

Introduction

This is a report for a project called QuickDraw. It is a game where you compete with friends about who is the best at fast accurate drawing.

The project can be found at the following repository:

<https://github.com/samelth/DAT076-Group4/>

Use-cases

Use case #	Name	Actor	Goal	Description
1	Join lobby	User	Join an existing lobby.	As a user I want the ability to join an existing lobby.
2	Create lobby	User	Create a new lobby.	As a user I want the ability to create a new lobby.
3	Start game	Lobby host	Start a new game session and move all participants to the game page.	As a host I want the ability to start a game with the lobby participants.
4	Extract drawing	Guess controller	Fetch a drawn image from the database	As the guess controller I want to provide the relevant picture to the guesser.
5	Send drawing	Game participant	Send the drawn picture to the database.	As a game participant I want the ability to submit my drawn picture to the database so it can be fetched when needed.
6	Guess drawing	Guesser	Guess what word has been drawn.	As a game session participant with "guesser" status I want to be able to guess what other participants drew.
7	Next round	Lobby Host	Start the next round of a game session.	As a host I want the ability to start new rounds in a game session.
8	Back to lobby	Lobby Host	Move all participants back to the lobby.	As a host I want the ability to move participants back

				to the lobby after a game session is finished.
9	Send Chat Message	Lobby Participant	Send a message that can be seen by all participants in the lobby.	As a user I would like to chat with other participants in my lobby.
10	Set difficulty	Lobby Host	Set the difficulty of the game before starting it.	As host of a lobby I want to be able to set the difficulty of the game to challenge the participants.

User manual

As a user, the first thing you meet is the start page. From here you can either host a new lobby or join an existing one. You can host a lobby by entering a username and pressing the button that says “Host Lobby”. To join a lobby, press the “Join Lobby” button and from there, enter a username and a lobby code given to you by a lobby host.

If you want to host and play a game by yourself, you’ll need at least two player sessions. The easiest way to achieve this is to open up an incognito tab in your browser in addition to the tab in your regular browser.

When you are in a lobby you can chat with the other users and read the rules. If you are the host you also have the option to set the difficulty and start a new game session with the players in the lobby. When a game session starts you get redirected to the game page where you are either a guesser or a drawer (the artistic kind, not the storage kind). The drawers draw the word prompted on the screen and then click submit and the guesser gets to guess what word the drawers drew in the order that they submitted the pictures. This incentivises fast drawing since the only one getting points is the guesser and the player that made the guesser guess correctly. However, taking a bit more time to draw increases the chance of the guesser guessing correctly so it is a balancing game.

Design

describe the technical construction of your app, including all libraries, frameworks and other technologies used

As for libraries, Java ServerFaces and PrimeFaces were selected to design the front-end for the application. They contain a myriad of features used in the project

and enable easy integration with the server side. The html2canvas library was used in order to capture screenshots in the form of dataURL links.

Selected as a framework, Bootstrap was used for the grid layout of the application and various elements inside. This provides a very clean flex design, as well as an optional borderless design choice.

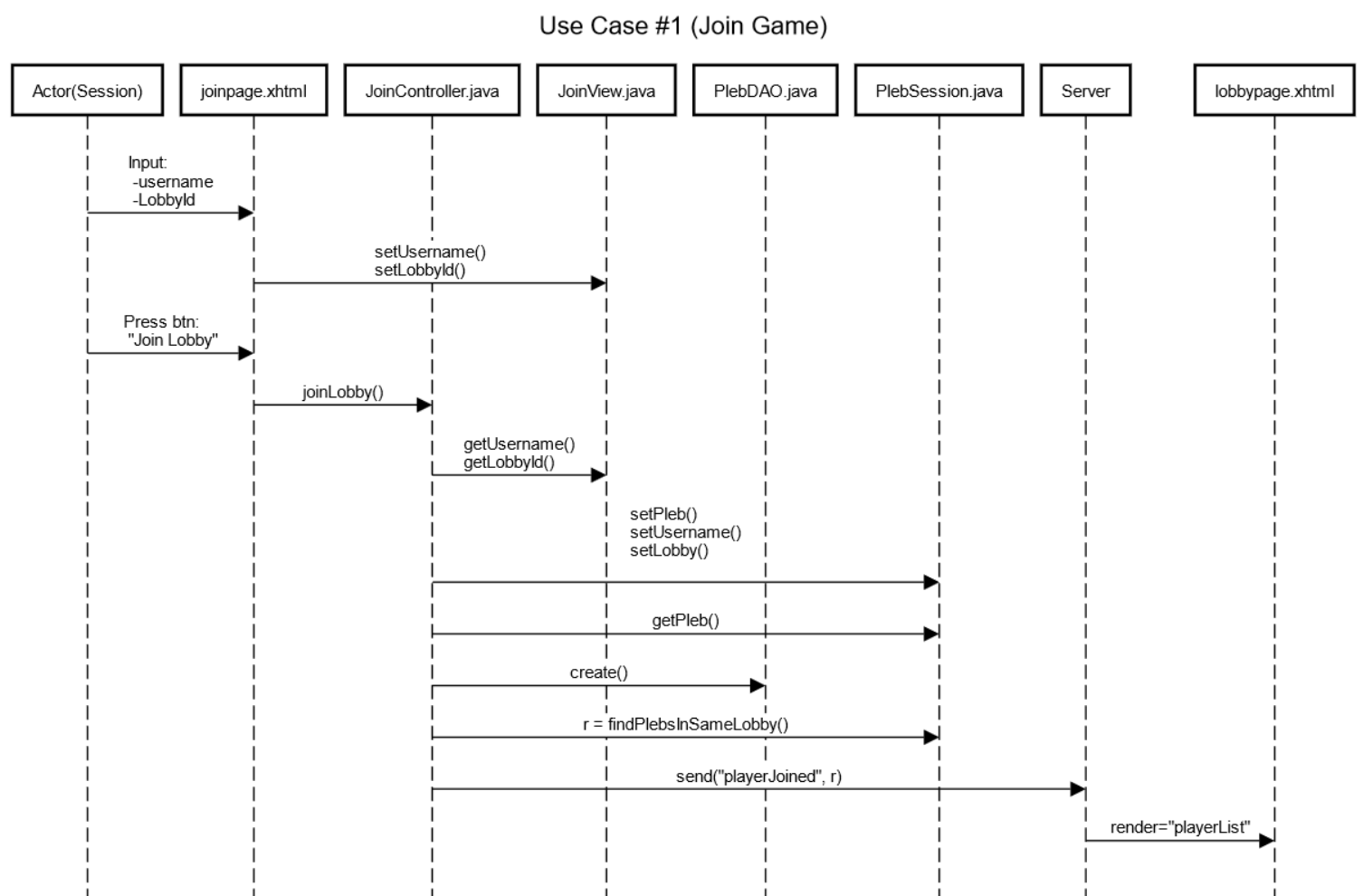
JUnit was used to perform unit tests. The tests are all independent of each other and therefore scalable. We used Arquillian as a unit testing framework. The tests that are executed are separated from the database.

Omnifaces websocket were used to notify users of certain events.

JavaScript was used for functionality such as canvas, countdown timers and button functions.

Application flow

Here are a few sequential diagrams for selected use cases.

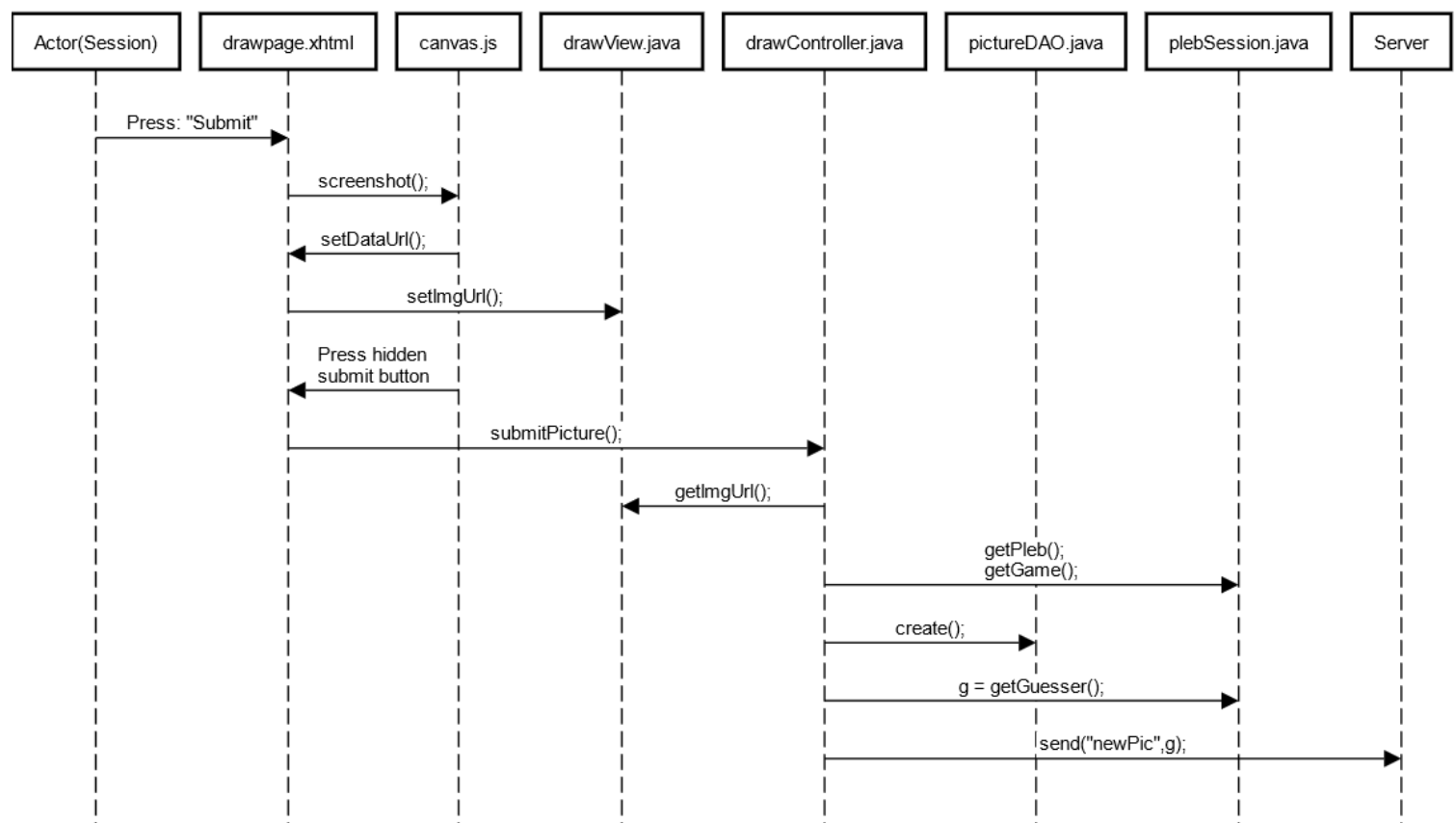


This illustrates the use case of joining an existing lobby.

The user fills in a form with username and lobby ID followed by a press of the “Join Lobby”-button. The button validates the input by checking that the username is valid and that the lobby ID corresponds to an existing lobby in the database. If validation succeeds it triggers a function call to the controller class for the join game view. The function does a number of things:

- It fetches the input from the view class.
- Creates a new player and stores it in the database.
- Updates the PlebSession class which acts as a connection between a session and a specific player in the database.
- Lastly it fetches all the players in the lobby and sends a message to them which triggers an ajax event that updates the list of players in the lobby.

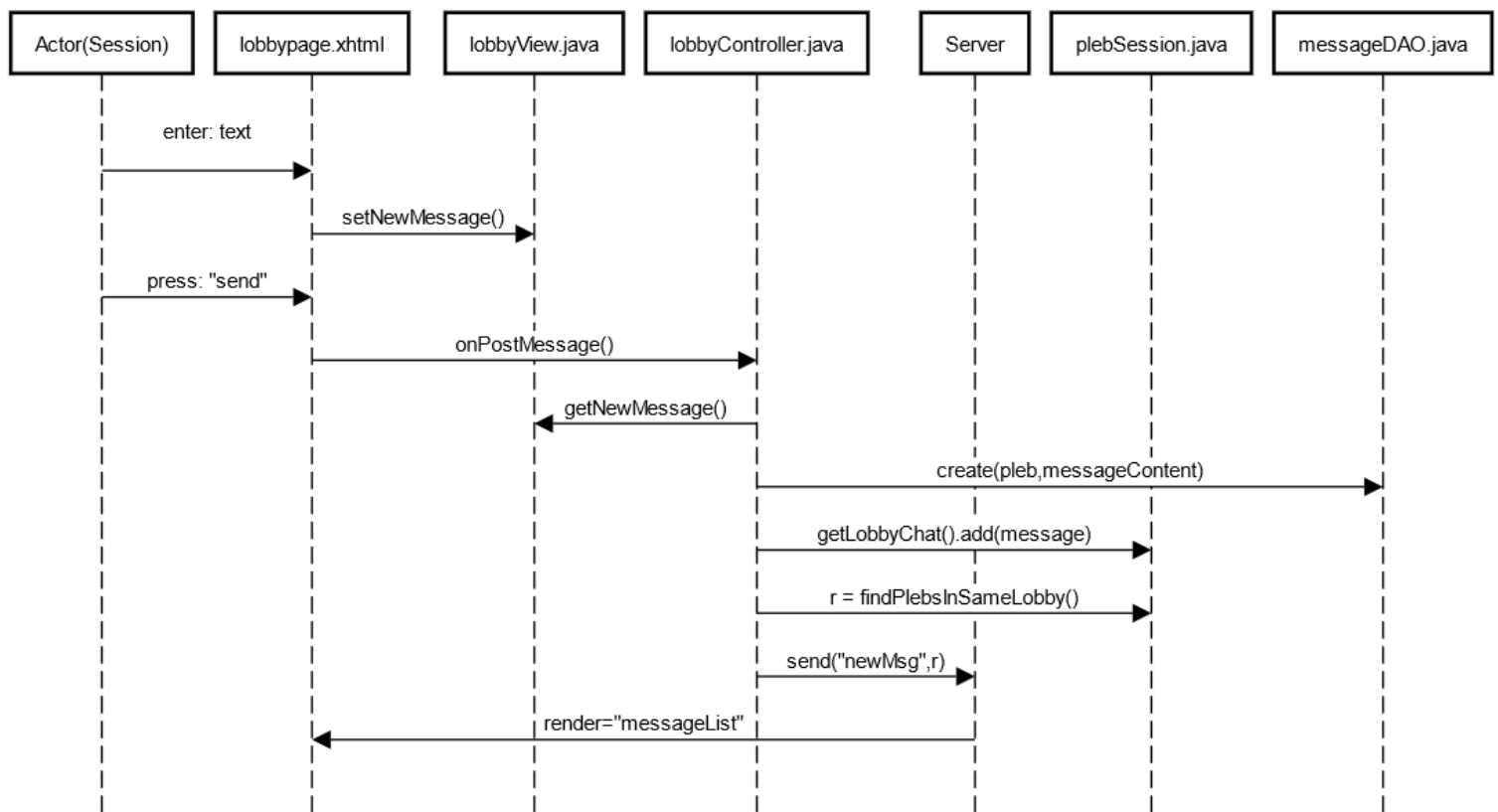
Use Case #5 (Send drawing)



This illustrates the use case of sending a drawing.

The user draws their picture and presses submit, this triggers a javascript function that fetches the canvas object and stores it as a string variable. This variable is then used to set the image variable in the drawView. After this it triggers a function call to drawController which fetches the picture string from the view and uses that to create a picture object in the database. The last step is to notify the user with guesser status that there is a new picture available via a websocket channel.

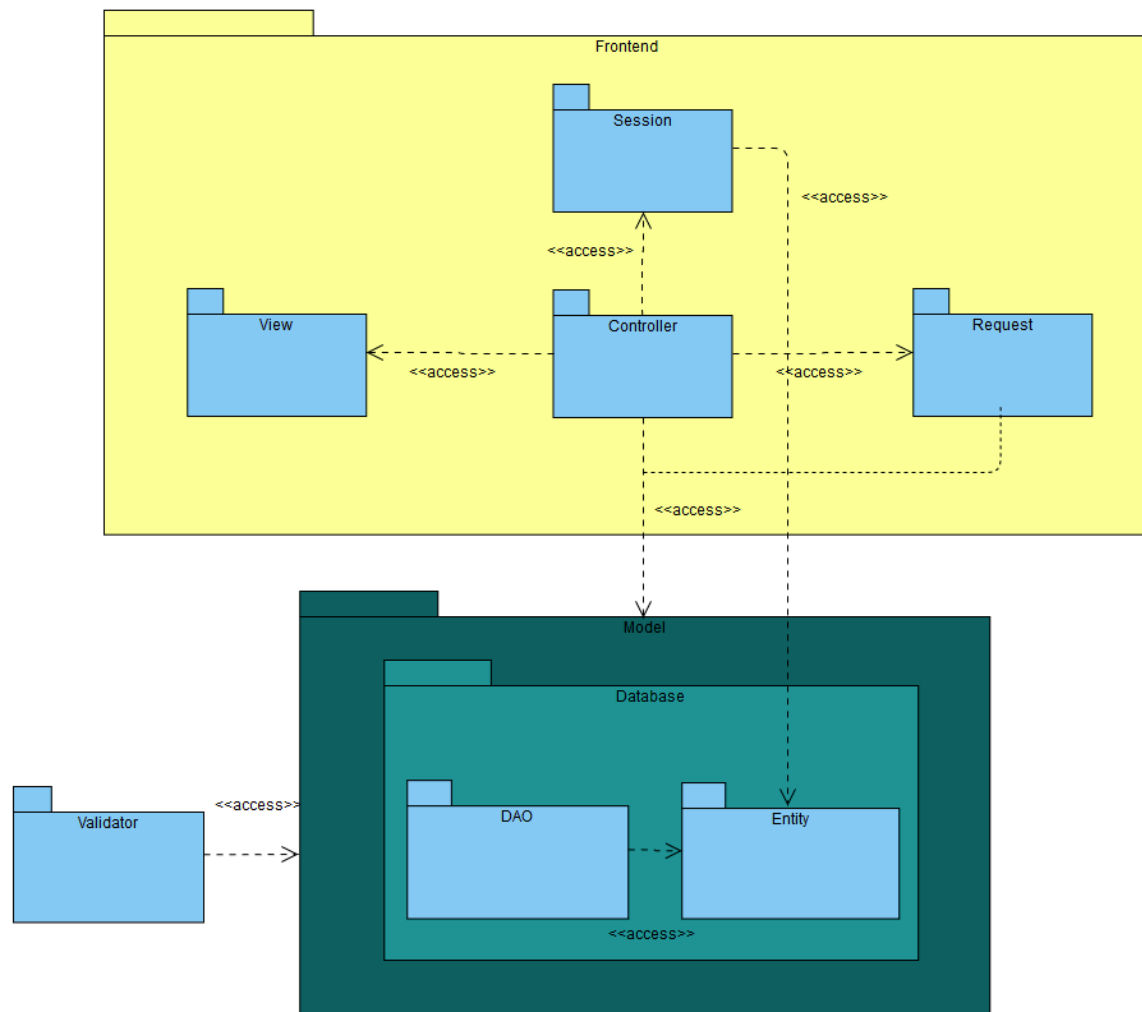
Use Case #9 (Send Chat Message)



This illustrates the use case of sending a message in the lobby chat.

The user inputs their desired message in the lobby chat text box. This input sets a message string in the lobby view class. When the user presses "Send" a function is called which fetches the message string from the view and the user from the pleb session. It then uses these data to compose a message and stores it in the database. When this is done it fetches a list of the users in the lobby and notifies them viwa websocket that their message list needs to update since there is a new message.

Package diagram



Here is the package diagram over the domain.

The Controller and View packages are self-explanatory (MVC).

The request package consists of classes that handle requests from the end-user.














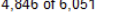
The session package consists of classes that the end-user uses during a whole session.

The validator package contains our validation.

In the model you will find the entity classes and DAO classes, which represent our model. The DAO package responsibility is communication to the database and the entity package representing the objects in the database.




























Code coverage report

All the code in the DAOs has been tested and in consequence of that some code has been tested in the entity classes. In the picture below you will see our test coverage for our respective packages.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
frontend.controller		0%		0%	358	358	200	200	147	147	6	6
frontend.view		0%		0%	111	111	17	17	52	52	6	6
model.database.entity		53%		58%	109	192	70	193	76	147	0	14
frontend.session		0%		0%	44	44	31	31	24	24	1	1
frontend.request		0%		0%	12	12	35	35	10	10	4	4
validator		0%		0%	10	10	25	25	4	4	1	1
com.mycompany.quickdraw		0%		n/a	1	1	1	1	1	1	1	1
model.database.dao		100%		n/a	0	36	0	103	0	36	0	8
Total	4,846 of 6,051	19%	633 of 686	7%	645	764	379	605	314	421	19	41

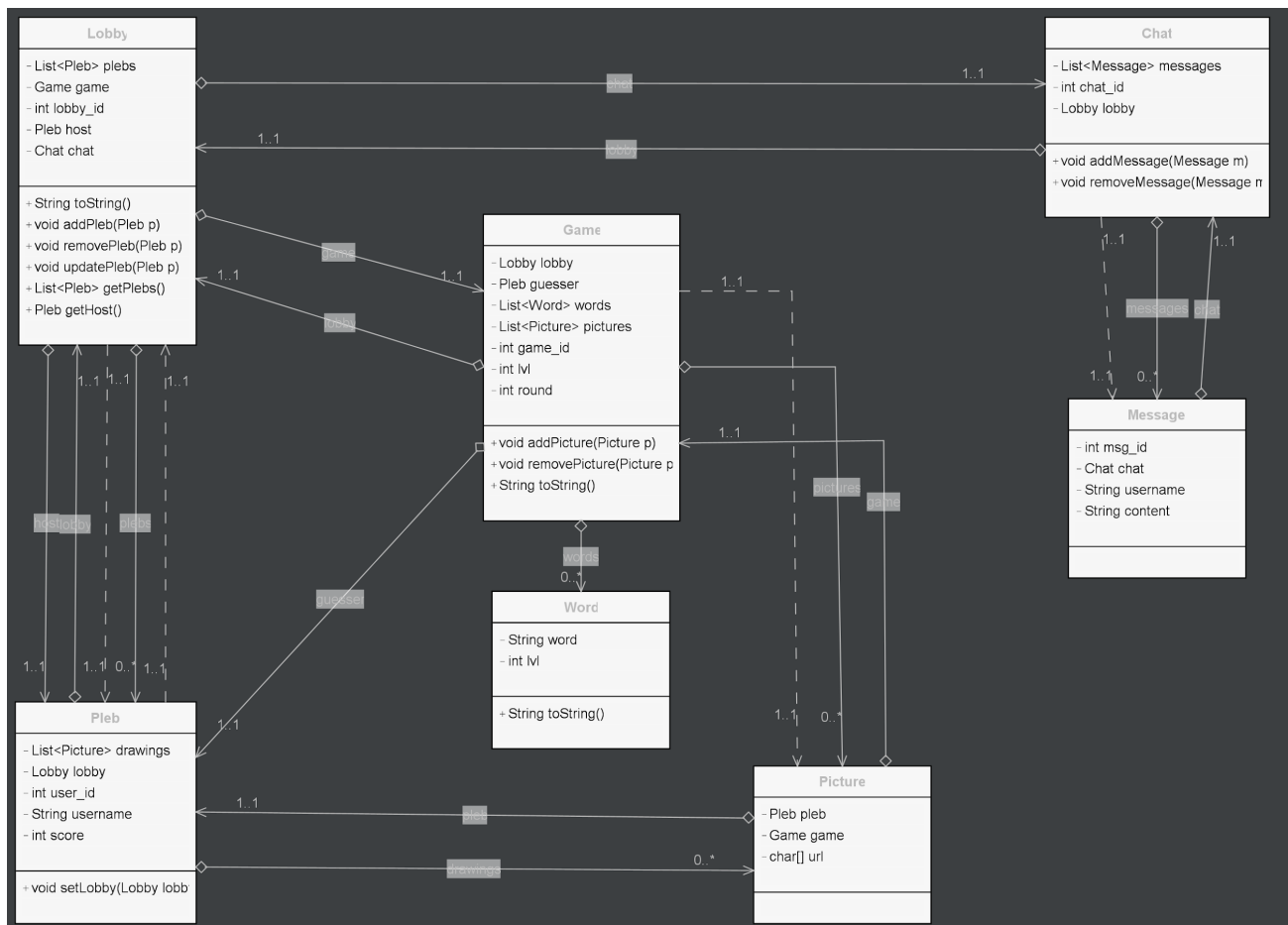
And here you see the coverage of our tests of our entities.

model.database.entity

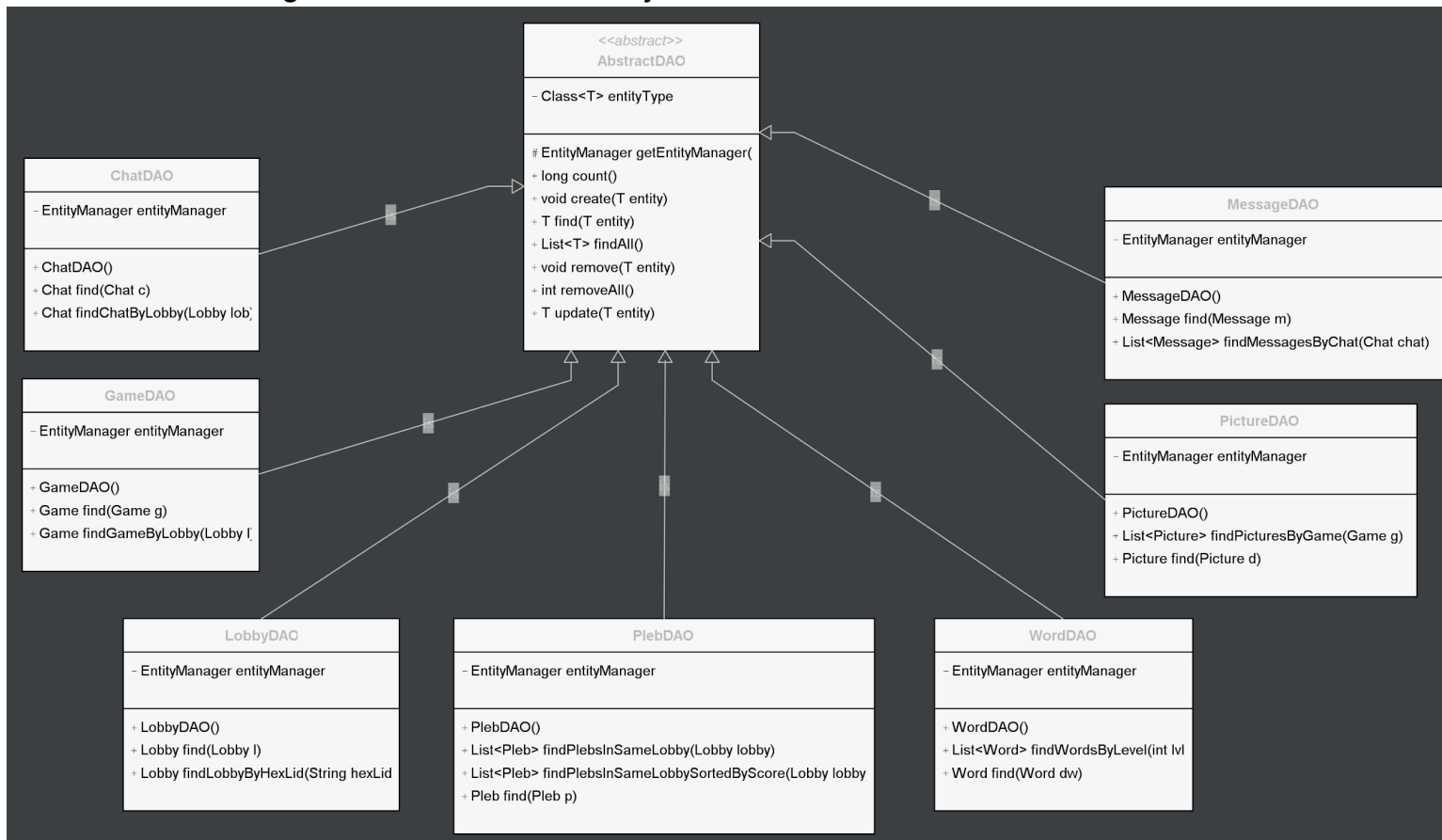
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Game		32%		50%	19	26	11	18	15	22	0	1
Pleb		35%		75%	11	20	4	12	9	16	0	1
Message		29%		62%	12	18	4	8	9	14	0	1
Chat		35%		50%	12	18	6	13	8	14	0	1
Picture		38%		42%	12	19	3	7	6	12	0	1
Lobby		57%		50%	13	24	7	20	8	19	0	1
Word		52%		42%	11	17	3	7	5	10	0	1
QWord		40%		n/a	2	4	4	9	2	4	0	1
QPicture		69%		50%	5	8	6	15	3	6	0	1
QMessage		67%		50%	4	7	6	16	3	6	0	1
QGame		84%		100%	2	8	4	19	2	6	0	1
QLobby		83%		100%	2	9	4	17	2	6	0	1
QPleb		79%		100%	2	7	4	17	2	6	0	1
QChat		76%		100%	2	7	4	15	2	6	0	1
Total	679 of 1,475	53%	37 of 90	58%	109	192	70	193	76	147	0	14

Model diagram

Class diagram for how entities are related to each other.



Class diagram for Data Access Objects



Responsibilities

The driver-passenger model of working was used under parts of the labs, and as such the git repository doesn't always reflect the commits by the group as a whole. Lab #3 was one such occurrence where work areas overlapped and coding was done together to avoid duplicate submissions to the repository.

During lab #2, everyone was responsible for their own .xhtml page for the mockup phase. This led to design looking different for almost every page, as no one had really made use of CSS or HTML before this project. This had to be amended at a later stage, partly to clean up, but also to get an overall theme and page-wide alignments going.

This is a **non-exhaustive** list of what members have been working on in the project:

William

- Front- and back-end validation

- Ajax events.
- Null-checks and input error handling
- Controllers
- Refactoring
- Test writing
- JavaScript functions
- Sockets and channels
- Database and cascade implementations

Karl

- Ajax
- Entities
- Methods
- Application logic
- Queries
- Point system for application
- Database work
- Gitignore
- Word implementation on drawpage

Samuel

- Ajax events
- Websockets
- Use cases
- Game logic
- Lobby chat w entities & queries
- Layout & design
- HTML
- JSF integration

Victor

- Word list (finding; parsing; translating into SQL statements)
- Structural refactoring
- Nomenclatural refactoring
- Game logic
- DAOs & DAO tests
- Supporting others in implementing various bits and bobs (e.g. displaying submitted pictures to the guesser, making sure all users get the same randomised list of words)

Fawzi

- CSS
 - Refactoring
 - Color themes
 - General borderless design implementation
- JavaScript
 - Canvas.js
 - Animation bar (aided by William)
 - Countdown timer and logic (aided by William)
- HTML
 - Refactoring
 - Making layout alike for all pages
- Drawing area (canvas)
 - Resize functionality
 - Draw/erase functionality
 - Implementation of canvas (aided by Victor)
- PrimeFaces
 - Implemented version 10 for aesthetics

Everyone

- Entity modelling and reviewing
- Testing of the application
 - Finding bugs
 - Reconsider how to implement features
 - Optimizing flow of application