ICT ENGINEERING
VIA UNIVERSITY COLLEGE

SEP2X-S19 Project Report

Poetry forum.

Members:

Lau Ravn Nielsen (280337)

Sébastien Malmberg (279937)

Tamás Fekete (266931)

Supervisors:

Ib Havn

Troels Mortensen

Number of characters:

47.635

56.527 with spaces

Deadline:

01:00 pm.

07 / 06 / 2019

Contents

List of Figures and Tables

Abstract

This project has aimed to create a multi-client server system with a connecting database, to provide a service for the poetry club at VIA University College. The service is a poetry forum, which purpose is to ease the communication between the poets at VIA and provide them with a platform in which they can post their poetry and give each other feedback. The system can be divided into three parts, the client side, server side and persistence. The client side has been implemented following the model view view model pattern and it communicates with the server through a socket connection. The server then provides a connection to the database through data access objects. The system successfully implements the six highest priority user stories. A user is now able to create an account, log in, and post poetry as well as view its own personal information. However, the overall result falls short compared to the aim of this project, as the system contains a fatal error and fails to implement high priority user stories.

Introduction

The poetry club at VIA university college consists of twenty members including the president and vice-president of the club. However, the club wishes to assemble a community much larger than their current member count. The members share their creations by reciting them to each other during club meetings. They draw inspiration from their discussions and from the internet. Afterwards they post their poems on a social-media group to archive their creations for later review. They write a large variety of poetry and therefore they want a manner of organizing their poetry, so that they may search for specific tags or sort all poetry by genre. Currently the poetry-club is utilising facebook.com to post their poetry, establish meetings, and give each other feedback. The poetry club at VIA university college in Horsens wishes for an application customized to assist and connect poets all over campus However, the poet society wishes to move their listing of poetry to a new virtual environment. They wish for a new forum-like application, that can replace the Facebook group with a safe environment in which they do not have to worry about privacy and copyright related issues that comes with using a third-party application. According to the Facebook terms they have the free right to use all the material posted on their platform (Facebook, 2019). Nevertheless, the club wishes for a new tool in which they can post their poetry, rate posted poetry and comment on poetry. In addition, the system should include a leader board based on the user's average poetry rating. The tool should also assist poets by inspiring them during the writing process by recommending rhymes and synonyms whilst also displaying inspirational images or similar poetry from the same genre. Lastly, each user wants a gallery in which only their own poetry is displayed.

Furthermore, the poetry club consists of international students who travel a lot in order to get to work and to visit their families, consequently they communicated the desire to write poems and to meet with the others, while on the go. Therefore, the poetry club has also asked for a mobile application in which all the previously mentioned features would be implemented as well.

The poetry club has previously tried using Discord, which is a freeware VoIP application and digital distribution company (Discord, 2017). Which is essentially a software which performs similar duties as a traditional telephone system (Barton, 2017). The application is customized for gamers and does not fulfil the needs of the poetry club. The application does not include a personal profile fit for the poetry club, nor does it include a rating system, and all posts are sorted by most recent.

Requirements

Functional requirements

The user stories below have been written with the presence and participation of the customer, this was done to ensure that there was a clean connection between what the customer wants and what the developers think the customer want. The list of requirements below is ordered by importance for the customer.

1. As a user I want to be able to create my own personal account, so I can participate in the community.
2. As a user I want to be able to login, so I can use the system.
3. As a user I wish to be able to make post(s) so that my poetry is stored in a feed.
4. As a user I want my posted poetry to be visible to other users, so other people can enjoy it
5. As a user I wish to see a common feed between all users, where all posted poetry is displayed, so that I can browse poetry posts.
6. As a user I want to have my own personal page, which displays information about me such as first name, last name, email, date of birth, username and a picture of my choice.
7. As a user I want the ability to remove a post I have posted, so I can remove them from the forum.
8. As a user I want to be able to edit my posts, so I can correct mistakes.
9. As a user i want to be able to delete my account so that I can leave the community.
10. As a user I want to be able to comment on others poetry so that I can give constructive feedback.
11. As a user I want to be able to have a friends list, so I can have a small community within a larger one.
12. As a user I want to be able to send friend requests to other users, so I can build my own personal community.
13. As a user I want to be able to remove my comments so that I can remove my input if I consider it inappropriate.
14. As a user I wish to have my own posted poetry displayed on my account/profile, so people can see a compilation of my poetry.
15. As a user I want to be able to sort all poetry in the common feed by rating, so I can see a list of the top-rated poetry.
16. As a user I want to be able to sort all poetry in the common feed by genre, so I can sort for the poetry I’m in the mood for.
17. As a user i want the ability to rate other people’s poetry so that we can have feedback from each other.
18. As a user I wish to see a leader board consisting of the top-rated poets, so I can see a list of the top-rated poets.
19. As a user I want to be able to delete friends, so I can manage my friends list.
20. As a user I want to be able to send private messages to other users so I can share my thoughts privately.

21. As a user I want to be able to receive private messages from other users, so I can talk in private.
22. As a user I want the ability to change my username, so I can keep it new and trendy.
23. As a user I want to be able to change my password, in order to keep my account secure.
24. As a user I want the friends list to display which users are currently active and which are not, so I can see a list of friends who are available to discuss poetry.
25. As a user I want the ability to report a post so that we can have inappropriate posts removed.
26. As a user I want the ability to ignore other accounts so that I will not see their poetry again.
27. As an admin I want to be able to delete other users' comments so that I can remove unwanted input.
28. As an admin I want to be able to delete other users posts so that we can avoid spam.
29. As an admin I want to be able to ban users so they can't bother other users.
30. As an admin I want to be able to lock a post so that no user can post further comments, in order to keep a civil tone on the forum.

Non-functional requirements

1. The program must be coded in Java.
2. The program must store data in a database.
3. The program must be operable with a keyboard and mouse.

Analysis

The analysis section was conducted in order to outline the problem domain. The section will start of with a use-case diagram made based on the user stories and the actors herein with additional assistance from the customer by asking the customer what the goal(s) of the system is. When that is done and dusted, the use-case descriptions can be made based on the completed use-case diagram. The use-case descriptions will give an in-depth description of what will happen in the different use-case. Afterwards, the most difficult and complex use-case descriptions will have a UML-format made to give a cleaner visualization of the use-cases by making either an Activity diagram of a System sequence diagram. The analysis section of the report will end with a problem domain model, which will be a made based on all previously completed analysis work.

Use-case Diagram

The use-case diagram below, shows the connections between the users and the various use-cases. The use-cases give a quick overview of what the intent of the various actors are when using the program.

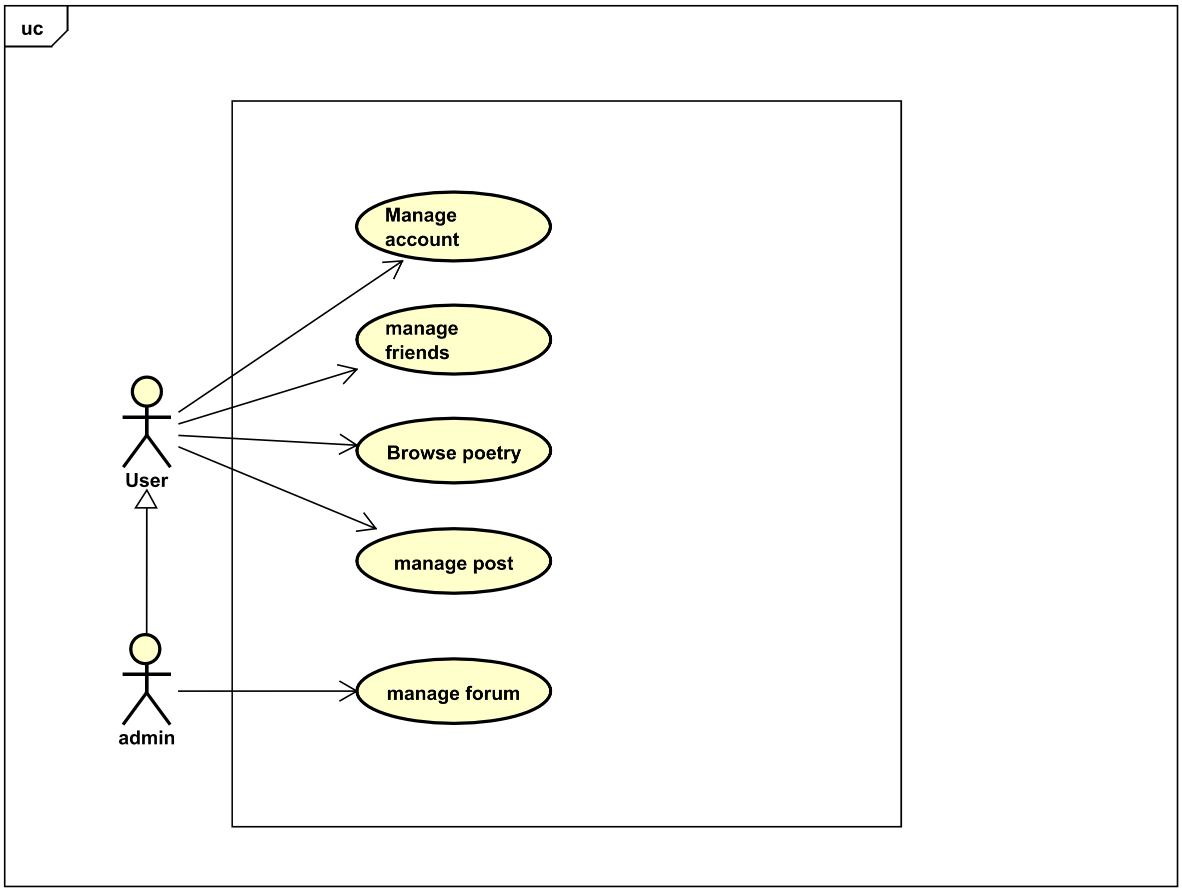


Figure 1 - Use-case diagram

Use-case Description

Based on the use-case diagram we have made detailed use-case descriptions in order to give a detailed textual explanation of what happens in the different use-cases. The use-case descriptions have been made as sub-use-case descriptions for every functionality in a given use-case. This was done to keep the use-case descriptions simple and understandable. Below is a use-case description of the sub-use-case <<Create Post>> from the use-case <<Manage Post>>. The remaining use-case descriptions can be found in Appendix A – Use case descriptions.

Use case description – Create post

Use Case Category Manage Post

Use Case Name: Create Post

Scope: Cool Poetry Assister

Level: User Goal

Primary Actor: User

Stakeholders and Interests:

User wants to create a post and make it public on the forum.

Preconditions: User is logged in.

Success Guarantee: The poetry is stored in the system.

Main Success Scenario:

1. User is browsing poetry.
2. User selects ‘create post’.
3. User writes his/her poetry.
4. User selects ‘submit post’
5. The post is submitted.
6. The post is pushed to the live feed.

Extensions:

4b. Post is empty

1. The post is empty
2. The post is not pushed to the live feed.

Frequency of occurrence: Whenever a user wants to create a post.

Open issues:

Activity diagram

The activity diagram below, showcasing a sub-use-case <<Create Post>> of the use-case <<Manage Post>>. The activity diagram has been made based on the use-case description of <<Create Post>>. Activity diagrams have been made for all the implemented use-case descriptions, and some use-case descriptions that we planned on implementing soon. These diagrams can be found in Appendix B – All UML Diagrams.

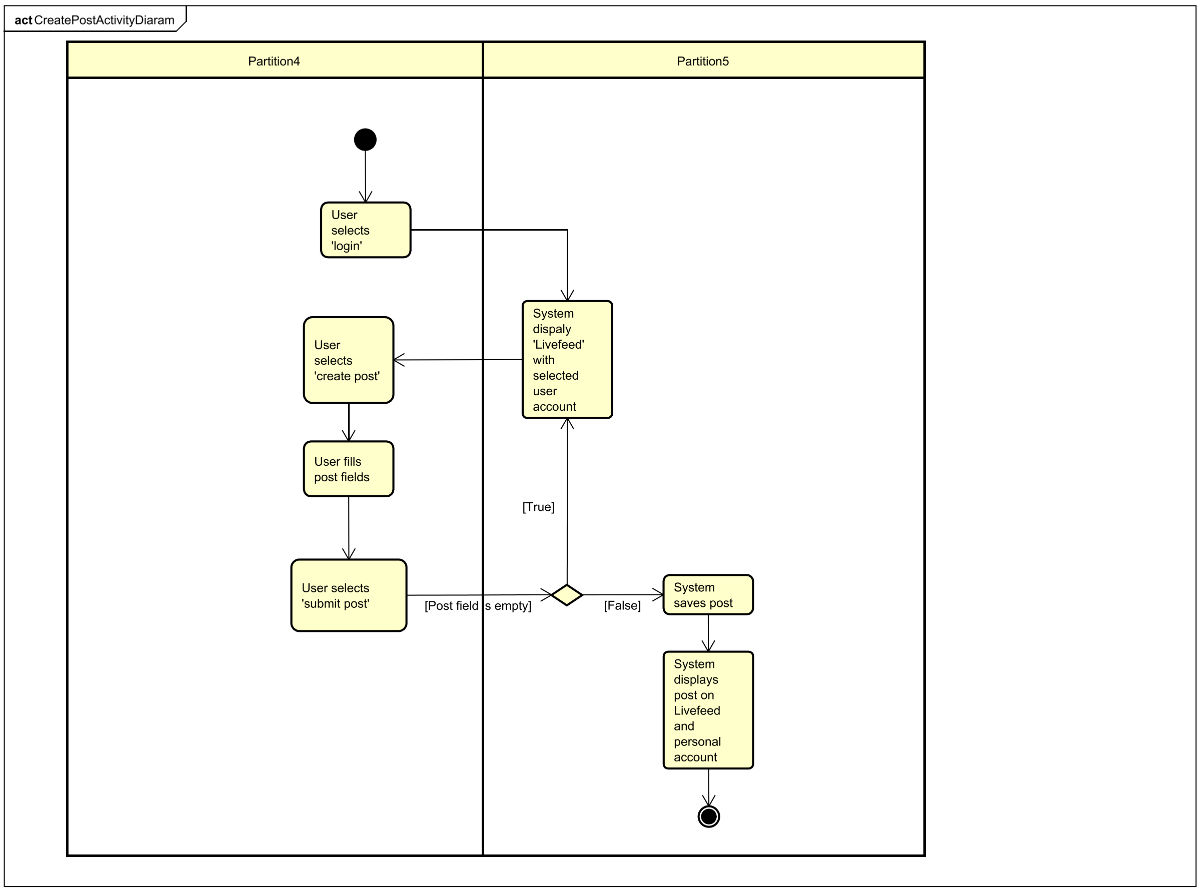

act CreatePostActivityDiaram
Partition4
Partition5
User selects 'login'
System dispaly 'Livefeed' with selected user account
User selects 'create post'
User fills post fields
[True]
User selects 'submit post'
[Post field is empty]
[False]
System saves post
System displays post on Livefeed and personal account

Figure 2 - Activity diagram – sub-use-case <<Create post>> in the use-case <<Manage post>>

As the diagram shows: The user selects login, then the user is directed to the live feed GUI. Then the user selects create post and writes his/her poetry. Afterwards the user selects submit post and then the system will check if the post is empty or not. If it is empty the user will be directed back to the live feed. If the post is not empty it will be saved in and pushed to the live feed.

Domain model

The domain model is constructed by doing thorough analysis on the requirements. The conceptual classes were found first, by analysing the requirements. Then duplicates were removed so there was consistency with the words used. Afterwards, the conceptual candidates were analysed in order to differentiate between classes and attributes. Then actors were analysed to see if they could be classes.

Afterwards, associations between classes were found by analysing the verbs used in the requirements. Then the domain was analysed for derived associations and reflective associations. The final product of the analysis is shown below as our domain model.

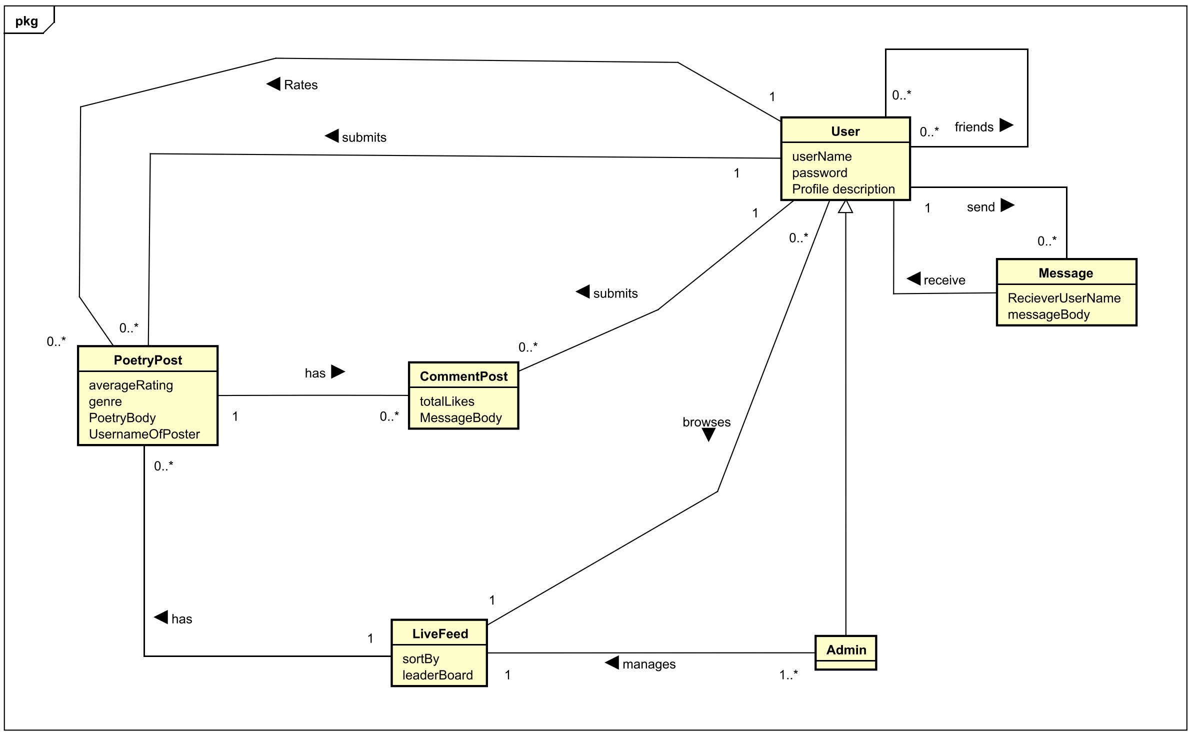


Figure 3 - Domain model

As the Domain model shows: A **User** has a reflexive relationship that allows zero to many **Users** to have zero to many friends (which are also **Users**, implied by the reflexive relationship. Additionally, one **User** can send zero to many messages to another **User** that receives the message. Furthermore, one **User** can rate zero to many **PoetryPosts**(rating is the equivalent of giving a thumbs up on Facebook, however, when rating it can also be a thumbs down) . Furthermore, one **User** can submit zero to many **PoetryPosts**. Moreover, one **User** can submit zero to many **CommentPosts**. Lastly, zero to many **Users** can browse one **LiveFeed**.

Additionally, one **PoetryPost** has zero to many **CommentPosts**. In addition, one **LiveFeed** has zero to many **PoetryPosts**.

Lastly an **Admin** is extending **User** meaning it can do anything a **User** can. Additionally, one to many **Admins** manages one **LiveFeed**, managing means that an admin can lock the poetry posts within the livefeed or ban the users posting on it etc.

Test cases

Test cases have been made based on the use-case descriptions and they will be used during the testing section for Blackbox testing.

The test case below is for create post in the use case manage post.

Test case 1

Table 1 - Test case 1

ACTION NO.	ACTION	REACTION
1.	Browse poetry	Verify that the livefeed is shown.
2.	Select create post	Verify that the you have selected the post cell.
3.	Write poetry	Verify that the system is recording your inputs.
4.	Submit post	Verify that the system submitted your post to the livefeed. Verify that the system pushed it to other users. Verify that you cannot post an empty post

The test case below is create account from manage account

Test case 2

Table 2 - Test case 2

ACTION NO.	ACTION	REACTION
1.	Opens the program	Verify that the login screen is shown.
2.	Select create account	Verify that the create account screen is shown.
3.	Fill in username, password, full name, date of birth, e-mail and a picture.	Verify that it checks if the two passwords are the same. Verify that fields cannot be left empty. Verify that a picture can be uploaded.
4.	Select create account	Verify that two account cannot have the same username. Verify that two account cannot share the same e-mail. Verify that the system is checking for empty fields.
5.	Select cancel	Verify that the system returns to the login screen.

The remaining test-cases can be found in Appendix C – Test cases

Design

The design section will be used to outline the process behind designing the structure and functionality of the program by expanding on the analysis section. This will be done by making a Design Class Diagram derived from the Domain model, and the use-case descriptions respectively. Sequence diagrams will be made interactively with the Design Class Diagram to ensure the SOLID principles are religiously followed. This is done so the software does not suffer from poor design and thus rigidity, fragility, immobility etc.
The design section will also cover the design of the database. The database design is visualized with a Conceptual ER-Diagram which is extracted from the Domain Model. Afterwards, a Global ER-Diagram will be made based on the Conceptual ER-Diagram through normalisation and mapping.

Design class diagram

The system is designed with the MVVM-design pattern and sockets. However, the diagram is structed by functionality. So, all the classes that provides GUI functionality are grouped in 'View'. The classes that oversee Client logic are in 'viewModel' and 'Model' respectively. The classes that handle the networking are grouped in 'Networking' and 'Server'. Lastly the persistence is found in 'Mediator'. All the above will be explained in further detail in the upcoming sections. The diagram can also be seen in Appendix B – All UML Diagrams.

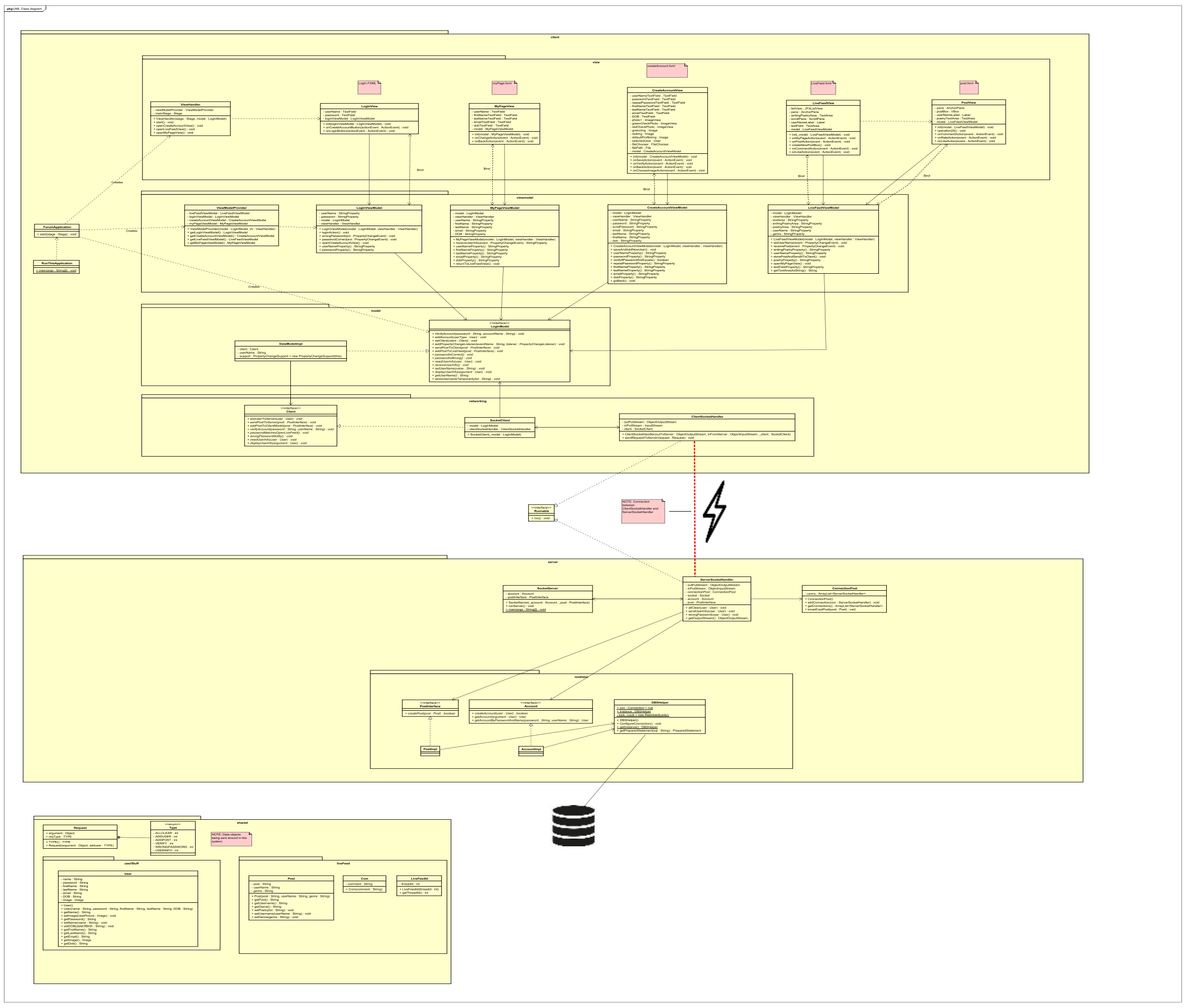


Figure 4 - Design class diagram

Model view view model

The implementation has strictly followed the Model view view model (MVVM) pattern. The pattern was chosen due the fact that responsibility can easily be distributed between classes. Furthermore, the pattern allows for easy modification of the code, and provides a good foundation which can easily adopt additional code and classes. The pattern has also helped with staying within the boundaries of the SOLID principles as it greatly encourages the single responsibility principle.

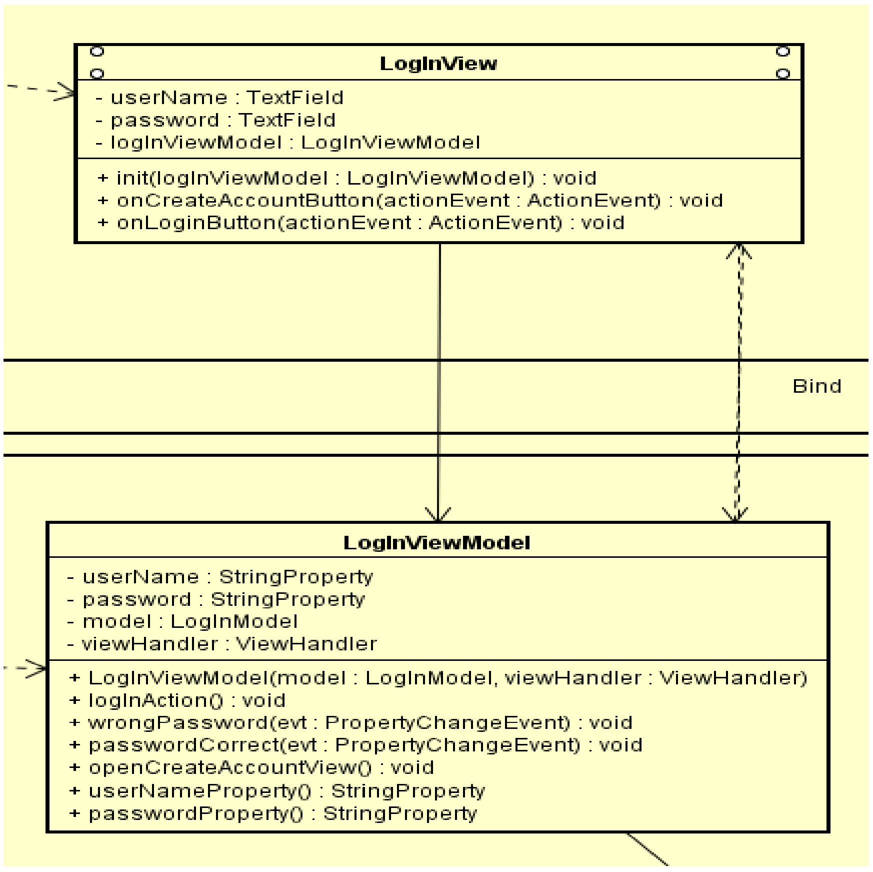
View the full sequence of logging in, Appendix B – All UML Diagrams. Also see entire class diagram in Appendix B – All UML Diagrams.

View

Also known as the controller. The view handles the presentation. It is tied together with the fxml file by JavaFX framework. It displays the information and does the rendering of data, graphs, images etc. Furthermore, the view also takes user input, as radio buttons, check boxes, text fields etc. The user's interaction with the system is handled here. See **LogInView** class in the diagram below. The View class always has a reference to its respective view model and often calls methods on the view model.

View model

The View model handles the events. When a button is pressed it will trigger an event or action. This event is passed on from the View class to the View model. The view model additionally handles the state of the graphical user interface ex. is a specific element selected? or has a button been disabled? These are things that the view model needs to keep track of. The view model also handles formatting. Since the GUI will always display information as strings, the view model formats each attribute into a fitting type because elements in the GUI are "bound" to properties in the view model. See **LogInViewModel** class in the diagram below. The view model has a reference to the model and calls methods on the model.

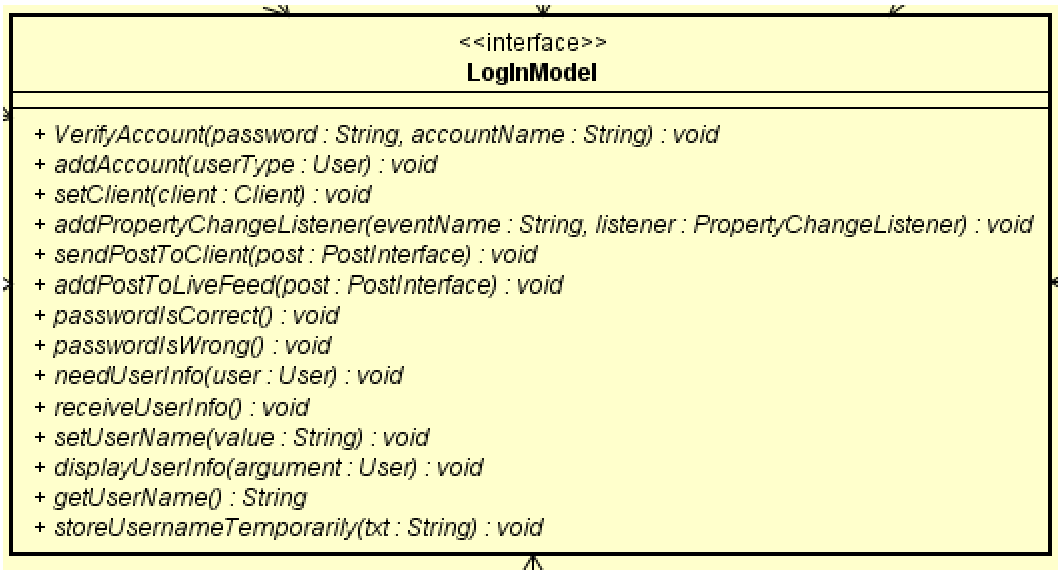


LogInView
- userName : TextField
- password : TextField
- logInViewModel : LogInViewModel
+ init(logInViewModel : LogInViewModel) : void
+ onCreateAccountButton(actionEvent : ActionEvent) : void
+ onLoginButton(actionEvent : ActionEvent) : void
Bind
LogInViewModel
- userName : StringProperty
- password : StringProperty
- model : LogInModel
- viewHandler : ViewHandler
+ LogInViewModel(model : LogInModel, viewHandler : ViewHandler)
+ logInAction() : void
+ wrongPassword(evt : PropertyChangeEvent) : void
+ passwordCorrect(evt : PropertyChangeEvent) : void
+ openCreateAccountView() : void
+ userNameProperty() : StringProperty
+ passwordProperty() : StringProperty

Figure 5 - UML for LogInView and LogInViewModel

Model

The Model takes the business logic and handles the data manipulation. Calculations, rules and validation is implemented here. Any type of information, data, or state the system needs to hold, goes to the model. Note, that the **LogInModel** is not cohesive. It violates the single responsibility principle as it includes all business logic of this system and not solemnly the logic behind logging in. This model also implements a property change listener, which fires property change events, these events are then caught in the view model where some action is taken. The implementation of this interface can be found in the **DataModelImpl** class.

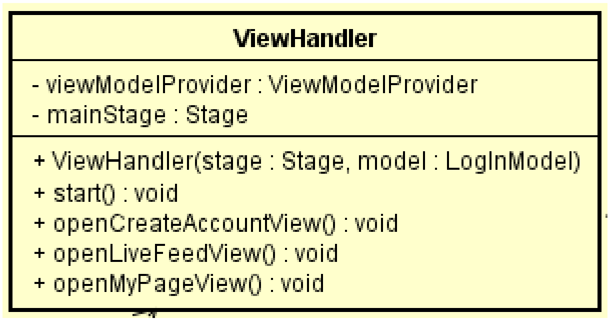


<<interface>>
LogInModel
+ VerifyAccount(password : String, accountName : String) : void
+ addAccount(userType : User) : void
+ setClient(client : Client) : void
+ addPropertyChangeListener(eventName : String, listener : PropertyChangeListener) : void
+ sendPostToClient(post : PostInterface) : void
+ addPostToLiveFeed(post : PostInterface) : void
+ passwordIsCorrect() : void
+ passwordIsWrong() : void
+ needUserInfo(user : User) : void
+ receiveUserInfo() : void
+ setUserName(value : String) : void
+ displayUserInfo(argument : User) : void
+ getUserName() : String
+ storeUsernameTemporarily(txt : String) : void

Figure 6 - UML for LogInModel

View Handler

The **ViewHandler** class, is responsible for creating all the views and switching between views. It Is created by the **RunApplication** class.



ViewHandler
- viewModelProvider : ViewModelProvider
- mainStage : Stage
+ ViewHandler(stage : Stage, model : LogInModel)
+ start() : void
+ openCreateAccountView() : void
+ openLiveFeedView() : void
+ openMyPageView() : void

Figure 7 - UML for ViewHandler

View model provider

The **ViewModelProvider** class is responsible for creating and holding all the view models and provides get methods for each view model.

ForumApplication

The **ForumApplication** class is responsible for creating the **ViewHandler, Client** and **LogInModel**.

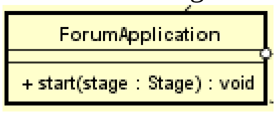


Figure 8 - UML for ForumApplication

Networking

The MVVM pattern allows for easy transportation of objects and simple code for networking between client and server. The **DataModelImpl** class has a reference to the client interface, the model can therefore send tasks to the client. The implementation of the **Client** interface exists in the **SocketClient** class.

SocketClient

The **SocketClient** handles the tasks which the **Client** interface is given by the model and sends them to the **ClientSocketHandler** class. The **SocketClient** class also hands out tasks to the **LogInModel** interface. The socket client attempts to create a new socket in its constructor. The socket is then used as a linkage between the **ClientSocketHandler** and the **ServerSocketHandler** as ObjectOutputStream and inputStream is set.

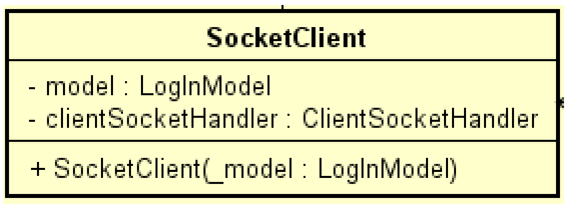

SocketClient
- model : LogInModel
- clientSocketHandler : ClientSocketHandler
+ SocketClient(_model : LogInModel)

Figure 9 - UML for SocketClient

ClientSocketHandler

This class is used to receive and send Objects from and to the server. The **ClientSocketHandler** has a "sendRequest" method which is called by the **SocketClient** to send objects to the server side. The class implements the already existing java Runnable interface, and thus it has a run method. In the run method, it attempts to receive objects from the server side (in form of a request) to then call methods on the **Client** interface.

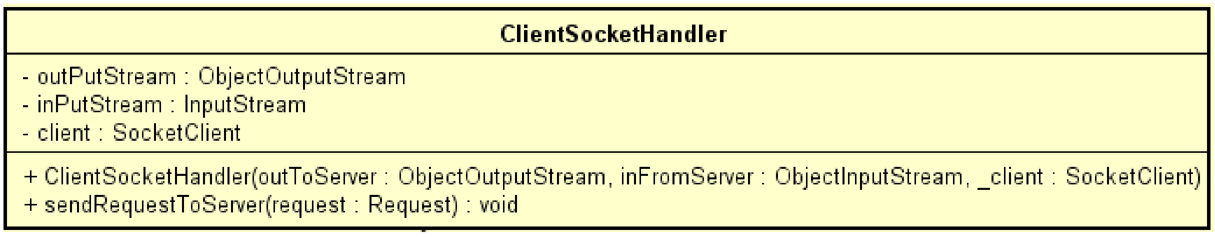

ClientSocketHandler
- outPutStream : ObjectOutputStream
- inPutStream : InputStream
- client : SocketClient
+ ClientSocketHandler(outToServer : ObjectOutputStream, inFromServer : ObjectInputStream, _client : SocketClient)
+ sendRequestToServer(request : Request) : void

Figure 10 - UML for ClientSocketHandler

ServerSocketHandler

This class implements the already existing java **Runnable** interface. In its run method, the class attempts to read the Objects sent from the client side to then perform some action. This system uses the **ServerSocketHandler** to call methods on data access objects ex. **Account** interface or **PostInter** interface. These data access objects grant a connection to the database through a **DBSHelper** class, *see Diagram on next page.*

As in the **SocketClient** class, the **ServerSocketHandler** also sets its ObjectOutputStream and inputStream in its constructor. The **ServerSocketHandler** references the **ConnectionPool** class which is used to broadcast a post to all connected clients.

The "broadCastPost" method, inside the **ConnectionPool** class goes through a loop of all connected clients, and for each connected client it sends an Object to the client side which in turn is read by the **ClientSocketHandler.**

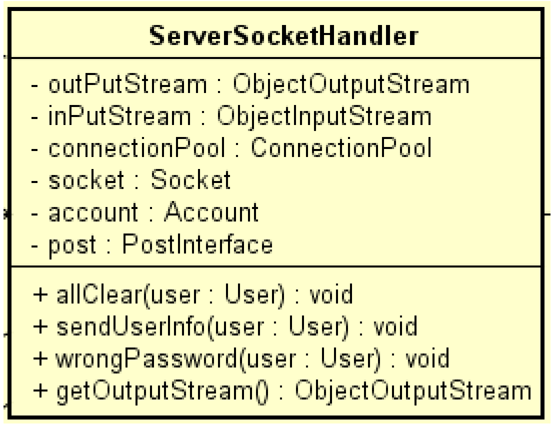

ServerSocketHandler
- outPutStream : ObjectOutputStream
- inPutStream : ObjectInputStream
- connectionPool : ConnectionPool
- socket : Socket
- account : Account
- post : PostInterface
+ allClear(user : User) : void
+ sendUserInfo(user : User) : void
+ wrongPassword(user : User) : void
+ getOutputStream() : ObjectOutputStream

Figure 11 - UML for SeverSocketHandler

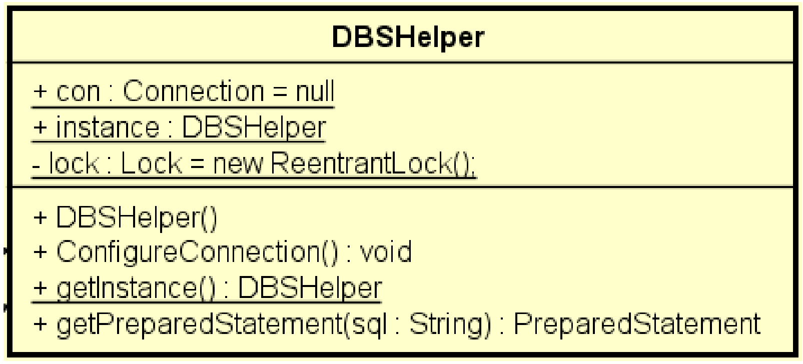

DBSHelper
+ con : Connection = null
+ instance : DBSHelper
- lock : Lock = new ReentrantLock();
+ DBSHelper()
+ ConfigureConnection() : void
+ getInstance() : DBSHelper
+ getPreparedStatement(sql : String) : PreparedStatement

Figure 12 - UML for DBSHelper

DBSHelper

The DBSHelper class has been built to establish a connection between server and database, to do that DBSHelper configurates and starts the database by calling the ConfigureConnection() method inside its constructor. ConfigureConnetion() stores users database information.

SocketServer

The **SocketServer** class has the run Server method, which is used in its main method to start the server. It creates a welcomeSocket of type ServerSocket, which takes a port as argument, the port to which the client will connect. As it accepts a connecting client, it instantiates the **ServerSocketHandler** to then create and start a thread which will take the **ServerSocketHandler** as argument.

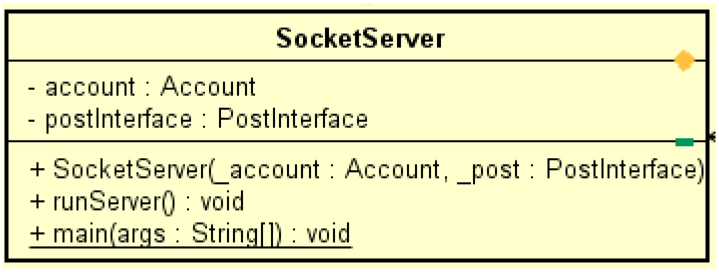

SocketServer
- account : Account
- postInterface : PostInterface
+ SocketServer(_account : Account, _post : PostInterface)
+ runServer() : void
+ main(args : String[]) : void

Figure 13 - UML for SocketServer

The shared package

The shared package contains all the data objects which are being sent around in the system.

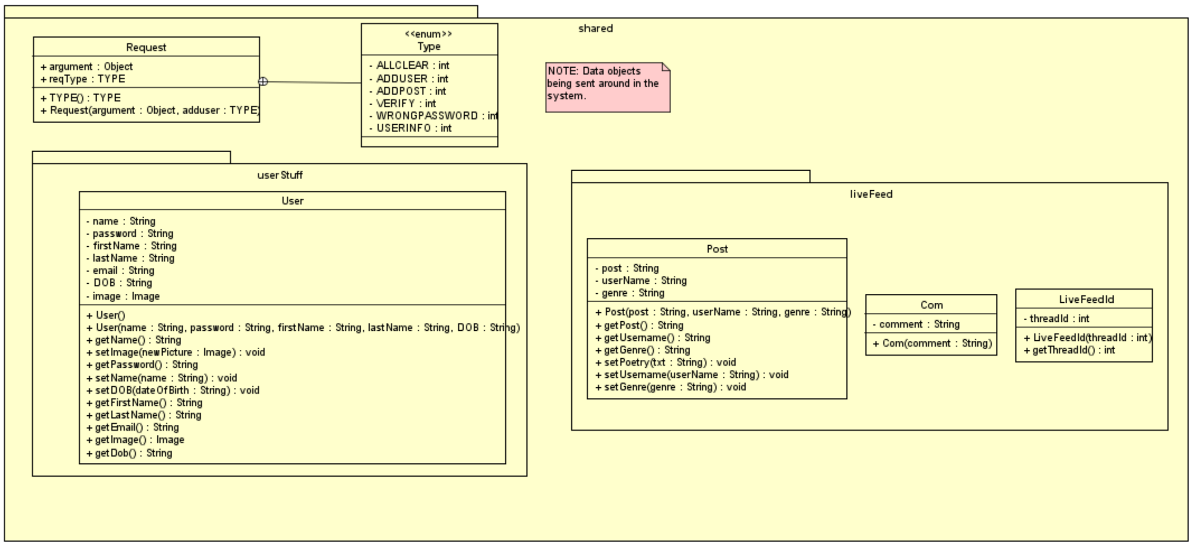


Figure 14 - UML Overview of the shared package

Sequence diagram

Log in sequence diagram

This diagram illustrates the sequence which is executed as a user attempts to log in.

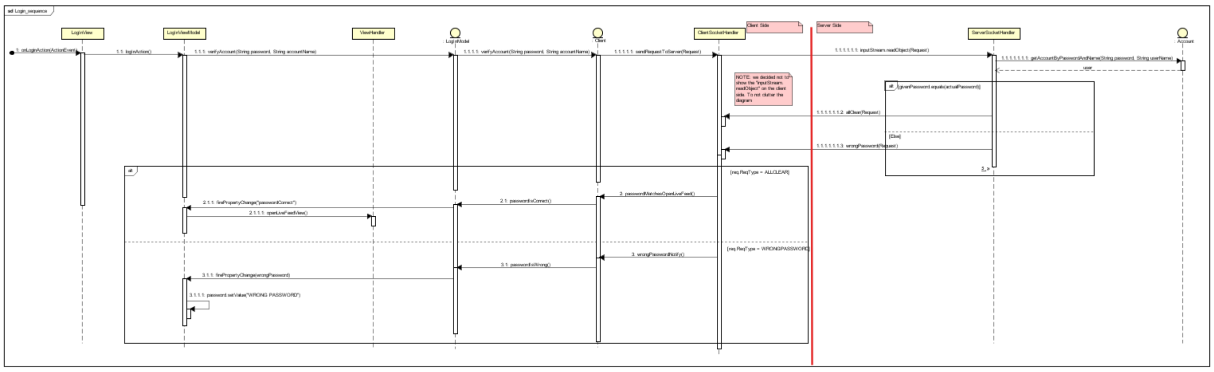

sd Login_sequence
LoginView
LoginViewModel
ViewHandler
LoginModel
Client
ClientSocketHandler
Client Side
Server Side
ServerSocketHandler
Account
1: onLoginAction(ActionEvent)
1.1: loginAction()
1.1.1: verifyAccount(String password, String accountName)
1.1.1.1: verifyAccount(String password, String accountName)
1.1.1.1.1: sendRequestToServer(Request)
1.1.1.1.1.1: inputStream.readObject(Request)
NOTE: we decided not to show the "inputStream.readObject" on the client side. To not clutter the diagram
2: passwordMatches(OpenLiveFeed)
2.1: passwordIsCorrect()
2.1.1: firePropertyChange("passwordCorrect")
2.1.1.1: openLiveFeedView()
3: wrongPasswordNotify()
3.1: passwordIsWrong()
3.1.1: firePropertyChange(wrongPassword)
3.1.1.1: password.setValue("WRONG PASSWORD")
[req.ReqType = ALLCLEAR]
[req.ReqType = WRONGPASSWORD]
[Else]

Figure 15 - LogIn Sequence Diagram

Create account sequence diagram

When a user does not have an already existing account a new one must be created. As a user inserts all the required user information, and presses save the following sequence is executed.

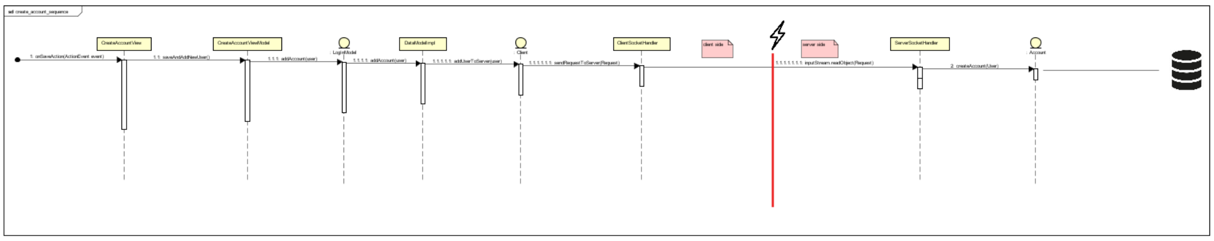

sd create_account_sequence
CreateAccountView
CreateAccountViewModel
LoginModel
DataModelImpl
Client
ClientSocketHandler
client side
server side
ServerSocketHandler
Account
1: onSaveAction(ActionEvent event)
1.1: saveAndAddNewUser()
1.1.1: addAccount(user)
1.1.1.1: addAccount(user)
1.1.1.1.1: addUserToServer(user)
1.1.1.1.1.1: sendRequestToServer(Request)
1.1.1.1.1.1.1: inputStream.readObject(Request)
2: createAccount(User)

Figure 16 - Create Account Sequence Diagram

Post poetry sequence diagram

When a user wishes to post poetry, he or she must first write some text and press the post button. This action will trigger the following sequence to occur.

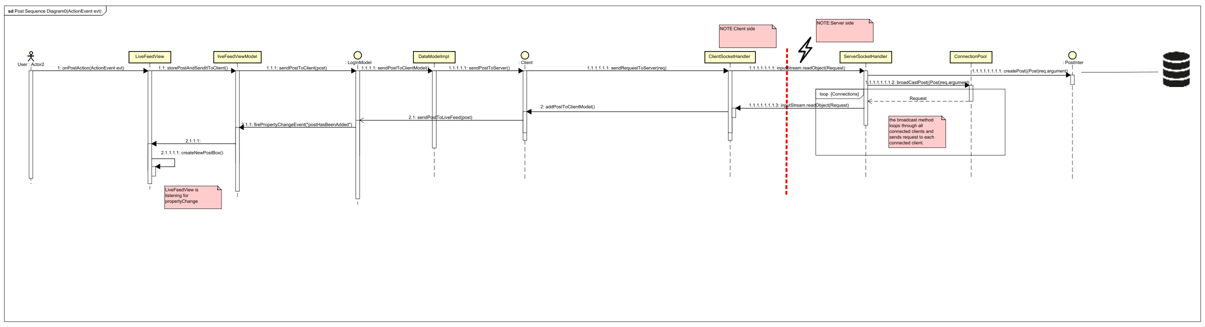

sd Post Sequence Diagram0(ActionEvent ev)
User : Actor2
LiveFeedView
liveFeedViewModel
LoginModel
DataModelImpl
Client
ClientSocketHandler
NOTE:Client side
NOTE:Server side
ServerSocketHandler
ConnectionPool
PostWriter
1: onPostAction(ActionEvent ev)
1.1: storePostAndSendItToClient()
1.1.1: sendPostToClient(post)
1.1.1.1: sendPostToClientModel()
1.1.1.1.1: sendPostToServer()
1.1.1.1.1.1: sendRequestToServer(req)
1.1.1.1.1.1.1: inputStream.readObject(Request)
1.1.1.1.1.1.2: broadCastPost(Post(req.argument))
1.1.1.1.1.1.1.1: createPost(Post(req.argument))
loop [Connections]
Request
1.1.1.1.1.1.3: inputStream.readObject(Request)
the broadcast method loops through all connected clients and sends request to each connected client.
2: addPostToClientModel()
2.1: sendPostToLiveFeed(post)
2.1.1: firePropertyChangeEvent("postHasBeenAdded")
2.1.1.1:
2.1.1.1.1: createNewPostBox()
LiveFeedView is listening for propertyChange

Figure 17 – Post Sequence Diagram

Conceptual ER-Diagram

The picture bellow shows the conceptual structure of the poet forums database. Since Poet forum is a private system for Poet Club at Via University, it can only be used by registered users, therefore creation of user table was required to store the user's credentials. Building post table to store information of posts was crucial, since the system also provides functionality for creating and sharing posts.

The user and post table have been connected one another through one attribute field called "threadId" which is the representative of Poet forums main page.

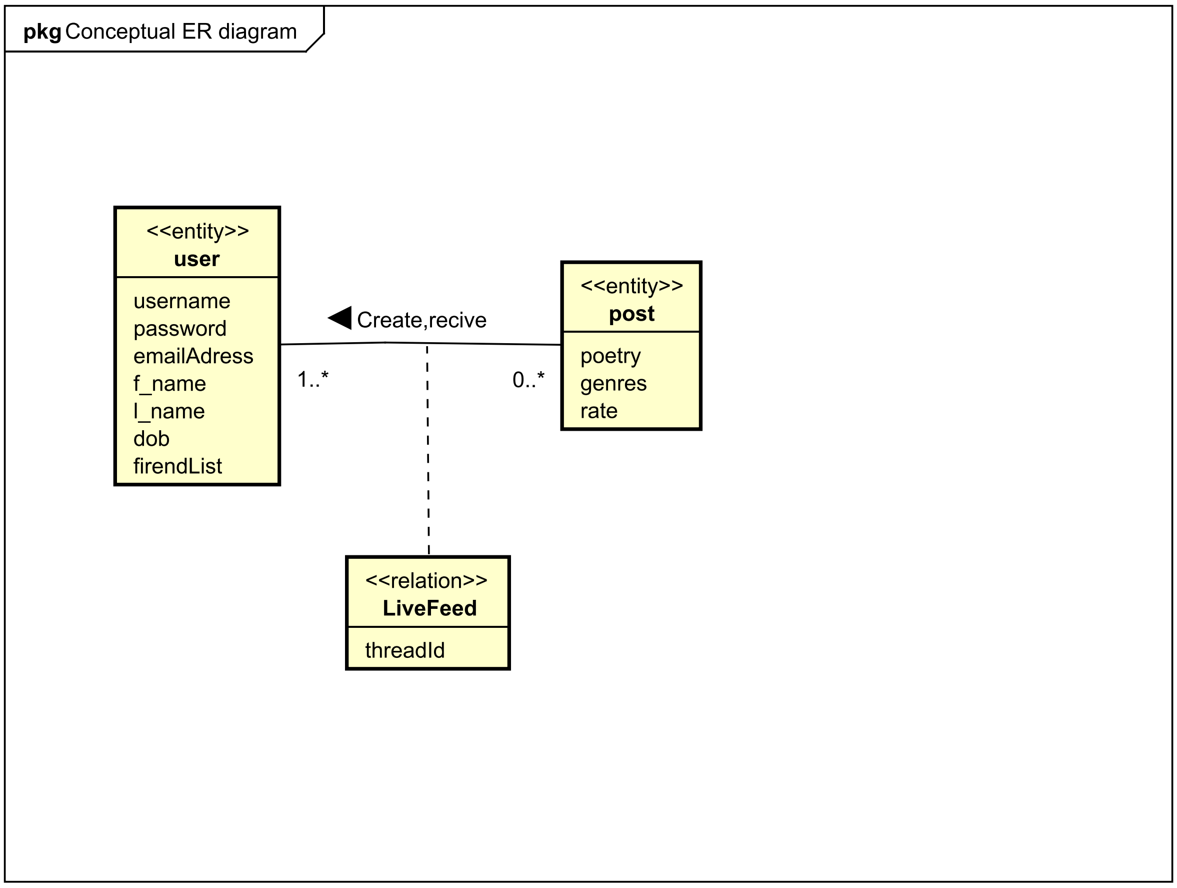


Figure 18 - Conceptual ER diagram

Global ER-Diagram

The final implementation of the database can be seen in the diagram bellow. As the picture shows the referenced field(threadId) was put it in 'livefeed' table along with the two foreign keys from 'mypage' table and 'posttable'.

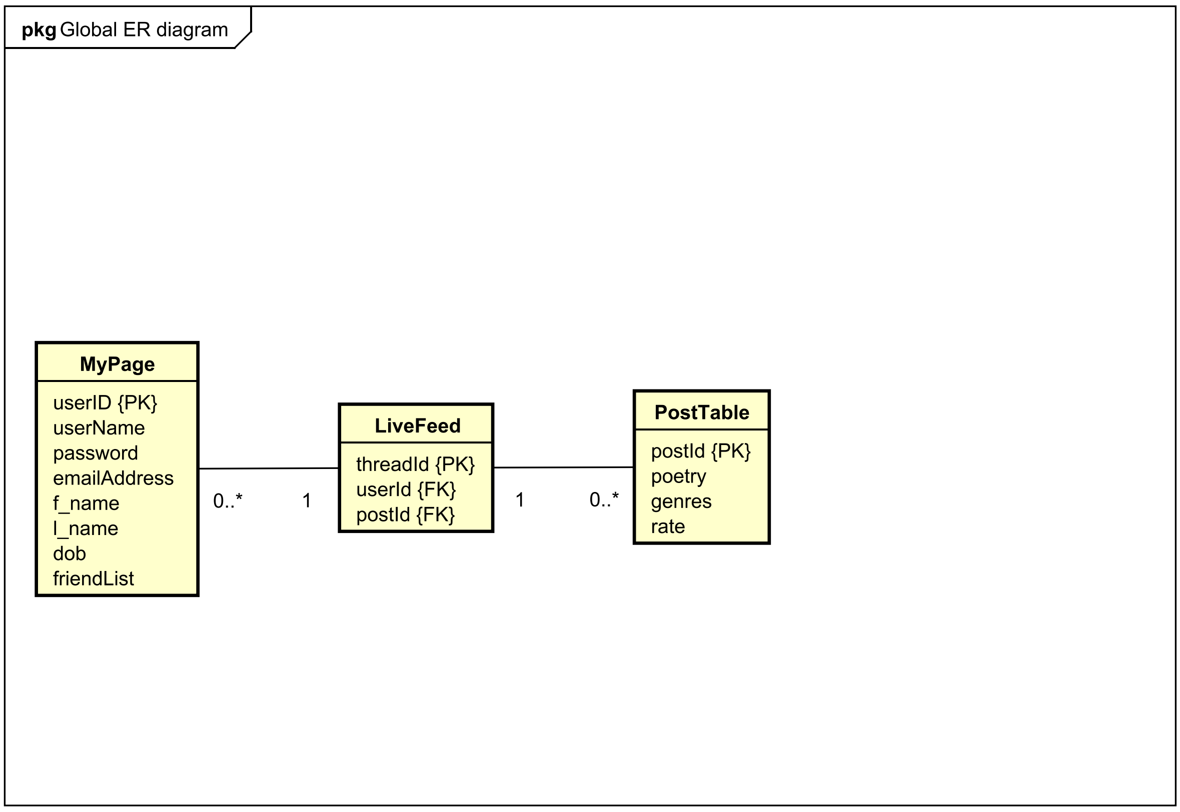


Figure 19 - Global ER-Diagram

Implementation

Implementation will aim to address the three main functions of this project, login, create account and post poetry. Secondly it will describe the client-server connection and server-database connection.

Note that the implementation has strictly followed the Model view view model(MVVM) pattern and uses socket connection between server and client side. See class diagram Appendix B – All UML Diagrams.

To see a user guide or installation guide see Appendix D – User Guide and Appendix E – Installation Guide.

Sequence diagrams:

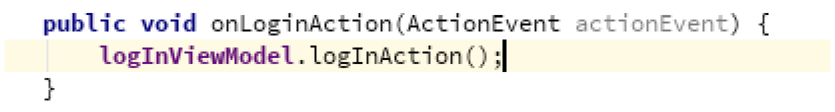
- Login sequence diagram – Appendix B – All UML Diagrams.
- Create account diagram – Appendix B – All UML Diagrams.
- Post poetry diagram – Appendix B – All UML Diagrams.

External libraries:

- JFoenix (JFoenix, 2015) Or use the .jar file in Appendix F - JFoenix
- Postgresql - 42.2.5 (The PostgreSQL Global Development Group, 2018) Or use the .jar file in Appendix G – PostgreSQL.

Following the functionality behind the sequence of logging in

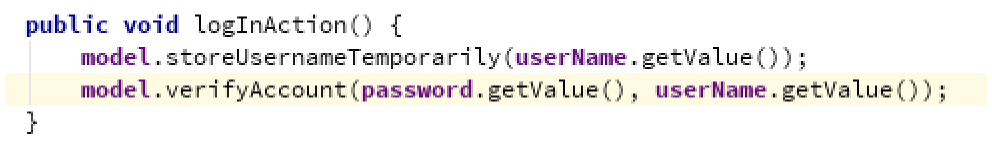
The first interaction a user will experience with the system is the login page. As a user fills in the required fields, username and password, and clicks the login button, the “onLoginAction” method, which takes an ActionEvent as argument, is called. The method itself calls another method on the LogInViewModel “logInAction”. See code example below.



```java
public void onLoginAction(ActionEvent actionEvent) {
    logInViewModel.logInAction();
}
```

Figure 20 - Class: LogInView

The logInAction method inside the LogInViewModel class then calls two methods on the model. See code example below.



```java
public void logInAction() {
    model.storeUsernameTemporarily(userName.getValue());
    model.verifyAccount(password.getValue(), userName.getValue());
}
```

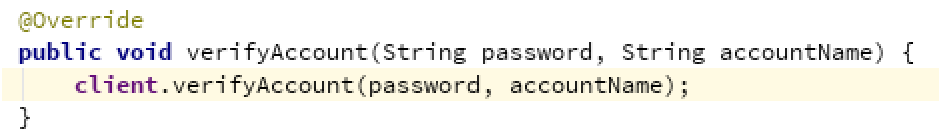
Figure 21 - Class: LogInViewModel

model.storeUserNameTemporarily(username.getValue());

Stores the username temporarily so that the liveFeedView class may retrieve the stored username and display in the username label.

model.verifyAccount(password.getValue(), userName.getValue());

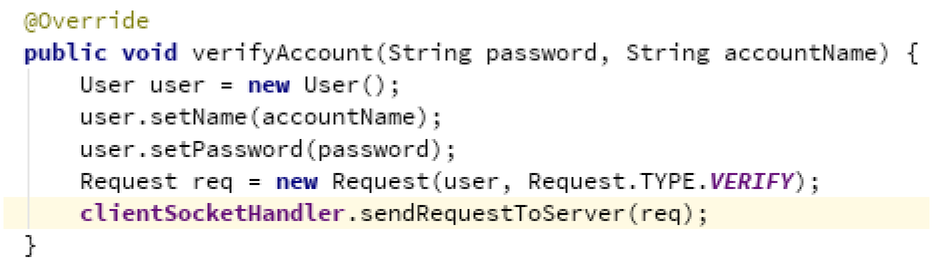
The verifyAccount method, which is called on the LogInModel interface, takes two arguments, password and username. These two values are retrieved from the login GUI and are then passed on from the model interface to the client interface. See code example below.



```
@Override
public void verifyAccount(String password, String accountName) {
    client.verifyAccount(password, accountName);
}
```

Figure 22 - Class: LogInModel

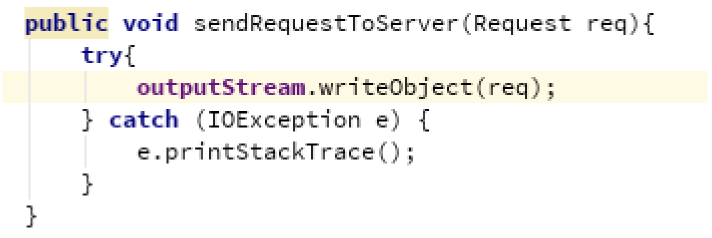
The implementation of verifyAccount called by the client, can be found in the SocketClient class. The method creates a new user object and sets its username field to be the given username and sets its password field to be the given password. It then creates a new Request of type "verify", which takes the newly created user as argument. Lastly, the method sendRequestToServer which takes the created Request as argument, is being called on the ClientSocketHandler class.



```
@Override
public void verifyAccount(String password, String accountName) {
    User user = new User();
    user.setName(accountName);
    user.setPassword(password);
    Request req = new Request(user, Request.TYPE.VERIFY);
    clientSocketHandler.sendRequestToServer(req);
}
```

Figure 23 - Class: SocketClient

In the ClientSocketHandler class lies the functionality behind the method sendRequestToServer. The method attempts to send the argument of type Request to the server by calling writeObject on ObjectOutputStream.

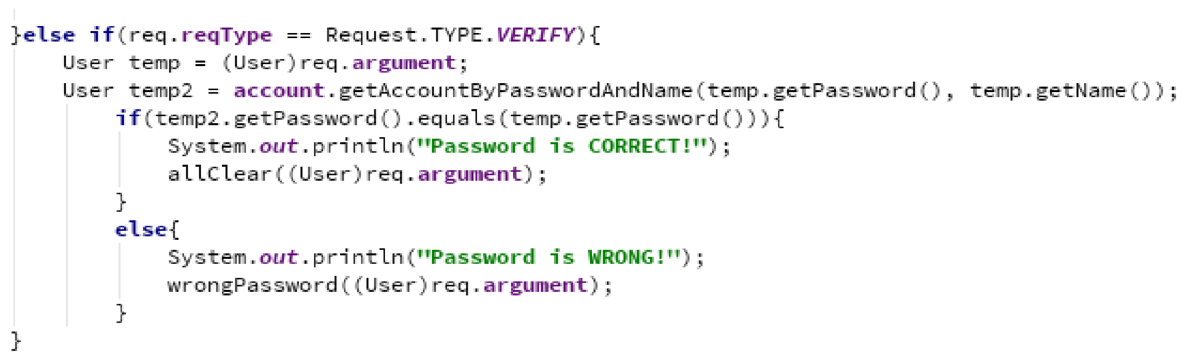


```
public void sendRequestToServer(Request req){
    try{
        outputStream.writeObject(req);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 24 - Class: ClientSocketHandler

On the server side, the ServerSocketHandler attempts to read the object sent from the client side in it's run method. The ServerSocketHandler then checks the type of the request, in this case the type is "verify".

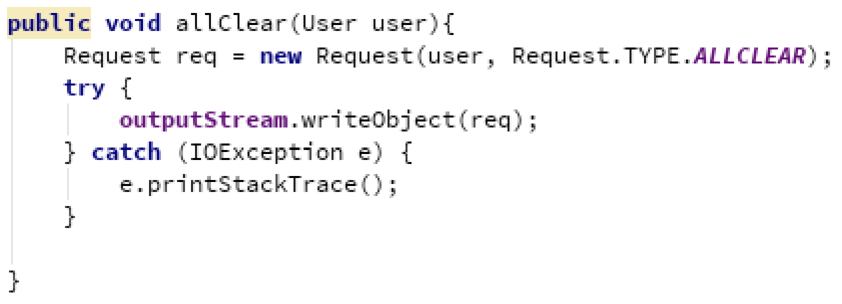
It will then create a temporary user setting its information to be equal to the argument sent from the client. It will then create another temporary user which information is set to be equal to the return statement of the method "getAccountByPasswordAndName" which is called on the Account Interface. This method takes the given username and password from the GUI and looks for an already existing account with a matching username.



```
}else if(req.reqType == Request.TYPE.VERIFY){
    User temp = (User)req.argument;
    User temp2 = account.getAccountByPasswordAndName(temp.getPassword(), temp.getName());
        if(temp2.getPassword().equals(temp.getPassword())){
            System.out.println("Password is CORRECT!");
            allClear((User)req.argument);
        }
        else{
            System.out.println("Password is WRONG!");
            wrongPassword((User)req.argument);
        }
}
```

Figure 25 - Class: ServerSocketHandler

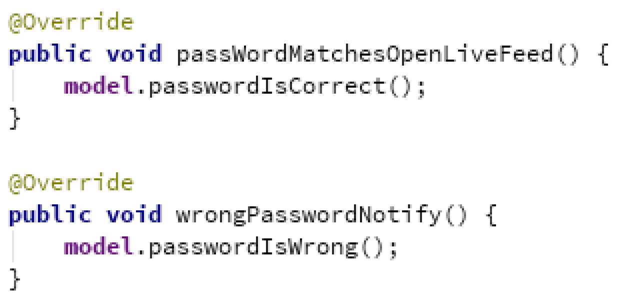
When the account has been found, it checks if the given password and the existing password are matching. If the passwords match, the SocketServerHandler calls its method "allClear" which attempts to send a new request to the client. Stating that the given password is correct, and that the user can proceed to the LiveFeedView. If the password is incorrect it will instead call the "wrongPassword" method.



```
public void allClear(User user){
    Request req = new Request(user, Request.TYPE.ALLCLEAR);
    try {
        outputStream.writeObject(req);
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

Figure 26 - Class: ServerSocketHandler

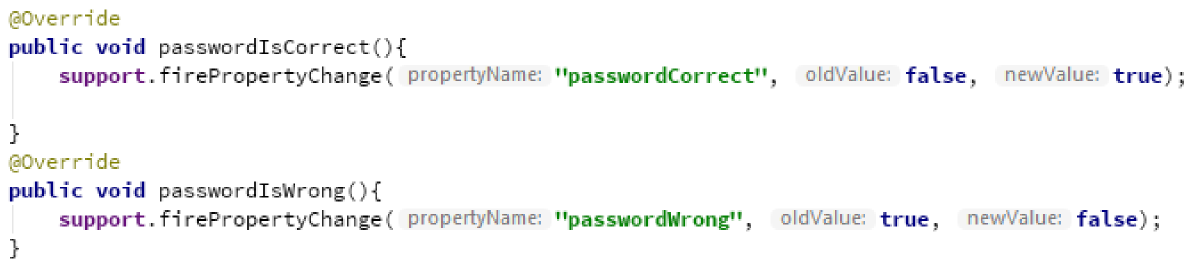
The ClientSocketHandler then receives the new Request in it's run method and proceeds with calling either "passWordMatchesOpenLiveFeed" or "wrongPasswordNotify" on the client interface, which implementation lies in the SocketClient class. The SocketClient then calls either "passwordIsCorrect" or “passwordIsWrong” on the LogInModel interface. The implementation of these two methods exist in the DataModelImpl class.



```
@Override
public void passWordMatchesOpenLiveFeed() {
    model.passwordIsCorrect();
}

@Override
public void wrongPasswordNotify() {
    model.passwordIsWrong();
}
```

Figure 27 - Class: SocketClient

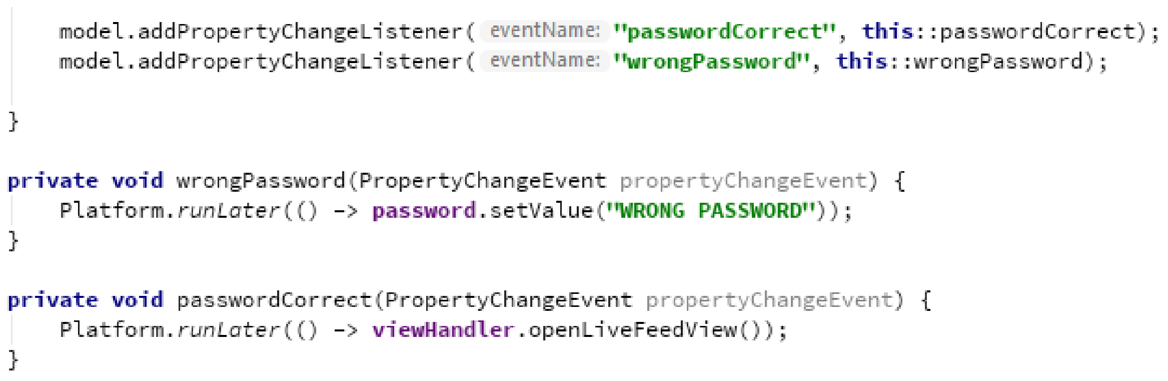
The DataModelImpl class, which implements the LogInModel interface calls the “firePropertyChange” method on the already existing java::beans::PropertyChangeSupport class.



```
@Override
public void passwordIsCorrect(){
    support.firePropertyChange( propertyName: "passwordCorrect",  oldValue: false,  newValue: true);

}
@Override
public void passwordIsWrong(){
    support.firePropertyChange( propertyName: "passwordWrong",  oldValue: true,  newValue: false);
}
```

Figure 28 - Class: DataModelImpl

Two listeners inside the LogInViewModel class then listens for the propertyChangeEvent and depending on which one is being fired, it either opens liveFeedView or sets the password TextField to “Wrong password”



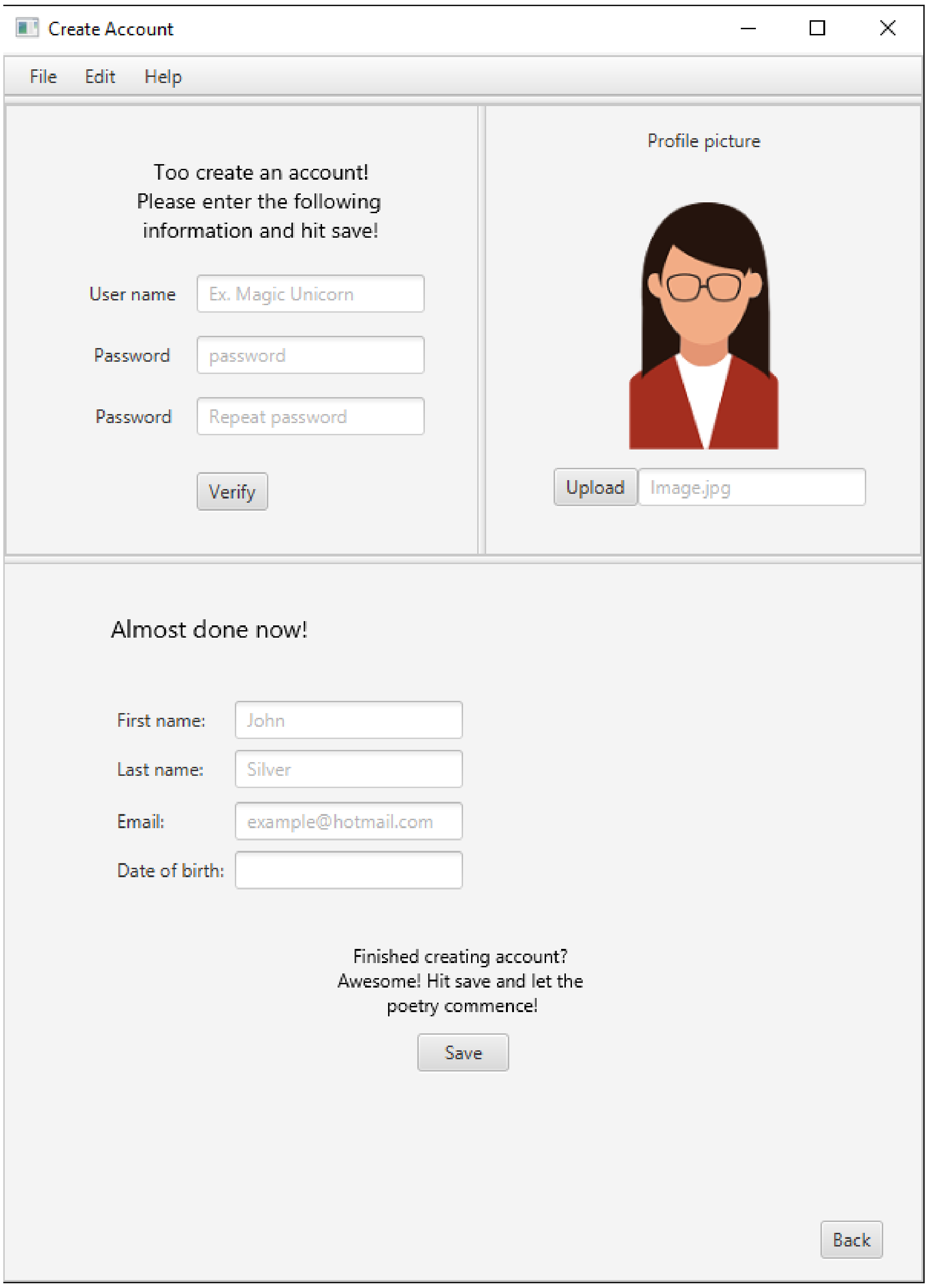
```
    model.addPropertyChangeListener( eventName: "passwordCorrect", this::passwordCorrect);
    model.addPropertyChangeListener( eventName: "wrongPassword", this::wrongPassword);

}

private void wrongPassword(PropertyChangeEvent propertyChangeEvent) {
    Platform.runLater(() -> password.setValue("WRONG PASSWORD"));
}

private void passwordCorrect(PropertyChangeEvent propertyChangeEvent) {
    Platform.runLater(() -> viewHandler.openLiveFeedView());
}
```

Figure 29 - Class: LogInViewModel

End of following the functionality behind the sequence of logging in.

Following the functionality behind the sequence of creating an account

If the user does not have an already existing account one must be created before being able to log in. The user must click on the "create account" button which triggers the "onCreateAccountAction" method inside the LogInView class which takes an ActionEvent as argument. Inside the LogInViewModel the openCreateAccountView method will then be called on the ViewHandler class which will load the GUI. See image of GUI below.



Too create an account!
Please enter the following
information and hit save!

User name	Ex. Magic Unicorn
Password	password
Password	Repeat password

Image.jpg

Almost done now!

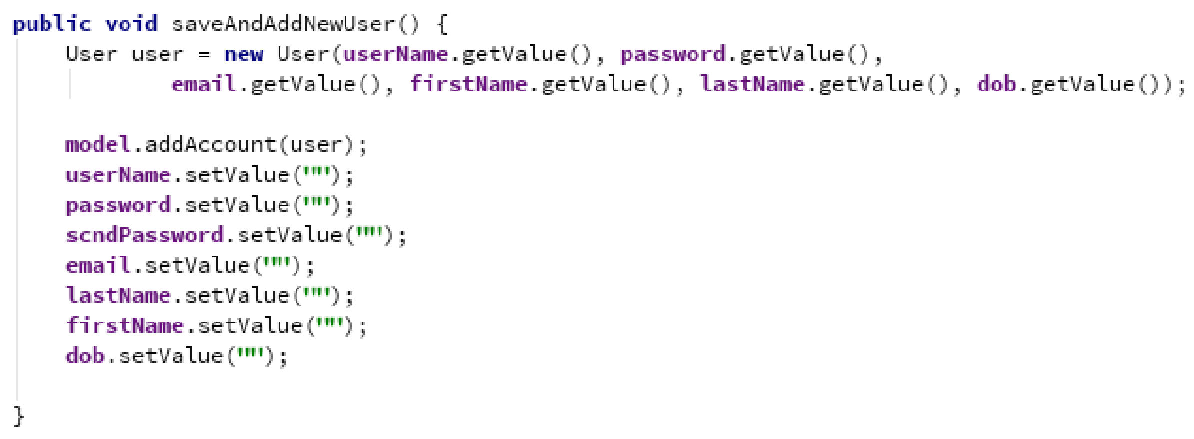
First name:	John
Last name:	Silver
Email:	example@hotmail.com
Date of birth:	

Finished creating account?
Awesome! Hit save and let the
poetry commence!

Figure 30 - Create account GUI

When the user has inserted the required information and pressed the save button, the "onSaveAction" method which takes an ActionEvent as argument inside the CreateAccountView class will be called. This method then calls "saveAndAddNewUser" on the CreateAccountViewModel class.

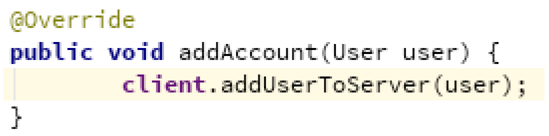
The method creates a new user which takes all of the given information inside the create account GUI. Afterwards it calls the "addAccount" method which takes the newly created user as argument. This method is called on the LogInModel interface.



```java
public void saveAndAddNewUser() {
    User user = new User(userName.getValue(), password.getValue(),
            email.getValue(), firstName.getValue(), lastName.getValue(), dob.getValue());

    model.addAccount(user);
    userName.setValue("");
    password.setValue("");
    scndPassword.setValue("");
    email.setValue("");
    lastName.setValue("");
    firstName.setValue("");
    dob.setValue("");

}
```

Figure 31 - Class: CreateAccountViewModel

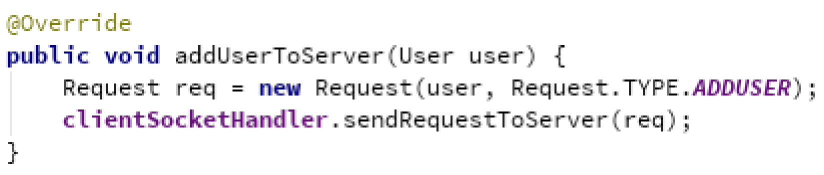
The implementation of the method "addAccount" exists in the DataModelImpl class. Here the method proceeds with calling the "addUserToServer" on the client interface. This method takes the user as argument.



```java
@Override
public void addAccount(User user) {
        client.addUserToServer(user);
}
```

Figure 32 - Class: DataModelImpl

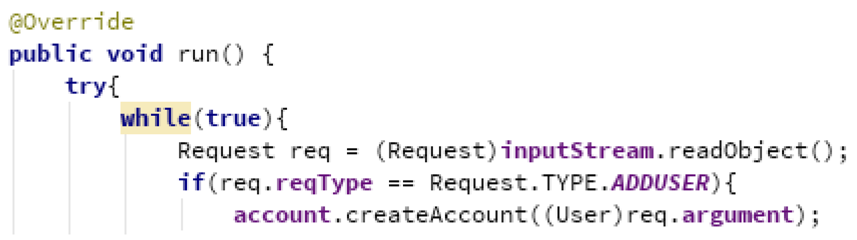
The implementation of the method "addUserToServer" exists in the SocketClient class. Where it creates a new Request of type "ADDUSER" and takes the user as argument. It then calls the sendRequestToServer on the ClientSocketHandler which attempts to send the request to the server side.



```java
@Override
public void addUserToServer(User user) {
    Request req = new Request(user, Request.TYPE.ADDUSER);
    clientSocketHandler.sendRequestToServer(req);
}
```

Figure 33 - Class: SocketClient

The request is then caught by the ServerSocketHandler in its run method. Which then calls the "createAccount" method which takes the argument of the request (which is the user) as argument. This method is being called on the Account interface.

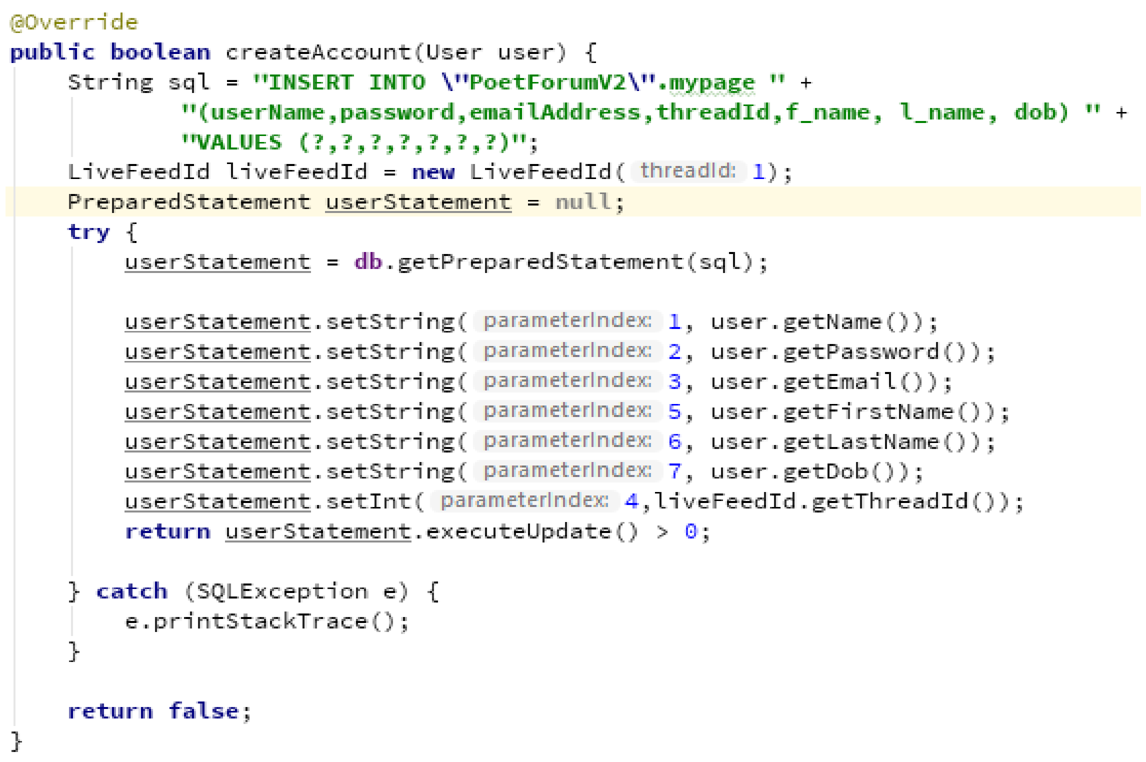


```
@Override
public void run() {
    try{
        while(true){
            Request req = (Request)inputStream.readObject();
            if(req.reqType == Request.TYPE.ADDUSER){
                account.createAccount((User)req.argument);
```

Figure 34 - Class: ServerSocketHandler

The implementation of the createAccount method which takes the user as argument, exists in the AccountImpl class. It creates a sql statement of type String. It then creates a new LiveFeedId which is set to be 1. Afterwards it will attempt to create a new "userStatement" which is gotten from the DBSHelper class with the method "getPreparedStatement" which takes a String as argument.

The method createAccount then creates a new row in the MyPage table, in which it sets each column value to be equal to the corresponding values which were given in the GUI. It then updates the userStatement with the "executeUpdate" method, if its bigger than 0.



```
@Override
public boolean createAccount(User user) {
    String sql = "INSERT INTO \"PoetForumV2\".mypage " +
            "(userName,password,emailAddress,threadId,f_name, l_name, dob) " +
            "VALUES (?,?,?,?,?,?,?)";
    LiveFeedId liveFeedId = new LiveFeedId( threadId: 1);
    PreparedStatement userStatement = null;
    try {
        userStatement = db.getPreparedStatement(sql);

        userStatement.setString( parameterIndex: 1, user.getName());
        userStatement.setString( parameterIndex: 2, user.getPassword());
        userStatement.setString( parameterIndex: 3, user.getEmail());
        userStatement.setString( parameterIndex: 5, user.getFirstName());
        userStatement.setString( parameterIndex: 6, user.getLastName());
        userStatement.setString( parameterIndex: 7, user.getDob());
        userStatement.setInt( parameterIndex: 4,liveFeedId.getThreadId());
        return userStatement.executeUpdate() > 0;

    } catch (SQLException e) {
        e.printStackTrace();
    }

    return false;
}
```

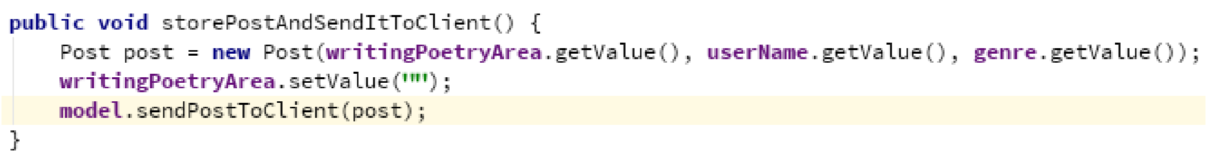
Figure 35 - Class: AccountImpl

End of following the functionality behind the sequence of creating an account. See full sequence

Following the functionality behind the sequence of post poetry

When a user wishes to post some poetry, he or she must type some text inside the textArea of the liveFeed GUI and then press the post button. This will trigger the “onPostAction” method which takes an ActionEvent as argument. The method exists in the LiveFeedView class. The method itself then calls “storePostAndSendItToClient” on the LiveFeedViewModel class.

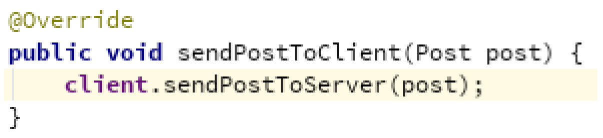
Inside the LiveFeedViewModel class, the method “storePostAndSendItToClient” is being called.
It creates a new post of type Post which takes the written poetry, username and the poetry genre as argument. It then resets the textArea and calls “sendPostToClient” on the LogInModel interface. The “sendPostToClient” method takes the newly creates post as argument.



```
public void storePostAndSendItToClient() {
    Post post = new Post(writingPoetryArea.getValue(), userName.getValue(), genre.getValue());
    writingPoetryArea.setValue("");
    model.sendPostToClient(post);
}
```

Figure 36 - Class: LiveFeedViewModel

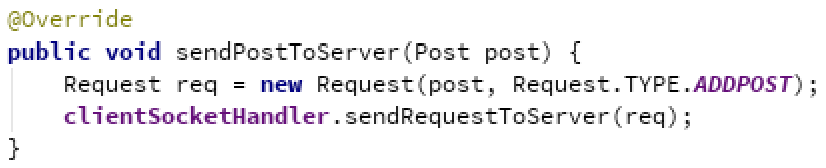
The implementation of the “sendPostToClient” method exists in the DataModelImpl. The method calls “sendPostToServer”, taking the post as argument. This method is being called on the Client interface.



```
@Override
public void sendPostToClient(Post post) {
    client.sendPostToServer(post);
}
```

Figure 37 - Class: DataModelImpl

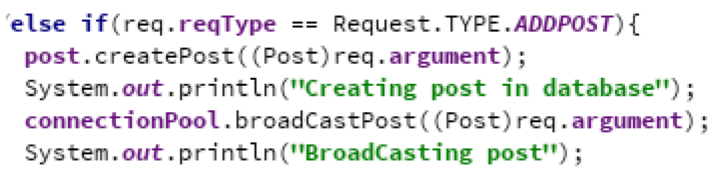
Inside the SocketClient class lies the implementation of the “sendPostToServer” method. It takes a post as argument. The method creates a new Request of type “ADDPOST” and it takes the post as argument.
The method then calls sendRequestToServer on the ClientSocketHandler, taking the newly created request as argument. The request is then being send from the ClientSocketHandler to ServerSocketHandler.



```
@Override
public void sendPostToServer(Post post) {
    Request req = new Request(post, Request.TYPE.ADDPOST);
    clientSocketHandler.sendRequestToServer(req);
}
```

Figure 38 - Class: SocketClient

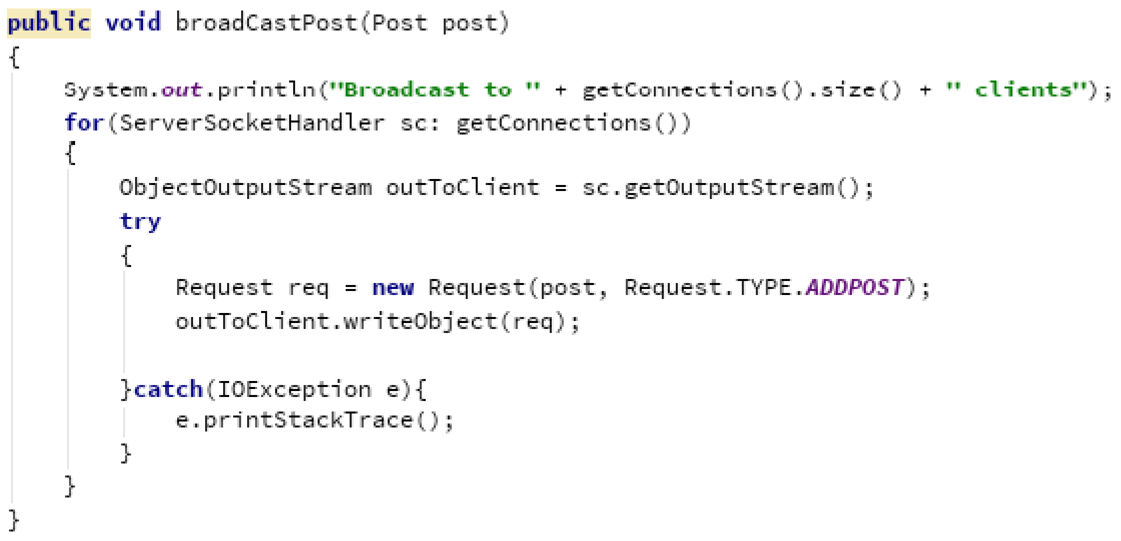
When the ServerSocketHandler class receives the request in its run method, it calls “createPost” on the PostInter class, taking the argument of the request (which is the post) as argument. This method will in turn create a new row in the table PostTable and insert the given values in each column. The method then calls “broadCastPost” on the ConnectionPool class.



```java
else if(req.reqType == Request.TYPE.ADDPOST){
 post.createPost((Post)req.argument);
 System.out.println("Creating post in database");
 connectionPool.broadCastPost((Post)req.argument);
 System.out.println("BroadCasting post");
```

Figure 39 - Class: ServerSocketHandler

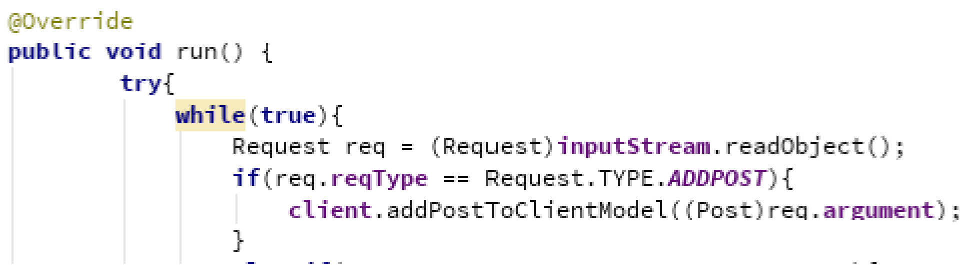
The ConnectionPool class has the “broadCastPost” method, which goes through a loop of all connected clients, and for each client sends a new Request of type “ADDPOST” with the post as argument to the Client side.



```java
public void broadCastPost(Post post)
{
    System.out.println("Broadcast to " + getConnections().size() + " clients");
    for(ServerSocketHandler sc: getConnections())
    {
        ObjectOutputStream outToClient = sc.getOutputStream();
        try
        {
            Request req = new Request(post, Request.TYPE.ADDPOST);
            outToClient.writeObject(req);

        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

Figure 40 - Class: ConnectionPool

The ClientSocketHandler will then receive the “ADDPOST” request in its run method and thus call the “addPostToClientModel” method each time. This method takes the argument of the request (which is the post) as argument.

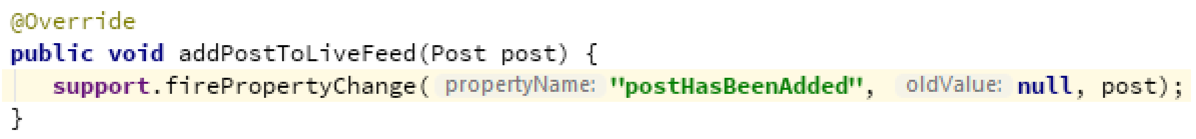


```java
@Override
public void run() {
        try{
            while(true){
                Request req = (Request)inputStream.readObject();
                if(req.reqType == Request.TYPE.ADDPOST){
                    client.addPostToClientModel((Post)req.argument);
                }
```

Figure 41 - Class: ClientSocketHandler

Inside the SocketClient exists the implementation of the “addPostToClientModel” method. The SocketClient simply calls “addPostToLiveFeed”, taking the post as argument. This method is being called on the LogInModel interface.

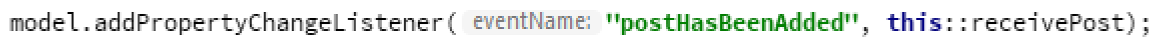
Inside the DataModelImpl exists the implementation of the “addPostToLiveFeed” method. This method fires a property change event on the java::beans::PropertyChangeSupport class. The property name being “postHasBeenAdded”, and the new value of the event being post.



```
@Override
public void addPostToLiveFeed(Post post) {
    support.firePropertyChange( propertyName: "postHasBeenAdded",  oldValue: null, post);
}
```

Figure 42 - Class: DataModelImpl

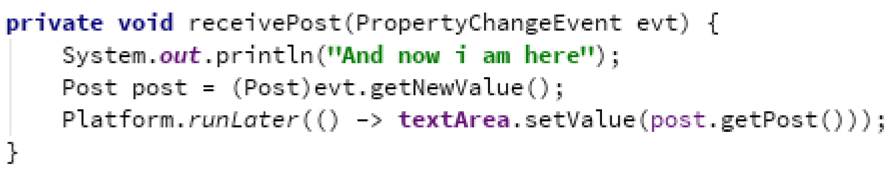
The LiveFeedViewModel class then listens for when this event is being fired.



```
model.addPropertyChangeListener( eventName: "postHasBeenAdded", this::receivePost);
```

Figure 43 - Class: LiveFeedViewModel

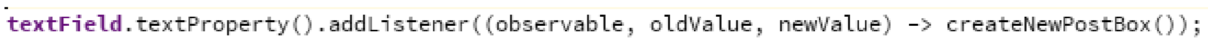
When the propertyChange has been fired in the DataModelImpl, the listener inside the LiveFeedViewModel will trigger the “receivePost” method, which takes a PropertyChangeEvent as argument. This method will create a new post and set it be equal to the new value of the event. The method will then set the value of the textArea of the LiveFeedView to be equal to the poetry inside the post Object.



```
private void receivePost(PropertyChangeEvent evt) {
    System.out.println("And now i am here");
    Post post = (Post)evt.getNewValue();
    Platform.runLater(() -> textArea.setValue(post.getPost()));
}
```

Figure 44 - Class: LiveFeedViewModel

Finally, the LiveFeedView class adds a listener to its textField inside the “init” method.



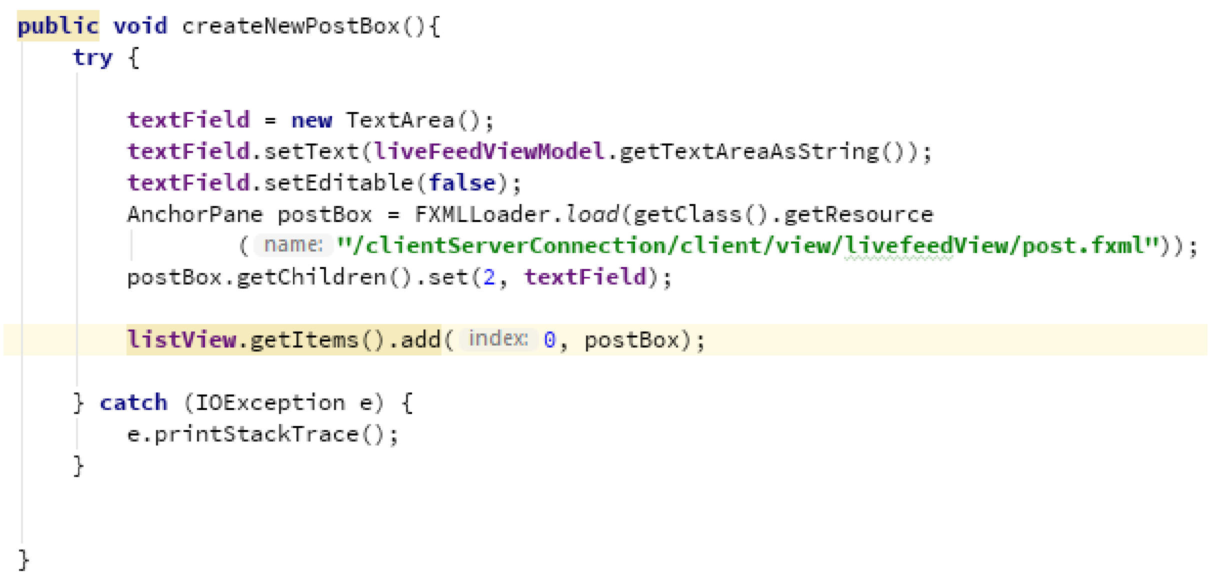
```
textField.textProperty().addListener((observable, oldValue, newValue) -> createNewPostBox());
```

Figure 45 - Class: LiveFeedView

When a new value has been set on the textField it triggers the “createNewPostBox” method to run. The “createNewPostBox” method creates a new textField and sets its text to be equal to what the textArea was set to inside the LiveFeedViewModel.

It then loads an external fxml file called “post” which contains all the desired elements a post should have. It then replaces the TextArea element at index 2, with the newly created textField.

Lastly it adds the “postBox” to a JFXListView. Note that JFoenix libraries have been imported to the LiveFeedView for the use of JFX elements.



```
public void createNewPostBox(){
    try {

        textField = new TextArea();
        textField.setText(liveFeedViewModel.getTextAreaAsString());
        textField.setEditable(false);
        AnchorPane postBox = FXMLLoader.load(getClass().getResource
                ( name: "/clientServerConnection/client/view/livefeedView/post.fxml"));
        postBox.getChildren().set(2, textField);

        listView.getItems().add( index: 0, postBox);

    } catch (IOException e) {
        e.printStackTrace();
    }


}
```

Figure 46 - Class: LiveFeedView

End of following the functionality behind the sequence of posting poetry.

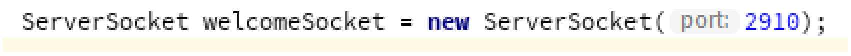
Client – Server Connection

The Client Server connection was established using sockets.

SocketServer Class:

The SocketServer creates a welcomeSocket of type ServerSocket in its “runServer” method.

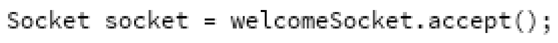
This welcomeSocket is assigned a port, “2910”, the port which the SocketClient will connect to.



```java
ServerSocket welcomeSocket = new ServerSocket( port: 2910);
```

Figure 47 - Initialization of a ServerSocket

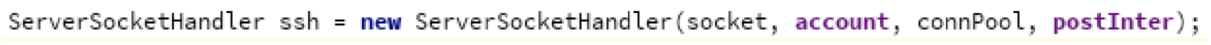
The runServer method will then accept the connecting client.



```java
Socket socket = welcomeSocket.accept();
```

Figure 48 - Acceptance of connecting clients

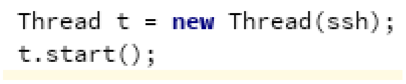
It Will create a new ServerSocketHandler with the socket being one of the arguments.



```java
ServerSocketHandler ssh = new ServerSocketHandler(socket, account, connPool, postInter);
```

Figure 49 - Initialization of a new ServerSocketHandler

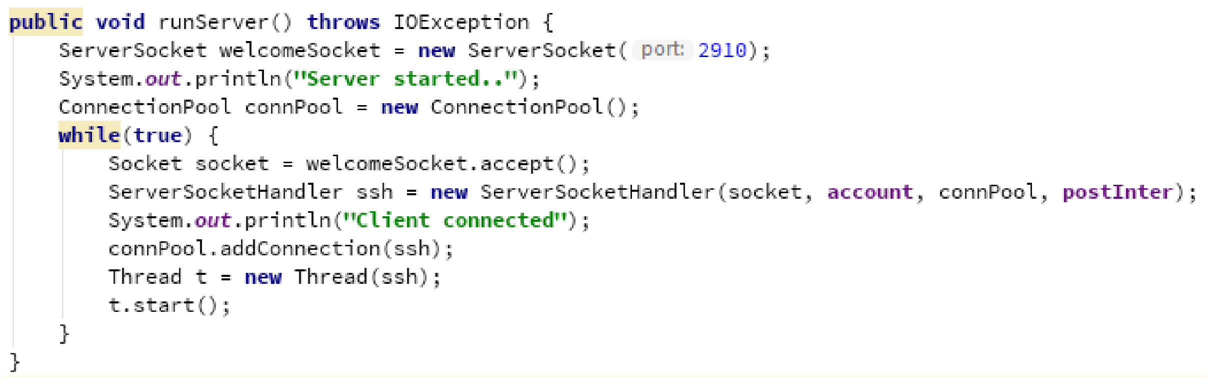
And then create a new Thread which takes the newly created ServerSocketHandler as argument and lastly starts the Thread.



```java
Thread t = new Thread(ssh);
t.start();
```

Figure 50 - Thread initialization and starting

See entire method below.

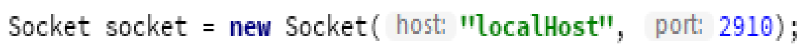


```java
public void runServer() throws IOException {
    ServerSocket welcomeSocket = new ServerSocket( port: 2910);
    System.out.println("Server started..");
    ConnectionPool connPool = new ConnectionPool();
    while(true) {
        Socket socket = welcomeSocket.accept();
        ServerSocketHandler ssh = new ServerSocketHandler(socket, account, connPool, postInter);
        System.out.println("Client connected");
        connPool.addConnection(ssh);
        Thread t = new Thread(ssh);
        t.start();
    }
}
```

Figure 51 - Full overview of the above-mentioned figures.

SocketClient class:

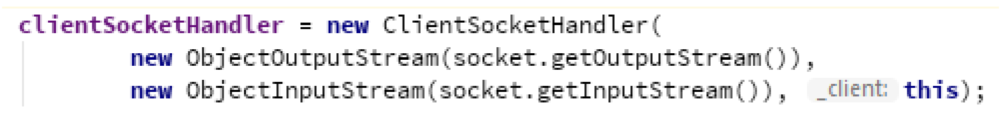
The SocketClient class which Implements the Client interface attempts to create a socket of Type Socket in its constructor. The socket takes two arguments, "localhost" as host and port as "2910".



```
Socket socket = new Socket( host: "localHost",  port: 2910);
```

Figure 52 - New socket initialization

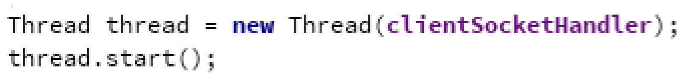
It then creates a new ClientSocketHandler, which takes ObjectInputStream and ObjectOutputStream as arguments. The "socket.getOutputStream()" and "socket.getInputStream()", enables the ClientSocketHandler to receive Objects and Send Objects to the Server side.



```
clientSocketHandler = new ClientSocketHandler(
        new ObjectOutputStream(socket.getOutputStream()),
        new ObjectInputStream(socket.getInputStream()),  _client: this);
```

Figure 53 - ClientSocketHandler initialization and creation of ObjectIn- & OutputStreams

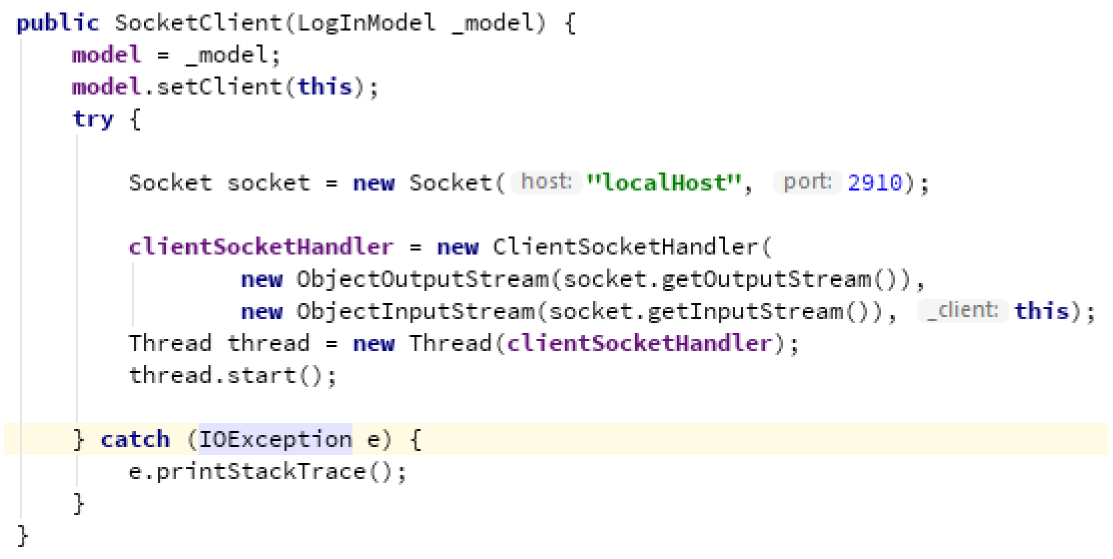
The SocketClient then creates a new Thread which takes a ClientSocketHandler and then starts the Thread.



```
Thread thread = new Thread(clientSocketHandler);
thread.start();
```

Figure 54 - Thread initialization for the ClientSocketHandler.

See entire constructor below.



```
public SocketClient(LogInModel _model) {
    model = _model;
    model.setClient(this);
    try {

        Socket socket = new Socket( host: "localHost",  port: 2910);

        clientSocketHandler = new ClientSocketHandler(
                new ObjectOutputStream(socket.getOutputStream()),
                new ObjectInputStream(socket.getInputStream()),  _client: this);
        Thread thread = new Thread(clientSocketHandler);
        thread.start();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

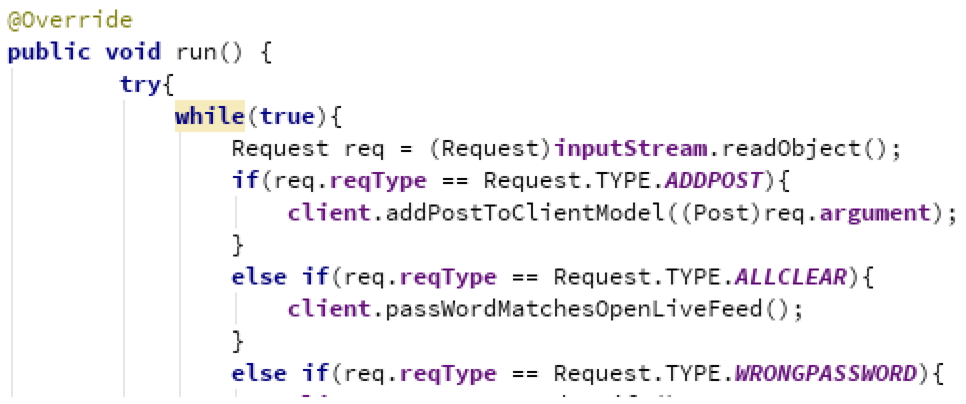
Figure 55 - Full overview of the above-mentioned figures

ClientSocketHandler Class:

The class implements the already existing java Runnable interface.

The ClientSocketHandler has three private field variables. ObjectOutputStream, ObjectInputStream and a Socket. Which are included as arguments for the constructor.

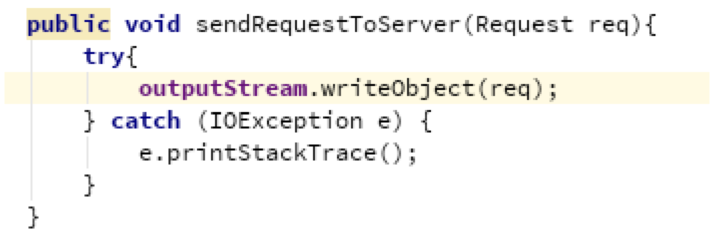
In its run method it attempts to create a new Request which is set to be equal to the Object being sent from the server. inputStream.readObject, the Object being sent from the server, in this case, a request. It then goes through all of the if else statements to check which request it has received and what to do then.



```
@Override
public void run() {
        try{
            while(true){
                Request req = (Request)inputStream.readObject();
                if(req.reqType == Request.TYPE.ADDPOST){
                    client.addPostToClientModel((Post)req.argument);
                }
                else if(req.reqType == Request.TYPE.ALLCLEAR){
                    client.passWordMatchesOpenLiveFeed();
                }
                else if(req.reqType == Request.TYPE.WRONGPASSWORD){
```

Figure 56 - Checking for objects equal to the one sent from the server

This system only sends requests between the Server side and Client side. Therefore, the ClientSocketHandler has a "sendRequestToServer" method, which takes a Request as argument. This method is responsible for sending a variety of Requests to the server side.



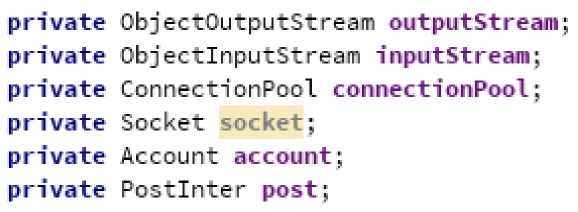
```
public void sendRequestToServer(Request req){
    try{
        outputStream.writeObject(req);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 57 - Sending request to the server

ServerSocketHandler Class:

The class implements the already existing java Runnable interface.

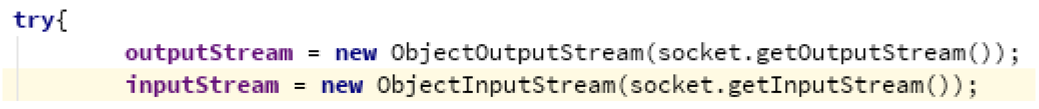
The ServerSocketHandler has six private field variables.



```java
private ObjectOutputStream outputStream;
private ObjectInputStream inputStream;
private ConnectionPool connectionPool;
private Socket socket;
private Account account;
private PostInter post;
```

Figure 58 - Private field variables from ServerSocketHandler

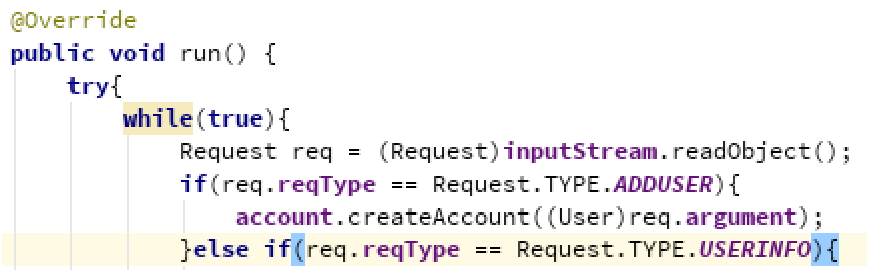
In its constructor the class attempts to set outputStream and inputStream. This establishes the connection between the ClientSocketHandler and the ServerSocketHandler, as inputStream becomes the Object which is sent from the client and outputStream becomes the Object which is sent from the server to the client.



```java
try{
        outputStream = new ObjectOutputStream(socket.getOutputStream());
        inputStream = new ObjectInputStream(socket.getInputStream());
```

Figure 59 - New ObjectIn & OutputStreams

In its run method, the class attempts to read each request being sent from the server to then execute the corresponding methods.



```java
@Override
public void run() {
    try{
        while(true){
            Request req = (Request)inputStream.readObject();
            if(req.reqType == Request.TYPE.ADDUSER){
                account.createAccount((User)req.argument);
            }else if(req.reqType == Request.TYPE.USERINFO){
```

Figure 60 - Run method to read requests from the server

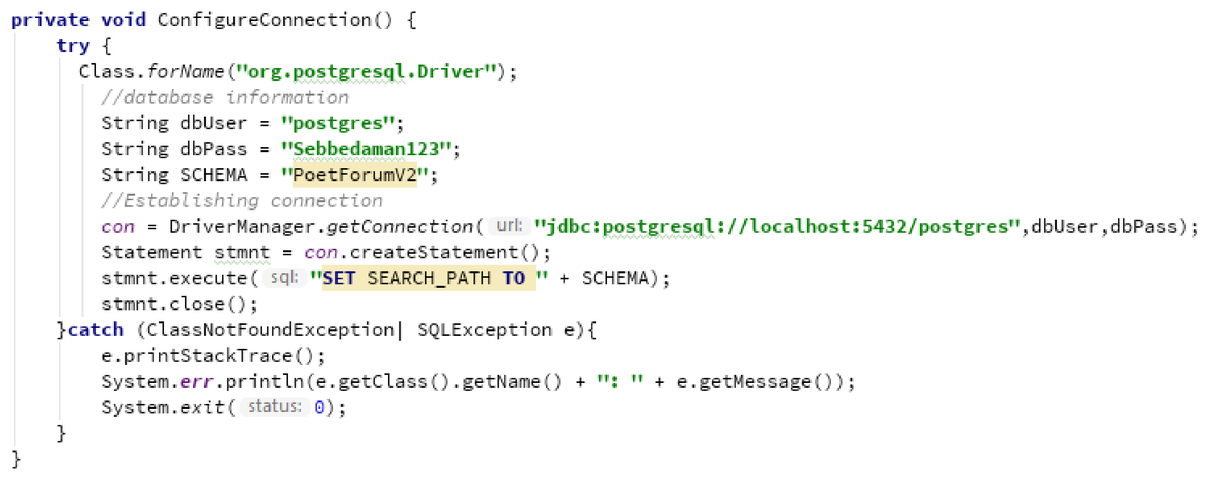
Database Connection

Database connection was established through ‘ConfigureConnection’ method inside DBSHelper class.

-Configuration () method:

First of all, “ConfigurationConnection()” method defines the chosen databases information in order to open needed pgadmin4 server. Then it creates connection by creating “con” variable using already existing java DriverManager class where it calls “getConnection()” method on it with previously defined database information. This method takes url, server username and server password as attributes with type String. After that it Creates a field variable called “stmnt” with already existing java Statement interface, and this variable get its value from con variables “createStatment ()” method.

“Stmnt” variable then runs “execute()” method which takes sql as argument with type String. It specifies which Schema will be used from chosen database server. Lastly “stmnt” is going to call “close()” method to shoot running database connection down.



```
private void ConfigureConnection() {
    try {
      Class.forName("org.postgresql.Driver");
        //database information
        String dbUser = "postgres";
        String dbPass = "Sebbedaman123";
        String SCHEMA = "PoetForumV2";
        //Establishing connection
        con = DriverManager.getConnection( url: "jdbc:postgresql://localhost:5432/postgres",dbUser,dbPass);
        Statement stmnt = con.createStatement();
        stmnt.execute( sql: "SET SEARCH_PATH TO " + SCHEMA);
        stmnt.close();
    }catch (ClassNotFoundException| SQLException e){
        e.printStackTrace();
        System.err.println(e.getClass().getName() + ": " + e.getMessage());
        System.exit( status: 0);
    }
}
```

Figure 61 - Database implementation

Test

Blackbox testing with test cases

Blackbox testing has been made to test the programs functionality without having insight in the code. The Blackbox testing was made with the test cases made in the analysis section. The table below describes the different levels of error that a test case can have.

Table 3 - Categories of errors

CATEGORY	DESCRIPTION	ACTION
1.	Fatal	Catastrophic errors. System is unusable. The test is rejected. The error must be corrected before the system can be accepted. This type of errors requires a completely new test be made.
2.	Critical	Critical errors. The error cannot be dealt with, but the system can still be used (for instance by using alternate means of control or configuration). The error must be corrected as fast as possible.
3.	Major	Significant error. The error must be corrected as fast as possible.
4.	Minor	Minor error on the functional level. The error must be corrected as fast as possible.
5.	Slight imperfections	Slight imperfections (spelling mistakes, formatting and placing of graphical objects). The imperfection is corrected when convenient.
6.	Improvements	Suggestions for improvements and/or changes.

Only the use-cases that have been implemented will be tested with Blackbox testing. Some test-cases tests two use-case descriptions which is the reason only four test scenarios are tested, but there are six implemented requirements.

Table 4 - Test case results

Test Scenario	Action	Error ref	Cat 1	Cat 2	Cat 3	Cat 4	Cat 5	Cat 6
1	4.1	1				X		
1	4.3	2				X		
2	3.2	3			X			
2	3.3	4			X			
2	4.1	5		X				
2	4.2	6		X				
2	4.3	7			X			
3	3.1	8	X					

In test scenario 1.4.1 the system would post the same message twice.

In test scenario 1.4.3 the system would post an empty post if the user had already posted a post that was not empty.

In test scenario 2.3.2 fields can be left empty.

In test scenario 2.3.3 a picture could not be uploaded on the create account page.

In test scenario 2.4.1 two accounts can have the same username.

In test scenario 2.4.2 two accounts can have the same e-mail

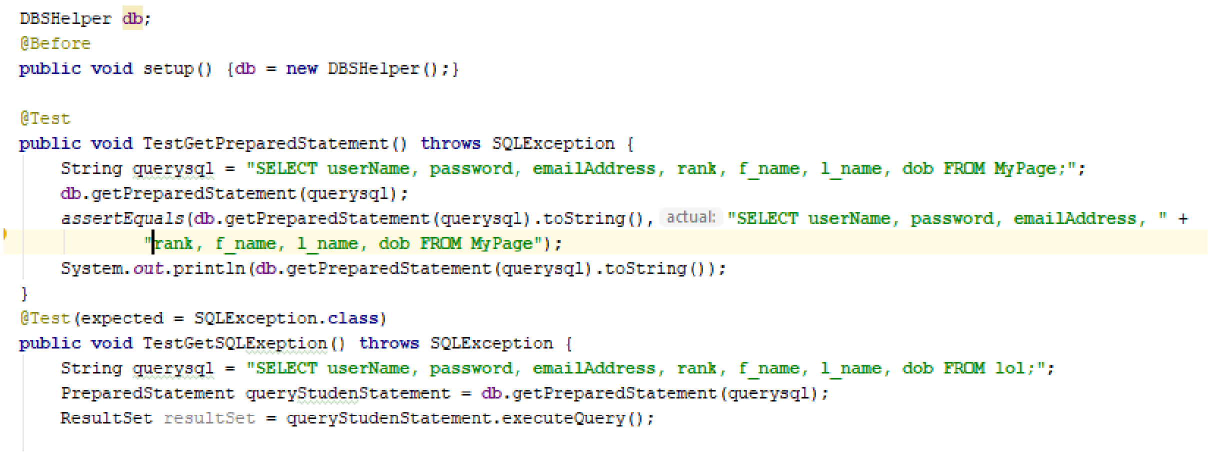
In test scenario 2.4.3 the system is not checking for empty fields.

In test scenario 3.3.1 a user would crash the entire server if they used wrong log in information. To solve this, it is required to restart the entire system, including the server.

Junit for a class

Poet Forum has been tested using both White- and Black-box testing. White box testing has access to the actual code, and it follows the implementation with all its methods, sections and exceptions. It is from the programmer's point of view, unlike black-box which follows the users view.

Poet forum has been tested with white- box testing on DBSHelper class where getPrepareStatement() method was tested with two different tests. To do this first, setup() method was declared where DBSHelper field had been created. Then under the first @Test prepareStatement() method was checked whether it gives back the expected values or not. With the second @Test, it was expected to get a SQLExeption, to achieve that, we created an incorrect query what has been executed. Both tests passed. See Test implementation down below.



```
DBSHelper db;
@Before
public void setup() {db = new DBSHelper();}

@Test
public void TestGetPreparedStatement() throws SQLException {
    String querysql = "SELECT userName, password, emailAddress, rank, f_name, l_name, dob FROM MyPage;";
    db.getPreparedStatement(querysql);
    assertEquals(db.getPreparedStatement(querysql).toString(), actual: "SELECT userName, password, emailAddress, " +
            "rank, f_name, l_name, dob FROM MyPage");
    System.out.println(db.getPreparedStatement(querysql).toString());
}
@Test(expected = SQLException.class)
public void TestGetSQLExeption() throws SQLException {
    String querysql = "SELECT userName, password, emailAddress, rank, f_name, l_name, dob FROM lol;";
    PreparedStatement queryStudenStatement = db.getPreparedStatement(querysql);
    ResultSet resultSet = queryStudenStatement.executeQuery();
```

Figure 62 - JUnit testing class.

Results and Discussion

Results

Based on the Blackbox test in the test section where all the implemented use-case descriptions we can deduct which user stories have been implemented and which user stories have not been implemented.

Functional requirements

Table 5 - List of implemented and not implemented features

User-story	**Implemented / Not implemented**
As a user I want to be able to create my own personal account, so I can participate in the community.	**Implemented**
As a user I want to be able to login, so I can use the system.	**Implemented**
As a user I wish to be able to make post(s) so that my poetry is stored in a feed.	**Implemented**
As a user I want my posted poetry to be visible to other users, so other people can enjoy it.	**Implemented**
As a user I wish to see a common feed between all users, where all posted poetry is displayed, so that I can browse poetry posts.	**Implemented**
As a user I want to have my own personal page, which displays information about me such as first name, last name, email, date of birth, username and a picture of my choice.	**Implemented**
As a user I want the ability to remove a post I have posted, so I can remove them from the forum.	**Not implemented**
As a user I want to be able to edit my posts, so I can correct mistakes.	**Not implemented**
As a user i want to be able to delete my account so that I can leave the community.	**Not implemented**
As a user I want to be able to comment on others poetry so that I can give constructive feedback.	**Not implemented**
As a user I want to be able to have a friends list, so I can have a small community within a larger one.	**Not implemented**

As a user I want to be able to send friend requests to other users, so I can build my own personal community.	**Not implemented**
As a user I want to be able to remove my comments so that I can remove my input if I consider it inappropriate.	**Not implemented**
As a user I wish to have my own posted poetry displayed on my account/profile, so people can see a compilation of my poetry.	**Not Implemented**
As a user I want to be able to sort all poetry in the common feed by rating, so I can see a list of the top-rated poetry.	**Not implemented**
As a user I want to be able to sort all poetry in the common feed by genre, so I can sort for the poetry I’m in the mood for.	**Not implemented**
As a user I want the ability to rate other people’s poetry so that we can have feedback from each other.	**Not implemented**
As a user I wish to see a leader board consisting of the top-rated poets, so I can see a list of the top-rated poets.	**Not implemented**
As a user I want to be able to delete friends, so I can manage my friends list.	**Not implemented**
As a user I want to be able to send private messages to other users so I can share my thoughts privately.	**Not implemented**
As a user I want to be able to receive private messages from other users, so I can talk in private.	**Not implemented**
As a user I want the ability to change my username, so I can keep it new and trendy.	**Not implemented**
As a user I want to be able to change my password, in order to keep my account secure.	**Not implemented**
As a user I want the friends list to display which users are currently active and which are not, so I can see a list of friends who are available to discuss poetry.	**Not implemented**
As a user I want the ability to report a post so that we can have inappropriate posts removed.	**Not implemented**

As a user I want the ability to ignore other accounts so that I will not see their poetry again.	**Not implemented**
As an admin I want to be able to delete other users' comments so that I can remove unwanted input.	**Not implemented**
As an admin I want to be able to delete other users posts so that we can avoid spam.	**Not implemented**
As an admin I want to be able to ban users so they can't bother other users.	**Not implemented**
As an admin I want to be able to lock a post so that no user can post further comments, in order to keep a civil tone on the forum.	**Not implemented**

Non-functional requirements

Table 6 - List of non-functional requirements

Non-functional requirements	**Comments**
The program must be coded in Java	**The system is coded in Java**
The program must store data in a database	**The system is storing data in a database**
The program must be operable with a keyboard and mouse	**The system is operable with keyboard and mouse.**

Discussion

The system has successfully implemented the six highest priority user stories. A user can now create an account, log in, and post poetry as well as view personal information. However, the aim of this project was to create a custom-made system for the poetry club at VIA university college, a system efficient enough to replace already existing ones such as Discord or Facebook. It can thus be established that the poetry forum lacks implementation and is not yet ready for launch as it also contains a fatal error.

As established the system lacks in implementation, view result section, however, the highest priority should be to fix the fatal error. The fatal error occurs as the user attempts to log in using the wrong user information, this action will throw a null pointer exception in the **ServerSocketHandler** which will prevent any client from connecting unless the server is restarted.

Furthermore, although this system is quite simple it has potential to become a quite large. Thus, perhaps a better solution to this project would have been to implement an RMI connection rather than a socket connection, as RMI is aimed for high-level Java-to-Java distributed computing whilst sockets suits better for low-level network communication. Although one could argue, that RMI is difficult to configure and deploy.

Additionally, socket connection provides no authentication nor security in comparison with RMI which handles basic network errors, authentication and security issues.

Conclusion

Considering all the things that have been stated in the earlier sections of this paper here is the conclusion which summarises every key aspect of our project.

The poetry club at university college requested a poetry forum. The solution to this request is the system that has been analysed, designed and implemented throughout the report. Although the system is functional and fulfils the highest prioritized requirements, however, it is falling short in comparison to the goal due to the lack of implemented requirements such as; remove post, edit post, comment and sort poetry etc. As of now the system contains a fatal error that was recently discovered. This error prevents the system from being launched unless it is corrected, or the login implementation withdrawn.

The remaining elements of the system that have been implemented are functional with some minor errors the system allowing empty text fields when creating accounts or posts. It should also be noted that the graphical user interface is not responsive.

In conclusion, despite the lack of implantation and the relatively high number of bugs, the system has the potential to become a quite large functional system due to its design the code. The systems design allows for easy modification of the code and classes can easily be extended.

Project future

Now that Poet Forum has been made It is essential to have an overview from a technical point of view and discuss the possible updates which can occur in the future.

The implementation has briefly followed the Solid principles with some minor mistakes, for example the “LogInModel” interface, which has a diverse set of functionalities is not cohesive. It is therefore recommended to divide the responsibility of the Model interface, to not violate the SOLID principles, specifically the Single responsibility principle. Other than that, the code can be simply modified and edited. Model View View Model (MVVM) has been chosen as design pattern in order to organize Poet forums implementation and divide responsibility between classes.

Furthermore, since it is expected that the poetry club will post a lot of poetry, project future could include a proxy pattern for the liveFeedView and the MyPageView, to not fully load all the objects and thus limit the strain on the system.

Additionally, a flyweight pattern could be implemented with the purpose to reuse views rather than creating new ones. The downside would be that each view would only be minimized while not in use, and thus require RAM and CPU power.

Moreover, implementing a singleton to log actions taking place in the system would be beneficial for an improved overview of what is happening inside the system.

It is recommended to fix the null pointer exception which is thrown in the **ServerSocketHandler** when a user attempts to log in and types in the wrong user information. The null pointer exception crashes the server, and no client can no longer connect unless the server is restarted.

Continuing the implementation of user stories, should prioritize finishing the comment functionality as well as the filtering of poetry in the live feed.

Sources of Information

Facebook, 2018. Datapolicy. [online] Available at: <www.facebook.com/terms.php> [Accessed 20 February 2019].

Discord,2017. About.[online] Available at: <https://discordapp.com/company>
[Accessed 21 February 2019].

Barton, J., 2017. VOIP Phone System Guide Proprietary vs Open Source Solutions. Pascom.net VOIP blog, [blog] 28 oktober. Available at: <https://www.pascom.net/en/blog/voip-phone-system-guide-proprietary-vs-open-source-solutions/>
[Accessed 21 February 2019].

JFoenix, 2015. JFoenix Libraries. (8.0.8). [external Java library] JFoenix. Available at: https://github.com/jfoenixadmin/JFoenix

The PostgreSQL Global Development Group. (42.2.5). [Binary Jar file] PostgreSQL. Available at: https://jdbc.postgresql.org/download.html

Appendices

Appendix A - Use case descriptions

Appendix B - All UML Diagrams

Appendix C - Test cases

Appendix D - User Guide

Appendix E - Installation guide

Appendix F - JFoenix

Appendix G - PostgreSQL

Appendix H - Project description

Appendix I - Source Code

Appendix J - Group Description from SEP1

Appendix K - SCRUM

VIA UNIVERSITY COLLEGE
ICT ENGINEERING

SEP2X-S19 Process Report

Group 5.

Members:

Lau Ravn Nielsen (280337)

Sébastien Malmberg (279937)

Tamás Fekete (266931)

Supervisors:

Ib Havn

Troels Mortensen

Number of characters:

17.717

Deadline:

01:00 pm.

07 / 06 / 2019

Contents

Introduction

SEP2 like SEP1 is an interdisciplinary course, where we as a group must draw on the knowledge and skills acquired from the courses running in parallel with SEP2 this semester. These courses consist of: SDJ2, where we have been taught Java and design patterns. SWE where we have been taught UML and software development processes. And finally, DBS, where we have been taught database implementation and the corresponding UML. All these courses have played a huge part in shaping the end-product in some way or another. However, SEP2 was different from SEP1 in the sense that we had the freedom to choose and find our own problem that needed solving. Our group ended up formulating a fictional case for a fictional poetry club at VIA university. The poetry club wished for a new forum, where they could have the opportunity to share their poetry with other members of the club. With the problem at hand we set out to try our best to create a solution to their problem by putting our newfound skills to the test.

Group description

The group consisted of four people when the project initially began. However, a group member realised that he was not where he wanted to be in life. So, Denis Turcu, choose to leave the project group and go back to Romania about half way through the semester. Losing Denis was detrimental to the group, he was a cornerstone in our group work and we could feel something missing after he left. Therefore, we feel like the best way to describe our group is to reference our group description from SEP1 with our E-stimate profiles and risk analysis, which can be found in appendix §§.

Project initiation

Forming the group that would end up being group 5 was simple. We had all worked together on SEP1 which turned out to be a success. During SEP1 we all learned how to work together, and how one another worked best, so we were ready to repeat the success.

The group had a thorough discussion revolving around what we should work on for our SEP2 project. We knew the requirements: It had to be a server-client system with a database connection.
There were multiple ideas on the table… We discussed a multiplayer tic-tac-toe, a speed-reading competition software, a forum or the more niche idea; a poetry forum. Denis had a passion for poetry, so the idea was initially his. We argued for and against every idea, however, Denis was adamant about choosing the poetry forum… and so we did with the hopes that Denis' motivation and interest in the subject would spread to the remaining members of the group. However, like SEP1 not much work was done during initiation. Everyone was waiting for directions from the supervisors before proceeding.

Project description

The project description took up most of the inception phase of our project. We worked tightly with the fictional customer, whom Denis was a mouthpiece for. During our initial version of the project description we struggled finding delimitations for our project and roughly planning our project with a milestone-line based on unified processes. However, with constructive feedback from the supervisors, we were on our way to correct our mistakes and make a better version of the project description. Which we did, we took the feedback to heart and applied our new knowledge to better our delimitations and correct the mistakes in our milestone-line. The final project description can be seen in Appendix §§.

It was during the Inception phase that we started having weekly meeting every Thursday when it was needed in terms of workload. If needed the group would have a meeting after school. However, these meetings were not documented with any minutes of meetings or scrum meetings, since we had not started on SCRUM yet.

Project execution

To guide us through the harsh problems that software development throws at groups, we set out to use SCRUM in connection with Unified processes to guide us through the storms. However, SCRUM and Unified processes brought new challenges to the already overflowing table. SCRUM was a learning process that overwhelmed the group. At first SCRUM did not bring much value to the team, we struggled with the process and spent too much time on the different steps just to end up doing it wrong anyways. However, like any other learning experience we fought through it, we asked for guidance and most importantly we listened and corrected our mistakes. And low and behold, step by step, we slowly but surely managed to get SCRUM to a state where it provided value to our process instead of adding problems. All our SCRUM documentation can be found in Appendix §§.

We decided to run sprints of three workdays of 8 hours, this resulted in a total of 24 hours per developer per sprint. This meant that the team had a total of 72 working hours per sprint. We set out to have a standing scrum meeting every day, where each group member would answer the following three questions:

1) What did I do yesterday that help the Development Team meet the Sprint Goal?
2) What will I do today to help the Development Team meet the Sprint Goal?
3) Do I see any impediment that prevents me or the Development Team from meeting the Sprint Goal?

These questions would give the other developers a quick overview of what the others had done, would do and what they were struggling with. Every meeting was logged, and a burndown chart was made after the meeting had concluded. Additionally, our definition of "done" was: **Done means it is reviewed, implemented, tested, integrated, documented and the code documented with JavaDoc.**

We started the SCRUM process on the 9th of May, which is when we planned our first sprint with planning poker. We ended up planning a total of seven tasks which had a total estimated workload of 72 hours.

Table 1 - Planned tasks for sprint 1

TASK ID	NAME	WEIGHT
1	Main architecture	13
2	As a user I want to create an account	9
3	As a user I want to login	4
4	As a user I want to make a post	14.5
5	As a user I want my posted poetry to be visible to other users	12
6	As a user I want the ability to filter all poetry posted on the forum	8.5
7	As a user I want to be able to comment on posts	11
		Total: 72

Every task was divided into smaller tasks which was then assigned an owner whom would be responsible for completing that specific task as can be seen in the table below.

Table 2 - Task 1 with its respective sub tasks and their owners

TASK 1	WEIGHT	OWNER	STATUS

CREATE CLIENT-SERVER CONNECTION	4.5	Sebastien
MAKE SERVER - DATABASE CONNECTION	4.5	Tamas
TEST	2.5	Lau
DOCUMENTATION	1.5	Sebastien, Tamas
TOTAL:	**13**	

However, not only did our first sprint span a total of 4 days (including a full day of planning). We also ending up with a burndown chart looking like this;

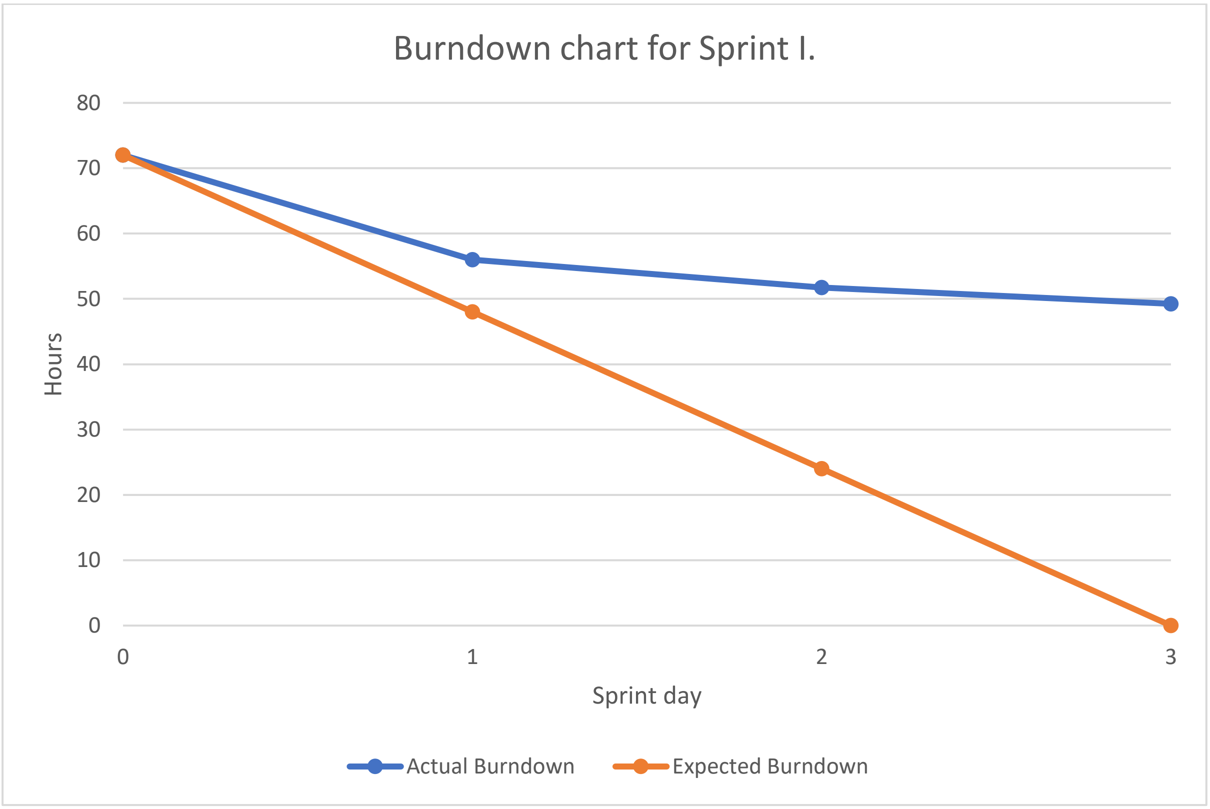


Figure 1 - Burndown chart sprint 1

Not only were the team 50 hours behind, but we had also spent a tremendous amount of time working with scrum without any results to show for it. However, we tied our shoes and kept going. We reviewed the work that had been worked on, even though nothing was considered "done". Moreover, we reflected on the process and how the first sprint went, we sought guidance from Troels to get feedback on our process. We identified four key problems in sprint one, that we were determined to correct in sprint two. These key problems were:

1. Spending too much time on planning.
2. Working on tasks that are not in the backlog.
3. Optimistic time estimates.
4. Assigning owners to tasks

We discussed solutions to the key problems and came up with the following.

1. We originally thought the team had 72 hours to work on backlog tasks, however it was brought to our attention that every part of the SCRUM process takes away time from the 72 hours. We therefore revised the 72 hours and came to a new number: 59.5 hours
2. After Troels' supervision we now knew the difference between a product backlog and a sprint backlog. This allowed us to add tasks that had nothing to do with the product backlog to the sprints.
3. During the sprint planning we had been overly optimistic with the time estimates. Everyone was overly confident that they knew exactly how the product backlog items should be implemented and expected no bumps in the road. However, the road was not as even as we originally thought. So, we agreed that next sprint we would add buffer time to most tasks to make up for eventual challenges.
4. Lastly because we had assigned owners to all tasks during the sprint planning, the scrum process was less agile that it normally is. With supervision we discovered this major flaw in our planning process, and we would make sure that during the next sprint no developers would be assigned to different tasks.

This process of trial and error with scrum continued till the end of the project and sprint 5. The major changes throughout the sprints have been:

After sprint 2 it was brought to our attention that it is not a requirement to JavaDoc our code and we therefore, changed our definition of done to; **Done means it is reviewed, implemented, tested, integrated, documented.** Additionally, we acknowledged that our expected total hours were still too high since we had not accounted for eventual breaks during the 8-hour days. So, we agreed that we hold an average of 60 minutes of breaks per day. Resulting in the new total hours being 50.5. During sprint 3 we attempted to make our version of the SCRUM process even more agile by dividing bigger tasks into smaller tasks, so we could help one another. This resolved the problem that some group members would be stuck on tasks too big to complete during a sprint. Additionally, we reviewed our product backlog with the customer and ended up removing few backlog items. During sprint 4 we realized that we had been doing the sprint review incorrectly. We noticed that our review and our retrospective were awfully similar and repetitive. We therefore, focused more on reviewing the work "done" during the sprint review instead of discussing the sprint, which should be done in the retrospective. The full scrum document can be found in Appendix §

We have miraculously managed to unconsciously apply unified processes in our sprints, even though not much thought was put into it during the sprints. It is clear that our first sprint and second sprint were Elaboration iterations because we spent the majority of the time analysing, designing and implementing. Whereas, the third and fourth sprint were construction iterations because design and analysis started fading out, whereas testing started taking more and more of our time. Lastly our last sprint was deployment, we were almost done implementing, very busy getting everything ready for hand-in. Furthermore, working in iterations has allowed us to change the course of the project throughout the project period.

Personal reflections

Lau Ravn Nielsen

As always, working with problem-based learning is a bliss. Getting to apply the knowledge you learn throughout the semester in actual projects is why I dropped out from Aarhus university and choose VIA instead. It truly does give one a new perspective on the knowledge and skills you learn in the class room when you can get hands on. It was of course a shame that Denis choose to drop out mid semester, the loss was felt in the group work. Especially since he was the one who picked the project idea in the first place. That is something I'll make sure does not happen in future projects, that three group members agree to work on an idea because one person likes it a lot. I think that decision hurt this project greatly. Mainly because Denis was supposed to be the main drive behind this project and motivate the rest of us. Afterall, he was the one who was adamant about working with poetry, no one else in the group really cares about poetry. That especially has put its mark on our system, which is barely poetry related as a finished product. The teamwork has worked great despite the hardships the team has had with both Denis leaving and Scrum giving us a hard time in the beginning. It definitely helped that we had worked together on first semester and thus knew the strengths and weaknesses of every member. However, I feel like I've let the team down a bit, because in some way, we tried to maintain the same work structure as we had on the first semester. Where Denis, Sebastien and Tamas would be coding, and I would be focusing on the reports. However, with Denis gone, whom was the main coder, someone had to try and fill that void, which I failed to do. However, there have been no major conflicts, and everyone has respected one another's opinion even if the discussion could get a little heated from time to time. SCRUM was a thorn in the eye for a few weeks, we were confused… the instructions were unclear however Troels' 150 slide power-point really helped with the understanding of how SCRUM was supposed to be used. Ever since that session we managed to turn the SCRUM ship around and started moving in the right direction. So, in the end SCRUM proved valuable to us as a group, it helped us organize and manage the tasks that needed to be done. And the introduction to iterative development has improved the way I look at software development greatly compared to last semester when we used the waterfall method.

All in all, I think the teamwork was great, like last time and I would write another semester project with the same group if it were not for Sebastien leaving as well. I personally think we made the best of a shitty situation with Denis leaving, our project might be mediocre, but we have pushed ourselves and even if we have made mistakes here and there, we will take those mistakes and the feedback that comes with them and learn from it.

Sebastien Malmberg

How did the project go?

The project went rather well, although we failed to properly manage our time, which resulted in a very basic code implementation, I still believe we all did our best to produce the outcome. However, the poor time management resulted in three heavily loaded work weeks. In fact, the last three weeks have been so busy, that in my opinion, it has been difficult to keep track of what each group member is working with.

How did the teamwork go?

The teamwork has gone overall very well, as we all exceed in communicating what needs to get done.

However, because we lost a team member which often took the leader roll in our group, we fail to see to it, that it in fact, gets done. And as the tasks stack up, time becomes of the essence. This would imply that the team has worked unproductively to achieve the outcome. Nevertheless, it should not be neglected that the idea for this project belonged to the team member who left. Therefore, perhaps the motivation to complete this poetry forum overall decreased as he left.

What would you do different next time?

Next time I will aspire to work on a project which motivates and inspires me to push harder and give more! I would also recommend to my future self to sit down and take my time with designing sequence diagrams and class diagrams. During this project I have often had to go back and make changes to these diagrams as I implement the code, due to sloppy work. But I have now realised that the time you spend making great design, is time you save implementing,

How did the scrum work for you?

The scum work has gone well for a first try! Sometimes we failed to work in iterations, ex. we would write down every single possible user story then make every use case diagram which later resulted in a finished domain model. This was all done in one iteration. We also failed to stick to the 15-minute scrum planning meeting, which would instead result in hours of planning in the beginning.

I have greatly enjoyed the “freedom” of using scrum however as I can always go back and change things.

Tamas Fekete

How did the project go?

It wasn’t that bad, but we need an extra week to finish all functionalities what we couldn’t implement to the final system. We learnt a lot during the project period and could finish with a presentable program so I’m quite happy about it.

What would you do different next time?

Next time I would like to have more tasks from implementation like creating views, server connection. I will pay more attention of the project design in order to reduce the chance of fatigue during code implementation. Furthermore, now that we know the basics of scrum next time we can perform even better due to the new experience gained in this field.

How did the teamwork go?

Since we didn’t have a team leader in our group it was sometimes a little bit difficult. We had to find our motivation to push ourselves forward with the teamwork, especially when the only group member who was enthusiastic about the project left the group behind. Other than that, it was pretty much alright we could finish our parts sooner or later compare to the burning chart.

How did the scrum work for you?

It wasn’t that difficult to understand the usage of scrum, but when it comes to follow it, this is where the problems occur first. It took several weeks to learn how to go through a proper sprint.

Group reflection

We have chosen to re-use our personal descriptions and risk analysis for the group members from SEP1, mainly because we did not have any requirements when it came to the process report this semester. So, we felt like the description from last semester was enough especially because it also had a description of Denis in it. In the description the theory of Hofstede’s 6 cultural dimensions is used to analyse the cultural traits of the different group members, which was used to predict what could happen in our group when we started working together. However, it also included our E-stimate profiles, which gives a more personal view of how each group member might act in a group environment. SEP1 was a nice stepping stone to SEP2. The connection between the two is clear, although it SEP1 was a better in the sense that we have a real-world case to work on, with a real customer. However, SEP2 brought new challenges that SEP1 did not have. For example, the supervisors held our hand and guided us through SEP1, however, SEP2 has been a lot more independent. The number of lectures on SEP related curriculum has gone down drastically. The hours the supervisors have been available during SEP Thursdays has gone down a lot compared to SEP1. However, we feel like that is to be expected since it encourages a more individualistic approach from the teams. SEP1 taught us a lot of things, however, with SEP2 we have taken those crucial skills, and added some toppings. Everyone was very vaguely explained in SEP1, however, with our new knowledge not only from SWE but from SDJ, we have been able to go much more in depth with our analysis, design and implementation. Even testing has gotten more advanced in SEP2, so we have really had a feeling of progression since SEP1 to SEP2. The biggest shift has been going from the waterfall method to SCRUM, even though we did not get SCRUM in the beginning, it still caused less confusion than the waterfall method. Being able to work in iterations and the ability to adjust the analysis and design as you go have improved the software development process a lot. Although SEP has improved a lot in one semester, SEP1 was superior when it came to clarity from the supervisors and what they wanted to see in the appendices. Nonetheless with all of those arguments on the table, we feel like the group project was alright, as mentioned many times before; it sucked that we had to lose Denis the way we did and thus forcing us into a project where none of us had motivation drive or aspiration to grind through.

Supervision

We have used our supervisors for a variety of things; however, we have used them for guidance when in doubt. Aforementioned, proved particularly useful when we needed urgent help with SCRUM. Nonetheless, the supervision has been great, there have been supervisions where we have walked parted ways with the supervisor and felt that we learned more during those 30-40 minutes than we did during class.

Conclusion

The group summary on what to do and not do in group work

- Communicate, learn how to talk to your teammates.
- Distribute the work-load equally.
- When faced with a bad idea, discuss the idea and not the person who said it.
- Attend group work sessions and scheduled meetings.
- Attend SWE and SEP.
- Respect each other and each other’s opinions.
- Treat each other as equals although group roles are present.
- Do not be late for scheduled meetings.
- Do not insult each other.
- Do not be afraid of confrontation or conflict
- Do not forget to work independently.

Appendices