CDIO del 1

Projektopgave forår 2017

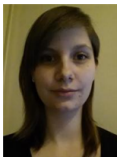
Projektnavn: CDIO 1
Gruppe: 23
Afleveringsfrist: Lørdag d. 25. februar 2017

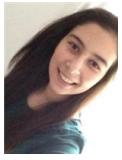
Denne rapport er afleveret via Campusnet (der skrives ikke under).
Denne rapport indeholder 13 sider ekskl. forside og bilag.


Christian Niemann, s165220


Mads Pedersen, s165204


Frederik Værnegaard, s165234


Sarina Bibæk, s154837


Iman Chelhi, s165228


Viktor Poulsen, s113403

Danmarks Tekniske Universitet
DTU Compute
02324 Videregående programmering
Undervisere: Stig Høgh & Finn Gustafsson

1 Timeregnskab

I alt per deltager	
Christian Niemann	23,5
Frederik Værnegaard	10,5
Mads Pedersen	10,5
Viktor Poulsen	10
Iman Chelhi	8
Sarina Bibæk	2,75
I alt for alle deltagere	65,25

Tabel 1: Timeregnskab

Se Bilag 8.1 for en detaljeret oversigt

Indhold

2 Indledning

Denne opgave er en del af et større projekt, som vi skal arbejde på gennem hele dette semester, og vi har derfor fra starten lagt vægt på, at det skal udformes på en sådan måde, at dele af dette program kan føres over, og blive brugt i de kommende projekter.

Vi benytter os af teknikker, som vi har lært i kurset 02324 Videregående programmering, og anvender i dette projekt interfaces med implementering af 3-lags modellen, for at gøre uddelegering af arbejdet nemmere og udførelsen af projektet mere effektivt. Ydermere giver 3-lags modellen en struktur, der er lettere at overskue, vedligeholde og genanvende.

Rapporten har til formål, at gennemgå processen af en enkelt iteration, i udviklingen af et større projekt, fra analyse af kundens ønsker til design og implementering. Vi inkluderer relevante diagrammer fra vores analyse og design, gennemgår essentielle kode stumper, og dokumenterer hvordan vi har testet programmet.

Til at opbygge en god rapport med sammenhængende dokumentation, benytter vi os af de færdigheder vi har tilegnet os fra sidste semester gennem kurserne:

- 02313 - Udviklingsmetoder til IT-systemer
- 02315 - Versionstyring og test

Rapporten er udarbejdet i ShareLatex.

3 Problemformulering

I kurset 02324 - Videregående programmering har vi fået stillet den opgave at udvikle et brugeradministrationsmodul, der har en tilhørende simpel tekstbaseret TUI brugergrænseflade.

Dette modul skal kunne udføre følgende handlinger:

- Oprette brugere
- Vise brugere
- Opdatere brugere
- Slette brugere
- Afslutte program

Derudover skal programmet kunne køre på computerne i DTU's databarer, og gemme data til næste session.

4 Analyse

4.1 Kravsspecifikation

Punkter fra FURPS+ uden krav er udeladt.

Functional

R.1 En bruger skal tildeles 1-4 rolle ved oprettelse (Admin, Farmaceut, Operatør, Værkfører)

R.2 Alle brugere skal kunne oprette brugere

R.3 Alle brugere skal kunne vise brugere i systemet

R.4 Alle brugere skal kunne redigere brugere i systemet

R.5 Alle brugere skal kunne slette brugere i systemet.

R.6 Bruger skal oprettes med fælgende information:

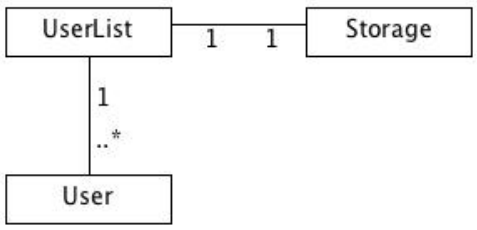
- Et unikt user-Id mellem 11-99
- Et unikt brugernavn mellem 2-20 tegn
- Et unikt initial mellem 2-4 tegn
- Et gyldigt cpr nr.
- Et ukrypteret password der overholder DTU's retningslinjer https://password.dtu.dk/
- En rolle

R.7 Brugerne skal gemmes efter hver session.

Implementation

R.8 Programmet skal kunne køres på Windows maskinerne i databarerne på DTU.

4.2 Domænemodel



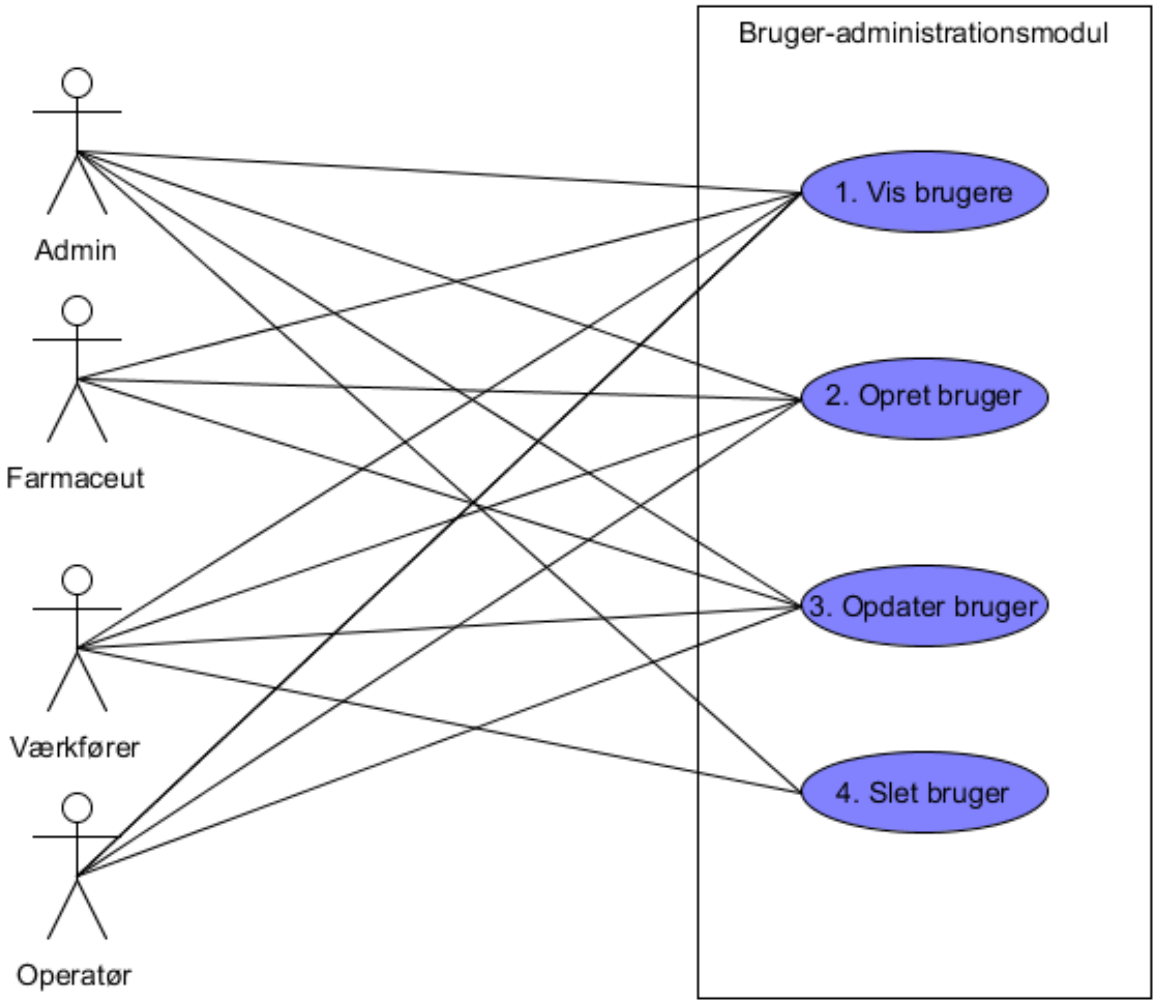
Figur 1: Domænemodel

Foroven ses en domænemodel af systemet som blev lavet i starten af processen, og som giver et overblik over objekterne i det fysiske domæne. Denne domænemodel består af tre konstruktioner som interagerer med hinanden; UserList, User og Storage.

De tre konstruktioner repræsenterer fysiske objekter, som ikke er direkte forbundet til koden.

De tre konstruktioner er knyttet på en sådan måde at UserList har nogle brugere og noget plads, og multipliciteten viser at forholdet mellem UserList og User er 1 til mange. Det er fordi UserList konstruktionen kan have flere brugere, men User konstruktionen kan kun have én UserList. Det kan så ses at forholdet mellem UserList og Storage virker på samme måde, men her er forholdet 1 til 1.

4.3 Use-case Diagram



Figur 2: Use case diagram

På ovenstående use-case diagram, ses vores use-cases. Der er i alt fire use-cases:

- Vis brugere.
- Opret bruger.
- Opdater bruger.
- Slet bruger.

Hvis vi kigger på ovenstående use-case diagram, ses det at vi har fire aktører og fire use-cases. De fire aktøre: Admin, Farmaceut, Værkfører og Operatør har forskellige rettigheder. Da vores system er så småt, vil man i dette tilfælde ikke kunne uddelegere nok til, at kunne skelne mellem aktørerne, så derfor har de alle rettighed til: Vis brugere, Opret bruger og Opdater bruger. Dette er relevant for dem alle, da f.eks en farmaceut skal kunne oprette og vise brugere i systemet og evt. opdatere brugere. Slet bruger use-casen er kun tilgængelig for Admin og Operatør, dette skyldes at vi mente, ikke alle burde kunne slette brugere i systemet.

Vi har i vorers implementering antaget at TUI'en tilgås som superadmin og har derfor ikke implementeret et decideret login og check af brugerrettigheder hertil. Dette vil blive tilføjet i senere CDIO-opgaver.

4.4 Use Case Scenarier

4.4.1 Use-Case 03: Opdater bruger

Use Case Name: Opdater bruger

ID: 3

Brief description: Der ønskes, at rette i en brugers data.

Primary Actor: Admin, Farmaceut, Operatør og Værkfører

Secondary actors: -

Preconditions:

- Programmet er åbnet.

Main Flow

- Der vælges "edit" i hovedmenuen.
- Der vælges en bruger vha. brugerens ID.
- Der vælges hvilken data der ønskes rettet i.
- Den nye værdi indskrives.
- Brugeren opdateres efter godkendelsesbesked er udført.

Post conditions:

- En brugers data er blevet opdateret med de oplysninger.

Alternative flow:

- Der vælges cancel, til at bryde ud af "edit"og komme tilbage til hovedmenuen.
- Bruger returneres til hovedemenu.

Ovenstående use-case beskriver et scenarie, hvor en aktør ønsker, at opdatere i en brugers data. Der er fire primære aktørere i denne usecase. Det er relevant for dem alle, at skulle rette i en brugers data, så derfor har de alle rettigheder til dette. Vi har i dette tilfælde kun et alternative flow. Hvis man ønsker, at komme tilbage til hovedmenuen, kan man på ethvert tidspunkt skrive "cancel".

4.5 Requirement Tracing

Da krav R.8 er et implementeringskrav kan det ikke dækkes af usecases for systemet. Vi ser dog at alle funktionelle krav er dækket af en eller flere usecases, hvilket fortæller os at ingen ligegyldige krav eksisterer.

	Vis Brugere	Opret Bruger	Opdater Bruger	Slet Bruger
R.1				
R.2				
R.3				
R.4				
R.5				
R.6				
R.7				
R.8				

Tabel 2: Requirement Tracing

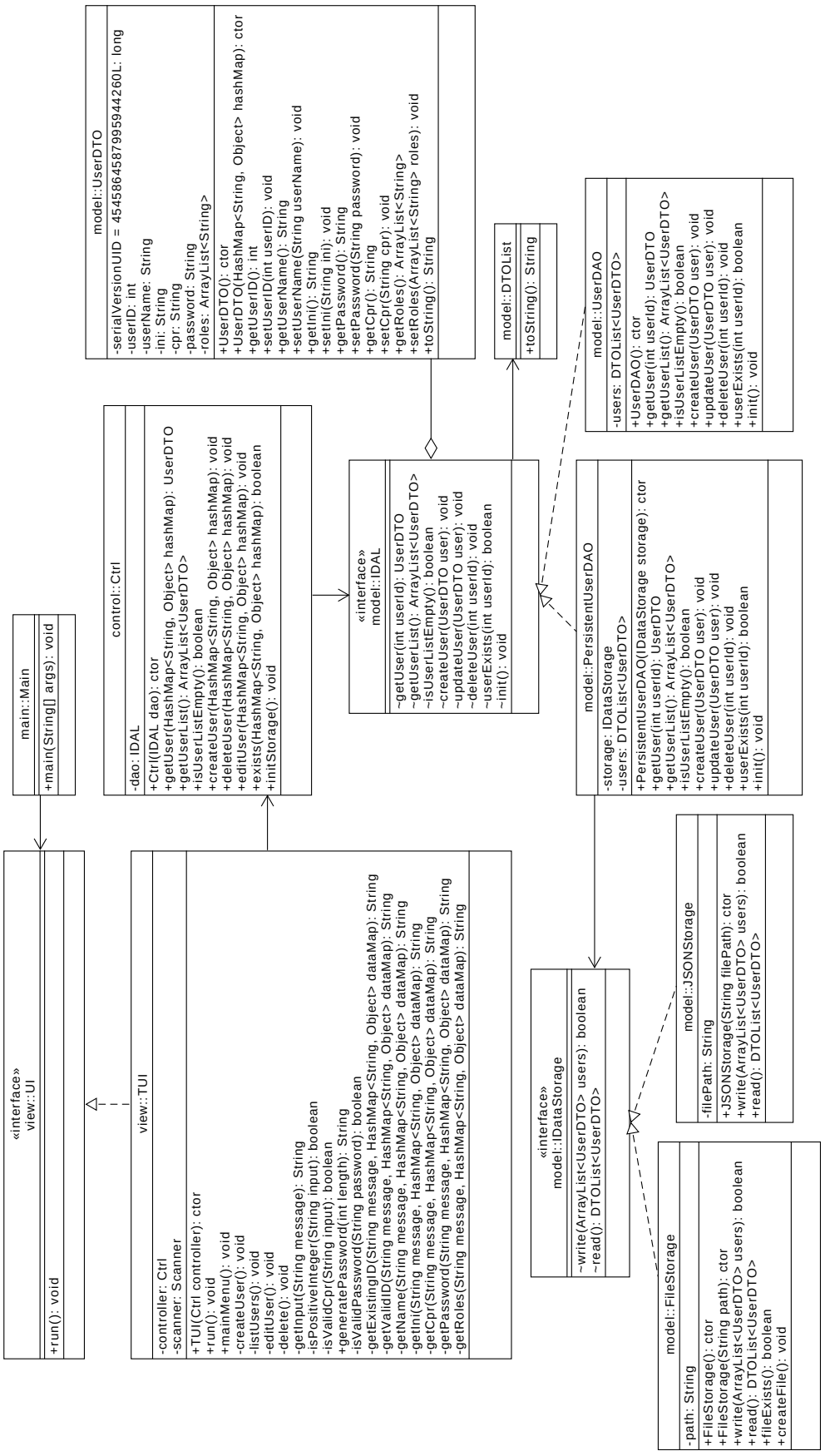
5 Design

5.1 Design Klasse Diagram

På Figur 3 ses vores klassediagram, der viser strukturen af vores system. Vores system tager udgangspunkt i tre lags modellen, der vises udfra opdelingen og navngivningen af vores pakker - model, view, control. Det yderste lag, view, består af vores user interface (UI-klassen) samt den implementerende TUI-klasse. Det midterste lag, control, består blot af klassen Ctrl, mens resten af klasserne ligger i model-laget. Da vi gør brug af tre lags modellen kalder vi kun udefra og ind.

Al interaktion med brugeren foregår selvfølgelig gennem vores text user interface, som styrer det overordnede flow, samt snakker med controlleren, der kan tilgå vores data. Når vores TUI-klasse modtager input fra brugeren oprettes det i et HashMap, der sendes videre til controlleren, som dernæst omskriver det til et UserDTO objekt og sender objektet videre til datalaget.

I vores data access layer gør vi brug af klassen DTOList - som extender ArrayList - da vi ønsker en brugerdefineret toString()-metode til udprint af vores brugerinfo. IDAL interfacet kan gøre brug af en af de tre forskellige måder at implementere UserDAO. Enten en af de to persistente DataStorage-klasser, FileStorage eller JSONStorage, der gør det muligt at gemme i en '.txt' eller en '.json' fil - eller blot uden persistent data, der ikke gemmer information mellem sessioner.



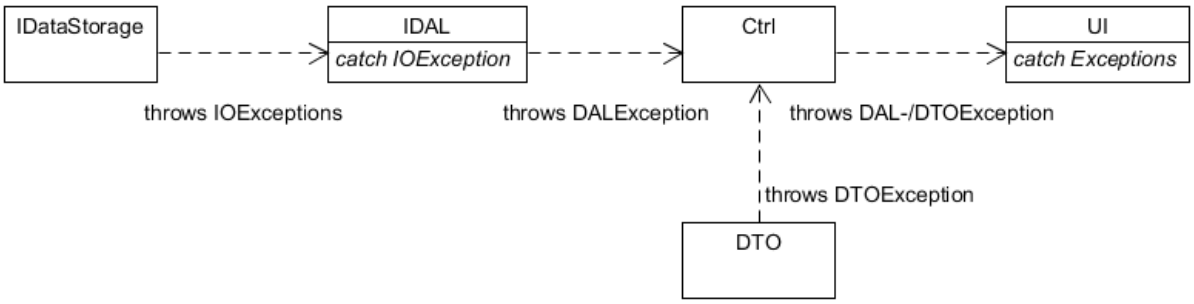
Figur 3: Design Klasse Diagram

6 Implementering

6.1 Kildekode

I dette afsnit vil vi fremhæve interessante dele af projektets kildekode, for at vise brugen af de koncepter kurset har undervist i frem til nu, herunder (men ikke begrænset til) exception handling og Collections.

6.1.1 Exception Handling



Figur 4: Exception handling design

Da vi i vores system både læser fra og skriver til en fil, der gør at vi kan opretholde information mellem sessioner, er vi nødt til at håndtere exceptions der kan opstå i denne sammenhæng. Vi har derfor valgt at designe vores håndtering af exceptions således at alle exceptions, som opstår vil blive kastet mod det yderste lag (til vores user interface), hvor der besluttes, hvad der skal ske. Eksempel på dette kan ses på Figur 4.

I FileStorage-klassen kaster vi derfor IOExceptions fra de metoder der tilgår vores fil, heraf write(), read() og createFile(). Alle IOExceptions kastes op til vores data access object (PersistentUserDAO-klassen), som fanger de kastede exceptions fra vores file storage. Når exceptions fanges i UserDAO kaster vi dem da videre til vores controller som DALExceptions (eksempel på dette kan ses i nedenstående stykke kode), der er en brugerdefineret exception klasse. Controlleren modtager derfor kun DALExceptions fra UserDAO og kaster blot disse videre som DALExceptions til vores user interface, hvor vi vælger at fange og håndtere vores DALExceptions.

Derudover kan vores UserDTO-klasse kaste en DTOException - der også er en brugerdefineret exception - fra konstruktøren, idet der ikke gives tilstrækkelige oplysninger til at kunne oprette et sådan objekt. Denne exception kastes kun når en bruger skal oprettes i systemet og bliver derfor kastet fra UserDTO op til controlleren - og derfra op til vores user interface, hvor alle exceptions håndteres.

```
public void updateUser(UserDTO user) throws DALException {
        deleteUser(user.getUserID());
        createUser(user);

        try {
                storage.write(users);
        }catch (IOException e){
                throw new DALException("IOException", e);
        }
}
```

Ovenstående stykke kode viser metoden updateUser(UserDTO user) fra PersistentUserDAO-klassen. Metoden gør netop brug af vores write() metode, der kan kaste en IOException. Opstår sådan en situation vil IOException blive fanget og blot kastet videre som en DALException.

6.1.2 UserDTO Constructor

I vores implementation har vi valgt at anvende HashMaps til at sende bruger-objekter mellem systemets forskellige klasser. Dette er i sidste ende bundet sammen i UserDTO, som repræsenterer selve brugerobjektet vha. dennes konstruktør. Når et nyt brugerobjetkt forsøges instantieret sendes et HashMap til konstruktøren, som derefter tjekkes for påkrævede felter (og kaster exceptions ved mangler), og bruger klassens getter- og setter-metoder til at sætte dataen på objektet. HashMap'et har samme struktur som vores klasse og er på den måde en repræsentation af selve UserDTO-klassen.

```
public UserDTO(HashMap<String, Object> hashMap) throws DTOException {
        if(hashMap.containsKey("ID")){
            this.userID = (int) hashMap.get("ID");
        }else {
            throw new DTOException("No ID Provided!");
        }
        if (hashMap.containsKey("userName")) {
            this.userName = hashMap.get("userName").toString();
        } else {
            throw new DTOException("No user name provided");
        }
        if (hashMap.containsKey("ini")) {
            this.setIni(hashMap.get("ini").toString());
        } else {
            throw new DTOException("No initials provided");
        }
        if (hashMap.containsKey("cpr")) {
            this.setCpr(hashMap.get("cpr").toString());
        } else {
            throw new DTOException("No CPR provided");
        }
        if (hashMap.containsKey("password")) {
            this.setPassword(hashMap.get("password").toString());
        } else {
            throw new DTOException("No password provided");
        }
        if (hashMap.containsKey("roles")) {
            this.roles = (ArrayList<String>) hashMap.get("roles");
        } else {
            throw new DTOException("No role provided");
        }
    }
```

6.1.3 Validation

Til validering af vores input, har vi lavet en separat klasse med statiske metoder, så de kan tilgås fra alle steder ligesom Math biblioteket. Dette har vi valgt fordi vi med dette valideringsbibliotek, kan bruge de samme metoder, til at validere input i flere forskellige user interfaces. Lige nu har vi kun en TUI, men når vi senere laver en GUI kan vi bruge de samme metoder til at validere input i den.

Herunder ses et eksempel på en af vores valideringsmetoder; den modtager en string, som den skal validere om er et ægte cpr-nummer. Vi har valgt at behandle cpr-numre som strings, for ikke at få overflow på en integer hvis folk har fødselsdag sidst på måneden. Til at starte med tjekker vi, om de første 6 cifre danner en gyldig dato fra år 1900 og frem til dags dato. Dette sætter den begrænsning, at det ikke er sikkert at ens cpr-nummer bliver valideret hvis man er over 117 år, men det har vi valgt at acceptere som begrænsning for vores program. Såfremt det er en gyldig dato, bliver det derefter valideret om alle de 10 cifre til sammen danner et gyldigt cpr-nummer. Dette gøres ved at gange hvert ciffer med en bestemt værdi, og summen af alle de produkter skal være deleligt med 11.

```
public static boolean isValidCpr(String cpr) {
        if (isPositiveInteger(cpr)) {
                int month = Integer.parseInt(cpr.substring(2, 4));
                // Checks if month is valid
                if (month > 0 && month < 13) {
                        for (int i = 1900; i < 2100; i += 100) {
                                int day = Integer.parseInt(cpr.substring(0, 2));
                                int year = i + Integer.parseInt(cpr.substring(4, 6));
                                // Creates a calendar object and sets year and month
                                Calendar cprDate = new GregorianCalendar(year, month-1, 1);
                                // Get the number of days in that month
                                int daysInMonth = cprDate.getActualMaximum(Calendar.DAY_OF_MONTH);
                                // Checks if day is valid
                                if (day > 0 && day <= daysInMonth) {
                                        // check if the date later than todays date
                                        cprDate.set(year, month-1, day);
                                        Calendar currentDate = new GregorianCalendar();
                                        if (cprDate.compareTo(currentDate) <= 0) {
                                                // check if full cpr is valid
                                                int CprProductSum = 0;
                                                int[] multiplyBy = {4, 3, 2, 7, 6, 5, 4, 3, 2, 1};
                                                for (int j = 0; j < cpr.length(); j++) {
                                                        CprProductSum +=
                                                             Integer.parseInt(cpr.substring(j, j+1)) *
                                                             multiplyBy[j];
                                                }
                                                if (CprProductSum % 11 == 0) {
                                                        return true;
                                                }
                                        }
                                }
                        }
                }
        }
        return false;
}
```

6.2 Argumentation for test

Til at teste vores program, ville vi rigtig gerne have lavet en integrationstest, der bestod af et sæt unit tests, der testede alle systemets 3 lag samtidigt. Dette ville vi gøre ved at ændre på output stream destinationen, så vores test kunne tale med TUI'en gennem kommandoer, ligesom hvis en bruger sad og skrev kommandoer. Dette har vi imidlertid ikke kunnet få til at fungere, vi ved ej heller, om det overhovedet kan lade sig gøre i praksis, men vi arbejder videre på sagen, hen mod senere CDIO iterationer på dette projekt. Til denne afleveringsfrist har vi måttet sænke forventningerne lidt, og gå uden om vores user interface, så vi kun tester controller og datalaget, som er de 2 inderste lag, samt valideringsmetoderne.

Vi opnår stadig det smarte i, at disse tests også kan bruges, efter man har implementeret nye lagringsmetoder, så kan man bare ændre lagringstypen i setup, og køre testen, for at se om den nye lagringsmetode fungerer. Det er det store fokus i vores test af dette program, at vi har en test, der fremadrettet kan bruges til test-driven developement. Vi får med disse tests ikke testet vores TUI, men det håber vi at få inkluderet i disse tests senere.

7 Konklusion

Under udviklingsprocessen af dette projekt, har vi ud fra opgavebeskrivelsen, arbejdet på et bruger-administrationsmodul, som skal kunne bruges, som del af et større projekt. Vi har vha. af viden fra kurserne 02313 Udviklingsmetoder til IT-systemer og 02315 Versionstyring og Test, designet et system samt dokumenteret processen. Samtidig har vi brugt vi færdigheder fra 02324 videregående programmering til, at implementere vores design.

Et vigtigt krav til dette system, har været anvendelsen af 3-lags modellen med interfaces. Det yderste lag, View, består af user interface, som er det brugeren ser. I denne iteration er vores user interface implementeret som en TUI. Det midterste lag, control består af én klasse, som håndterer udvekslingen mellem de to andre lag, i denne iteration er der ikke særlig meget logik i controller klassen, det kommer først når man skal til at kunne logge ind. Sidst har vi model-laget, som er det nederste lag, der opbevarer og håndterer data. Vi har ud fra denne model, gjort vores system mere overskueligt, da de tre lag har specifikt begrænsede ansvarsområder og der kun bliver kaldt udefra og ind.

Vi startede med, ud fra kundens vision, at opstillet en række krav i forhold til FURPS+. Derefter udarbejdede vi en domænemodel, for at skabe overblik over, hvilke entiteter vi skulle bruge i vores model-lag. Dernæst analyserede vi, ud fra kundes ønskede use-cases, hvordan flow'et i systemet skulle være. Sidst i analyse processen, har vi sporet vores krav i forhold til use-cases, for at sikre, at det hele var dækket tilstrækkeligt. Designet af projektet, har bestået af et design klasse diagram. Vi vi mente at dette var tilstrækkeligt, da vi gennem disse processer opnåede nok indsigt i problemstillingen, ti lat implementere vores løsning. Til slut har vi testet vores to inderste lag med en integrationstest, der godkender at vores program opfylder kundens krav.

8 Bilag

8.1 Tidsregistrering

Dato	Deltager	Design	Impl.	Test	Dok.	Andet	I alt
12/02/2017	Christian Niemann		4				4,0
13/02/2017	Christian Niemann		2				2,0
14/02/2017	Christian Niemann	0,5	2				2,5
15/02/2017	Christian Niemann		0,5	1,5			2,0
16/02/2017	Christian Niemann		1,5				1,5
19/02/2017	Christian Niemann	0,5	1,5	3			5,0
21/02/2017	Christian Niemann		1,5	2	1,5		5,0
22/02/2017	Christian Niemann			0,5			0,5
23/02/2017	Christian Niemann				1		1,0
13/02/2017	Frederik Værnegaard		2				2,0
14/02/2017	Frederik Værnegaard		2				2,0
17/02/2017	Frederik Værnegaard		0,5				0,5
19/02/2017	Frederik Værnegaard		0,5	0,5	1		2,0
20/02/2017	Frederik Værnegaard		0,5	1	1		2,5
21/02/2017	Frederik Værnegaard				1,5		1,5
12/02/2017	Iman Chelhi		4				4,0
18/02/2017	Iman Chelhi				1		1,0
21/02/2017	Iman Chelhi	3					3,0
12/02/2017	Mads Pedersen				2	1	3,0
13/02/2017	Mads Pedersen		1				1,0
17/02/2017	Mads Pedersen				2		2,0
20/02/2017	Mads Pedersen				2		2,0
22/02/2017	Mads Pedersen				1		1,0
23/02/2017	Mads Pedersen					1,5	1,5
12/02/2017	Sarina Bibæk		0,5				0,5
18/02/2017	Sarina Bibæk				1		1,0
19/02/2017	Sarina Bibæk				1,25		1,3
13/02/2017	Viktor Poulsen		2				2,0
19/02/2017	Viktor Poulsen		2				2,0
20/02/2017	Viktor Poulsen				2		2,0
21/02/2017	Viktor Poulsen		2		1		3,0
22/02/2017	Viktor Poulsen				1		1,0

Tabel 3: Detaljeret Timeregnskab

8.2 Kildekode

Al kildekode samt udviklingshistorik kan gennemgås på https://github.com/DTU23/CDIO1