YELLOW

System design document (SDD)

Joakim Agnemyr, Edwin Eliasson, Mona Kilsgård and Viktor Valadi.

TDA 367

2018-10-21

Version 2

Table of Contents

1. Introduction

This document describes a desktop application in which a user can be a part of groups with several users. In every group it is possible to share inventories and rent out items to customers. Yellow should be an easily extendable application used for inventory and renting. The goal is to have a model that is not coupled at all with the graphical view and only loosely coupled with the controller. This is to make it easy to use the model with other graphical libraries.

1.1 Definitions, acronyms and abbreviations.

- **GUI** - Graphical User Interface.
- **User** - The person that uses the application.
- **Item** - Items that can be rented and added to inventories.
- **Inventory** - The inventory items are placed in.
- **Group** - Groups users take part in and sharing inventories.
- **Order** - A order put on an inventory to rent out one or more items for a period of time.
- **Renter** - The person renting the items.
- **MVC** - Stands for Model-View-Controller. **MVC** is an application design model comprised of three interconnected parts. They include the model (data), the view (user interface), and the controller (processes that handle input).
- **Interface** - A interface class connected to the model to reduce dependencies.
- **Dependency inversion principle** - A design principle where you use a interface to reduce coupling between packages.
- **Java** - The object oriented programming language we use for developing the application.
- **Encapsulation** - Hiding properties in objects from the “outside” to protect from manipulation and to bundle related methods
- **Single Responsibility principle** - Is a **principle** that states that every module or class should have **responsibility** over a **single** part of the functionality.
- **Coupled** - Refers to how much for example a class know about the other class. Tight coupling means that if class A knows more than it should about class B, they often change together.
- **Polymorphism** - When you can reference a parent class instead of the child class. For example, a boat or car are both classes that could extend a vehicle class.
- **Dependency inversion principle** - Is a principle that decouples modules or classes. Makes the application more abstract.
- **Encapsulation** - By keeping variables private you prevent other classes to get access to them.
- **JFoenix** - A graphic library which can be used with Scene Builder.
- **Gradle** - A build tool for Java.
- **JavaFX** - A set of graphics and media packages.

2. System architecture

The application will be written in Java. MVC will be implemented. Our GUI will be designed in Scenebuilder and FXML-files.

The application is divided into following packages:

- Model - Where the data is stored when the application is running, for example Users and Groups.
- Controller - Handles input from the user and sends commands to the model.
- Resources - Stores all the different views, icons and images. It is also here that Yellow stores information in a local “server”.
- View - Has all the ViewHandlers which connects the .fxml files to the application.

2.1 Subsystem decomposition

The system uses the MVC-pattern which means that the model takes care of the logic of the data, the view represents the data to the user and the controller serve as a middleman between the view and model to remove dependencies between the two. This will make it easier to replace the graphical representation of the data in the future.

2.1.1 Model

The model will take care of all the logic and manipulation of data. The model gets stored data from the controller package which it uses when the app is running.

The model consists of:

- **User:** Represents the user. Holds information such as username and password.
- **Group:** Represents the groups users can be a part of.
- **Inventory:** Represents the inventory which the users can put items in.
- **Item:** Represents an item the user can put into a selected inventory.
- **Order:** The order in which the user can see who rented what etc.
- **Renter:** Represents the person who wants to rent something.
- **YellowHandler:** Works as a connection between the controller and the model and delegates the specific tasks to the correct class.

  The model is represented by the design model (see figure 1).

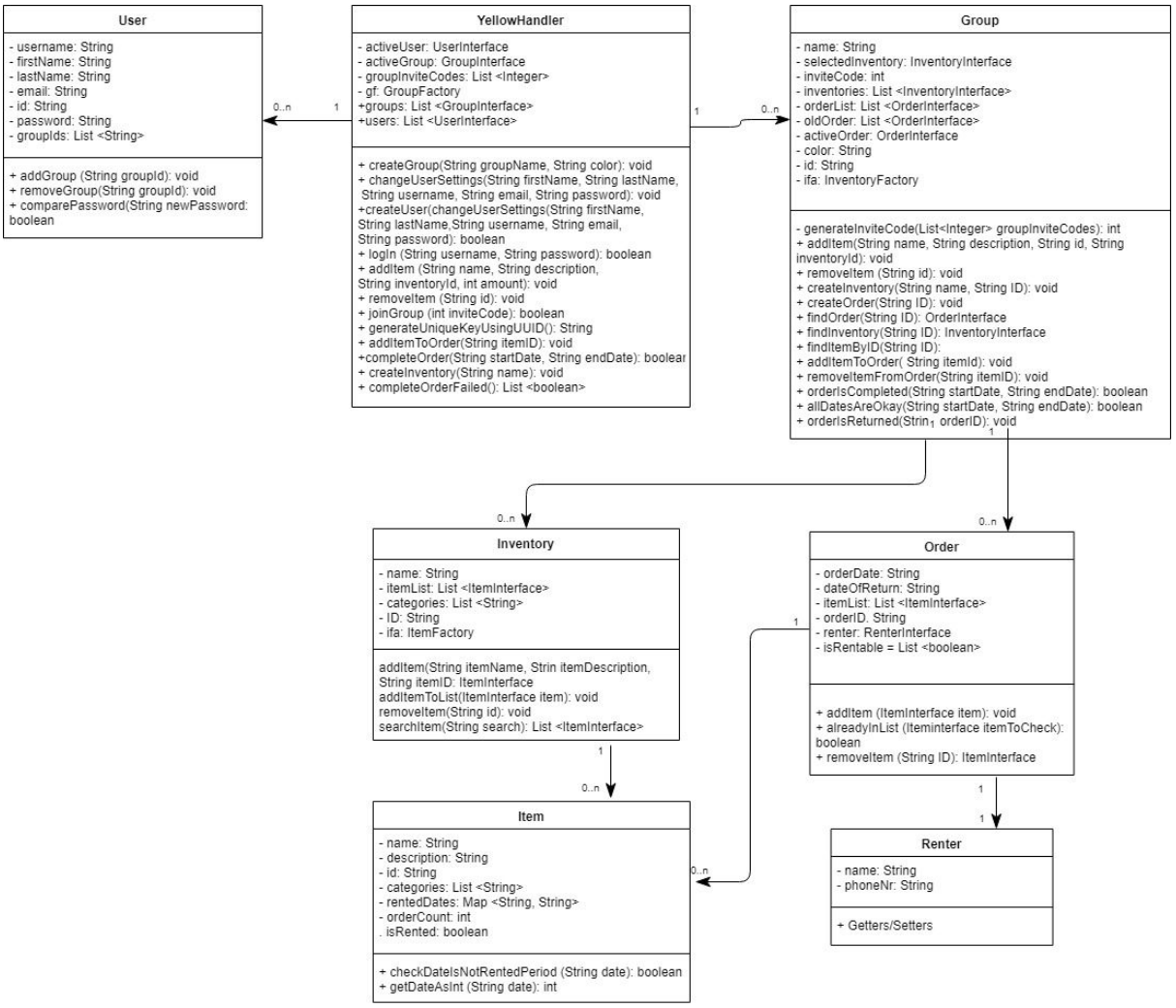

Figure 1: Design Model

Following the Single Responsibility principle all classes in the model are supposed to handle functionality referring to their specific tasks. To get the benefits of polymorphism the classes in the model implements the dependency inversion principle through implementing their respective interfaces. The dependency inversion principle helps the project being more abstract and remove unnecessary dependencies. An example of the dependency inversion pattern can be seen in figure 2.

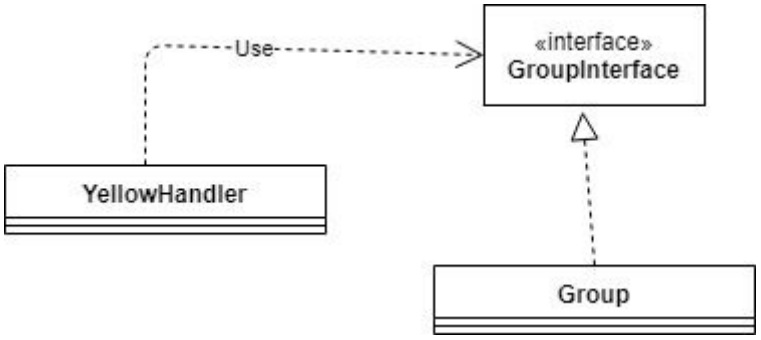

Figure 2: Dependency Inversion Principle

To fully implement this a factory pattern is also implemented to remove dependencies when for example YellowHandler will create a new group (see figure 2).

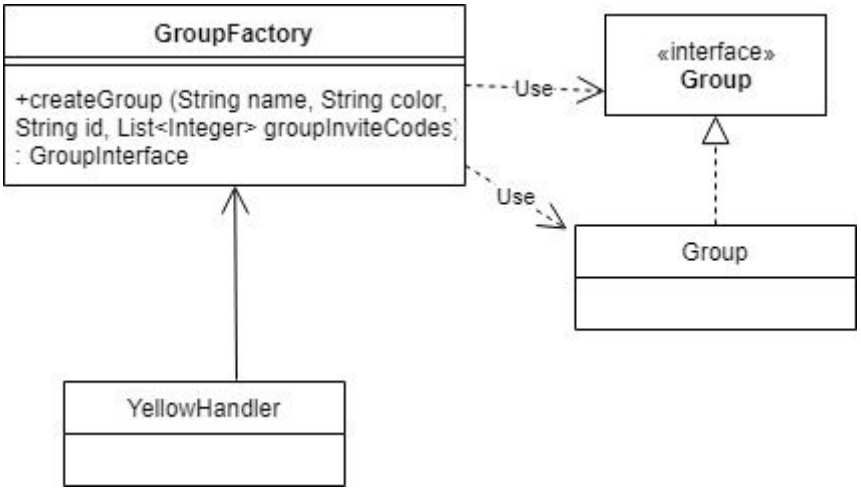

Figure 3: Factory Pattern

The goal is to hide as much data as possible in the application. Encapsulation helps preventing modification of data that should not be manipulated from outside the program. In the long run encapsulation will also help make the application easier to expand and change.

The model has a class called YellowHandler which binds the model together.
When the controller wants data from the model YellowHandler will start delegating the task to the correct class (see figure 4) This design also decreases the controllers dependency on the model.

YellowHandler is a part of the observer pattern in the application, which means that when YellowHandler is doing something with the data, it will notify all the observers which in our case is the viewhandlers. That is why YellowHandler extends the class Observable. The view can get data from YellowHandler.

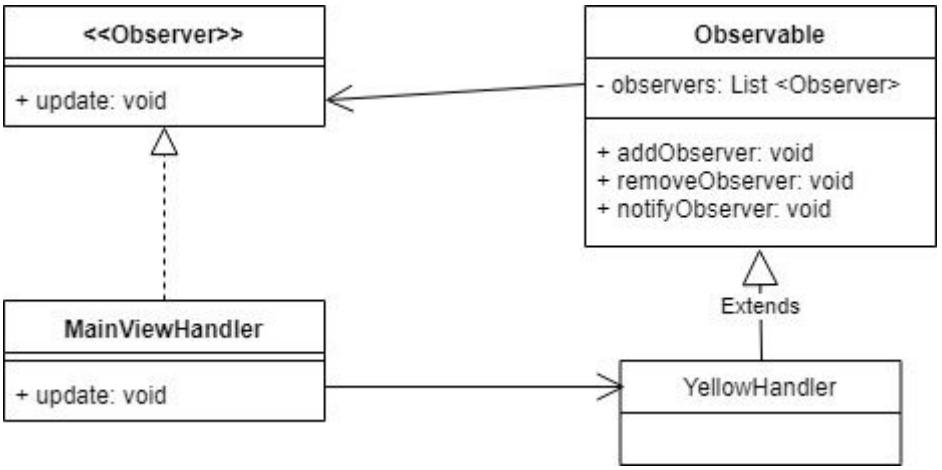

Figure 4: Observer Pattern

2.1.2 View

The layout is done with Scene Builder and the JFoenix library, and are saved in .fxml files in the resources directory. The reason for this is because of gradles tree structure, and the .fxml should be seen as “images” which are connected to their respective view controller. In the view package there is a viewhandler for each view in the application. The handlers works as a sort of controller for the different .fxml files, and collects data from the model. When data is updated, the viewhandlers will be notified by the model that there has been a change, and the view controller shows the new or modified data. The controller needs to be notified by the view when an action has happened. Since the view should not know about the controller, some buttons sends out an event to everyone that listens to it.

2.1.3 Controller

The application’s main class works as a controller. The controller listens to different buttons in the view, and tells the model to do something when they are clicked. This is how the view communicates to the model. The controller manages. The controller loads all the .fxml files which instantiates the viewhandlers. The connections between the model, view and controller can be seen in figure 5. The controller also loads and saves data when the application starts and stops.

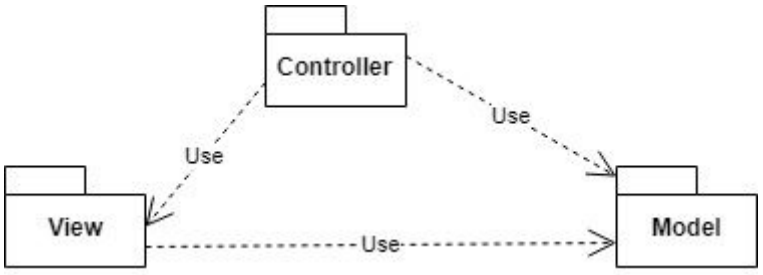

Figure 5: Package dependencies

2.1.4 Testing

The application has an extensive test package located in the src map that tests every relevant function in the model. Every class has a corresponding test class made for testing its functionality. The controller also has tests that checks if the save and load function is working. The rest of the methods in the controller are tested by the GUI itself. The class that has most of the tests is the YellowHandler class. Since the YellowHandler handles the functionality of the model it test methods through the entire model from here and test that everything comes together.

3. Persistent data management

3.1 Saving data

When the application starts the controller will send all the data to the model. This is because we want the model to handle the management of data, not store the data itself and also to remove dependencies from the model to the controller. When the application exits all that data is sent back from the model to the controller. The controller will then run a method that saves the lists of groups and users encoded into .ser documents.

The saving and loading is done through javas own mechanism called serialization. We serialize whole objects into a document that later can be deserialized to get a copy of the object back. The serialized object is represented through a sequences of bytes representing everything contained in that object.

3.2 Images and Icons

All images and Icons are saved in a directory named Img in the resources directory.