YELLOW

System design document (SDD)

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

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 dependent at all of the rest of the project. 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** - An 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. 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. 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 1*):

- 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.

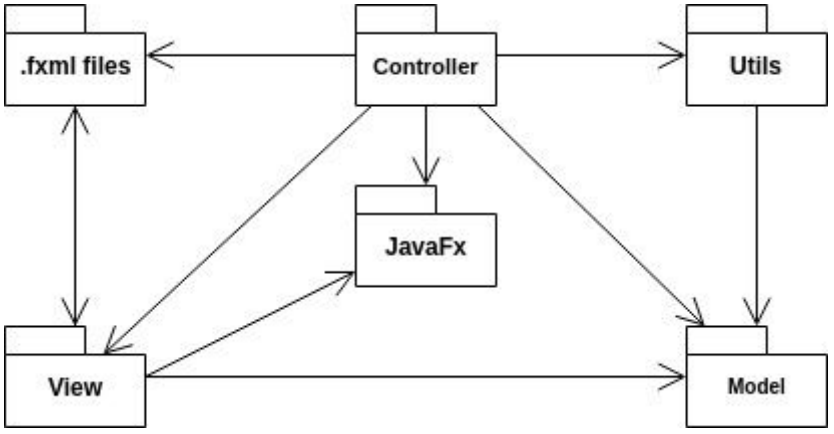

Figure 1: Package dependencies

2.1 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 2).

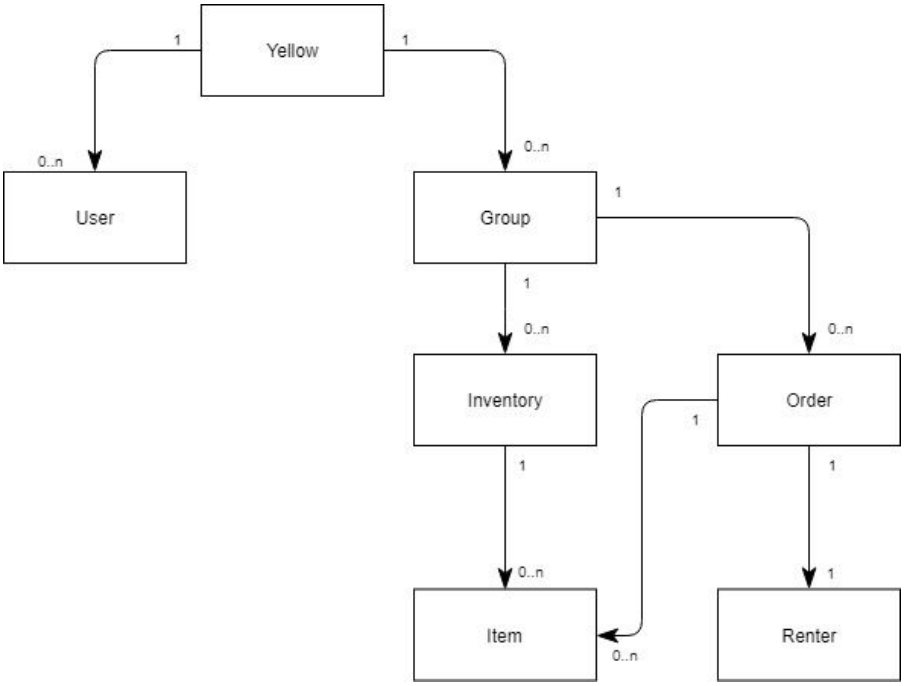

Figure 2: Domain Model

The design model for the project describes how the domain model is implemented in code.

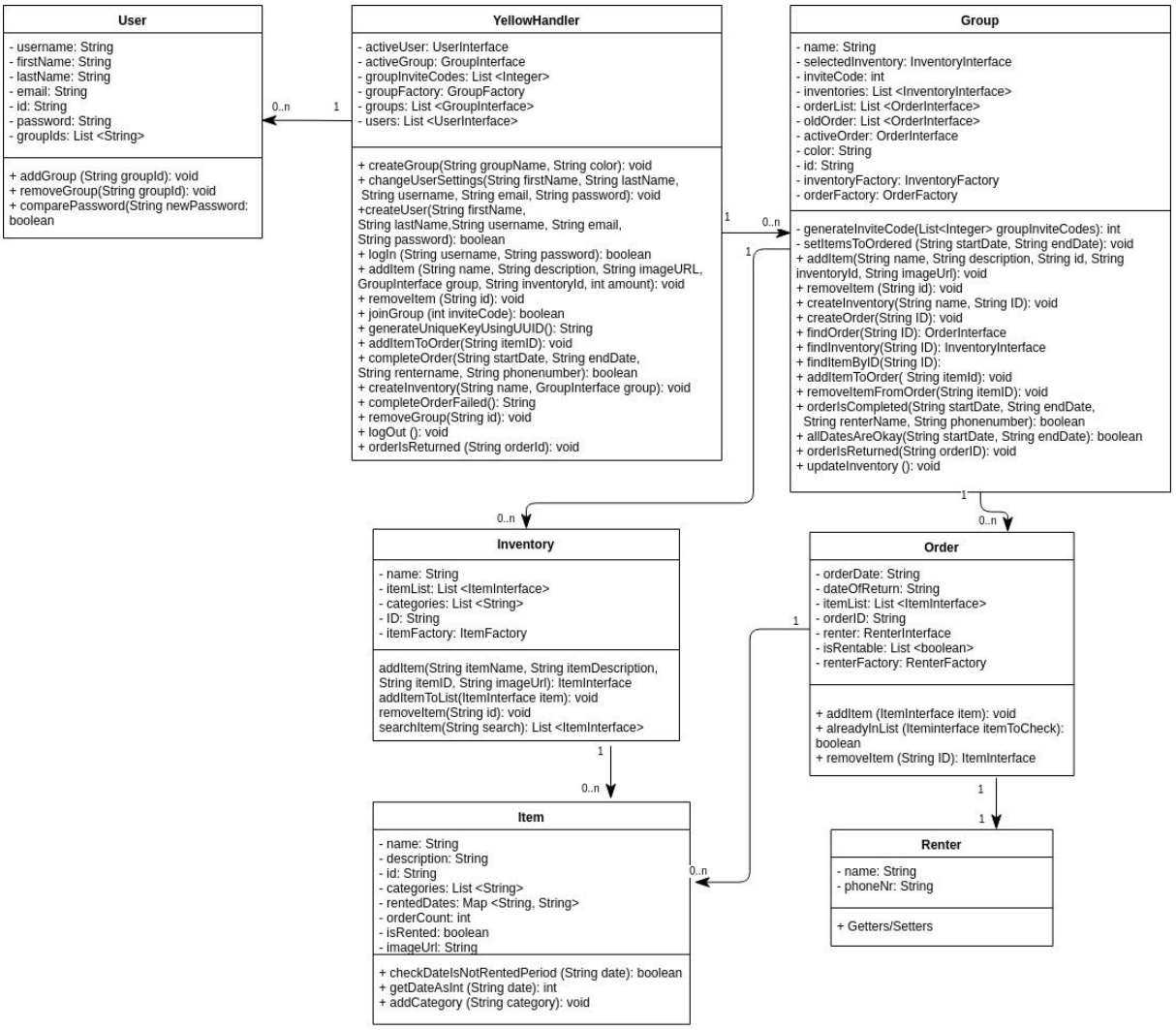

Figure 3: Design Model

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 4. This improves the low coupling of the project aswell.

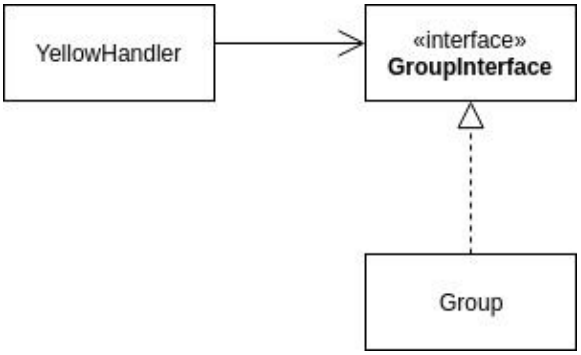

Figure 4: Dependency Inversion Principle

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 5). 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.

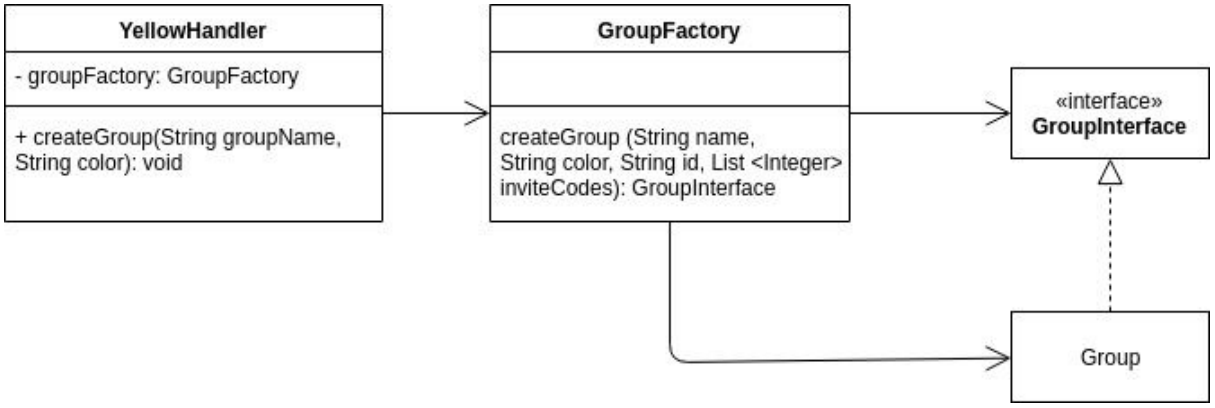

Figure 5: Factory Pattern when creating groups

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 6). Since the classes follows the single responsibility principle, they need to do this.

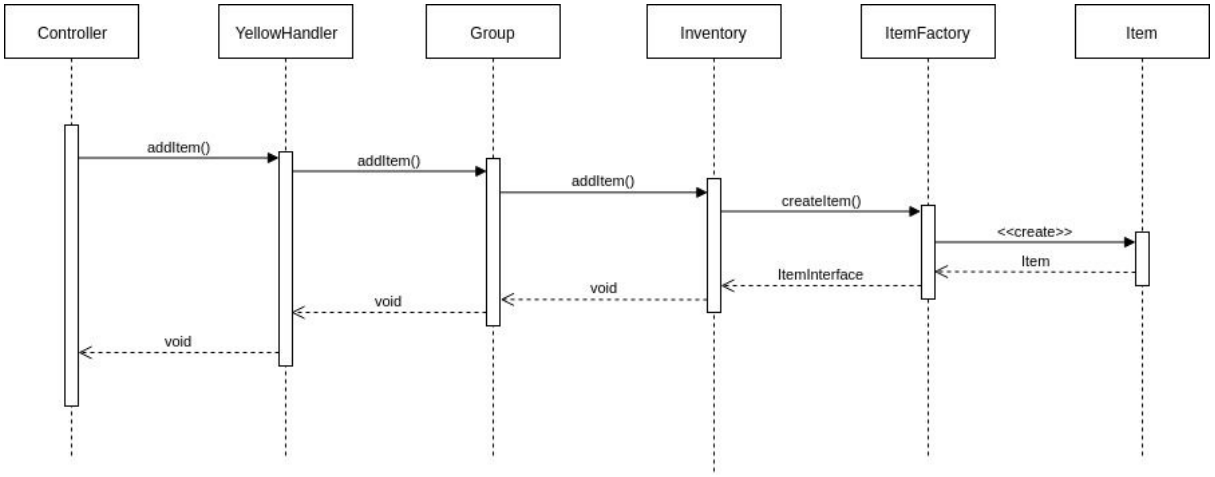

Figure 6: Sequence Diagram for adding an item to an inventory.

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.

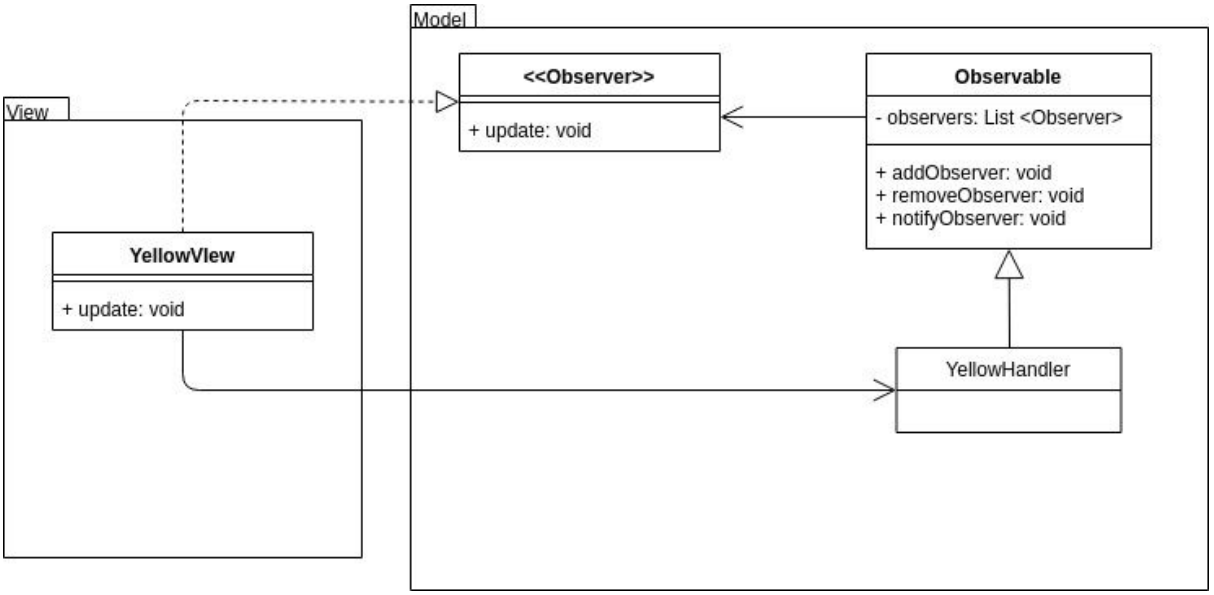

Figure 7: Observer Pattern

2.1.2 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.

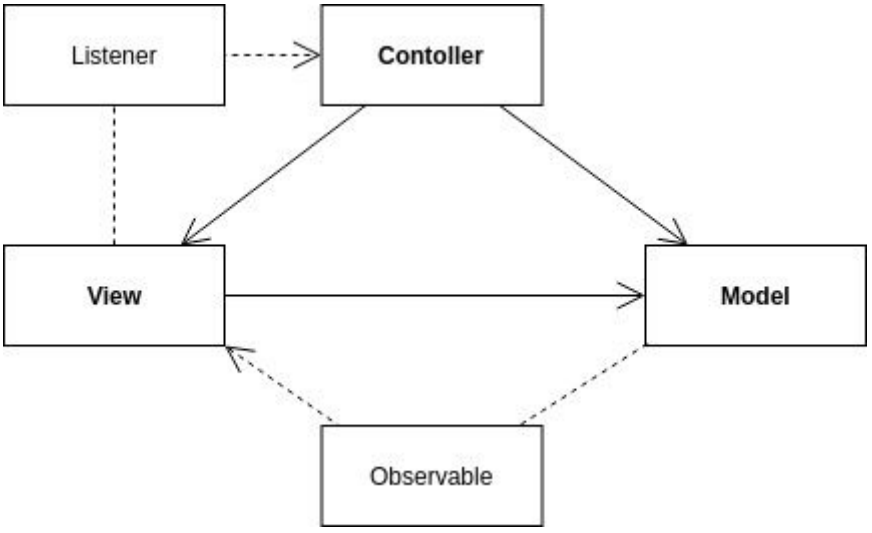

Figure 8: How the project communicates between view, controller and model.

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.

2.1.3 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.

2.1.4 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.

2.1.5 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.

3. Persistent data management

3.1 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.

3.2 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 9 for a visual representation of serialization and deserialization.

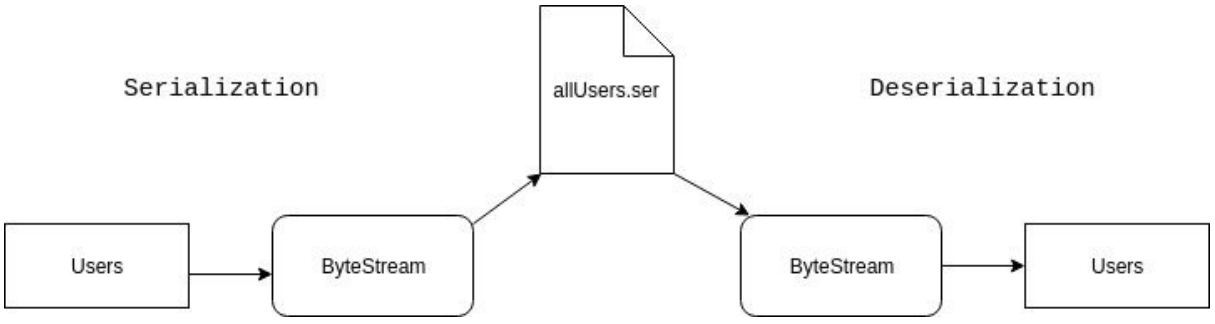

Figure 9: Serialization and deserialization of the Users in the application

3.3 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.

4. 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.