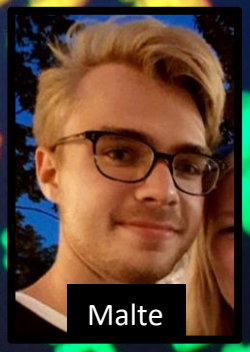
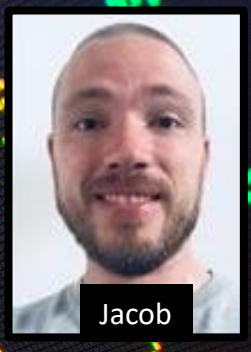
DTU Compute


Claes


Simon


Andreas


Malte


Jacob


Kristian

Pong
CDIO Final

https://github.com/AndreasBGJensen/OnlinePong

Kursus: 02324 - F19

S183018 Claes J. S. Lindhardt
S130016 Andreas BG Jensen
S185122 Jacob Riis Jensen
S185139 Malte Pedersen
S180945 Simon Woldbye-Lyng
S185125 Kristian Andersen

Bachelor of Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Projekt Titel:
CDIO opgave

Projekt periode:
15/02-2019-02/03-2019

Temaer:
Computer Science,
Software Ingenør Kunskaber

Vejledere:
Stig Høgh

Relavante kurser:
02324

Billag:
A-C

GitHub
link: https://github.com/AndreasBGJensen/OnlinePong

Abstract:
This project is to demonstrate our ability to make system architecture diagrams as well as operation diagrams of an IT-System. It is also to demonstrate pratical and theoritical understanding of REST principles. This is all executed by building a webside where more than one player can play Online Pong, a multiplayer Pong game

Readers guide:

This report is made in LaTeX, and uses hyperlinks. These however only work in pdf format. It is also written in Danish.

The relevant parts of sources will be given in footnotes as this example [1]

the first thing in the footnote is
a number in [ ], which refers to a number in the bibliography. Here you can find the source described in debt. Then it will be described what part of the source is relevant. Typically it is a page number, but should the source be video it might be a time frame.

DTU diplom
Lautrupvang 15,
2750 Ballerup
http://www.diplom.dtu.dk

[1] Source [1]: this is a footnote to go see the first source

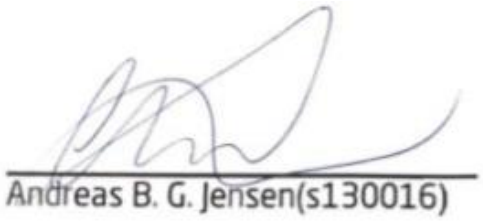
Brugervejledning

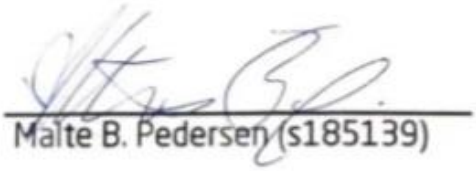
Denne applikation kører i browserne: Chrome, Firefix og Microsoft Edge.

1. Opstart af applikation
   1.1. Server applikationen skal køres på den laptop/computer der er tilslutte routerens netværk. Denne computer skal have installeret JavaEE JDK 8.
   1.2. I din IDEA laves der et nyt projekt fra en eksisterende kilde. Eventuelt kan projektet importeres fra GitHub fra URL'en https://github.com/AndreasBGJensen/OnlinePong.git
   1.3. Når projektet er importeret kan Tomcat serveren startes ved at main metoden i Main klassen.
   1.4. I Main klassen kan der vælges om databasen skal være implamenteret som en ArrayListe eller i MySQL. Dette ændres ved at ændre variablen USE_ARRAY_DB til true.
   1.5. Der findes to måder, at afprøve spillet på:
   1.6. Vælg Run main, og tomcatserveren er deployed. NB: Der kan kun køres en enkelt main af gangen.
      - Hvis at programmet er opstillet som beskrevet ovenfor, så kan der på de to computere der er tilsluttet andet netværk til går routerens IP og port 8080. Eks. 62.79.16.17:8080. - og applikationen er igang.
      - Vil man teste på en lokal server kan der anvendes en laptop/computer. Når Tomcat serveren er startet åbnes der to seperate browsere, hvor der indtastet "localhost:8080"i browserens søgefelt.
2. Start et spil
   2.1. Du har nu startet og tilgået Server Applikationen.
   2.2. Opret dig i databasen ved at klikke på linket "Registrer".
   2.3. Hvis registreringen er succesfuld logges der ind direkte. Hvis man har en bruger kan man logge ind uden at registrere sig.
   2.4. På begge laptops/computere logges der ind på de to brugere.
   2.5. Hvis i begge databaser findes der følgende users som kan anvendes uden at man behøver at registrere sig.
      - Brugernavn: Karsten, password: 1234
      - Brugernavn: Kenneth, password: 1234
      - Brugernavn: Torben, password: 1234
      - Brugernavn: Bent, password: 1234
   2.6. Der dukker en knap op der hedder "Start spil". Klik på denne.
   2.7. Game Serveren vil nu pare de to klienter og spillet vil gå i gang.

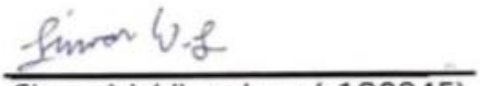
NB: Eftersom at der er implementeres elorating vil der være en risiko for at det vil tage til at finde en modstander, hvis at deres highscore/elo rating ligger fra hinanden. Derfor kan det godt tage et par minutter at finde et spil. Eventuelt log in som to spillere, hvis highscore ligger tæt på hinanden.
Såfremt at to spilleres highscore er adskilt af mere en 500 point vil der ikke kunne dannes et spil imellem disse to brugere.

Disclaimer: The content of this report is publicly available, but must not be published without permission from the authors.

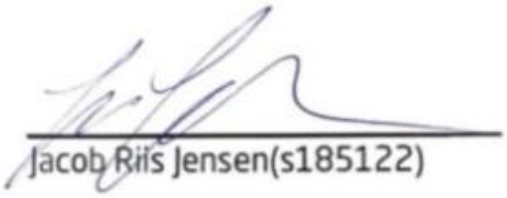

Andreas B. G. Jensen(s130016)


Malte B. Pedersen (s185139)


Claes J. S. Lindhardt(s183018)

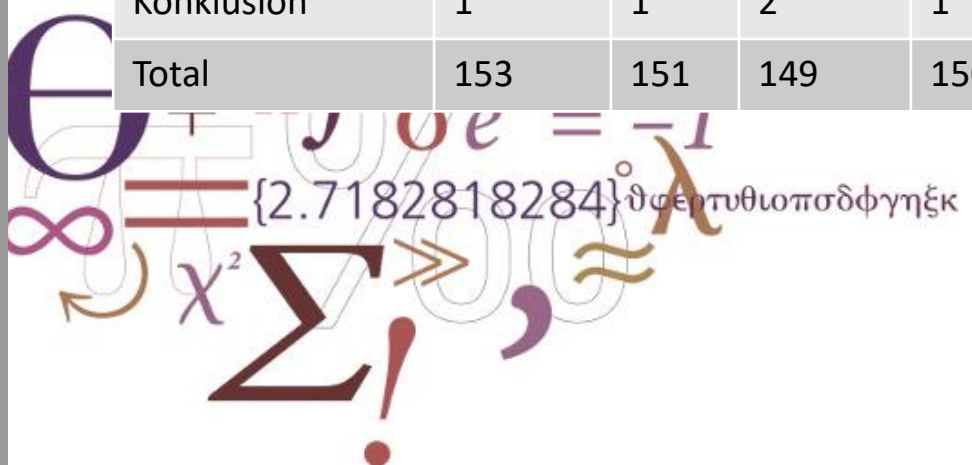

Simon Woldbye-Lyng(s180945)


Kristian Andersen(s185125)


Jacob Riis Jensen(s185122)

Time Regnskab:

	Andreas	Claes	Kristian	Simon	Jacob	Malte
Formål	5	2	4	5	8	2
Analyse	23	17	30	0	26	19
Design	30	45	40	28	42	42
Implementering	73	68	73	90	73	70
Test	21	18	0	26	0	19
Konklusion	1	1	2	1	1	1
Total	153	151	149	150	150	153



	Malte	Claes	Andreas	Kristian	Simon	Jacob	S - skrev på det U - samlede info til det K - korrekturlæste det A - var ansvarlig for det.
Krav	U / K	A / S	A / K / S	K		K / U	
Usecase	K		A / U / S	K		K / U	
IT-systemet som helhed	A / S / K / U	A / S / U	A / S / K	K	K	K	
IT-systemets elementer	K / S / U	S/ K/ U	K	K	S / U	K / S / U	
Pong-Spillet	K / S	S / K	K	A / S / K	K	A / S / K / U	
API og Rest impl.	K	A / U / S	K / S	K	A / U / S	K	
Database	K	A / U / S	K	S	A / U / S	K	Dette diagram viser ansvar osv. for både kodning og rapport skrivning
GameServer	A / S / K / U	K	A / K /S /U	K	K	K	
Webside	K / S	K	K / S	A / S / K	K	A / S / K / U	
Test	S / K / A		A / U / S		S / U / K		

Indhold

Del I
Introduktion

Det primære formål med denne opgave er at udvikle det klassiske Pong-spil i en *online multi-player* version. Dermed er det målet at to spillere skal kunne spille mod hinanden på hver deres computer, i hver deres browser.
Herudover ønskes, der at udvikle et *brugersystem* der muliggør at en bruger kan have en personlig "konto/log in. For at give et incitament for at spille spillet ønskes der at lave en form for ranking-system, hvorved en spiller kan få point når der vindes et spil samt tabe point når et spil bliver tabt.

Eftersom at CDIO Final er en undervisningsrelateret opgave har vi til hensigt at inddrage teknologier fra forlæsninger i kurset 02324 "Videregående Programmering".
Dermed ønsker vi at systemet som en helhed skal gøre brug af følgende sprog / teknologier:

- Java
- HTML
- CSS
- JavaScript
- REST API

Dette lægger op til at spillet implementeres på en website, og datahåndtering, herunder det online-multiplayer-funktionalitet, implementeres i en Java backend.

Følgende rapport er en udredelse af, hvordan systemet er designet samt, hvordan teknologierne er implementeret i store træk.

ProjektPlan
For projektet blev der oprettet milestones med realistiske mål der skulle nås.

Milestone 1 (11/06):

- Lav spillogik
- Website
- User Matching
- Synkroniser spil online
- Opsætning af database / API

Milestone 2 (18/06):

- Forbindelser imellem systemets elementer
- Ratingsystem
- Brugerhåndtering (login, oprettelse)
- Web site

- Tests af GameServer og REST API

Milestone 3 (21/06):

- Findpuds program / optimeringer af kode
- Systemtests
- Lav rapport

Del II
Analyse

1 Use case

Systemet har 3 use cases:

- En bruger kan oprette en profil
- En bruger kan slette sin profil
- En bruger logger ind og spille imod en anden spiller

Hver use case er beskrevet 'fully-dressed' herunder:

En bruger opretter sig i en databasen.	
Primær aktør	Brugere
Interessenter	Brugere: Vil gerne fordrive tiden ved at spille.
Preconditions	Spillet tilgår websitet, hvorefter at filer downloades. Det forventes at brugere logger ind i web siden.
Postconditions	En bruger er blevet oprettet i databasen.
	1. En spille går ind på websitet 2. En spiller bliver præsenteret for en login side, hvor der er mulighed for at klikke på en knap, der leder brugeren hen til en web side, designet for bruger registrering. 3. brugeren udfylder formularen med "Username", "Password"og "Confirm Password"og klikker submit. 4. If: Brugerens "Username"findes i databasen printes der en fejlmeddelelse og bliver ikke oprettet i databasen. Else: En bruger bliver oprettet i databasen, og han bliver logget ind. 5. Spillern er nu registreret og kan spille spillet.

Tabel 1: Use Case: En bruger opretter sig i en databasen.

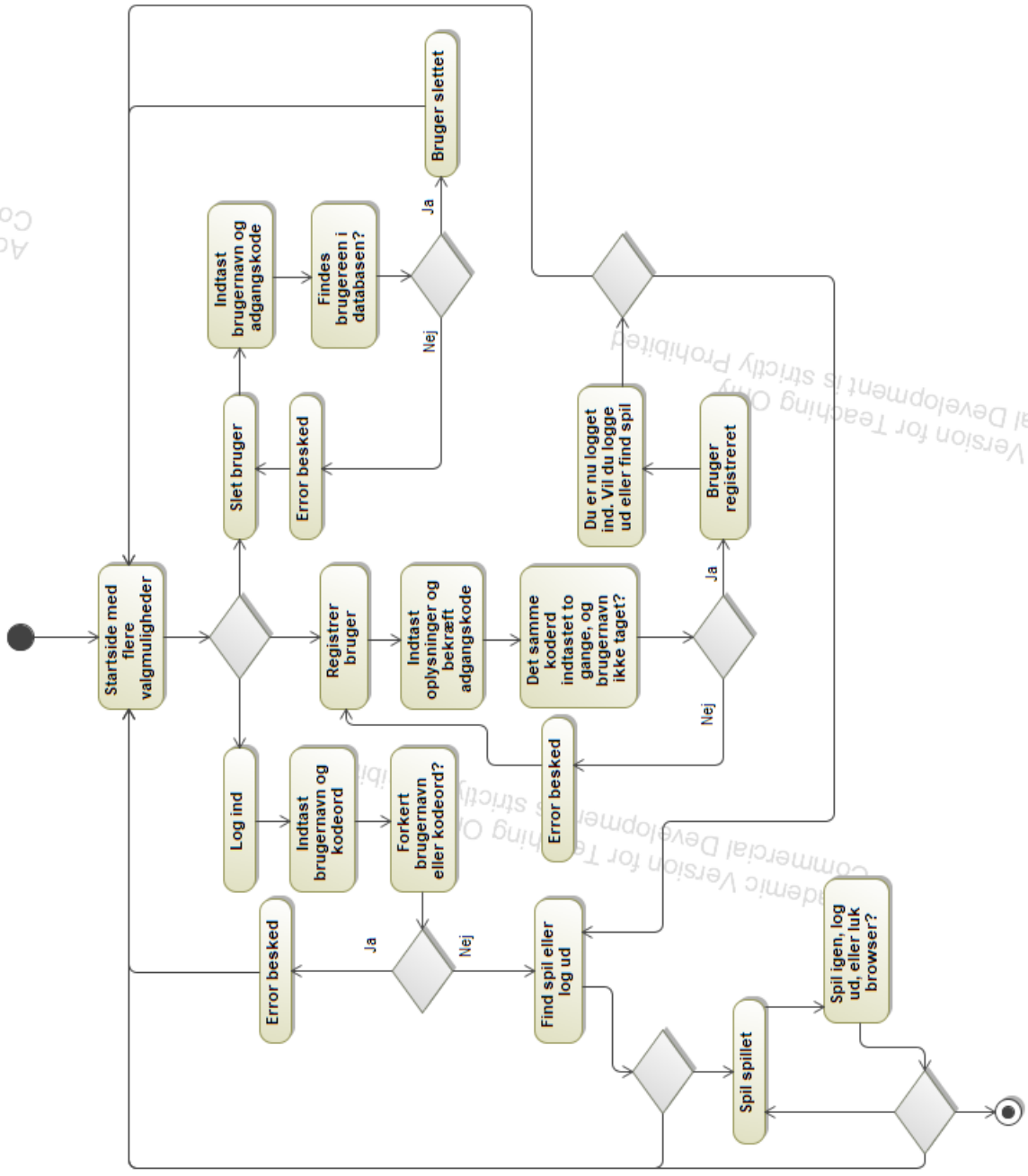
En bruger sletter sig fra databasen.	
Primær aktør	Brugere
Interessenter	Bruger
Preconditions	Brugeren tilgår web siden. Det forventes at brugeren har en profil på web siden.
Postconditions	En bruger er blevet slettet fra databasen.
	1. En bruger går ind på web siden 2. En bruger bliver præsenteret for en login side, hvor der er mulighed for at klikke på en knap, "Delete", der leder spilleren hen til en web side, designet til at slette en bruger. 3. Brugeren udfylde formularen med "Username", "Password"klikker submit. 4. If: Brugerens "username"ikke findes i databasen printes der en fejlmeddelelse. Else: En bruger bliver slettet fra databasen og får en succes meddelse på webpagen.

Tabel 2: Use Case: En bruger sletter sig i databasen.

En bruger spiller imod en anden bruger.	
Primær aktør	Brugere
Interessenter	Bruger: Vil gerne fordrive tiden ved at spille.
Preconditions	Spillet tilgår web siden. Det forventes at brugere logger ind på web siden.
Postconditions	En bruger har vundet spillet og brugerens elo-rating bliver justeret. Begge brugere har et login.
	1. En bruger går ind på web siden. 2. En bruger bliver præsenteret for en login side. Brugeren indtaster sit login og der gives adgang til brugeren. 3. Til højre i skærmbilledet findes der en top ti liste af de bedste spillere der findes i databasen. Til venstre i billedet findes der en liste over spillets regler. 4. Brugeren der er logget ind klikker på knappen "Find spil". Programmet printer informationer til brugeren om at der søges efter en passende modstander. 5. Når der er fundet en passende modstander går spillet i gang hos begge brugere. Brugeren får informationer om navnet på den modstander som der spilles imod. 6. Begge brugere spiller spillet indtil at en af de to brugere har fået det antal point der kræves for at spillet er vundet. 7. Når en af de to brugere har vundet bliver der printet til dem om brugeren har vundet eller tabt, samt hvad der ligges til eller trækkes fra begge brugeres elo-rating. Brugeren får mulighed for at starte et nyt spil uden at skulle logge ind igen. If: Brugeren klikker på knappen "Find new game"bliver den ovenforstående process gentaget fra punkt 5. Else: Brugeren kan logge ud eller lukke browseren, hvis den ikke ønsker at spille igen.

Tabel 3: Use Case: En bruger logger ind og spiller mod en anden bruger

Aktivitets-Diagram nedenfor opsumerer den ovenstående fully dressed.



Figur 1: Brugerens interaktion med systemet

2 Krav

Her kan der findes krav, hvoraf nogen kan markeres med nice to have. Disse er også markeret med status for det færdige projekt:
Funktionelle Krav

1. Database (MySQL) (Skal tale med REST API)
    1.1. Brugeroplysninger **Opfyldt**
    1.2. Match historik [nice to have] **Ikke opfyldt**
    1.3. Score historik [nice to have] **Ikke opfyldt**
    1.4. Opbevare password i hashet tilstand [nice to have] **Opfyldt**
2. Rest API (Java) (Skal tale med Database og Game server)
    2.1. Verificere en bruger **Opfyldt**
    2.2. Poste en bruger **Opfyldt**
    2.3. Poste score **Opfyldt**
    2.4. Hente spillerinformation **Opfyldt**
3. WebServer (Website) (HTML, JS, CSS) (skal tale med REST API og Game server)
    3.1. Login system **Opfyldt**
    3.2. Finde spil **Opfyldt**
    3.3. Se statistik over spiller [nice to have] **Ikke opfyldt**
    3.4. Top ti leaderboard **Opfyldt**
4. Pong Spillet(skal køres af hjemmesiden)
    4.1. Kommunikation til Game serveren **Opfyldt**
    4.2. Håndtere to spillere eller mere **Opfyldt**
    4.3. Boldens hastighed skal være uafhængig af dens retning **Opfyldt**
    4.4. Boldens retning skal påvirkes af batslag **Opfyldt**
    4.5. Boldens retning skal påvirkes af væggene og batslag **Opfyldt**
    4.6. Bolden skal kunne bevæge sig i alle retninger på et 2-dimensionelt plan **Delvist opfyldt**
    4.7. Spil score (point registrering) **Opfyldt**
    4.8. Vælge, hvilke taster spillere skal spille med (taster skal være valgfrie) [nice to have] **Ikke opfyldt**
    4.9. Exit funktion [nice to have] **Ikke opfyldt**
5. Game server
    5.1. En klient skal kunne forbinde sig til game serveren **Opfyldt**
    5.2. Lave en tråd når der er fundet to klienter **Opfyldt**
    5.3. Sikre at tråden kan kommunikere med klienterne **Opfyldt**
    5.4. Holde styr på igangværende spil **Opfyldt**

5.5. Sende spildata til anden part **Opfyldt**

5.6. Vise, hvor mange spil der er i gang. [nice to have] **ikke opfyldt**

Ikke-Funktionelle Krav

1. Systemet skal opbevare følgende informationer omkring hver bruger: selvvalgt brugernavn, password. **Opfyldt**
2. Brugernavne skal være unikke i systemet **Opfyldt**
3. Systemet må ikke gå ned ved forkerte brugerinput, og skal give brugeren en fejlmeddelelse hvis sådanne inputs forekommer. **Opfyldt**
4. Brugeren skal informeres, hvis der sker fejl i systemet, via beskrivende fejlmeddelelser **Delvist opfyldt**

Del III
Design

I dette afsnit bliver systemets overordnede- og dets indre elementers arkitektur og design beskrevet.

3 IT-systemet som helhed

3.1 Spil-kommunikation

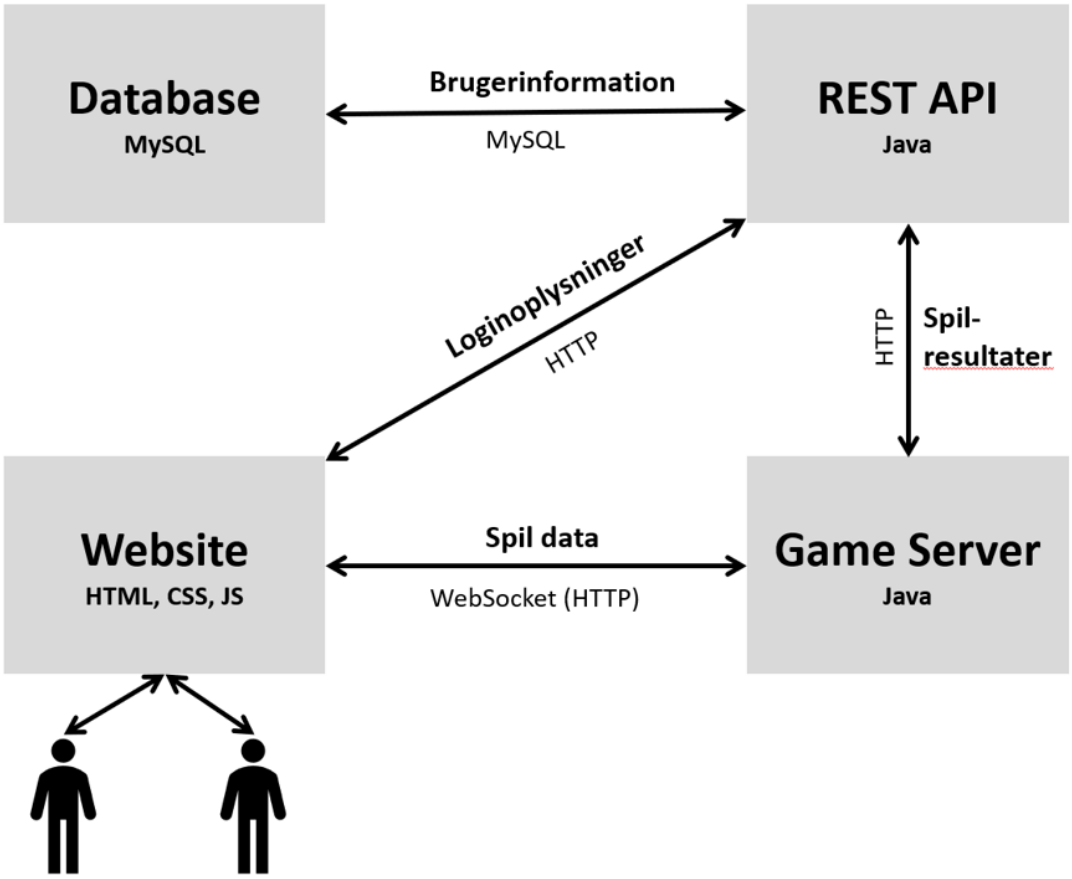
Vi har valgt en 'client-server' arkitektur til at facilitere online-multiplayer funktionen. To brugere finder hinanden ved at forbinde til den centrale game server. Spillet "synkroniseres" med modstanderen, ved at sende spildata (f.eks. boldposition) gennem game serveren. Brugerne kommunikerer derved ikke direkte.

3.2 Overordnet Arkitektur

Systemet bygges som **4 separate** software-applikationer, der praktisk talt kan køre på 4 forskellige servere 4 forskellige steder i verden:

1. **Website:** Hjemmesiden som brugerne interagerer med.
    - Køre selve spillet, og dets logik/fysik.
    - Oprettelse af brugerprofil.
2. **Game Server:** Faciliterer kommunikationen mellem spillere.
    - Finde modstandere (matchmaker) til spillere.
    - Videresende spildata mellem spillerne under et spil.
    - Uploade spilresultater til databasen via API'en.
3. **REST API:** Giver offentlig adgang til systemets database.
    - Tilbyde en række funktioner som henter og manipulerer data i databasen.
    - Benytte ikke under et spil, men før og efter.
4. **Database:** Indeholder data der identificerer brugere og deres oplysninger.

Kommunikationen er visualiseret i figur 2.



Figur 2: Oversigt over applikationerne og deres kommunikation ml. hinanden.

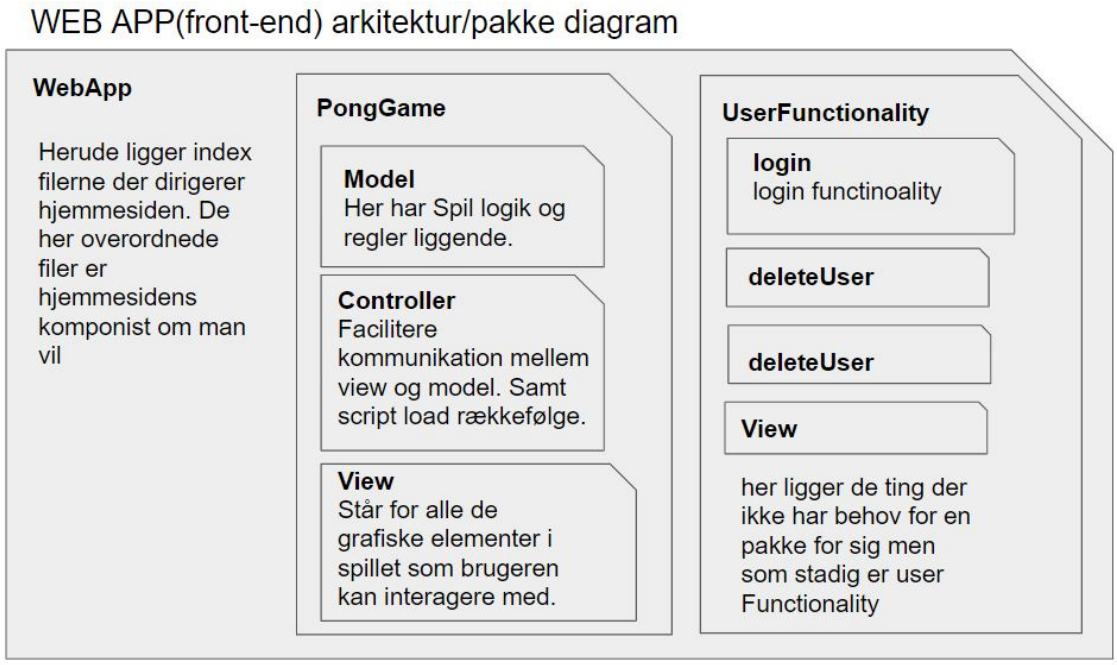
Systemets 4 elementer kan deployes på 4 forskellige servere såvel som en enkelt server. Men for at lette udviklingen af denne applikation er det et design valg at hele systemet (på nær MySQL databasen) er blevet deployet på den samme tomcat server, hvor webapplikationen er tilgået på localhost. Dette er en mulighed eftersom at en Tomcat server både kan have deployet webapplikationer og servlets. I afsnit 9 kan der findes informationer om, hvordan systemet kan deployes og tilgåes på et netværk.

4 IT-systemets elementer

I dette afsnit bliver designet af sytemets fire elementer, website, game server, REST API og database beskrevet. Dertil vil kommunikationen imellem de centrale dele af applikationen blive beskrevet.

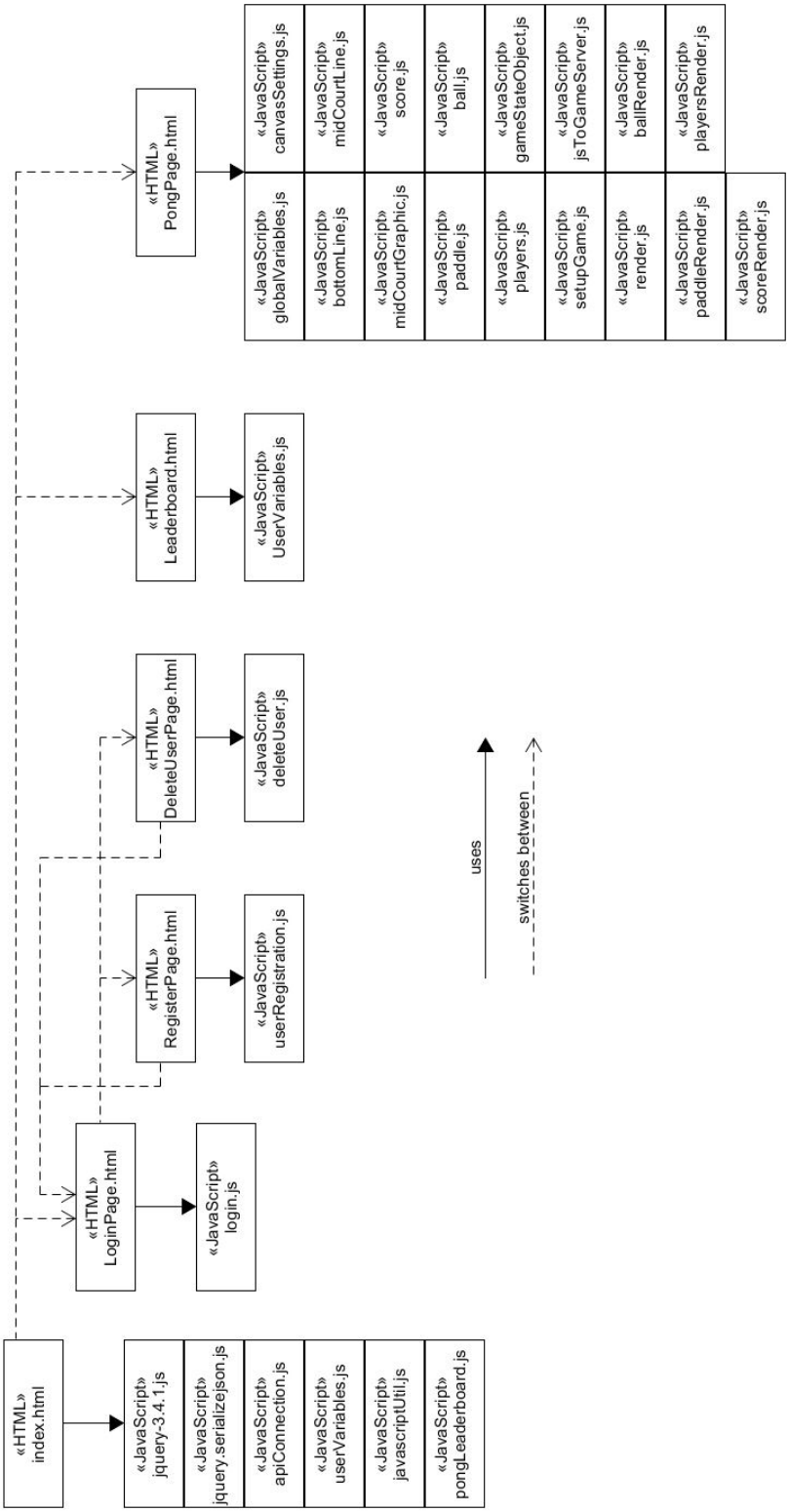
4.1 Website

Hjemmesiden for Pong spillet er blevet bygget af HTML, CSS og JavaScript filer. Her er der valgt at inddele spillet og håndteringen af brugere i to mapper. Spillet, "pongGame", vil der blive lavet efter standard MVC-model, da den kan relateres til en objektorienteret opbygning. Brugerhåndteringen, "userFunctionality", vil blive inddelt efter hvilken use case de håndtere på brugeren



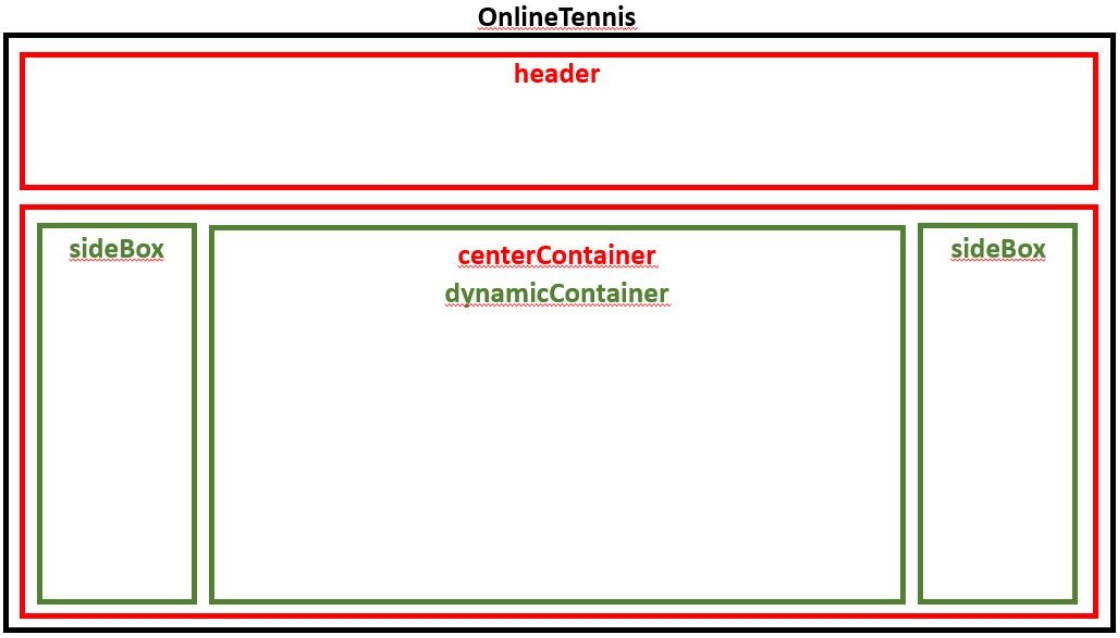
Figur 3: Overblik over websidens arkitektur

For at vise de HTML filer der bruger JavaScript funktionalitet har vi udarbejdet følgene diagram (figur 4) HTML, sammen med CSS filerne, står for det visuelle, og bruger JavaScript'sne til det logiske arbejde.



Figur 4: HTML's klassernes afhængighed af JavaScripts

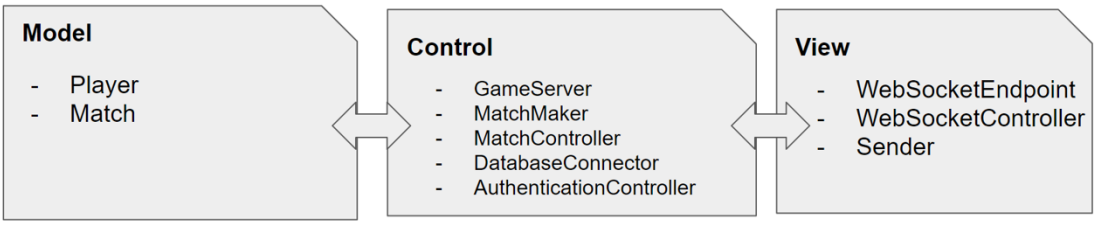
Hele hjemmesiden startes ved index.html-filen. Denne fil er bygget op, så den har sidens header, og en centerContainer division. Denne centerContainer indeholder de to sidebare der er på siden og en 'dynamicContainer' hvor de forskellige dele af hjemmesiden vil blive vist, såsom login-siden og registerings-siden (se figur 5.).



Figur 5: Hjemmesidens overordnede design

4.2 Game Server

Game serveren er en Java applikation, der kommunikerer med web siden og med REST API'en. Applikationen er opbygget som en MVC-arkitektur, hvor View repræsenterer 'connection-teknologien' og Controller repræsenterer selve Game server logikken (videresende beskeder og matchmaking):



Figur 6: Overblik over GameServerens klasser i en MVC struktur.

Game serverens klasser er vist i figur 7, og deres ansvar og formål er beskrevet herunder:

- **GameServer**
  Den centrale controller, som connection-teknologien opretter og videresende beskeder til.

Dens rolle er primært at analysere beskeder, og uddelegerer arbejdet til hjælpende controllere.

- **DatabaseConnector**
  Står for at oprette forbindelse til en vilkårlig database, som game serveren skal benytte I dette tilfælde er det en MySQL databasen, som tilgås via REST API'en. Dette interface implementeres af klassen APIConnector.

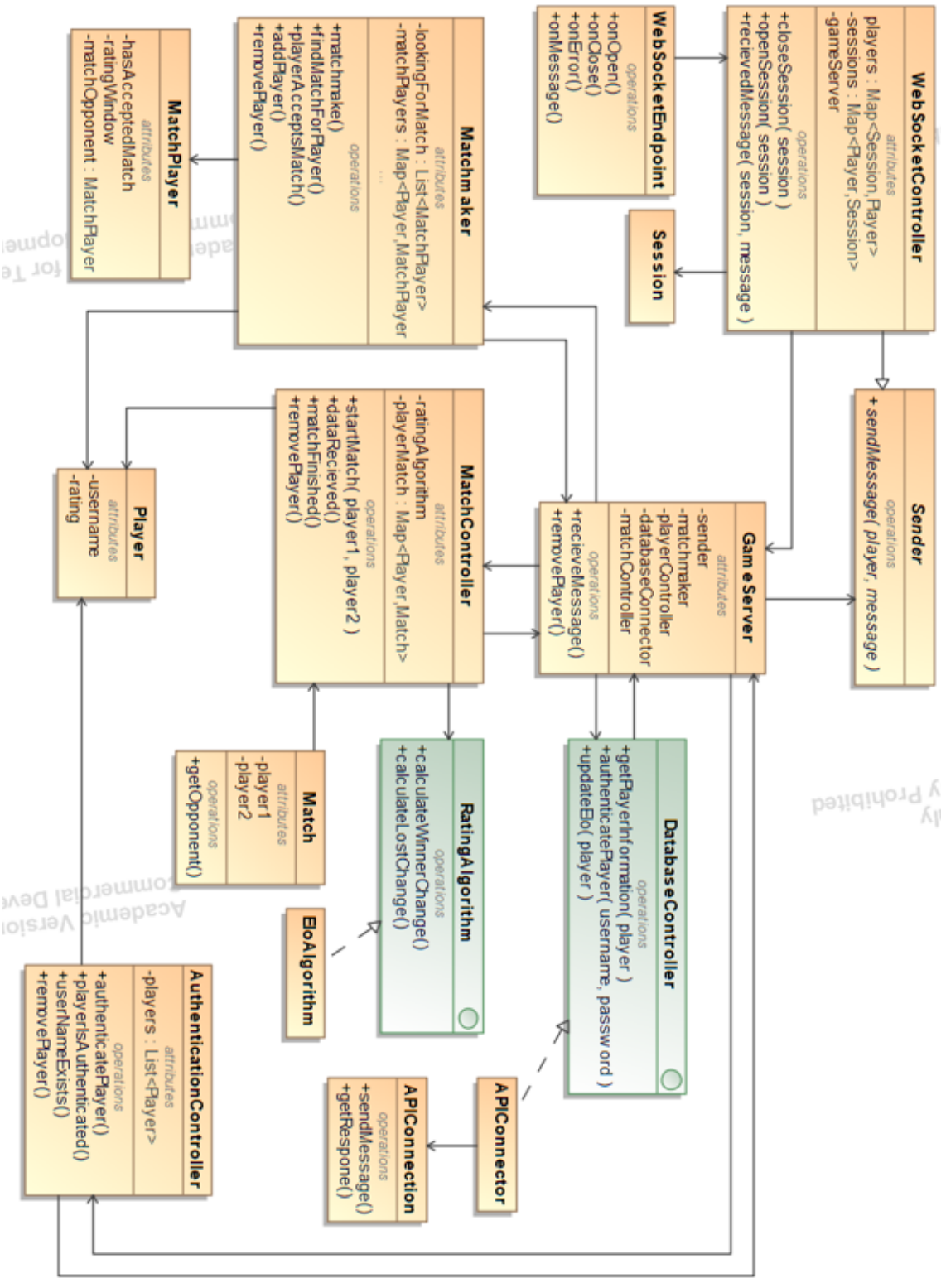
- **MatchMaker**
  Finder modstandere til brugerne, og igangsætter spillene. Klassen kører på sin egen tråd, hvor den tjekker for mulige 'matches' i et fast interval. Den bruger MatchPlayer som en "wrapper" af Player-objektet for at holde styr på ekstra informationer – eksempelvis den tilladte rating-forskel for en spiller før han kan matches.

- **MatchController**
  Håndterer igangværende spil, herunder at videresender beskeder modtaget fra en spiller til spillerens modstander. Derudover håndterer den afslutningen af et spil, ved at benytte implementeringen af RatingAlgorithm til at beregne justeringen af begges spilleres rating.

- **Sender/WebSocket**
  Game server logikken er uafhængig af teknologien, der bruges til forbinde til web siden. Det er opnået ved at GameServer-objektet tager imod beskeder uden at vide hvor de kommer fra (recieveMessage), og sender beskeder via referencen det abstrakte Sender-objekt

  - *WebSocket*
    Som teknologi til at forbinde game serveren og web siden, benyttes WebSocket. JavaScript (web siden) tilbyder ikke "rå" socket programmering, og WebSocket har en to primære fordele over HTTP ift. spillet:
    - *Full Duplex*: Begge kan sende på samme tid
    - *Stateful*: Den kan identificerer at serier af requests, kommer fra samme sted.
  - *WebSocketEndpoint*
    Implementerer den faktiske WebSocket. Klassen definerer et WebSocket-endpoint og er stedet hvor beskeder og forbindelser lander, og beskeder send

  - *WebSocketController*
    Primære rolle er at skabe omdanne forbindelser (Sessions) til spillere (Player), som GameServeren kan modtage og håndtere. Derudover implementerer den Sender-interfacet, og objektet GameServeren til at sende beskeder til en spiller.



Figur 7: Klassediagram over Game Server applikationen (Java), der viser hvem der har *referencer* til hvem

4.3 REST API

REST API'en skal tilbyde en række funktioner, der henter eller manipulerer data fra databasen:

- Oprette en ny bruger
- Hente data om brugeren
- Verificere brugerens password
- Opdatere brugens elo-rating

.

For en RESTful API, er der en række krav som skal opfyldes:

1. **Server/Client**
   Koden der kontakter API'en og koden for selve API'en er to forskellige koder. De taler ikke med hinanden på andre tidspunkter, end når en request sendes eller modtages. Således er 'user interface concerns' altid separeret fra vores 'data storage concerns'

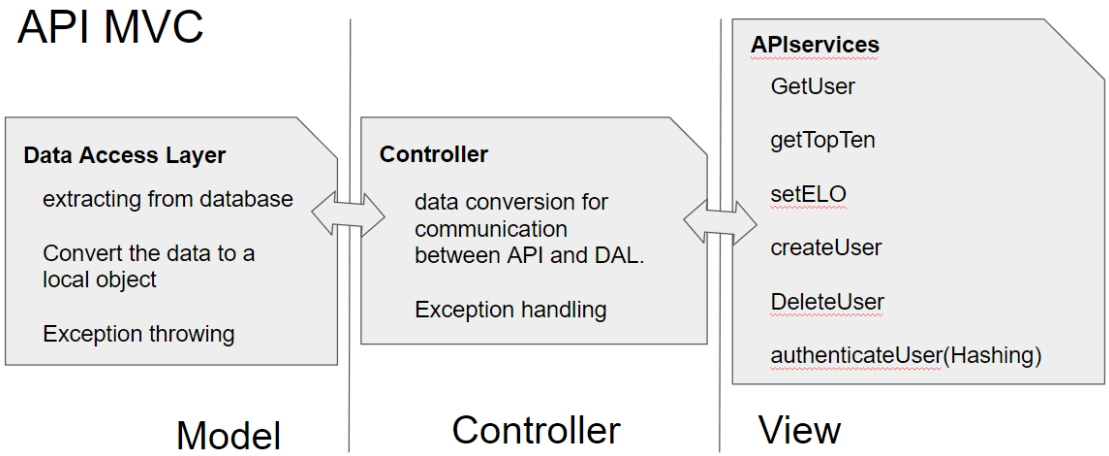
2. **Uniform interface**
   Dette kan delvist sikres nu, ved at give UserService funktionerne gode navne, som er sigene i forhold til den funktionalitet de udbyder.

3. **Layered-System**
   Dette kan vi sikre nu, ved at bruge MVC modellen, Det følgene diagram viser funktionaliteten af hvert lag:

4. **Stateless**
   Stateless beskriver at der ikke bruges context om den user der laver en request under behandling af den request. Altså at alle requests skal være ens. Dette kan først sikres under implementation.

5. **Cacheable**
   Der er ingen use-case i dette system, hvor caching ville være en fordel. Det er fordi vi ikke har nogle requests der er "idempotente". fx ville createUser ikke have samme response hvis du kalder den to gange. Og da spiller-ELOs ændre sig efter hvert spil, er det usikkert at 'getUser' / 'getTopTen' ville have samme respons.

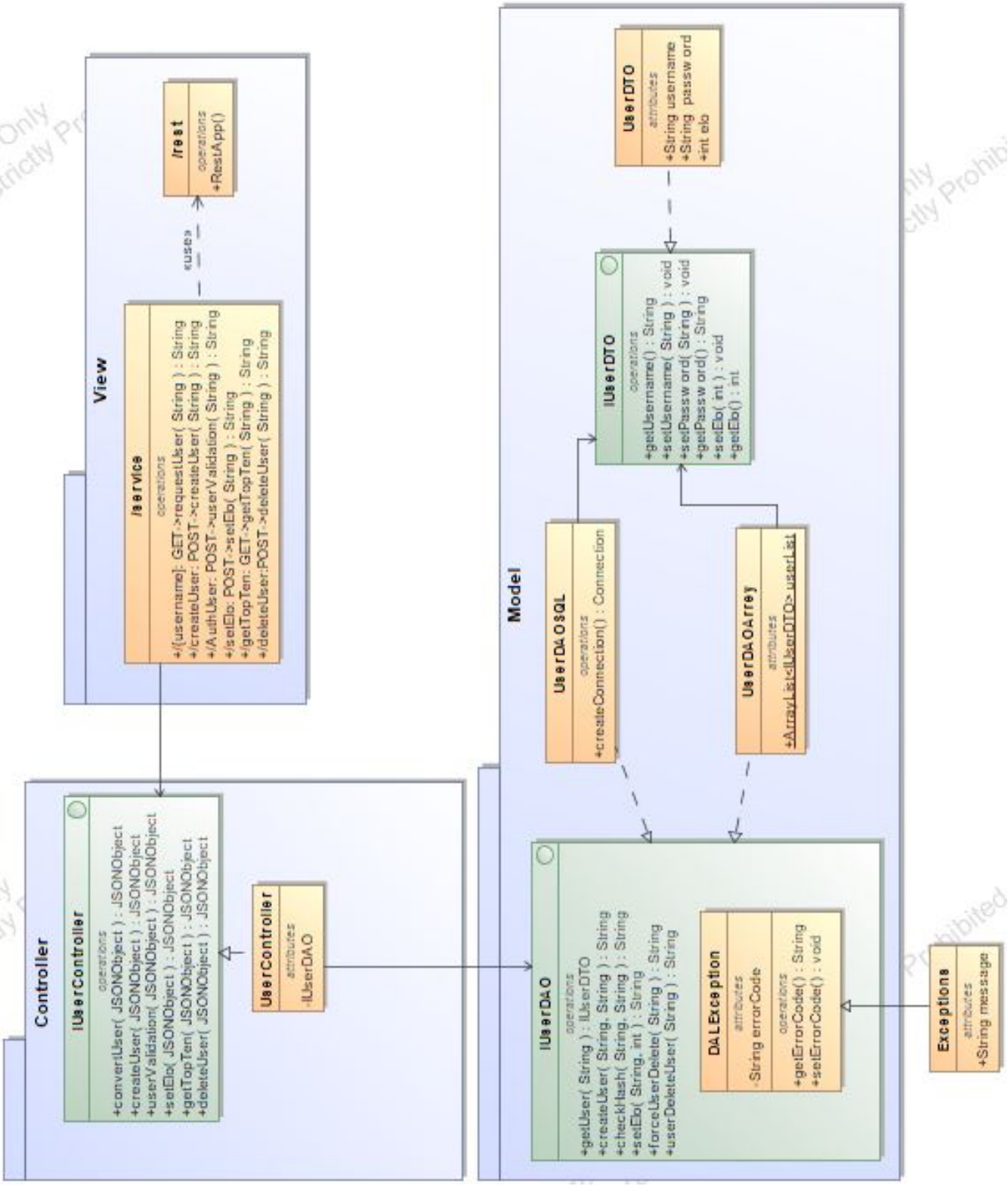
design.PNG



Figur 8: En oversigt over hvordan vi forstillede os at bygge vores API

I figur 11 tager 'view' imod JSON objecter og sender dem videre til 'controllerne' som konvertere JSON-objected til forskellige værdiger som vores 'model' / 'Data-Access-Layer' kan forstå.

Figur9 ses et Design Klasse Diagram for Rest API'en. Det fremgår at applikationen er designet således at applikationen tilgås igennem ressourcen /rest", hvorefter den ønskede ressource kan tilgås via /service/ønsket ressource.
Det ses at der findes to klasser: UserDAOSQL og UserDAOArray, som begge implementere IUserDAO. Dette er gjort sådan så, (hvis SQL databasen gik ned), ville en array kunne bruges til at midlertidigt agere database. Det at de begge implementere samme interface gør at vores controller kan arbejde sammen med begge uden problemer.
Vores databaser skal kunne smide exceptions, og derfor er 'DALException' (som nedarver exceptions) oprettet.
Disse exceptions bliver håndteret af controller laget, og sendes videre (i JSON format) til 'view', og så til brugeren.



Figur 9: Design Klasse Diagram - API'et

4.4 Beskedsystem

Applikationerne skal kommunikere med hinanden, og dette kræver at der er et aftalt fælles sprog de benytter. Til dette formål er der oprettet to kodesystemer: ét system til Game Serveren og ét til REST API'en.

Begge systemer bygger på at sende beskeder, der indeholder en kode, som identificere beskeden og eventuel data.

Game Server Game serveren bruger beskederne til at kommunikere med websitet, når spilleren ønsker at spille, samt til at sende data mellem spillerne. Beskedsekvensen der sendes imellem game server og web server kan ses i figur 10. Alle definerede beskeder kan ses i billag C.
API'ets Kodesystem er også implementeret[2], for at der kan kommunikeres med både game serveren og web serveren.

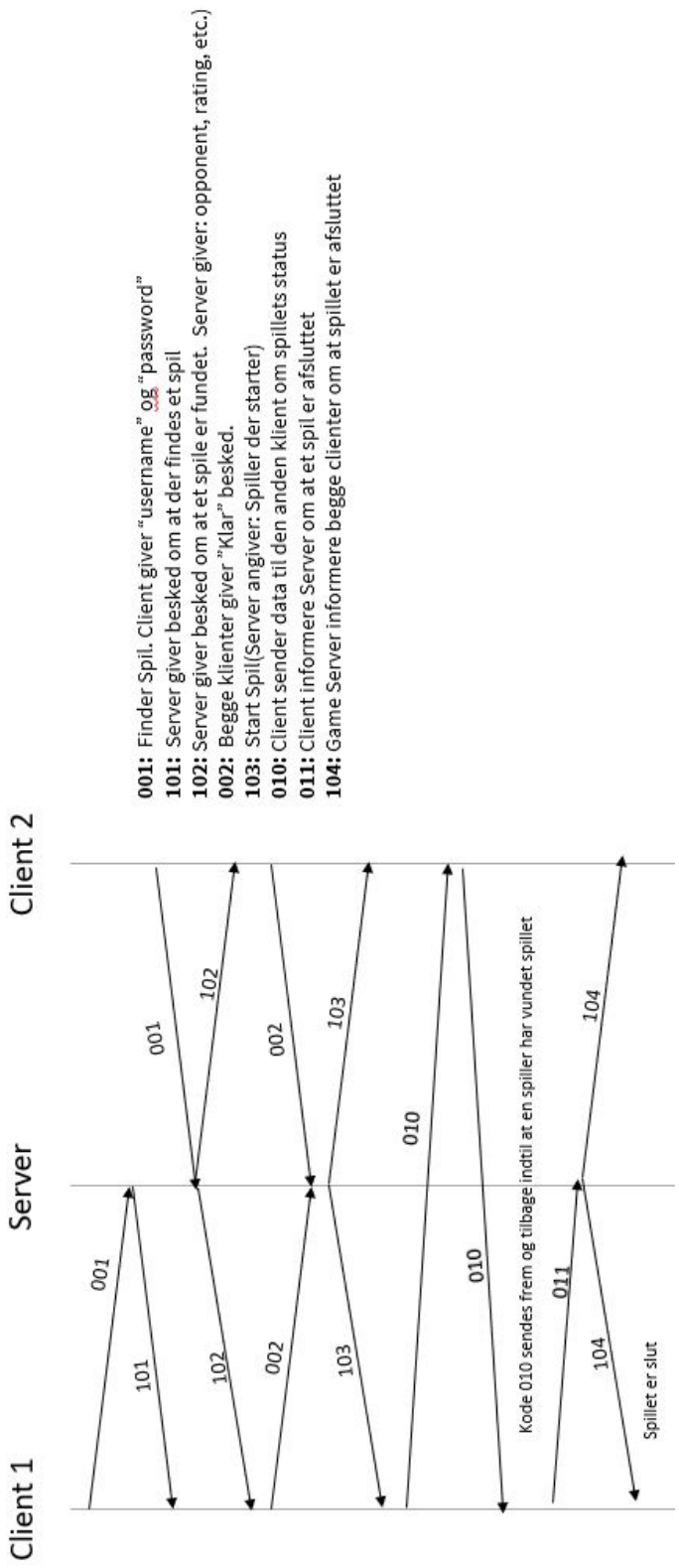
Kommunikations koder fra API'et		
Funktion	code	Beskrivelse
createUser	201 409	User Created User already exist
AuthUser	200 401 410	OK (Accepted) Unarthorized (forkert password) user dont' exist
getUser	200 410	OK (Accepted) user dont' exist
deleteUser	200 410	OK (user deleted) user dont't exist
getTopTen	200 410	OK) user dont't exist
setElo	200 410	OK) user dont't exist
JSONString format	code: "code", "description: "	

Tabel 4: Mest anvendte API kommunikations koder i forhold til dets metoder

Sker der et generelt problem med databasen, f.eks. at der ikke kan oprettes forbindelse, gives der en kode '500'.

Disse koder sendes i JSON-format[3]. JSON formaten øger også skalerbarheden, da en JSONStreng kan tilgås fra flere programmerings sprog

[2]Se tabel 4
[3]For eksempel, se nederst tabel 4



Figur 10: Beskedflow imellem Clienter og Game Server

4.5 Elo-Rating

"Elo-rating"[4] er en rating system brugt til at give spillere en rating baseret på hvor god til spillet de er. Vi vil bruge ELO til at 'matche' to spillere der er nogen lunde lige gode til spillet, når en spiller prøver at finde et spil.

[4]Kilde [1], Afsnittet på siden der hedder 'History'

Del IV
Implementering

5 Webside og UI

5.1 Skift af HTML sider

Som nævnt under Design er index.html en container for hele hjemmesiden. Dette fungerer ved hjælp af en metode kaldt "switchPage(String)", der kan findes i javascriptUtil.js.

```
function switchPage(page){
    $("#dynamicContainer").load(page);
}
```

Denne metode tager mod en .html reference, i en format index.html kan finde, og sætter den ind i dynamicContainer. På denne måde kan der skiftes i mellem de forskellige sider af hjemmesiden, så som login og registrering. Derudover bliver der også brugt to metoder, hide() og show(), der kan gemme og vise divisions af html sider. Dette bliver brugt når der skal vises at der arbejdes på server siden, så brugeren ikke kan give flere kald til systemet.

6 Spillet

I dette afsnit beskrives de vigtigste elementer for at få Pong spillet til at køre.

6.1 Jquery / Ajax

Jquery er et JavaScrit bibliotek som forsimpler JavaScript programmering.
Ajax (Asynchronous JavaScript And XML) er en del af Jquery biblioteket, som vi bruger til at kontakte REST API'en.

```
function apiRequest( type, resourceUrl, successCallback, data=null ){
    $.ajax({
        type: type,              //PUT, POST, DELETE, GET
        url: API_URL + resourceUrl, // URL til REST API + metode
        contentType: "application/json", //Hvad for noget content det er
        data: data,               //Hvad for noget data der bliver sendt (JSON)
        success: successCallback, //Metoden der kaldes ved et successfuldt kald
        error: function(data){    //Metoden der kaldes ved fejl
            console.log(data);
        }
    })
}
```

6.2 Spilkørsel

Når to spillere er blevet matchet bliver initializeGame() kaldt som starter spillets kørsel.

```
function initializeGame(){
    enableGame(opponentName);
    setupGame(chosenScore);
    animate(runGame);
}
```

enableGame(opponentName); viser spilskærmen samt modstanderens navn ovenover.
setupGame(chosenScore); Opretter objekterne for spillet og sætter det i gang. Spillet kører til en spiller har scoret mål lig med den valgte score (chosenScore).
animeate(runGame); Sætter animationen af spillets objekter i gang, med funktionen 'runGame' som parameter

```
var animate =
    function(callback) {
        window.setTimeout(callback, 1000/40);
    };
```

Der er flere elementer i vores animate() metode.
setTimeout(function, milliseconds) kalder metoden, der er angivet som parameter efter et hvis antal millisekunder.
en metode med callback som parameter, sikrer sig at den metode den tager som parameter bliver kørt efter metoden med callback som parameter.
function(callback) { window.setTimeout()} kalder runGame med 40 kald i sekundet (40fps).
Kort fortalt så sætter den fps for spillet.
Den er blevet sat til 40 frames for at sikrer en pc kan køre det. Hvis den givne pc ikke kan køre den fps den er blevet sat til vil den køre mindre, hvilket medfører lag.

```
var runGame = function() {
    if(gameRunning) {
        update();
        render();
        animate(runGame);
    }
};
```

runGame() metoden er en rekursiv metode, der opdaterer det logiske og derefter gengive objekternes nye position, så længe spillet køres.

6.3 Opret og gengivelse elementer

Elementer oprettes via en constructor. Nedenfor ses et eksempel på, hvordan bold elementet er konstrueret.

```
function Ball(x,y) {
    this.x = x;
    this.y = y;
    this.speed = 6;           //Current max speed of the ball
```

```
    this.x_speed = this.speed; //Current horizontal speed of the ball
    this.y_speed = 0;         //Current vertical speed of the ball
    this.radius = 5;          //Radius of the ball object
    this.incrementSpeed = 0.4; //Speed to be incremented with, when a paddle is hit
}
```

Elementer bliver tegnet vha. en object.prototype.render = function(){}. Metoden prototype gør at man kan tilføje ekstra features til et objekts constructor.

```
Ball.prototype.render = function() {
    context.beginPath();
    context.arc(this.x, this.y, this.radius, 2 * Math.PI, false);
    context.fillStyle = "#FF0000";
    context.fill();
};
```

Alle de forskellige object.prototype.render metoder kaldes i render() metoden som køres i den rekursive runGame() metode, så de hele tiden bliver rendered.

```
var render = function(){
    context.fillStyle = "#000000";
    context.fillRect(0, 0, width, height);
    midCourtGraphics.render();
    player1.render();
    player2.render();
    ball.render();
    bottomLine.render();
};
```

6.4 Player og ball bevægelse

Når spilleren skal bevæge sin paddle op og ned bruges der addEventListener metoder som i dette tilfælde lytter efter om en tast holdes nede eller ej på keyboarded.

For at opdatere spillet kontinuerligt skal de bevægelige objekter opdateres. De får objekterne opdateres med object.prototype.update, ligesom i render.

```
var update = function(){
    player1.update();
    player2.update();
    ball.update(player1.paddle, player2.paddle);
};
```

6.5 Opdatering af spillere

Når spillet er i gang bliver der sendt JSON objekter fra klient til klient gennem spil serveren med kode 10. JSON objekterne indeholder spillerens paddle position, boldens position (kun opdateret

når den er på ens egen side for at give en mere glat oplevelse) samt spillerens score (Kun opdateret når en spiller har en højere score end før).

7 MySQL Database Implementering

7.1 Database

Databasen skal kunne indeholde information om brugerne. Denne information skal være:

- Brugernavn
- Elo Score
- Password

.
Dette kan gøres med én enkelt tabel

Field	Type	Null	Key	Default
username	varchar(15)	NO	PRI	NULL
password	varchar(500)	NO		NULL
elo	int(10)	YES		1000

Figur 11: Beskrivelse af databasens schema. NB: Databasen består af en enkelt tabel.

Her har hver række i tabellen et brugernavn, en elo score og et hashed password. Brugerens elo der viser hvor 'god' brugeren er efter en simpel version af et "Elo rating system". Alle passwords er hashede for sikkerheden i at de ikke kan læses direkte fra databasen.

8 API og REST Implementering

8.1 Principper for REST

For at en API kan kaldes 'RESTful', er der en række krav som den skal opfylde:

1. **Stateless**
   Ingen requests bruger stateful context om brugeren til at lave en response. Derfor ved vi at API'en er stateless.

2. **Uniform interface**
   For at have en uniform interface har vi en standard skema på alle vores JSON filer som API'en sender ud. Der er altid en 'code' (som følger HTTP kode definitioner) og en 'description' som indeholder en forklaring på den kode der er sendt med.

3. **Layered-System**
   Den MVC model vi har brugt sikre at alle lage har en "high level purpose"og at der ikke er direkte kommunikation mellem View (UserService) og Model(IUserDAO).

4. **Server/Client**
   Pakken API taler aldrig sammen med nogle andre pakker, den modtager kun requests. Således lever koden op til REST principperence for Client-Server-relationer.

5. **Cacheable**
   Fordi der (som beskrevet i design) ikke er idempotente requests i vores API, skal klienten ikke cache APIens request-reponses.

8.2 Jersey: View-layer

For at lave en URL så brugeren har adgang til vores API, bruges Jersey: JAX-RS library.[5]

```
@Path("/createUser")
@PUT
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public String createUser(String msg){
    IUserController userController = new UserController();
    JSONObject output = userController.createUser(new JSONObject(msg));
    return output.toString();
}
```

Figur 12: Eksempel på API Jersey implementation af "PUT: createUser"metoden

Denne metode 'consumer' (modtager) en JSON fil som den convertere til en String. Vi laver denne String om til en JSON fil igen, fordi det er det som controller laget modtager. Dette er ikke optimalt, men det skyldes at JAX-RS vil gerne konvertere JSON filen med det samme, og vil derfor ikke sende en ren JSON fil igennem til metoden.

8.3 Error Handling

SQLExceptions dækker ikke alle fejl der kan ske i databasen. Derfor er der oprettet en ny Exception klasse, som vi kan kaste hvis der fanges SQLException eller andre Exceptions. Denne har vi kaldt DALException[6], og bliver oprettet som en 'inner class' i IUserDAO. UserDAOArray og UserDAOSQL kan kaste DALExceptions, som så kastes videre til controlleren som handler dem. Controlleren sikre at brugeren får en besked tilbage (i JSON format) på hvad der gik galt, fx: "409: a user with that username already exists".[7]

8.4 Password Hashing

Til at hashe vores passwords bruger vi 'Argon2'. Argon2 er et library som hasher passwords med 'Salt', der er en tilfældig datamængde som kodeordet kan blive mixet med.

[5] Se figur 12
[6] Står for "Data-Access-Layer-Exception"
[7] For alle typer Exceptions og dybere forklaring af DALException, se punkt A under Bilag

9 Game server og Websockets

Game server er ansvarlig for at uddelegere spiltråde, spil og holder styr på at spillet forløber hensigtsmæssigt.

9.1 Websockets

Forbindelsen imellem web serveren og game serveren forgår via Websockets. Websockets er full-dublex og muliggøre at kommunikation imellem server og klient går begge veje. Altså kan Game serveren sende beskeder til klienten, uden at der er foretaget en forespørgsel. En forbindelse mellem klient og server er etableret via websockets mindsker latency og kræver mindre båndbredde da hver besked ikke er påkrævet en header.
En Websocket betragtes som et 'endpoint' i en forbindelse. Når en Websocket skal programmeres anvendes der følgende annotationer.

- @ServerEndpoint: En annotation for en klasse der garanterer at en WebSocket er til gængelig når en specifik URI forespørges.
- @ClientEndpoint: En annotation for en klasse der garanterer at en pågældende klasse bliver betragtet som en Websocket.
- @OnOpen: En annotation for en metode. annotationen sikrer at en metode bliver iværksat når en Websocket bliver etableret.
- @OnMessage: En annotation for en metode. En metode med denne annotation modtager en besked fra en WebServercontainer når en besked er sendt til et endpoint.

9.2 Events og Threads

Game server-logikken igangsættes af 'events' gennem WebSockets. Når WebSocketEndpoint modtager en ny forbindelse eller besked, startes en ny thread (inputtråd), hvilket betyder at game serveren håndterer flere spiller inputs på samme tid.

For at gøre matchmakingen uafhængige af spillerinputs, startes en separat tråd, der med faste intervaller tjekker for mulige 'matches' mellem spillerne.

Da denne gennemgår en liste af mulige spillere, som potentielt opdateres af inputtrådene samtidigt, kan der forekomme en 'ConcurrentModificationException'. Dette undgås ved at samle koden der manipulerer listen i et synchronize kald:

```
synchronized( lookingFormatch ){
  // matchmake
}
```

Dette låser objektet, så det ikke kan manipuleres af andre tråde

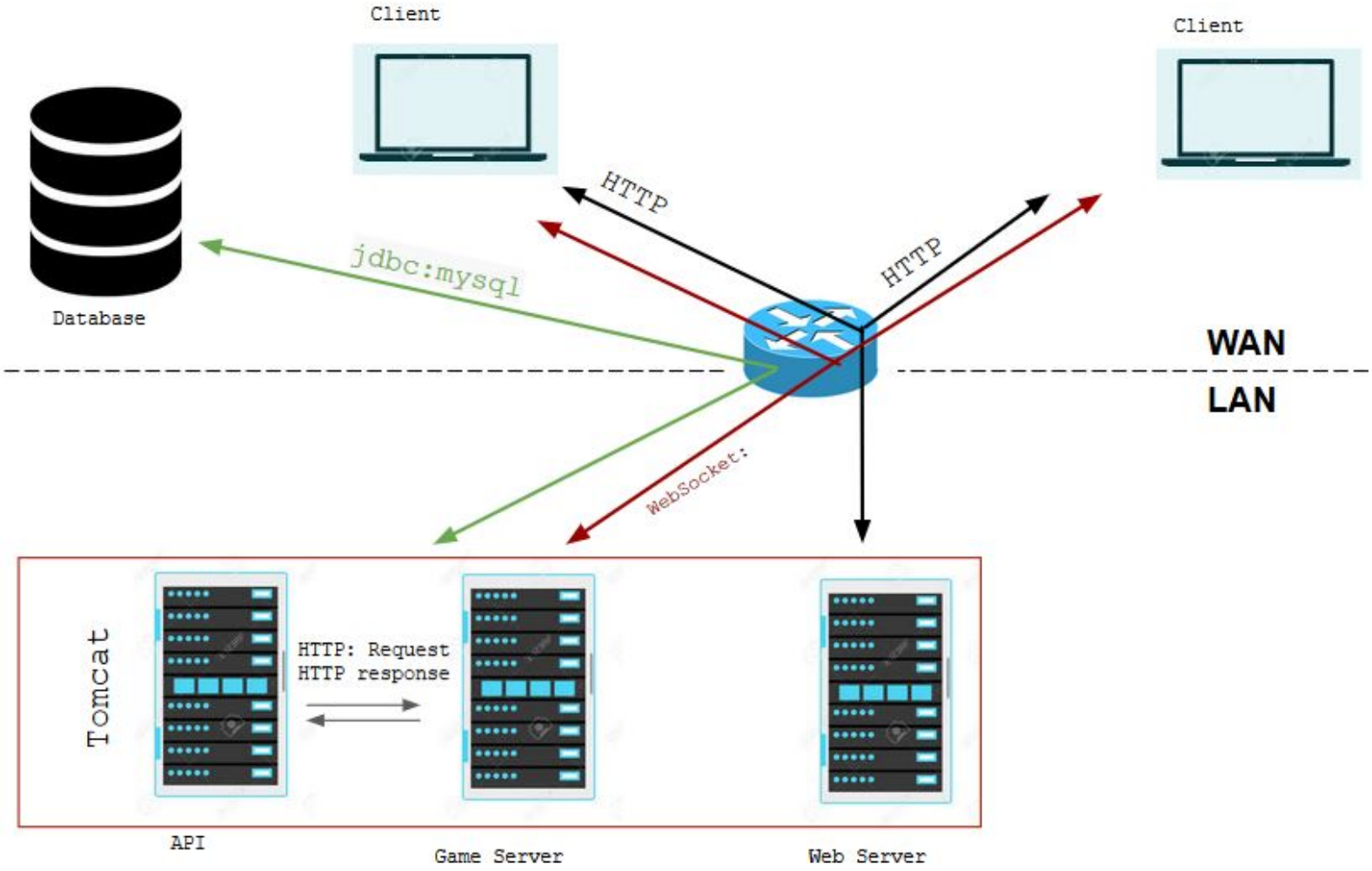
9.3 MatchPlayer (Inner Class)

Matchmaker-klassen bruger som den eneste ekstra information omkring brugeren. I stedet for at opbevare det i Player-objektet, så alle klasser kan tilgå det, benyttes en 'wrapper'-klasse:

MatchPlayer. Den indeholder et Player-objekt, samt ekstra information omkring den Player (ratingWindow, matchedOpponent etc.), og opbevares i et HashMap hvor Player-objektet er nøglen. Da Matchmaker-klassen er den *eneste*, der bruger denne klasse, er den implementeret som en *inner-class* i Matchmaker.

10 Praktisk opsætning af Applikationen

I dette afsnit vil det blive beskrevet hvordan systemet kan blive implementeret således at den kan tilgås fra et globalt netværk.
Systemets fire elementer (Web Server, Game Server, Rest API og database) er separeret sådan så de kan køre på hver deres server placeret forskellige steder i verden. Alle applikationer, på nær databasen, kommunikerer via HTTP. Websitet kan blot hostes på en web server. Java-applikationerne (game server og Rest API), kan ikke hostes på en almindelig web server. Derfor benyttes Apache Tomcat, der kan køre Java-applikationer i et HTTP-miljø.
Figuren nedenfor viser hvordan systemet kan eksekvere og tilgås på det globale netværk, samt, hvordan elementerne er placeret i forhold til hinanden på netværket.



Figur 13: Netværkets topologi. Figuren viser HUAWEI routeren der adskiller henholdsvis WAN - og LAN netværket. Serverene er opstillet på LAN siden af routeren mens at klienterne findes på WAN siden af routerne.
Kommunikation med GameServeren er markeret med rød, kommunikation med database er markeret med grøn og kommunikation med WebServer er markeret med sort.

Figur 13 viser 2 PC'er der er forbundet til det globale netværk. Servere og servlets (WebServerer, Game Server og API'et) findes på LAN siden af en router og kan være deployed på hver deres separate server eller på en enkelt Tomcat, som der er blevet gjort for denne opgave. Figuren illustrerer endvidere at Web Serveren kommunikere med Game Serveren og API'et mens at klienterne kun kommunikere med Web Serveren.

Del V
Test

11 Test af applikationens opførelse på et netværk

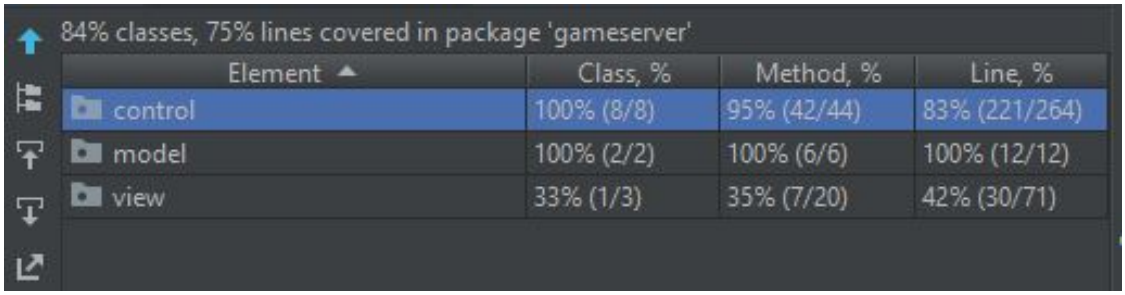
For løbende at få et realistisk bud på netværks-forsinkelsen mellem to spillere i to forskellige netværk, er der benyttet en router med en statisk IP, som isolerer alle applikationerne i sit eget lukkede netværk.
Testopstillingen er som illustreret på figur 13, hvor webserver, Java-applikationer bliver deployed på den samme tomcat server.
Testopstillingen har været følgende:

1. Opstilling
    1.1. 3x computere
    1.2. Tænd HUAWAI router (router med simkort der har sit eget trådløse netværk).
    1.3. En enkelt computer/laptop tilsluttes routeren netværk.
    1.4. Øvrige laptops/computere tilsluttes andet netwærk.
2. Router konfiguration
    2.1. Tilgå den router konfigurationer via laptop/computer der er tilsluttet routerens netværk ved at anvende taste dens IP adresse ind i en browser. eks. 62.79.16.17.
    2.2. På routerens website logges der ind ved at anvende username: "admin"og password: "admin".
    2.3. Under fane security vælges Virtual Servers. Her indsættes IP adressen og port på den laptop/computer der er tilsluttet routerens netværk. Porten sættes til 8080 (konvention for port til en webserver).
3. Start tomcat server og spil spillet.
    3.1. Applikationen sættes igang og main metoden (i klassen Main) startes.
    3.2. Klienter kontakter applikationens URL 62.79.16.17:8080 og kommer ind på websitet.
    3.3. Der startes et spil imellem de to klienter.

Resultat af test:
Af testen fremgik det at forsinkelsen af boldens, samt modstanderens paddle, opdateringer blev forøget. Dette kunne også bekræftes ved at sende en 'ping' til serveren fra én af klienterne. Når der anvendes en router var den gennemsnitlige ping en gennemsnits RTT på ca. 50 ms. Dette står i skarp kontrast til en gennemsnitlig RTT på ca. 1 ms når pong spillet bliver eksekveret lokalt.

12 Unit test: Game Server

Der er lavet UnitTest af langt de fleste metoder i applikationen game server. Figuren nedenfor viser code coverage af de metoder som er blevet "ramt"af testen.

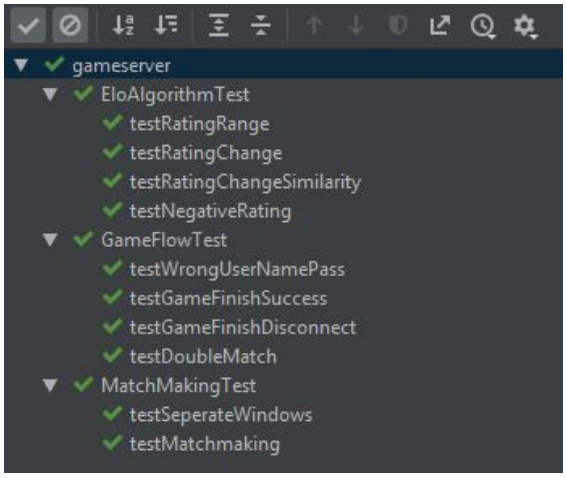


Figur 14: Code coverage af testet kode i 'gameserver'

Af figur 14 fremgår det at 42 ud af 44 metoder i pakken 'control' er blevet testet. For at holde en isoleret test af metoderne er der ikke testet de metoder, der holder kontakt med rest API'et. Disse metoder er:

- dataRecieved
- updateElo
- getPlayerInformation

Figur 15 nedenfor viser de tests der er blevet udført for Game Server applikationen.



Figur 15: Test udført på game server Applikationen

13 Unit test: API

Der er lavet unitTest på 37 af 48 metoder i API pakken. Figur 16 nedenfor viser code coverege af de metoder som er blevet "ramt"af testen.

Element	Class, %	Method, %	Line, %
Controller	100% (1/1)	100% (9/9)	74% (83/111)
DataLayer	100% (4/4)	90% (28/31)	80% (170/212)
Jersey	0% (0/2)	0% (0/8)	0% (0/21)

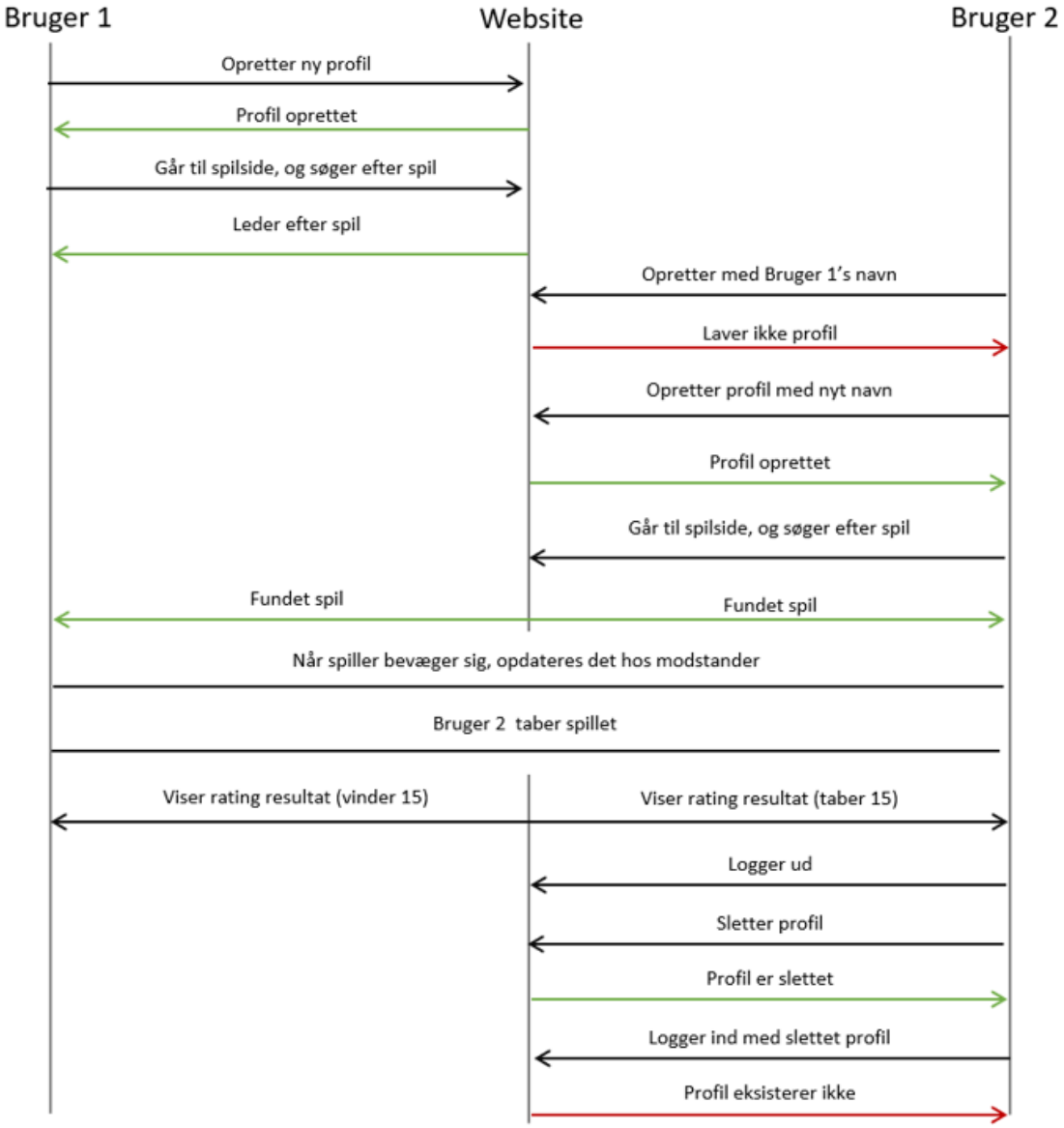
Figur 16: Code coverage af testet kode i 'API'

Hver test metode tjekker et success scenario. Dvs. der ikke er testet om fejl giver de rigtige DALExceptions. Dette ville kræve at der blev skrevet en jUnit test for hver scenario, for hver metode i API'en. Der er ikke lavet test af Jersey pakken, da funktionerne kun sender deres input fra Jersey, videre til 'controller' laget.

14 Systemtest

Systemet er testet i sin helhed ved at interagere med websitet, og anse det bagvedliggende system, efter 'black-box' princippet.
Testen[8] er udført ved at køre hele systemet på en DTU Databar maskine - dvs. game server, API og website. På samme maskine åbnes to browsere, som hver repræsenterer en bruger (spiller 1 og spiller 2). De to brugere udfører en række handlinger, og websitet forventes at svare med noget bestemt. Testet blev fuldendt med forventede resultater.

[8] Se figur 17 for visuel gennemgang af testen



Figur 17: Handlinger og resultater af systemtest.

Del VI
Opsamling

15 Diskussion

De 4 applikationer, som systemet består af, er hver især ikke rige på features - de tilbyder få og simple funktioner. De funktioner der er implementeret anses dog for at være finpudset og blot med få fejl og mangler. Trods finpudsningen, er produktet, set med en brugers perspektiv på funktioner, dog ikke imponerende. Det vigtige ved systemet, og dér hvor arbejdet er langt, er at implementere en nødvendig infrastruktur der standardiserer kommunikation mellem applikationerne. Denne infrastruktur er teknisk kompliceret, og har krævet stort fokus, men er til gengæld rygraden for at udvikle en række nye features og optimeringer til systemet. Eksempler på dette er udvidelse af brugerinformationer, interaktion mellem brugere, flere features i selve spillet osv.

16 Konklusion

Der er blevet udarbejdet et funktionelt Pong spil, der kan spilles online af to spillere i hver deres browser over en server. Derudover en database der gemmer deres bruger informationer enten lokalt eller i en remote SQLDatabse. Grundet dette projekt har været et selvfundet projekt og ikke det foreslåede, har det givet god mulighed for at gå udenfor pensum og meget har været nye problemstillinger. Der kan optimeres på det, men det afleverede projekt giver en oplevelse af en fungerende website hvor der kan oprettes en profil og spilles pong - mod andre online modstanere.

Del VII
Kilder

Referencer

[1] Volunteers for wikipedia.org. *Elo Rating System*. URL: `https://en.wikipedia.org/wiki/Elo_rating_system`. (senest tilgået 20.06.2019).

Del VIII
Bilag

Disclaimer: Dele af disse billag kommer fra tidligere afleverede raporter der kan derfor forekomme selvplagiat:

A Forklaring af Exceptions som vi anvender

Når et program kører, kan det ske at det støder ind i et problem der vil stoppe dets normale flow eller helt crashe det. Dette problem som programmet løber ind i kaldes en 'exception'.
For at undgå at programmet crasher kan man tage og 'catche' disse exceptions for derefter at behandle dem og/eller melde tilbage til brugeren at der er sket en fejl og om muligt hvilken. Denne praksis kaldes for "exception handling".
Helt teknisk, når der sker en fejl i en metode, så bliver der skabt et exception objekt, med informationer, som bliver afleveret til runtime systemet. Som derefter vil lukke programmet og skrive hvad der er galt.
Når en metode har kastet en exception, så vil systemet automatisk prøve at finde en exception håndtering i call stakken (de metoder koden er gået igennem indtil fejlen kom).

A.1 Try-catch

Try-catch er en måde, hvor der bliver afprøvet en stump kode, i en try blok, og som derefter fanger en eller flere exceptions, hvis de er der, hvorefter man kan håndtere dem.
Kort sagt bruges try-catch til at omkranse noget kode som man tror vil smide en exception.
Her er nogle punkter man skal huske på:

- Man skal kun omkranse det kode som man tror vil smide en exception, da al resterende kode i try-blokken ikke vil blive kørt, hvis der bliver smidt en exception. Kun koden efter catch-blokken bliver kørt.
- Man behøver ikke nødvendigvis specificere den exception man tror vil blive smidt i catch. Man kan også bare skrive en af dens parent klasser, helt op til den kaldet ´Exception´.
- Man kan både skrive en custom besked i catch blokken, eller bare printe exception variable navnet for at se hvad det er for en exception og hvad der sker:
  - System.out.println("custom message");
  - System.out.println(exception);
- Man skal kun bruge try-blokken til at afprøve kode for exceptions, og man skal kun bruge catch-blokken til at håndtere exceptions.
- Man kan have flere catches til en try-blok. Start med de nederst prioriterede exceptions og gå op mod dens parent exceptions, sluttende med Exception klassen. Der bliver dog kun printet 1 exception og det er den der bliver fanget først i try-blokken. Hvis der ikke er den specifikke exception i catchen printer den parent Exception catchen, der burde fange alle exceptions.
- Man kan også lave nested try-catch.

Try-catch syntax:

```
try
{
    //Udsagn der kan smide en exception
}
catch (exception(type) e(objekt))
{
    //Fejlhaandterings kode
}
```

A.2 Throw

'Throw' nøgleordet bruges til at kaste en exceptions. Man kan både kaste 'Checked' og 'Unchecked' exceptions.
Det bliver primært brugt til at kaste custom exceptions:

```
    static void fun() {
        try{
            throw new NullPointerException("Custom besked");
        }
        catch(Exception e){
            System.out.println("Fanget i fun().");
            throw e; // kaster igen
        }
    }

    public static void main(String[] args){
        try{
            fun();
        }
        catch(Exception e){
            System.out.println("Fanget i main.");
        }
    }
```

Dette ville printe begge de printede sætninger

A.3 Throws

Med ´throws´ nøgleordet, deklarerer man at der måske vil ske en exception under metodens kørsel, og der nok skal tages forholdsregler i form a try-catch senere i call stakken.
Når den forventede exception sker, vil metoden stoppe og den vil blive kastet tilbage til hvor metoden blev kaldt fra. Her er det så vigtigt at man har en try-catch klar, hvis man ikke thrower den videre.
Med throws nøgleordet håndterer man kun checked exceptions, da unchecked exceptions, er under ens kontrol og kan rettes ved at ordne koden, og errors er uden for ens kontrol.

Throws syntax:

```
    void eksempel() throws ArithmeticException{
```

```
//Udsagn
 }
```

A.4 Throw vs Throws

Throw	Throws
Throw bruges kun til at kaste en exception	Throws bruges til at deklarere en exception
Man kan ikke propagate exceptions	Checked exceptions kan propagates med Throws
Throw er fulgt af et exception objekt	Throws er fulgt af en klasse
Throw bruges inde i en metode	Throws bruges i en metodes signatur
Man kan ikke 'Throw' flere exceptions samtidig	Man kan deklarere flere exceptions samtidig

A.5 Typer af Exceptions

I Java kendes exceptions under klassen Exception og mere specifikt de andre klasser, der arver fra Exception.

Checked exceptions
Checked exception bliver tjekket når koden compiles. Det betyder at hvis en metode smider en checked exception så skal den håndteres i en try-catch blok, eller metoden så skal metoden deklarere den exception med et throws nøgleord.

- SQLException - Fejl der har med en database at gøre.
- IOException - Fejlslagne eller afbrudte I/O operationer.
- ClassNotFoundException - Når der bliver brugt en klasse som ikke eksisterer.

Unchecked exceptions
Unchecked exceptions bliver ikke tjekket når koden compiles. Det betyder at der ikke kommer en compilation fejl, også når den ikke er håndteret, når den exception bliver smidt.
De kommer tit når en bruger laver input som ikke dur. Som programmør skal man derfor huske at håndtere forkerte input i koden så de ikke kan opstå.

- NullPointerException - Det kan ske når du prøver at interagere med en tom reference.
- OutOfBoundsException - Hvis man prøver at interagere med et index længere end arrayet er langt.
- ArithmeticException - Fx at dividere med 0.
- NumberFormatException - En String, uden rette format, prøver at blive konverteret til en nummer type (int, long etc.)
- IllegalArgumentException - En metodes argument er ikke brugtbart.

Errors
Errors er ikke til at redde. Det kan fx være OutOfMemoryError, VirtualMachineError og AssertionError.

A.6 DALException

I det afleverede system er der blevet lavet en custom exception, DALException, der ligger i klassen IUserDAO.
Dette er en exception der, ligesom de predefinerede exceptions såsom NullPointerException, arver fra Exception klassen.
Denne er lavet for at håndtere de fejl der kan ske med SQL databasen, i dette tilfælde SQLExcception's. Hvis der bliver fanget sådan en exception bliver der smidt en 'error code' som kan give information til brugeren om hvad der er sket. For dette system vil DALException smide en "ERROR_DALERROR", der er kendt under en systemvalgt int, så der kan blive kommunikeret til brugeren at der er sket en fejl.

A.7 Error code

Inde i systemets V_IUI klasse kan man finde såkaldte "ERROR_something"der er en kendt int værdi. Denne int værdi kan man referer til når der skal give en fejlmelding til brugeren. Dette kan ses blive brugt i 'showError()' metoden, hvor at der bliver printet en meddelelse til kunden, alt efter hvilken identificerende int den har fået tilsendt. Her kan man enten få meget præcise errors printet ud til brugeren eller have en generel fejlbesked som den førnævnte "ERROR_DALERROR", der printer meddelelsen "System have run into a problem with the database". Vi har valgt at implementere dette således, for lade det være op til klassen der implementerer V_IUI hvordan disse fejl skal vises til brugeren.

B Interface

Et interface er en reference type i Java. Man refererer til den når man skal bruge Dens metoder i andre klasser. Der er dog ikke implementeret noget i metoderne i en interface. De er abstrakte. Et interface er en slags klasse med en samling af abstrakte metoder, som en anden klasse kan implementere vha. implements nøgleordet.
Når en klasse implementerer et interface så laver den en slags kontrakt med interfacet om at implementere funktionaliteten på metoderne, som interfacet har defineret.

Når man skal lave en interface, skriver man interface i stedet for class.
En klasse beskriver attributter og adfærd som et objekt har, mens en interface indeholder adfærd som en klasse implementerer.
Medmindre klassen som implementerer interfacen er abstrakt, så skal alle metoderne fra interfacet implementeres i klassen.

En interface er forskellig fra en klasse på følgene måder:

- Man kan ikke instansiere en interface.
- En interface har ingen constructor, defineret eller ej.
- Alle metoderne i interfacet er abstrakte. De har ingen tuborg klammer men ender på semikolon.
- Interfaces kan kun indeholde konstanter, som er pr. definition public, static og final.
- En klasse kan implementere flere interfaces, men kun extende en anden klasse
- Et interface kan ikke implementere andre interfaces eller klasser, men kan extend x antal interfaces dog ingen klasser.

Interfaces er en vigtig mekanisme til at få lav kopling (GRASP) fordi at ens kode kun referer til interfacet, og ikke til den konkrete implementering af metoderne i interfacet.
Det gør også koden mere skalerbar.
Det gør at man kan implementere anden kode ud fra den interface uden at det har nogen effekt på ens kode.

B.1 Eksempel på interface

```
public interface IUserDAO {
    /**@author Claes, Simon
     * The purpose of this class is being able to access the UserData.
     * As is also stated in the name Data Access Object (DAO)
     * @return Userdata
     * @throws SQLException
     */

    /**
     * Retrieves a user by ID
     *
     * @param username  The username of the user
```

```
     * @return     User object
     * @throws DALException
     */
    IUserDTO getUser(String username) throws DALException;

    /**
     * Inserts a user into the database
     *
     * @param username    Users username
     * @param password    Users password
     * @return            String confirmation
     * @throws DALException
     */
    String createUser(String username, String password) throws DALException;

    /**
     * Controls the password is correct through hashing
     *
     * @param username      username of User
     * @param password Password of User
     * @return        True if password is correct, else false
     * @throws DALException
     */
    String checkHash(String username, String password) throws DALException;

    /**
     * updates the elo of a player in the database
     * @param username
     * @param elo
     * @return String: error message
     * @throws DALException
     */
    String setElo(String username, int elo) throws DALException;

    /**
     * @author Simon
     * Get the 10 users with highest elo rating.
     * @return IUserDTO List.
     * @throws DALException
     */
    List<IUserDTO> getTopTen() throws DALException;

    /**
     * @author Simon
     * Deleting users in the DB.
     * This function is exclusively for testing purposes.
     * @param username
     * @throws DALException
     */
    public void deleteUser(String username) throws DALException;


    /**
```

```
    * Customizable exception for explaining Database Access Layer exceptions
    */
   class DALException extends Exception {

      //Til Java serialisering...
      private static final long serialVersionUID = 7355418246336739229L;

      public DALException(String msg, Throwable e) {super(msg,e);}
      public DALException(String msg) { super(msg);}
   }
}
```

C Game Server Message System

Game Server Message System Version 2.0

Messages: Each message is sent as a JSON-object, and identified by its code. The message may also have some data attached.

Det første af de tre cifre forklarer hvilken type besked det er 0. Kontakt til server 1. Bekræftelse fra server 2. Fejlmelding fra server

De to næste cifre er til identificering af de forskellige beskeder.

Code	JSON-format	Description
0	GAME INITIALIZING - CLIENT	
001	{ "code" : 001, "username" : username, "password" : password }	FIND GAME Request to server to find game. User authentication via username and password
002	{ "code" : 002 }	ACCEPT GAME Ready / accept game offered by game server
010	{ "code" : 010, "ball" : { "x" : x, "y" : y, "xVel" : xVel, "yVel" : yVel, }, "paddle" : { "y" : y, "yVel" : yVel, } "scores" : [sending score, receiver score] }	GAME DATA Game data. The data update message which is sent between clients to synchronize game state
011	{ Spil }	GAME WON Opponent has won the game. Game server will read this message, forward it to opponent and close connection for both players..

1	GAME INITIALIZING – SERVER	
101	{ "code" : 101, "timeEstimate" : [time in minutes] }	FINDING GAME Finding game. State to client that game is being found. Implicit that username/password was correct. May be used to update the client about finding game.
102	{ "code" : 102, "opponent" : (user object) "gameRules" : (game rules object) }	FOUND GAME Game Found, and these are the informations. Don't start game. **Leave game rules empty for now**
103	{ "code" : 103, "initUpdate" : true/false }	START GAME Signals client that game should be started, and if the client should be the first to send an update message.
104	{ "code" : 104, "hasWon" : true/false, "ratingChange" : DIFF, "oppRatingChange": DIFF }	GAME FINISHED Signals client that the game has finished. Sent to both players after game server recieves 011 message. "winner" signals if the player recieving the message has won or not. "ratingChange" and "oppRatingChange" states the change in recieveing player and opponent's rating.

2	ERROR	
201	{ "code" : 201 }	Wrong username / password (response to 001).
202	{ "code" : 202 }	User is already logged in (response to 001).
203	{ "code" : 203 }	Cannot connect to authentication server
204	{ "code" : 204, }	Wrong message format.
205	{ "code" : 205 }	Not authenticated
210	{ "code" : 210, "ratingChange" : DIFF, "oppRatingChange" : DIFF }	OPPONENT_DISCONNECT Opponent disconnected, and client has won
211	{ "code" : 211 }	No game active