System design document for ShatApp

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

2018-10-01
version 1

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.

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 things related to infrastructure, 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.

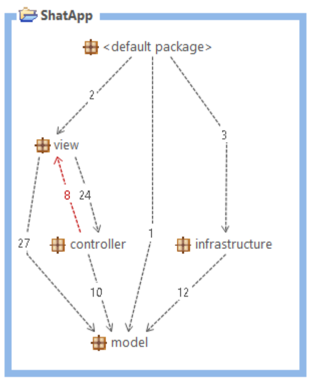


Figure 1: An overview of the system

2.1 Subsystem decomposition

Describe in this section each identified system component (that you have 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 a 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 just request it from the MainModel.

  The users and conversation that the MainModel holds are stored in HashMaps, for the reason that they are very easily searchable, since the key of the specific value is the same as the values id. 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 decide the outcome of most methods in the MainModel facade, such as the methods getContacts, and getUsersConversations.

- Design patterns
  Iterator: Iterator is used in the program when we just 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't 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 implements Observable and when it notifies observers it passes an enum, UpdateTypes, along with notifyObservers. The enum enables observers to interpret the modification to 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.

- Give a class diagram for the design model.

Diagrams

- Dependencies
  A STAN diagram of the dependencies in the Model can be seen in Figure 2.
- UML sequence diagram
  An UML sequence diagram can be viewed in Figure 3. The diagram depicts the process of sending a message and in it you can view the models part.

Quality

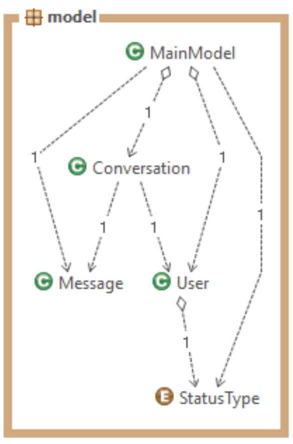


Figure 2: Dependency analysis using STAN on the Model package.

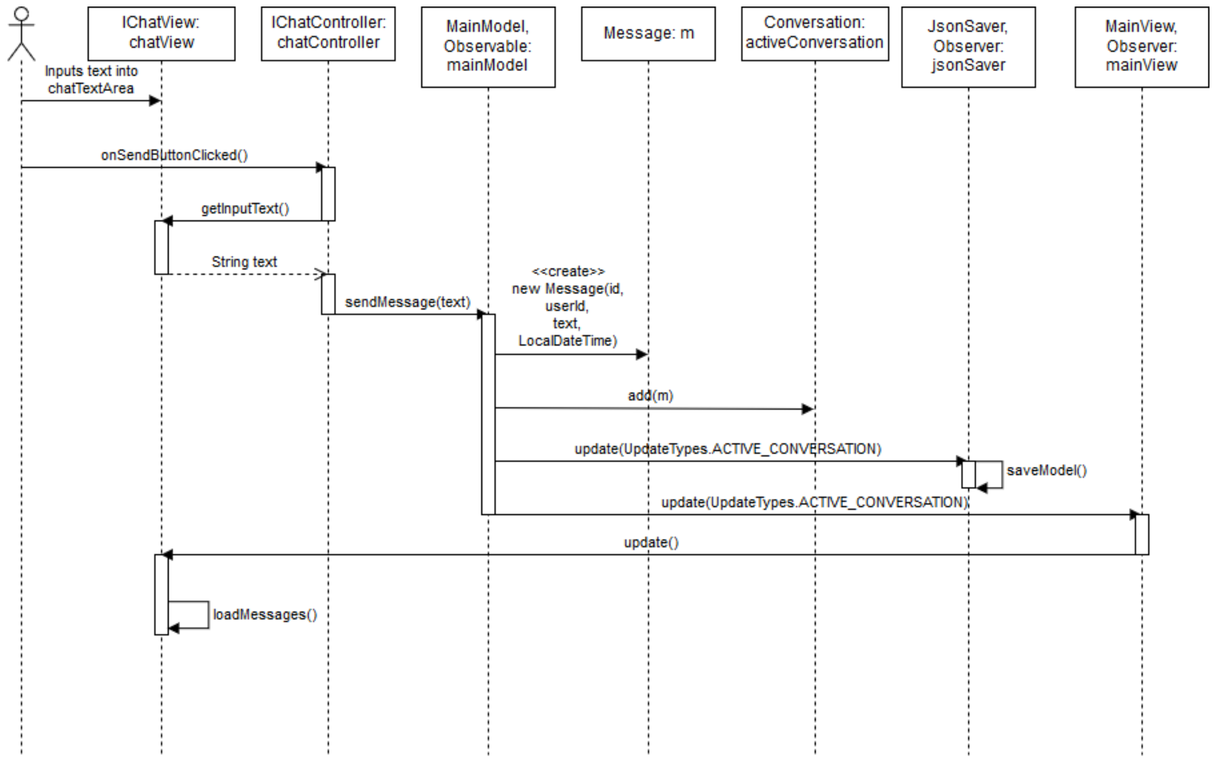
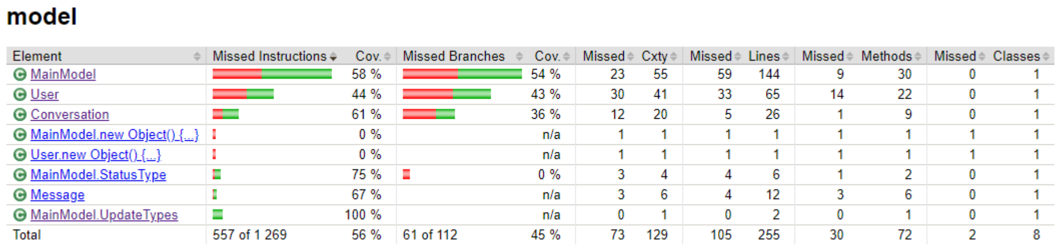

IChatView: chatView
IChatController: chatController
MainModel, Observable: mainModel
Message: m
Conversation: activeConversation
JsonSaver, Observer: jsonSaver
MainView, Observer: mainView
Inputs text into chatTextArea
onSendButtonClicked()
getInputText()
String text
sendMessage(text)
<<create>> new Message(id, userId, text, LocalDateTime)
add(m)
update(UpdateTypes.ACTIVE_CONVERSATION)
saveModel()
update(UpdateTypes.ACTIVE_CONVERSATION)
update()
loadMessages()

Figure 3: A sequence diagram for sending a message

- List of tests
  The tests are in src/test/model and can be run with JaCoCo coverage using "gradle test jacocoTestReport". It can be viewed at Figure 4. The test coverage in User is poor since the toString and hashCode methods are not tested properly. The coverage is bad in model since there is currently a method in it

for instantiating "filler" or "mock" information with users and conversations to make the program work. It will be corrected, and the filler method should not be in MainModel, it would be better to have a JSON-file preconfigured for testing purposes. Some methods in MainModel which have recently been written have no tests written for them.



model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MainModel	🟥🟩	58 %	🟥🟩	54 %	23	55	59	144	9	30	0	1
User	🟥🟩	44 %	🟥🟩	43 %	30	41	33	65	14	22	0	1
Conversation	🟥🟩	61 %	🟥🟩	36 %	12	20	5	26	1	9	0	1
MainModel.new Object() {...}	🟥	0 %		n/a	1	1	1	1	1	1	1	1
User.new Object() {...}	🟥	0 %		n/a	1	1	1	1	1	1	1	1
MainModel.StatusType	🟥🟩	75 %	🟥	0 %	3	4	4	6	1	2	0	1
Message	🟥🟩	67 %		n/a	3	6	4	12	3	6	0	1
MainModel.UpdateTypes	🟩	100 %		n/a	0	1	0	2	0	1	0	1
Total	557 of 1 269	56 %	61 of 112	45 %	73	129	105	255	30	72	2	8

Figure 4: A JaCoCo report for the model package.

- Quality, known issues
  Pmd with quickstart ruleset reports no issues.

2.3 View

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

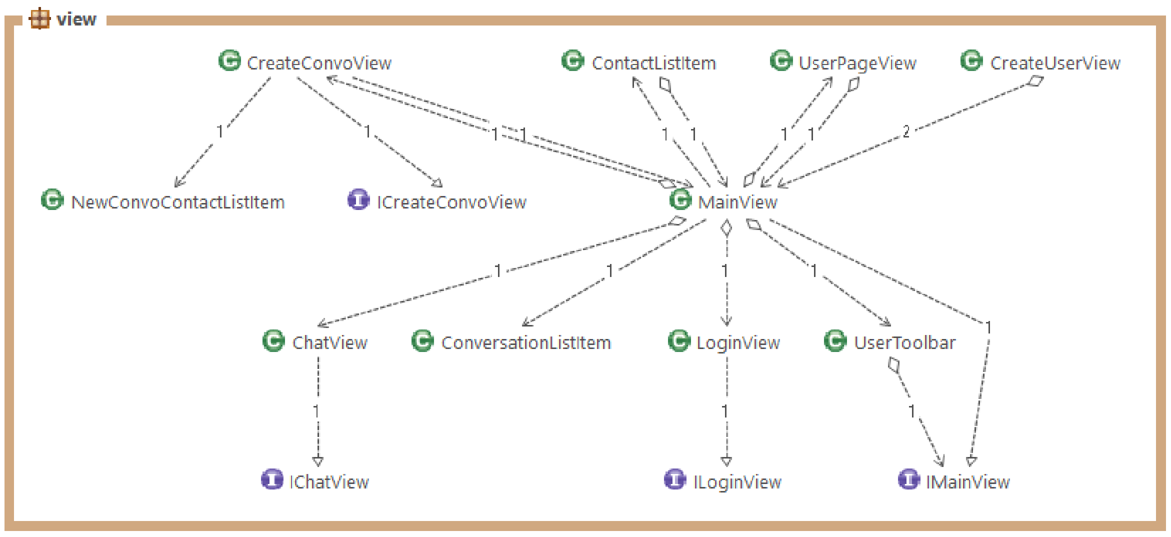


Figure 5: View package dependency analysis using STAN.

- 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
  An overview of the design model can be seen in Figure 5.
  The view is comprised of a main class called MainView that 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 also implement a corresponding interface. MainView is also an Observer of MainModel.
  The view components are seperated into smaller classes like this because it is easier to make sense of the code, and to make all the parts interchangeable, if one would like to make a new implementation of a certain view. The ChatView loads messages from the MainModel which it uses the inner class MessageItem class 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. 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 recieves updates since it is an observer to the MainModel. The reason for having multiple update types is so that the view only updates relevant info.

- 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.

- Give a class diagram for the design model.

Diagrams

- Dependencies
  The dependencies have been analyzed using STAN and can be viewed at Figure 5.

- UML sequence diagram
  An UML sequence diagram can be viewed in Figure 6. The diagram depicts the process of Logging in.

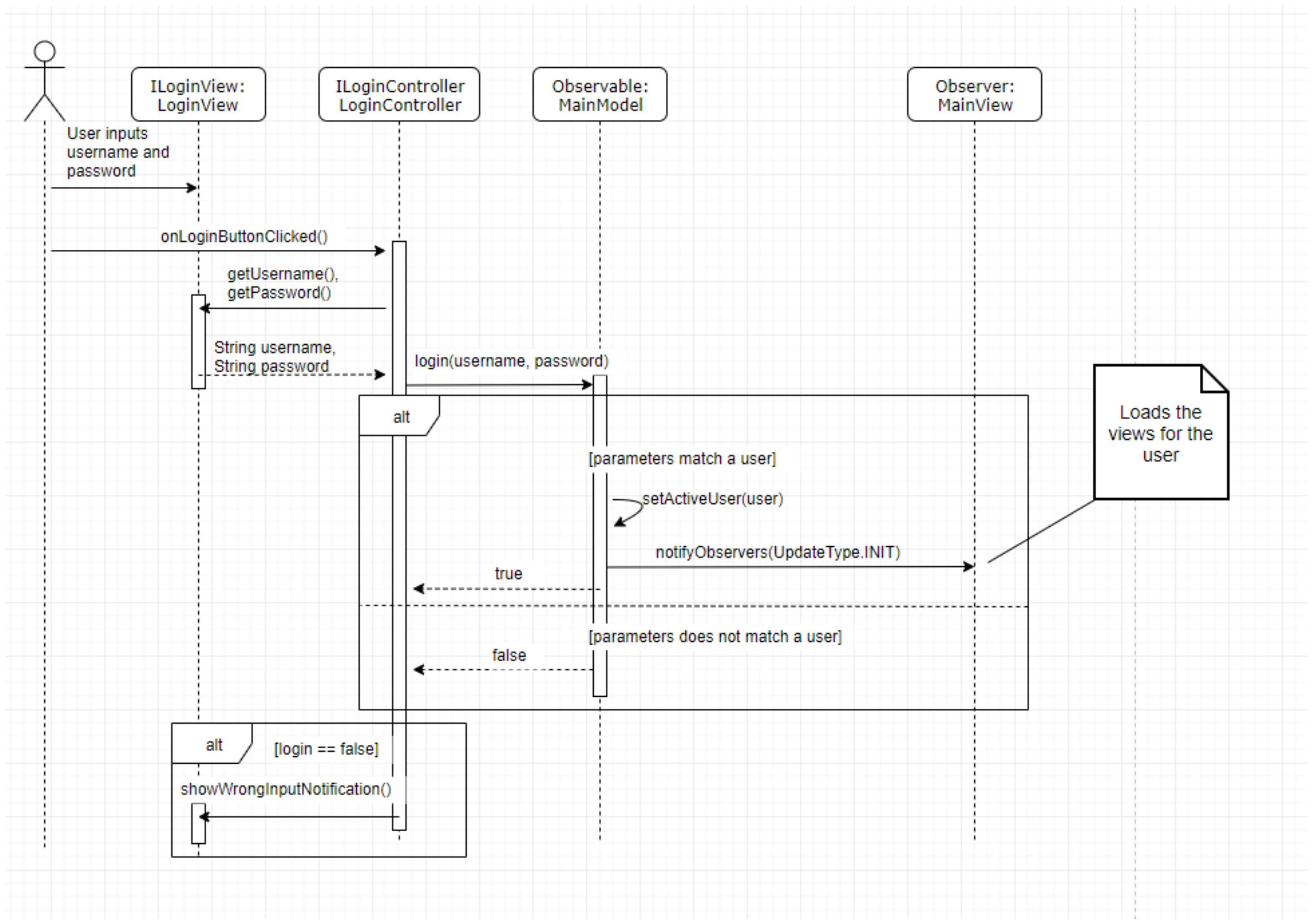

ILoginView:
LoginView
ILoginController
LoginController
Observable:
MainModel
Observer:
MainView
User inputs username and password
onLoginButtonClicked()
getUsername(), getPassword()
String username, String password
login(username, password)
alt
[parameters match a user]
setActiveUser(user)
notifyObservers(UpdateType.INIT)
true
[parameters does not match a user]
false
Loads the views for the user
alt
[login == false]
showWrongInputNotification()

Figure 6: A sequence diagram for Logging in

Quality

- Testing
  N/A
- Quality, known issues

2.4 Infrastructure

The infrastructure package is responsible for handling the data of the ShatApp. It loads the data on application start-up, and saves the data continuously through out the usage of the app. The data storage method that used for this application is Json.

- MVC implementation
  N/A

- Design model
  JsonSaver implements observer, 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 Observer 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 7.

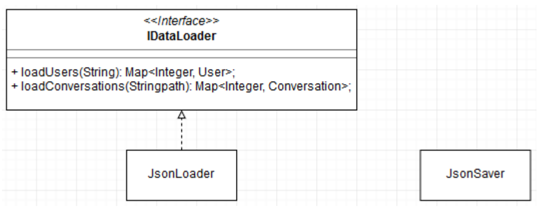

<<Interface>>
IDataLoader
+ loadUsers(String): Map<Integer, User>;
+ loadConversations(Stringpath): Map<Integer, Conversation>;
JsonLoader
JsonSaver

Figure 7: Infrastructure class diagram

Diagrams

- Dependencies
  Dependencies have been analyzed using STAN and can be viewed in Figure 8.

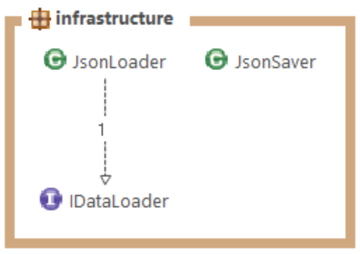


Figure 8: An analysis of the Infrastructure package created using STAN.

- UML sequence diagrams for flow.

Quality

- List of tests The tests are in src/test/infrastructure.

- Quality, known issues
  All Pmd errors reported with the quickstart ruleset were fixed, it reports no issues.

3 Persistent data management

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.
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