Final report for Yellow

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

TDA 367

2018-10-28

This version overrides all previous versions.

Table of Contents

1. Introduction

1.1 Purpose of application

The aim of this project is to create an inventory application where different groups of people can lend out and keep track of the items in their common inventory. The application will allow the user to know to who, when and during what period the item(s) were lended.

Yellow should be an easily extendable application used for inventory and renting. The goal is to have a model that is not dependent at all of the rest of the project. This is to make it easy to use the model with other graphical libraries.

1.2 General characteristics of application

The application will be a desktop application runnable on Mac/Windows/Linux platforms.

The application's purpose is to give members of different societies the possibility to keep track of their inventories and lend out items in these inventories. The users should be able to rent out anything they want by creating their own items. The user has the possibility to give the item a name, a description, a certain amount and add an image. When a customer wants to rent an item the user will be able to find the item in their inventory and rent it out. When an item is available for renting an order is created to give the user an overview of the renting, and input the renters personal info. When the order is created all users who shares the inventory can see the order and the items will be marked as rented during the picked date.

1.3 Scope of application

The application will not support rentals made by private individuals. The rental procedure can only be done by people associated with the group that owns the specific inventory. The renter therefore does not need to create an account and the entire lease is handled by the members of the group. However, this could be a development area for the application. By allowing users to complete their booking themselves, they reduce the workload for the group. But at the present time, the group gets a good opportunity to keep track of their inventory without incorrect and unwanted rentals made by renters.

1.4 Objectives and success criteria of the project

1. It should be possible to create an account.
2. It should be possible to create a group.
3. It should be possible to add items to an inventory.
4. It should be possible to view all the items in the inventory.
5. Users should be able to join different groups by entering a code that is generated when a user creates a group.
6. It should be possible to be part of multiple groups at the same time.
7. The users should be able to see what they have in "stock" and what is rented.

8. It should be possible to edit user information in the application.
9. It should be possible to create orders where all the items that the renter wants to borrow are gathered.
10. It should be possible to add who rents the items while placing an order.
11. It should be possible to view when a item is rented.

1.5 Definitions, acronyms, and abbreviations

- **Committee** - A group of people.
- **Yellow** - The name of this application
- **Student division** - Referring to a group of students with a common inventory.
- **App** - Referring to an application usable on desktops.
- **Renter** - The person who lend items from the group who owns the inventory to which the object belongs.
- **Group -** A group in the app that users can be a part of and where they can share inventories.
- **User Story** - Describes a feature that a user or customer of the system desires.
- **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.
- **Order** - An order put on an inventory to rent out one or more items for a period of time.
- **MVC** - Stands for Model-View-Controller. The program is separated into three different parts, the view which represents the user interface, the controller which handles data input, and the model that manipulates and controls data [1].
- **Interface** - Consist of methods without bodies. Can be used to describe behaviours which classes can implement.
- **Dependency inversion principle** - A design principle where you use an interface to reduce coupling between packages [2].
- **Java** - The object oriented programming language we use for developing the application.
- **Single Responsibility principle** - A design principle that states that classes and modules should have responsibility over a single part of the functionality [2].
- **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 [3].
- **Cohesion** - High cohesion means that a class has a well-focused purpose and should not be changed often [3].
- **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 [4].
- **Encapsulation** - By keeping variables private you prevent other classes to get access to them [5].
- **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.
- **S.O.L.I.D** - A set of principles which are used for programming which includes single responsibility principle, open-closed principle, liskov substitution principle, interface segregation principle and dependency inversion principle. These principles are essential for object-oriented programming [2].

- **Separation of concerns** - A design principle which separates a program into different separate sections [6].

2. Requirements

2.1 User stories

Listed below are the different user stories for Yellow. All user stories are identifiable by a unique story id and with an explanatory story name.

1. **Story Identifier:** U101
   Story Name: Creating a inventory
   Description: As a division member I want to be able to create an inventory where I can place items.

   Confirmation
   - It should be possible to create an inventory with a name.

   Functional
   - Can I create an inventory?
   - Can I add a name for my inventory?

   Non-functional
   - Is it possible to create an inventory when not in a group?

2. **Story Identifier:** U102
   Story Name: See an inventory
   Description: As a division member I want to see an inventory with my items because I want to know what items I have.

   Confirmation
   - It should be possible to see a list of items from a specific inventory.
   - It should be possible to see a list of all items in a group with two or more inventories.

   Functional
   - Can the person see all items from their inventories?
   - Can the person see the items for a specific inventory?

3. **Story Identifier:** U103
   Story Name: Create items
   Description: As a division member I want to be able to create items while adding my collection of items.

Confirmation

- It should be possible to create an item in my inventory.
- It should be possible to add a name, a short description for the item and enter an amount.
- It should be possible to add an image for the specific item.

Functional

- Can I create an item when in a group?
- Can I create an item with a selected inventory?
- Can I create an item with all parameters filled?

Non-functional

- Is it possible to create an item if not in a group?
- Is it possible to create an item if no inventory is selected ("All items" counts as an inventory).

4. **Story Identifier:** U104

Story Name: Expanding inventory

Description: As a division member I want to be able to expand my inventory because I might want to add items in the future.

Confirmation

- It should be possible to add an item into a selected inventory (already created inventory).
- When creating an item it should be possible to choose which inventory it belongs to.

Functional

- Can I add an item when in a group?
- Can I add an item with selected inventory?
- Can I add an item with all parameters filled?

Non-functional

- Is it possible to add an item if not in a group?
- Is it possible to add an item if no inventory is selected?

5. **Story Identifier:** U105

Story Name: Several inventories

Description: As a division member I want to create different inventories because I might have items in different places.

Confirmation

- It should be possible to add multiple inventories.

Functional

- Is it possible to create an inventory when in a group?
- Is it possible to have more than one inventory?
- Can I name my inventory?

- Does the inventory belong to a group?

6. **Story Identifier:** U106
   Story Name: View inventories
   Description: As a division member I want to see items that belongs to a specific inventory or see all my items at the same time.

   Confirmation
   - It should be possible to see inventories one at the time.
   - It should be possible to see all items in a group at the same time.

   Functional
   - Is it possible to switch between inventories?
   - Is it possible to see all items in a group?

7. **Story Identifier:** U107
   Story Name: Gathered inventories
   Description: As a division member I want to be able to gather my inventories into groups.

   Confirmation
   - The division member should be able to create a group.
   - The created group should have a name and a color.
   - The user should be able to choose which group the inventory should belong to while creating the inventory.

   Functional
   - Can I see the inventories that are in the selected group?
   - Can I see the items that are in the selected inventory?

8. **Story Identifier:** U108
   Story Name: Sharing inventories
   Description: As a division member I want to be able to share my inventory with others because there might be several people sharing an inventory.

   Confirmation
   - The user is able to invite other users to a group that it is a member of.
   - The user should be able to view invite codes to all the groups he/she is a member of.

   Functional
   - Can I view the invite codes to my groups?

9. **Story Identifier:** U109
   Story Name: Joining groups

Description: As a division member I want to be able join an already existing group to be a part of that inventory.

Confirmation

- The user should be able to join a group by being invited by a member of that specific group.
- It should be possible to insert the invite code given by another user.
- It should be possible to view all inventories and items specified in the specific group.
- The group should be listed with the users other groups.
- The division member should be able to view the invite code to the specific group after joining it.

Functional

- Can I join a group?
- Can I see the inventories that are in the joined group?
- Can I see the items that are in the selected inventory?
- Can I view the invite code to the group?
- Can I still see my other groups after joining another group?

Non-functional

- Can you join a group without the invite code?

10. **Story identifier:** U110

Story Name: Keeping groups and inventories private
Description: As a group member I don't want non-group members to see the groups inventory because I want to keep it private within my group.

Confirmation:

- The group member only has access to their own groups and inventories.

Functional

- Can I only see the inventories that I am a part of?
- Can only the people in the group see that specific groups invite code?

11. **Story Identifier:** U111

Story Name: Renting items
Description: As a division member I want to rent out item(s) because I want to meet the customers needs.

Confirmation:

- It should be possible to add items to an order.
- It should be possible to delete items from the order before it is confirmed.
- The order should include information about the renter.
- The order should have a starting date and return date.

Functional:

- Can I create an order with multiple items?
- Can I create an order with information about the renter?

12. **Story Identifier:** U112

Story Name: Item renting status

Description: As a division member I want to see what items are rented because I want to know if they are available for renting.

Confirmation

- It should be possible to add a renting time while creating a order.
- It should be possible to view when a item is rented in the item calendar.

Functional

- Does every item have a calendar with that specific items renting status?
- Is the status updated while placing a order?

13. **Story Identifier:** U113

Story Name: Sign in/sign up

Description: As a section member I want to have a account where I can find all my inventories.

Confirmation:

- It should be possible to create a account.
- It should possible to choose a name, username, email and password while creating an account.
- It should possible to sign in by inserting chosen username and password.
- It should possible to log out and sign in with the created account.
- The user should be able to add groups, inventories and items while signed in.

Functional:

- Is the account saved after it is created?
- Is the user obligated to insert their username and password?
- Can there be multiple accounts?
- Can only people with an account sign in?

14. **Story Identifier:** U114

Story Name: User Settings

Description: As a section member I want to change my user information because I might want a new username or password.

Confirmation:

- It should be possible for the user to change their username, name, email and password.

Functional:

- Can I save the settings for the next time I login?

Non-functional:

- Can the user change username to any other users username or password?

2.2 User interface

In general, Yellow has two main purposes. It should be possible to view and edit an inventory, and it should be possible to lend out items and know who has them and when. This notification has provided the basis for how Yellows user interface is designed.

2.2.1 Design structure

The application is mainly structured into two different sections with similar layouts, but with discernible differences.The different sections purposes is to indicate to the user which functions are available. In the first section the user will be able to view and edit groups, inventories and items while the other section handles the rental part of the application. This breakdown is made because a user is not believed to be interested in adding and changing items at the same time as placing a order for a renter.

2.2.2 Design and graphic details

The overall color scheme for Yellow is a darker background where important functionality is enhanced with a yellow color. Yellow also uses two different background images depending on whether the user is logged in or not. This is to clarify whether the user is in the application and can start using it or not. The background image also has a bit brighter color when the user is signed in. This to indicate even more that the user has finished the first step and now can start using the application.

Most labels are written in a white or light grey color to pop out from the background. Therefor labels and texts placed on a brighter background is displayed in a black or dark grey color.

Yellow also uses colour pickers to make it easier for users to distinguish their groups in the list. In addition to this, the application keeps a fairly neutral and easy-to-understand color use where, for example error messages and lended items are highlighted red, and available items green.

Icons are used by Yellow to clarify some features of the application a also to reduce the usage of texts which in large quantities can make the interface look messy.

2.2.3 Layout and flow

When starting up Yellow the user will get two options, either sign in or sign up (*Figure 1: Welcome screen*). These two options are displayed on the same screen to make it easy for the users to know

what is expected of them. If the user chose to sign up, a new, but similar page (*Figure 2: SignUp screen*) will be displayed where the user can create an account by typing in relevant information.

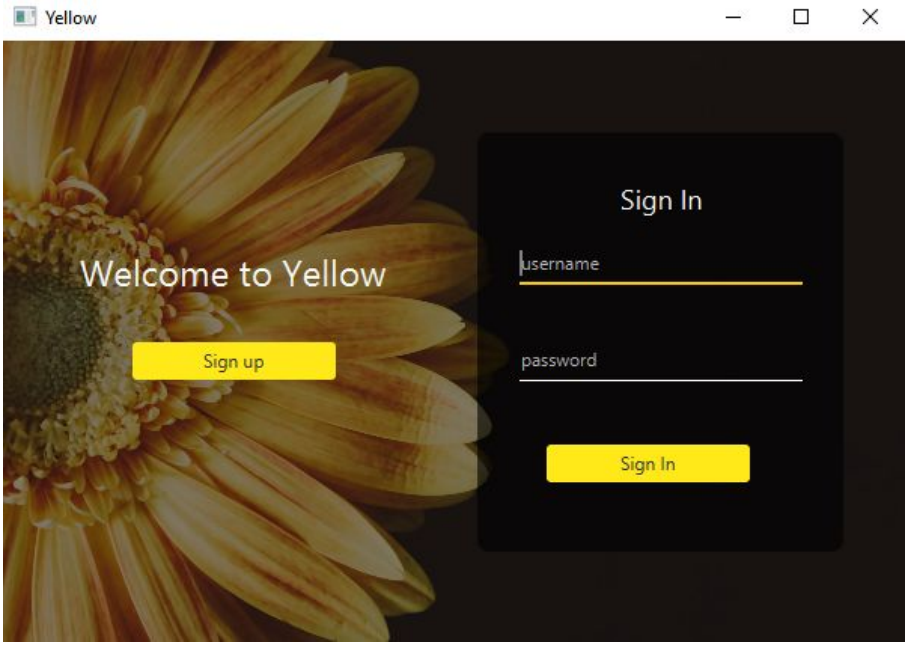


Figure 1 : Welcome screen

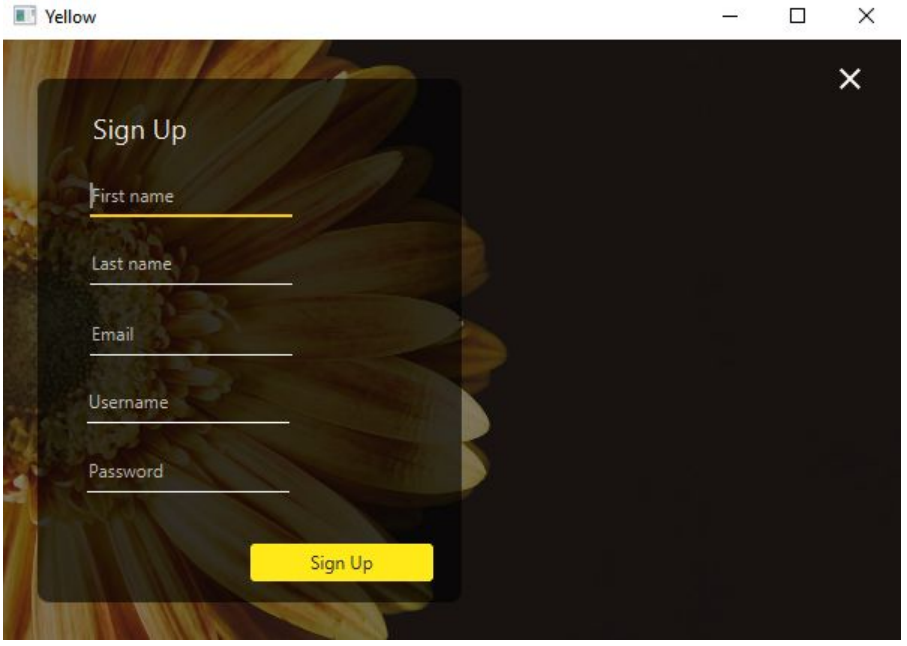


Figure 2: SignUp screen

After the user finished any of these two steps, a new screen with four new options will be displayed (*Figure 3: Yellow screen*). These options are believed to be relevant while first going into the application. By restricting the choices here, Yellow makes it easier for the user to find what they are looking for in the application.

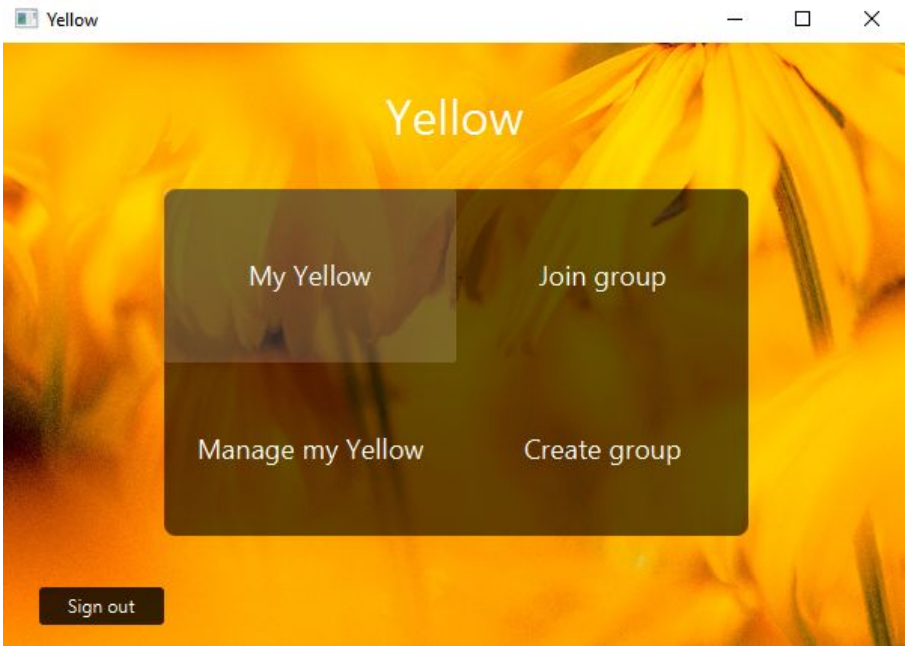


Figure 3: Yellow screen

In most cases the user have been using Yellow before and are already a part of a group. But if that is not the case, the options to either create a group or join a group is available here. After doing any of these two, the user will be taken to the "Manage my Yellow" page (*Figure 3: Manage my Yellow*). This is where you edit existing groups, inventories and items, create new ones or go directly to "My Yellow" (*Figure 9: My Yellow*), which is the part/section of the application where it is possible to lend out items.

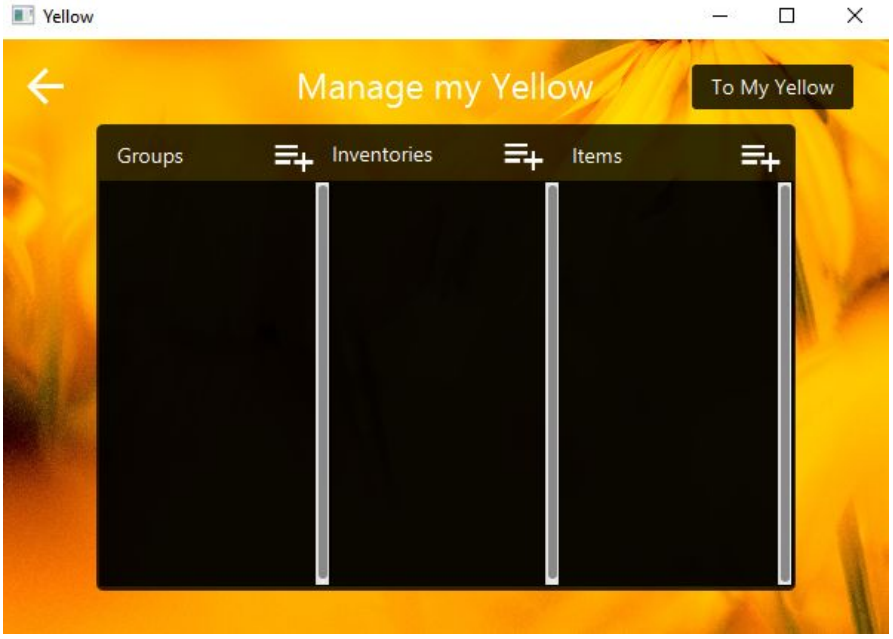


Figure 4: Manage my Yellow

If you are a previous user of Yellow and are familiar with the structure of the application, you probably already know if you want to edit your inventory or lend out items when signing in. Therefore

it is possible for the user to go directly from “Yellow screen” into “My Yellow” or “Manage my Yellow” after signing in (*Figure 3: Yellow screen*).

“Manage my Yellow” (*Figure 4: Manage my Yellow*) is divided into three different flowpanes/lists where groups, inventories and items are listed. The lists is structured in a way that the different inventories is shown based on which group is chosen and items based on chosen inventory. By doing this the user can easily see which inventories and items that belong to which group (*Figure 5: Manage my Yellow with groups, an inventory and a item)*.

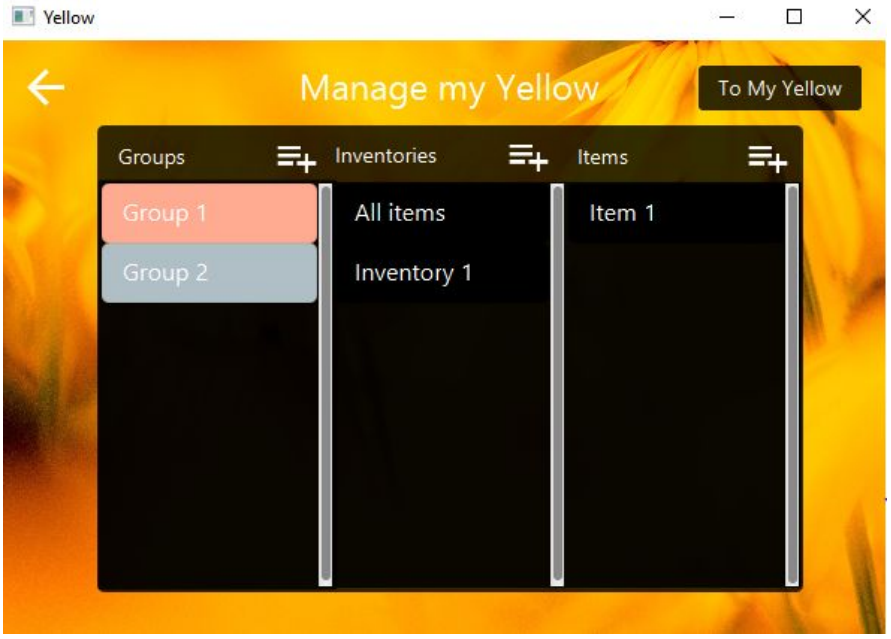


Figure 5: Manage my Yellow with groups, an inventory and a item.

In the “Manage my Yellow” window the user can also add new groups, inventories and items. This is easily done by clicking on the adding symbol placed next to the associated label (*Figure 5: Manage my Yellow with groups, an inventory and a item)*. By this action a new dialog is displayed where the user can insert relevant information. All the dialog screens in Yellow has the same layout which hopefully results in the user recognize himself making it easier to navigate around the application (*Figure 6, Figure 7 & Figure 8*).

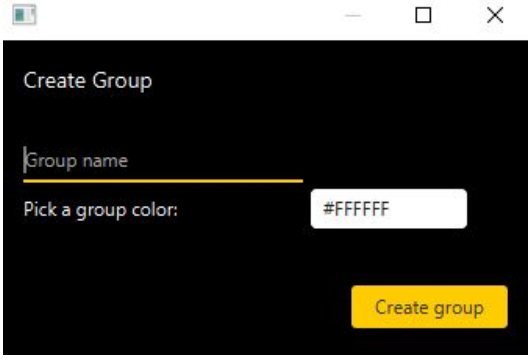


Figure 6: Create group

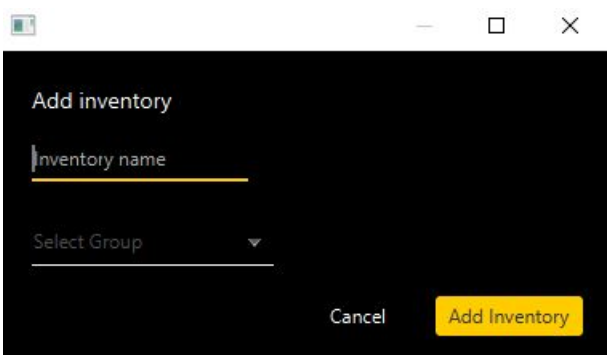


Figure 7: Add inventory

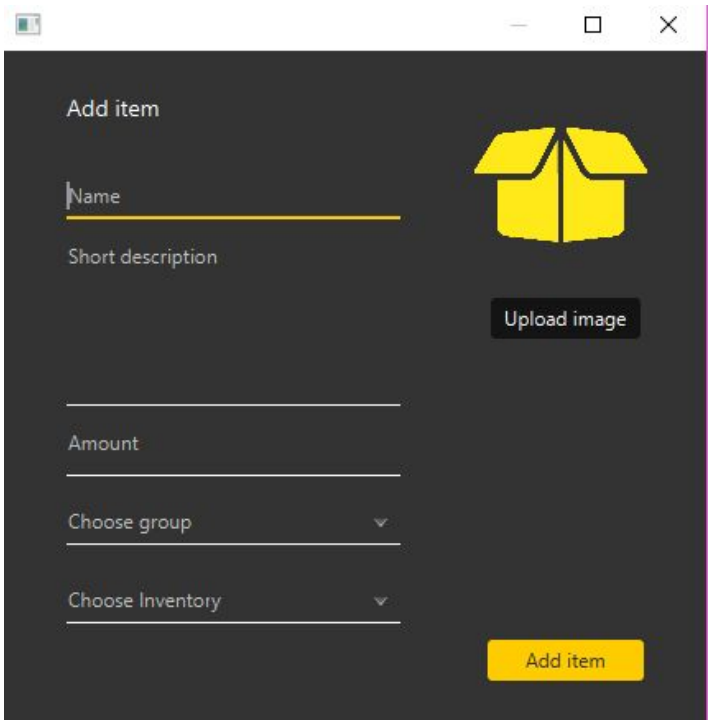


Figure 8: Add item

By choosing a group (by clicking on it) in “Manage my Yellow” an “edit” panel is revealed that takes the user to a almost identical dialog screen as the one that is shown while adding a new group. However, the differences in this view is that the user has the option to change already given information or deleting the object. This structure applies to both groups, inventories and items.

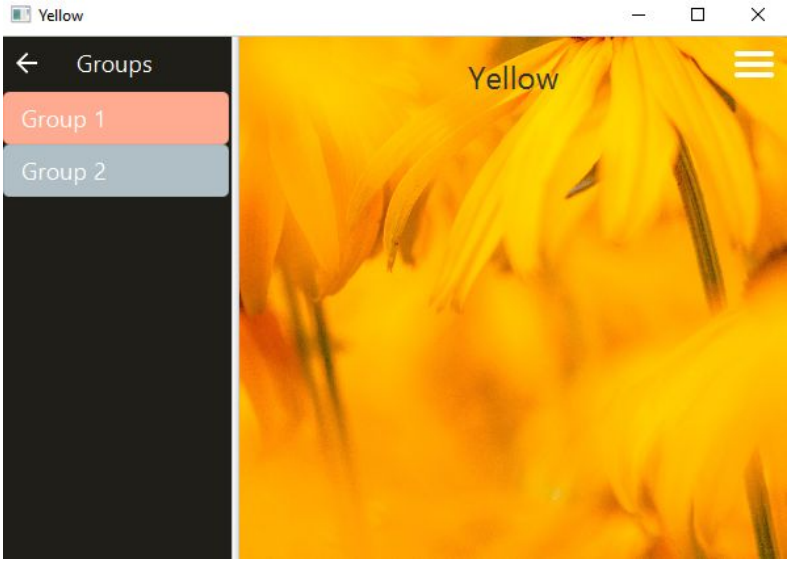


Figure 9: My Yellow

“My Yellow” (*Figure 9*) is the view where it is possible to create orders and view which items that are available. It is structured into two main sections. One list view and one bigger view where things such as items and orders are displayed. The list view has the same structure as “Manage my Yellow” and

by clicking on an available group, that groups inventories will be displayed while its items are shown in the main window.

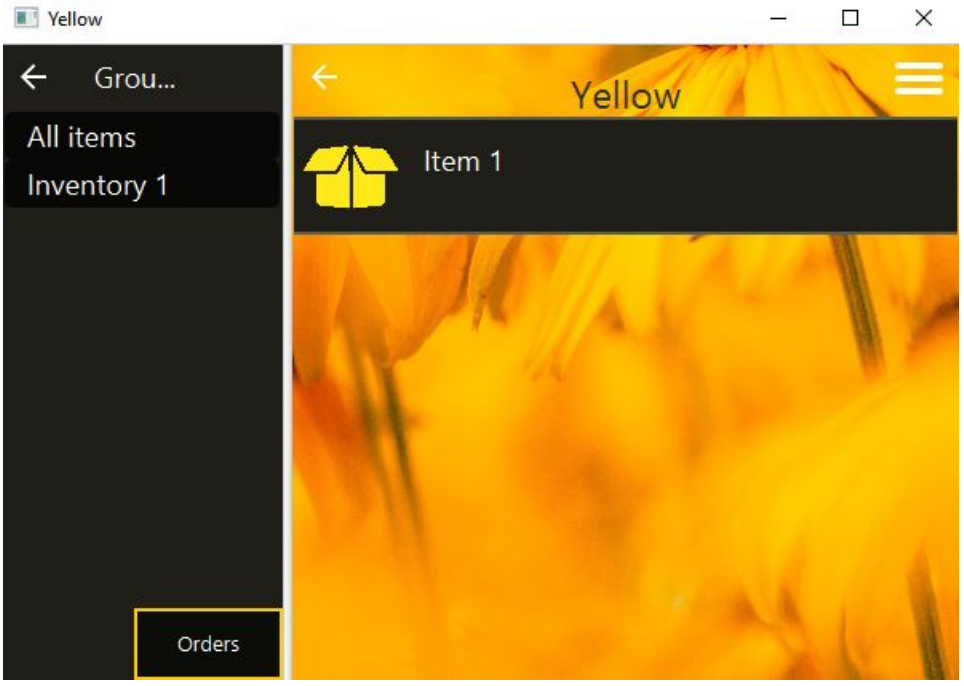


Figure 10: My Yellow → Chosen Group

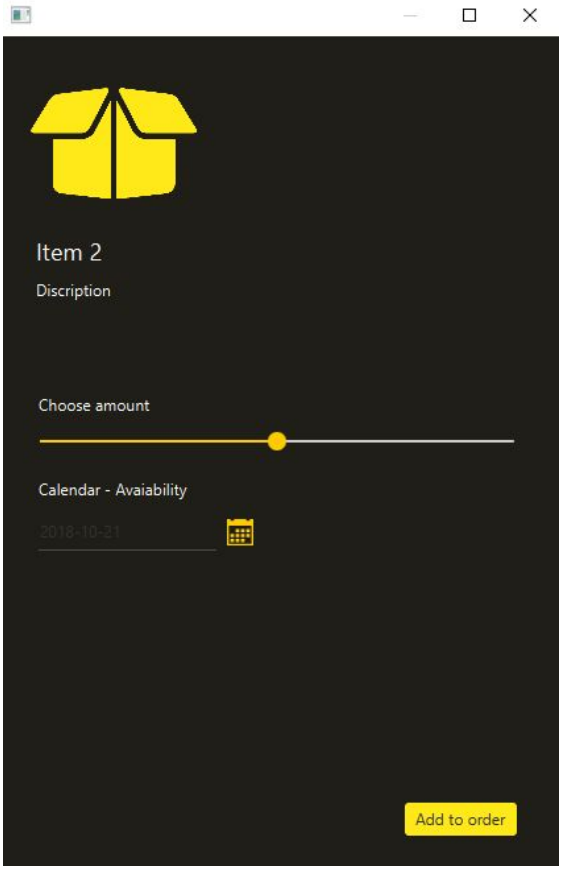


Figure 11: View Item

By clicking on a item in the list the user access the detailed information about that specific item which is shown in a dialog (*Figure 11: View Item*). It is also in this view that it is possible to add a item to the active order and view when, and how many items that is available.

While in the "My Yellow" window and a group is chosen from the list, one new button named Orders will reveal in the left bottom corner (*Figure 10: My Yellow → Chosen Group*).

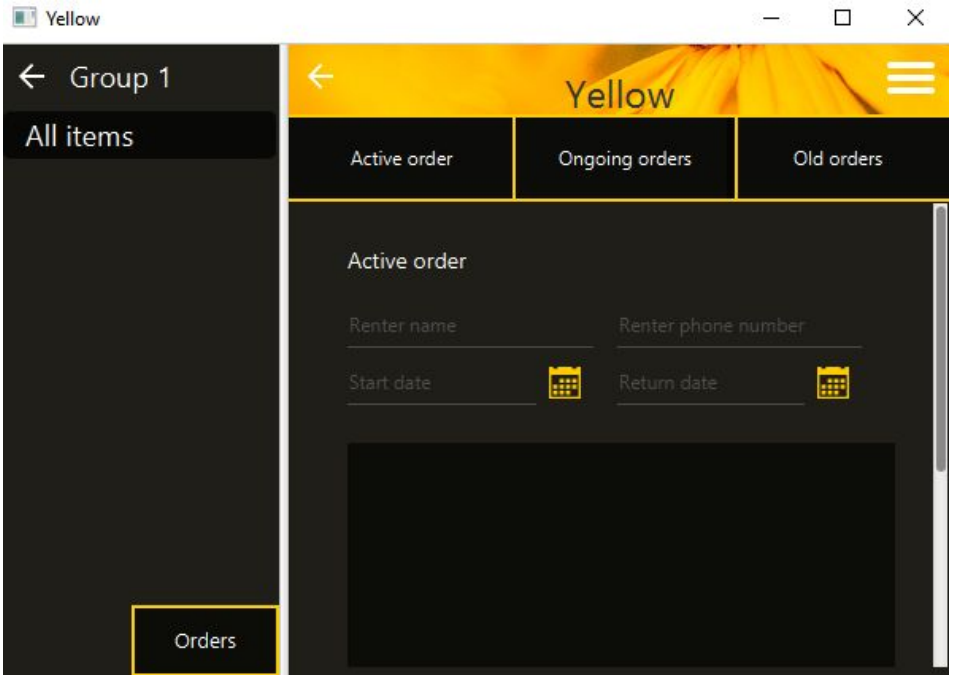


Figure 12: Active order

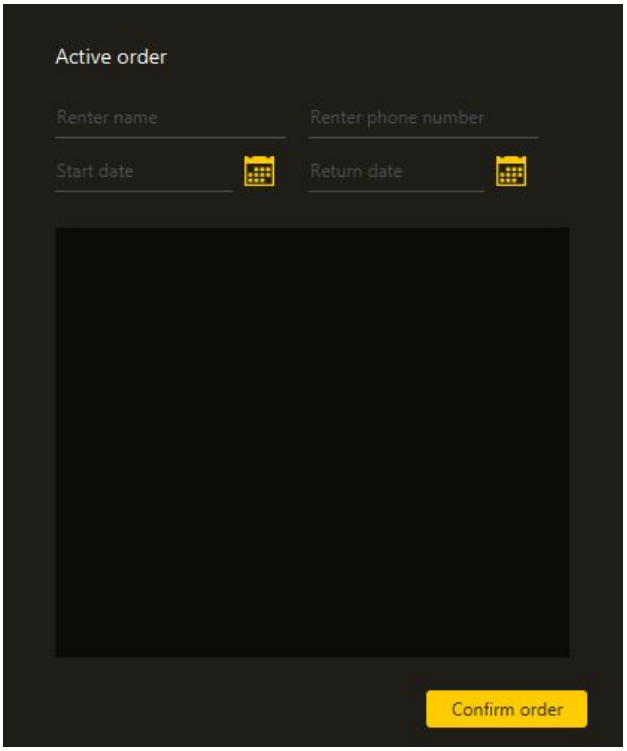


Figure 13: Active order full view

The order view is divided into three different views: Active order, Ongoing orders and Old orders (*Figure 12: Active order & Figure 13: Active order full view*). The active order shows all the items that is added to the order and is shown there until the order is confirmed (Figure 14: Items in active order).

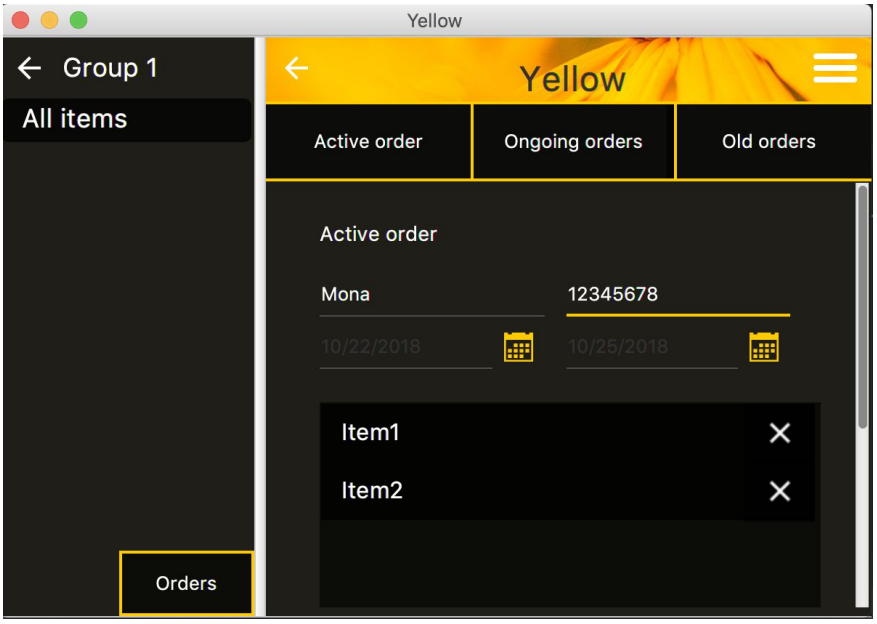


Figure 14: Items in active order

For an order to be confirmed, a renter must be entered and all the items must be available during the chosen rental period. If the items are not available during the picked date, a snackbar error message will pop up telling the user what items are not available and the order will not be placed. It is also not possible to add items from two different groups in the same order due to the fact that orders are linked to a specific group.

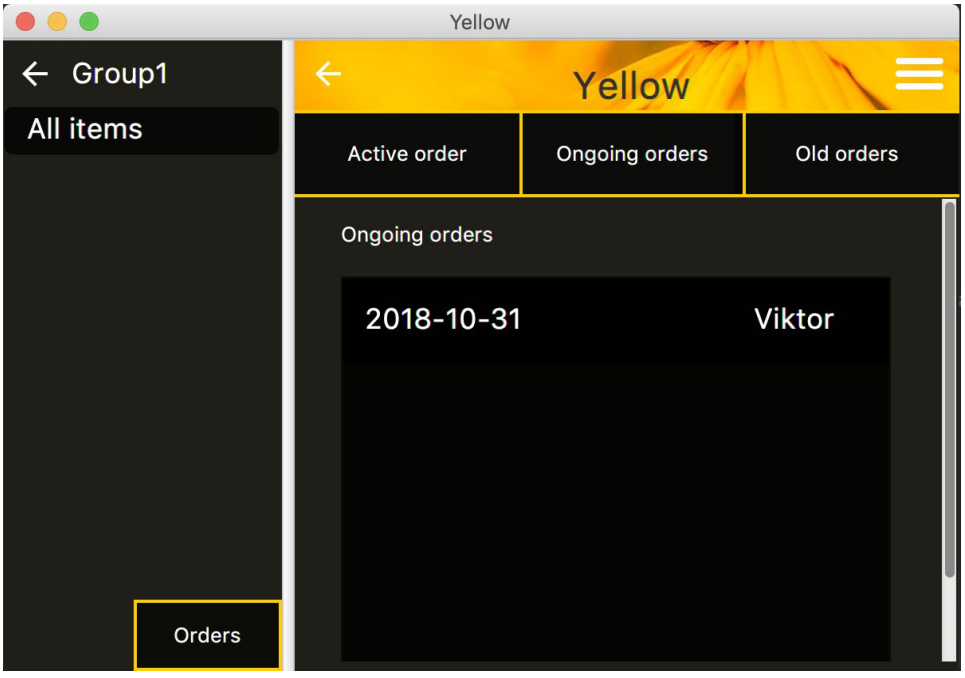


Figure 15: Ongoing orders list view

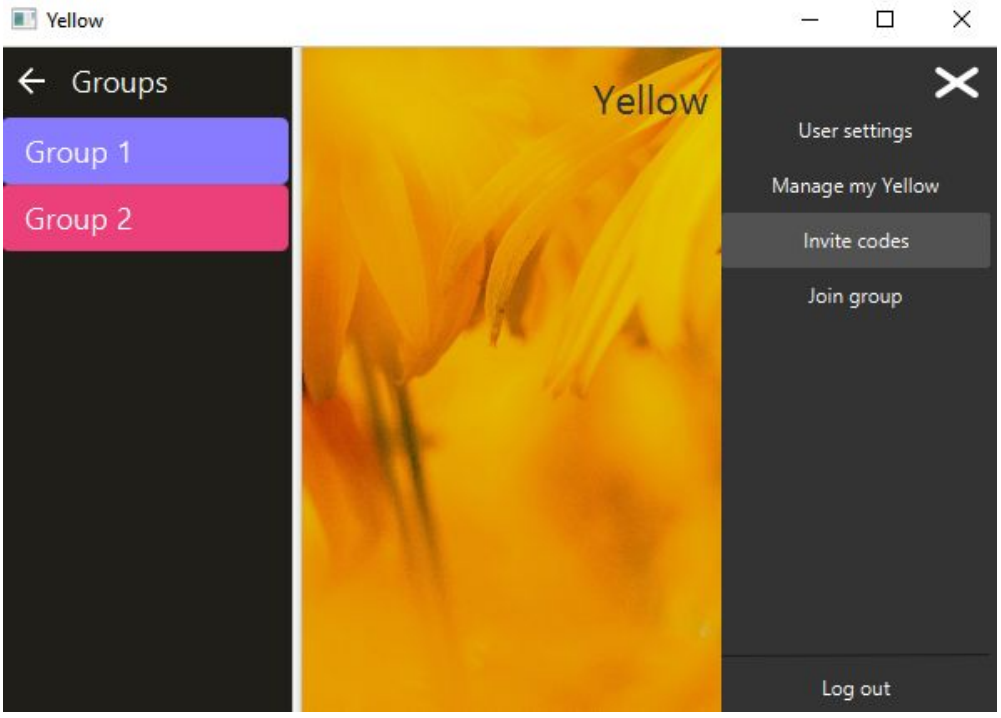


Figure 18: Drawer

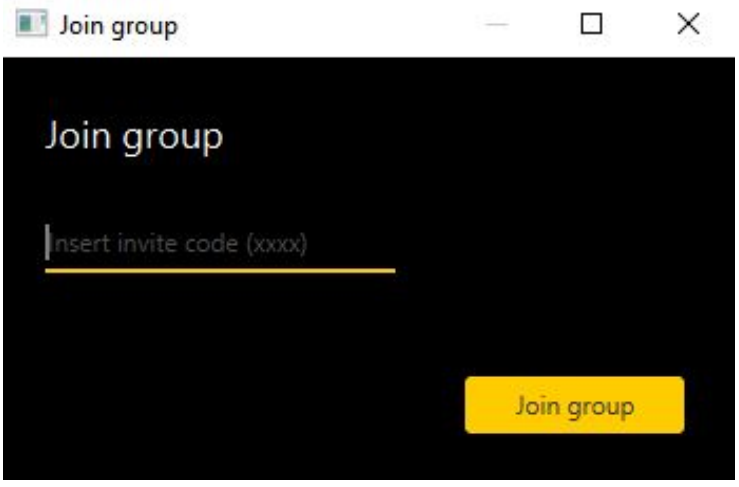


Figure 19: Join group dialog

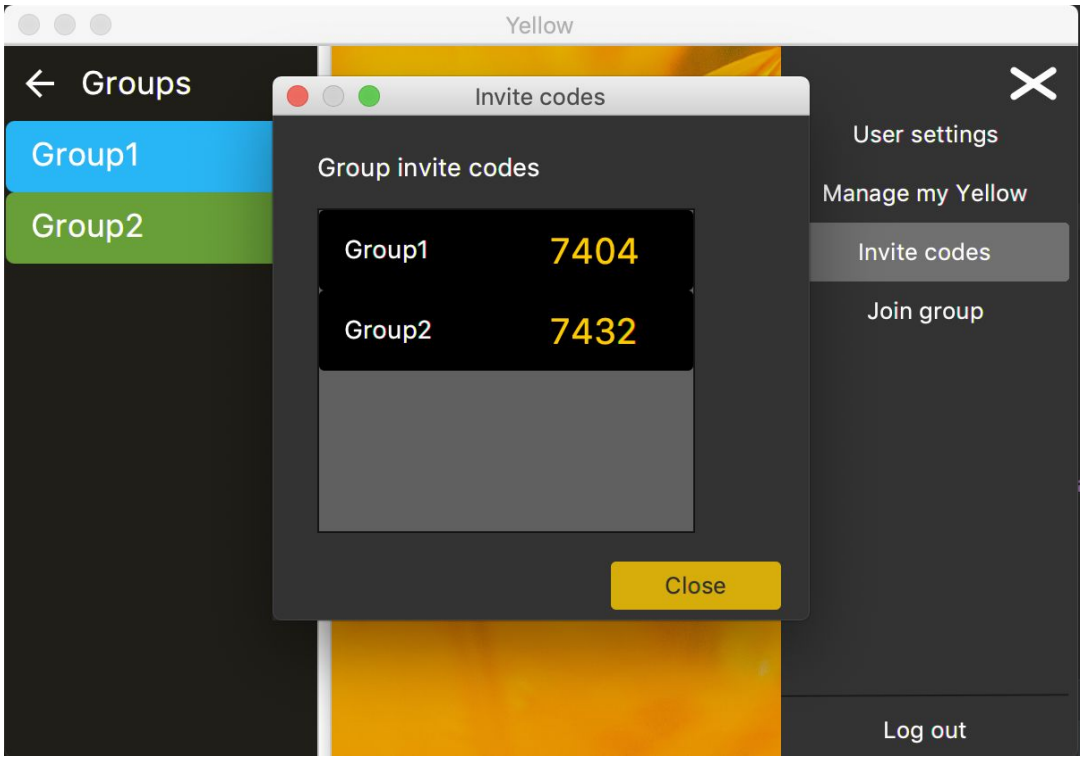
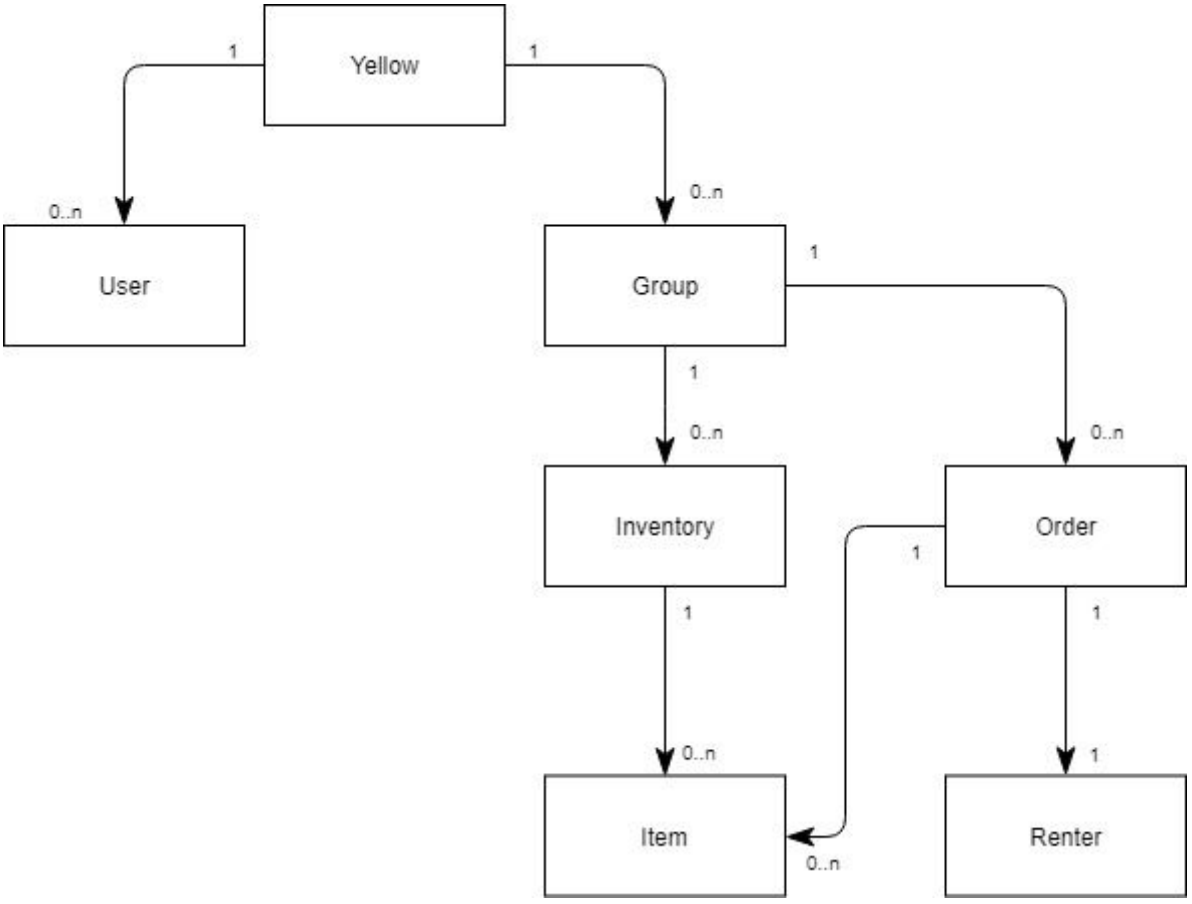


Figure 20: Show invite codes

Another essential view of "My Yellow" is the pop out menu that always is available in the upper right corner in "My Yellow" (*Figure 9: My Yellow*). By clicking on the icon a menu with multiple functions appear (*Figure 18: Drawer*). These functions are considered to be relevant to the user no matter where in "My Yellow" the user is located. This menu contains things such as user settings, a shortcut to "manage my yellow", join group function (*Figure 19: Join group popup*), logout function and a button to show a dialog screen with the invite codes to the users groups (*Figure 20: Show invite codes*).

3. Domain model



3.1 Class responsibilities

User: Represents the users of the app. It contains information about the user, such as name, username, password and what groups they belong to in the form of an id.

Yellow:. Is the bridge between User and Group as well as the manager of the model, delegating tasks to the other classes.

Group: Represents the groups users take part of and also contains what inventories and orders are in the group.

Inventory: Represents an inventory which users can put items in. Inventories are are connected to a specific group.

Item: Represents the item that can be added into an inventory.

Order: Is created to get an overview when an item/items are rented and to get information about the renting.

Renter: Represents the person who rents the item/items.

4. System architecture

The application will be written in Java. MVC will be implemented. The GUI will be designed with Scene Builder and JavaFX.

The application is divided into following packages (*see figure 20*):

- 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.
- View - Has all the view handlers which connects to the .fxml files in the resources directory of the application. The handlers handles the values and small functions that directly affects the .fxml files.

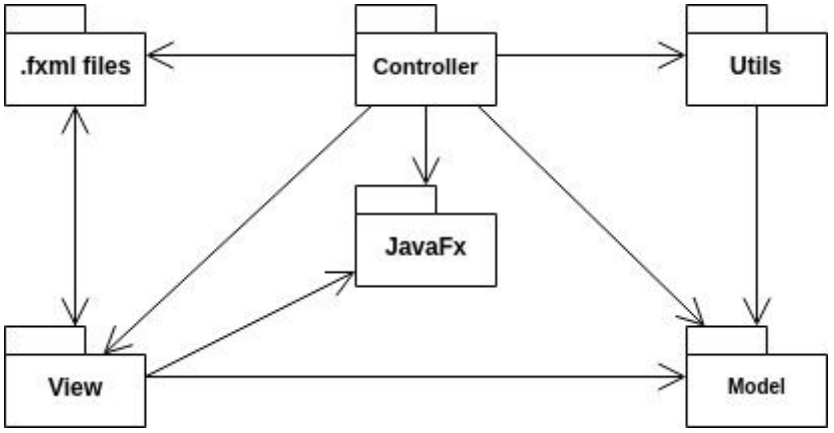


Figure 20: Package dependencies

4.1 Subsystem decomposition

The project strives to use the MVC-pattern which means that the model takes care of data and logic of the data, the view represents the data to the user and the controller serves as a middleman between the view and model [1] . By implementing MVC it will be easier to view the model in different ways in the future since the model is completely separated from everything else. The goal is to have a model with as low coupling and high cohesion as possible.

The project also strives to reach the goals of the S.O.L.I.D principles to make the application more understandable, flexible and maintainable.

The model will take care of all the logic and manipulation of data. The model gets stored data from the controller 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 what is rented and by who.
- **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 domain model shows how these are connected to each other (*see figure 21*).

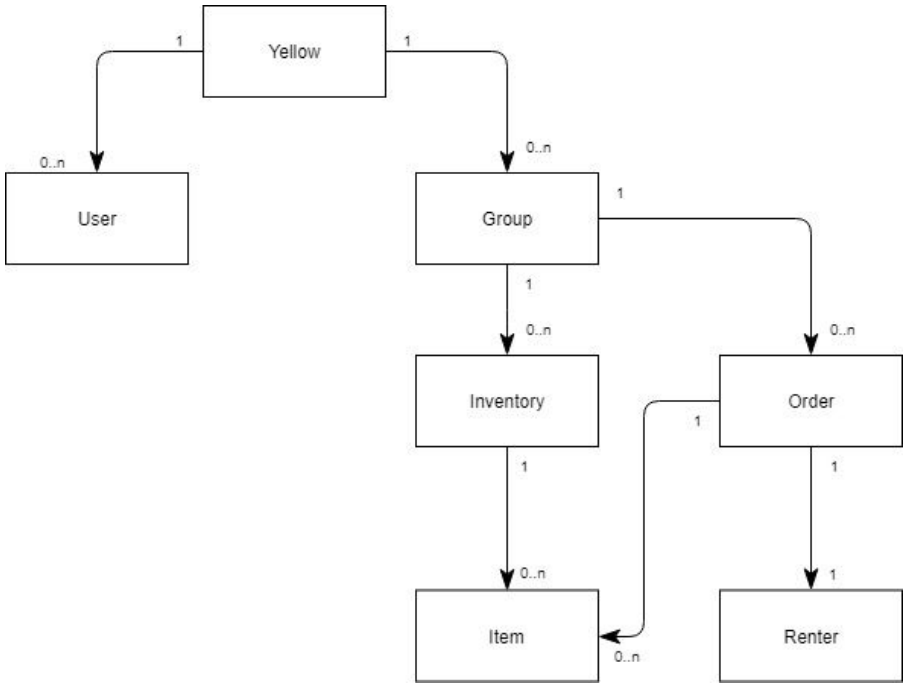


Figure 21: Domain Model

The design model for the project describes how the domain model is implemented in code (*See figure 22*).

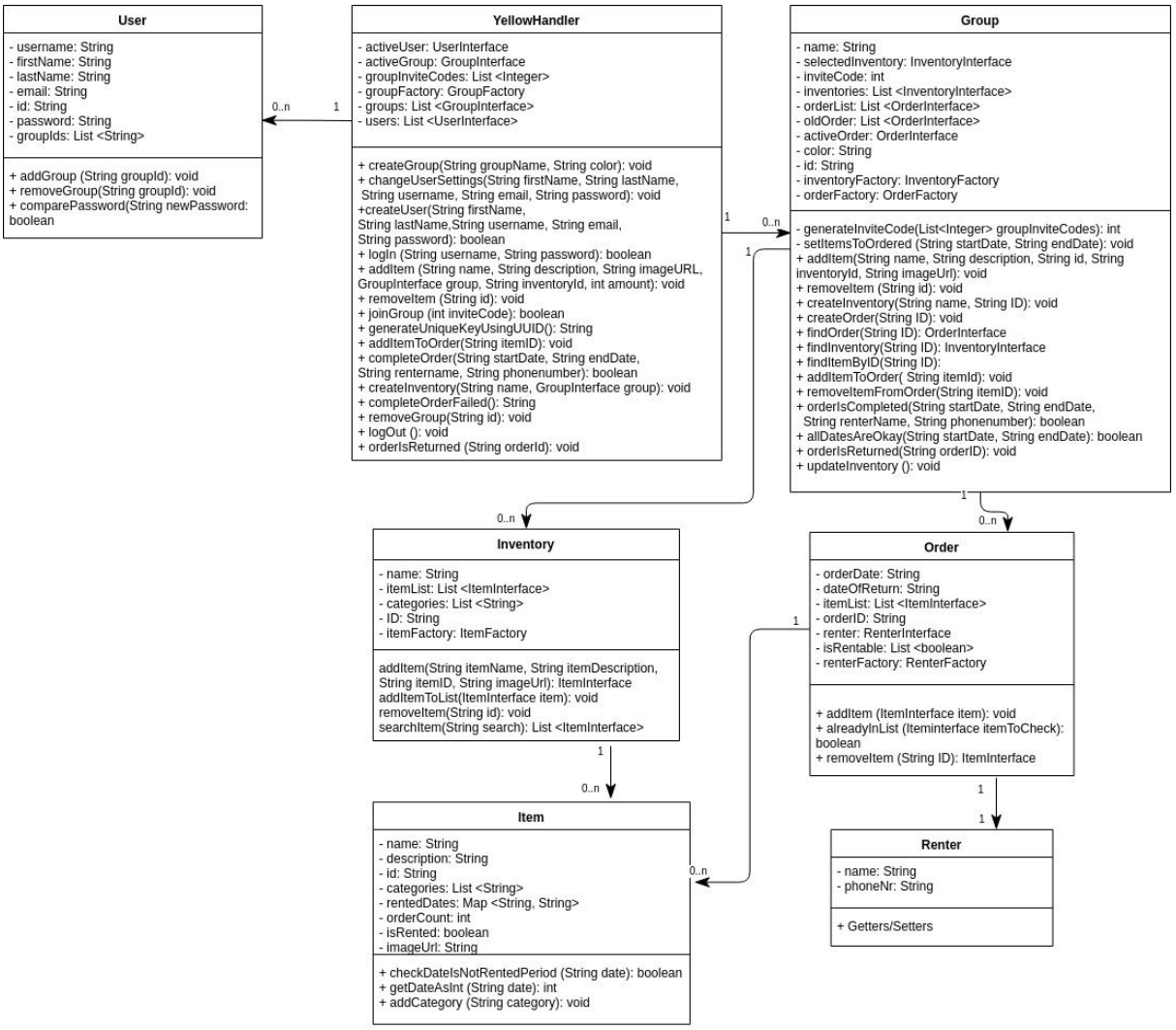


Figure 22: Design Model

To reap the benefits of polymorphism the classes in the model implements the dependency inversion principle through implementing their respective interfaces [2]. The dependency inversion principle helps the project being more abstract and remove unnecessary dependencies [4]. An example of the dependency inversion principle can be seen in figure 23. This improves the low coupling of the project aswell.

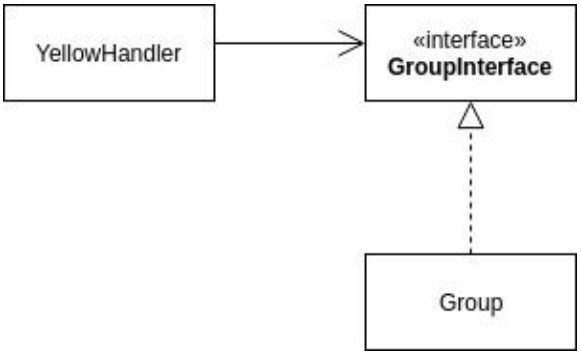


Figure 23: Dependency Inversion Principle

When a class creates a new object a problem is that the static type is an interface but the dynamic type makes the class depend on the class of the object it wants to create. To remove this dependency, a factory pattern is used (*see figure 24*). The factory pattern means that a new class is created solely for creating new objects [6] and in Yellows case, new groups. This makes YellowHandler depend on an interface instead of a class, which improves the low coupling in the model.

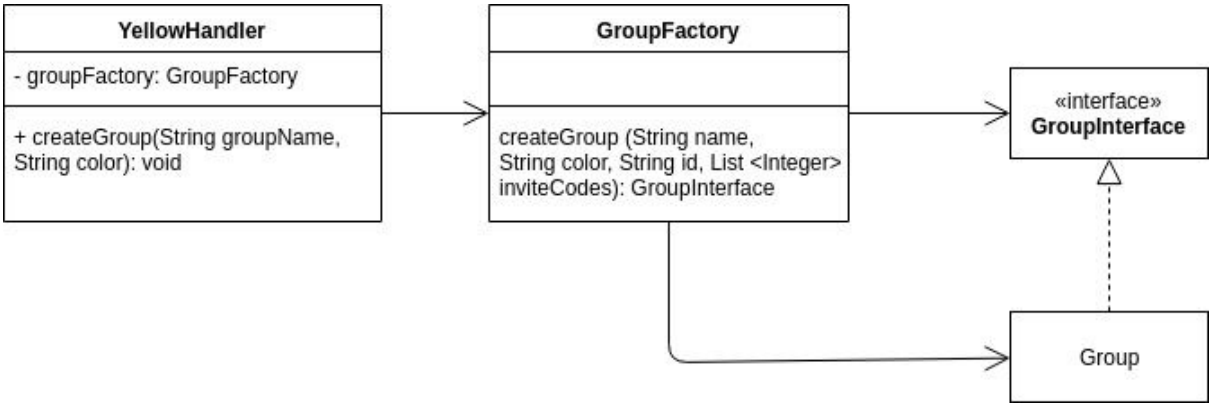


Figure 24: Factory Pattern when creating groups

The YellowHandler class is implemented to make the controller less dependant on the model. When the controller wants the model to do something YellowHandler will start delegating the task to the correct classes. For example when a user is creating a new item (which the user only can if they have created a group with an inventory), YellowHandler delegates the task further to the correct class, this is explained in a sequence diagram (*See figure 25*). Since the classes follows the single responsibility principle, they need to do this.

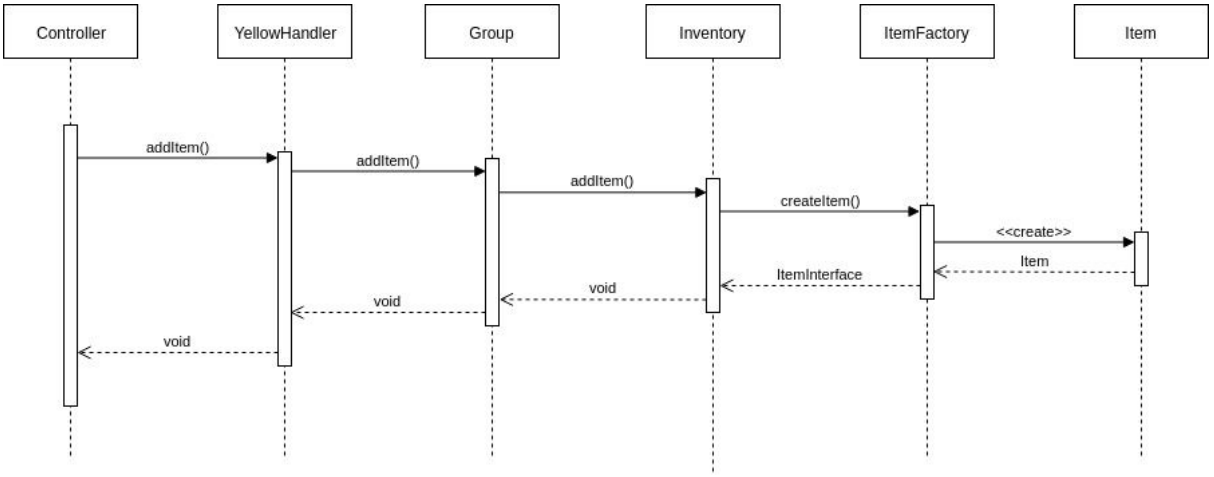


Figure 25: Sequence Diagram for adding an item to an inventory.

When data is edited the view needs to know when the model is updated so it can view the correct data. To solve this problem, the observer pattern is implemented. This means that YellowHandler extends a class Observable and notifies the view of its changed state. This class has a list of observers which are interfaces, and some methods that can add, remove and notify the observer. The view implements the

observer interface and when the model wants to update the view, it goes through the list of observers and calls on their update method [7]. Depending on which view gets updated, the update method is overridden and might be different for the different views (*See figure 26*).

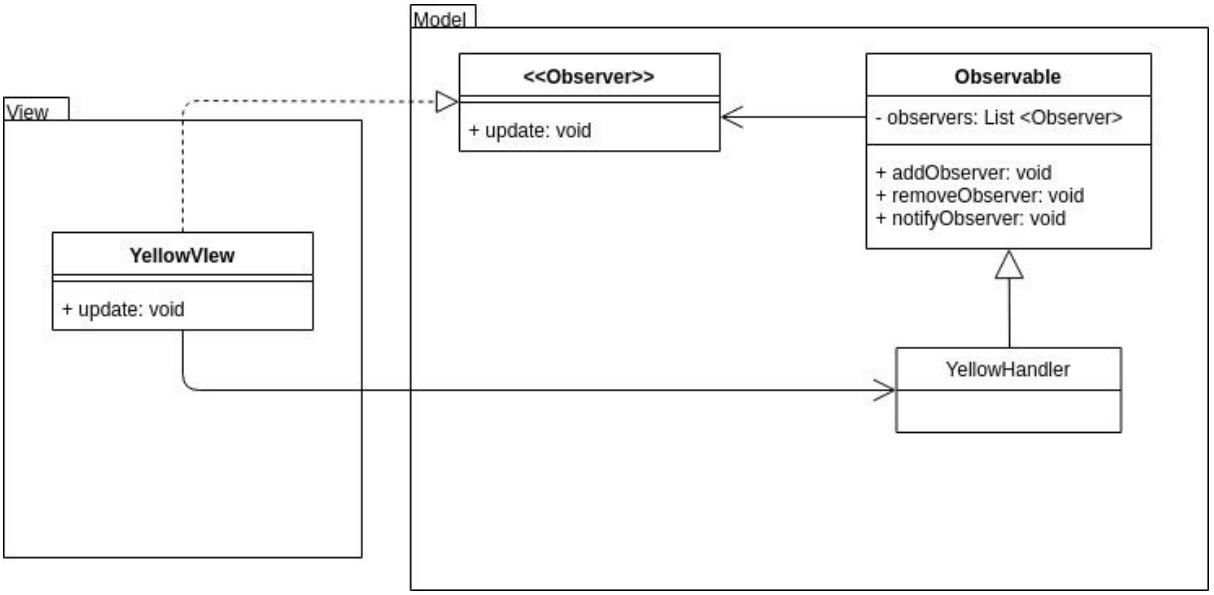


Figure 26: Observer Pattern

4.1.2 View

In this project, the view is represented by the view package and a resource folder full of .fxml files. JavaFX leans heavily towards having a tight coupling between the view and the controller, and it can be discussed whether the controller in JavaFX knows too much about the view or not to be called a MVC-pattern.

In regular JavaFX MVC , each .fxml file is connected to their own controller. This controller is instantiated when the .fxml file is loaded. The controller handles @FXML elements like buttons and textfields and connects the view to the project this way. The controller can then call on methods in the model when it notices that a button is clicked, and with the help of JavaFX properties it can update the .fxml view.
This means that the view only consist of .fxml files, the controller connects the buttons with the help of @FXML bindings, and the model updates the view with the help of specific JavaFX functionality [8].

This project is not implemented this way since one might argue that the controller is too tightly coupled with the view. The view still consists of .fxml files, but it also consist of "viewhandlers". The viewhandlers are still connected to each of their respective .fxml since they still act like the .fxml files "controllers". By naming the controllers "....-view" instead, as well as putting them in the view package, they become a part of the view, and a custom controller can be made to act like a bridge between the view and the model. This means that when a user is pressing a button, the .fxml shows the button, the viewhandler connects the button to the project and uses JavaFX listeners to update an event to communicate with potential listeners (*see figure 27*).

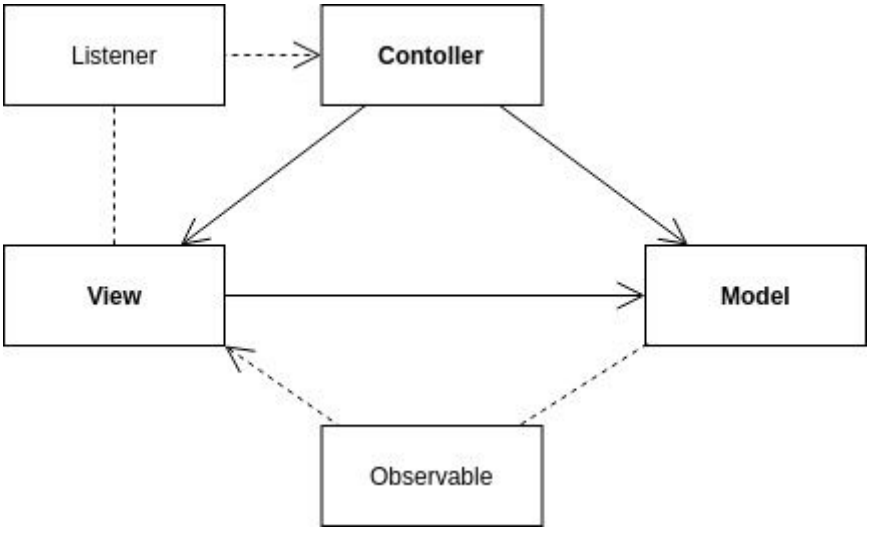


Figure 27: How the project communicates between view, controller and model.

Some of the viewhandlers are implementing the interface Observer and becomes part of the observer pattern, which means that they get updated that way instead of using JavaFX properties.

The project uses encapsulation to prevent direct access to objects components. It is implemented by making the attributes in the models classes private. The only way to use or change these attributes values is through getters and setters.

4.1.3 Controller

The main class in the project acts like the projects controller. The controller is the class that loads most of the .fxml files. In MVC, the view should not know anything about the controller [1]. To solve this the controller uses JavaFX listeners.

The controller extends JavaFX Application, which means that when the project starts, it goes into the Application class and launches the start() method which is overridden in the controller. There are several setup() methods which gets called upon in the start() method which loads the .fxml files and sends out events by the help of listeners. When a button is pressed in the view, the button updates the event, the controller reacts to this and tells the model to do something.

4.1.4 Problematics with JavaFX and MVC

The main problematics with the project has been to separate the view and the controller when using JavaFX. JavaFX encourages a tight coupling between the view and controller which one might argue affects the separation of concerns negatively which MVC is supposed to prevent. The view gets dependant on the controller while the controller also knows about the view which creates a double dependency that is not optimal.

Each .fxml file has a fx:controller which instantiates its respective controller when it is loaded without any arguments.

When a view contains another view, the viewhandler gets responsible for loading the .fxml file for the internal view. This means that viewhandler does things that the controller is currently doing, which is bad. This might be resolved if the two .fxml gets merged together and can use the same controller at the same time. The problem with this solution is that the .fxml might become very large and harder to work with instead.

Another problem is that the controller gets huge. Since just one controller handles all the setup methods with events etc, it is difficult to keep the coding short. A lot of methods that look the same can't get generated into a single method because of minor differences. This might be resolved by creating several controller classes and make them dependant on their respective view. The problem is though, if one was to implement this the controllers would need to be dependant on each other. When a controller catches a call from a button that changes the view, it would need to change view and call that method from another controller since that controller is responsible for the next view.

It is concluded that it is quite hard, or maybe even impossible to implement a clean MVC-pattern with JavaFX. It might work better with a Model-View-ViewModel pattern instead. Since MVVM allows a tighter connection between the view and the viewmodel, it would have been easier working with JavaFX that way.

4.1.5 Testing

The application has a 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 that everything comes together.

5. Persistent data management

5.1 Saving data

The application saves and loads entire objects through the java.io interface serializable. The objects that will be saved implements the serializable interface. That makes them runnable through an ObjectOutputStream, which translates the data to a stream of bytes. The ObjectOutputStream is linked to a file with the .ser extension. When that is completed entire lists of those objects (Users and Groups) are run through the ObjectOutputStream and saved down in the .ser file.

5.2 Loading data

After saving objects with the .ser files extension they can be loaded back into the application through an ObjectInputStream. The ObjectInputStream is linked to the already existing .ser file. When that is done the application use the ObjectInputStream to load the entire file back to the same state as it was saved it in.

The save and loading classes is located in the utility package. See figure 28 for a visual representation of serialization and deserialization.

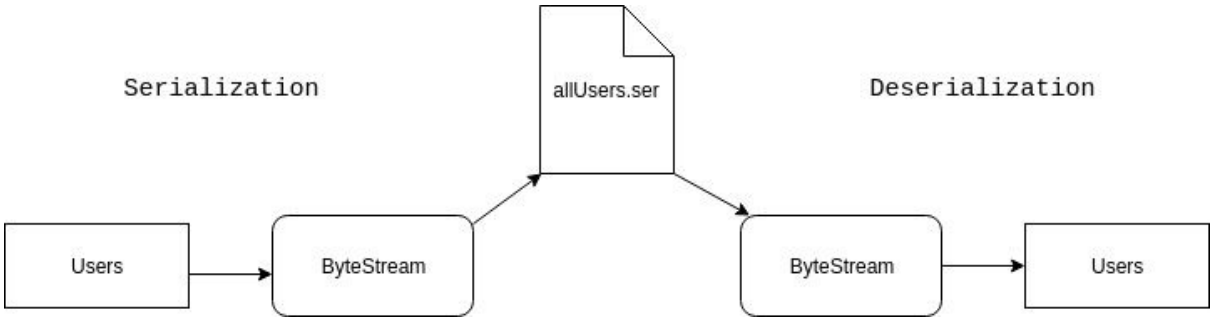


Figure 28: Serialization and deserialization of the Users in the application

5.3 Images and Icons

All images and Icons are saved in a directory named Img in the resources directory. When adding your own image to an item the application creates an output file with a representative name in the Img folder. The image will then be written into the file through the ImageIO.write function. The URL to the image is saved in the item to be used later by the application.

6. Peer review (of group 16´s project)

The project uses a consistent coding style, setters and getters are furthest down in every class. The classes are not too big and are easy to navigate through to find methods and functionality.

The whole project separate classes to their tasks and what they represent. In the model the classes only manage what they logically should know about which follows the Single Responsibility Pattern.

The code is documented with JavaDoc which helps understanding the code. The project also has describing and proper names for variables and classes.

The model has very few tests that only focuses on the model. But the model only consists of getters and setters, which may be why there is no need for further testing of the model. But tests for the viewmodel should be implemented since it does the most in the application. As it stands right now, no tests for the viewmodel are implemented which means that most of the application is not tested at all. We can also distinguish that multiple classes lack any form of testing.

The project follows the MVVM-design pattern and it seems like it is not implemented correctly [9]. We are not certain if the model should only consist of setters and getters while the view model does the heavy lifting of business logic and for example creating new exercies. It seems like the model is very thin and is not easily reusable if one were to use a new view. It seems like most of the business logic is placed in the viewmodel and not in the model, which according to microsoft [9] is not the correct way implement MVVM.

An example of what can be changed in the application is when the user starts a workout. The view tells its respective viewmodel what to do which is fine. The problem is that the viewmodel keeps information that the model should handle. The model has a class named ActiveWorkout which should have information about if it is set to true or false, not the viewmodel as it is now.

The model package implements the dependency inversion principle by making classes relying on interfaces instead of classes in the most of the application, but is not used to its fully potential, but this removes some dependencies between the different classes, and makes the classes rely on abstractions instead.

One thing they could improve to remove dependencies from the viewmodel to the model is when they are creating new objects of a class. For example in ExerciseCreatorViewModel:

```
IExercise exercise = new Exercise(name, unit, description, instructions, exerciseCategories);
    model.addCustomExercise(exercise);
```

The variable exercise is of the static type IExercise but the dynamic type is of the Exercise type. This means that the viewmodel knows about Exercise as well. If a factory pattern for exercises and other objects are made, this dependency can be removed.

The application uses a form of safe copy to avoid the data in model to be changed unwillingly. For example they create new list variables that copy the values of the original lists which makes it harder to manipulate the model (Code example: `workouts = new ArrayList<>(model.getWorkouts());`).

Overall the application does not have many functions more than adding things to different lists which makes it hard to see how more advanced problems are solved. The application only creates new objects and add these to a list or collects data from the model. Things to add to the application could be login functionality with different users for example, but might be hard since the model basically consist of getters and setters right now.

References

[1] R. Eckstein, "Java SE Application Design With MVC," 2007. [Online]. Available: https://www.oracle.com/technetwork/articles/javase/index-142890.html, retrieved: 2018-10-28.

[2] L. Gupta, “SOLID Principles in Java [With Examples]”, unknown. [Online]. Available: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/, retrieved: 2018-10-28.

[3] M. Sanaulla, “Cohesion and Coupling: Two OO Design Principles”, 2008. [Online]. Available: https://sanaulla.info/2008/06/26/cohesion-and-coupling-two-oo-design-principles/, retrieved: 2018-10-28.

[4] B. Venners, “Polymorphism and Interfaces”, 2018. [Online]. Available: https://www.artima.com/objectsandjava/webuscript/PolymorphismInterfaces1.html, retrieved: 2018-10-28.

[5] T. Janssen, “OOP Concept for Beginners: What is Encapsulation”, 2017. [Online]. Available: https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/, retrieved: 2018-10-28.

[6] L. Gupta, “Java Factory Pattern Explained”, unknown. [Online]. Available: https://howtodoinjava.com/design-patterns/creational/implementing-factory-design-pattern-in-java/, retrieved: 2018-10-28.

[7] R. Agrawala, “The Observer Pattern in Java”, 2014. [Online]. Avaibable: https://dzone.com/articles/observer-pattern-java, retrieved: 2018-10-28.

[8] Oracle, “Mastering FXML”, unknown. [Online]. Available: https://docs.oracle.com/javafx/2/fxml_get_started/whats_new.htm, retrieved: 2018-10-28.

[9] C,Dunn, D, Britch, & Poulad “The Model-View-ViewModel Pattern”, 2017. [Online]. Available:https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm?fbclid=IwAR3HjfmZGg-DwAQXYkwmXstScys-TPXROfAMQT_IqT_Vq_iT5VMQoarcJ68, Retrieved: 2018-10-28.