

Report for group 19, ShatApp

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

2018-10-28

version 1

1 Introduction

For years corporations have been using normal, ordinary, messaging applications in their day to day operations. This is not practical nor optimized for business flow. Basta_gruppens_LTD's messaging application takes care of those problems. The app makes it easy to communicate with other colleagues as well as starting multiple conversations for all your discussions and topics. All messages sent through the application will also be stored. Contacts and conversations can easily be searched for and created at the users demand. With companies being the applications main focus, the communication between colleagues will be seamless.

1.1 Definitions, acronyms, and abbreviations

- ShatApp is the name of our application.
- ShatApp in the UML: the Main class.
- Conversation in the UML: a class that holds information relevant to the conversation.
- User in the UML: the User class holds information about the user; such as: userId, name, profession etc.
- Message in the UML: the Message class represents a message. It has a date, a sender, a conversation id, the contained message.
- User information is the information that the User class holds. For example contacts, username, firstname, lastname etc.
- Classes are marked with typewriter font, for example JsonSaver

- JSON - JavaScript Object Notation. The format chosen for storing data.
- Gson[1] - a library that serializes and deserializes objects to and from JSON.
- JaCoCo[2] - Java Code Coverage. A code coverage library developed by the EclEmma team.
- Pmd[3] - a static code analyzer. It finds issues in the code based on a specified ruleset.
- JsonSaver is a class in the Infrastructure package that is used to save data into Json. JsonSaver is an observer of the MainModel, and when the MainModel calls update on it, the current state of the data of the MainModel is saved. This currently includes the conversations and users that the model holds.
- STAN - a structure analysis tool for Java. Produces dependency diagrams for Java projects.

2 Requirements

2.1 User Stories

The user stories are sorted by order of implementation.

2.1.1 Starting the program

Story Identifier: MSG001

Story Name: Starting the program

Description

As a user, I want to be able to start the program, so that I can use it.

Confirmation

Functional

Can I start the program?

Does a GUI show up upon starting the program?

Non-functional

Availability: Can I start the program at any time during the day?

Usability: Is it easy to start the program?

2.1.2 Sending messages

Story Identifier: MSG002

Story Name: Sending messages

Description

As a user I want to be able to type and send a message because I want to contact my colleagues without having to walk over to their desk.

Confirmation**Functional**

Can I press a button to send the message?

Is there an input field for me to type the message into?

Does my sent message get saved somewhere?

Does the text in the input field get removed when you press send?

Can I use the enter key to send the message?

Can I press shift + enter to create a new line on my message?

Non-functional

Availability: Can I send messages at any time during the day?

2.1.3 Loading messages

Story Identifier: MSG003

Story Name: Loading messages

Description

As a user, I want to be able to load messages into the chat window to be able to see what has been sent.

Confirmation**Functional**

Can i view the message(s) that have been sent?

Only messages from the given conversation are to be loaded, and all messages from the given conversation are to be loaded.

Non-functional

Availability: Can I view the messages that have been sent to me at any time during the day?

2.1.4 Create a conversation between two users

Story Identifier: MSG004

Story Name: Create a conversation between two users

Description

As a user, I want to be able to create a conversation so that I can send messages to my colleagues.

Confirmation**Functional**

Can I press a button to initiate a new conversation with a user of my choice?

2.1.5 Viewing contacts

Story Identifier: GUI001

Story Name: Viewing contactants

Description

As a user I want to be able to see my contacts so that I know who I can contact.

Confirmation**Functional**

Is there a dedicated place in the GUI for me to view my contacts?

Can I view only my own contacts?

Are my contacts loaded into the dedicated place in the application for contacts?

Non-functional

Security: can other users see my contacts?

2.1.6 Differentiating between Contacts and Conversations

Story Identifier: GUI002

Story Name: Differentiating between Contacts and Conversations

Description

As a user I want to be able to see my contacts and conversations separately in order to be able to distinguish between the two of them.

Confirmation**Functional**

Is there a tab that lists conversations?

Is there a tab that lists Contacts?

Is it possible to switch between the tabs?

Non-functional

Response time: Do Conversations and Contacts load within a reasonable time?

2.1.7 Login

Story Identifier: GUI003

Story Name: Login

Description

As a user, I want to be able to log in to the application so that other users can identify me.

Confirmation**Functional**

Is there a GUI for logging in?

Does entering the correct username and password allow the user to proceed to the application?

If the incorrect username and/or password is entered, is an error message displayed?

Non-functional

Security: Is it possible to guess a users login information based on the displayed error messages?

2.1.8 Change conversation name

Story Identifier: DATA001

Story Name: Change conversation name

Description

As a user I want to be able to change the name of a conversation because it's easier to tell the conversations apart.

Confirmation

Functional

Can I press a button that lets me change the name of conversation?

Can I press the conversation name, in order to change the name of the conversation?

Is there a limit on how long the conversation name can be?

2.1.9 Saving Conversations

Story Identifier: DATA002

Story Name: Saving Conversations

Description

As a User, I want my conversations to be saved, so that I can remember what has been written between me and my contacts.

Confirmation

Functional

Are my conversations saved when they are created?

If i close the program and then open it up again, can I see what messages have been sent and what conversations I am a part of?

Non-functional

Data integrity: Is the saved information valid and consistent over it's lifetime?

2.1.10 Saving User information

Story Identifier: DATA003

Story Name: Saving User Information

Description

As a user, I want my information to be saved, so that I don't have to input it again every time i start the application.

Confirmation

Functional

Is my user information stored when it is manipulated?

If I close the program and then open it up again, is my user information the same as when I closed the program?

Non-functional

Data integrity: Is the saved information valid and consistent over its lifetime?

2.1.11 Create a conversation with a group of contacts

Story Identifier: MSG005

Story Name: Create a conversation with a group of contacts

Description

As a user, I want to be able to create a conversation group of multiple contacts so that I can keep in contact with a group of people at the same time.

Confirmation

Functional

Can I initiate a conversation with multiple people?

2.1.12 Log out

Story Identifier: GUI004

Story Name: Log out

Description

As a user, I want to be able to log out so that other people can log in on my system.

Confirmation

Functional

Is there a GUI for logging out?

If I log out, can someone else then log in?

2.1.13 Changing Status

Story Identifier: DATA004

Story Name: Changing Status

Description

As a user I want to be able to change my current status so that my colleagues can see if I'm busy/in a meeting/active.

Confirmation

Functional

Is there a GUI for changing my status?

Can I change my status?

Can other users see my status?

2.1.14 View amount of participants in a conversation

Story Identifier: MSG006

Story Name: View amount of participants in a conversation

Description

As a user I want to be able to view the amount of participants in a conversation so that I can see how many people I'm sending messages to.

Confirmation**Functional**

Can I see the amount of participants?

2.1.15 Choose profile picture

Story Identifier: DATA005

Story Name: Choose profile picture

Description

As a user, I want to be able to choose a profile picture so that people can easily identify me

Confirmation**Functional**

Can I choose a profile picture?

Can I change my profile picture?

Non-functional

Usability: Is it possible to use different formats of pictures?

Data integrity: Is the saved picture valid and consistent over its lifetime?

2.1.16 Edit user info

Story Identifier: DATA006

Story Name: Edit user info

Description

As a user, I want to be able to edit my user information and profile picture so that colleagues can recognize my account

Confirmation**Functional**

Can I view my user info?

Can I update my user info?

Is my updated info saved?

Non-functional

Data integrity: Is the saved user information valid and consistent over its lifetime?

2.1.17 Remove users from conversation

Story Identifier: CONV001

Story Name: Remove users from conversation

Description

As a user, I want to be able to remove people from a conversation, so that they no longer have access to the conversation.

Confirmation**Functional**

Is there a place where I can see the participants of a conversation?

Is there an option to remove participants of the conversation?

Can the removed user access the conversation?

2.1.18 Add users to a conversation

Story Identifier: CONV002

Story Name: Add users to a conversation

Description

As a user, I want to be able to add other users to a already existing conversation, so I don't have to create a new conversation with my desired participants.

Confirmation**Functional**

Is there a place where I can see who is not a participant of the conversation?

Is there an option to add those users to the conversation?

Can the added participant see the conversation?

2.1.19 Search function

Story Identifier: GUI005

Story Name: Search function

Description

As a user I want to be able to search for a contact so that I can initiate communication with a desired contact without have to look through my entire contact list

Confirmation**Functional**

Is there a searchbox where I can search for my conversations and contacts?

Can I easily see the search results?

Can I interact with the results of the search?

2.1.20 Search for either conversation or contact

Story Identifier: GUI006

Story Name: Search function

Description

As a user I want to be able to search for either the name of the conversation or the

names of the participant/s of the conversation, because I might forget what I named the conversation but remember with whom I talked to

Confirmation

Functional

Can I use the searchbox for this function?

Does the search function look at both contact names and conversation names?

2.1.21 Manager mode

Story Identifier: MNG001

Story Name: Manager mode

Description

As a Manager, I want to create accounts for my employees, so that they can access the chat application

Confirmation

Functional

Can I create accounts if I am a manager?

Non-functional

Security: Can I enter manager mode if I am not a manager?

2.1.22 Adding new contacts

Story Identifier: DATA007

Story Name: Adding new contacts

Description

As a user I want to be able to search for other users so that I can add them to my contacts.

Confirmation

Functional

Can I search for users that are not my contacts?

Can I add the users to my contact list?

2.1.23 Leave Conversation

Story Identifier: CONV003

Story Name: Leave active conversation

Description

As a user, I want to leave a conversation, because the conversation is not relevant anymore.

Confirmation

Functional

Is there a button I can press to leave the chat?

Is the chat no longer listed for me?

2.2 User interface

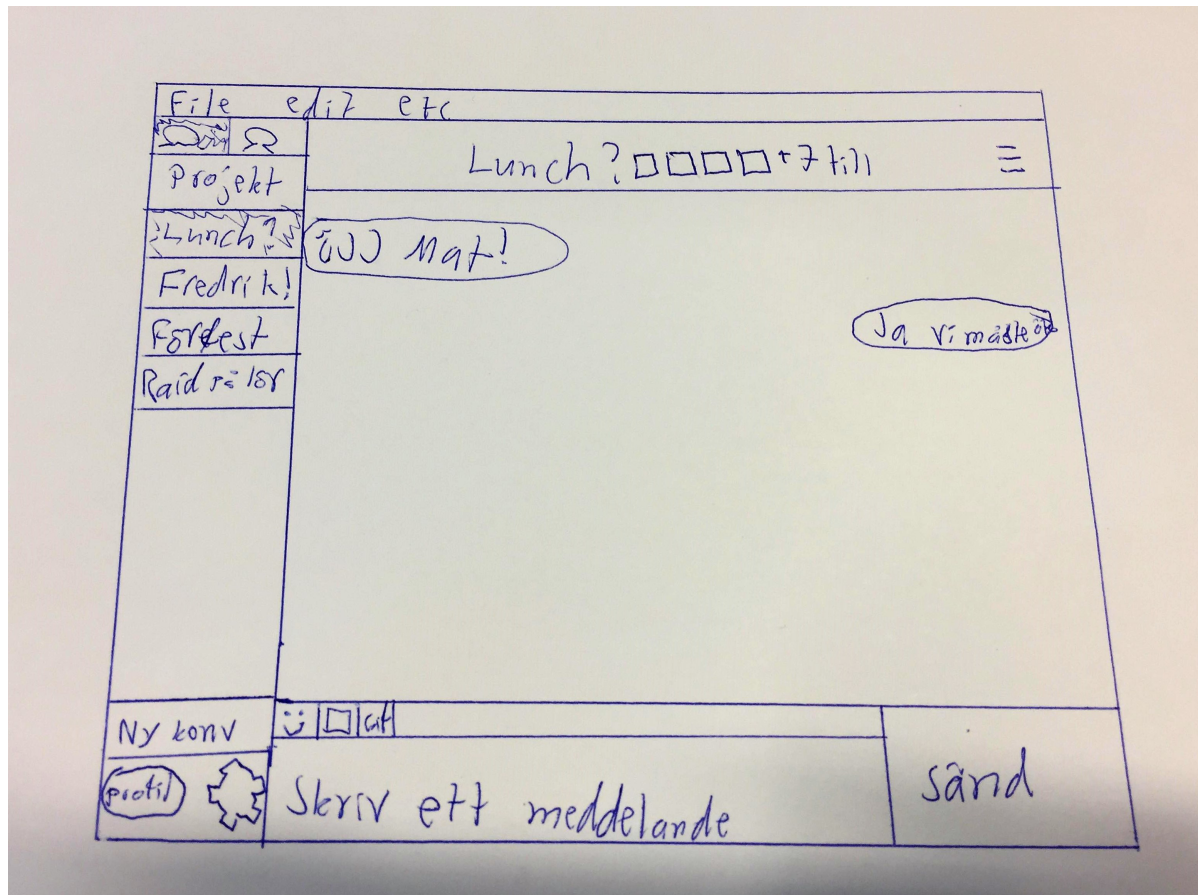


Figure 1: An early sketch of the GUI

3 Domain model

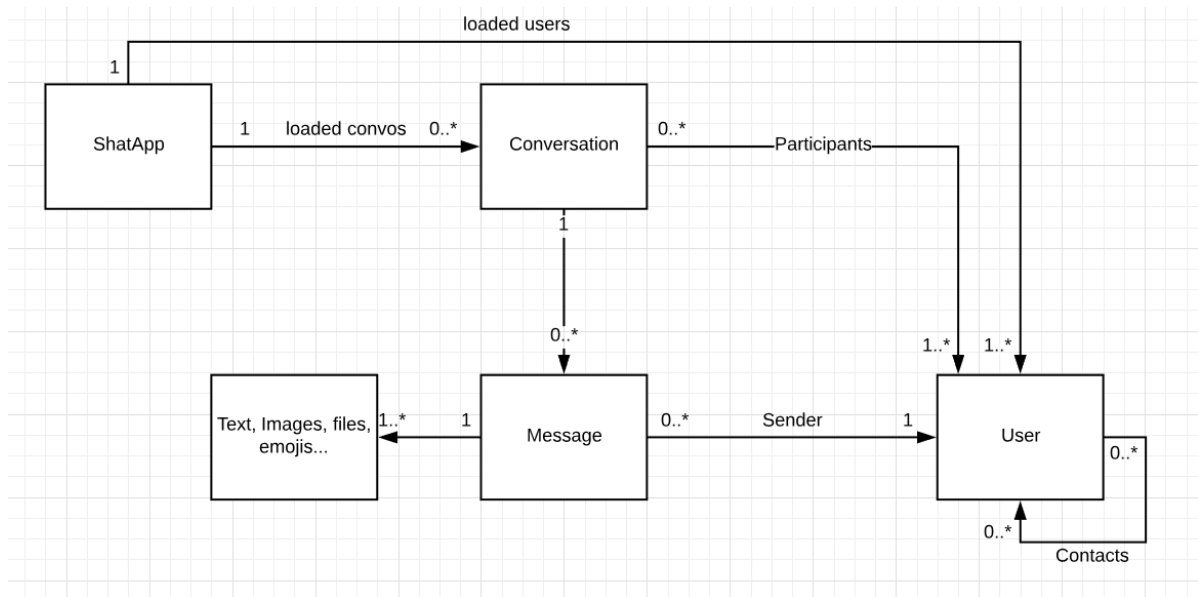


Figure 2: Domain model

3.1 Class responsibilities

Explanation of responsibilities of classes in the domain model diagram.

- **ShatApp** ShatApp is the core of the application. It contains the data and logic for how the application is going to function.
- **Conversation** Conversation will keep and hold all the necessary information required in a conversation. This includes, but is not limited to: participants, messages, conversation id.
- **User** User contains information about a user; such as userId, firstName, last-Name etc.
- **Message** Message will keep the content of the message, as well as meta data such as time sent, and sender.

4 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 application

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. The relationships between the model, view and controller can be seen in Figure 3 and 4

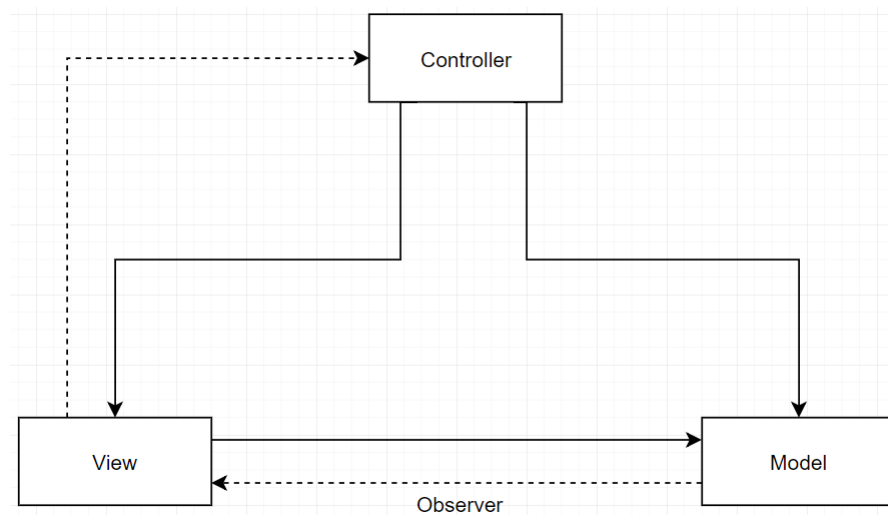


Figure 3: A system overview that shows how MVC is implemented.

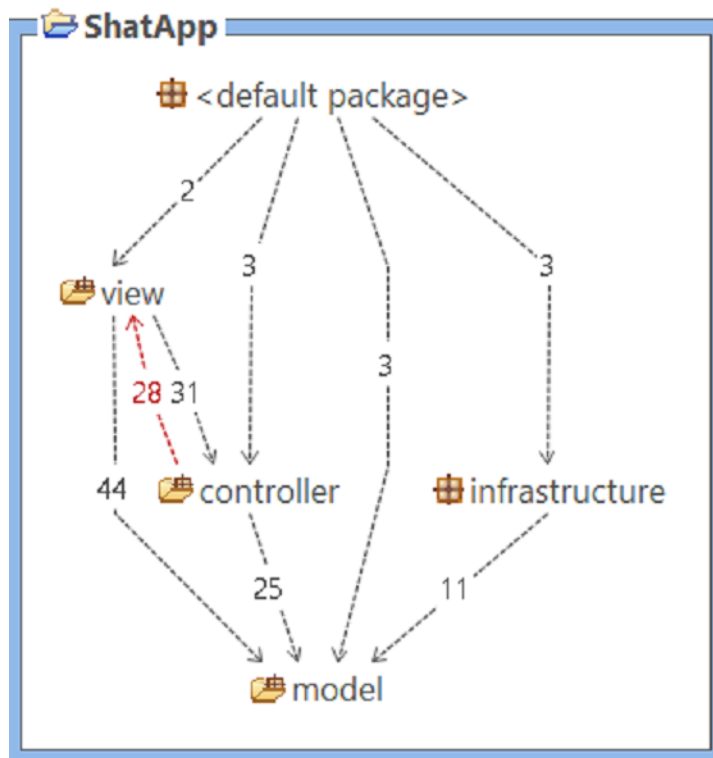


Figure 4: An overview of the system, note that the red dependencies from controller to view all are interfaces

4.1 Subsystem decomposition

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

4.2 Model

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

- 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, the view and JsonSaver, so that they can update themselves accordingly.
- Design model
The class diagram for design model can be viewed in Figure 5.
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 made the classes much easier to store, and avoided creating an infinite loop in which the first user have 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 application, 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 increases the type safety of the method. The files found in this package is a `ModelObserver` interface, a `ModelObservable` class and an enumeration `UpdateType`, which defines the possible update types that can be sent from the `ModelObservable`.

- Design patterns

Iterator: Iterator is used in the application 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 can not 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

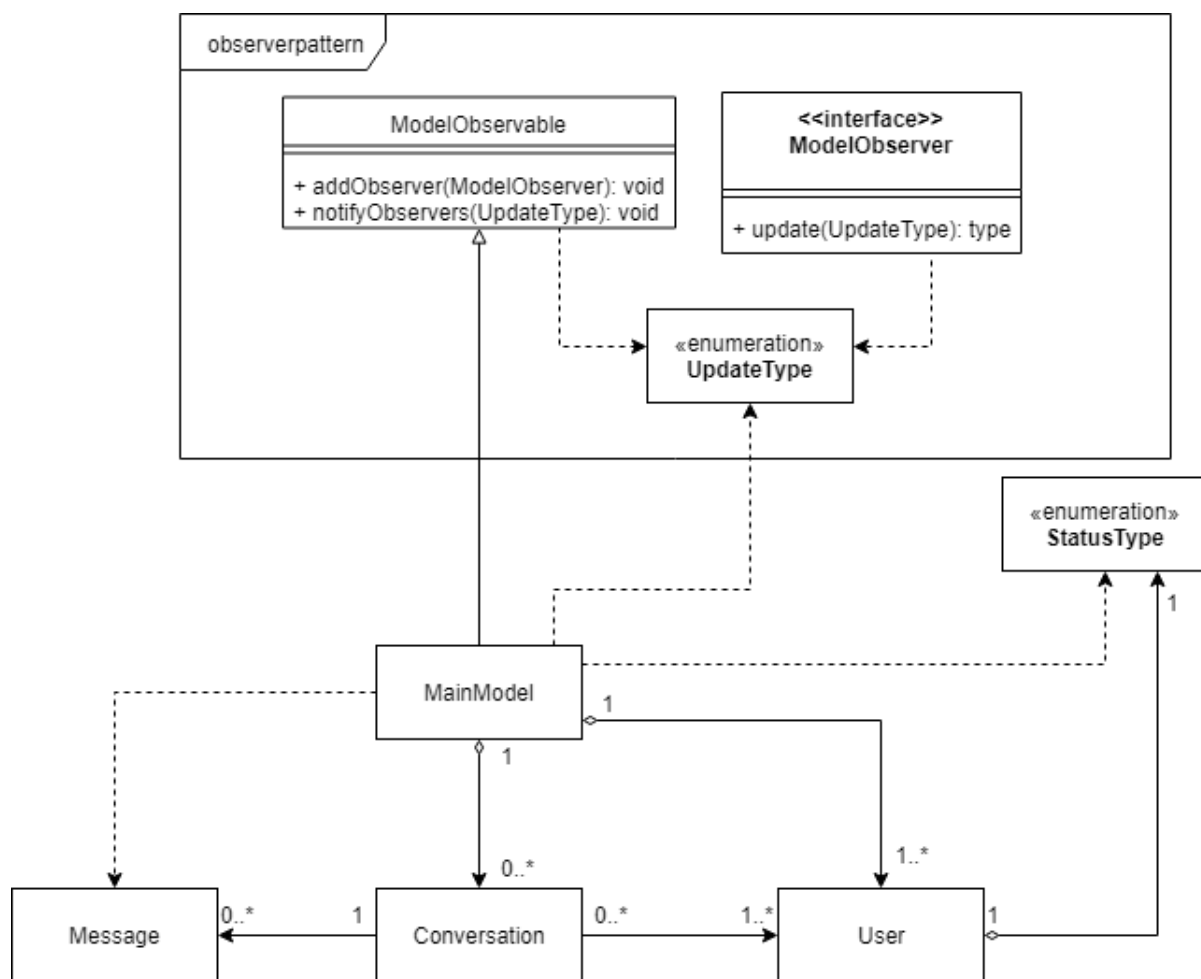


Figure 5: Class diagram for the model

use the `MainModel`, using simple method calls directly to the `MainModel` instead of using method chaining.

Diagrams

- Dependencies

A STAN diagram of the dependencies in the model can be seen in Figure 6.

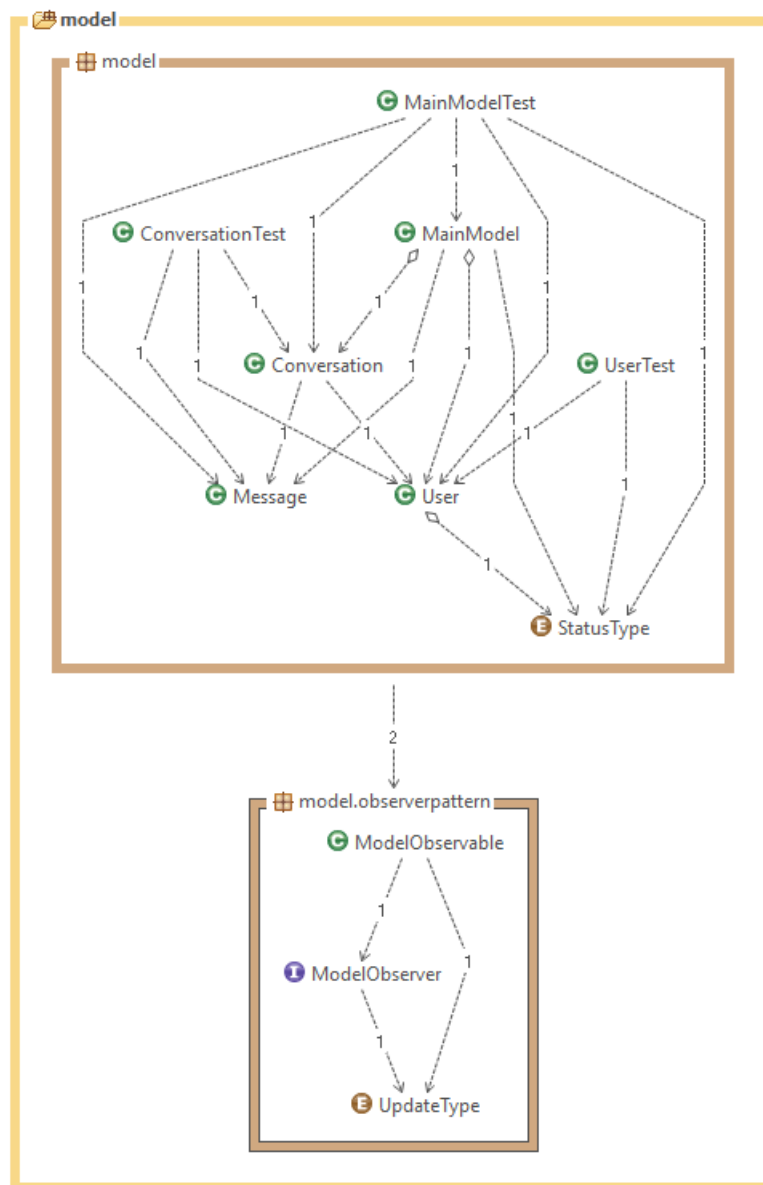


Figure 6: Dependency analysis using STAN on the Model package.

- UML sequence diagram

A UML sequence diagram can be viewed in Figure 7. The diagram depicts the process of sending a message and in it you can view the models part.

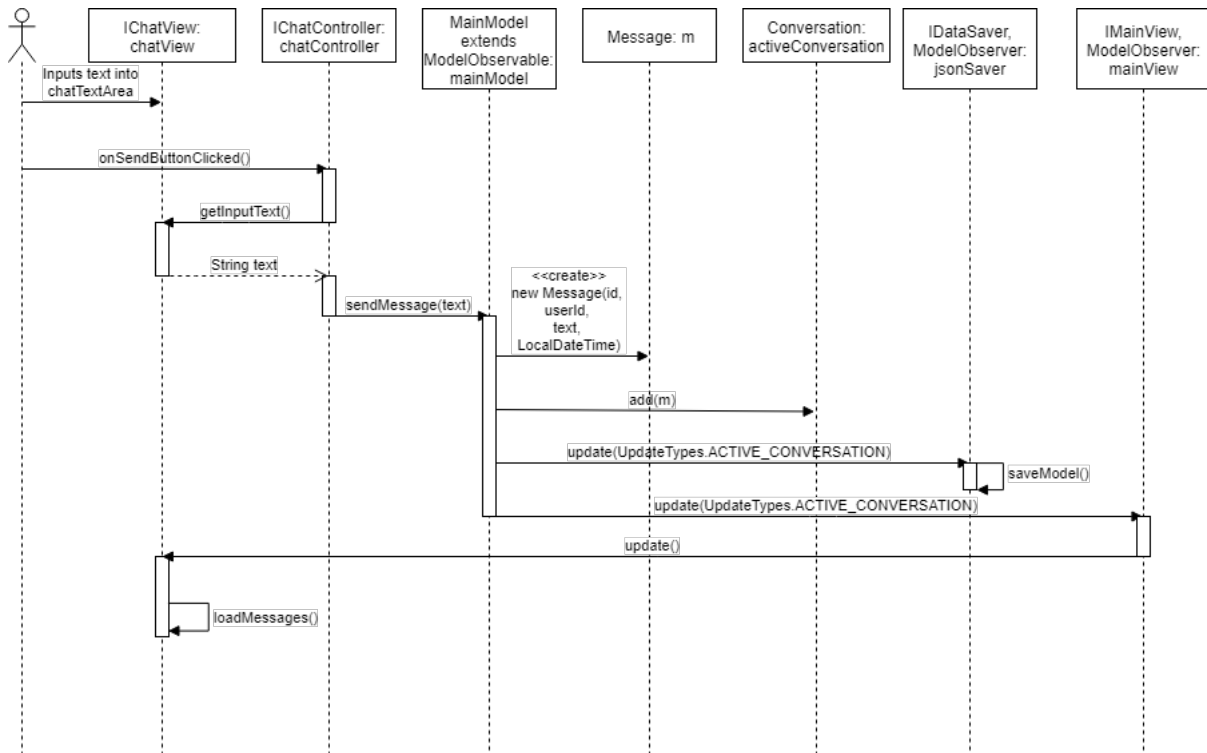


Figure 7: A sequence diagram for sending a message

Quality

- List of tests

The tests are in `src/test/model` and can be run with JaCoCo coverage using `gradlew test jacocoTestReport`, this is true for all tests. The JaCoCo report can be viewed in Figure 8. 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`. Setters and getters remain mostly untested, and they are the reason as to why the rest of the model doesn't have 100 percent coverage.

Pmd with quickstart ruleset[4] reports no issues. The quickstart ruleset contains many generally applicable rules.

model









Element	Missed Instructions	Cov.	Missed Branches	Cov.
User		89 %		66 %
MainModel		96 %		94 %
StatusType.new Object() {...}		0 %		n/a
StatusType		75 %		0 %
Message		67 %		n/a
Conversation		93 %		70 %
User.new Object() {...}		0 %		n/a
Total	112 of 1 407	92 %	26 of 138	81 %

Figure 8: A JaCoCo report for the model package.

- Known issues

There is currently no support for changing what kind of content there is in your messages. This could be solved by implementing interfaces for either the content in the message or the messages themselves, and then an enum that specifies the content or message type. This would make it easier to expand types of messages or contents in the future as well. There was however no time to implement it, mostly because of issues with serialization and deserialization of interfaces.

4.3 View

The view is responsible for displaying the data that the model holds to the user. An overview of the package can be viewed in the STAN analysis at Figure 10.

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

- Design model

The UML diagram for view can be viewed in Figure 9.

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, ContactDetailView, CreateUserView 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

the MainView being a large god object.

The ChatView loads messages from the MainModel and uses the 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 11. After this, the rest of the view is initialized.

The UserToolBar displays info and ways to edit the current user, such as a drop-down 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 of the MainModel. The reason for having multiple update types is so that the view only updates relevant info.

The view package holds a lot of interfaces, and these interfaces are not meant for MainView to have an association to. 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 implemented differently, 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.

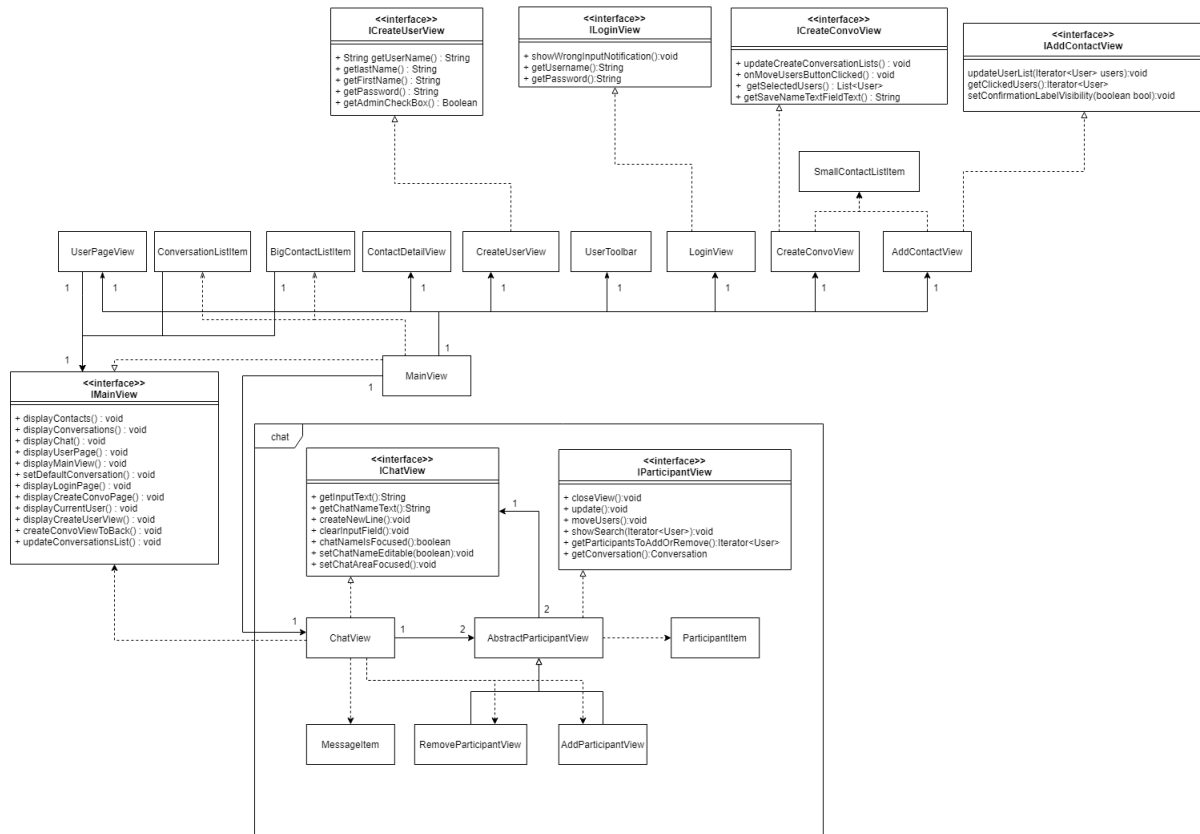


Figure 9: The UML diagram for the view package

- Design patterns

The observer pattern is implemented: the view is an observer of the MainModel, and updates itself when the model is modified.

Strategy pattern: Used in the AbstractParticipantView. The views extending this class bind themselves to an IParticipantsController, but there are different types of participant controllers, and their methods act different from one another.

Diagrams

- Dependencies

The dependencies have been analyzed using STAN and can be viewed in Figure 10. The double dependency between the view.chat and view packages is caused by a dependency on an interface, and is therefore not a hard dependency. Creating a separate package for the chat was considered to increase the modularity, maintainability and comprehensibility of the application and the interface dependency was therefore considered negligible.

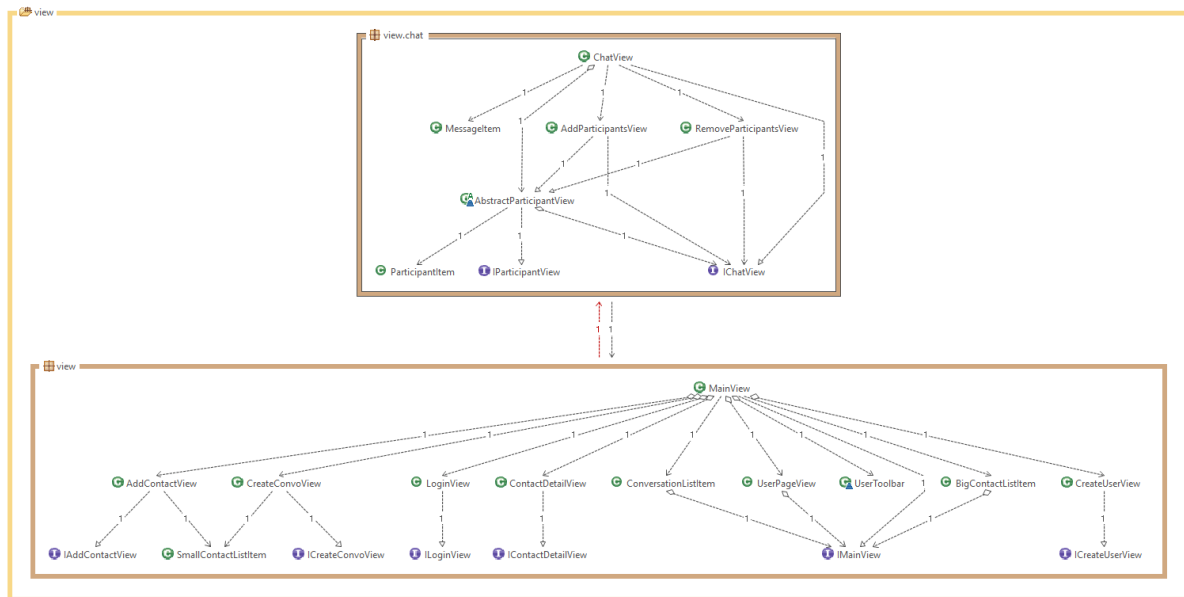


Figure 10: View package dependency analysis using STAN.

- UML sequence diagram
A UML sequence diagram can be viewed in Figure 11. The diagram depicts the process of logging in.

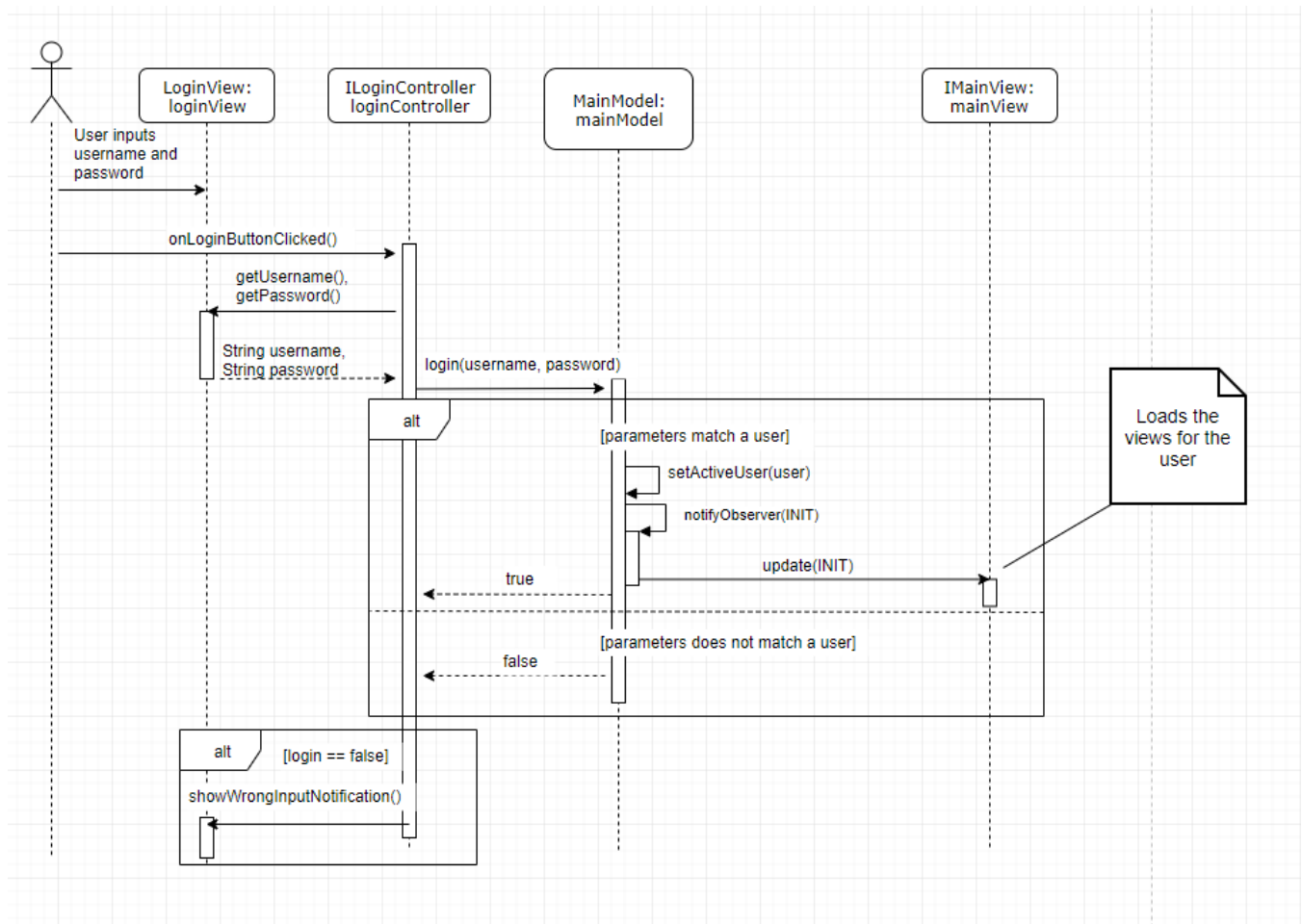


Figure 11: A sequence diagram for logging in

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 application will cast a `NullPointerException` when trying to remove participants, add participants or leave the conversation.

When creating a conversation with only one person, the image for the conversation will be of the other users face. This will however also translate over to what the other user will see. Basically the image for the conversation will be set

globally for all users.

4.4 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 the user interacts with the controller.
- Design model
Main creates a `controllerFactory` and a controller for `MainView`. The factory is sent as an argument to the `MainView` and the controller is set using a `bindController` method. The `MainView` then creates most of the other views, their controllers and binds those controllers. Each `bindController` links the event handlers of that view 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 application 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

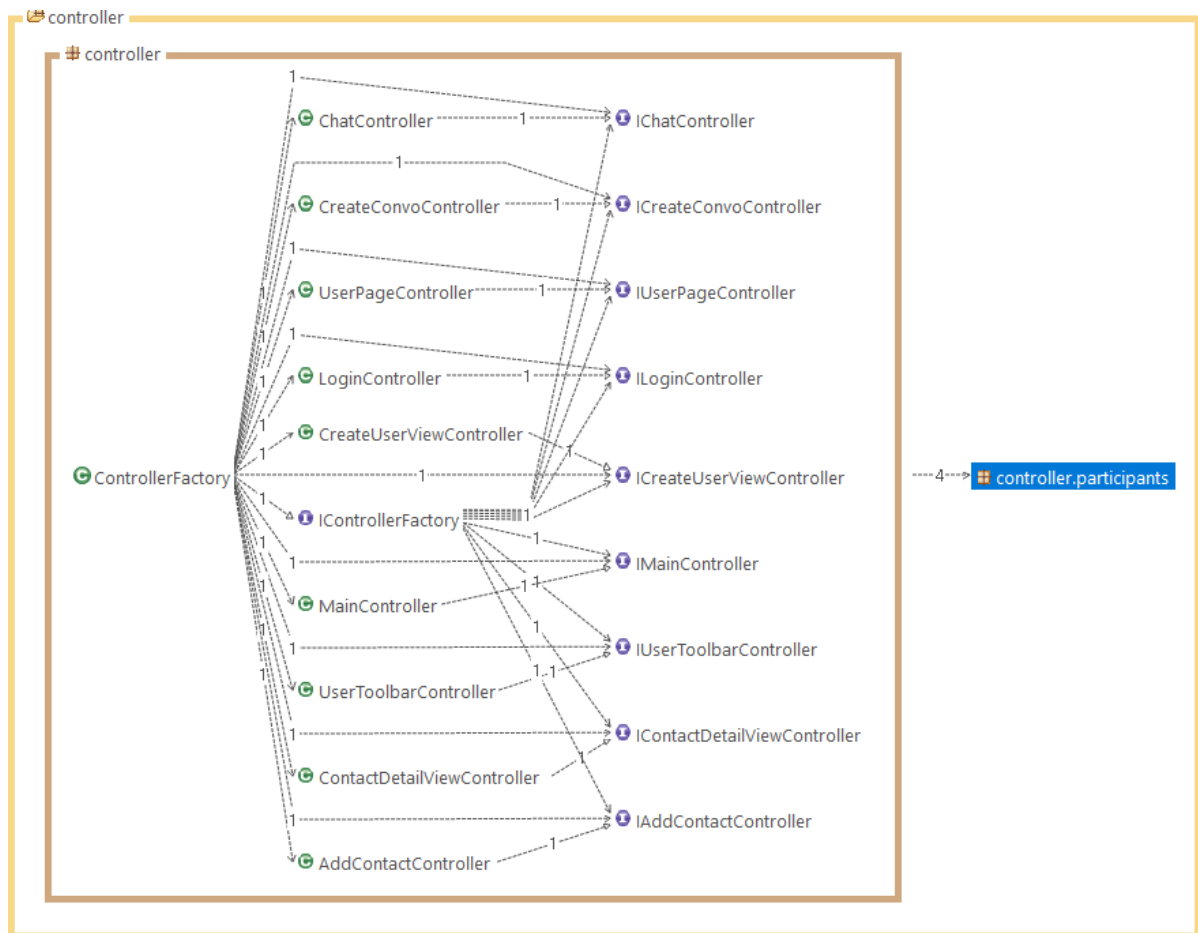


Figure 12: The Controller package analyzed using STAN

- UML: Can be seen in figure 14.

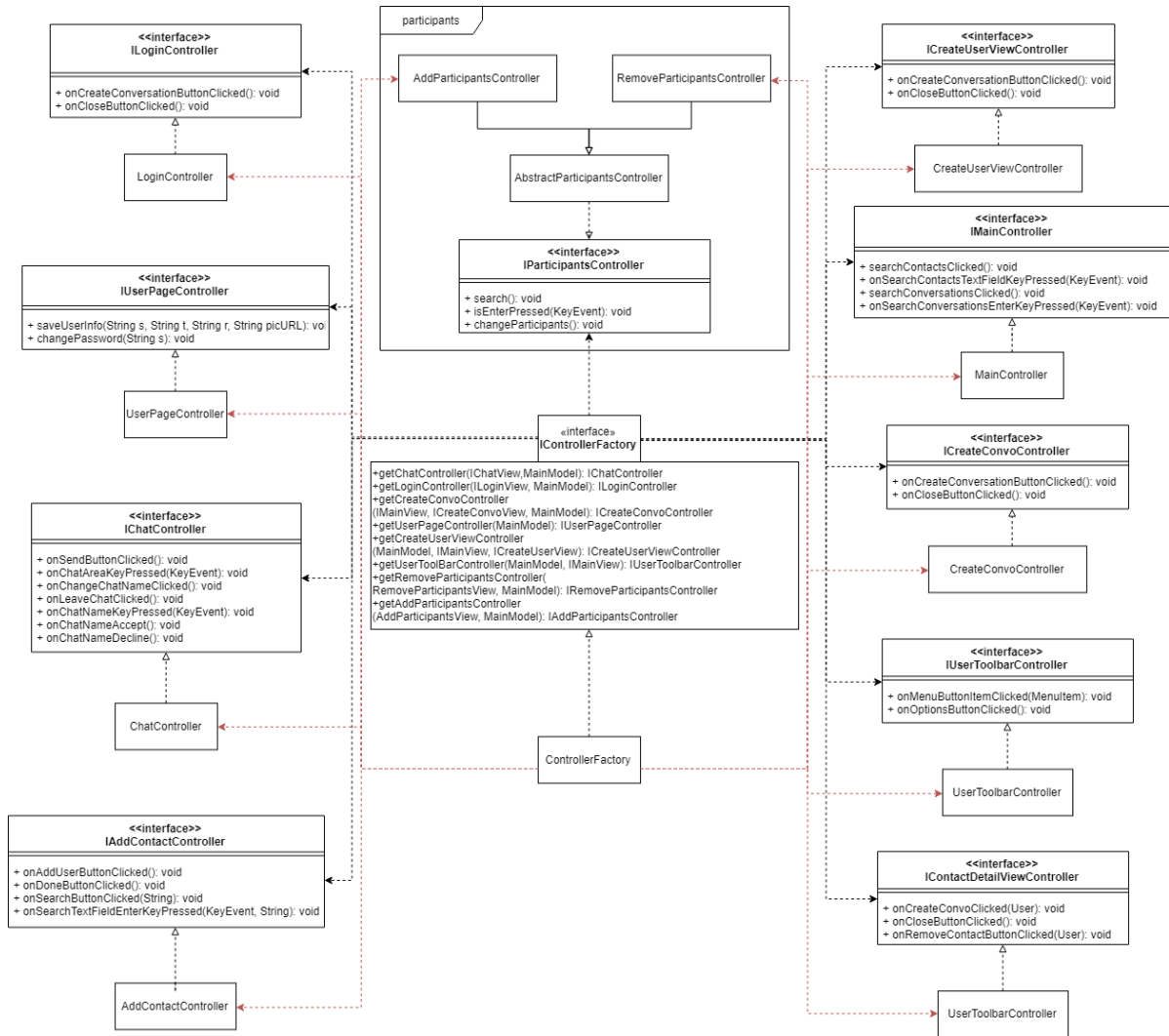


Figure 13: The Controller package described with UML

Quality

- Tests

Tests have not been written for the controller as it would require starting a graphical component.

4.5 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 us-

age 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. IDataSaver and IDataLoader contain methods needed to save and load data, and the JsonSaver and JsonLoader are concrete implementations that save and load to and from JSON. The JsonSaver observes the model, and it saves the relevant data from the model to JSON 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 14.

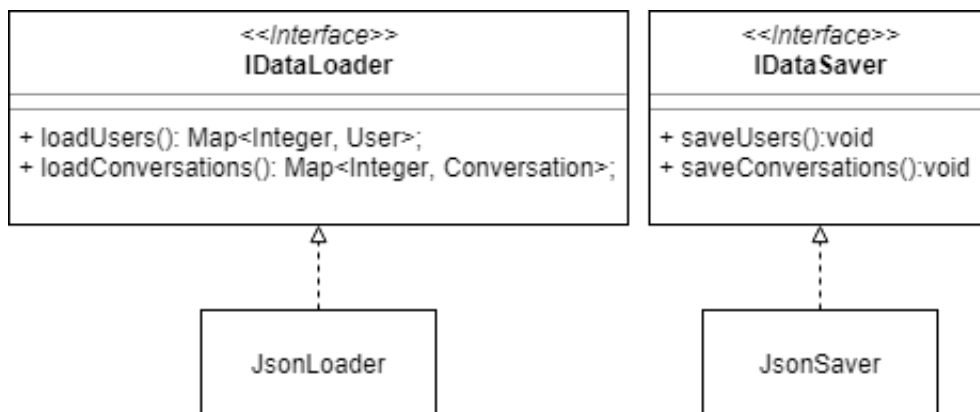


Figure 14: Infrastructure class diagram

Diagrams

- Dependencies

Dependencies have been analyzed using STAN and can be viewed in Figure 15.

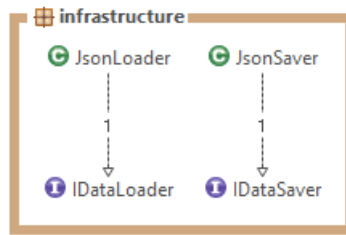


Figure 15: A dependency analysis of the Infrastructure package created using STAN.

- UML sequence diagram. A UML sequence diagram can be viewed in Figure 17, in chapter three.

Quality

- List of tests

The tests are found in `src/test/infrastructure` and they are run with coverage when running `gradlew test jacocoTestReport`. The coverage result can be viewed in figure 16. The tests are comprehensive, but the coverage is not full since JaCoCo cannot test Exceptions[5].

infrastructure

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

Figure 16: The JaCoCo coverage report for the infrastructure package.

- Known issues

The infrastructure can at the moment not handle interface serialization and de-serialization, and this created problems when trying to make messages more abstract in order to increase expandability.

5 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".

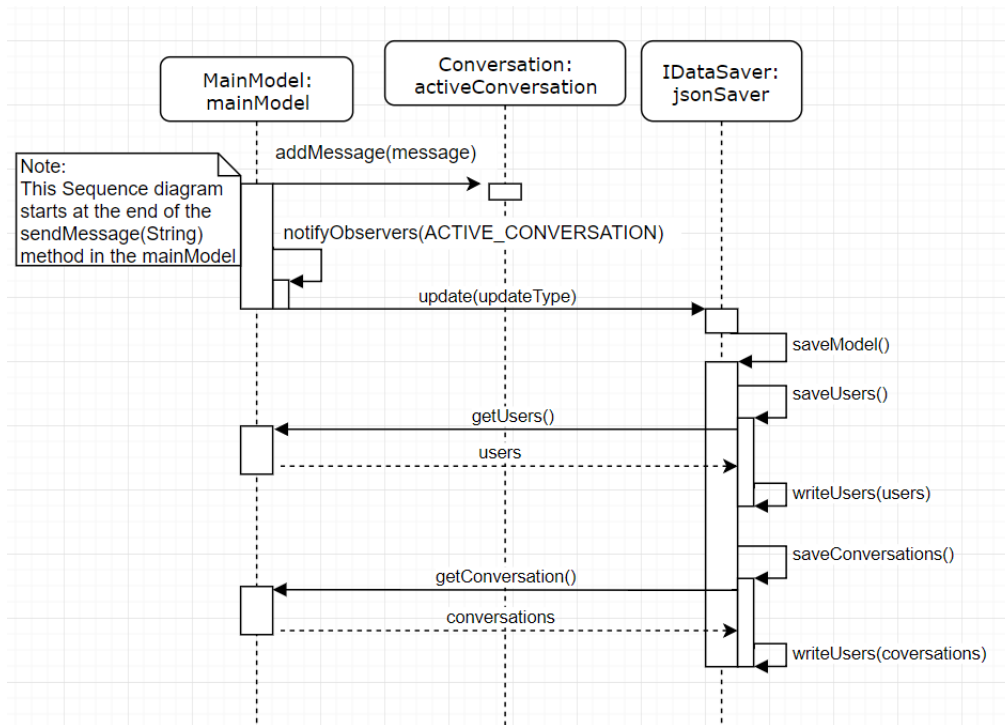


Figure 17: A sequence diagram of saving a message.

6 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. This could be improved upon, and each user could be given e.g. an enum that represents its access level. This would also be relevant since the application is used in a company setting, and there are usually hierarchies within companies. This was however not implemented since there was no time for it.

7 Peer review of group 20s project

- S.O.L.I.D design principles.
 - Single responsibility principle is not followed in e.g. `PaintController`. `ISaveLoad` and its concrete implementation also breaks this principle. It is however very well followed in other places, e.g. the pixel package.
 - Open-closed principle is not followed. E.g. the `ImageModel` instantiates all the tools, and therefore you cannot expand the toolset without modifying the `ImageModel`.
 - Liskov substitution principle is followed in the model. The most notable use being the classes that implement the `ITool` interface and the inherit from the `AbstractPaintTool`. These classes have very similar behaviour and can be substituted for each other.
 - The Interface Segregation Principle is generally followed, however in some classes it's broken. For instance, many tool-classes (e.g. `AbstractPaintTool`, `ZoomTool`, `EyeDropperTool`, etc) have unused methods from the `ITool` interface. Some tools (e.g. `ZoomTool` and `Eyedrop`) don't need the `updateSettings` method in `ITool` since these tools don't use the tool settings at all.
 - The Dependency Inversion principle is broken e.g. in `ImageModel` by having dependencies on all tools when it could depend on abstractions instead. It is also broken by creating the `ImageModel` in `PaintController`, it should depend on `IModel` instead. The view and controller are locked to only work with JavaFX. There are no interfaces for views nor controller which makes them irreplaceable.
- The code is generally consistent in coding style, but could use some reformatting and cleaning up. In many areas it does not follow java standard of code formatting (e.g. `PaintController.java:223`).
- The code is mostly not reusable, with the exception being the pixel package.
- It is easy to maintain the code since it is very well structured but could benefit from better comments. Parameters and methods are mostly given understandable names, however there are exceptions (e.g. variables in `StraightLineStrategy`). There are however classes which are hard to understand as to why they reside at certain places. Examples include the `IModel` interface in the tools package, the absence of a controller package, and `LayerListController` extending a `StackPane` but not residing in the view package.
- It is not very easy to add functionality to the application, but it is not very hard either. The structure of the controller makes it easy to understand how to add or remove a button and how to link it to a certain function. The `ITool` interface makes it easy to create new tools, but adding them to the application is cumbersome since they have to be added to `ImageModel` as well.

- We have detected implementations of Strategy pattern, Observer pattern, MVC and Template Method pattern.
- The documentation is inconsistent and alternates between javadoc and plain comments. There are header comments in some classes but not all of them. Some methods have short comments explaining functionality, but they could have been more in-depth. Most methods are completely lacking documentation, making them hard to understand. The lack of documentation also makes it hard to add more functionality.
- Proper names are mostly used, however there are a few discrepancies. The `ImageModelObserver` interface has a method `notifyObservers()`, which sounds like it would lie within an observable class. A better name would be `update` or something similar.
- The code is mostly well tested with good code coverage except for several missed branches in the model. There are also a few methods here and there that aren't tested at all. The methods that are tested are however well tested.
Missed testing branches in following classes, among others: `PaintColor`, `Pixel`, `SelectTool` (`drawSelectedToolArea()`, `onDrag()`..), `ImageModel` (`setToolShape()`, `clearLayer()`..) and `loadImage`.
- In general there were no performance issues. When using higher sizes and lower hardness the performance will, however, drop notably.
- The code is mostly easy to understand but placing classes into proper packages and adding more comments would make it easier. The application has a clear MVC structure with regards to how it works but the structure of classes in packages is unclear. There is no Controller package, and all parts of the view are not in the view package. The model package is very well isolated from the other parts of the application.
- The design is mostly very good, but by improving the abstractions it would be more modular and almost all issues found would be fixed.
The gradle wrapper does not work, and a local gradle installation is required to build and run the application. It would be beneficial if the gradle wrapper worked, since that would make the project easier to run "out of the box". Testing could be improved by using a code coverage library, for example JaCoCo. The coverage report would aid in ensuring full test coverage.

8 References

References

- [1] Google, "Gson," <https://github.com/google/gson>, [Online; accessed 2018-10-28].
- [2] M. G. . C. KG and Contributors, "JaCoCo," <https://www.eclemma.org/jacoco/>, [Online; accessed 2018-10-20].
- [3] P. O. S. Project., "Pmd," <https://pmd.github.io/>, [Online; accessed 2018-10-20].
- [4] —, "Pmd Java Rules," https://pmd.github.io/pmd-6.8.0/pmd_rules_java.html, [Online; accessed 2018-10-20].
- [5] M. G. . C. KG and Contributors, "JaCoCo FAQ," <https://www.eclemma.org/jacoco/trunk/doc/faq.html>, [Online; accessed 2018-10-20].