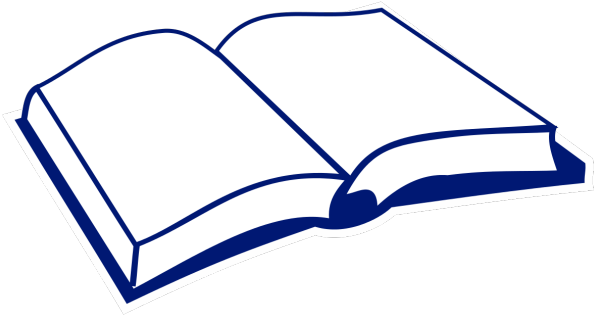

Chalmers Book Market

A detailed account of the development of a mobile application for buying and selling used course literature

Pegah Amanzadeh, Simon Holst, Kevin Pham, Carl-Magnus Wall

Supervisor: Leo Carlsson

October 24, 2021

Abstract

The purpose of this report is to account for the process behind creating a mobile application for buying and selling used course literature. The theory regarding object-oriented structure is laid out in order to provide a basis for the methodology of the development. The implementation cycle is based on agile development, where the team continuously hold meetings, where goals are defined, tasks are assigned, and where progress is discussed. The main, initial design choices of the application are discussed in detail.

The final result is a prototype with the basic features of a mobile book market. It is easy to navigate and use, but does not function as a real marketplace as of the current iteration. The features and flow of the prototype are detailed in figures.

Improvements on the initial design choices as well as the main concern regarding MVC and SceneBuilder is discussed in full, and it is concluded that it is probably advisable to use a different design tool for this type of application. The idea behind the application relies heavily on support from Chalmers, and access to their internal systems. It is concluded that a strong prototype might inspire such support.

Acknowledgements

The authors would like to thank their supervisor, Leo Carlsson, for his guidance during the project.

Contents

List of Abbreviations

CID	Chalmers ID
MVC	Model-View-Controller

Chapter 1

Introduction

In today's society, there are numerous ways to acquire course literature. However, from an environmental perspective, it would be advisable to try ones best to look for second-hand books whenever possible. While there are many options for buying and selling used literature, there is no option for Chalmers students specifically. The idea behind the application is that students of Chalmers could in theory have access to a marketplace for used course literature via their CIDs. This would mean that all account information is already available, and eliminates the need for a third-party solution where a separate account is needed. Students are meant be encouraged to settle their business on campus, while they are present for other activities.

1.1 Aims and objectives

The purpose of this project is to create a prototype of a book market application to show the benefits of such a product. The aim of this report is to account for the development process, including difficulties, and the design choices and the potential real-world application of the product.

Chapter 2

Theory

This project is based on object-oriented programming, which can be used for a myriad of different applications. Having objects that can be modified seems to be very appropriate for what this application seeks to do. The data of the application consists of users, books, listings and various utilities related to these elements. These elements can be categorized easily with the use of classes, and level of containment of their data is easy to manage. Since these objects are meant to be modified according to end-user inputs, it is crucial that the data is hidden as well as possible. It can be tricky to make sure that all information is contained properly, so it is very important to consistently test the code for vulnerabilities. The end-user should only be concerned with what they can do, not with what happens behind the scenes of the program when a button is pressed.

When working with many different objects and methods it is very important to maintain consistent naming conventions as well as clear class responsibilities, so that the code is easily accessible to foreign developers. For classes that share similar elements or functionality, it is important to work with abstraction and interface segregation, in order to make the code modular and easily re-usable.

Chapter 3

Methodology

In this chapter, the process behind realizing the application is described in detail. In short, an agile workflow is implemented to some extent. This means the process is iterative, where regularly scheduled meetings are utilized to plan and discuss what needs to be done within a given time span. At the start of each iteration the progress is discussed and any unfinished task is re-evaluated.

3.1 The implementation cycle

The first step of the process consists of basic planning, with the initial step including discussions about what types of applications that will be considered. When the choice has been made, the discussion moves on to the planning phase. The planning phase includes discussions of design approaches, task assignments and feedback. Each new iteration starts with this step, and to make sure everything is running smoothly, mid-iteration- and supervision meetings are held on a regular basis. These allow the developers to share any progress or difficulties they might have along the way. Communication is vital for this type of process.

Once an iteration has been properly planned and laid out, with goals clearly defined, the implementation step commences. Ideally, each developer will have individual tasks that they need to finish by the end of the iteration. However, this does not limit collaboration; in fact, collaboration is commended. As long as all tasks are done by the end of the iteration. In the case of a task remaining unfinished, the task will be extensively re-evaluated, and split into sub tasks if necessary.

At the end of each iteration, the developers are invited to share any feedback they might have on the project. The developers are encouraged to disclose any problems they might have, even if it includes having too little to do. This feedback is invaluable for the planning stage of the next iteration.

3.2 Main design pattern - Model-View-Controller

The application is object-oriented and strives to follow the MVC design pattern to a tee. Working with a book marketplace type application makes this is quite easy to realize. The model consists of various data, regarding users, books, listings and notifications, which can be controlled accordingly. The views represent the data and can also be controlled to display the correct data. The importance lies in the communication between the elements in the

pattern. The internal representation of the model should be as tucked away as possible, and the views should have no direct access to it. However, the views need know when something in the model has changed. The Observer pattern is utilized to achieve proper communication between the elements.

3.2.1 Secondary design pattern - Observer

The Observer pattern makes it possible to inform the views that something in the model has changed without them knowing about the internal operation of the model. When a user interacts with, for example, the search bar, the controller sends information to the model, the model calculates the results, and then notifies any observer. The pertaining views are subsequently updated according to the changes.

3.3 Maintaining best practices

To maintain good code structure, several tools are used to analyse the code on a regular basis. The main tools are SpotBugs, JDepend and PMD. These tools provide different types of information regarding the overall structural integrity of the code. Any bad practices, dependencies or bugs are reported swiftly and effortlessly. Thanks to using Maven as a build tool, both SpotBugs and JDepend can be integrated without any external installations. Consistently running these tools on the code helps keeping the code as clean as possible.

3.4 Summary

In this chapter it was discussed how an iterative agile workflow can enhance software development and, with the help from using best-practice conventions along with well established design patterns, result in exponential project growth; in quality as well as quantity.

Chapter 4

Testing and validation

To make sure the code is working as intended, unit tests of logic and methods are performed. For this purpose JUnit5 is used. Tests are divided into proper test packages within the Test root folder. This root folder is located in src/main. Test coverage is 91% as of the current iteration.

4.1 Testing methodology

The objects that are mutable and are capable of modification are located in the model of the code. The model is composed of several lower level classes with internal methods, of which the majority are package-private. The ViewController communicates with the model when an action has been performed. The highest level of the model, called ApplicationModel, has numerous public methods that delegate operations to lower levels of the model. These public methods are the most vital methods to test, since they implicitly test the more hidden methods of the lower levels of the model. The goal is to have 90% test coverage.

Chapter 5

Results

The current iteration of the application does not make use of any external components. The functionality is contained within the code itself. When launching the application, the user is greeted by a login screen. The idea is that users will enter their CID and their respective password. Once logged in, the user will see the main browsing page, where popular books are presented in lists, and where one can search for books one might be interested in. At the bottom there is a global navigation bar, where the user can access their account, add a book for sale, and access the main shop page. Users are encouraged to browse the application freely. Like previously stated, in the current iteration, the applications functionality is contained within the code. When the application is closed, the session will not be saved as of the current version.

A detailed showcase of the resulting prototype can be found in Appendix B.

Chapter 6

Discussion and Analysis

During the project, several changes and improvements were made to the code. Initially, only two design patterns were actively pursued. Below, the discovery and solutions to problems faced during the development process are discussed.

6.1 SceneBuilder and MVC structure

To construct and design the view, SceneBuilder is used. However, this leads to certain difficulties pertaining to the view and controller elements of the code. The pattern enforces modularity through its division of the model, view and controller. However, SceneBuilder tightly couples the view and controller; a connection that prohibits a package division between the two. As consequence, a clean separation is impractical and to dissociate them and not leave the two elements sharing a package would be counterproductive.

If we look beyond the unique implementation there is still a tremendous focus on separation of concern. The joint package is further conducted in order to avoid masking a bad design as an aftermath of the merged elements. The *model* and *viewcontroller* packages are completely separated; this is partially established by bug inquiries found through maintenance software (see 3.3).

6.2 Database handling - Singleton pattern

The application's archetype data management consists of prototype databases. Given the project's time frame, a decision was made early on to not implement concrete databases rather than mocked versions. In order to ensure that only one database, per data type, exists the Singleton pattern is used. It serves its purpose well in the design, despite being acknowledged as an anti-pattern by industry standards. Since the drawbacks could violate *Single Responsibility Principe*, as well as camouflage faulty design, the implementation is strictly to avoid mutability where it is unwanted. The only appraised danger is that the current utilization does not support multi-threaded activity; this is secure in the current iteration since the client is single-threaded.

The benefits of the implementation is inhibiting multiple instances of the database types. This benefits the model since no client code with malicious intent can create unintended instances of the databases.

6.3 Information-hiding - Prototype pattern

In order to provide the client code with information, in the form of an immutable set of primitive variables, the *Prototype Pattern* was implemented. When the client code requires classes representing Listings, Books and Users to be sent the implementation protects the internal representation of the classes from being unnecessarily exposed, and create dependency as consequence. Another perk favoring the design is syntax clarity; an object calling the *clone*-method mediates that the object is being copied, without having to worry the client that underlying reference variables are not being safely copied.

6.4 Dividing responsibilities

During the development process the code was review by a group of peers, and one of the key takeaways was that the User class had a bit too much responsibility. The User class held methods and logic for ratings, as well as quite a bit of duplicated code pertaining to lists of listings and subscriptions. The ratings logic is currently placed in a UserRating class and a UserAsset class is implemented to handle different types of data that share similar attributes. Reinforced with general types, the UserAsset class imposes polymorphic abilities and can be dynamically applied to the type of data the domain code is interested in implementing.

6.5 Code analysis and testing

Since the prototype is developed in Java 16, there are issues with some of the suggested tools for structure- and quality analysis. Both STAN and FindBugs seem to be Java 8 exclusive, so the usage of them were out of the question. Since these issues persisted, analyses were postponed until they were desperately needed. Research into PMD did not begin until the code was assessed by a third party. PMD helped sorting out quite a few issues that were present in the code. During the final three weeks, substitutes for STAN and FindBugs were found, namely JDepend and SpotBugs (successor of FindBugs). When these tools had finished their analyses, it became clear that the code was rather smelly. While most bugs and bad practices were handled with appropriate methods, it definitely stole some development time when it would have been better spent on other aspects of the code.

Testing was also something that came along as an afterthought in a lot of cases. It was pure luck that great test coverage was achieved without much modification of currently applicable methods.

6.6 Unfinished business

During the planning phase, numerous features were considered. There are still a few features that remain unimplemented for various reasons; main reason being time constraints. Adding pictures to listings remains completely unimplemented, for example. There is also a ratings system planned; a class and logic is put in place, but it lacks operations and a related GUI. It is also not possible to add listings of books that are not in the predefined database; currently, nothing happens when trying to do so. The plan was to display a modal panel that informs the user that the book code entered is not present in the database. The user would then be asked if they would like to add the book anyway or cancel the operation. If the user chooses to add the book, the listing will be pending until approved by an administrator.

The notification system is only partly implemented, with some notifications working as intended. A seller does currently no receive a notification when somebody reserves a book that the seller has up for sale, and buyers do not get a notification when they have successfully reserved a book. The logic is, however, in place for future development.

Chapter 7

Conclusions

While the application works as a proof-of-concept, any further development towards a functioning marketplace would depend on support from Chalmers and access to their internal systems. For this particular course, which focuses on the planning, implementation and completion of an object-oriented project, limiting the scope to a prototype is more than sufficient. Scoping is an essential part of project planning, and there is little reason to overestimate what can be done within a given time span. Working in an iterative fashion really helps with tracking progress and seeing if the scope is of reasonable size.

SceneBuilder is, in various ways, not very user-friendly. There are countless small things to consider while working with the software. For example, if some code is pushed to a repository, random issues can seemingly arise if SceneBuilder is not properly closed beforehand. This is extremely frustrating to deal with, and causes unnecessary backtracking. The software seems to be past its heyday, in regards to both functionality and usability.

In Chapter 6 it is mentioned that are certain features that are not yet implemented, but there is room for expansion. The unfinished features can be added without much effort. This is great practice, and provides encouragement for future development. The main idea behind the application is to let students at Chalmers have an easily accessible market for used course literature, where they are encouraged to complete transactions at the university campus. However, if the application is to be Chalmers-contained, and the login method is to be restricted to one's CID, then, like stated, direct support from the school is required. This kind of support cannot just be handed out blindly, however, it needs to be justified. Bringing a solid product, or prototype, to the table could inspire such support. This is an important lesson and something to consider for future endeavours.

Chapter 8

Reflection

During the process of developing this application, we learned a great deal about teamwork in coding situations. When working collaboratively on something as delicate as code, it would seem that communication is the key to success. Initially, we cannot say we were a well-oiled machine in terms of communication. There are quite a few things we could have improved on, including sharing our previous experiences with programming projects and programming in general. Of course, since we are all software engineering students at Chalmers, we have partaken in the same classes, but discussing what each one of us are comfortable with early on could be very beneficial for the initial stages of development. We still managed to get the ball rolling very quickly, since we were able to decide on a project on day one. However, we might have moved on from the planning stage a bit too early. Quite a few design ideas, both in regards to the code and the GUI, came along later rather than sooner. While some of these ideas were mere improvements we found along the way, we believe that proper planning could have led to an earlier discovery of them.

Despite the initial bumps in the road, we met on campus at least three times a week, to conduct meetings and work as a group. This let us get to know each other well, and eventually the workflow became very natural. We never really stood still, and we are proud about this fact. However, this does not mean we did not face any crippling setbacks. The main issue we encountered was the integration of SceneBuilder. We spent hours, desperately trying to separate the views from the controllers, and this was not time well-spent. While we want to commend our efforts trying to solve this independently, we should have reached out to our teachers earlier to straighten this issue out and get on our way. The solution we decided on is still compliant with the MVC pattern, which is what matters in the end. If we were to work with GUI design and functionality in the future, we would most likely pick a different route. We did not experience many devastating merge conflicts, but the majority of them were SceneBuilder-related. We think this is a horrible piece of software for collaborative work.

Overall, we are pleased with how the application turned out. We also feel like we established reasonably sized scope for the project and we certainly think we delivered within that scope, especially given the time frame.

Appendix A

System Design Document for Chalmers Book Market

A.1 Introduction

In this document the system architecture of the application Chalmers Book Market will be asserted and evaluated. Chalmers Book Market is intended to function as a meeting place for people who wants to deal in used course literature. The idea behind the project is that students of Chalmers might benefit from a book market application that is directly connected to their Chalmers ID. The students would be easily able to interact with each other in order to buy or sell their used course literature.

It is vital that the system architecture is extensible and modular; this ensures future expansion for Chalmers internally, as well as for other possible adopters. This design document discloses the inner workings of the system engineering as well as its qualities and flaws.

A.1.1 Definitions, acronyms, and abbreviations

- CBM - Chalmers Book Market
- CID - Chalmers ID

A.2 System architecture

Upon launching the application the model is instantiated, consequently creating the database in its constructor. The databases are predefined with mocked books and listings which are meditated, through the observer pattern, to the views to be displayed. Since the model authorizes access to the databases it is responsible for the login logic. It can, without exposing internal representation of the user database, decide if login credentials are valid or not.

Following entering credible login information, the corresponding user in the database is allocated as an alias in the model representing the logged in user, performing actions in the application. With the model as interface, the user can later be mutated desirably through the methods in the model. To empower the model with information, enabling it to make valid decisions, it also utilizes the listings and books from the other databases. There is a fine line between being a super class, and delegating work with composition and high-level logic accordingly. This structure is preferred since it allows low-logic classes to perform these actions without acknowledging each other's existence.

The entire model, contained in the *model* package, is honed to safely copy reference variables. As result the data comparisons are inclined to prefer value semantics over reference, since memory allocation comparisons are inconceivable.

The current iteration of the application does not make use of any external components. The functionality is contained within the code itself.

A.3 System design

Due to technical difficulties with SceneBuilder, more specifically the FXMLLoader-class, the separation of view and controller, to establish the MVC-pattern, has been inconceivable. The main reason is that FXMLLoader requires a controller when the view is initiated, at which state the controller-objects has not yet been instantiated, due to the fact that they want to have a reference to the view objects. The end result is a necessity to merge the view and controller into the same classes, as shown in Figure A.1.

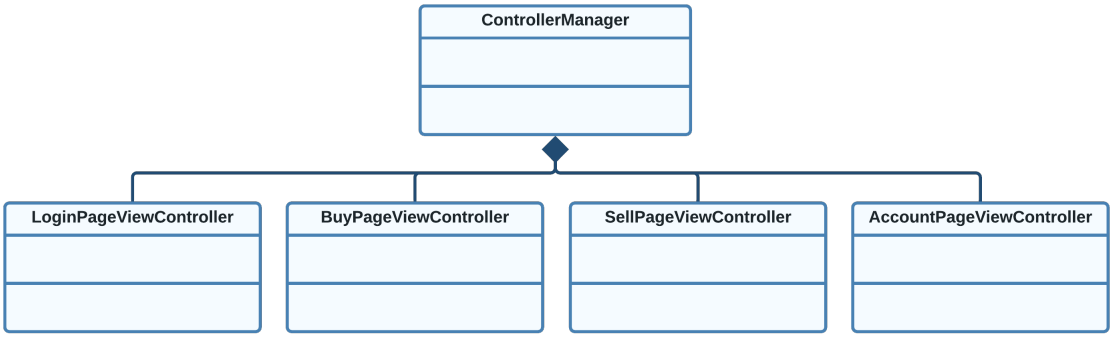

Figure A.1: The application's solution to SceneBuilder conflicting standard MVC implementation.

The consequence of this becomes a reduction of packages, since the controller and view must share the same package of classes, as shown in Figure A.2.

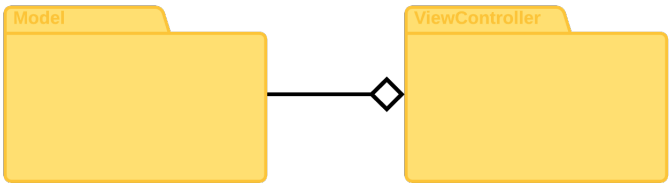

Figure A.2: The application's package division.

The controller has a reference of the model, enabling it to call the required methods in the model corresponding to the graphical interface elements interacted with by the user. The model changes accordingly and notifies the view that an update has been made and that it should change.

The current iteration has not yet separated the elements of MVC into the two different packages, but the separation is handled through proper uses of composition, aggregation and the Observer pattern. The current structure of the application is shown in Figure A.3.

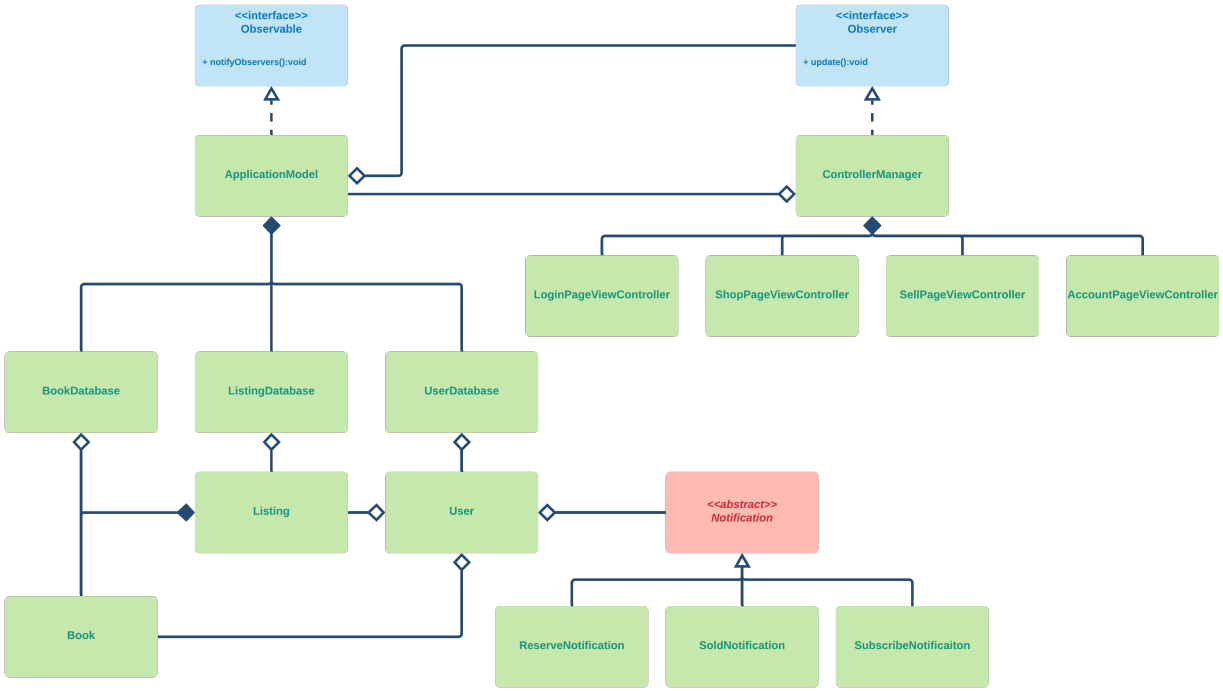

Figure A.3: The application's design UML diagram

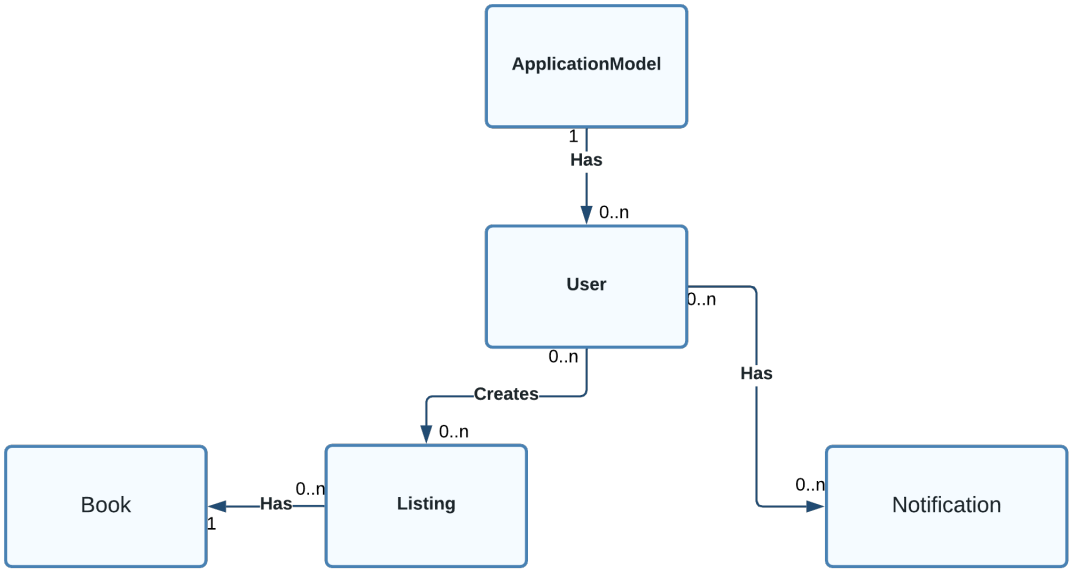

Figure A.4: The application's domain model.

The domain model describes the basic composition and functionality of the application. The main idea is that the model consists of users, listings, notifications and books, and these are the building blocks of the application. The design model shows how the model data is contained on a more local, detailed level, and how it can be controlled and represented by the GUI. It describes in detail how the classes of the application are connected and how they are meant to communicate.

A.4 Persistent data management

The application insists on having having three different database classes, where books, listings and users are stored in their respective database class. Due to time constraints, the decision to go for a proof-of-concept approach was made quite early on in the project. The current version does not make use of any persistent data, other than stock images. A simple solution would be to dump the data, requiring saving between instances, in text files. A more sophisticated approach would be to make use of proper databases. Currently the stock images for books and listings are stored in a folder in the repository.

A.5 Quality

To make sure the code is working as intended, unit tests of logic and methods are performed. For this purpose JUnit5 is used. Tests are divided into proper test packages within the Test root folder. This root folder is located in src/main. Test coverage is xx% as of the current iteration.

To maintain good code structure, several tools are used to analyse the code on a regular basis. The main tools are SpotBugs, JDepend and PMD. These tools provide different types of information regarding the overall structural integrity of the code. Any bad practices, dependencies or bugs are reported swiftly and effortlessly. Thanks to using Maven as a build tool, both SpotBugs and JDepend can be integrated without any external installations. PMD does have a Maven plugin, Consistently running these tools on the code helps keeping the code as clean as possible.

A.5.1 Access control and security

The application has user logins, but these consist of mocked up data in the current iteration.

A.5.2 Known bugs

Currently the Singletons of the application are not thread safe, but since this is a prototype, and only one instance of the application is running at any point in time, this is not a major issue.

Another observed flaw is exposed upon trying to create a listing with an invalid book. This results in no back-end changes, even though the front-end graphical user interface still behaves as if a new listing has been added to *Published Books*. If the scope of the project was increased and the time frame was extended the solution would have been to display a modal panel to the user, informing them that the book currently does not exist in the book database; they are offered the choice to manually enter all book specifications which will be reviewed by an administrator before being published to the application. The benefit of letting them enter the information themselves is reduced workload as well as allowing the user to create the listing before review, rather than being forced to wait for an administrator to create the book.

A.6 References

The following platforms, build tools, libraries and design tools were used to develop the application.

A.6.1 IDE

JetBrains IntelliJ IDEA

A.6.2 Build Tools

Maven 3.8.1

Imported libraries

- JUnit5
- JavaFX 16

Structure analysis

JDepend 2.10

Quality analysis

- PMD 6.39.0
- SpotBugs 4.4.1

A.6.3 Design Tools

- LucidChart for UML and domain model design
- Figma for GUI design
- SceneBuilder to construct the GUI and connect it to the code

Appendix B

Requirements and Analysis Document for Chalmers Book Market

B.1 Introduction

The application's purpose is to benefit all Chalmers students, creating an universal platform where previously owned student literature can be reacquired for second hand use.

Potential stakeholders include, but are not limited to, Chalmers University of Technology, which might be interested in creating a more sustainable market of student literature with an increased life span.

The target group of the product will be the students at Chalmers University of Technology. Although this application may not be relevant for an extended list of schools yet, this document aims to serve as a foundation, discussing and analysing requirements that could prove useful for extended models.

The main functionality of the application is to be able to purchase and sell used course literature. The application is targeted to a mobile audience.

B.1.1 Definitions, acronyms, and abbreviations

- CBM - Chalmers Book Market
- CID - Chalmers ID

B.2 Requirements

The application seeks to work as a prototype of a mobile book market, tailored specifically for students at Chalmers. This means the application should at the very least showcase the idea of logging in with a CID, being able to search for different types of course literature, being able to put a book up for sale, being able to subscribe to certain literature and being able to reserve (i.e. buy) books. Ultimately, the software is a proof-of-concept more than anything else.

To realize the application, the following user stories and criteria are considered.

B.2.1 User Stories

1. Story Identifier: CBM001 - IMPLEMENTED
   Story Name: Main Feature - Ability to sell

   Description:

   As a Chalmers student I would like to be able to easily sell my used literature, in order to make some extra money.

   Confirmation:

   - Are all elements pertaining to selling implemented?

   Functional:

   - Can I freely pick a price point?
   - Can I add an indication of book condition?
   - Can I edit or remove a listing?

   Availability:

   - Can I list a book at any time of the day?
   - Can I access my account page at any time of the day?
   - Will notifications regarding my listings be sent immediately?

   Security:

   - Am I the only who can see whom I have sold a book to?

1.2. Story Identifier: CBM012 - IMPLEMENTED
Story Name: Listing Details

Description:

As seller I want to be able to upload a description and price to increase the odds of getting consumers.

Confirmation:

- Is it possible to add a listing in the application?

Functional:

- Can I adjust the price freely, in order to attract different customers?
- Can I freely add a proper description of the book I am trying to sell?
- Can I adjust these after publishing the listing?

Availability:

- Can I edit my listing at any time of the day?

Security:

- Am I the only who can edit my listings?

1.3. Story Identifier: CBM013 - NOT IMPLEMENTED
Story Name: Listing picture

Description:

As seller I want to be able to upload a picture of the book, in order to prove the condition of the book.

Confirmation:

- Are picture uploads implemented?

Functional:

- Can I upload a picture while adding a listing?
- Can I upload in JPEG or PNG?
- Can remove sale notifications?

Availability:

- Is it possible to upload a picture at any time of the day.

Security:

- Are my uploads safe?

1.4. Story Identifier: CBM014 - NOT FULLY IMPLEMENTED
Story Name: Sale notification

Description:

As user I want to receive a notification whenever there has been an update regarding book reservations, book subscriptions, or if I have made a sale.

Confirmation:

- Are all different notifications implemented, and are they pushed accordingly?

Functional:

- Can I see when somebody has bought or reserved one of my books?
- Can I reach my sale notifications from the home screen?
- Can remove sale notifications?

Availability:

- Will notifications be sent immediately when sales or reservations are made?

Security:

- Am I the only who can see my notifications?

2. Story Identifier: CBM002 - IMPLEMENTED
   Story Name: Main Feature - Ability to buy

   Description:

   As a Chalmers student I would like to be able to easily purchase used course literature from other Chalmers students, in order to save some money.

   Confirmation:

   - Are all elements pertaining to purchasing implemented?

   Functional:

   - Can I reserve literature and make a purchase later?
   - Can I remove a reservation?
   - Can I abort a purchase?

   Availability:

   - Can I list a book at any time of the day?
   - Can I access my account page at any time of the day?
   - Will notifications regarding my listings be sent immediately?

   Security:

   - Am I the only who can see who I have sold a book to?

2.2. Story Identifier: CBM022 - IMPLEMENTED
Story Name: Book reservation

Description:

As a buyer I want to be able to reserve a book, in order for me to make a purchase at a later point in time.

Confirmation:

- Is it possible to reserve a book?

Functional:

- Can I reserve literature and make a purchase later?
- Can I remove a reservation?
- Can I see all my current reservations?

Availability:

- Can I reserve a book at any time of the day?
- Can I see my reservations at any time of the day?

Security:

- Am I the only who can see which books I have reserved?

2.3. Story Identifier: CBM023 - NOT FULLY IMPLEMENTED
Story Name: Seller rating

Description:

As a buyer I want to be able to see the seller's rating, in order for me to know whether the seller is credible or not.

Confirmation:

- Is it possible to see a seller's rating when looking at a listing?
- Can I rate a seller after making a purchase?

Functional:

- Can I rate the sellers I buy from?
- Can I clearly see how highly or lowly rated a particular seller is?

Availability:

- Are the ratings updated immediately after a seller has received a rating?
- Are the seller ratings available at any time of the day?

Security:

- Can others see what ratings I have given?
- Can I see what ratings buyers have given me?

2.4. Story Identifier: CBM024 - NOT FULLY IMPLEMENTED
Story Name: Purchase confirmation

Description:

As buyer I want to be able to get a confirmation of purchase to ensure the purchase was successful.

Confirmation:

- Is the notification system implemented?

Functional:

- Do I get a confirmation when a purchase has been completed?

Availability:

- Do I get a notification as soon as a purchase has been completed?

Security:

- Can others see that I've completed a purchase?

2.5. Story Identifier: CBM025 - IMPLEMENTED
Story Name: Book search

Description:

As buyer I want to be able to search for books in order to find exactly what I seek.

Confirmation:

- Is it possible to search for books? Do I get a list of results that I can interact with?
- Can I search for author, book code or category?

Functional:

- Can I search for books?
- Can I search by book name?

Availability:

- Can I search for books at any time of the day?

Security:

- Can others see my search history?

2.6. Story Identifier: CBM026 - IMPLEMENTED
Story Name: Listing details 2

Description:

As buyer I want to see a book description in a listing in order to confirm that I have found the correct book.

Confirmation:

- Does all listings have descriptions?

Functional:

- Can I see a description of the book I am looking to buy?

Availability:

- Is a book description required?
- Can I always read the descriptions?

Security:

- N/A

3. Story Identifier: CBM003 - IMPLEMENTED
   Story name: Personal account

   Description:

   As student I want to be able to have a personal account so I can have personal functions such as being able to subscribe to books, see my purchase history etc.

   Confirmation:

   - Are all elements pertaining to the personal account implemented?

   Functionality:

   - Can I log in with my CID?
   - Can I see a list of my reserved books?
   - Can I see a list my published books?
   - Can I see a list my previous purchases?
   - Can I see a list my subscribed books?
   - Can I see my E-mail address?
   - Can I see my rating?

   Availability:

   - Can I access my account at any time of the day?
   - Can I edit my different lists at any time of the day?

   Security:

   - Can others see my account details?
   - Can others see my reservations, purchases and subscriptions?

B.2.2 Definition of Done

For each user story, certain criteria regarding functionality will be set up. Each task adapted for a certain user story is directly connected to logic and methods in the code. To assert that a task is done, proper testing of the methods and the logic behind it will be required. Testing will be performed using unit testing with JUnit. Each piece of logic, and each method will be put through a series of tests to assert that they are performing as expected. Once all tasks related to a user story pass testing, then the tasks will be reviewed by all members of the group. If nobody has any objections, the tasks and the corresponding user story will be considered as done.

B.2.3 User interface

In the sketch, which commenced the development, a global navigation towards the bottom of the screen was established, as shown in Figure B.1.

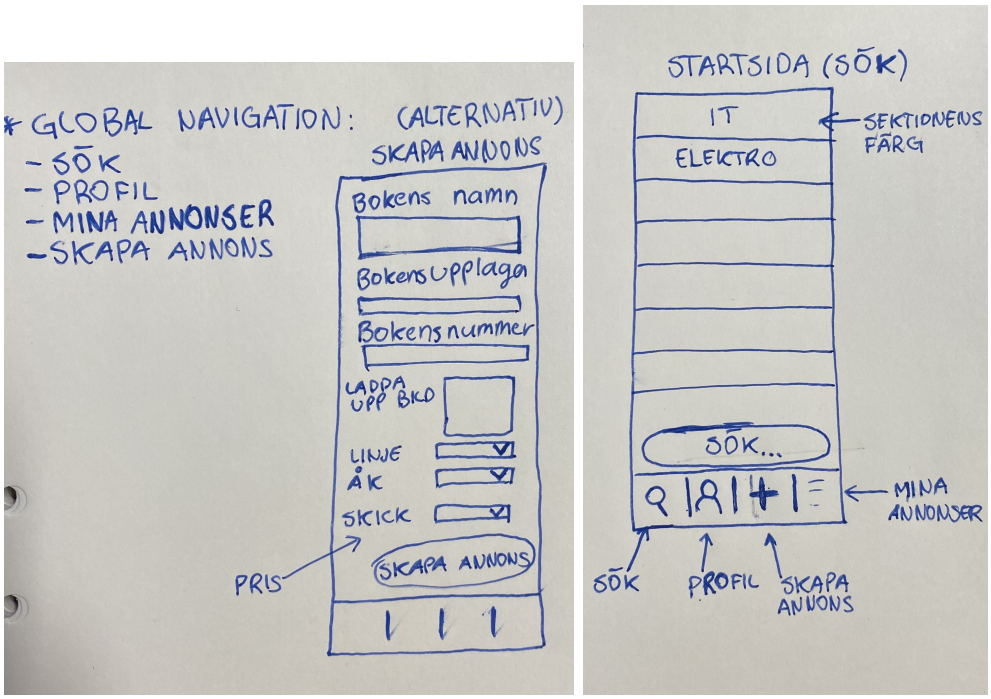

Figure B.1: Initial sketch of the application's user interface. Left: Page destined to create new listings. Right: Categorized books under different school institutions.

Asserting an always-on-display navigational option offers an escape hatch to the user as well as the convenience of reducing navigational excise.
When the paper-sketch felt thought through and roughly finished the design process was continued in the digital design tool Figma. Several concepts were created to diversify the design in order to explore all conceivable possibilities, as shown in Figure B.2.

Figure B.2: Multiple interpretations of the application's user interface.

After the majority of our design model was realized in actual code the application's graphical user interface was finalized and polished.

Upon entering the application a login screen greets the user, encouraging them to enter their CID as well as their password, see Figure B.3.

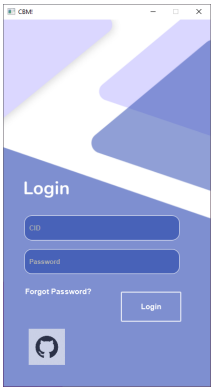

Figure B.3: Application's login page.

Subsequently, the user accesses the shopping page; the view offers a wide variety of categorized

books to the customer, see Figure B.4.

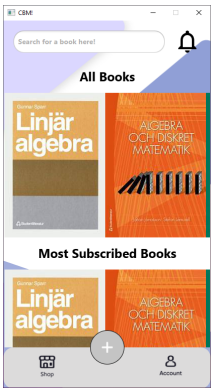

Figure B.4: Application's shop page.

From this page the user can browse books by categories or searching for a specific book. When a compelling book is found the customer can click on its cover to enter the detail view, see Figure B.5.

Figure B.5: Application's detailed view of a book. Left: Subscribed to a book. Right: Unsubscribed to a book.

Multiple possibilities become available to the user; it can either subscribe to the book to receive notifications when new listings are published with the desired book, or view currently published listings in the form of a list at the bottom. If any published listing seems lucrative the user is able to click the listing in order to bring up its individual page, see Figure B.6.

Figure B.6: Application's detailed view of a listing.

Detailed information is amassed here, including a description where the seller can specify deviations regarding the book interesting to the buyer. If the user decides to reserve the book, by clicking the *Reserve*-button, the listing disappears from the shop page and is only displayed in the account page, see Figure B.7.

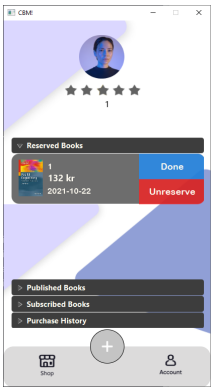

Figure B.7: Application's account page, currently displaying the *Reserved Books* section.

From here the user is able to unreserve the listing if the reservation was accidental. Otherwise the customer can contact the seller through their Chalmers mail to setup a meeting on campus where the trade can be completed. When both parts feel satisfied with the exchange they press the *Done*-button to confirm that the trade i successfully complete. Consequently the trade is moved to the section *Purchase History*, see Figure B.8.

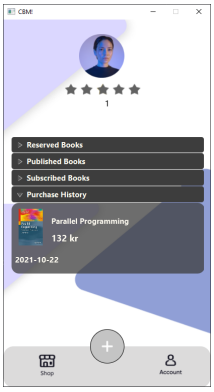

Figure B.8: *Purchase History*, displaying the recently made purchase.

Additionally, published and subscribed books can be view here, see Figure B.9.

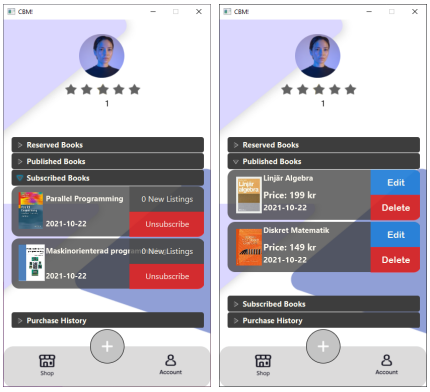

Figure B.9: Left: *Subscribed Books*. Right: *Published Books*.

In order to publish a book one must navigate through the global navigation to the sell page, see Figure B.10.

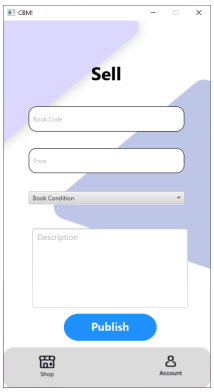

Figure B.10: Application's sell page.

All corporated, we have a complete graphical interface capable to assist the user in acquiring their desired course literature.

B.3 Domain model

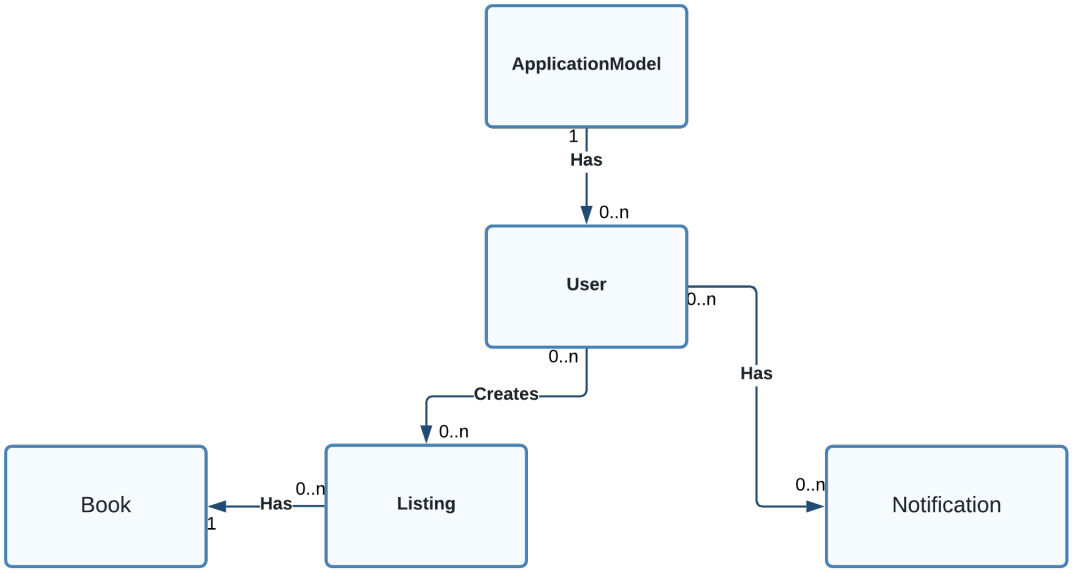

Figure B.11: The application's domain model.

B.3.1 Class responsibilities

The classes are divided to induce modularity and affirm the desired object-oriented design.

ApplicationModel

This class contains the core functionality of the application's model. Superior application tasks such as: creating new listings, updating views through observers and has multiple databases containing users, listings and books.

User

The User class holds all data as well as represents an instance of a specific user.

Listing

The Listing class holds all data pertaining to a specific book listing.

Book

The Book class serves as a container of information of a specific book type. The differentiation from the Listing class is that a Book object is not for sale nor does it have a condition or a seller.

Notification

The Notification class holds and manages all types of different notifications.

Empty classes: EmptyUser, EmptyBook and EmptyListing

The *Empty* classes are placeholders, intended to avoid managing null values in the model. They are not purposed for use, but rather skeletons imitating an actual object.

B.4 References

The following platforms, build tools, libraries and design tools were used to develop the application.

B.4.1 IDE

JetBrains IntelliJ IDEA

B.4.2 Build Tools

Maven 3.8.1

Imported libraries

- JUnit
- JavaFX 16

Structure analysis

JDepend

Quality analysis

- SpotBugs
- PMD

B.4.3 Design Tools

- LucidChart for UML and domain model design
- Figma for GUI design
- SceneBuilder to construct the GUI and connect it to the code

Appendix C

Peer review of Grapefruit (Swedish)

Kodanalys

Model

GameModel:

- Både observers och players är package-private, varför? GameBoard är public, farligt då vem som helst har tillgång till referensvariabeln.

GameBoard:

- Alla globala variabler är package-private, avsiktligt? Om inte borde dessa sättas till private.
- movePlayer-metoden *overloadas* med två olika input-parametrar, men enligt medföljande JavaDoc ska metoderna göra två skilda saker. En av metoderna borde byta namn.
- Metoden rollDice returnerar för närvarande ett mock-värde 2, den senare implementerade logiken bör extraheras till en Dice klass som har en liknande metod, precis som ni har tänkt i er UML.

Map:

- Metoden deHighlight "släcker" samtliga noder, detta borde eventuellt reflekteras i namnet på metoden.

Node:

- Metoden addRelatedNode har följande kod:

```
if (relatedNodes == null){
    relatedNodes = new ArrayList<>();
}
    relatedNodes.add(node);
```

  Lättare vore att bara initiera listan direkt uppe vid konstruktorn som ej har listan som parameter:

```
public Node(IPosition position) {
    this.position = position;
    relatedNodes = new ArrayList<>();
}
```

MapFactory:

- Klassen har en publik implicit konstruktor, så oavsett att den har en statisk initiationsmetod kan man initiera den med en tom konstruktor, vilket inte är intentionen och borde förhindras.

PlayerFactory:

- Klassen har en publik implicit konstruktor, så oavsett att den har en statisk initiationsmetod kan man initiera den med en tom konstruktor, vilket inte är intentionen och borde förhindras.
- Returnerar null om spelarantalet som angetts är över 4, farlig kod generellt. antalet skulle kunna kontrolleras innan listan skapas, den borde definitivt inte vara null. Initieringenmetoden skulle också kunna kasta ett undantag om metodförhållandena inte uppfylls.

PlayerColor:

- JavaDoc för metoden evaluateResourceString lyder "Evaluate the player color and match it with the right *view-resource*", då låter det som att denna resurs borde ligga i view-paketet, inte i modellen.

NormalPosition:

- Här är också visuella aspekter av programmet, i modellen:

```
 @Override
public String getResourceString() {
    if (isHighlighted){
        return "node-view-highlighted.fxml";
    }
    return "node-view.fxml";
}
```

Dessa borde också flyttas till view.

View

GameBoardView:

- Alla globala variabler är package-private, avsiktligt eller kan de evenutellt göras private? För nuvarande kan de det då paketet endast innefattar NodeView, som inte utnyttjar tillgången.
- Behöver den ha en public konstruktor? För tillfället kan man instansiera klassen direkt från den publika konstruktorn.
- GameBoardView är FXMLLoader klassens controller, känns inte helt intuitivt?
- Metoden redrawChildren skapar nya FXMLLoaders för varje positionerbart objekt när update-metoden kallas, förmodligen inte så processoreffektivt? Dessutom använder den en "enhanced for-loop", men använder en yttre variabel *i* för att inkrementera och se efter när loopandet är färdigt. Känns som att det skulle vara snyggare med en konvetionell loop.

NodeView:

- Variabelnamnen kan vara mer genomtänkta (e.g. x,y; i förhållande till vad?, position som refererar till en FXML-cirkel).
- Återkomstmodifierare används blandat (e.g position är public, varför?)

Controller

GameBoardController:

- GameBoardView initieras här, borde försökas initieras i HelloApplication istället. Om inte view kan initieras utan controller känns separationen mellan de två onödig då de ändå har ett starkt beroende av varandra.

Övrigt

Koden är i stora drag byggt efter MVC-mönstret och uppfyller det i stor utsträckning, med de inbakade visuella aspekterna som undantag.

Designprinciper följs och en separation av koncern mellan klasser görs. Ett exempel är implementring av designmönstret *Observer*; förövrigt används *Interface Segregation* flitigt. Dessutom använder flera klasser mönstret *Factory*.

Koden är i största laget konsekvent, med såväl konventioner som intuitiv namngivning på paket, klasser, metoder och variabler.

På grund av stadig implementing av gränssnitt är koden betydligt lättare att återanvända då klienten är mer begränsad i vilka användningsområden koden kan nyttjas.

Eftersom klassuppdelningen är gedigen blir underhållningen av koden betydligt lättare. Funktionalitet kan enkelt läggas till / tas bort vilket är tacksamt så man slipper utföra *shotgun surgery*.

JavaDoc finns på flertalet metoder men är lite inkonsekvent på andra. Ett behov av klassbeskrivningar finns på samtliga klasser. En mindre dokumenteringsinsats skulle eliminiera problemet.

Koden är för tillfället väldigt bergränsat testad och kommer i framtiden behöva utöka testarean, vilket gruppen själva specificerar i sin *definition av färdig*-sektion.

Det finns vissa säkerhetsproblem med offentliga åtkomstmodifierare som omotiverat definierats så. Detta borde definitivt ändras.

Den kontinuerliga initieringen av nya FXMLLoader-objekt skulle på större skala kunna orsaka prestandaproblem och borde överses för en bättre lösning.

Koden är väldigt enkel att förstå och tydligt strukturerad på en syntax-nivå. Ett utförligt UML-diagram hade underlättat analysen.

Appendix D

Peer review by HandyMen

Does the design and implementation follow design principles?

Does the project use a consistent coding style?

- There exist a class that starts with lower case letter. This is probably just a mistake but worth noting.
- Good naming of methods with consistent naming of adjectives.

Is the code reusable?

- The tight coupling between the model and the FXML classes prevents this as of now.

Is the code easy to understand?

- Not having the files in packages makes it hard to get a overview of the code by just looking in the src folder. Clear UML diagrams and good documentation makes up for this though.
- Listing is well specified and self contained.
- Good choice to at gather the controllers in a manager class with current code structure.

Are design patterns used?

- Observer pattern is implemented in a straightforward way which is nice.
- Singleton pattern is used frequently which could risk introducing high coupling. Maybe should be more careful about this.

Is the code documented?

- The JavaDoc coverage is very good as almost all methods and classes are documented. However,most documentation of classes could be more insightful on what the class represents and what it is does. The JavaDoc also has everyone in the group as authors for every class which seems highly unlikely, and if true they use an inefficient working method

Is the design modular?

- Even though we guess the database is a mock-up, having the database as a singleton and without an interface in-between the client code risks introducing tight coupling that would make it hard to modify the code in the future.
- Just as you mention in the SDD, the code does not follow the MVC-pattern. However, we do not agree that the use of the FXMLLoader class hinders the implementation of MVC. The FXML-Controllers could be seen as the view class that only passes on the inputs from the user to the actual controller. This controller would translate the input to corresponding functionality in the model.
- There are several unnecessary dependencies as many classes have unused imports, these can be found in the PMD-report.
- There are instances where two different classes hold instances of each other (e.g. BookDetailViewController and ControllerManager), this causes too tight coupling between them.

Possible improvements?

- There is a possible violation of Liskov Substitution Principle by having NotLoggedInUser in-herit from User. Should a non existent user offer more functionality than an existing? Maybe restructure this relationship?
- ApplicationModel has a lot of responsibility which could be a conscience design choice to make a facade of the model for the clients to use. This seems unlikely though as there is code in "ViewControllers" that access and manipulates low level classes of the model.
- Consider making the abstract class Notification an interface instead. Furthermore, the user class probably shouldn't handle notifications. It makes more sense if the listing class does this and pushes a notification to the owner of the book when it's sold. When a new listing is created it should notify all users that subscribe to the given book that it is now available.
- No testing has been done which makes it hard to see if the prototype is actually working.
- The round() is misplaced in the User class and possible solution could be to introduce a utility class with a static method instead.
- The rating system used in class User could be separated into its own class to make User follow Single Responsibility Principle. This would also increase the object-oriented aspect of the code.