Final Report for TDA367

Carl Östling, Yazan Ghafir, Erik Gunnarsson, Mohamad Almasri
Group 21
CEYMChat

2018-10-28

A big thank you to our supervisor Mazdak for invaluable tips and guidance.

Contents

Chapter 1

Requirements and Analysis

1.1 Introduction

This document will analyze and discuss the project ”CEYMChat” created by group 21. The problem that the ”CEYMChat” application tries to solve lies within human communication. By creating a simple chat application with an understandable Graphical User Interface (GUI), human interaction and communication will flourish. Anyone who wishes to communicate via online chat with friends or other users can do so using ”CEYMChat”. The application requires less personal data than other existing chat applications on the market. ”CEYMChat” is a lightweight chat application developed using the java.net library. It requires no installation of software on the users device and requires minimal personal information during sign-up. No files need to be saved on the users device.

1.1.1 Definitions, acronyms and abbreviations

”Group 21” refers to Erik Gunnarsson, Carl stling, Yazan Ghafir and Mohamad Almasri.
”CEYMChat” refers to the application built by group 21 during this project.
”GUI” refers to a Graphical User Interface.
”.net” refers to the java.net library used for network connections in the project.
”Multithreading” refers to the usage of multiple threads in a program in order to execute several tasks at the same time.
”CLI” refers to ”Command Line Interface”
”FXML” refer to an XML-based user interface markup-language.
”IO” is an abbreviation for Input/Output.
”WAN” is an abbreviation for Wide Area Network. Also known as ”The Internet”

1.2 Requirements

1.2.1 User Stories

Story Identifier: CEYMCHAT001

Story Name: Send a message

Description
As a user, I would like to be able to send a message to another user of choice to communicate.

Confirmation
- Message is sent to the server.
- Message is received by another user.
- Message is received by the correct user.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT002

Story Name: Server handles a coming message

Description
As a server, I would like to receive messages to distribute them to the appropriate recipient.

Confirmation
- Message is received by the Server.
- Message is redistributed.
- Message is sent to the correct client/recipient.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT003

Story Name: Chat on a GUI

Description
As a user, I want to have a GUI so I can use the program outside a command line.

Confirmation
- The client has got a working GUI interface.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT004

Story Name: Register as a user

Description
As a user, I want to register an account so I can have my own profile.

Confirmation
- Client recognizes a user object.
- Client recognizes data in a user object in order to change its appearance.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT005

Story name: Server receives commands
Description
As a server, I want to receive commands to execute the corresponding method.

Confirmation
- Server receives messages containing commands.

- Server executes commands it has received.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT006

Story name: User can save chat history

Description

As a user, I want to be able to save my chat history so I can access it any time I want.

Confirmation
- Chat history can be exported to some readable file.
- A file containing chat history can be saved locally.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT007

Story name: User can choose whom to chat with

Description

As a user, I want to be able to click on another user in a list so we can communicate.

Confirmation
- Logged in users are displayed in the GUI.
- Users can be clicked to choose who to send a message to.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT008

Story name: User can send files

Description

As a user, I want to be able to send files so I can share them with my friends.

Confirmation
- Users can choose a file in the GUI to send.
- Chosen file is sent to the server.
- Server redistributes the file to the correct user.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT009

Story name: User can see when other people are online

Description

As a user, I would like to see who is online to decide whom to chat with

Confirmation

- Other users are shown in the GUI.
- Currently online users are marked as such.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT010

Story name: User can add friends to a friendslist

Description

As as user, I want to add another user to a friendslist so I can have quick access to them.

Confirmation
- Users in the current users friendslist are shown in the GUI.
- Users can be added to the userlist at run-time.
- Tests passed.

Non-functional
- Code well documented.

Story Identifier: CEYMCHAT011

Story name: User can send emojis in a message

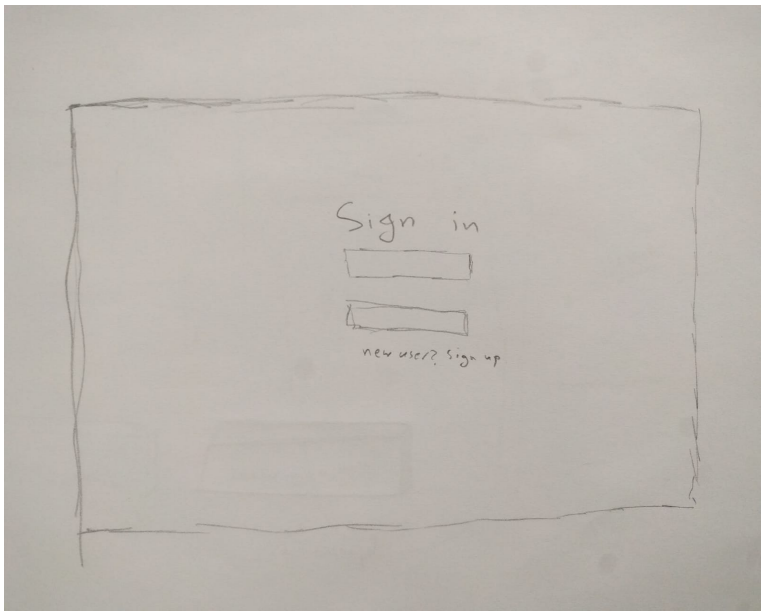
Description

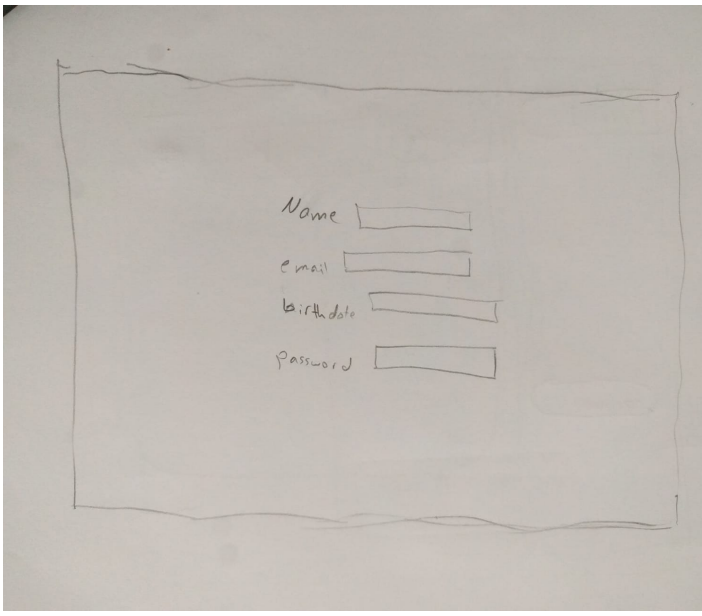
As as user, I want to add send and receive emojis in my messages so i can visualize my feelings.

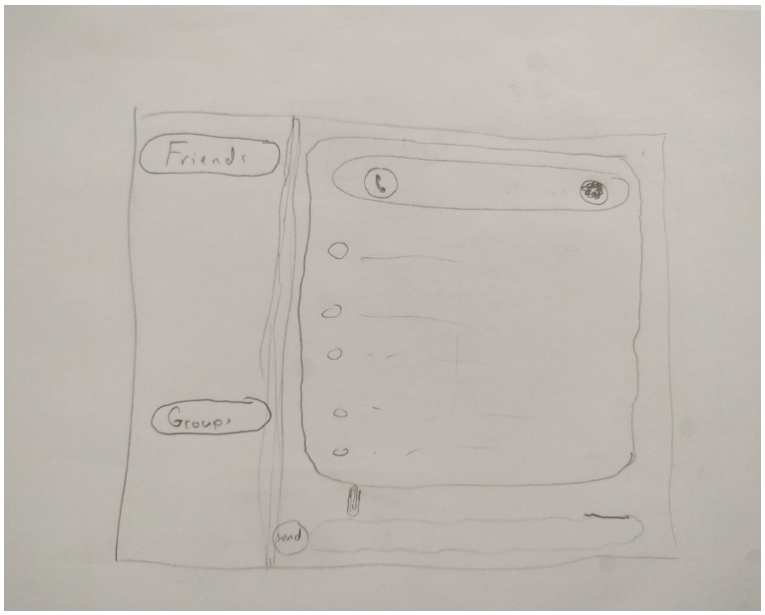
Confirmation
- Emojis can be chosen from a list of emojis.
- Emojis are shown in the sending text box and at the sent message in the chat box.
- Tests passed.

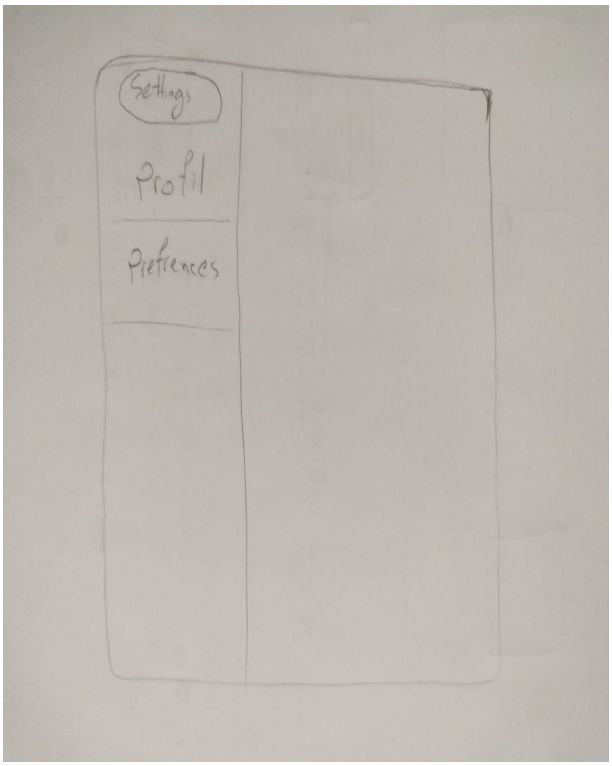
Non-functional
- Code well documented.

1.2.2 User interface









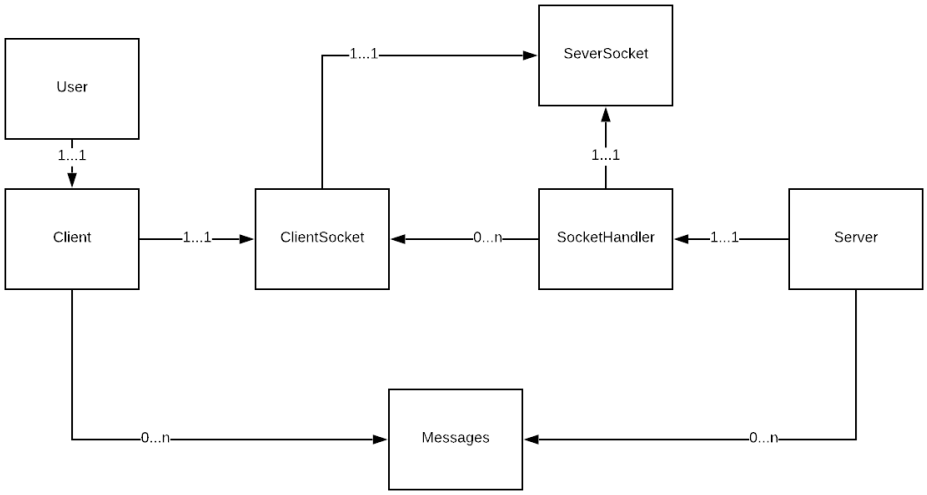


Figure 1.1: High-level UML-diagram of the application. Sockets represent a network connection between server and client

1.3 Domain model

1.3.1 Class responsibilities

Client module

Name	Package	Summary
ClientMain	none	This is the runnable class of the Client. It launches the program.
ClientModel	Model	This class contains the model of the Client.
InputService	Services	Maintains the input that comes from the server.
OutputService	Services	Maintains the output to be sent to the server.
ServiceFactory	Services	A factory for creating services.
PlayBackThread	Services	A Thread to playback a soundfile.
RecordThread	Services	A Thread to record a soundfile.
ClientController	Controller	Controller class to function as input for model and to send updates to GUI.
FriendListItem	Controller	Controller used for a GUI element that displays a friend.
EmojiItem	Controller	Controller used for a GUI element that displays an Emoji.
ReceivedTextMessage	Controller	Controller used for a GUI element that displays a received textmessage.
SentTextMessage	Controller	Controller used for a GUI element that displays a sent textmessage.
Emoji	View	Holds a certain emoji's data.
EmojisMap	View	A class that holds a map containing each and every emoji.

model-lib module

Name	Summary
Message	A generic class that encapsulates the data to be sent/received by the actors. A Message is not to be confused with a simple message containing text, it's rather an encapsulation of data that the involved actors can handle. For example, a Message can contain a command, a simple textmessage or a file.
MessageFactory	Factory for the Message-class. Narrows the generic element in Message to only be constructed with specified objects.
Command	Can be sent by a client to the server to request the server to perform the issued command.
CommandName	Enum used to identify which commands can be recognized by the server.
AudioMessage	A type of message that can contain audio.
MessageFile	A type of message that can contain a file.
UserInfo	A class containing specified information about a user in the server that is also interpreted by the client.
MessageType	An Enum used to identify the type of Message sent/received(eg. File, Audio, Command..).

Server module

Name	Package	Summary
SocketHandler	none	Runs as a Thread to continously accept incoming client connection requests to establish connection between server and client and creates user objects for new connections.
ServerMain	none	This is the runnable class of the server. It launches the server.
ServerModel	Model	This class contains the model for the server.
User	Model	A user object is created for each user in order to differentiate them and store relevant information. Contains for example the writer for sending messages to the users connected client.
Reader	Server	A Reader runs as a Thread to continuously monitor and process incoming Messages from a client.
Writer	Server	A Writer is used to send Messages to a client.

Chapter 2

System Design

This is an in-depth documentation for the System Design of CEYMChat. For a complete design-model, please see DesignModel.pdf in repo.

2.1 System architecture

2.1.1 Description of subsystems involved

The Server

All communication will pass through the server. The server will handle all stored data including logged in users and stored messages. It runs on any machine capable of running a .jar-file and with the necessary ports open to the WAN.
The Server is simply stopped by exiting the CLI.

The Client

The Client is the users application that runs on a desktop. It allows the user to chat with another user on a separate machine, anywhere in the world.
The Client is simply stopped by exiting the window.

The "Flow"

Before any interaction between other users can happen, a connection is established to the server. The server will uniquely identify you by the username that you entered upon connecting.
In order to chat with another client, a receiver is selected in the GUI. Thereafter, the message that the sender wishes to send passes through the socket-connection with the server. The server then processes the message and passes it on to the selected receiver. The receiver receives the message, processes it and then prints it to the client GUI.

2.1.2 Subsystem decomposition

The Server

The Services package is composed of two child-packages, FileServices and RemoteServices, dealing with file related I/O and remote server communication respectively. The FileServices packages hosts services reading and writing to and from files on the local machine aswell as a service for recording and replaying audio. For services reading from the local machine there is an Interface ILoadMessages which contains a single method, loadSavedMessages, which returns a List¡String¿ of all past messages. Services writing to the local machine have a similar Interface ISaveMessages containing the method saveMessages, which is meant to save all sent and received messages into a local file. In this child-package there is also a Configurations class which handles the loading of configurations within the client, simply reading from a properties

file. The RemoteServices package contains interfaces for Input and Output services communicating with the server. IInput is responsible for reading all content sent by the server whereas the IOutput Interface handles the sending of data to the Server with its ”sendMessage” method.

The Client

The client is composed of a model, a view, a controller and services. The model is responsible for all logic, the view for representing the program as a GUI and the services are responsible for all I/O connections on the client-side. Alongside this there is a controller connecting the model to the view. The controller mainly works as a gateway for GUI events such as keystrokes and mouse-clicks.

Model-Lib

A library consisting of a Message class, a Command class, a UserInfo class as well as a MessageFactory. The Message class is a generic class consisting of some sort of data, a string containing the senders username as well as a UserInfo object containing the receiver. The command class was implemented to let the server know that a message was sent to the server itself rather than a user so that a command such as ”SETUSER” or something similar could be sent to the server without the risk of sending that message on to another user as it may contain personal data. UserInfo is the object instantiated so that a users data can be stored in the same object, such as username or profile picture. Both the server and the client handles this class to differentiate between users. Lastly there is a MessageFactory responsible for creating messages in a controlled manner. Considering Message has a generic field ”data <T>” a Message could, with a public constructor, contain any type of object. In order to restrict the types of objects that can be put in this generic field, the constructor of Message is made package-private and only accessible through the facade and factory MessageFactory.

2.1.3 Client

The Model

The Model package holds classes that are responsible for keeping track of the current state of the client application and classes handling the logic needed to complete the clients responsibilities.

The View

The View package holds classes responsible for the client GUI.

The Controller

The Controller package contains classes responsible for controlling the model and view by either taking inputs from the view, send messages via the services or run logic in the model.

The Services

The Services package is composed of two child-packages, FileServices and RemoteServices, dealing with file related I/O and remote server communication respectively. The FileServices packages hosts services reading and writing to and from files on the local machine aswell as a service for recording and replaying audio. For services reading from the local machine there is an Interface ILoadMessages which contains a single method, loadSavedMessages, which returns a List¡String¿ of all past messages. Services writing to the local machine have a similar Interface ISaveMessages containing the method saveMessages, which is meant to save all sent and received messages into a local file. In this child-package there is also a Configurations class which handles the loading of configurations within the client, simply reading from a properties

file. The RemoteServices package contains interfaces for Input and Output services communicating with the server. IInput is responsible for reading all content sent by the server whereas the IOutput Interface handles the sending of data to the Server with its ”sendMessage” method.

Diagrams

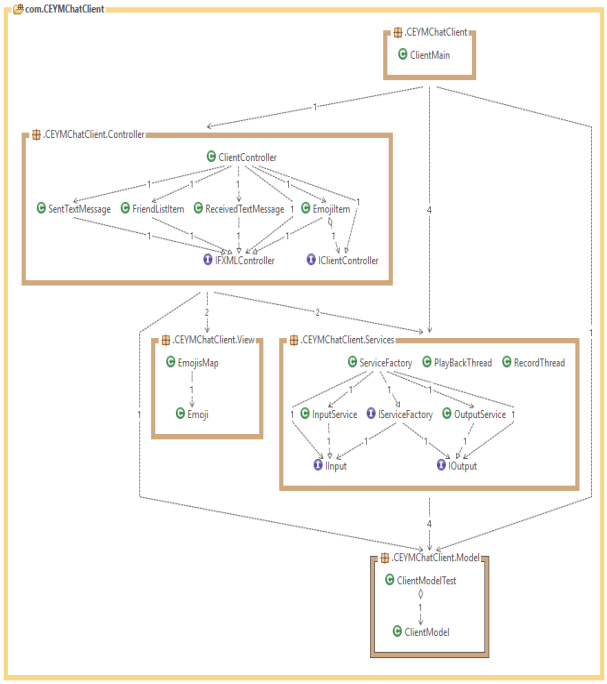


Figure 2.1: STAN4J generated dependency diagram of Client module

For a class diagram, click here.

2.1.4 Model-lib

This module has no child-packages.

Diagrams

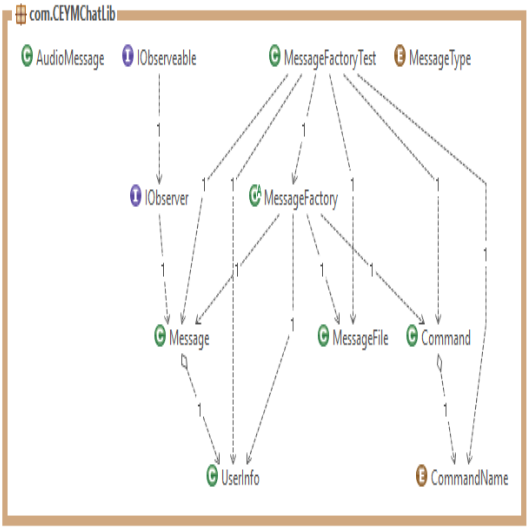


Figure 2.2: STAN4J generated dependency diagram of model-lib module

For a class diagram, click here.

2.1.5 Server

In the server package there is a serverMain class and a sockethandler. the serverMain is the class which initiates the server. The sockethandler class is responsible for continuously looking for new clients trying to connect to the server. Every time a client connects, the sockethandler adds some information in the model aswell as starting both a Reader and Writer to read and write data coming from and to that client.

Model

This package holds the serverModel class which is responsible for holding the state of the server aswell as logic for executing commands if a client sends a command. Serverlogic in general is handled in the model. The User object located in this package is a representation of a user, it is responsible for sending out data to a specific client via the socket provided when that client first connects to the server.

Services

Contains network I/O-classes responsible for sending/receiving and processing messages. Also contains the service for connecting new clients to the model. The sockethandler in the parent-package which is started in the Main method of the server creates a User object whenever a new connection has been made which starts a Writer responsible for sending data back to the client which the User is representing, alongside this the sockethandler also creates a Reader, which continuously reads data from an input stream corresponding to the client. There is an Interface for both the Writer and the Reader.

Diagrams

For a class diagram, click here.

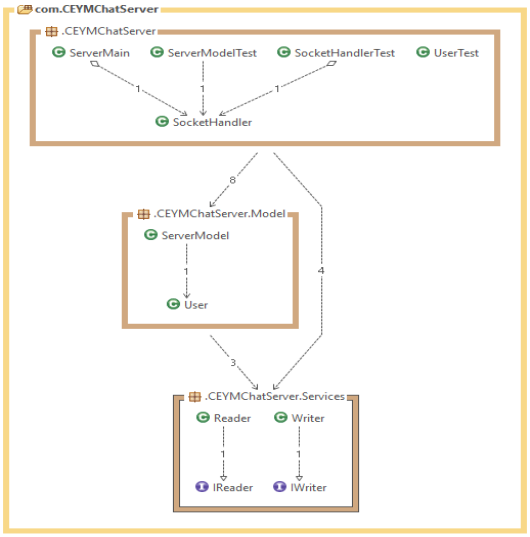


Figure 2.3: STAN4J generated dependency diagram of Server module

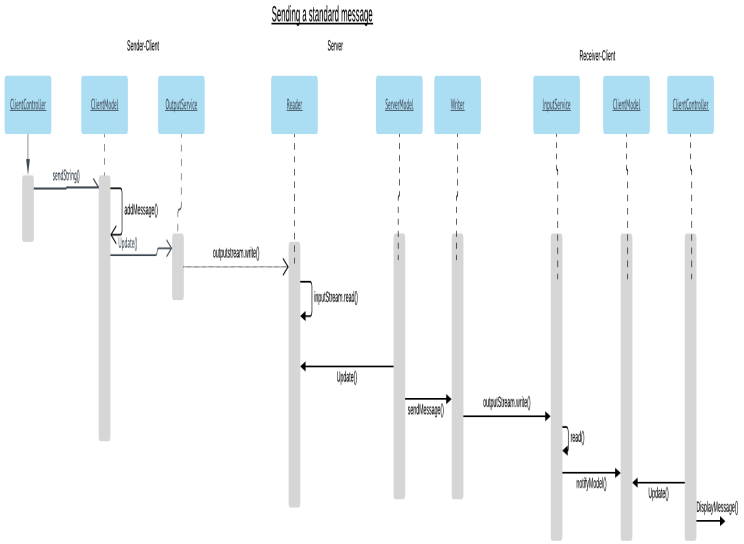


Figure 2.4: Sequence Diagram of sending a standard message from Client to Client

2.1.6 Quality

- Tests for each module can be find in their respective /src/test/java/com
- Click here for a PMD report.

2.2 Persistent data management

All the visible resources are stored in a resource folder that are then used by the client. When a user is sent a file, that file is then stored in a folder. When a user closes the client, their chatlog is saved locally on their computer and will reload upon launch. Note that the user itself is not stored, only the chat history. Blocked and friendlisted users are therefore not saved upon shutdown.

2.3 Access control and security

There are no ”admin”-roles in the client, instead an admin has control of the server meaning that only the operator of the system can look at critical data and overview the system at run-

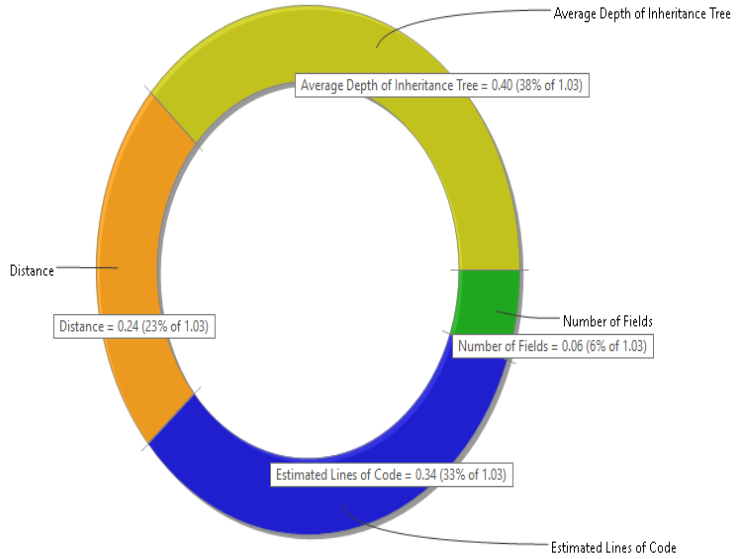


Figure 2.5: STAN4J generated Pollution Diagram

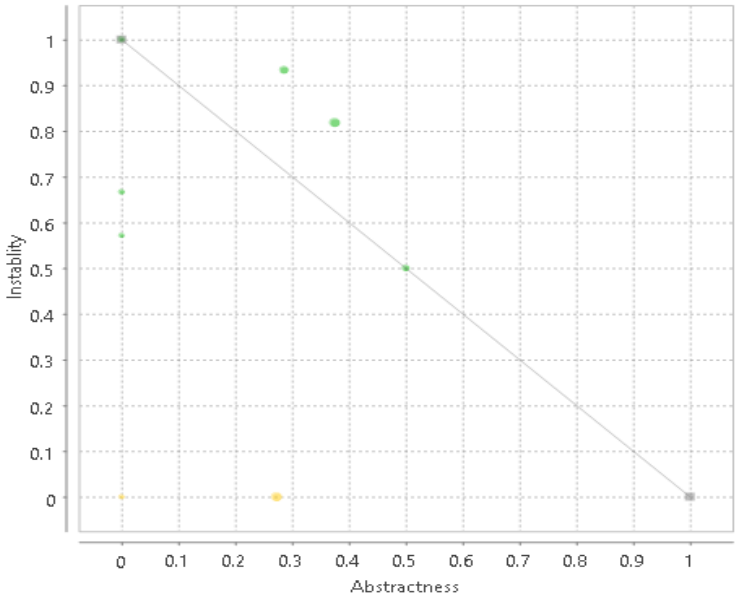


Figure 2.6: STAN4J generated Distance Diagram

time. The server class should never be shared with anyone but those developing it meaning that usage of the server will be limited. All users of the client class have the same privileges and will not gain access to information they shouldn't have access to.

Chapter 3

Peer-Review for group 18

1. **Do the design and implementation follow design principles? Single Responsibility Principle** is followed well. Every class has a name that describes the functionality.
   Open-Closed Principle Even though there is an abundance of interfaces, it doesn't always allow of easy extension. Some interfaces are hard-coded and doesn't really allow for much different implementations. This infringes the Open-Closed Principle.
   Composition over Inheritance The project uses inheritance within different interfaces in a controlled manner. This might spawn issues if a ”super-interface” is changed. This can lead to concrete implementations implementing a ”sub-interface” getting unwanted methods.
   We think it would be wiser to simply implement more interfaces instead of implementing one that could potentially give unwanted methods. This would in-turn lead to infringing on the **Interface Segregation Principle**. However, the project does not currently break this principle. **Dependency Inversion Principle** is followed well. Most dependencies is of an interface and not an implementation. There are however some infringements. For example, LoginController depends upon DataHandler when there's an abstraction ”IDataHandler” that could be depended upon instead. This seems to happen on quite a few additional places.

2. **Does the project use a consistent coding style?** Things such as indentations, whitespaces and naming looks to be following a coherent coding style. It looks like the project is using camel-case naming convention on both functions and variables.

3. **Is the code reusable?** The code in it's entirety won't be reausable for something else, but some parts of it can be. For example, the encryption tools can be extracted. Thanks to the MVC-structure the components can be used in conjunction with other modules allowing reuse.

4. **Is it easy to maintain?** The code should be easy to maintain considering eveything is of a neat MVC-structure and local .json files. If something breaks, it should be easy to locate and fix considering the good naming and packaging.

5. **Can we easily add/remove functionality?** This project is built to run on the same machine and if you'd want to run it through an ”actual” server, the same functionality cannot easily be achieved through simple refactoring. The functionality would have to be built from the ground up.
   To add new functionality such as adding friends should be quite easy because of the use of interfaces and a well implemented MVC-Structure.

6. **Are design patterns used?**

   - Observer pattern is utilized to update other clients when a new message has been sent.
   - Adapter pattern is utilized in the JBCryptAdapter class.
   - Command pattern is utilized to send commands such as ”kick”.

- Factory and Facade is used in MessageFactory and ViewComponentsFactory.

7. **Is the code documented?** The code in its whole is well documented. A few variables are not commented, but they are self-explanatory thanks to good naming. The main-class Wack.java lacks comments.

8. **Are proper names used?** Naming is done well, it's easy to understand what a variable or class is for and what a function does even without reading the javadoc.

9. **Is the design modular? Are there any unnecessary dependencies?** With the use of a neat MVC-Structure the design is quite modular, however it's not all perfect. Many implementations seem to implement interfaces from another package creating unnecessary dependencies between packages. It would make sense to refactor these into the same package so that the Views package is not depending on the Controller package through these interfaces. We suggest that the interfaces should be refactored to the Controllers package. For example Views/IChannelViewController is implemented by Controllers/ChannelViewController.

10. **Does the code use proper abstractions?** The code uses interfaces for most classes to ensure that the code is easy to maintain and build upon. There are also a few instances where interfaces inherit from each other which might not be ideal as discussed in item 3. The use of generics could have been apparent in the MessageContent class or in the Message class to make it easier to implement other functionality in the future.

11. **Is the code well tested?** The Controller lacks tests entirely. The Model and the Services have 100% coverage. Most getters, setters and the ”real” functionality seems to have proper tests to test the functionality.

12. **Are there any security problems, are there any performance issues?** Considering the server and the different clients are running on the same machine, one user has access to all the other users passwords through the memory of the PC even though passwords are encrypted because you have access to the decryption used.

13. **Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?** The MVC structure is well defined and allows for switching out modules via interfaces. However, because the server and the client share the same model-package it is hard swap out clients and servers for new ones. You can swap them out using their interfaces, but the functionality will be very much the same and limited. From our perspective, the server isn't much of a server, it mostly consists of functionality for the model.

14. **Can the design or code be improved? Are there better solutions?** If the server had its own model and it's own main function to run as a separate application, the functionality could be improved beyond the limitations of using one machine(which brings a lot of potential functionality). If this was done from the start and correct abstractions were made using this setup, the project could be easily scaleable and modular.