards ((findIndexInList (row, col))+colNo)
        matchClothes (findTop botCard) (findBot card)
    else true

(* matchRight: Given a coordinate, match that card with the card immediately
to the right. *)
(* Notice, cards at the rightmost column fulfils this automatically.
    *)
let matchRight (row, col, cards) card =
  if col > 0 then
    let rightCard = List.nth cards ((findIndexInList(row, col))+1)
    matchClothes (findLeft rightCard) (findRight card)
   else true

(* There is ONE error in the code below - and it never terminates *)
(* If you correct this one error - everything will work just fine *)
(* You must explain the error as a comment here                   *)
```

```
(* Jakob Melnyk comment: The add function was called with 0 as the
   n parameter. This meant the board never changed index in the list
   was accessed when cards were matched - meaning it could keep
   finding the same filled board over and over again or possibly
   never fill out the board.
        *)
let rec findSol (rest: 'a list) alreadyTried ((row,col,cards) as board) sols =
  match (rest,alreadyTried) with
    | ([],[]) -> board::sols (* No rest and alreadyTried is empty, that is,
 solution found *)
    | ([],_) -> sols         (* No solution if alreadyTried is non empty. *)
    | (x::rest, alreadyTried) ->
      let sols' =
        if Match board x then
          (* If there is a match, then go on with the rest of the cards *)
          let (row', col') = add 1 (row, col)
          findSol(rest@alreadyTried) [] (row', col', cards@[x]) sols
        else sols (* If no match then no new solutions found. *)
      (* Put the card x in alreadyTried and move on. *)
      findSol rest (x::alreadyTried) board sols'
```

H.4 F# Handins - Niklas Hansen

H.4.1 HandIn 1

```
// Author: Niklas Hansen <nikl@itu.dk>
module Handin1

// Exercise 1
let sqr (x:int) =
    x * x

// Exercise 2
let pow (x:float) (y:float) =
    x ** y

// Exercise 3
let dup (s:string) =
    s + s

// Exercise 4 - v1
let rec dupn (s:string) (n:int) =
    match n with
    | 0 -> ""
    | _ -> s + dupn s (n-1)

// Exercise 4 - v2
//let rec dupn (s:string) = function
//    | 0 -> ""
```

```
//     | n -> s + dupn s (n-1)

// Exercise 5
let timediff (h1:int, m1:int) (h2:int, m2:int) =
    ((h2 * 60) + m2) - ((h1 * 60) + m1)

// Exercise 6
let minutes (hh:int, mm:int) =
    timediff (00, 00) (hh, mm)


printfn "1. Sqr 3: %i" (sqr 3)
printfn "2. pow 3 2: %f" (pow 3.0 2.0)
printfn "3: dup \"Hi \": %s" (dup "Hi ")
printfn "4. dupn \"Hi \" 3: %s" (dupn "Hi " 3)

printfn "5a. timediff (12, 34) (11, 35): %i" (timediff (12, 34) (11, 35))
printfn "5b. timediff (12, 34) (13, 35): %i" (timediff (12, 34) (13, 35))

printfn "6a. minutes (14, 24): %i" (minutes (14, 24))
printfn "6b. minutes (23, 1):  %i" (minutes (23, 1))
```

H.4.2 HandIn 2

```
// Author: Niklas Hansen <nikl@itu.dk>
module Handin2

// Exercise 7a
let rec downTo (n:int) =
    if n > 0
        then n :: downTo (n-1)
        else []

// Exercise 7b
let rec downTo2 (n:int) =
    match n with
    | n when n <= 0 -> []
    | _ -> n :: downTo2 (n-1)

// Exercise 7b v2
//let rec downTo2 = function
//     | n when n <= 0 -> []
//     | n -> n :: downTo2 (n-1)

// Exercise 8
let rec removeEven = function
    | [] -> []
    | [n] -> [n]
    | n :: m :: tl -> n :: removeEven tl

// Exercise 9
```

```
let rec combinePair = function
    | []-> []
    | [n] -> []
    | n :: m :: tl -> (n, m) :: combinePair tl

// Exercise 10a
let explode (s:string) =
    let chars = s.ToCharArray()
    List.ofArray(chars)

// Exercise 10b
let rec explode2 (s:string) =
    match s with
    | "" -> []
    | _ -> s.Chars 0 :: explode2 (s.Remove(0, 1))

// Exercise 11a
let implode (c:char list) =
    List.foldBack (fun x y -> sprintf "%c%s" x y) c ""

// Exercise 11b
let implodeRev (c:char list) =
    List.fold (fun x y -> sprintf "%c%s" y x) "" c

// Exercise 12a
let toUpper (s:string) =
    implode (List.map (fun x -> System.Char.ToUpper(x)) (explode s))

// Exercise 12b
let toUpper1 (s:string) =
    (explode >> List.map (fun x -> System.Char.ToUpper(x)) >> implode) s

// Exercise 12c
let toUpper2 (s:string) =
    s |> (implode << List.map (fun x -> System.Char.ToUpper(x)) << explode)

// Exercise 13
let palindrome (s:string) =
    let org = s.ToLower().Replace(" ", "")
    let rev = new string (Array.rev (org.ToCharArray()))
    org = rev

// Exercise 14
let rec ack (m:int, n:int) =
    match m, n with
    | (0, n) -> n + 1
    | (m, 0) when m > 0 -> ack(m - 1, 1)
    | (m, n) when m > 0 && n > 0 -> ack(m - 1, ack(m, n - 1))
    | (m, n) -> failwith "Invalid input!"

// Addon for Exercise 15
```

```
let time f =
    let start = System.DateTime.Now
    let res = f ()
    let finish = System.DateTime.Now
    (res, finish - start)

// Exercise 15
let timeArg1 f a =
    time (fun () -> f a)


printfn "7a. downTo 5: %s" ((downTo 5).ToString())
printfn "7a. downTo -3: %s" ((downTo -3).ToString())
printfn "7b. downTo2 5: %s" ((downTo2 5).ToString())

printfn "8. removeEven [1; 2; 3; 4; 5]: %s" ((removeEven [1; 2; 3; 4; 5]).ToString())
printfn "8. removeEven []: %s" ((removeEven []).ToString())
printfn "8. removeEven [1]: %s" ((removeEven [1]).ToString())

printfn "9. combinePair [1; 2; 3; 4]: %s" ((combinePair [1; 2; 3; 4]).ToString())
printfn "9. combinePair [1; 2; 3]: %s" ((combinePair [1; 2; 3]).ToString())
printfn "9. combinePair [1; 2]: %s" ((combinePair [1; 2]).ToString())
printfn "9. combinePair []: %s" ((combinePair []).ToString())
printfn "9. combinePair [1]: %s" ((combinePair [1]).ToString())

printfn "10a. explode \"star\": %s" ((explode "star").ToString())
printfn "10b. explode2 \"star\": %s" ((explode2 "star").ToString())

printfn "11a. implode ['a';'b';'c']: %s" (implode ['a';'b';'c'])
printfn "11b. implodeRev ['a';'b';'c']: %s" (implodeRev ['a';'b';'c'])

printfn "12a. toUpper \"Hej\": %s " (toUpper "Hej")
printfn "12b. toUpper1 \"Hej\": %s " (toUpper1 "Hej")
printfn "12c. toUpper2 \"Hej\": %s " (toUpper2 "Hej")

printfn "13. palindrome \"Anna\": %s" ((palindrome "Anna").ToString())
printfn "13. palindrome \"Ann\": %s" ((palindrome "Ann").ToString())

printfn "14. ack(3, 11): %i" (ack(3, 11))

printfn "Extra. time: %s" ((time (fun () -> ack (3,11))).ToString())
printfn "15. timeArg1 ack (3, 11): %s" ((timeArg1 ack (3, 11)).ToString())

System.Console.ReadKey(true)
```

H.4.3 HandIn 3

H.4.4 HandIn 4 & 5

H.5 F# Handins - Ulrik Flænø Damm

H.5.1 HandIn 1

```
KF 02
Handin 1
Ulrik Damm (ulfd@itu.dk)

let sqr x = x * x;;

let pow x n = System.Math.Pow (x, n);;

let dup s = s + s;;

let rec dupn s n =
        match n with
        | 0 -> ""
        | 1 -> s
        | n -> s + dupn s (n-1)
        ;;

let timediff (time11, time12) (time21, time22) = (time21 - time11) * 60 +
(time22 - time12);

let minutes (time1, time2) = timediff (0, 0) (time1, time2);

let Main =
        printfn "%i" (sqr 2);
        printfn "%f" (pow 2.0 3.0);
        printfn "%s" (dup "Hi ");
        printfn "%s" (dupn "Hi " 3);
        printfn "%i" (timediff (12,34) (11,35));
        printfn "%i" (timediff (12,34) (13,35));
        printfn "%i" (minutes (14,24));
        printfn "%i" (minutes (23,1));
```

H.5.2 HandIn 2

```
let rec downTo n =
    if n < 2
    then raise (new System.Exception("Invalid value"))
    else if n = 1
    then [1]
    else n :: downTo (n-1);;

let rec downTo2 = function
    | n when n < 1 -> raise (new System.Exception("Invalid value"))
    | 1 -> [1]
    | n -> n :: downTo2 (n-1);;

let rec removeEven = function
```

```
    | [] -> []
    | [n] -> [n]
    | [n; _] -> [n]
    | n :: m :: tl -> removeEven [n; m] @ removeEven tl;;

let rec combinePair = function
    | [] -> []
    | [n] -> []
    | [n; m] -> [n, m]
    | n :: m :: tl -> combinePair [n; m] @ combinePair tl;;

let explode (str : string) = List.ofArray(str.ToCharArray());;

let rec explode2 (str : string) =
        if str.Length = 0 then []
        else if str.Length = 1 then [str.Chars(0)]
        else str.Chars(0) :: explode2 (str.Remove(0, 1));;

let implode str = List.foldBack (fun x y -> sprintf "%c%s" x y) str "";;

let implodeRev str = List.fold (fun x y -> sprintf "%c%s" y x) "" str;;

let toUpper str =
     implode (List.map (fun x -> System.Char.ToUpper x) (explode str));;

let toUpper1 str =
     (explode >> (List.map (fun x -> System.Char.ToUpper x)) >> implode) str;;

let toUpper2 str =
     (implode << ((fun x -> System.Char.ToUpper x) |> List.map) << explode) str;;

let rec palindrome (str : string) =
    if str.Length <= 1 then true
    else if str.Chars(0) = str.Chars(str.Length - 1)
         then palindrome (str.Substring(1, str.Length - 2))
    else false;;

let rec ack = function
    | (0, n) -> n + 1
    | (m, 0) -> ack (m - 1, 1)
    | (m, n) -> ack (m - 1, ack (m, n - 1));;

let time f =
    let start = System.DateTime.Now in
    let res = f () in
    let finish = System.DateTime.Now in
    (res, finish - start);

let timeArg1 f a = time (fun x -> f a);;

let Main = printfn "lol";;
```

H.5.3 HandIn 3

H.5.4 HandIn 4 & 5