System Design Document for Chalmers Book Market

An application encouraging reuse of student literature

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

Group X

TDA367 Object-oriented programming project

Software Engineering
Chalmers University of Technology
October 2021

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.

1.1 Definitions, acronyms, and abbreviations

- CBM - Chalmers Book Market
- CID - Chalmers ID

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.

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 1.

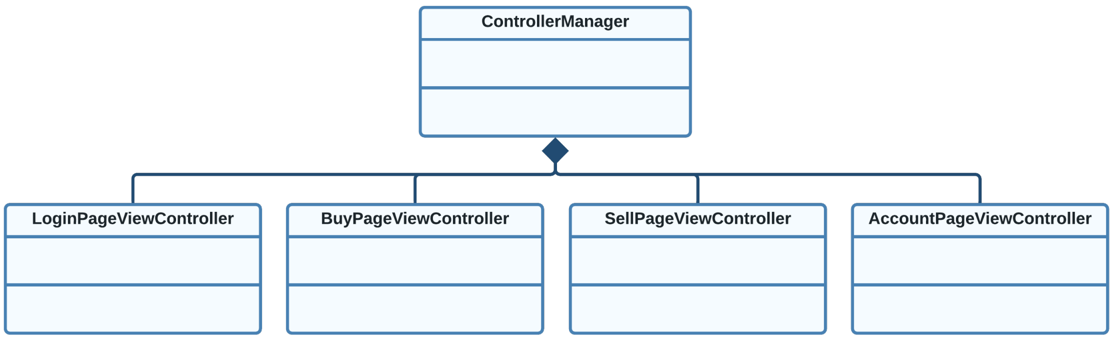

Figure 1: The application's solution to SceneBuilder conflicting standard MVC implementation.

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 2.

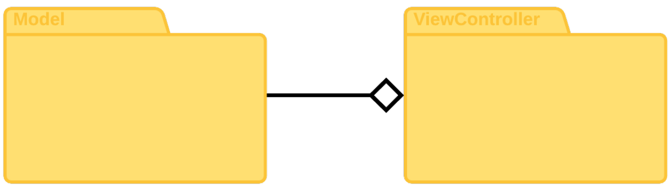

Figure 2: The application's package division.

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 3.

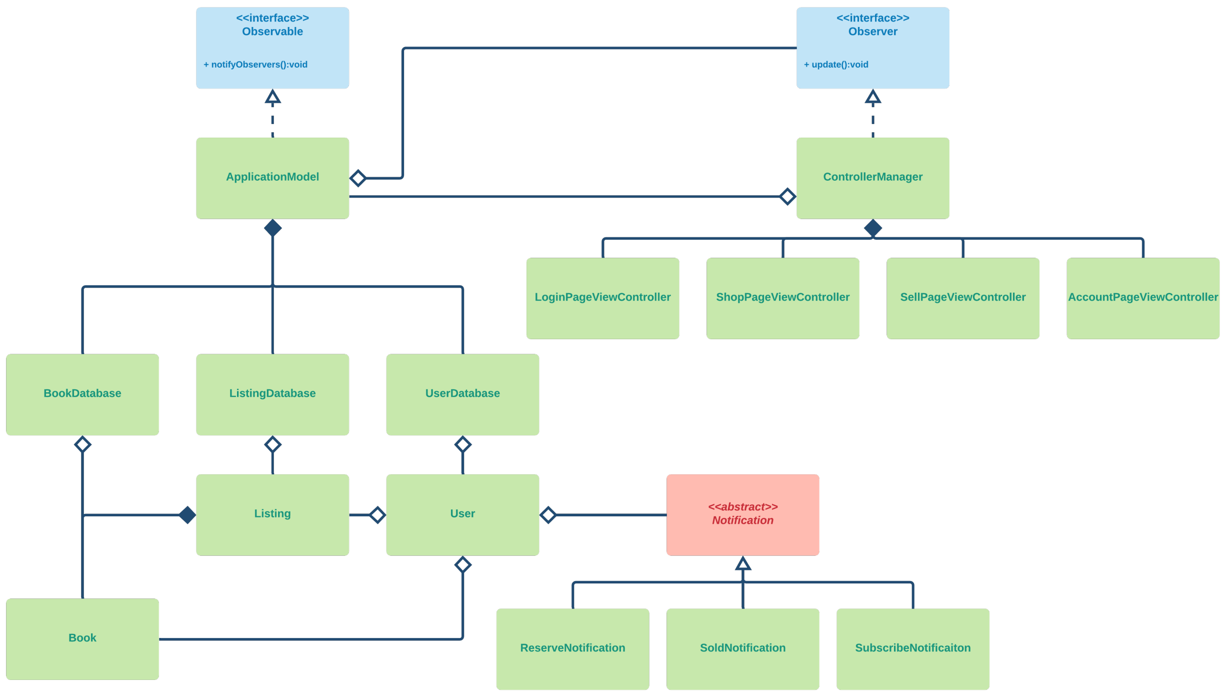

Figure 3: The application's design UML diagram

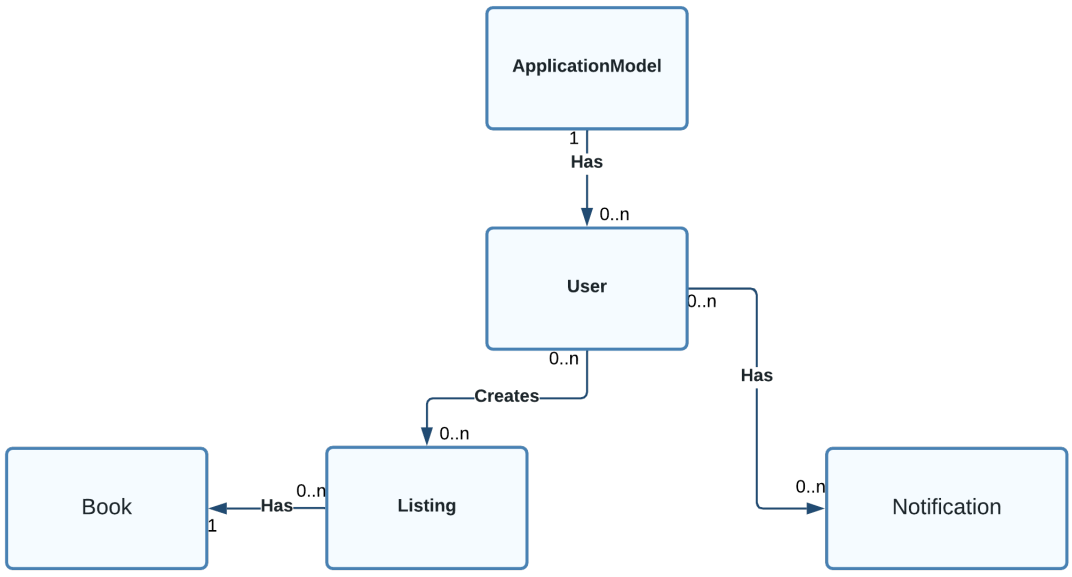

Figure 4: The application’s domain model.

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.

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.

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.

5.1 Access control and security

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

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.

6 References

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

6.1 IDE

JetBrains IntelliJ IDEA

6.2 Build Tools

Maven 3.8.1

6.2.1 Imported libraries

- JUnit5
- JavaFX 16

6.2.2 Structure analysis

JDepend 2.10

6.2.3 Quality analysis

- PMD 6.39.0
- SpotBugs 4.4.1

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