

Final Report Hyro

Eimer Ahlstedt, Emil Lindblad, Sebastian Kvaldén, Timothy Nilsson, Erik Larsson

October 24, 2021

Requirements and Analysis Document for Hyro

TDA367, Chalmers tekniska högskola

Eimer Ahlstedt, Emil Lindblad, Sebastian Kvaldén, Timothy Nilsson, Erik Larsson

24 oktober 2021

Table of contents

1	Introduction	1
1.1	Definitions, acronyms and abbreviations	1
2	Requirements	1
2.1	User Stories	1
2.1.1	Runnable Program	1
2.1.2	Data Storage	2
2.1.3	Log-In Screen and Functionality	2
2.1.4	Views and Navigation Buttons	2
2.1.5	User Settings	3
2.1.6	Grid of Listings and Detail View	3
2.1.7	Add listings page and functionality	3
2.1.8	Create Account Page	4
2.1.9	Rent-Button	4
2.1.10	Browse View for Listings	4
2.1.11	Sort and search	4
2.1.12	Log-Out Button	5
2.1.13	Edit Listings	5
2.1.14	Images	5
2.1.15	Remove Listings	5
2.1.16	Remove Completed Bookings	6
2.1.17	Book in specific time frames	6
2.1.18	Decline bookings	6
2.1.19	Cancel booking requests	6
2.2	Definition of Done	7
2.3	User interface	7
3	Domain Model	10
3.1	Class responsibilities	10

1 Introduction

The purpose of this application is to enable people to participate in sharing economy, by allowing users to put their own items up for rent as well as rent other users' items. This is achieved by creating a virtual marketplace, where users can create an account, browse listings and also add their own items for rent.

People who can benefit from this application can be divided into two categories: the ones who rent items and the ones who supply items for rent.

Imagine that you have just bought some new shelves and realizing that you do not have a hammer drill for mounting them to your concrete wall. Instead of spending a lot of on a new one or leasing one with improper and complicated agreements, you can just check if anyone nearby has a hammer drill for rent on Hyro.

In the other case, you might have a tool or item that you only use a few times a year, like a pressure washer or a trailer. Instead of letting these items collect dust in your garage, you can rent them out on Hyro. Making them available for more people, decreasing their environmental footprint.

1.1 Definitions, acronyms and abbreviations

- **Jira** - A tool used for Agile software management.
- **JSON** - JavaScript Object Notation. A human readable data format usually used for data interchange with web servers. We use it in our application for storing and persisting data between sessions.
- **JavaFX** - A Java framework used for creating graphical users interfaces.
- **GitHub** - A tool used for storing code repositories.
- **FXML file** - A type of file used by JavaFX. Contains information on visual elements and what code is connected to each element.
- **MVC design pattern** - A design pattern of object oriented programming. The goal of the pattern is to divide a code base into three parts, one Model, one View and one Controller, each one with different responsibilities.

2 Requirements

2.1 User Stories

The following user stories is implemented or should be implemented in the future.

2.1.1 Runnable Program

Status: DONE

Description

As a user I want to be able to run the program, so that I can start using it.

Confirmation

Functional:

Program compiles and basic window opens.

Non-Functional:

Project created and accessible through GitHub.

Configure maven and JavaFX.

2.1.2 Data Storage

Status: DONE

Description

As a User I want to be able to see the same information when I restart the application so that I can continue renting later.

Confirmation

Functional:

A User should be able to see the same listings after restarting the application.

Non-Functional:

JSON-files created and tested for User and Listing-classes.

Fully functional read/write-methods to access the JSON-files

2.1.3 Log-In Screen and Functionality

Status: DONE

Description

As a User, I want to be able to enter my login credentials, so that I can be logged in.

Confirmation

Functional:

Accessible Log-In page for users to enter their credentials or create account.

Non-Functional:

2.1.4 Views and Navigation Buttons

Status: DONE

Description

As a User I want to be able to navigate through the different pages, so that I can explore the application.

Confirmation

Functional:

Access different views as a user. Such as Log-In and Settings screen.

Non-Functional:

2.1.5 User Settings

Status: DONE

Description

As a User I want to be able to change my personal information (address, password etc).

Confirmation

Functional:

View that shows and enables changes to the logged in user's information.

Non-Functional:

Check for correct syntax in user input and logged in user.

2.1.6 Grid of Listings and Detail View

Status: DONE

Description

As a User I want to be able to view a product, so that I can determine if I want to rent it.

Confirmation

Functional:

A user should be able to view a pre-defined product in the application.

2.1.7 Add listings page and functionality

Status: DONE

Description

As a User I want to be able to add a product for rent. So that I can show other users that I have products for rent.

Confirmation

Functional:

Logged-in user can create a product and let other users rent it.

2.1.8 Create Account Page

Status: DONE

Description

As a User, I want to be able to create an account so that I can keep track of my data and log in.

Confirmation

Functional:

Create Account-button on log-in page for new users to create account.

2.1.9 Rent-Button

Status: DONE

Description

As a User I want to be able to request to rent a product, so that can I show the owner that I am interested.

Confirmation

Functional:

A logged-in user can request to rent a product that another user has advertised as available.

2.1.10 Browse View for Listings

Status: DONE

Description

As a User, I want to be able to see a list of all products that are available for renting, so that I can browse and decide what I want to rent.

Confirmation

Functional:

A main page where a user can browse products.

2.1.11 Sort and search

Status: DONE

Description

As a User I want to be able to sort and search for products that are available.

Confirmation

Functional:

Search-function is usable and shows relevant products. Category buttons changes shown products to match the specific category.

2.1.12 Log-Out Button

Status: DONE

Description

As a User I want to be able to log-out of the application when i am done using it.

Confirmation

Functional:

Functioning log-out button that takes a user back to log-in screen.

2.1.13 Edit Listings

Status: DONE

Description

As a User I want to be able to edit my already existing listings so that i can supply the correct and latest information.

Confirmation

Functional:

Functional page to see the users current listings and be able to change their information.

2.1.14 Images

Status: DONE

Description

As a User I want to be able to upload a picture of my listing, so other users can get a better understanding of my offer.

Confirmation

Functional:

Pictures are shown with listing objects.

2.1.15 Remove Listings

Status: DONE

Description

As a User I want to be able to remove my listings so they cant be rented anymore.

Confirmation

Functional:

Remove Objectbutton at detail view for my listing. Listings that are already out for rent can ´t be removed, however available objects are remove able.

2.1.16 Remove Completed Bookings

Status: DONE

Description

As a User I want to be able to remove my completed bookings, so that they don't show anymore.

Confirmation

Functional:

Remove-button at bookings i have previously made to remove them from my list.

2.1.17 Book in specific time frames

Status: NOT IMPLEMENTED

Description

As a User I want to be able to rent a product during a specific date range, so that i can specify when i want to rent an item.

Confirmation

Functional:

User has the ability to book a product for a future date.

2.1.18 Decline bookings

Status: NOT IMPLEMENTED

Description

As a User I want to be able to decline a booking request, so that I don't have to rent my product.

Confirmation

Functional:

Decline button for the owner of the product to use if they don't want to rent it out.

2.1.19 Cancel booking requests

Status: NOT IMPLEMENTED

Description

As a User I want to be able to cancel my booking request, so that i don't have to follow through with a booking i don't want anymore.

Confirmation

Functional:

The user has the ability to cancel a booking request that they made before it is accepted by

the product owner.

2.2 Definition of Done

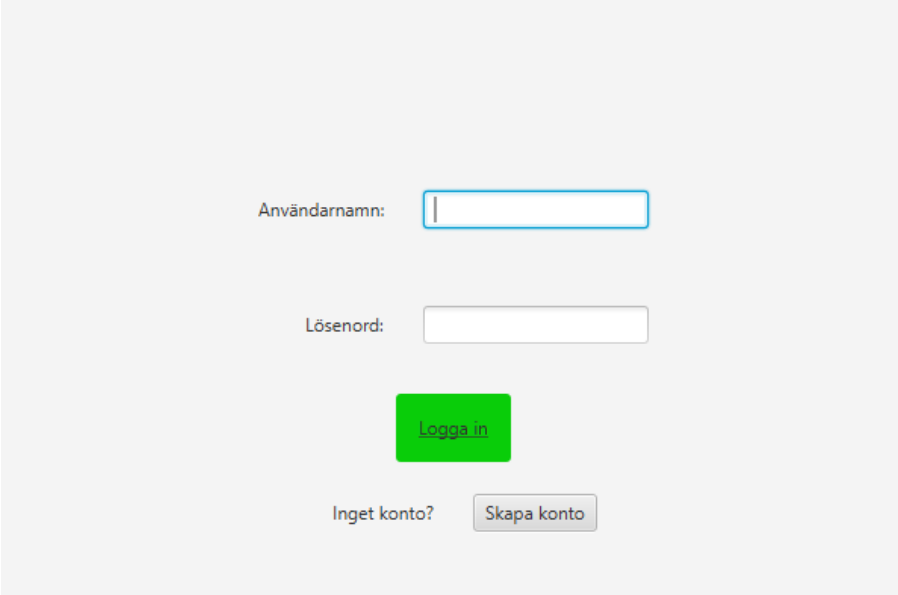
New features are worked on in a new branch

Public parts of code are documented.

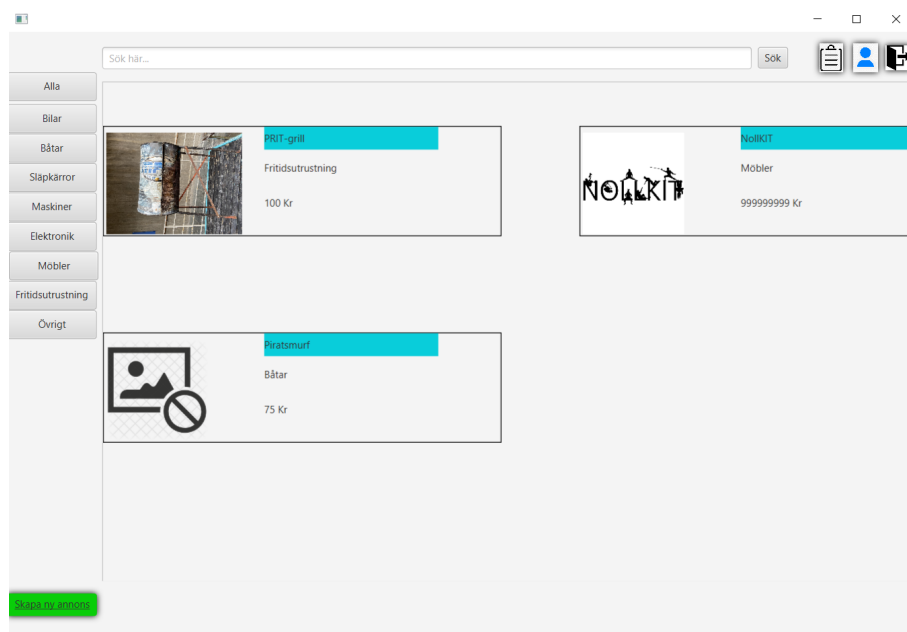
90% Test Coverage of relevant code in Model classes. Simple getters and setters and states are considered irrelevant for tests. All checks pass and merge to master is successful

2.3 User interface

The User Interface is created using JavaFX and consists of multiple FXML files. Each view that the user can navigate to is a Scene which is where all the data gets passed through, both input and the output. Each Scene has its own controller that is in charge of changing the scene that user wants to see. So if for example a users presses Log in button and all log in checks passes, the controller will request the SceneHandler to then switch the scene the browse view. The Scene Handler then loads the respective FXML file and then displays it in the application.

A screenshot of a login view with a light gray background. It features two text input fields: the first is labeled 'Användarnamn:' and the second is labeled 'Lösenord:'. Below the password field is a green button with the text 'Logga in'. At the bottom, there is a link 'Inget konto?' followed by a button labeled 'Skapa konto'.

Figur 1: Login in view where users can log in to an existing account or create a new one



Figur 2: Browse view where logged in users can see available listings for rent and also create a new listing

Produktnamn:

Beskrivning:

Kategori:
Välj kategori

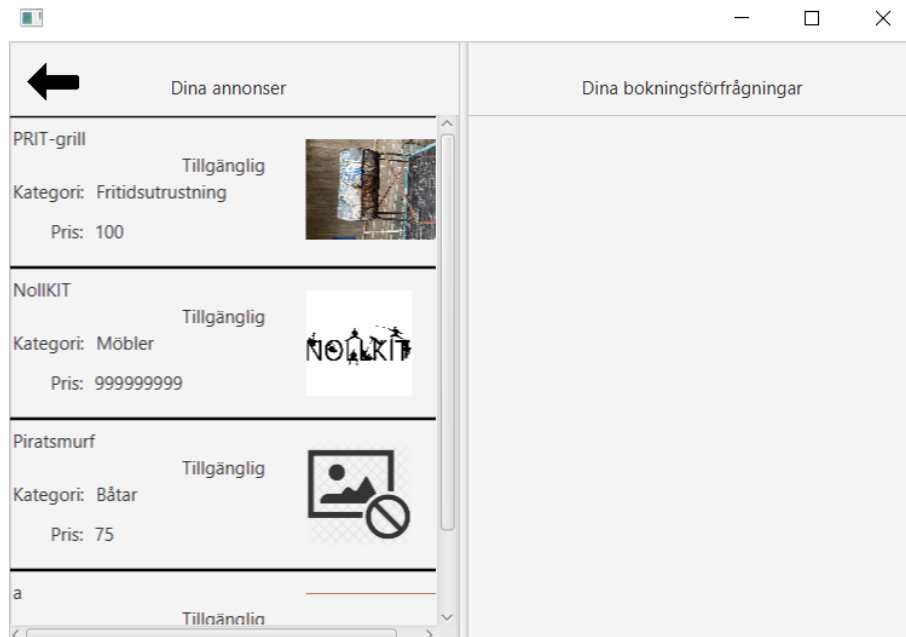
Pris:

Availability

☐

Skapa annons

Figur 3: The standardized view for listing settings, used both to create and edit listings.



Figur 4: View for a users own listings and active booking requests. Also used for accepting booking requests and making payments.

The screenshot shows the "Användarinställningar" (User settings) form. It contains the following fields:

- Förnamn:** e
- Efternamn:** e
- Gatunamn:** e
- Postadress / Stad:** 12345 e
- Land:** e
- Användarnamn:** e
- Lösenord:** •
- Bekräfta Lösenord:** Lösenord
- Telefonnummer:** 1
- Bankkontonummer:** 1

An "Ändra" (Change) button is located at the bottom right of the form.

Figur 5: The standardized view for account settings, used both to create and edit accounts.

3 Domain Model

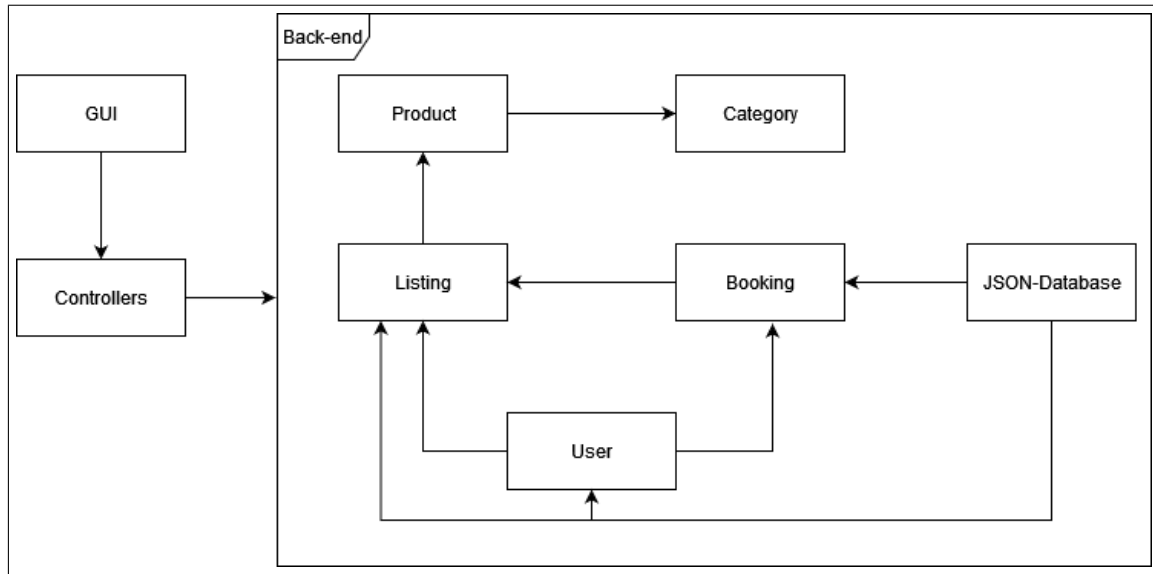


Figure 6: The current Domain Model of Hyro

3.1 Class responsibilities

Above is the current state of the Hyro Domain Model.

The program is built around the MVC design pattern. The class "GUT" contains all FXML files and their controller classes, making up the View part. FXML controller class is not to be confused with MVC Controller; FXML controller classes are simply the Java classes connected to FXML files.

"Controllers" represents the MVC Controller classes. View classes chain calls to Controllers, who in turn update the View with information from the Back-end, or Model.

The Back-end, or Model, contain the logic and data needed in the program.

The User class is for keeping track of an individual users contact information, currently listed products and active booking requests.

Listing exists to contain information on what product has been listed, and by which user. It also keeps track of the current state of the listing such as if the listing is available or has a booking request etc etc.

The purpose of Booking is to be the middle ground between a customer wanting to rent a product and the owner of the product. It keeps track of the state of the booking and controls the process of booking, paying and returning.

Product and Category are simple. A product is the item available for rent, and the Product class contains information on the item, such as its category. The Category is used to filter searches, for example if a user wants to rent any grill, it could browse a hypothetical "grill"-category.

References

NA

System Design Document for Hyro

TDA367, Chalmers tekniska högskola

Eimer Ahlstedt, Emil Lindblad, Sebastian Kvaldén, Timothy Nilsson, Erik Larsson

24 oktober 2021

Table of contents

1	Introduction	1
1.1	Definitions, acronyms and abbreviations	1
2	System Architecture	1
2.1	Model	1
2.2	View	1
2.3	Controller	1
2.4	Databases	1
2.5	Application flow	2
3	System Design	5
3.1	Model View Controller	5
3.2	Relation between Design Model and Domain model	6
3.3	Design Patterns	6
4	Persistent Data Management	7
4.1	User Data	7
4.2	Images	7
5	Quality	7
5.1	Tests and Issues	7
5.2	Analytics	7
5.2.1	Project Dependencies	7
5.2.2	Code Dependencies	8
5.2.3	Quality - PMD	8
5.3	Access Control and Security	8
A	Appendix	10

1 Introduction

The purpose of this document is to give insight into the technical side of the application Hyro. Descriptions of testing methodology, software architecture, system design and more will be given in the following chapters.

Hyro, the application in question, is a simple platform enabling its users to participate in a sharing economy, and thus save both time and money. Using Hyro, users can both list items they own as available for rent and browse the listings of other users. The platform is meant to offer a beginning to endsolution, including adding items for rent, browsing, booking and paying.

Through Hyro users will be able to save money by renting items at a lower price than from big companies, minimize their environmental impact by maximising the potential of already produced items and let their personal belongings generate passive income.

1.1 Definitions, acronyms and abbreviations

- **Jira** - A tool used for Agile software management.
- **JSON** - JavaScript Object Notation. A human readable data format usually used for data interchange with web servers. We use it in our application for storing and persisting data between sessions.
- **JavaFX** - A Java framework used for creating graphical users interfaces.
- **GitHub** - A tool used for storing code repositories.
- **FXML file** - A type of file used by JavaFX. Contains information on visual elements and what code is connected to each element.
- **MVC design pattern** - A design pattern of object oriented programming. The goal of the pattern is to divide a code base into three parts, one Model, one View and one Controller, each one with different responsibilities.

2 System Architecture

The application is designed around the MVC design pattern, which means that the structure can be broken down into three different components: Model, View and Controller. All these components, each with their own responsibility work together to

2.1 Model

The Model component is the central part of the application. It contains and manages all the data in the application. It receives instructions from the controller on what to do. The model is then split up in internal packages such as User, Listing and Booking packages.

2.2 View

Anything that the end user can see and interact with is displayed and created by the View component. The specific views are sub classes to an abstract class to ensure they all behave the same and are exchangeable throughout the program.

2.3 Controller

Connection between the View and the Model is handled by the Controller component. The controller takes the user input from the View and decides what to get from the model and updates the view with the new data.

2.4 Databases

The application uses a basic form of databases to store data while the application is not in use. This data is then loaded into the model when the application is started and saved back to the files

when exited instead of continuously loading data from the database. The databases are connected to the data-handlers in the model via a *Read/Write*-classes.

2.5 Application flow

A good way to describe how all these components work together is with a sequence diagram. In the example below (1), we will look at the flow though the application when creating a new listing.

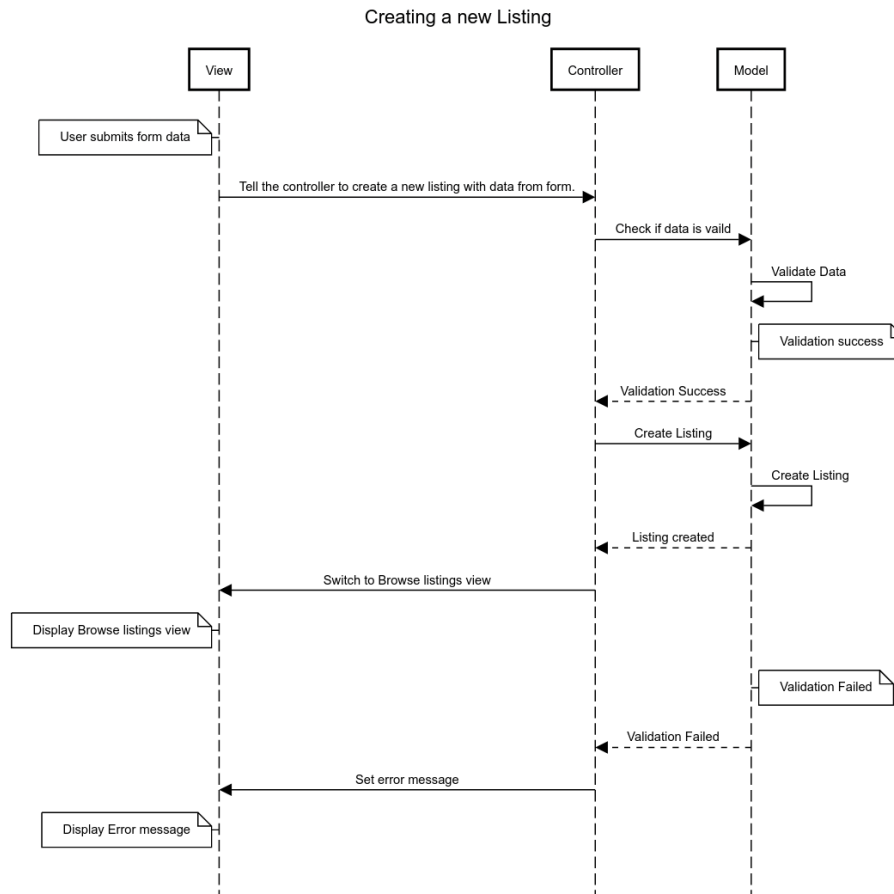


Figure 1: A sequence diagram for creating a new listing

A similar example of the flow though the application is when a users attempts to login:

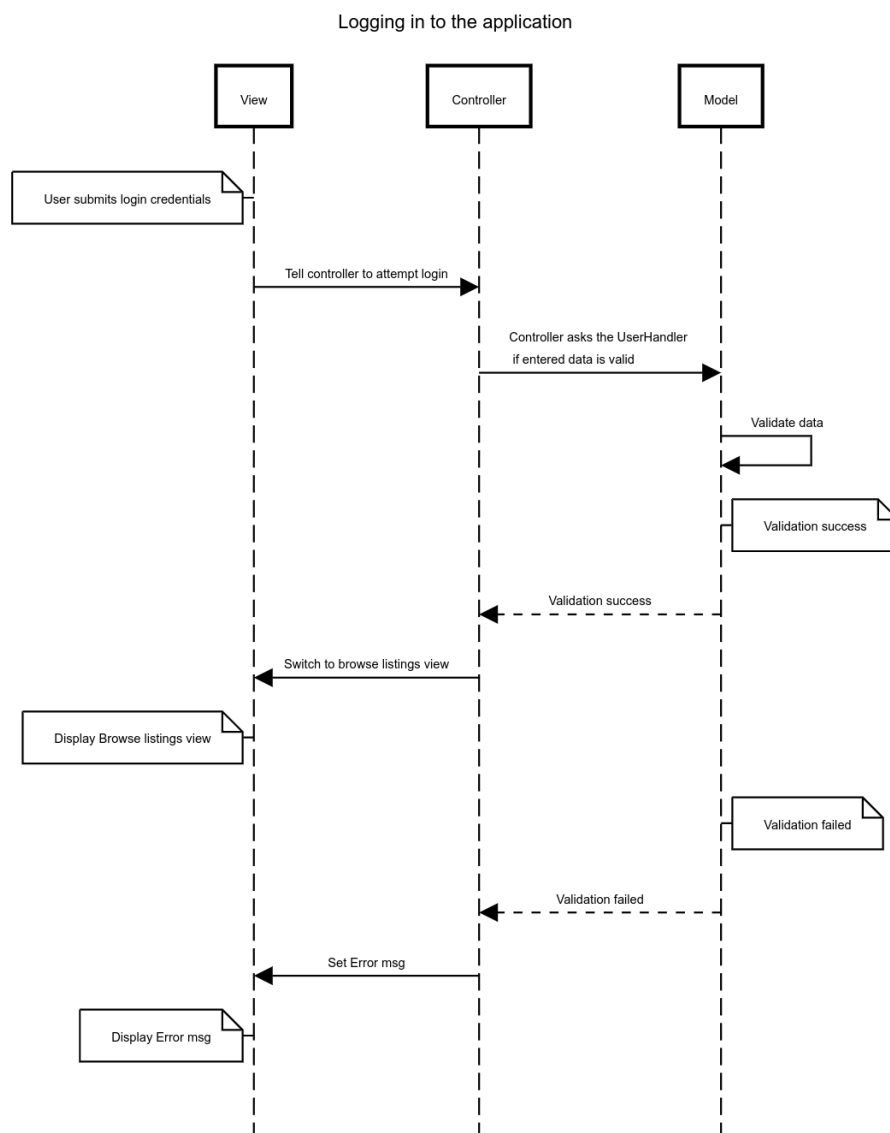


Figure 2: A sequence diagram for logging in to the application

This final example shows the flow for creating a booking of a listing

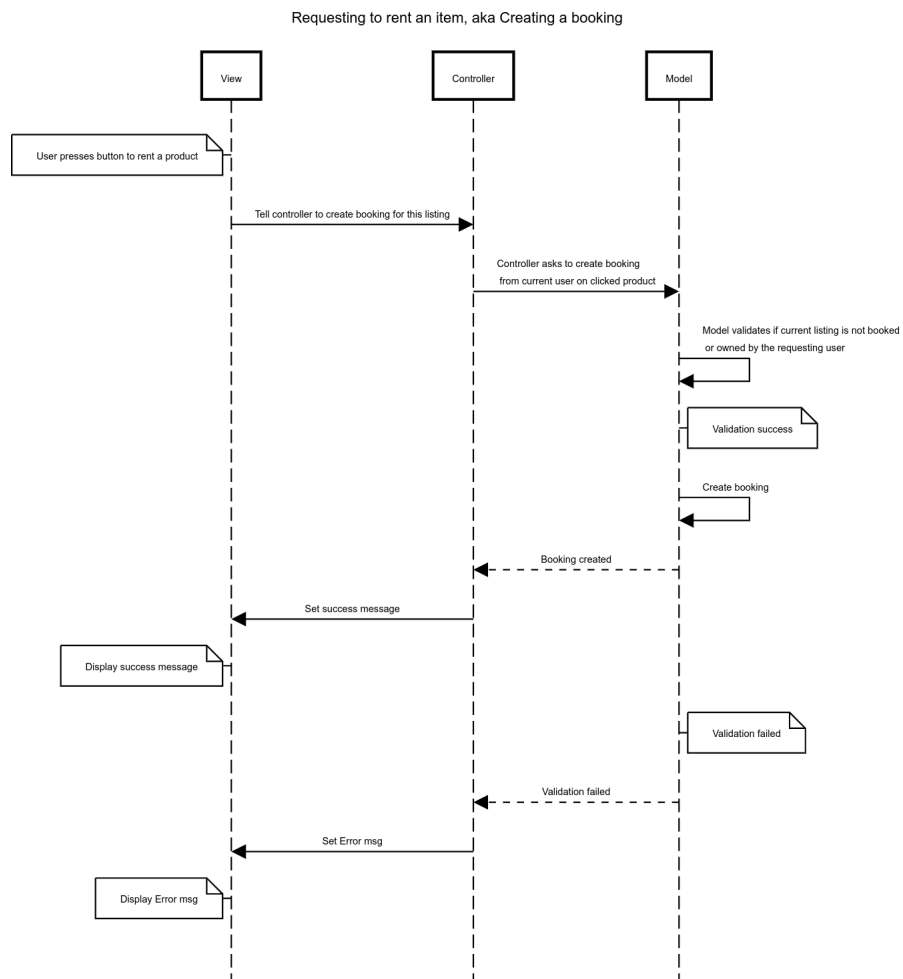
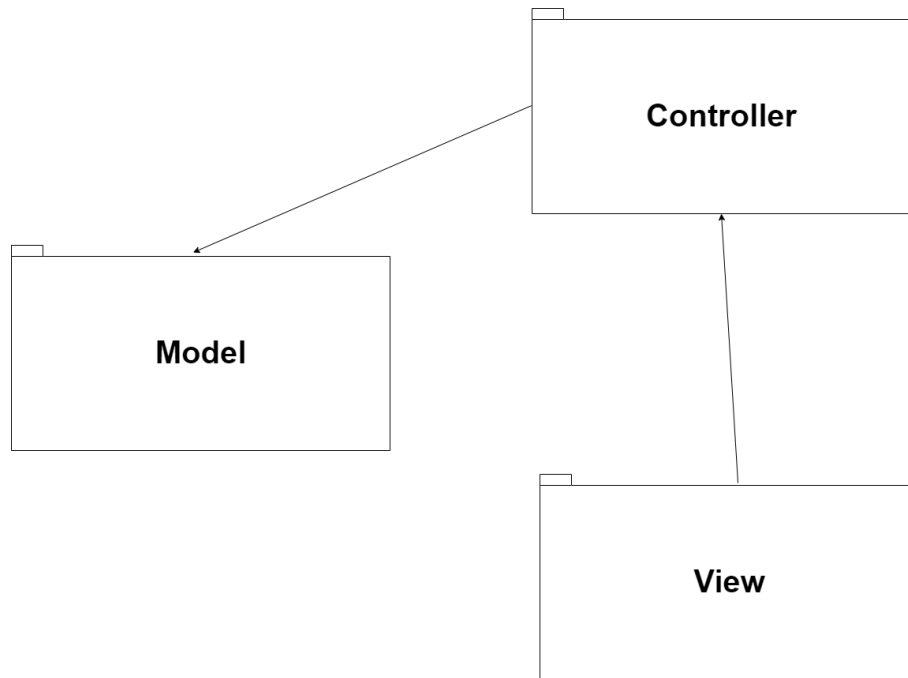


Figure 3: A sequence diagram for creating a booking

3 System Design



Figur 4: Package Diagram MVC

3.1 Model View Controller

MVC is implemented with the mindset that view should be "dumb", controller should be "thin" and model should be "smart". The model is also completely independent from the others and controllers and view could be replaced without having to change the model.

argument returning the state. This largely defeats the purpose of the State pattern, however should one be able to properly serialize an interface field using GSON, the process of refactoring back into a proper State pattern is very easy.

At one point of time in development Booking was a listener to Listing through the Observer pattern. This caused issues when saving to the JSON file however. Keeping the Observer pattern would have been a cleaner solution, but due to time constraints and JSON being hard to work with, the pattern could not make it into the final version.

Singleton pattern is used in various parts of the application in classes where we only want one instance to be created. This is because the various handlers is handling database entries in JSON. If multiple instances of handlers were to be created it would be nearly impossible to display all entries that should exist in run-time.

4 Persistent Data Management

4.1 User Data

The application uses a database to store information about users and their products. This functionality is the responsibility of the JSON package which consists of the classes JSONWriter and JSONReader. They both use Gson [1], a serialization/deserialization library, to convert Java Objects to and from the JSON file format. The .json files are stored in the resource folder JSONFiles and are written and read by the java.io Writer and Reader respectively.

4.2 Images

All images are stored in a folder within the application where they can be directly accessed and loaded into JavaFX. All images are handled using a utility class called ImageHandler.

ImageHandler is a singleton, that is in charge of loading and fetching images. Whenever a part of the application needs to display an image, the ImageHandler first checks if the image has previously been loaded. If it has it then simply fetches the image from the "cache" and displays it with JavaFX. If the images has not previously been loaded it loads the images by matching the name of the file with correspondent one in the Images folder. After it has been loaded it also places the image in the HashMap to avoid loading the same images multiple times during runtime.

5 Quality

5.1 Tests and Issues

We are using JUnit to write unit tests for our model classes. The tests are located in /src/test/java.

We use Travis CI for continuous integration. <https://app.travis-ci.com/github/emillindblad/17E-00P-Project>

5.2 Analytics

5.2.1 Project Dependencies

Compile			
GroupId	ArtifactId	Version	Type
com.google.code.gson	gson	2.8.8	jar
org.openjfx	javafx-controls	16	jar
org.openjfx	javafx-fxml	16	jar
Test			
junit	junit	4.11	jar

5.2.2 Code Dependencies

Code - JDepend		
Class/Package	Used by	Uses
Model.UserPackage	Model.Booking, Model.ListingLinker, Controllers, View	None
Model.Listing	Controllers, Model.Booking, View	ListingLinker
Model.Booking	Controllers	Model.Listing, Model.UserPackage
Model.InputChecker	Controllers	Model.Listing
Model.ListingLinker	Model.Booking, View	Model.UserPackage

5.2.3 Quality - PMD

No major discrepancies between our code and best practices were found. Minor design flaws were flagged that were either insignificant or simply incorrect assessments by PMD. Documentation could be improved in Controllers and View.

Many JUnit assertions could be improved by including a message and simplified by changing assertion type.

5.3 Access Control and Security

To ensure that no one except the logged-in user can access their information everything passes through a *Handler*-class in the *User Package* where no information is accessible unless a successful login has been made. This is implemented so that the program only shows the logged in users information and all other information that are connected to users are handled internally in the user package except for user ID-markers that enables us to connect a listing to a user by only showing the client code a string and not the entire User object.

Furthermore the *handler* is structured in a way that there are no public methods to access the complete list of users and the only information that will be visible from the client side is that of the logged-in user and internal identification markers the client will use when sending calls to the *handler* when an interaction with another user is made.

The *getLoggedInUser*-method that enables access to the users data are in the current implementation only called by the client code when the program is in a state where a user has left the log-in screen and are therefor already logged-in. If the usage of this method would be used somewhere else where it is not intended to (when no user is logged in) it would create a "NullPointerException"-exception and close the application. In the current development state of the program this is acceptable because if the client code would call this method without a logged-in user that signals that something has gone wrong and the application is in a state where we have not intended it to be.

The *Listing Package* has a similar structure as the *User package* centered around a *handler* that holds the listing data. This is to control what listings to be show at what time and to who. Both the User and Listing handler are constructed as Singletons to ensure that all data objects end up in the same handler. This could be considered a problem because these classes drift towards becoming *god-classes* but with no real database this problem could be solved differently to avoid this.

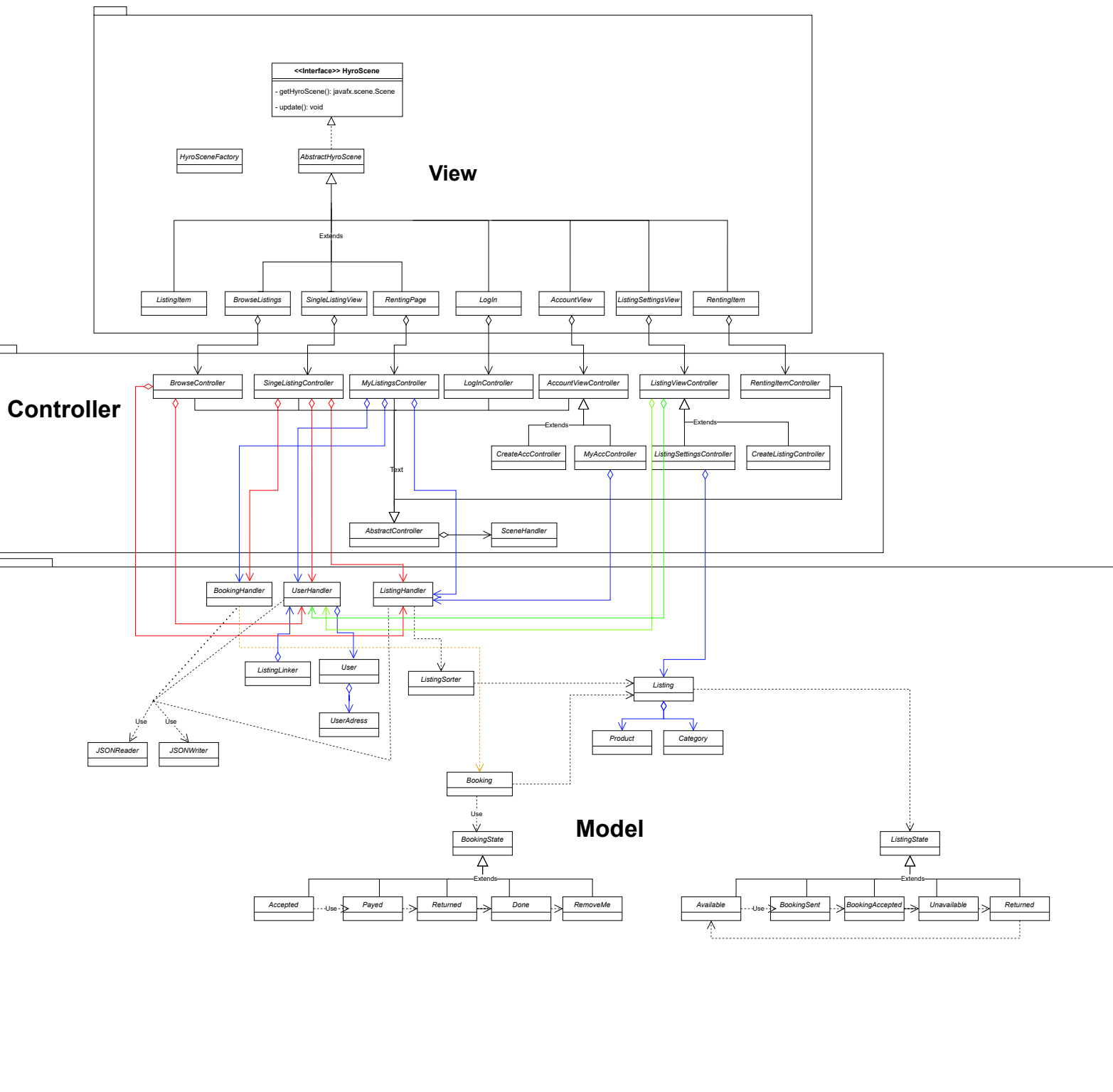
Finally, a known security issue is that user's passwords are stored as unencrypted plain text in the users.json file which is extremely bad for data security. However this application is not web based in its current version and users data could be seen as a proof of concept and therefore this issue has not been prioritised.

References

- [1] URL: <https://github.com/google/gson>.

A Appendix

UML Diagram:



Peer review

MVC Pattern

The code is structured around the Model-View-Control design pattern. Each part of the MVC is divided into its own package. The model is well isolated and through the use of interfaces it is easy to grant access only to the necessary parts of the model. A drawback of the current design is that the model exposes many important direct references, creating a risk for aliasing issues. This could be improved by re-designing getters in the Model class to pass copies of objects instead of direct references. The controller part is certainly thin as it chains calls to the model and does basically nothing else. As for the view, it contains some logic, however since it is all related to rendering, it's fine. The view classes are large, however that is inevitable since all GUI objects are created in code.

Path Class

- Abstract class path, good for extendability.
- Method createMapCollision in Path very big, use functional decomposition. Currently it is very hard to grasp.
- Usage of the Factory pattern in Path Factory, good for extensibility and easy to add new paths if needed in the future.

Projectiles

Abstract class projectiles consist of a lot of instance variables that only has getters but no setters, variables should be set to final if they are not meant to be changed once they are created.

Projectiles has a boolean variable “canRemove” that only gets used when projectiles are added to a list with objects that should be removed from the map. Instead of setting the variable to “true” the method could chain the object directly to the list to skip the boolean.

Towers

Method for changing target modes is confusing and not extensible. Would make more sense to use a target mode as a parameter. Solution would also make choosing target mode easier in game, with a dedicated button for each mode instead of buttons to scroll through the modes. Target mode could also follow the state-pattern to simplify its usage and changes.

Another solution to changing target mode would be to implement the state pattern and let each tower have a state and let each target mode be a different state. Implement it in the way

Group:17E

Date: 2021-10-17

that each state knows what comes before and after and each time a new state is needed simply create a new instance of the state.

Stats for towers are set in factory class, which might be more confusing than having them be set in the actual class for each tower.

Spawning Viruses

The matrix “SpawnInfo” that contains the spawn information for viruses in each level is “hard coded” and therefore not very reusable. Good practice would probably be some form of algorithm to calculate what/when to spawn viruses.

Utilities

The Utilities package is located outside the model package although some classes in the utilities package are not just “helper classes” but classes on their own such as GameTimer. Generally if you want to have utility classes they should be stateless and only consist of static methods, a good example is their Calculate class which is simply a helper class to the model. that they should consider being in the model. Suggestion is to move the instantiable classes such as GameTimer to a part of the model since it is just not a “helper class” but an important part of the application.

Unit tests

The model has great coverage in tests, and the tests connected to it are well made. There are also tests connected to the view which isn’t optimal for these types of tests and when they are run some errors are created but the tests pass non the less.

Coding style and documentation

The code is written with a consistent style and each class is well structured with related fields mostly placed close together, improving readability. Variables, methods and classes have names that make sense. Most of the code is covered by JavaDocs, and the JavaDocs comments are generally well written. However, the Model class, which has a lot of important public methods, completely lacks documentation.

Structure and inheritance

Inheritance is well used throughout the code, which leads to minimal code duplication. Both interfaces and abstract classes contribute to making extending the program easy, improving code reuse and creating good encapsulation. Over all, the code is well encapsulated through the use of private/package private fields, and dependencies are mostly abstract thanks to interfaces and abstract classes.

Group:17E
Date: 2021-10-17

Platform independence

We were not able to run the application on both Linux(Arch btw) and Mac following the specified instructions. This might imply that there are other steps necessary to running the application on different platforms and instructions for these steps were missing.