

# System design document for ShatApp

Benjamin Vinnerholt, Filip Andréasson,  
Gustaf Spjut, Gustav Häger, Jonathan Köre

2018-10-17  
version 3

## 1 Introduction

This is the system design document for ShatApp, a chatting application created to be used in a company setting.

### 1.1 Definitions, acronyms, and abbreviations

JSON - JavaScript Object Notation. The format chosen to be used for storing data.

JaCoCo[1] - Java Code Coverage. A code coverage library developed by the EclEmma team.

Pmd[2] - a static code analyzer. It finds issues in the code based on a specified ruleset.

## 2 System architecture

Only one machine is involved in running the application. The different system components are the Model, View, Controller and Infrastructure. The Model is responsible for the applications data and logic. It is essentially a model of how the program works at its core. The Controller and View are part of the interface towards the user. The View presents the Model for the user, and the Controller manipulates the Model based on the users interaction with the View. The Infrastructure contains components outside of the domain model that are needed for the handling of information, in this case it contains the database. The data is stored and loaded from here. If the application had a server, it would have been in Infrastructure as well.

The View is dependent on the Model to get data. The Controller manipulates the Model based on user input from the View. The Model notifies the View and the Infrastructure through Observer Pattern when it is changed.

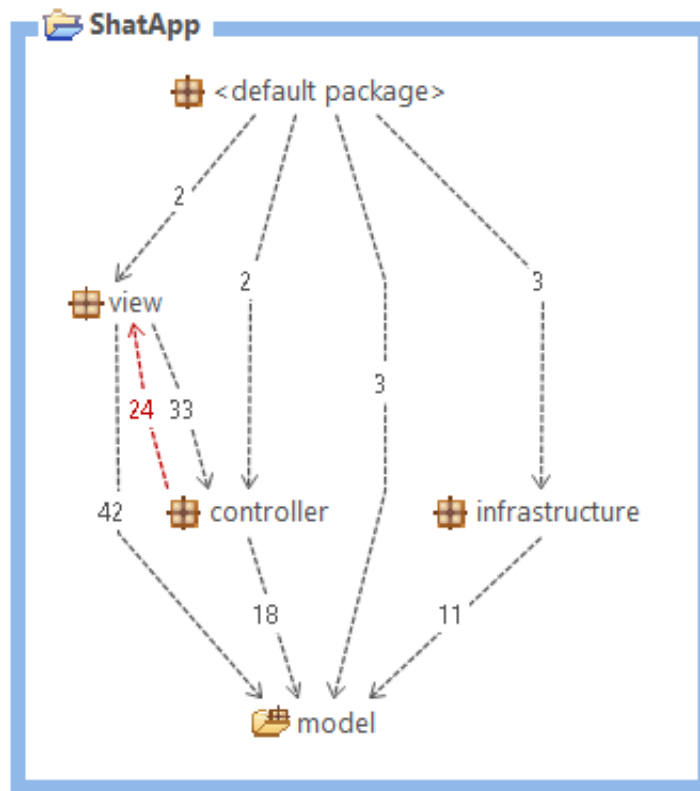


Figure 1: An overview of the system

## 2.1 Subsystem decomposition

This section describes each identified system component that has been implemented.

## 2.2 Model

This is the core of the program. The model holds all the data and modifies it accordingly when the user interacts with the program.

- MVC implementation

The Model holds all the data, and has methods for manipulating it. The controller calls on the model when the user has made an input regarding data modification. When data modification has occurred, an update message will be broadcasted to its observers, so that they can update themselves accordingly.

- Design model

The MainModel acts as a facade for the model, and makes it easy to access conversations, messages and users, as well as the fundamental functions such as sending a message. The data is represented by the classes Conversation, Message and User, and mainly holds primitive types. The Message and User does not hold any Users themselves, but rather IDs to other users. This is because it made the classes much easier to store, and because serializing the contacts as a list of users would create an infinite loop, with the first user having a second user as a contact, which has the first user as a contact and so on. Now, when a view needs to show info about a user, it can simply request it from the MainModel.

The MainModel holds a variable for the current user of the program, called `activeUser`, and a variable for the conversation the `activeUser` is currently in. This variable is called `activeConversation`. These variables decides the outcome of most methods in the MainModel facade, such as the methods `getContacts`, and `getUsersConversations`.

A small component in the model package is the `observerpattern` package. We decided to implement our own version of the pattern. Our implementation differs from the method `update(Observable, Object)` in the `java.util.Observer`, to instead simply be `update(UpdateType)`. This makes it so that we can guarantee the `Object` passed with this method to be an `UpdateType`, since this is the only thing that we want to send in our application. Our implementation also removes the first parameter, "`Observable`", since we always know that the `Observable` that called the method was the `MainModel`. The files found in this package is a `ModelObserver` class and a `ModelObservable` interface.

- Design patterns

**Iterator:** Iterator is used in the program when we want to read what is in a list or map, without being able to add things to these objects. For example: When the View requests to get the conversations from the MainModel, the MainModel returns an iterator with the conversations. This makes it easy for the View to reach all the data from the conversations, but it cannot add a new conversation to the Map that resides in the MainModel directly. This is because we want control over how things are added to these Maps. The conversation must for example be initialized with a valid ID, which it gets in the models `addConversation` method.

**Observer:** The MainModel extends `ModelObservable` and when it notifies observers it passes an enum, `UpdateTypes`, to these observers. The enum enables observers to interpret the modification of the model, and decide how to react based on that.

Facade: The MainModel acts as a facade for the rest of the model package. This is because we want to have control in what happens when something is being added or altered. For instance, if a users info is updated, we want the model to notify its observers (here being the View and the JsonSaver) so that they can be updated with the new info of the User. It also makes it easy to understand and use the MainModel, using simple method calls directly to the MainModel instead of using method chaining.

- Class diagram for design model can be viewed in Figure 2

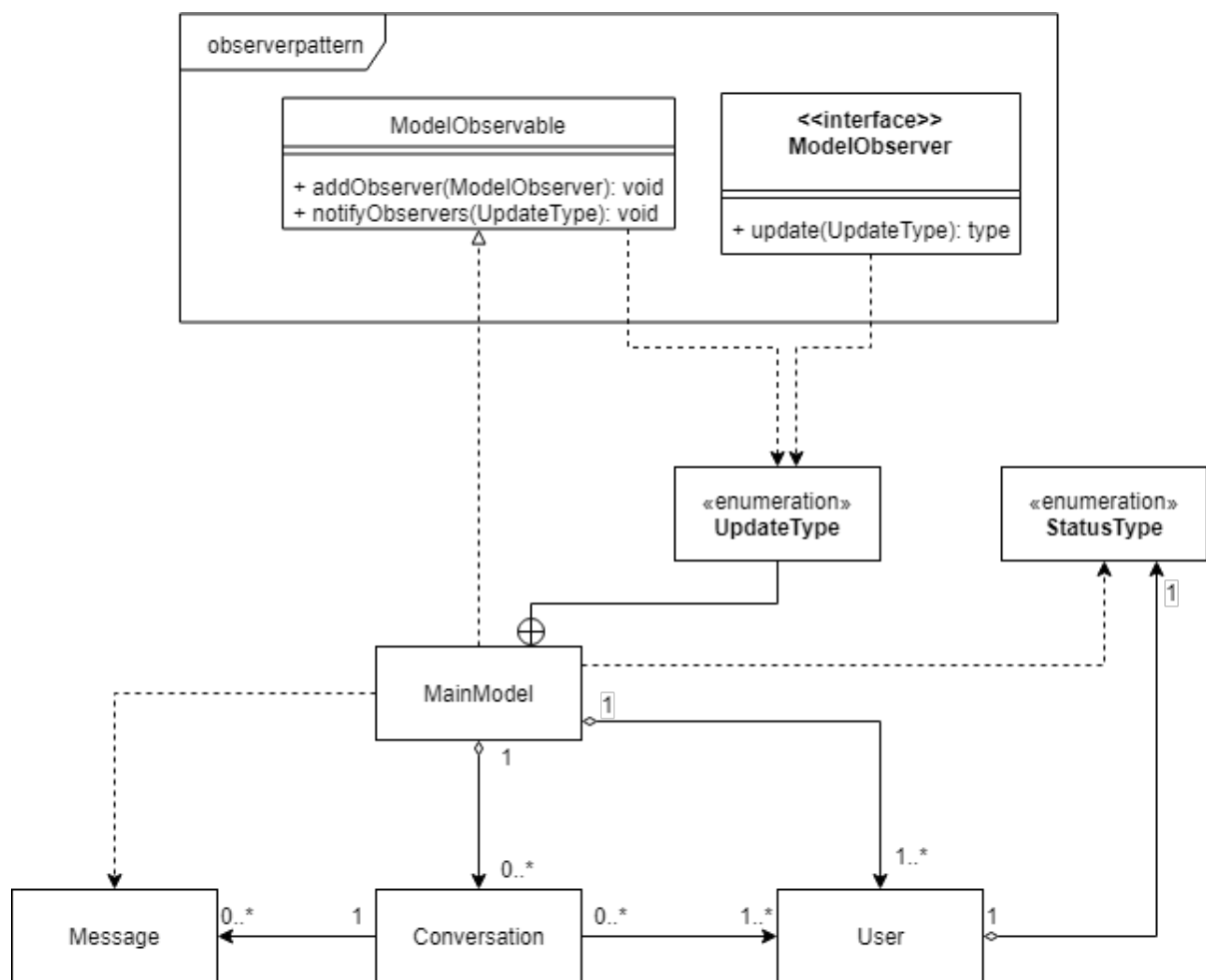


Figure 2: Class diagram for the model

Diagrams

- Dependencies

A STAN diagram of the dependencies in the Model can be seen in Figure 3.

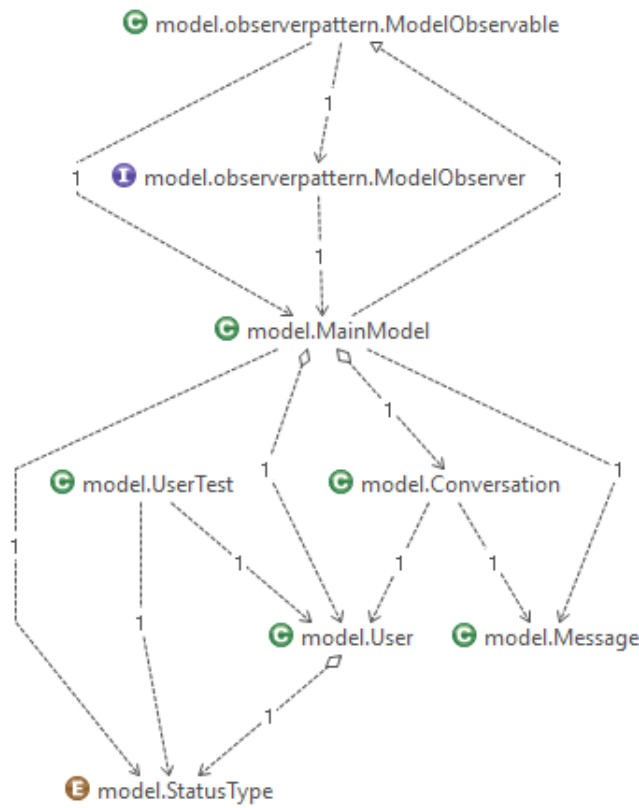


Figure 3: Dependency analysis using STAN on the Model package.

- UML sequence diagram

An UML sequence diagram can be viewed in Figure 4. The diagram depicts the process of sending a message and in it you can view the models part.

### Quality

- List of tests

The tests are in `src/test/model` and can be run with JaCoCo coverage using `"gradlew test jacocoTestReport"`. It can be viewed in Figure 5. The test coverage in `User` is poor since the `toString` and `hashCode` methods are not tested with every possible case, however their functionality is tested and is deemed to be in working condition. The same can be said about `Conversation`. Some methods in `MainModel` which have recently been written have no tests written for them.

- Known issues

Pmd with quickstart ruleset[3], which applies many general rules, reports no issues.

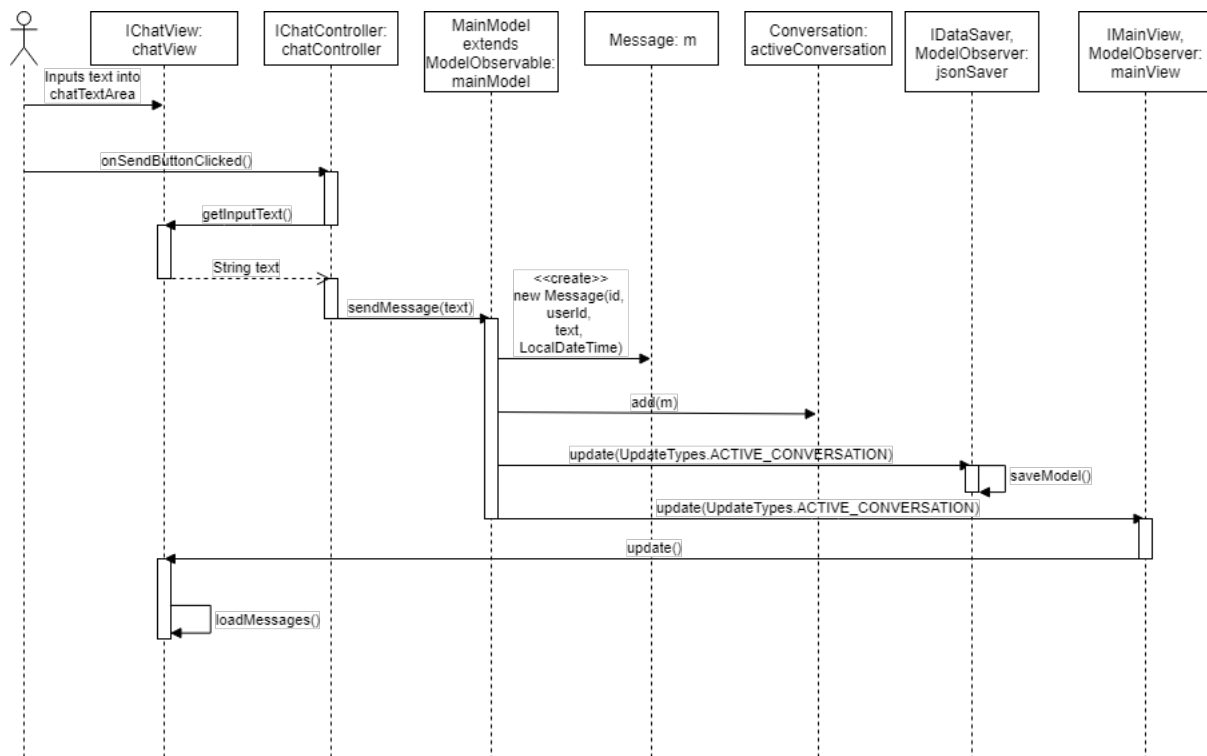


Figure 4: A sequence diagram for sending a message

## model

Element	Missed Instructions	Cov.	Missed Branches	Cov.
User	<div><div></div></div>	82 %	<div><div></div></div>	64 %
MainModel	<div><div></div></div>	92 %	<div><div></div></div>	88 %
StatusType.new Object().{...}	<div><div></div></div>	0 %		n/a
StatusType	<div><div></div></div>	75 %	<div><div></div></div>	0 %
Message	<div><div></div></div>	67 %		n/a
Conversation	<div><div></div></div>	93 %	<div><div></div></div>	70 %
User.new Object().{...}	<div><div></div></div>	89 %		n/a
MainModel.UpdateTypes	<div><div></div></div>	100 %		n/a
Total	159 of 1 338	88 %	31 of 132	76 %

Figure 5: A JaCoCo report for the model package.

## 2.3 View

The view is responsible for displaying the data that the model holds to the user.

- MVC implementation

The view knows when to update itself via observer pattern, and sometimes via the controller. This means that the model does not know about the view. The view only has simple methods for altering itself, making the view relatively dumb, which it should be. Buttons and other inputs from the user are directed

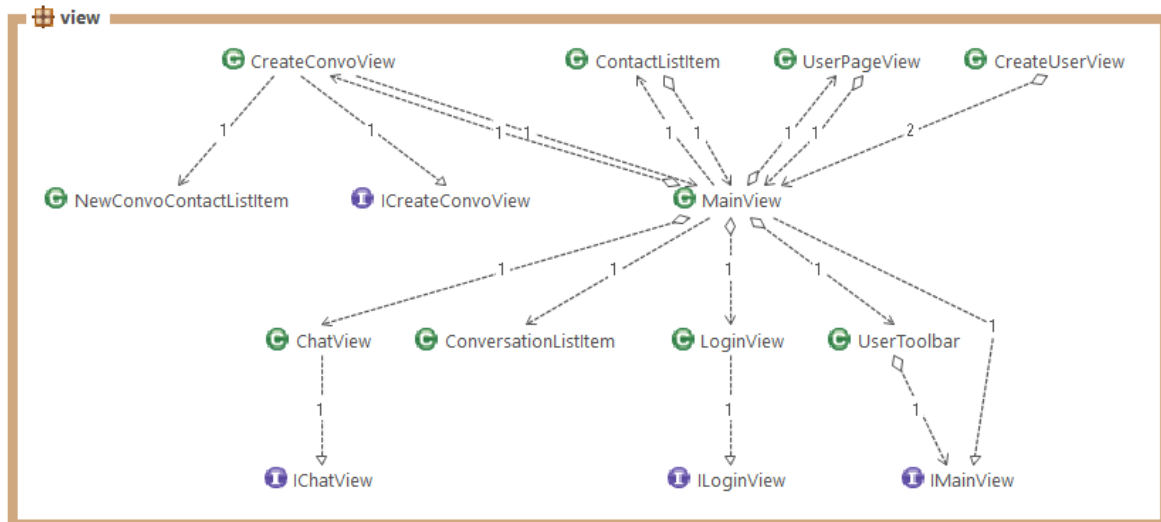


Figure 6: View package dependency analysis using STAN.

to methods in the controller, via EventHandler. This means that the view is not strongly associated with the controller.

- Design model

An overview of the design model can be seen in Figure 6.

The view is comprised of a main class called MainView which extends a JavaFX AnchorPane and implements the Initializable, IMainView, and Observer interfaces. MainView aggregates the other view related classes. It houses the LoginView, ChatView, UserPageView and UserToolbar. These views all extend AnchorPanes so that they can be added onto the MainView. All the views implement a corresponding interface. MainView is an Observer of MainModel.

The view components are separated into smaller classes to make the code easier to read, since having them all together would result in a god object. The ChatView loads messages from the MainModel and uses the inner class MessageItem to display as messages in the chat area.

The LoginView presents a standard Login interface to the user. Upon a successful login, the model is set up to hold data which is relevant to the user that logged in. The process of logging in can be seen in Figure 9. After this, the rest of the view is initialized.

The UserToolbar displays info and methods to change the current user, such as a dropdown for setting a status, and a button that brings up a UserPageView, where the more basic settings such as a first and last name can be set.

The MainView receives updates since it is an observer to the MainModel. The reason for having multiple update types is so that the view only updates rele-

vant info.

The view package holds a lot of interfaces, and these interfaces are not meant for the view to depend on. This is because these views themselves must extend JavaFX classes to be added onto our MainView. The real user of these interfaces is the controller package. This means that the views can be interchanged, and it would not have an impact on the controller. This makes it possible to switch out the whole view and still use the same controller.

The interfaces of the view package are not using JavaFX, making them possible to change for another type of visual representation.

- Design patterns

The observer pattern is implemented: the view is an observer of the MainModel, and updates itself based on modifications on the model which are broadcasted via parameters in the update method.

- The UML diagram for view can be viewed in 7 and 8.

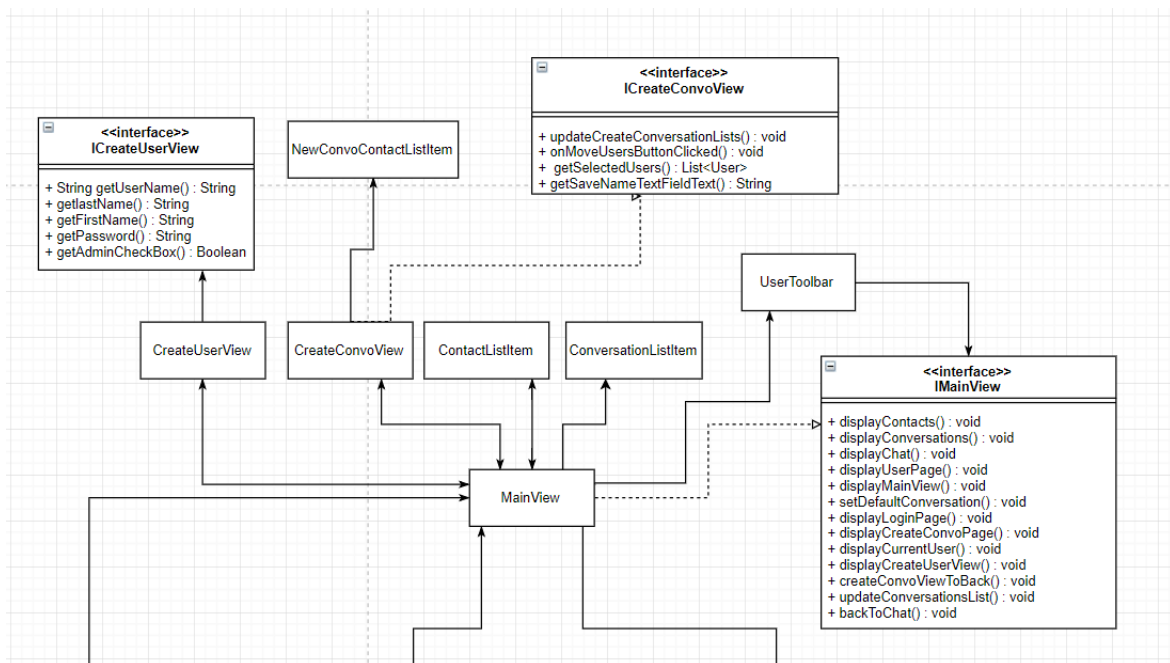


Figure 7: Upper part of the UML diagram for the View

## Diagrams



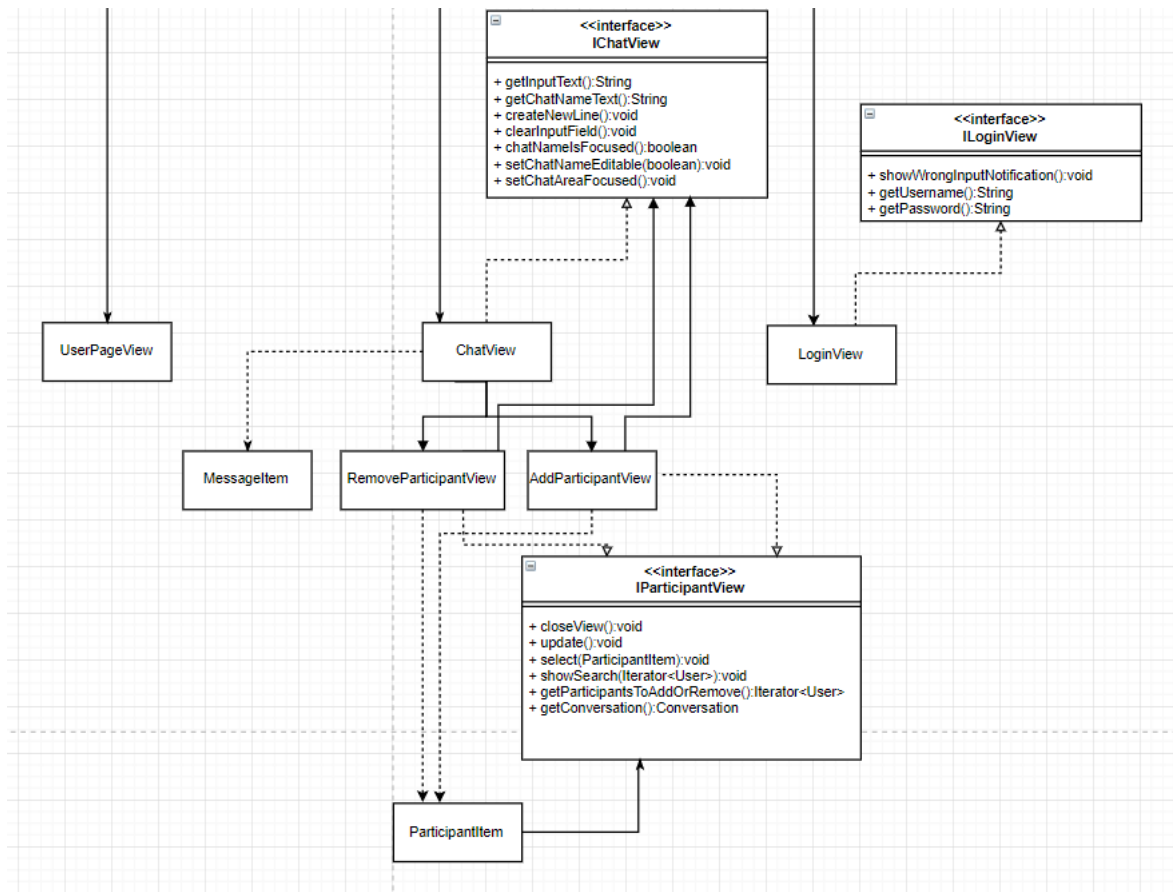


Figure 8: Lower part of the UML diagram for the View

- Dependencies  
The dependencies have been analyzed using STAN and can be viewed in Figure 6.
- UML sequence diagram  
An UML sequence diagram can be viewed in Figure 9. The diagram depicts the process of Logging in.

## Quality

- Testing  
No testing is done for the view since it is mostly comprised of JavaFX which is presumed to work.
- Known issues  
If no conversations are found on startup, the ChatView will display "Placeholder" as name of the conversation. The application will also crash when trying to remove participants, add participants or leave conversation.

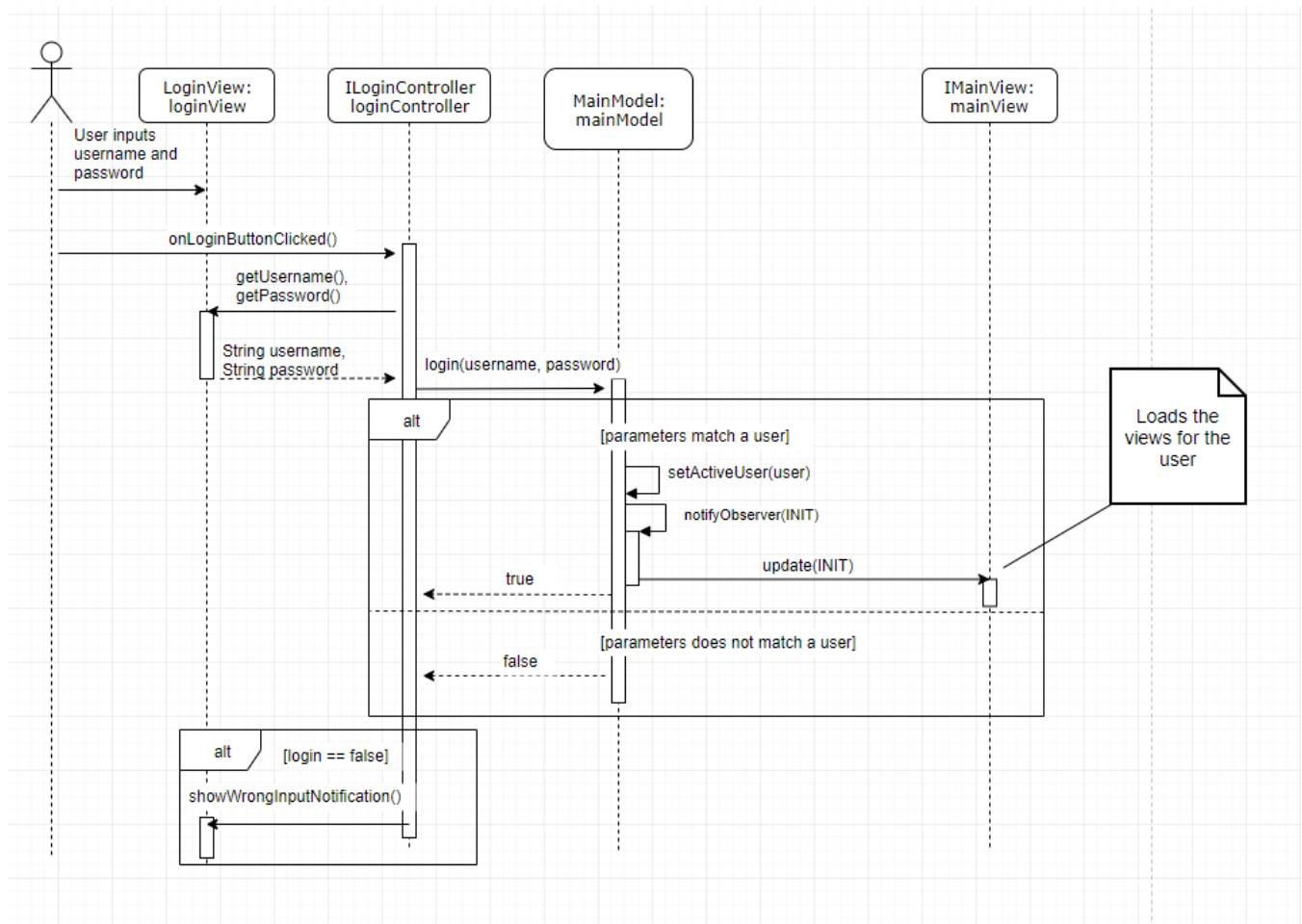


Figure 9: A sequence diagram for Logging in

## 2.4 Infrastructure

The infrastructure package is responsible for handling the data of ShatApp. It loads the data on application start-up, and saves the data continuously throughout the usage of the app. The data is stored in JSON.

- MVC implementation  
The JsonSaver could be considered a View since its main function is to "present" the Model in JSON. MVC is not otherwise applicable to this package.
- Design model  
JsonSaver implements modelObserver and IDataSaver, and JsonLoader implements the IDataLoader interface. IDataLoader contains methods needed to load data, and the JsonLoader implementation loads from JSON. The JsonSaver is an observer of the model, and it saves the relevant data in the model every time an update is called.

- Design patterns  
Observer pattern is used and JsonSaver implements the ModelObserver interface. It is an observer of the model, and it writes the conversations and users from the model when an update is fired.
- A class diagram for the design model can be viewed in Figure 10.

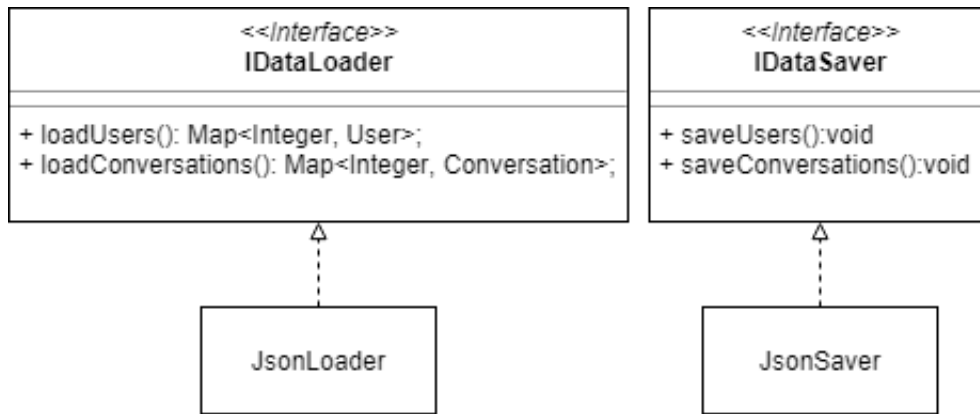


Figure 10: Infrastructure class diagram

## Diagrams

- Dependencies  
Dependencies have been analyzed using STAN and can be viewed in Figure 11.

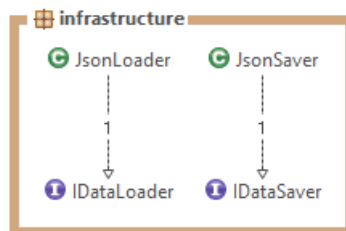


Figure 11: A dependency analysis of the Infrastructure package created using STAN.

- UML sequence diagram. An UML sequence diagram can be viewed in Figure 15.

## Quality

- List of tests: The tests are in `src/test/infrastructure` and they are run with coverage when running `"gradlew test jacocoTestReport"`. The coverage result can be viewed in figure 12. The tests are comprehensive, but the coverage is not full since JaCoCo cannot test Exceptions[4].

## infrastructure






Element	Missed Instructions	Cov.	Missed Branches	Cov.
<a href="#">JsonLoader</a>		90 %		83 %
<a href="#">JsonSaver</a>		92 %		n/a
<a href="#">JsonLoader.new TypeToken().{...}</a>		100 %		n/a
<a href="#">JsonLoader.new TypeToken().{...}</a>		100 %		n/a
Total	22 of 256	91 %	2 of 12	83 %

Figure 12: The JaCoCo coverage report for the infrastructure package.

- Known issues  
All errors detected with Pmd quickstart ruleset[3], were fixed. The quickstart ruleset contains many generally applicable rules.

## 2.5 Controller

The controller package listens to the view and acts based on user input and makes the view update whenever the model has been updated.

- MVC implementation  
The controller listens to the view and tells the model to change data when the user affects data with his/her actions. The controller also tells the view to alter itself in some cases where to user interacts with the controller.
- Design model  
Each view creates its own controller using the factory interface and calling the relevant method which creates and returns a controller of desired type. When the controller is created, the view links its buttons and various key inputs to the controller. After the links have been made, the controller is not saved by the view, meaning that it has a dependency on the controller rather than an association. This dependency is only on the interface of the controller, since it is the factory which has the associations with the controllers themselves. The factory itself also has an interface. Thus making the program more flexible.

The interfaces in the controller avoid using JavaFX, to make it possible for them to be used by other frameworks. However, some of the interfaces still use the JavaFX KeyEvent. This is because no other class was found that could hold user input as well as this class.

- Design Patterns Factory pattern: A factory pattern has been implemented, the different views use it to create their respective controls.
- Dependencies

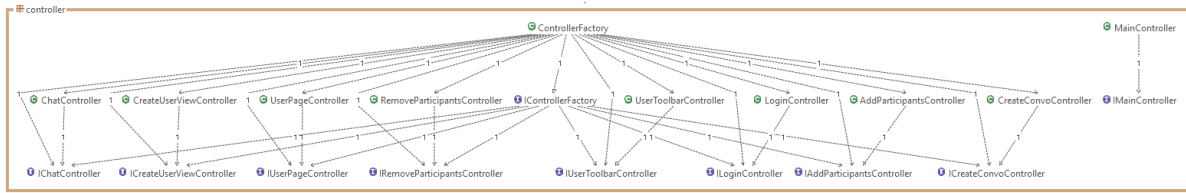


Figure 13: The Controller package analyzed using STAN

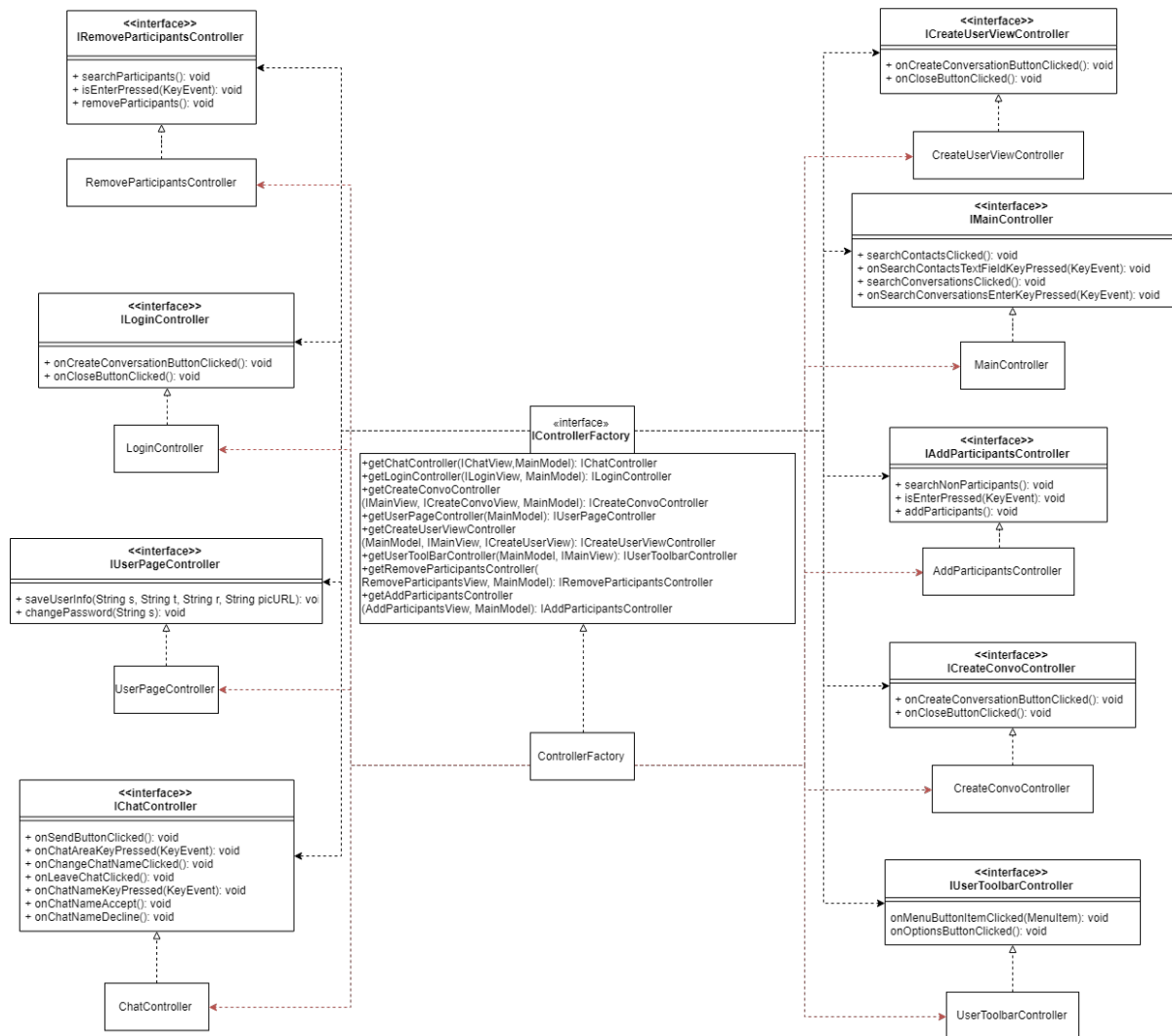


Figure 14: The Controller package described with UML

- UML: Can be seen in figure 14.

Quality

- Known issues  
Tests have not been made on the controller as it would require starting a graphical component.

### 3 Persistent data management

- Data handlers  
The application serializes the data from the model into JSON using Gson to store it. It also uses Gson to deserialize the JSON into objects again. This is done by two classes, JsonSaver and JsonLoader, implementing the interfaces, "IDataSaver" respectively "IDataLoader". The JsonSaver, JsonLoader, IDataSaver and IDataLoader as well as the JSON files reside in the Infrastructure package. Users are saved into users.json and conversations to conversations.json. Since the Conversation holds a list of Message these need not be saved separately, instead the list is simply serialized as is.
- Resource files  
There is a resource package in the main package called "resources" which houses the fxml-documents used for the view in a package called "fxml". The pictures used by the application by default are also stored in resources, in the package "pics".

### 4 Access control and security

Every User currently has a boolean "isManager" which indicates if the user is a manager or not. If the user is a manager it can create new Users which are added to the Model.

### 5 References

#### References

- [1] M. G. . C. KG and Contributors, "JaCoCo," <https://www.eclemma.org/jacoco/>, [Online; accessed 2018-10-20].
- [2] P. O. S. Project., "Pmd," <https://pmd.github.io/>, [Online; accessed 2018-10-20].
- [3] —, "Pmd Java Rules," [https://pmd.github.io/pmd-6.8.0/pmd\\_rules\\_java.html](https://pmd.github.io/pmd-6.8.0/pmd_rules_java.html), [Online; accessed 2018-10-20].
- [4] M. G. . C. KG and Contributors, "JaCoCo FAQ," <https://www.eclemma.org/jacoco/trunk/doc/faq.html>, [Online; accessed 2018-10-20].

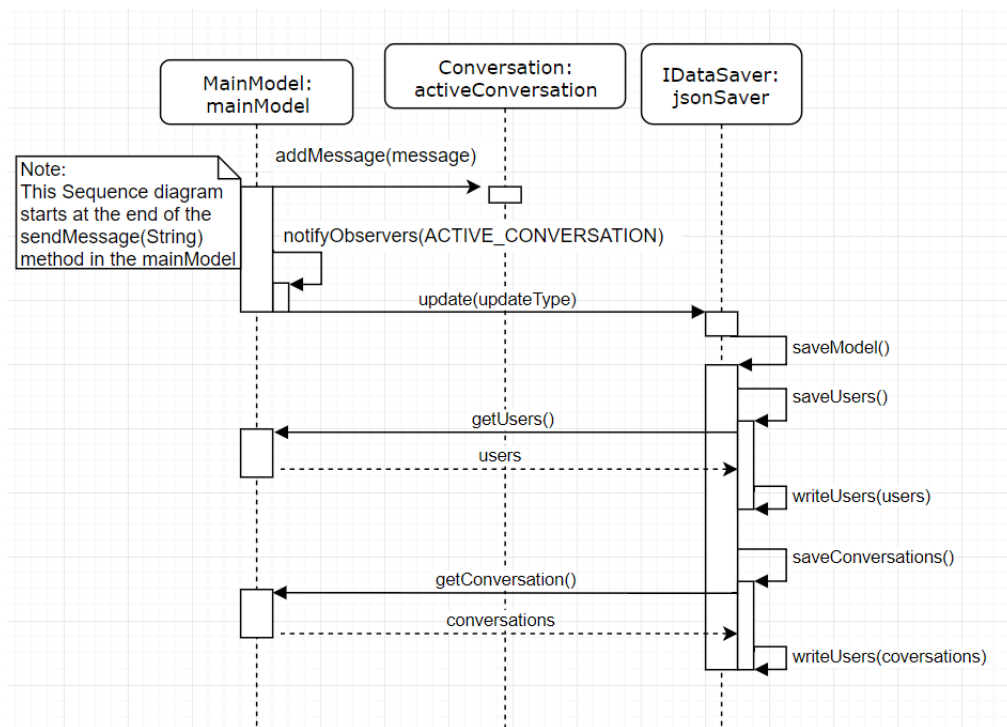


Figure 15: A sequence diagram of saving a message.