System Design Document for Min5a

Naomi Espinosa, Robin Hallabro-Kokko, Theodor Khademi, Hanna Tärnåsen
& Johanna Wiberg

Date: 25/10
Version: 1

1. Introduction

Min5a is a help application for primarily university students that gives its users access to tools and statistics regarding their studies. The project employs the design pattern “Model-View-Controller” as its structure, using JavaFX as its visual base. The project uses Gradle for build-automation and a continuous integration system with the help of Travis.

The overall goal of this application was to give the user a platform to access a collection of all relevant information about their studies, which currently is located in many different providers. The user is given the liberty to freely add courses, to-do’s, deadlines and contacts. Furthermore, the application features a timer that implements a Pomodoro study technique. The timer can be used on its own but can also be connected to a course, where it calculates the total amount of time a user has studied. This information is stored in Statistics page where the user can see how many hours and minutes they have spent studying on each of their courses. Additionally, the page also displays the average time a user has spent to achieve a certain grade in the course.

In this document mainly four things will be discussed. **The Systems Architecture,** which describes how the user flows through the application and how the application works on a top-level. **The Systems Design**, which will present the applications design model through a UML diagram of main components and describe design patterns used. It will also relate the design model to the domain model. The **Persistent Data Management** which describes how the application saves data. Finally the **Quality,** which describes how the application is tested and lists any remaining known issues.

1.1 Definition, acronyms and abbreviations

User story: A user story is a short description of a feature a user or customer wants. It is written in the perspective of the customer and starts with: “As a user I want...”.

Pomodoro: Pomodoro is a studying technique that makes your work more effective. It’s made up from a 25 minute studying interval and a five minute break.

Code coverage: Code coverage is a measurement on how many lines, methods or blocks is tested in an application. Meaning if every line in an application is tested the code coverage is 100%.

JavaDoc: JavaDoc is a documentation generator. You use it to document your code so other programmers can easily understand what you have written.

GUI: An abbreviation that stands for **G**raphical **U**ser **I**nterface and points to the part of the application that the user sees while interacting with the application.

Hash: Can be used to describe some form of cypher of a plaintext message or data(such as a password string), can also describe as the action of creating a cypher from a plaintext.

UML: Stands for **U**nified **M**odeling **L**anguage and is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the software system.

Continuous Integration (CI): A development practice that requires developers to integrate code into a shared repository. Each check-in is then verified by an automated build, allowing teams to detect problems early.

Build Automation: The process of automating the creation of a software build and the associated processes including: compiling computer source code into binary code, packaging binary code, and running automated tests.

2. System Architecture

The applications model is implemented through an aggregate class named “Min5a”. This class holds all the users that are created inside the application using a HashMap. These users are unique and hold specific data for that user and that user only. For example what courses they have, how many hours they have studied, what contacts they have etc. are saved to the specific user when they are logged in to the application. The users themselves are saved using a json-file and accessed when the user has correctly typed in their username and password at the applications login screen.

The project uses a well separated modular design to improve maintainability of the code and ensure that the separate modules can easily be changed into another of similar characteristics. MVC is the pattern we used to separate and define the modules. This would mean that the **M**odel-part of our MVC can be reused in other projects without any issues. **M**odel is responsible for all the domain-specific data and business logic. **V**iew is the graphics the user can see and interact with and they are fxml files. Lastly the **C**ontroller module connects the model to the view and make sure the right things happen when the users interact with the GUI. In order for the MVC to be implemented correctly and to ensure the modularity and maintainability of the code, the model does not have any dependencies on the view or controller. The controllers are dependent on the model and the view is dependent on the controllers. The view however is also dependent on the model, to display updated graphics as the model changes. To do this Events gets sent out when some specific change in the model has been done, which in turn will call for some specific update of the GUI via a method that subscribes to the event.

The flow of the application proceeds like this:

When the user starts the application it is presented with the mainpage. Here, the user can login with an existing account, or create a new account.

When the user is logged in the first thing the user sees is the mainpage. Here the user has easy access to its active courses or add new ones as well as editing its account settings. You can also find a side panel to the left. This panel can be reached from every page of the application to make sure that you don't get lost.

Through the side panel the user can access all of the pages of the application, but most likely the user wants to see a specific courses information. To reach a “course” page the user can click on one of the courses in “aktiva kurser” or click on the “kurser” button in the side panel to get to the course selection page.

On the “course” selection page the user sees a list of active courses and a list of inactive courses that they can choose to inspect further. On this page the user can also create new courses if they want.

On the specific course page the user can keep track of or add/remove their assignments and to-do items for that course and see their latest activities. From this page is it possible to edit the “course” name, “course” code, study period and study year.

From here the user might press the “Starta Timer” button to navigate to the timer page. Here they can start a new study session of 25 minutes and a 5 minute break.

After a study session, the user might want to see how many hours in total they have spent studying for this course. This can be accessed in the “Statistik” page.

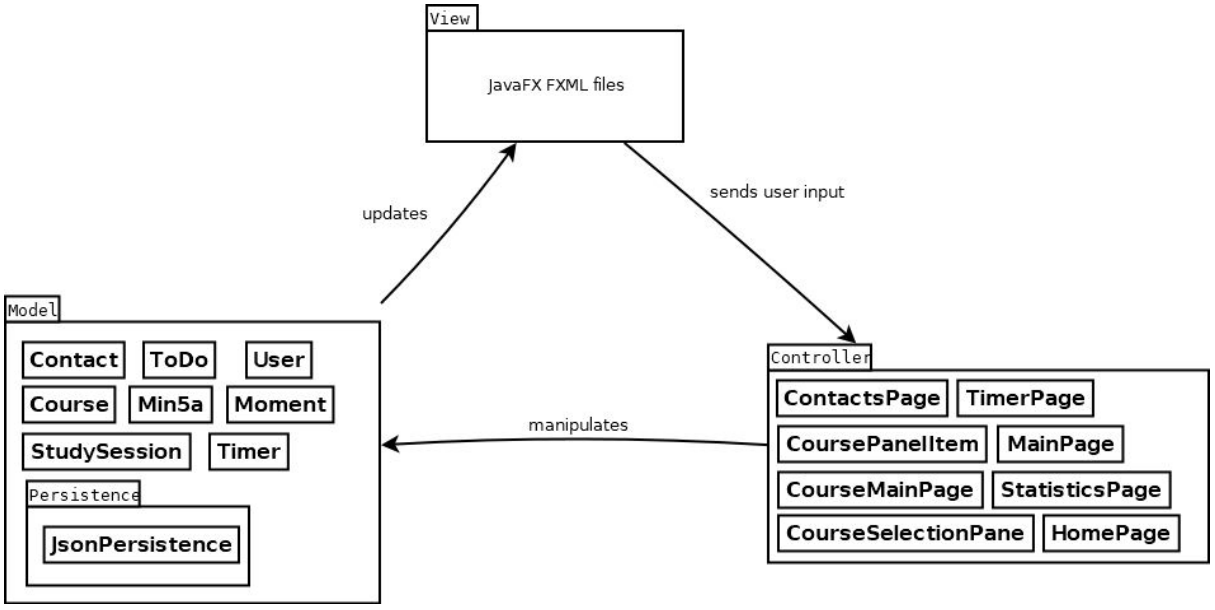
At any time the user may want to add a contact for a specific course or just another student. This is done the “Kontakter” page.

From there the user might want to continue using the timer for studying or exit the application.

3. System design

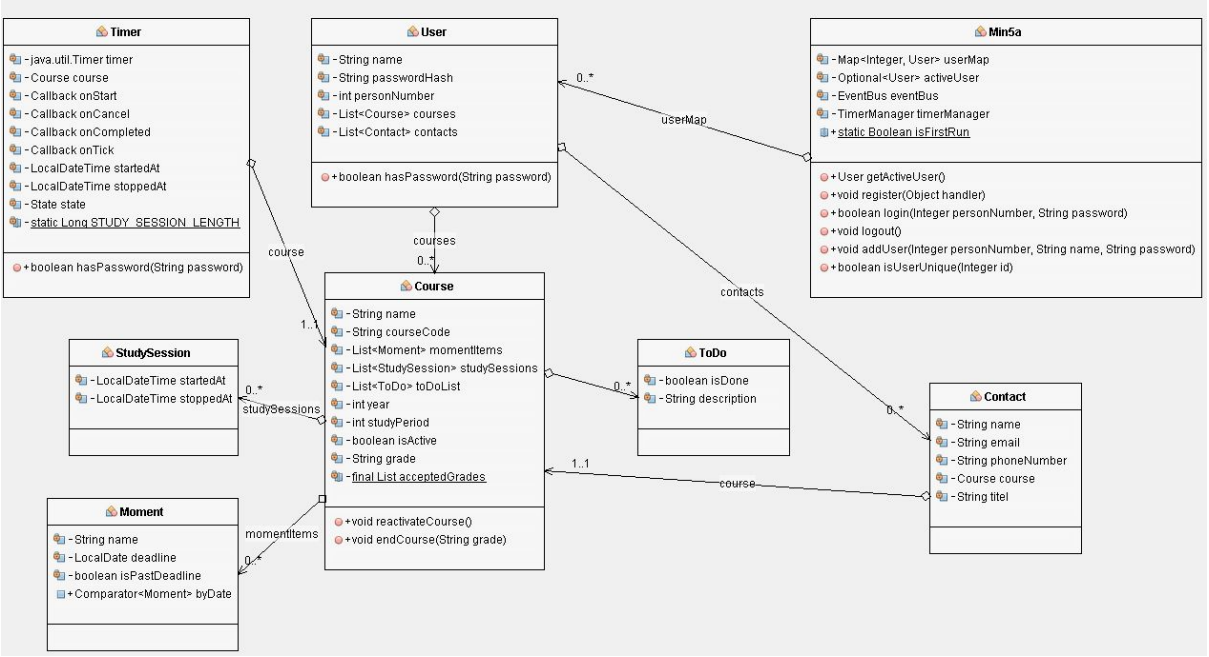
3.1 UML diagrams

As the application uses the JavaFX framework the view layer is made up of FXML files and is read by the controllers. The model represents the business logic layer, including persistence.

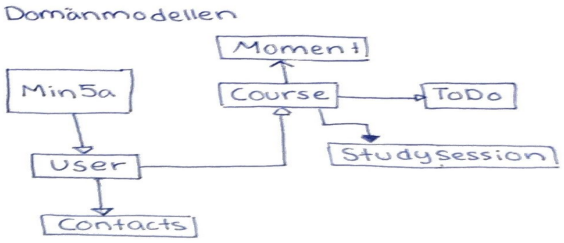


3.1.1 Design model

The central part of the application is the Course class. It is user specific and may not be shared with other users. The aggregate class `Min5a` holds references and manages most aspects of our model, such as the currently logged in user (if any).



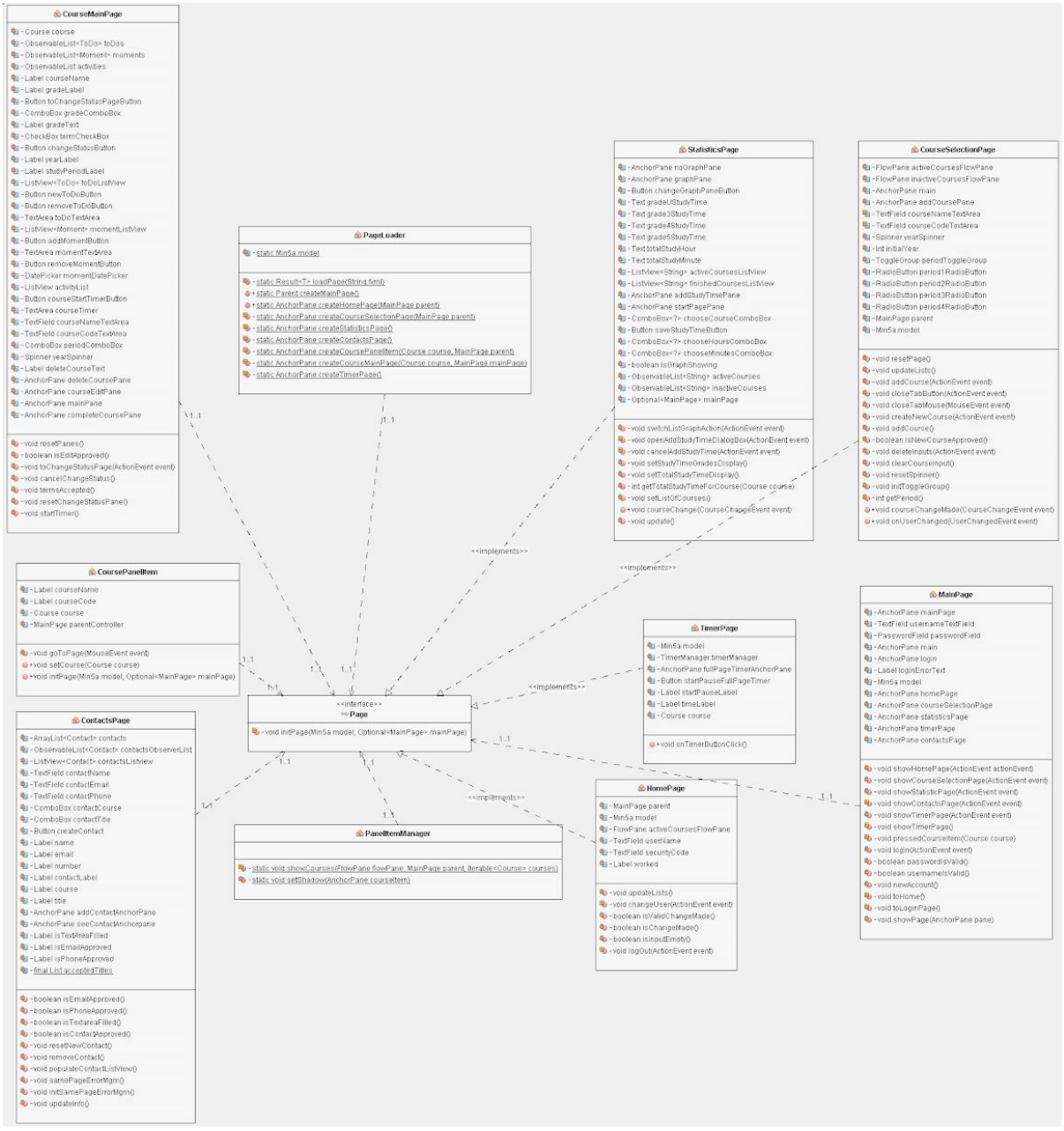
The domain model:



The design and domain model likeness for the most part holds true, the only component that was not present in the domain model was the timer class.

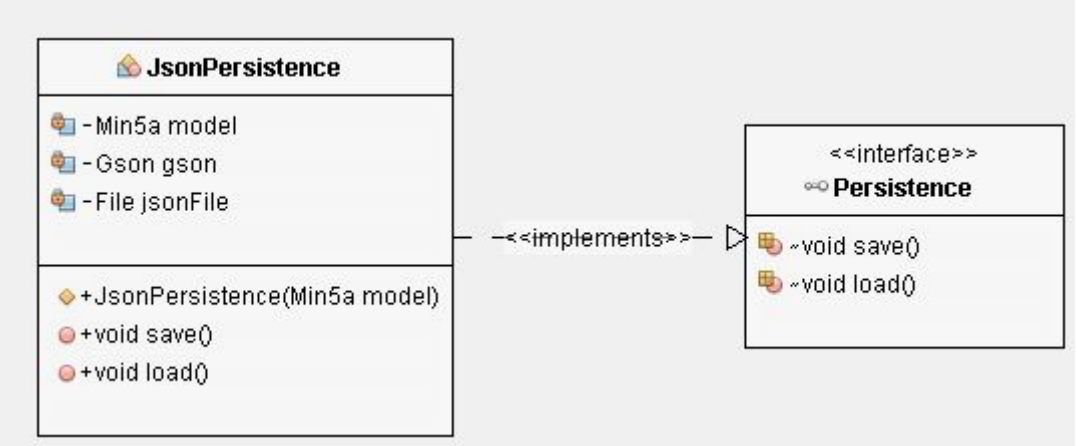
3.1.2 ViewController

The ViewController package consists of all pages of the application. All pages implement a basic interface called `Page`.



3.1.3 Persistence

The persistence package exposes a simple interface for saving and loading data. The current implementation is done using JSON serialization but may in the future contain other implementations such as a SQL backed database.



3.2 Design patterns and principles

- Interface segregation principle
  Allows us to reduce coupling by not depending upon concrete classes.

- Single responsibility principle
  The classes have been adapted to only hold one responsibility.

- Separation of Concern
  This principle has been adopted since the use of MVC and that the persistence module.

- Observer pattern
  Used by many components in our view to notify user of changes such as timer completion.

- State pattern
  Used by the Timer class to store timer state such as paused, active and completed.

- MVC
  An overall structure in the project where it is divided into model, view and (view)controller.

- Factory Method
  PageLoader is a factory in the sense that it creates all pages and their controllers.

4. Persistent data management

As the application is user-specific, and all data that a user accumulates while using the app is saved under the user object, most of the data in the application is persistent data. Every user has a list of courses and contacts and under those categories further collections of data, thus a system to store this persistent data was put in place. The application uses JSON, which is an open-standard file format that saves objects of data as regular text strings in a JSON file. By using JSON we implemented a simple but efficient way to store the data of objects in a file which we could also easily access and read manually if need be.

The other form of persistent data in the app are icons and images used in the GUI. These are saved in a resources folder within the project structure so that every user can see the GUI as it is supposed to look.

The data is not saved continually during usage, but instead we chose to save the user instance as the application is closed. The reason for this being that during app use there are not many openings where you might accidentally delete/lose important data. We have also tried to design the application to eliminate the risk of accidentally losing unsaved data. Of course, the app could be improved by saving data more often and thus offering the user more reassurance for their safety.

5. Quality

The application has several ground pillars that we rely on to assure the quality of our project.

- Build automation: Gradle was used to make working on the project more automated and eliminate a lot of the risk factors associated with human error.

- Code style: We implemented the Google Java Style Guide and configured Gradle to allow for automatic reformatting of the code base. An enforced common code style makes it easier for developers to navigate the code.

- Testing: JUnit is the de-facto standard of testing in a Java environment and is also our framework of choice for testing. Most of our tests are unit tests and some of our tests are functional tests. We aimed at testing most of our model while not testing more complex parts of the application such as the user interface. The overall code coverage was 80%.

- Continuous integration: To ensure a high quality of code in our master branch we execute a code style compliance check and execute the entire test suite upon every new commit to our master branch - and upon each newly submitted pull request on GitHub. This is done using the online service Travis CI.

  The current build status is found at https://travis-ci.org/naomiespinosa/TDA367project/builds.

- Static code analysis has been done using PMD showing a total of 183 warnings. The report may be found at: https://drive.google.com/open?id=1bVg6NrdWjiJMTTGw_vSQGsLKDmsHEC7o

5.1 Known issues

1. Relations between entities (for example between contacts and courses) are known to cause cascading deletes which is not the intended behavior. This may be because the application data is persisted in a JSON-serialized file which does not support relations.
2. After creating a contact, a page with the contact that was just created is supposed to show up but right now that page is empty.
3. The program is not suited for Swedish characters "Å", "Ä" and "Ö" and can not be shown properly.
4. The CSS file produces many warnings on start because it recognize certain variables.

5.2 Access control and security

The application currently has one form of account and that is the standard "customer" login account which enables all the functionality the application contains. Access is regulated by a login feature. The user can create an account with the date of birth in combination to their social security number as well as a password.

For extra security the password is not saved in plaintext but instead it is saved as a hashed version of the password - unreadable and generally very difficult to recover the original content from. The password is also concealed when typing in the password text fields in accordance to accepted practice. All the password hashes are saved in the JSON file which also contains the other user data.

The chosen hashing algorithm MD5 is known to be weak and is easily cracked. Future iterations of this applications should seek to implement a more secure way of hashing password.

6. References

Gradle : https://gradle.org/

Junit: https://junit.org/junit5/

Guava EventBus: https://github.com/google/guava/wiki/EventBusExplained

Google Java Format: https://google.github.io/styleguide/javaguide.html

GitHub: https://github.com/

Travis CI: https://travis-ci.org/

DIA UML Tool: http://dia-installer.de/