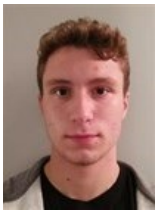
Danmarks Tekniske Universitet

CDIO 1

13 March 2020

Lavet af: Gruppe 25




Oliver Poulsen
s185112

Thomas Hohnen
s195455


Mohamad Abdulfatah Ashmar
s176492

Martin Mårtensson
s195469


Andrey Baskakov
s147199

Daniel Styrbæk-Petersen
s143861

Github Repository https://github.com/GRP25/CDIO_1

Resumé

The purpose of this project is to create a Text User Interface, giving the options of creating, updating, deleting and showing users or closing the program. The user is first asked which type of a database they would like to use with options being (memory based, text based and SQL based), they are then given 5 different functions as mentioned above. Depending on the database choice of the user, information is then either temporary, stored in a file or saved in a SQL database.

We start off by creating a simple usecase and then identifying functional and non-functional requirements, we then make a fully dressed usecase of one of the main functions of the program.

We then design a basic domain model to clarify an idea of how our program might be structured, later a class diagram is made to solidify the program architecture. Then the overall implementation is explained in basic language.

All software used and their versions have been listed below, aswell as the version control usage, few necessary tests have also been documented.

1 Timeregnskab

Skemaet herunder indeholder gruppens regnskab over timer brugt til hver del af opgaven. Denne er blevet opdateret løbende.

1.1 Timeregnskab

Områder/Navne	Oliver	Thomas	Mohamad	Andrey	Martin	Daniel	I alt
Projektplanlægning	1	1	1	1	1	1	**6**
Rapportskrivning	10	15	4	4	3	3	**39**
Kravspecifikation	2	2	2	2	2	2	**12**
Use cases	0	0	2	2	0	0	**4**
Programmering	1	5	5	5	15	15	**46**
Versionsstyring	1	1	1	1	2	2	**8**
Testing	0	1	3	3	0	0	**7**
Gennemlæsning	1	1	1	1	1	1	**6**
Samlet tid i timer	**16**	**26**	**19**	**19**	**24**	**24**	**128**

Tabel 1: Tabel over timeregnskab

Indhold

2 Indledning

Projektet går ud på at lave et lille program, som skal kunne oprette, ændrer, vise og/eller slette brugere. Vi har valgt at brugerdata bliver gemt i en af flere private databaser. Programmet får en simpel TUI grænseflade som enhver bruger nemt kan interagere med.

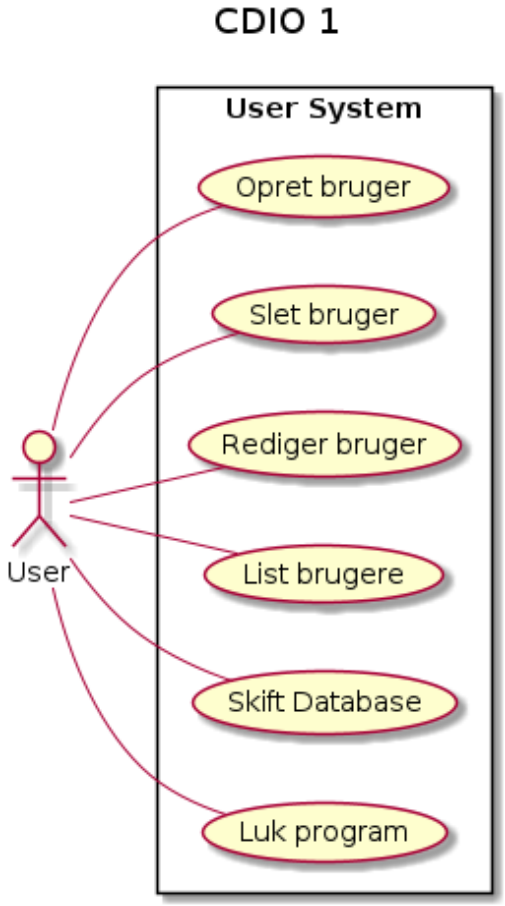
I dette projekt vil det hurtigt blive opdaget at der ikke er blevet taget højde for noget sikkerthed af nogen form. Dette er fordi der er blevet taget udgangspunkt i at brugen allerede er logget ind i systemet.

3 Krav

Da dette projekt blev udleveret blev der samlet set givet 4 krav. Opret en bruger, slet en bruger, Rediger en bruger og vis brugere. Desuden skulle der også laves en TUI/GUI altså et user interface for brugeren. Det sidste krav der også er blevet taget højde for er tre lags modellen. Altså at der er et user interface lag, et funktionallitets lag og et data lag.

3.1 Usecases

Vi har fundet frem til følgende usecases i systemet



Figur 1: Usecase diagram over usermanagement systemet

3.2 Funktionelle krav

1. Opret Bruger
2. Vis brugere
3. Opdater bruger
4. Slet Bruger
5. Lav en TUI

3.3 Ikke-funktionelle krav

1. At man kan opdater mere end en attribut af gangen
2. At kunne lave 3 typer af database
   (a) SQL
   (b) Memory
   (c) Text Fil
3. Lav en GUI
4. Lav sikkerhedssystem

3.4 Moscow Modellen

Til dette projekt er der blevet taget brug af Moscow modellen. Altså Must have, Should have, Could have og Wont have. Her er de forsekllige krav det op i denne model.

M

K1 Bruger Kan Oprette en bruger.
K2 Bruger kan slette en bruger.
K3 Bruger kan rediger en bruger.
K4 Bruger kan få en liste af brugerene.

S

K5 Bruger kan checke om en specifik bruger allerede eksisterer.
K6 Bruger kan filtrere sin søgning om andre brugere.
K7 Bruger kan få en enkelt bruger.

C

K8 Programmet kører på DTU's databars computere
K9 Bruger kan skifte database midt i programmet

W

K10 Programmet skal køre på java 8.
K11 Programmet kører over en GUI.

4 Analyse

4.1 Fully dressed beskrivelse af UC1 (Opret bruger)

Dette afsnit vil omhandle hvordan use casen ”Opret Bruger”bliver fully dressed i dette projekt.

Primær aktør:	Bruger
Forudsætninger:	Bruger skal have personlig information (name, CPR, roles)
Success scenarie:	Bruger data ændringer gemt succesfuldt. Bruger liste returneret.

Main flow

1. Programmet starter og giver mulighed for at oprette en bruger.
2. Brugeren vælger at oprette en bruger.
3. Brugeren indtaster det nødvendige data.
4. Ny bruger er oprettet.

punkt 1 til 4 gentages indtil bruger vælger exit.

Alternate flows

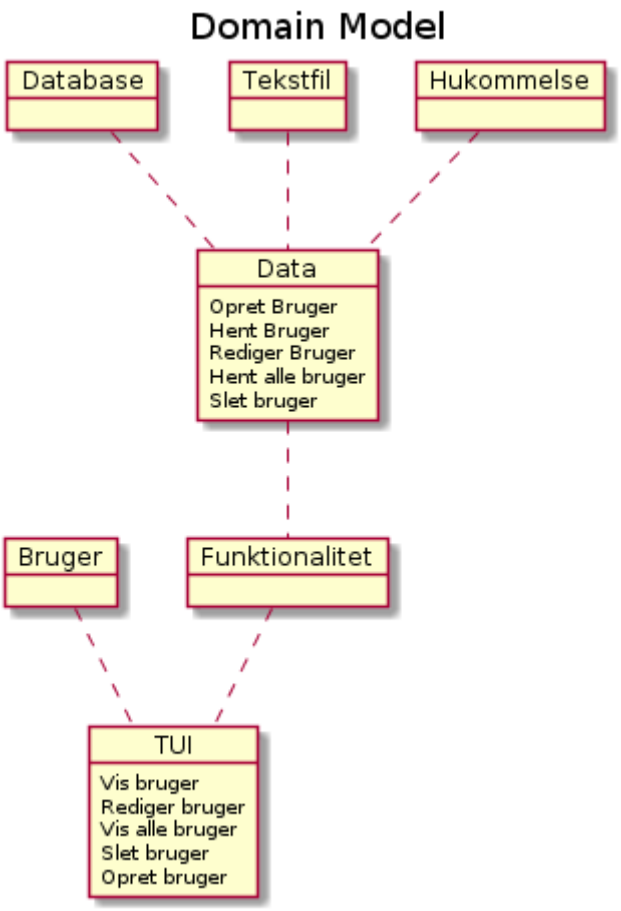
1a Bruger prøver at oprette en ny bruger med et navn som går udenfor maks længde.
   (a) Programmet giver en besked om at navnet er for langt.
   (b) Programmet spørger bruger om at prøve igen.

2a Bruger prøver at oprette en bruger med et CPR-nummer som er utilpas (for kort, for lang) eller som allerede eksisterer.
   (a) Programmet giver en besked om at CPR-nummeret er utilpas.
   (b) Programmet spørger bruger om at prøve igen.

3a Bruger prøver at oprette en ny bruger med en rolle som ikke eksisterer.
   (a) Programmet giver valgmulighed om hvilke roller der kan blive valgt.
   (b) Programmet giver besked om forkert indtastning.
   (c) Programmet spørger bruger om at prøve igen.

4.2 Domain Model

Ud fra use cases er der blevet udviklet en domain model, hvori relationen mellem de forskellige klasser er illustret. I figur 2 er modellen blevet vist. Den viser at brugeren kommunikere med TUI’en, og resten vil blive håndteret gennem system. Data’en som vil blive tilgået i systemet, vil blive håndteret af dataklassen som vil kommunikeret til lokationen hvor data’en er gemt.



Figur 2: Domain Model. Lavet i PlantText UML

5 Design

5.1 Klassediagram

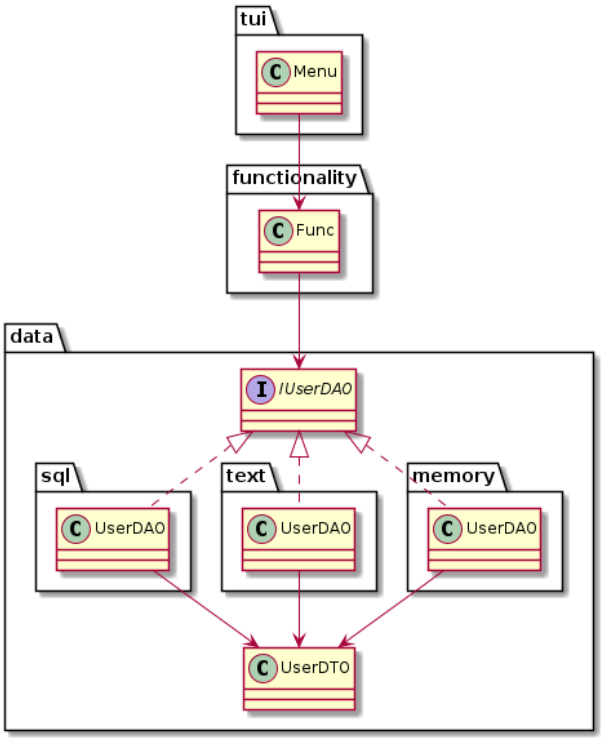
I figur 3 kan man se det klasse diagram der er blevet designet til dette projekt. Dette er et digram der undervejs har fået nogle ændringer for at være sikker på ikke at låse sig fast på noget som kunne ende med at være en fejl.

På klassediagrammet kan man se at projektet er delt op i 3 forskellige lag. Den primære model der er blevet brugt til at bygge dette projekt op er 3-lagsmodellen. Dette er en model hvor alle dele af et projekt er inddelt i 3 forskellige lag. Disse lag er Præsentationslag, Funktionalitetslag og datalag. Når man andvender 3-lagsmodellen ordenligt anvender man automatisk også et af GRASP principperne hvilket er lav kobling. Altså at alle klasser kun snakker sammen med de klasser som er nødvendigt. Her kommer interface også ind i billedet. Selvom et interface ikke er en klasse skal alle klasser fra datalaget snakke igennem et interface for at komme op til funktionslaget. Det er vigtigt at understrege at Det er kun funktionalitetslaget laget der kan snakke sammen med både datalaget og præsentationslaget, altså datalaget kan ikke snakke sammen med præsentationslaget.

- Præsentationslag: Dette lag indeholder klasser som brugeren direkte interagerer med, dette er i denne omgang en menu som brugeren tilgår igennem terminalen

på sin computer.

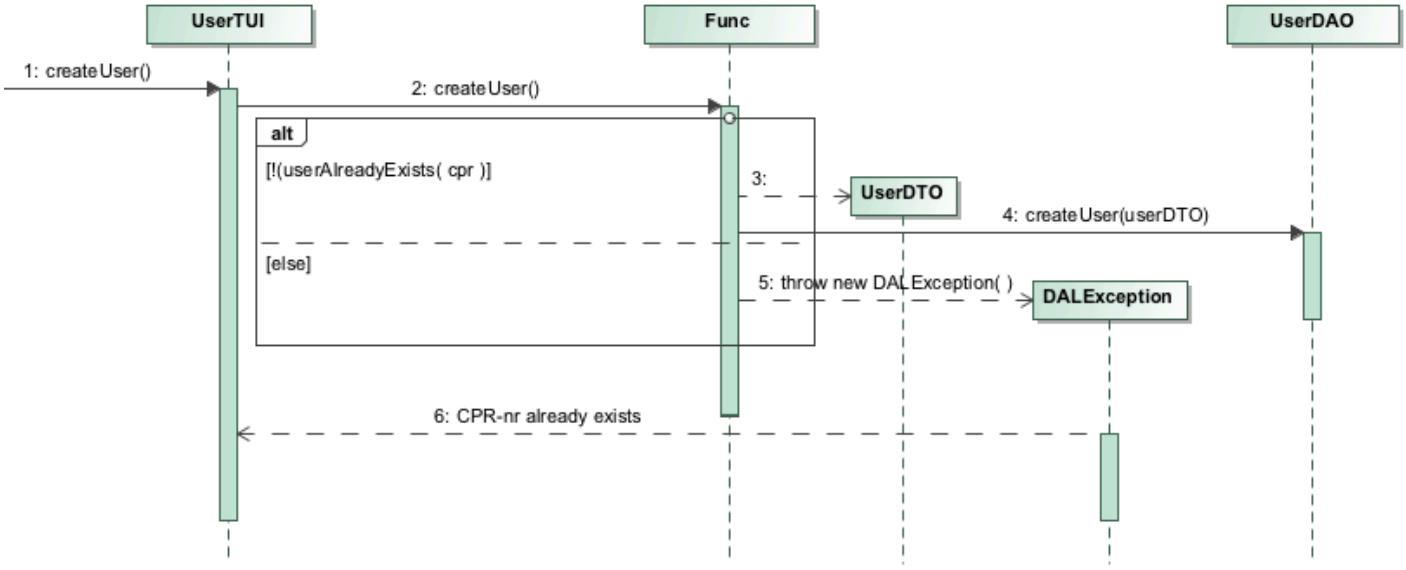
- Funktionalitetslag: Dette lag indeholder funktioner som hverken har at gøre med bruger input eller selve databasen. D.V.S. funktioner som manipulere data og sender det frem og tilbage imellem de 2 andre lag uden at være afhængig af de andre lags ansvarsområder.
- Datalag: Dette lags ansvarsområde er at hente og gemme data, i diverse steder i hukommelsen, og videre sende dem til funktionalitetslaget.



Figur 3: Klassediagram over bruger systemet

5.2 Sequence Diagram

Her viser vores sekvensdiagram, hvordan man opretter en ny bruger, og hvis brugeren allerede findes, så viser DALExpation klasse en meddelelse om, at brugeren allerede findes.



Figur 4: Sequence Digram over oprette brugeren

6 Implementering

6.1 Præsentationslaget

I præsentations klassen har vi valgt at implementere en TUI (*text user interface*), præsentations klassen har kun afgang til snakke med funktionalitetsklassen, og har ikke adgang til database klasserne. TUI'en bliver først kaldt fra *Main()* ved at kalde på metoden *chooseDatabase*, hvor brugeren bliver bedt om at vælge hvilken database brugeren ønsker at gøre brug af.

Når brugeren vælger hvilken database de kunne tænke sig at bruge bliver func klassen initialiseret, hvor den så peger på enten text, memory eller sql. På denne måde sikre vi os at den samme database bliver brugt under hele sessionen, som især er vigtigt for memory databasen. Da det skal være den samme memory der er i brug under hele sessionen. Det er dog ikke muligt at skifte database, imens sessionen er i gang.

Når databasen er initialiseret, kalder TUI'en sin egen *showMenu()* metode, hvor brugeren kan kører de forskellige operationer som programmet har. *createUser*, *deleteUser*, *updateUser*, *showUserList* og *exit*.

I slutningen af hver metode kalds eksekvering, bliver showMenu kaldt igen så programmet ikke terminere før brugeren selv giver en exit kommando.

Vi har en række metoder til at validere input fra brugeren, navnligt metoderne *validateString*, som sikrer os at brugeren indtaster en korrekt string. Dette gøres ved brug at et *do-while* loop, som tjekker hvorvidt om der er nogle digits (integers) i brugerens input. De to andre validerings metoder vi bruger er *validateCPR* og *validateAttr()*, som fungere på samme måde ved hjælp af *do-while*.

6.2 Funktionalitetslaget

I funktionalitets klassen, anvendes *IUserDAO* interface'et, hvori metoderne til kommunikation med datalaget er blevet defineret. Ved oprettelsen af *Func* objektet, defineres det konkrete datalag objekt, som bliver benyttet, i konstruktøren, hvorved data'erne kan blive kan blive læst og skrevet.

Programmet indeholder kun en funktionslags klasse, som User Interfacet bruger. Den indeholder de nødvendige funktioner, til kommunikation med datalaget.

Funktionalitetslaget står for at kommunikationen og logikken, mellem datalaget og User Interface'et. Ved oprettelse af brugere, bliver alle de nødvendige parameter send med *createUser* metoden, og funktionen selv står for oprettelse af dets unikke bruger id til oprettelsen af *UserDTO* objektet.

Data'erne som modtages via datalaget kræver ikke yderligere manipulation, da datalaget returnere bruger som et *UserDTO* objekt.

6.3 Datalaget

De tre klasser i datalaget, håndtere data'en til henholdsvis Memory, Tekst filen og SQL databasen. Alle klasserne implementere interface'et *IUserDAO*. Med brugen af interface'et, kan funktionsnalitits laget kalde på de oblikatoriskte metoder, i klassen. Dette gør at man

gennem interfacet kan hente informationer gennem en af de 3 klasser da alle klasserne har den samme metode men som måske skal skrives på en anden måde.

Ved at bruge interface'et, skal alle datalags klasserne, som implementere interface'et, indeholde alle dets metoder. Dette betyder dog også at der godt kan være andre metoder i klassen og en klasse skal heller ikke nødvendig vis bruge de metoder der er blevet give i interfacet. Som sagt er der blevet lavaet 3 forskellige datalags klasser. Det første er MemoryDataDAO klassen:

Memory I denne klasse bliver alle informationerne gemt i RAM i form af et HashMap. HashMappet er bygget op således at nøglen er en Integer og elementet er en UserDTO, altså en bruger. Dette gør at nøglen kan blive sat til at være det id, som en bruger har, og derved kan man finde brugen ved at søge efter vedkommendes id.

Fil I denne klasse bliver informationerne om brugerne først gemt i en arraylist, derefter bliver indholdet af arraylisten gemt over til en tekstfil ved hjælp af saveUsers metoden. Vi henter information fra tekstfilen ved hjælp af loadUsers metoden. Information på brugerændringer bliver håndteret ved at slette først og derefter genoprette den samme bruger med de valgte ændringer.

SQL Vi har gjort det muligt at gemme data til en SQLite database, Dette gør vi ved hjælp af Xerial's SQLite-JDBC bibliotek som vi har tilføjet til projektet ved hjælp af maven.

Når man køre metoden "createUser", så sendes UserDTO klassen til metoden i User-DAO'en og derefter bruges JDBC's PreparedStatement som en slags serializer som konverterer objektet til en række i en tabel i SQL databasen.

Hvis man køre metoden "getUserLists"så bruges JDBC klassen Statement til at spørge databasen om al data fra alle "users"og det blicer så returneret i JDBC's ResultSet som er et slags array, dette ResultSet bruges så til at lave et ArrayList af UserDTO's som returneres af metoden "getUserLists"

6.4 DTO og DAO

Gennem det her projekt er der blevet tager brug af to "typer"af klasser. DTO og DAO klasser. En DTO klasse er en Data Transer Obejct klasse som i dette projekt bliver brugt til at persistere den bruger data der bliver indtastet og kan derved bruges til at sende det videre til en database. En DAO er en klasse som kan bruges til at tilgå en database. Dette betyder at der i dette projekt er 3 forskellige DAO'er, en til memory, en til text filer og en til en SQL database.

6.5 Interface

Ved hvert af de 3 lag er der blevet brugt et Interface. Et interface bruges til at flere forskellige klasser skal kunne gøre brug er metoder med det samme navn. Med andre ord kan man kalde et Interface som en slags skabelon for hvordan en klasse skal bygges op og få en ide om hvad klassen skal kunne gøre. I dette projekt er Interface blevet brugt til at få flere klasser til at kunne snakke sammen gennem et interface for at kunne få adgang til forskellige databaser. Dog kan et interface også bruges hvis der er flere programmører der arbejder sammen siden alle har det samme skelet at arbejde ud fra.

7 Versionering og konfiguration

7.1 Versionsstyring

Til dette projekt er der blevet brugt Git til at holde styr på det lokale og derved også Github.com til et online repository. Det at der er blevet brugt Github.com har gjort at projketet er blevet delt meget nemmere mellem gruppe medlemmer og hvilket også har resulteret i at det har været nemmere at kunne arbejde sammen om det samme projeket. Her kan man også se linket til det repository der er blevet brugt til projeket, som også medfører alle commits.

https://github.com/GRP25/CDIO_1

Branching

Et meget vigtigt element der kommer med versionsstyring er brahcning. Gennem det her projekt er der blevet gjort meget brug af branching ved at lave mange forskellige branches når der skulle laves en ny feture. Som udgangspunkt begynder man med at have 2 branches dev og master. Derved har der gennem dette projekt blevet sørget for at master banchen kun er blevet opdateret når programmet kører uden problemer, og dev branchen er der hvor der typisk bliver arbejdet på. Dog er der ud fra dev blevet lavet mange andre branches, som har gjort at der kunne blive arbejdet på flere fetures på en gang uden at give andre problemer. En ting der kommer med når man bruger branching er merge konflikter. Altså hvis der er 2 forskellige ting skrevet på den samme linje kan git ikke finde ud af hvilken version den skal tage som den endelige. Dette bliver løst ved at kigge på begge versioner side om side og kigge der hvor konflikten opstår og derved vælge hvad der skal bruges.

7.2 Konfigurationsstyring

Den store forskel der har været i udviklings miljøet har været at nogle enkelte har arbejdet med Linux og Mac OSX hvor imod at der har været nogle som har arbejdet med windows. Dog har alle udviklerne arbejdet i samme IDE som er IntelliJ. Der er desuden også blevet gjort brug af et maven bibliotek.

- IntelliJ IDEA 2019.3.1 (Ultimate Edition)
- Windows 10: Version 1903 (Build 18362.535)
- Arch Linux
- Mac OSX
- Java 1.8
- Maven: Junit: Junit:4.12

8 Test

8.1 Test

For at teste klasser har vi benyttet JUnit test.

TestCase	Gem data i tekst fil
Scenarie	Brugeren begynder at oprette en bruger
Krav	K1
Preconditions	Ingen
Postcondition s	Brugeren blev oprettet
Testprocedue	Brugeren starter med at oprette ny bruger og indtaster de rigtige parametre, hvis Id allerede findes, så brugeren får en exception ellers bliver brugeren skabt
Testresultat	Brugeren blev skabt som forventet.
Testet af	Andrey og Mohamad
Dato	12/03/2020
Testmiljø	JUnit 4

Figur 5: Brugeren tester oprettelse af en bruger

```java
public class UserDAOTest {
    TextUserDAO userdao = new TextUserDAO();
    ArrayList<String> arrayList = new ArrayList<>();
    UserDTO  userDTO1 = new UserDTO(
            name: "Andrey", initials: "AB", password: "0000", arrayList, id: 1, cpr: "091194-3441");

    @org.junit.Test
    public void getUser() throws DALException {
        UserDTO actual = userdao.getUser( userID: 1);
        assertEquals(userDTO1.getId(), actual.getId());
    }

    @org.junit.Test
    public void updateUser() throws DALException {
        UserDTO user = new UserDTO(
                name: "Andrey", initials: "AB", password: "0000", arrayList, id: 1, cpr: "091194-3441");
        arrayList.add("Chif");
        userdao.updateUser(user);
        System.out.println(userdao.getUser( userID: 111).getName());
        assertTrue(userDTO1.getName() == userdao.getUser( userID: 111).getName());

    }

    @org.junit.Test
    public void createUser() throws DALException {
        arrayList.add("Admin");
        userdao.createUser(userDTO1);
        assertTrue(userdao.getUser( userID: 1).getId() == userDTO1.getId());
    }
    @org.junit.Test
    public void deleteUser() throws DALException {
        userdao.deleteUser( userID: 1);
        assertEquals(null, userdao.getUser( userID: 1));
    }
```

Figur 6: UserDAOTest klasse

Billedet ovenover viser nogle af de tests vi har lavet til fil klassen. Først instantierer vi en String type arraylist, en TextUserDAO og en UserDTO objekt til testbrug.

Først bliver getUser() metoden testet ved at kalde selve metoden på et UserDTO objekt og ser om den returnerer det korrekte ID. Testen sluttede succesfuldt.

updateUser() metoden bliver testet ved at kalde metoden på et UserDTO objekt og sammenligner det opdaterede navn i filen med det navn som det helst skulle være opdateret til. Testen sluttede succesfuldt.

createUser() metoden indsætter en ”Admin”rolle i det midlertidlige test array for at skabe et test objekt, createUser() metoden bliver kaldt på UserDTO objektet, derefter bliver der checket om den skabte bruger blev gemt i tekst filen, ved at returnerer dets ID fra filen og sammenligne det med selve objektet som blev gemt i filen. Testen sluttede succesfuldt.

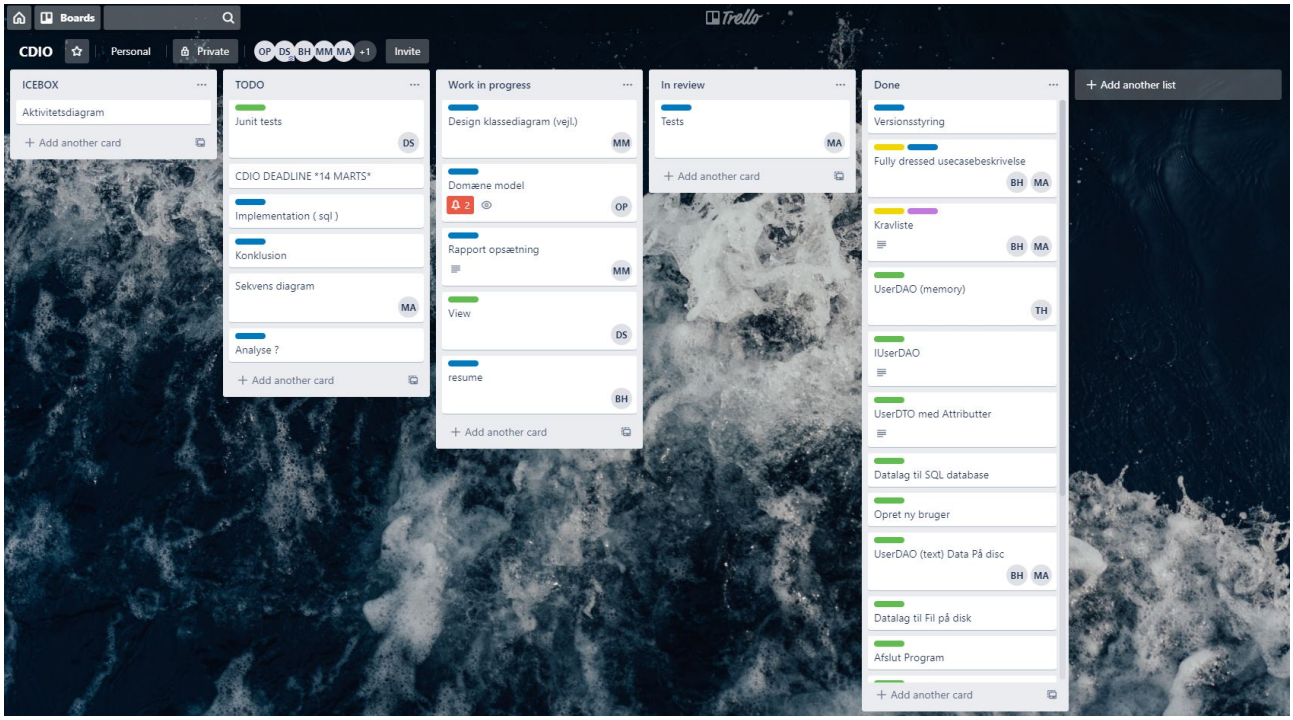
deleteUser() metoden testes ved at blive kaldt på en gemt bruger med ID = 1, efter at have gemt brugeren med createUser() metoden, der checkes derefter om brugeren med ID = 1 stadig eksisterer i filen eller ej. Testen sluttede succesfuldt.

9 Projektplanlægning

Til projektplanlægning blev der anvendt Trello Projects Boards, for nemmere at holde overblik over hvilke dele der var færdige, hvad der manglede. Hele projektet, med rapport og produkt, blev inddelt i mange mindre små dele.

På boardet blev punkterne inddelt i 5 forskellige kolonner, Icebox, TODO, Work in progjress, In Review og Done.

1. **Icebox:** Her gemmes flere funktioner som kan implementeres, hvis TODO kolonnen er tom.
2. **TODO:** Næste punkter som skal implementeres
3. **Work in progress:** Hvilke punkter som er i gang med at blive implementeret
4. **In review:** Færdige punkter, som skal godkendes inden i kommer de merge's ind i hoved udviklings branch'en.
5. **Done:** Færdig og implementerede punkter.



Figur 7: Trello board

10 Konklusion

Til dette projekt er der blevet udviklet et system til at kunne tilgå information om forskellige brugere i nogle forskellige databaser. Som udgangspunkt er systemet blevet udviklet helt efter planen og alle de krav der blev sat i begyndelsen af projektet er endt med at blive implementeret. Der er selvfølgelig altid ting der kan forbedres på i et projekt og dette er bestemt ikke nogen undtagelse. F.eks. kan der kun i øjeblikket blevet rettet en attribut af gangen. Dette er en del som kunne blive forbedret til næste gang.

Dog kan der konkluderes at projektet har været en succes og virker som planlagt. Dette er tildels pga. at der er blevet gjort godt brug af branching til at undgå fejl undervejs og brugen af 3-lagsmodellen som har gjort det nemmere at arbejde med projektet som en heldhed.

Figurer

Tabeller

11 Bilag

All diagrammer kan findes i mappen Bilag.

11.1 Bilag 1

Bilag 1 kan findes i mappen Bilag som filen ”KlasseDiagram.png”.