CDIO 1
02324 - Videregående Programmering

Gruppe Nr.: 25
Afleveringsfrist: Lørdag d. 25/02-2017 kl. 05:00

Denne rapport afleveres via CampusNet (der skrives ikke under).
Denne rapport indeholder 25 sider inkl. denne side og bilag

		
s165225 Al-Alak, Mehdi A.	s165221 Jensen, Lasse A.	s165206 Soelberg, Andreas

	
s165223 Jørgensen, Jimmy	s165246 Muslu, Yanki

Timeregnskab

	Analyse	Design	Impl	Test	Doku	Total
Andreas	2	2	3	2	6	15
Jimmy	0,5	0,5	13	0,5	2	16,5
Lasse	0,5	0.5	11	1	3	16
Mehdi	2,5	0,5	3		2	8
Yanki	2,5	2,5	10		5	20
Total	1,5	0,5	15,5	0	0	**17,5**

Indholdsfortegnelse

Resume

Denne rapport omhandler en gennemgang af udarbejdelsen af et bruger-administrationsmodul. Rapporten starter med en række krav for hvad programmet skal indeholde og hvordan det skal fungere, efterfulgt af et kort afsnit med konfigurationsstyring. Derefter vil vi komme ind på analyse og design, hvor der vil forklares om forskellige modeller vi har lavet for at få et overblik over hvordan programmet skal bygges op. Det næste afsnit omhandler implementationen af programmet, som dette dækker over hvordan koden er blevet skrevet. Til sidst konklusion, der opsummerer over resultatet for dette projekt. Formålet med denne rapport er at give en indsigt over hvordan vi har behandlet denne opgave, ud fra den givet opgavebeskrivelse.

Indledning

I dette projekt har vi fået til opgave at implementere et bruger-administrationsmodul. Vi har valgt at fremstille vores bruger-administrationsmodul i Java og ud fra opgavebeskrivelsen har vi opstillet en række krav for at specificere indholdet af programmet. Vi har her brugt en tekstbaseret brugergrænse overflade, hvor vi starter med en start menu, her skal man kunne vælge imellem 4 forskellige funktioner. Den første funktion er at oprette en bruger. Når man vælger denne funktion, skal man kunne indtaste det der kræves, og man skal modtage et ID og en kode. Desuden skal brugeren gemmes. Den anden funktion skal hente en liste med alle oprettede brugere. Den tredje funktion opdaterer en bruger. Man har her muligheden for at hente en brugers information og ændre den. Via den sidste funktion i menuen skal man have muligheden for at slette en bruger. Et vigtigt krav til dette projekt er at man skal kunne gemme alt information i en database eller på en fil. Målet for projektet er at implementere et program der lever op til alle kravspecifikationerne, der er blevet stillet, uden problemer.

Kravspecifikationer

Functionality

F1. userID skal være i et interval mellem 11-99.
F2. En bruger vil få tildelt et userID ved oprettelse.
F3. Brugernavnet skal indeholde 2-20 tegn.
F4. Initialer skal indeholde 2-4 tegn.
F5. Password skal være ukrypteret, og skal have samme krav som et som et DTU password. dvs. Adgangskoden skal indeholde mindst 6 tegn af mindst tre af de følgende fire kategorier
    - Små bogstaver: a-z
    - Store bogstaver: A-Z
    - Tal: 0-9
    - Specielle tegn: {'.', '-', '_', '+', '!', '?', '='}
F6. En bruger skal få tildelt en password ved oprettelse.
F7. En bruger skal have mulighed for at udfylde en eller flere af følgende roller:
    - Admin
    - Pharmacist
    - Foreman
    - Operator.
F8. En bruger skal ikke have mulighed for at være både Pharmacist og Foreman.

Usability

F9. Programmet skal have en start menu, hvor man kan vælge mellem funktioner
F10. Man skal kunne oprette en bruger fra start menuen
F11. Man skal kunne slette en bruger fra start menuen
F12. Man skal kunne få en liste af alle brugere fra start menuen.
F13. Man skal kunne ændre en bruger fra start menuen.
F14. Input skal foregå via TUI'en

Reliability

NF1. Programmet skal kunne oprette, slette, hente, gemme og ændre brugerinfo uden problemer.

Supportability

NF2. Programmets Strings skal kunne findes et samlet sted.

Implementation constraints

NF3. Programmet skal skrives i Java.
NF4. 3-lags modellen med interfaces skal anvendes.

Konfigurationsstyring

System specifikationer

Windows

- Version 7 eller nyere.

Krav for database

- Hav mySQL version 5.5.54 winx64.
- Opdateret java version.

Kørselsguide for mySQL 5.5.54 winx64

1. Gå til lokationen for mysql-5.5.54-winx64\bin i kommandoprompt.
2. Kør kommandoen ”mysqld -u root”
3. Åben et nyt kommandoprompt vindue og gå til ovenstående sti igen.
4. Kør kommandoen ”mysql -u root”
5. Kør kommandoen ”create database 25cdio01”
6. Kør programmet via main.
7. Programmet skal printe ”Connected.” hvis programmet kan tilgå databasen.
    - Ønskes en anden URL, username og/eller kodeord kan dette tilgås i klassen DatabaseSaver i linje 225.

Vejledning til import af Git-repository i Eclipse

1. Åben din browser
2. Gå til denne webside https://github.com/Gruppe25DTU/CDIO1
3. Kopier klonings URL til højre (https://github.com/Gruppe25DTU/CDIO1.git)
4. Åben ”Eclipse”
5. File -> Import
6. Tryk next
7. Git -> Projects from Git
8. Tryk next
9. Close URl
10. Tryk next
11. Sæt den kopierede URl i feltet
12. Tryk next
13. Tryk next
14. Vælg din sti til at være dit workspace
15. Tryk next
16. Tryk finish
17. Åben projektet i Eclipse
18. Åben pakken kaldet ”main”
19. Åben klassen kaldet ”Main”
20. Tryk ”Run” (Den grønne pil i toppen)
21. Programmet starter nu i konsollen.

Analyse

Use case beskrivelser

Følgende use case beskrivelser skitserer vores implementering af de use cases, som var beskrevet i det udleverede materiale. Med disse beskrivelser har gruppen en fælles forståelse for hvilke input der forventes af brugeren for at nå et mål, samt en sammenfatning af de skridt systemet tager i denne proces. Det kan derefter diskuteres hvordan systemet skal udføre disse skridt, og om det ville være logisk at gruppere eller separere nogle af disse handlinger på baggrund af GRASP princippet, som er beskrevet senere i denne rapport.

Use Case Name:	**Create User**
ID:	**1**
Actors	Admin
Pre Condition	Actor is in the TUI (Main) Menu Actor is prompted for command
Post Condition	User with provided information now exists in the system
Flow:	1. Actor types Create User-command (“1”) 2. System enters the “Create User” menu 3. System creates new User 4. System generates ID and password 5. System prompts for username, initials, cpr, roles, listing requirements for each 6. System displays the User and asks for confirmation 7. Actor confirms details 8. System saves User
Alternate Flow:	• **[Entered information did not meet requirements]** a. System lists requirements for that field b. System prompts for new input c. **(Return to Flow.5)**
Exceptional Flow:	• **[User quits during creation process]** a. System goes back to the previous menu b. New User is not saved • **[Actor does not confirm]** a. System goes back to the previous menu b. New User is not saved

Use Case Name:	**List Users**
ID:	**2**

Actors	Admin
Pre Condition	Actor is prompted for command
Post Condition	Actor is presented with list of Users
Flow:	1. Actor types List Users-commands (“2”) 2. System lists all existing Users
Alternate Flow:	• **[No user exists]** a. System informs Actor no User exists
Exceptional Flow:	

Use Case Name:	**Update User**
ID:	**3**
Actors	Admin
Pre Condition	Actor is in the TUI (Main) Menu Actor is prompted for command
Post Condition	Information of User with provided ID has changed to provided information
Flow:	1. Actor types Update User-command (“3”) 2. System prompts for ID 3. System creates a temporary User to hold new information 4. System lists mutable information and command to change each 5. System displays Save Changes-command 6. Actor types command to change any information 7. System prompts for new information, listing relevant requirements 8. Actor enters new information 9. Return to Flow.4
Alternate Flow:	• **[Entered information did not meet requirements]** a. System lists requirements for that field b. System prompts for new input c. Actor enters new information d. **(Return to Flow.4)** • **[Actor types Save Changes-command (“6”)]** a. System displays the temporary User b. System asks for confirmation c. Actor confirms details d. System overwrites original User with temporary User
Exceptional Flow:	• **[User quits during update process]** a. System goes back to the previous menu b. Original User unchanged

	• **[Actor does not confirm]** a. System goes back to the previous menu b. Original User unchanged

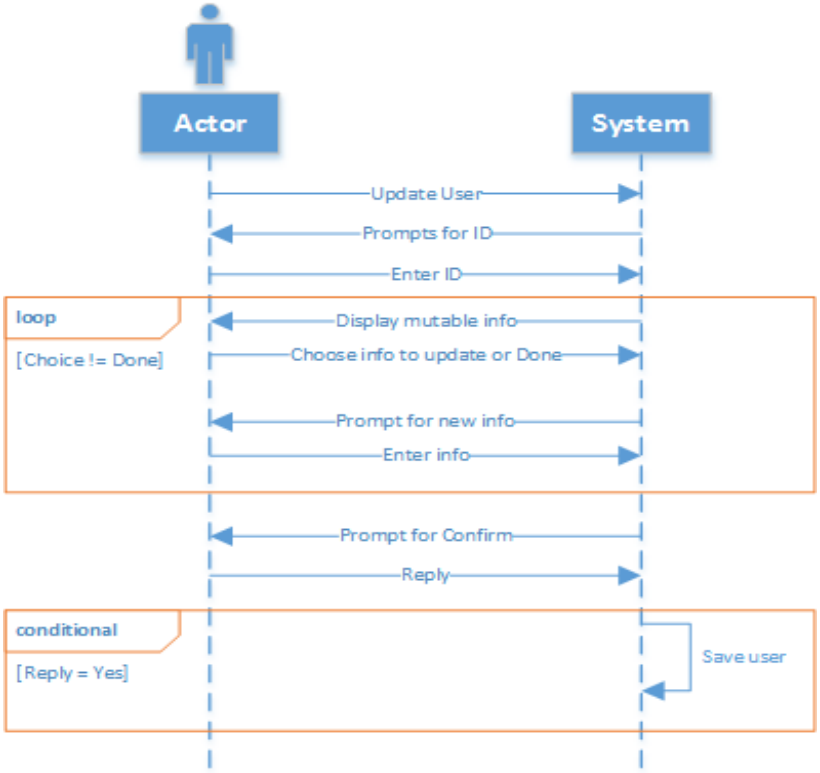
Use Case Name:	Delete User
ID:	4
Actors	Admin
Pre Condition	Actor is in the TUI (Main) Menu Actor is prompted for command
Post Condition	User with provided ID no longer exists
Flow:	1. Actor types Delete-command (“4”) 2. System prompts for ID 3. System display selected User and asks for confirmation 4. Actor confirms 5. System deletes User with provided ID
Alternate Flow:	• **[ID does not exist]** a. System informs Actor no User with provided ID exists b. System informs Actor of ‘list’ command c. System prompts for new ID d. System deletes User with provided ID e. System informs Actor that User is deleted
Exceptional Flow:	• **[User quits during creation process]** a. System informs Actor no User was deleted • **[Actor does not confirm]** a. System goes back to the previous menu b. User is left unchanged

Use Case Name:	Exit program
ID:	5
Actors	Admin

Pre Condition	Actor is prompted for command
Post Condition	Program closed successfully
Flow:	1. Actor types Quit-command (quit()) 2. Program exits
Alternate Flow:	• **[Actor was not in the main menu]** a. System goes back to the previous menu
Exceptional Flow:	

System sekvensdiagrammer

System sekvensdiagrammer skitsere, på samme måde som use case beskrivelser, hvordan et system understøtter en given use case. Da dette blot er en grafisk repræsentation af use case beskrivelser, valgte gruppen kun at inkludere ét diagram i rapporten. Diagrammerne for de resterende use case beskrivelser kan findes i bilag. Klik på mig for at gå til diagrammer.

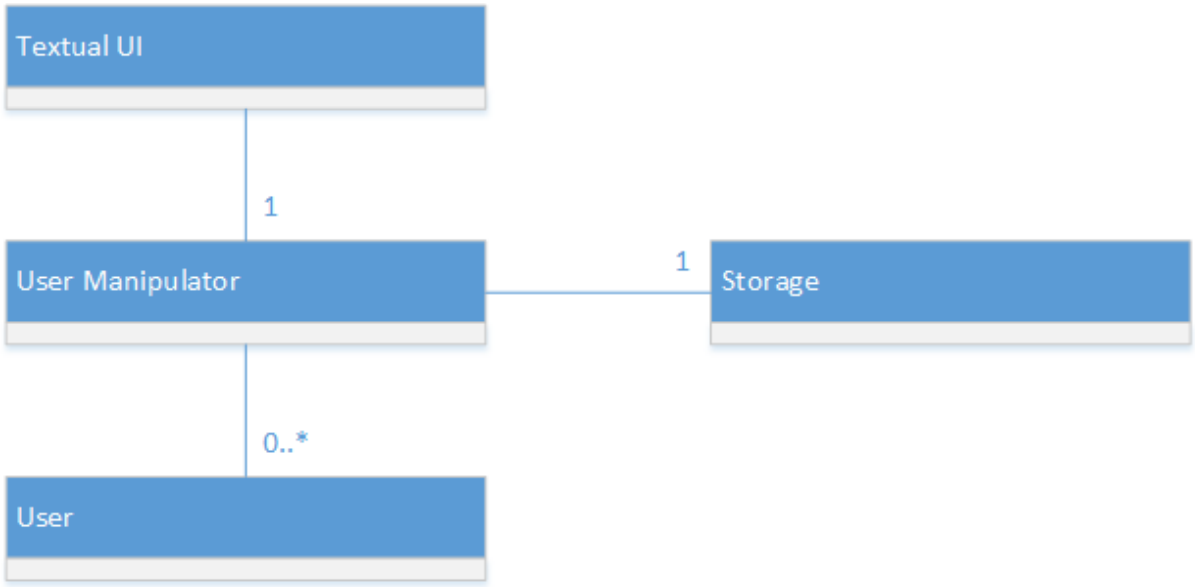


Figur 1

Som det kan ses på Figur 1, skal man kunne opdatere en bruger i systemet, ved at opdatere de tilgængelige informationer enkeltvis i en vilkårlig rækkefølge, og til hver en tid stoppe ved at angive at man er færdig med at opdatere.

Domænemodel

Målet med en domænemodel er at skabe et groft overblik over de elementer, som skal indgå i et projekt for at imødekomme diverse use cases. Følgende diagram giver ingen detaljer vedrørende implementationen af disse elementer, og det er ofte nødvendigt at dele elementerne op i underdele, dog burde den overordnede struktur forblive uændret. Domænemodellen for dette system er meget simpelt, da systemet ikke er særlig komplekst, og indeholder kun fire elementer; en brugergrænseflade, en måde at manipulere data på, en måde at gemme data på, samt et objekt til at repræsentere en bruger.



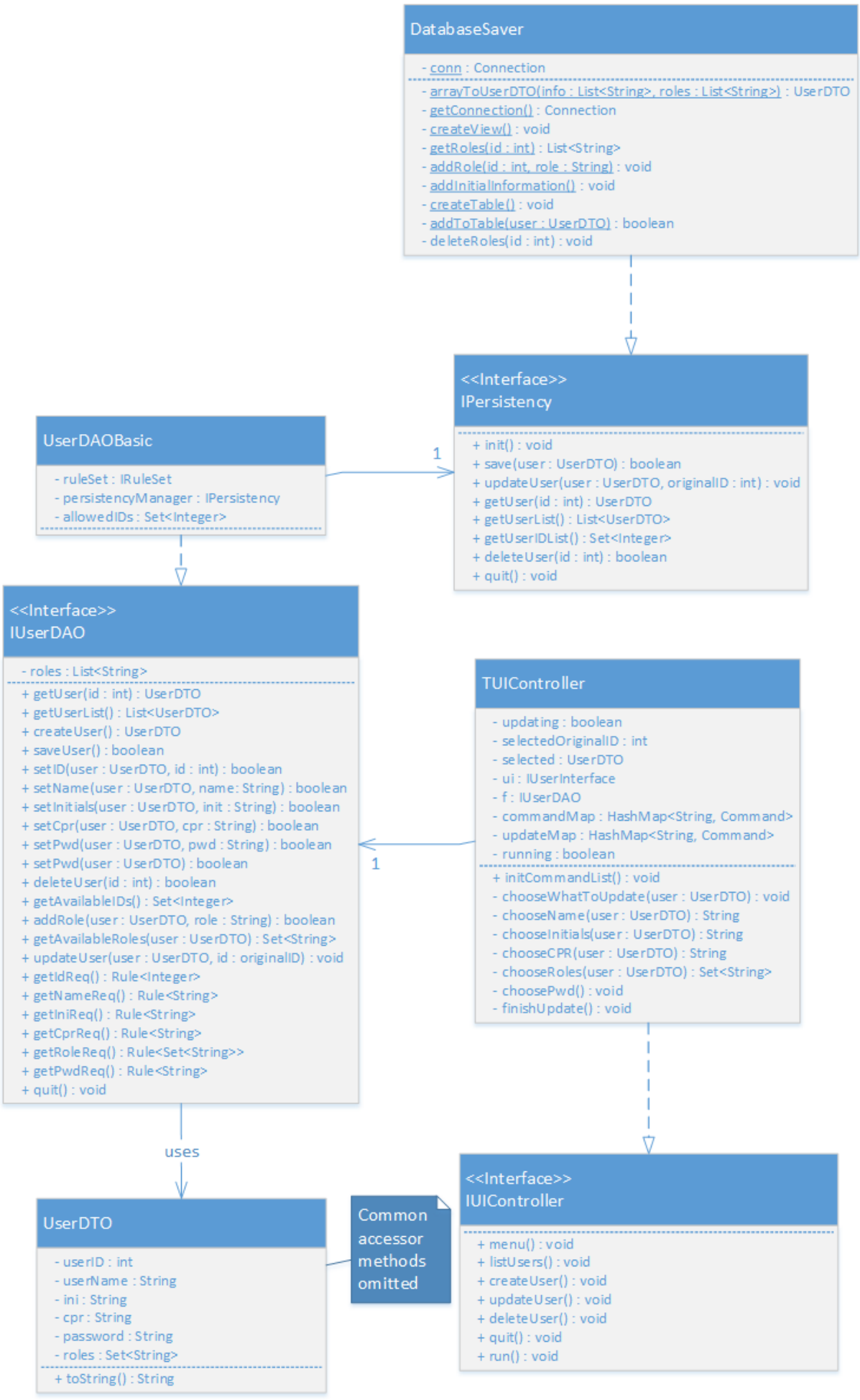
Figur 2

Design

Klassediagram

På baggrund af analysefasen, er det nu muligt at lave et klassediagram. I modsætning til domænemodellen, indeholder klassediagrammet ikke blot et groft overblik, men illustrerer derimod alle klasser som skal implementeres, hvilke metoder og variabler disse indeholder, samt forholdet mellem diverse klasser, som muliggør det at imødekomme alle påkrævede use cases og samtidig overholde den aftalte arkitektur. Arkitekturen bliver beskrevet i næste kapitel.

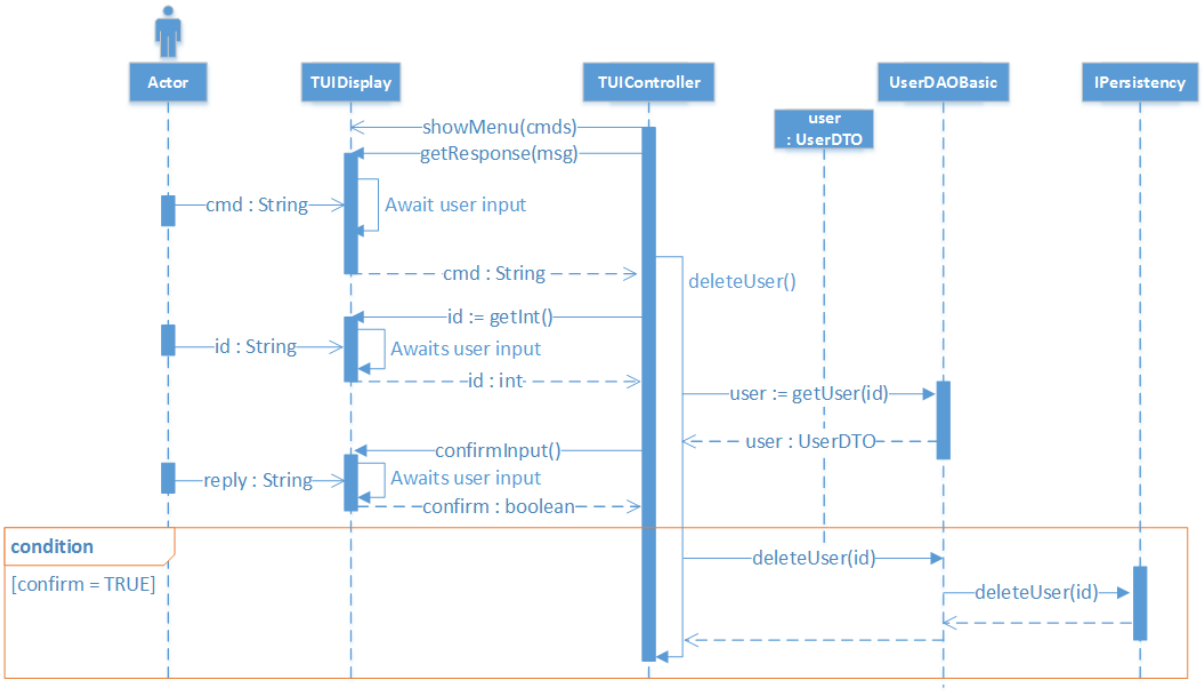
For at spare på plads, viser Figur 3 kun dele af klassediagrammet. Det fulde klassediagram kan findes i bilag (Klik på mig for at gå til bilag) Bemærk at IPersistency anvendes af den specifikke implementation af IUserDAO og ikke af interfacet selv. Dette tillader at andre (fremtidige) implementationer af IUserDAO kan opnå persistens uden at anvende vores persistens-klasser. For brugergrænsefladen vil dette ikke ændre noget, da dette lag stadig blot kalder IUserDAO.saveUser() for at gemme brugere, uanset om der anvendes IPersistency eller andet for at opnå persistens.


DatabaseSaver
- conn : Connection
- arrayToUserDTO(info : List<String>, roles : List<String>) : UserDTO
- getConnection() : Connection
- createView() : void
- getRoles(id : int) : List<String>
- addRole(id : int, role : String) : void
- addInitialInformation() : void
- createTable() : void
- addToTable(user : UserDTO) : boolean
- deleteRoles(id : int) : void
<<Interface>>
IPersistency
+ init() : void
+ save(user : UserDTO) : boolean
+ updateUser(user : UserDTO, originalID : int) : void
+ getUser(id : int) : UserDTO
+ getUserList() : List<UserDTO>
+ getUserIDList() : Set<Integer>
+ deleteUser(id : int) : boolean
+ quit() : void
UserDAOBasic
- ruleSet : IRuleSet
- persistencyManager : IPersistency
- allowedIDs : Set<Integer>
1
<<Interface>>
IUserDAO
- roles : List<String>
+ getUser(id : int) : UserDTO
+ getUserList() : List<UserDTO>
+ createUser() : UserDTO
+ saveUser() : boolean
+ setID(user : UserDTO, id : int) : boolean
+ setName(user : UserDTO, name: String) : boolean
+ setInitials(user : UserDTO, init : String) : boolean
+ setCpr(user : UserDTO, cpr : String) : boolean
+ setPwd(user : UserDTO, pwd : String) : boolean
+ setPwd(user : UserDTO) : boolean
+ deleteUser(id : int) : boolean
+ getAvailableIDs() : Set<Integer>
+ addRole(user : UserDTO, role : String) : boolean
+ getAvailableRoles(user : UserDTO) : Set<String>
+ updateUser(user : UserDTO, id : originalID) : void
+ getIdReq() : Rule<Integer>
+ getNameReq() : Rule<String>
+ getIniReq() : Rule<String>
+ getCprReq() : Rule<String>
+ getRoleReq() : Rule<Set<String>>
+ getPwdReq() : Rule<String>
+ quit() : void
1
TUIController
- updating : boolean
- selectedOriginalID : int
- selected : UserDTO
- ui : IUserInterface
- f : IUserDAO
- commandMap : HashMap<String, Command>
- updateMap : HashMap<String, Command>
- running : boolean
+ initCommandList() : void
- chooseWhatToUpdate(user : UserDTO) : void
- chooseName(user : UserDTO) : String
- chooseInitials(user : UserDTO) : String
- chooseCPR(user : UserDTO) : String
- chooseRoles(user : UserDTO) : Set<String>
- choosePwd() : void
- finishUpdate() : void
uses
UserDTO
- userID : int
- userName : String
- ini : String
- cpr : String
- password : String
- roles : Set<String>
+ toString() : String
Common accessor methods omitted
<<Interface>>
IUIController
+ menu() : void
+ listUsers() : void
+ createUser() : void
+ updateUser() : void
+ deleteUser() : void
+ quit() : void
+ run() : void

Figur 3

Design sekvensdiagram

Hvor et systemsekvensdiagram undlader alle detaljer vedrørende systemet, og blot illustrerer den simple interaktion mellem brugeren og systemet, viser design sekvensdiagrammer yderligere alle de elementer af systemet som anvendes for at imødekomme en use case, hvilke skridt de udfører samt hvordan disse elementer kommunikerer med hinanden og med brugeren. Figur 4 er et design sekvensdiagram for use case 4: "Delete User".



Figur 4

Det kan på figuren ses, at der er dele af sekvensen som kun udføres i visse tilfælde, specifikt, hvis det bliver bekræftet at den valgte bruger skal slettes. Bemærk at det er TUIControlleren som påbegynder sekvensen, ved at give brugeren en liste af muligheder og derefter afvente input fra brugeren. Vi har valgt at indikere alle de steder, hvor der forventes input fra brugeren, ved at give Actor en aktivitetsblok. Diagrammet fremviser derudover vores tre-lags arkitektur.

Implementation

Arkitektur

Et af kravene var, at programmet skulle konstrueres med henblik på en 3-lags arkitektur med anvendelse af interfaces. Vi har derfor inddelt pakker med tilhørende klasser derefter. I grænsefladen har vi en view-pakke hvori, at vores TUI med tilhørende interface og controller ligger. I funktionalitetslaget har vi en pakke "dal". "dal" indeholder metoder som sikrer, at oprettelse af brugere følger de retningslinjer som er pålagt. I datalaget har vi pakkerne "persistency" og "dto"(Data transfer Object). "persistency's" formål er, at gemme data midlertidigt per session, mulighed for at gemme/loade fil i hver session, og mulighed for at gemme/loade i database konstant under hver session, hvor data gemmes mellem sessionerne. "dto" indeholder data klassen UserDTO, som indeholder den data, der repræsentere en bruger, og som er det objekt, der bliver manipuleret i funktionslaget og lagret i persistenslaget.
Fordelen ved denne arkitektur er, at det er nemmere at overskue, hvor der sker hvad. Samtidig er det nemmere at vedligeholde og videreudvikle.

GRASP

En bivirkning af brugen af 3 lags modellen og interfaces er at mange klasser er fokuseret på deres individuelle rolle, hvilket betyder at der er lav kobling og høj sammenhængskraft. F.eks. håndterer DatabaseSaver udelukkende information til og fra databasen. Efter GRASP princippet *information expert* er klasser blevet navngivet efter deres rolle. F.eks. har vi TUIController, der kontrollerer tekst UI'en.

Datastrukturer

Datastrukturer handler om at organisere sine data, så man både bruger så lidt plads som muligt, og kan tilgå dataen hurtigt. I denne anledning har vi valgt f.eks. at bruge Maps, hvor man kan definere typen for en nøgle og en værdi. Her skal hver værdi være knyttet til en unik nøgle. Da map er et interface, bruger vi hertil den konkrete HashMap. Måden den virker på er at den konvertere nøglen til en hashværdi, som virker som en reference til dens tilhørende værdi. Det betyder at vi kan tilgår enhver værdi i konstant tid, og i modsætning til f.eks. en ArrayListe, så kan man tilgå dataen med andre typer end integers. Dette er smart i vores menu da man kan knytte en menu kommando (en string) til en kommando objekt. Dette kommando objekt indeholder et runnable objekt, som vil starte en af vores metoder op, når man køre den. Den store fordel her er f.eks, at vi ikke længere har brug for en switch case eller if/else sætninger, til at tjekke hvert mulige input. Da det ikke er strengt nødvendigt at vedligeholde rækkefølgen, som kommandoerne bliver nævnt i, har vi valgt at bruge HashMaps fremfor TreeMaps.

Vi har også anvendt Sets (mængder), som adskiller sig fra ArrayLister med at de ikke kan indeholde de samme værdier flere gange. Det nyttige ved Sets er, hvis vi gerne vil have et Set af alle de bruger Id'er, som ikke er i brug. Så har vi kun brug for et Set med alle bruger Id'er fra 11-99 og alle de Id'er der er blevet brugt. Derefter bruger vi mængde operationen
AllowedID \ UsedID = AvailableID. i Java hedder denne operation **removeAll()**

```
133     public Set<Integer> getAvailableIDs() throws DALException {
134         Set<Integer> usedIDs = persistencyManager.getUserIDList();
135         Set<Integer> availableIDs = new HashSet<>(allowedIDs);
136         availableIDs.removeAll(usedIDs);
```

Figur 5

Koden foroven viser også oprettelse af et HashSet ved at angive et Set som dets konstruktør-argument, hvilket sikrer at alle argumentets elementer placeres i det nye Set.

Indlejrede klasser

Vi har den indlejrede klasse (indre klasse) Command i TUIController, som repræsenterer en menukommando. Den har en Runnable objekt, der kan køre en metode i TUIController, og en String, der beskriver hvad kommandoen gør. og som nævnt tidligere samles disse kommandoer i Maps, så man hurtigt kan få fat i dem og starte den metode op, som de er knyttet til. Command har også en String, der hedder help, men denne har ikke fundet et formål endnu.

Grunden til at man laver indre klasser, er f.eks når den ydre og indre klasse har et meget tæt forhold. Ofte er det fordi den ydre klasse er ment som at være den eneste, der bruger den indre, ligesom i vores Command eksempel.

Interfaces

Interfaces er i dette projekt blevet brugt til at adskille 2 stykker kode for at åbne lav kobling. Dette har medført at vi kan ændre er klasses funktioner alt det vi vil, så længe at de implementerer interfacets funktioner korrekt. Dette har medført at koden er blevet meget mere robust, da en ændring i en klasse der implementere et interface, ikke medfører ændringer andre steder i koden. Brugen af interfaces har ført til brug at polymorfi, da man kan bestemme hvilken klasse der implementere interfacet.

Polymorfi

Polymorfi er i dette projekt blevet brugt i forhold til, hvilken måde systemet gemmer data på. Programmet er blevet designet på den måde, at alt efter hvilken klasse der implementerer interfacet *IPersistency,* Så gemmer den på forskellige måde. Dette er nyttigt da vi derved ikke behøver at ændre på programmet for at hente informationen. Gengivet på nedenstående stykke kode er alle funktionerne som alle persistens klasser implementerer. De 3 forskellige klasser er *MemorySaver*, der gemmer data midlertidigt, *FileSaver,* der gemmer dataene på en fil når programmet lukkes, og DatabaseSaver, som tilgår en database, hvor den gemmer data efter disse er blevet oprettet.
Det er muligt at tilføje andre måde at gemme dataene på, så længe at de implementerer interfacet.

```
10 public interface IPersistency {
11
12     void init() throws DALException;
13
14     boolean save(UserDTO user);
15
16     void updateUser(UserDTO user, int i);
17
18     UserDTO getUser(int id) throws DALException;
19
20     ArrayList<UserDTO> getUserList() throws DALException;
21
22     Set<Integer> getUserIDList() throws DALException;
23
24     boolean deleteUser(int userID);
25
26     void quit();
27
28 }
```

Figur 6

Det er derefter muligt at anvende klasser, som implementerer IPersistency, uden at bekymre sig om den specifikke klasse. Dette anvender vi i oprettelse af UserDAOBasic objekter, hvor konstruktøren forventer et IUserDAO, som forventer et IPersistency objekt. Denne funktion accepterer alle tre typer af persistens-

klasser, som er nævnt tidligere, uden eksplicit at nævne de klasser. Dets persistens argument kan altså tage flere former, hvilket udgør polymorfi.

Undtagelser

Undtagelser (Exceptions) bruges i koden til at meddele fejl, der måtte forekomme under kørslen af programmet, eller tilstande, som ikke er helt almindelige. Vi bruger til tider "try-catch"-udtryk til at indfange disse undtagelser når de opstår hvorefter de kan håndteres. I nogle funktioner vælger vi at undlade disse udtryk, og sender blot undtagelser videre i stacken. Dette er tilladt såfremt man i funktionsdefinitionen indfører "throws" efterfulgt af undtagelser, som funktionen muligvis kunne videresende. Til tider opstår fejlen i vores egne funktioner, eller også fanger vi generiske undtagelser, og vil gerne kaste en ny undtagelse som er mere sigende. Dette opnår vi med et "throw new Exception" udtryk. Vi bruger derudover undtagelserne til debugging, ved at få en eventuel undtagelse til at printe en stack trace, som følger fejlene helt tilbage til, hvor de opstår. Dette giver et godt overblik når fejl opstår. Andre fejl som man forventer at der kan opstå, kan også overskueliggøres med undtagelser. Eksempelvis indlæsning af en fil, hvor filen man vil læse fra ikke eksisterer. Når vi læser fra en fil kan der opstå flere slags fejl, i dette tilfælde kastes en "FileNotFoundException", der fortæller brugeren, at filen ikke findes, og brugeren informeres dermed, at det ikke er en eventuel "IOException", som skaber problemer i indlæsningen af filen. Ved at type-inddele undtagelserne, giver man brugeren et overblik over, hvilke slags fejl der opstår, og man kan dermed hurtigt konkludere om det en fejl som kan ignoreres eller en fejl som kræver håndtering via et "try-catch"-udtryk.

Lambda udtryk

I vores kode findes der to forskellige anvendelser af lambda udtryk.
Den ene anvendelse, som ses på Figur 7, var mulig eftersom Rule-klassens konstruktør forventer to argumenter, hvoraf det ene er et Predicate objekt. Predicate er et funktionelt-interface, eftersom det kun har én metode, og koden til at generere et nyt Predicate objekt kan derfor erstattes med et lambda udtryk.

```
23          Rule<Integer> idRule = new Rule<>
24                  ("ID must be between" + minID + " and " + maxID
25                          , t -> t > minID && t < maxID);
```

Figur 7

Som det kan ses på figuren, er Predicate konstruktøren erstattet med et lambda udtryk, som opretter et Predicate objekt og sikrer at objektets ene funktion (***test()***) returnerer hvorvidt dets input er mindre end ***minID*** og størrer end ***maxID***.

Den anden form lambda udtryk kan ses i Figur 8. Denne form for et lambda udtryk minder om et Delegate som det kendes fra C# og tillader os at referere til metoder. Vi har oprettet en indlejret klasse ved navn Command, som forventer tre argumenter til sin konstruktør; en Runnable og to Strings. Ligesom Predicate, er Runnable et funktionelt-interface, og lambda udtrykket i følgende figur opretter da et Runnable objekt og lader kroppen af dets ene funktion være et kald til den metode vi angiver.

```
70          Command create = new Command(this::createUser, "Create a new User", "Help for Create User goes here!");
```

Figur 8

Eftersom nogle af de funktioner, hvorpå vi godt kunne tænke os at anvende denne form for lambda udtryk, kastede undtagelser, indførte vi et interface, som extender Runnable, og kun indeholder én funktion som ikke er prædefineret, hvorved interfacet bliver et funktionelt-interface. Klassen indeholder en default implementation af ***Run()*** funktionen, som kalder den metode der angives ved lambda udtrykket eller ved

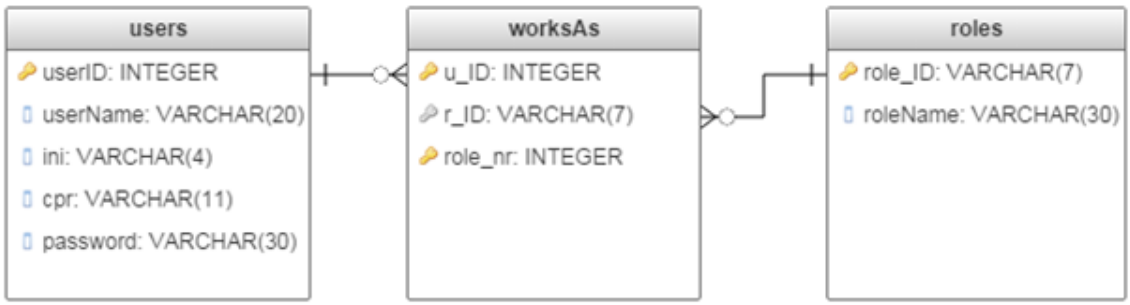
oprettelse af et objekt af interfacet. ***Run()*** funktionen har vi givet ansvar for at fange og håndtere undtagelser. I en bedre løsning afhænger håndtering af undtagelser af metoden som kaster den. I vores simple program var denne løsning dog tilstrækkelig da vi i alle tilfælde vil reagere ens (nemlig ved at afslutte nuværende metode og gå tilbage til den forrige menu).

```
57      private interface UpdateMethod extends Runnable {
58          void method() throws DALException, Keyboard.DALKeyboardInterruptException;
59
60          default void run() {
61              try {
62                  method();
63              } catch (DALException | Keyboard.DALKeyboardInterruptException e) {
64
65              }
66          }
67      }
```

Figur 9

Persistens

Persistence eller ”Persistency” pakken står for datamanipulation. Pakken er delt i 3 dele, hvoraf disse dele tilgås via et interface kaldet ”IPersistency”. Dette giver mulighed for at anvende persistency pakken i andre sammenhænge, da andre programmer kan tilgå interfacet. Som tidligere nævnt, er pakken delt i 3 dele. MemorySaver, FileSaver, og Database saver. Disse 3 dele er uafhængige af hinanden og tilbyder 3 forskellige måder at gemme data på. Midlertidigt, på en fil, og på en database. Den midlertidige hukommelse *MemorySaver* har som bonus, at dette er meget simpelt, da eventuelle objekter kan gemme i ArrayLister. Dog nytter dette ikke noget, hvis programmet skal bruges over flere sessioner. Hertil kan *FileSaver* bruges. Her gemmes data i en fil hver gang systemet lukkes ned. Ulempen er her, at hvis programmet lukkes på en unaturlig måde, så bliver eventuelle ændringer og tilføjelser til systemet ikke gemt. For at løse dette problem har vi brugt en database. Ved brug af databasen tilgås og gemmes alt data i databasen. Dette betyder at når f.eks. en bruger er blevet oprettet, gemmes denne i databasen med det samme. For at gemme ting i databasen og undgå redundant data har vi oprettet 3 tabeller; *roles*, som beskriver de mulige roller, *users*, som gemmer bruger informationen, og en link tabel mellem disse, *worksAs,* der beskriver hvilke roller som en ’user’ har. primær nøglen *role_nr* er til for at skabe en unik kandidat nøgle, for at relationen kan identificeres.
Nedenunder ses database skemaet og relationerne mellem disse. Variablene har en gylden nøgle til venstre for sig, hvis de er primær-nøgler, og en sølv nøgle, hvis de er fremmed-nøgler.



Figur 10

Konklusion

Vi har fremstillet et program, ud fra de beskrevne krav til funktion og implementering, som var stillet til opgaven. Resultatet er blevet et robust og vedligeholdelsesvenligt program, som opfylder kravene, som blev stillet. Derudover indeholder programmet funktionalitet til både at gemme i en database eller på fil, alt efter hvad en bruger ønsker.

Kilder

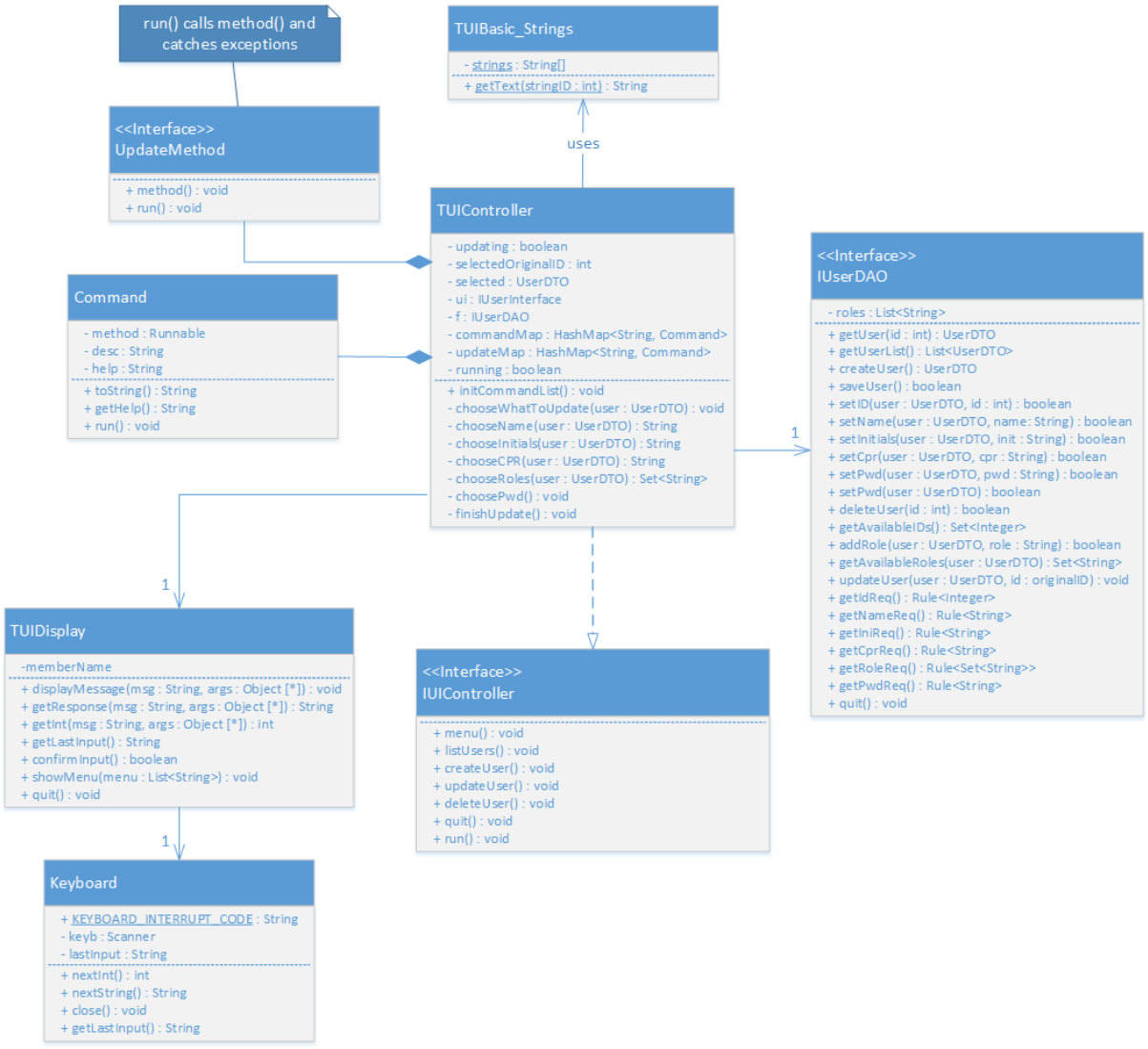
Password Generator - https://www.youtube.com/watch?v=PBAX8r9YWW0

Bilag

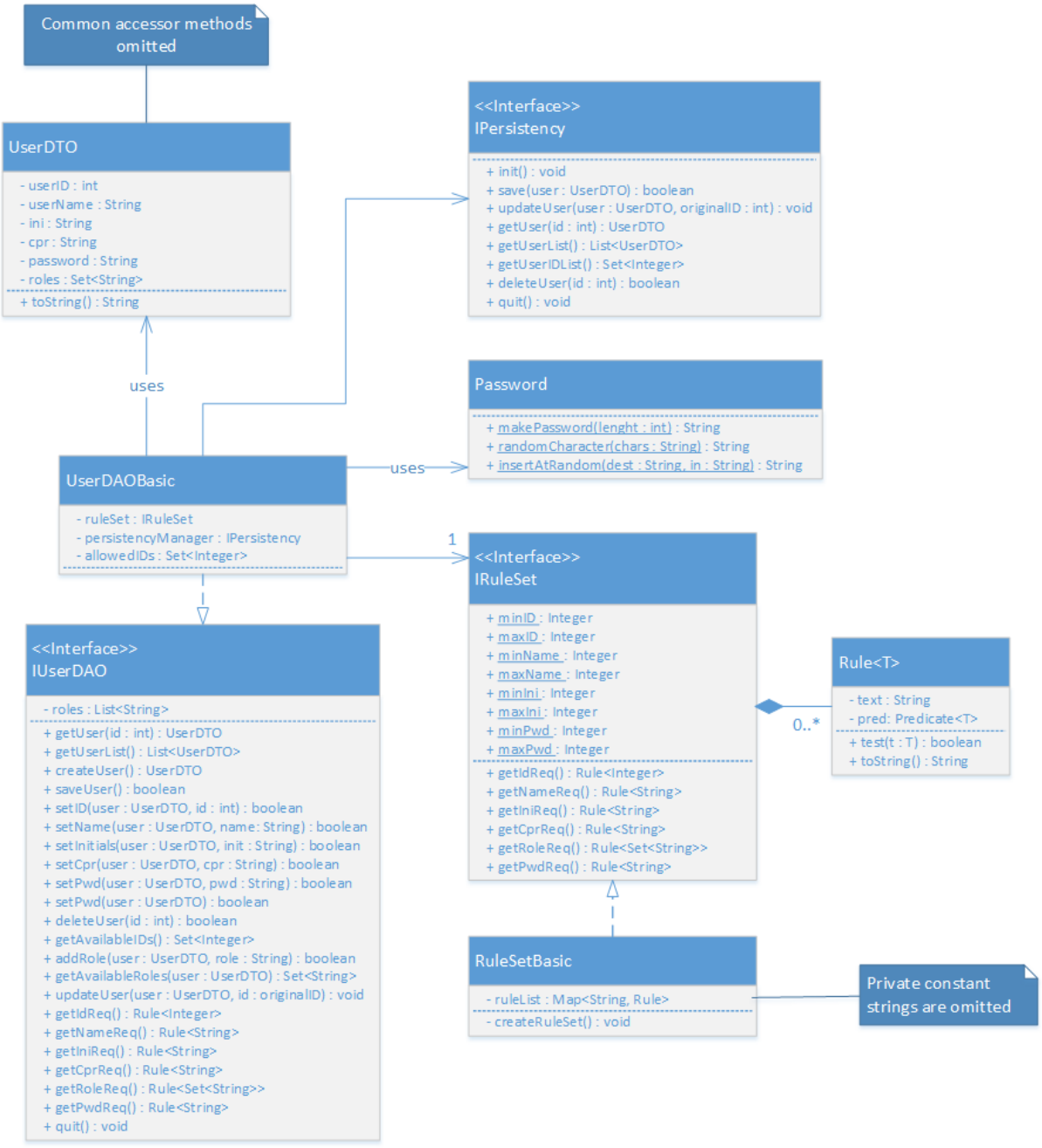
A. Diagrammer

A.1 Klassediagram

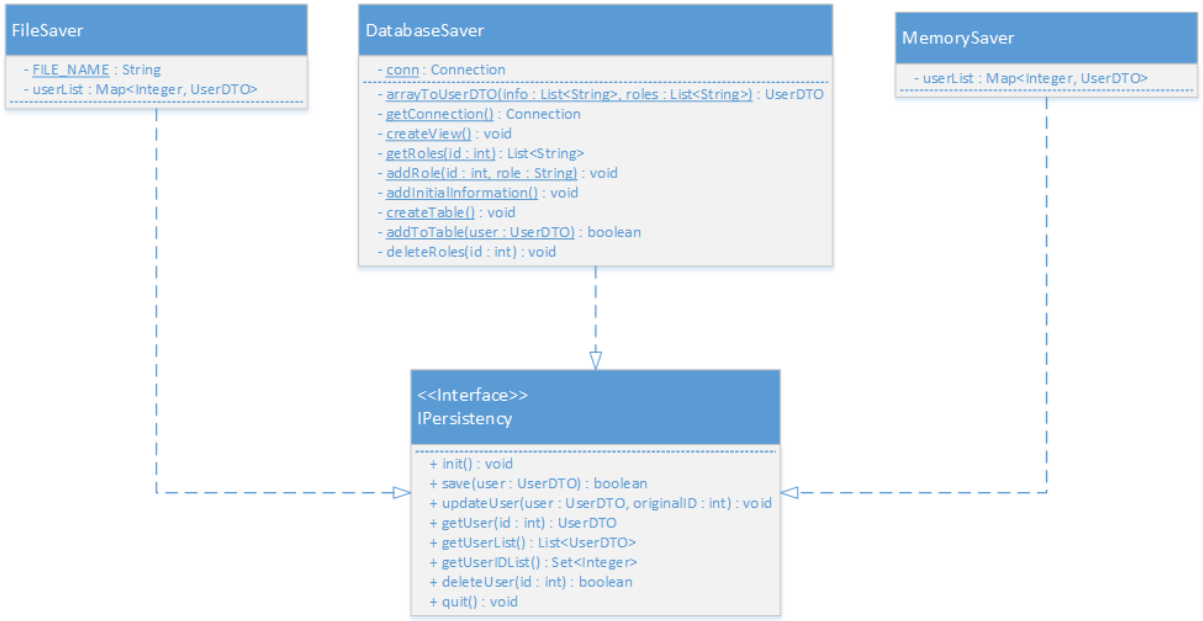
Brugergrænsefladen og dets kendskab til Funktionslaget



Funktionslaget og dets kendskab til Persistenslaget

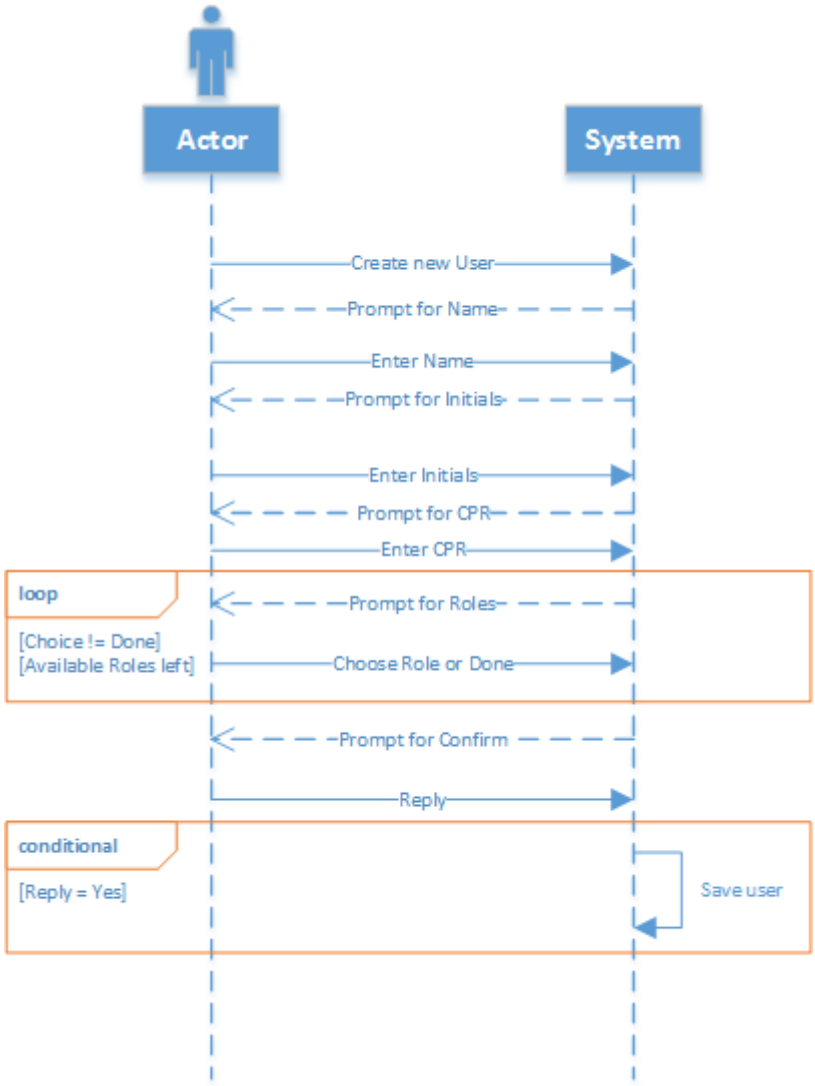


Persistenslaget

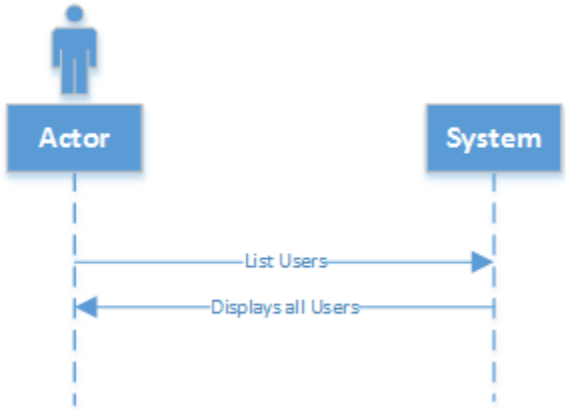


A.2 System sekvensdiagrammer

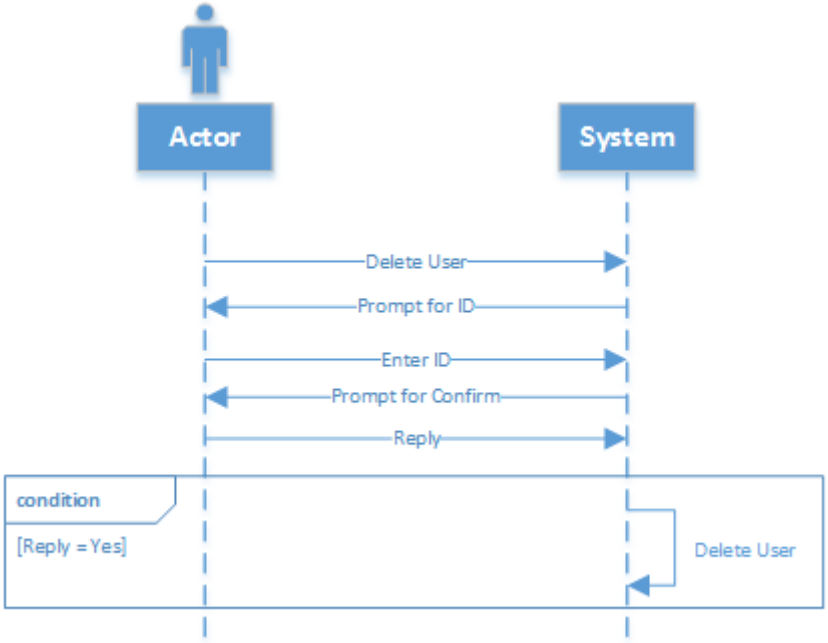
Create User



List Users

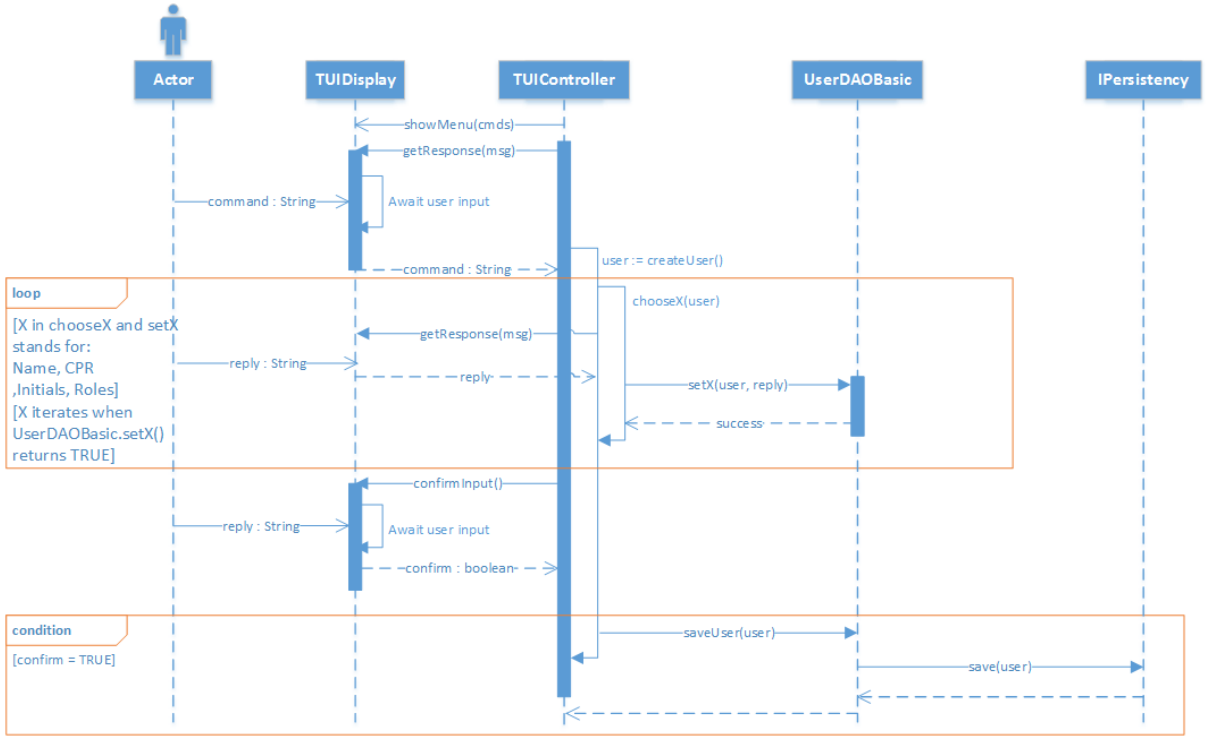


Delete User

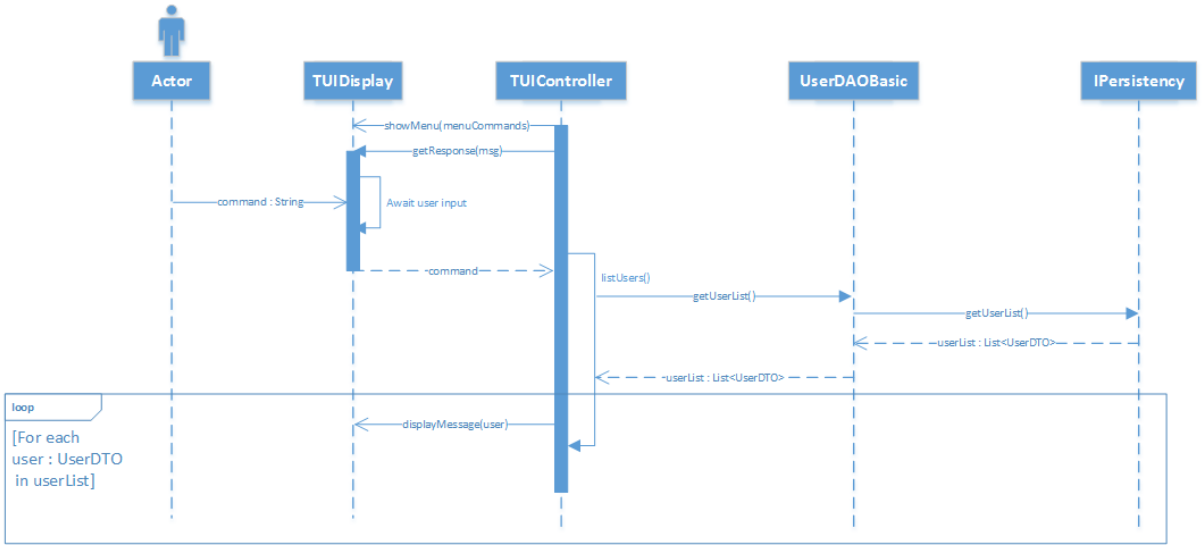


A.3 Design sekvensdiagrammer

Create User



List Users



Update User