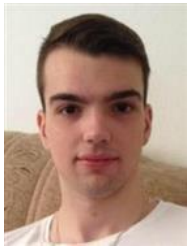
CDIO part 1

Mikkel
Leth
s153073

Sebastian
Hjorth
s153255



Senad
Begovic
s153349

Thomas
Madsen
s154174

Rasmus
Olsen
s153953

Nicki
Rasmussen
s153448

Dato for aflevering: 27-02-2016

Institute: Denmarks Technical University

Assignment type: CDIO project

Class: 02324

Advisors: Ronnie Dalsgaard, Christian and Stig Høgh

Time table

Time table							
Date	Participant	Design	Implantation	Test	Documentation	Other	Total
2015-26-02	Mikkel	3	3	1	2		9
	Nicki	1	2	2	4		9
	Rasmus	3	2	1	3		9
	Sebastian	2	4	2	2		10
	Senad	2	3	1	3		9
	Thomas	3	2	2	2		9

Table of content

Introduction

Course goals

We are a group of students at the Technical University of Denmark. Who have been given the task to make a login system using our knowledge gained from the course: “Higher Programming”. With this at mind, we are going to implement the four terms (CDIO) to this assignment in the best way possible to satisfy our costumer. We will use our skills to analyze, design, implement and test our system, so it will fit the demands and requirements set down by the customer, which will be specified in the section below.

Purpose

Vi har fået tildelt en opgave, der går ud på at lave en bruger administrationsmodul med en simpel brugergrænseflade. Formålet med denne opgave er, at vi skal videre arbejde på programmet senere hen i 3-ugers perioden.

Term definition

CDIO

The four design principles: Conceive, Design, Implement and Operate, are what make up CDIO.

Resume

The first steps in making the program was analyzing the requirements, which is detailed in the next chapter in “Requirements”. With these in mind, we made some initial diagrams visualizing the structure of the program and the user interaction, respectively by domain model and use case descriptions, which led to our system sequence diagram.

The domain model were developed into our class diagram, which details exactly which responsibilities each class has, and their connection to the other classes. We then developed the design sequence diagram, which declared the method calls between the classes. We made a design sequence diagram for each use case description.

The program had to be structured around the three-layer rule; therefore, we split the program into four different packages: controllers, data, functionality and interfaces. The last three were made with interfaces to contain them.

At last we tested the program and made sure that all requirements were fulfilled. Inputs were also secured to make sure that any illegal inputs would not be accepted.

Main section

Requirements

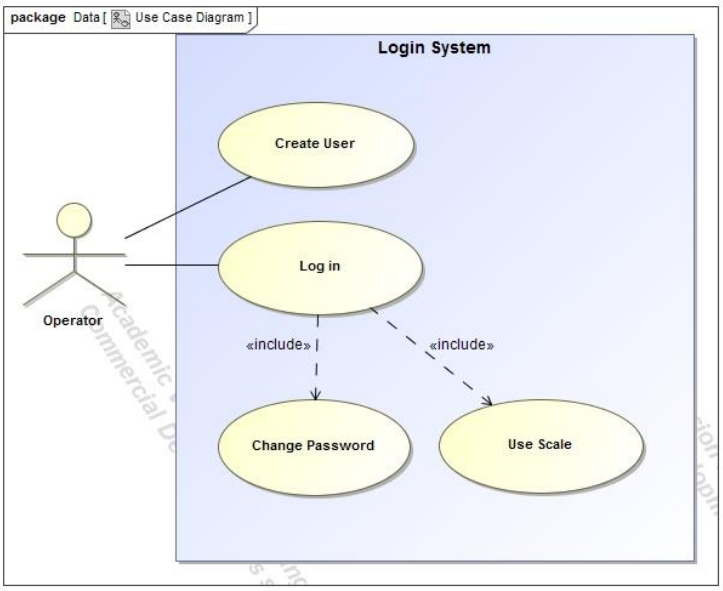
For this project, we've been asked to implement the following things for our user interface:

Functional requirements	Nonfunctional requirements
The admin should be able to create, see a list of, update and delete users.	Different menus
The user should be able to use the weight after logging in	Follows 3 layer model
The user should be able to change their password	
The user ID (int 11-99) is automatically generated by the program.	
the first given password (string) must be autogenerated by the program and has to meet the same requirements as those set by DTU (https://password.dtu.dk/)	

After the requirements we started to work on the domain-model. This model is very good at having a quick look at the actual physical things that depend and work with each other. The domain-model can be seen down below:

Along the work of the domain-model we figured out, that only 3 objects are actual domains in this program. We have the operator or the user, who are dependent of the weight. The weight are dependent of the object that needs to be weighed. And last the object, that the user wants to weigh.

We then proceeded on to the use case diagram and the use case descriptions. The purpose of the use case diagram was to give a good "before coding" look of what the user should be able to do. What should the user be able to perform? Down below we have the use case diagram.

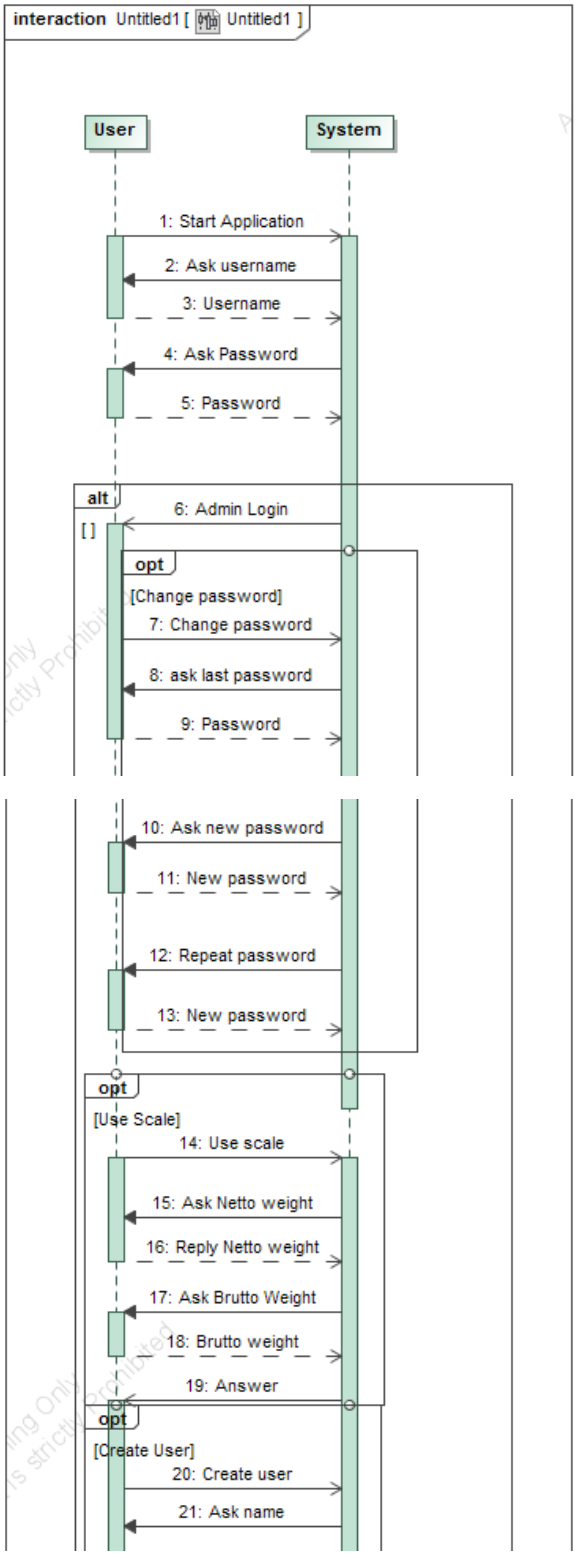


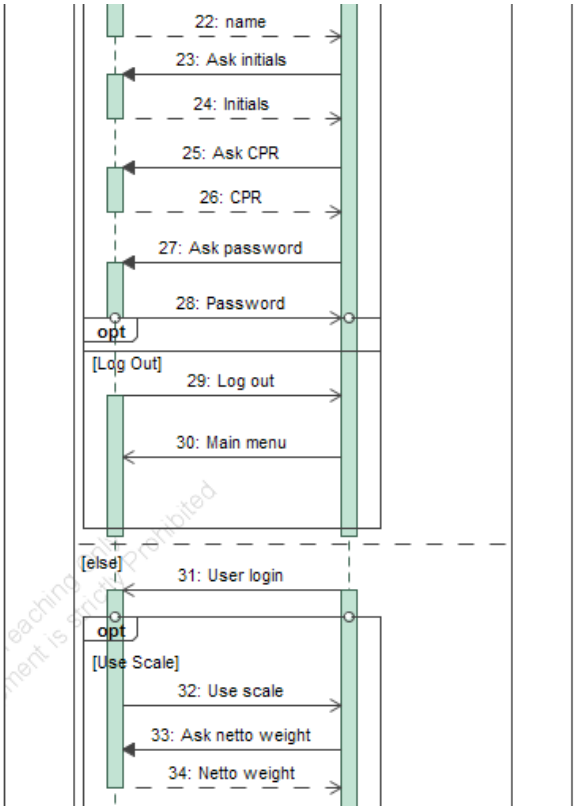
After this we wanted to take the first steps towards the code part of the program. We wanted to do that by making use case descriptions. We wanted a step by step procedure of what needs to happen, when a user wants to perform a given function of the program.
Here we have an example of a use case description, this one is about creating a new user.

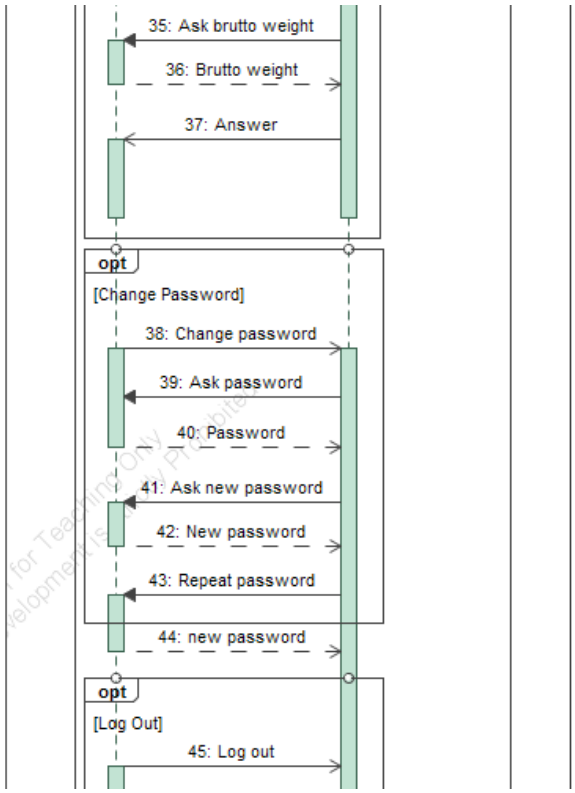
Use case ID:	1
Use case name:	Create User
Actors:	Operator
Description:	The operator tries to create an account for logging in
Preconditions:	Operator does not already have an account
Post conditions:	A new account is created
Normal flow:	1. Application starts 2. User is presented with login screen 3. User chooses to create a new account 4. User is asked for his/her name 5. User inputs name 6. User is asked for his cpr-number 7. User inputs cpr-number 8. User is asked for a password 9. User inputs a password 10. User is asked to verify the password 11. User verifies password a. User fails to verify and is asked to choose new password 12. Account is created 13. User is sent back to login screen
Alternative flow:	
Exceptions:	

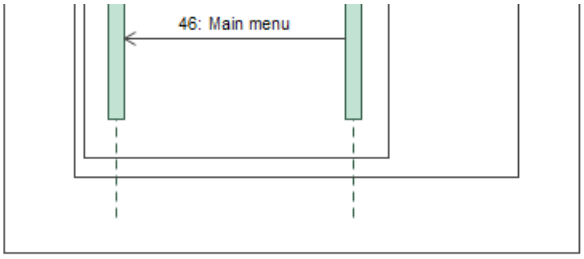
The rest of the use case descriptions are in the appendix'.
For the "log in" use case description, see appendix 1. For the "change password" use case description see appendix 2. And finally for the "use scale" use case description, see appendix 3.

From here we needed to work a bit on how the communication between the user and the system should be. We therefore made a system sequence diagram, where the system is seen as a black box. We therefore only see messages and replies between the system and the user. The system sequence diagram can be seen here:


interaction Untitled1 [Untitled1]
User
System
1: Start Application
2: Ask username
3: Username
4: Ask Password
5: Password
alt
[ ]
6: Admin Login
opt
[Change password]
7: Change password
8: ask last password
9: Password
10: Ask new password
11: New password
12: Repeat password
13: New password
opt
[Use Scale]
14: Use scale
15: Ask Netto weight
16: Reply Netto weight
17: Ask Brutto Weight
18: Brutto weight
19: Answer
opt
[Create User]
20: Create user
21: Ask name


22: name
23: Ask initials
24: Initials
25: Ask CPR
26: CPR
27: Ask password
28: Password
opt
[Log Out]
29: Log out
30: Main menu
[else]
31: User login
opt
[Use Scale]
32: Use scale
33: Ask netto weight
34: Netto weight

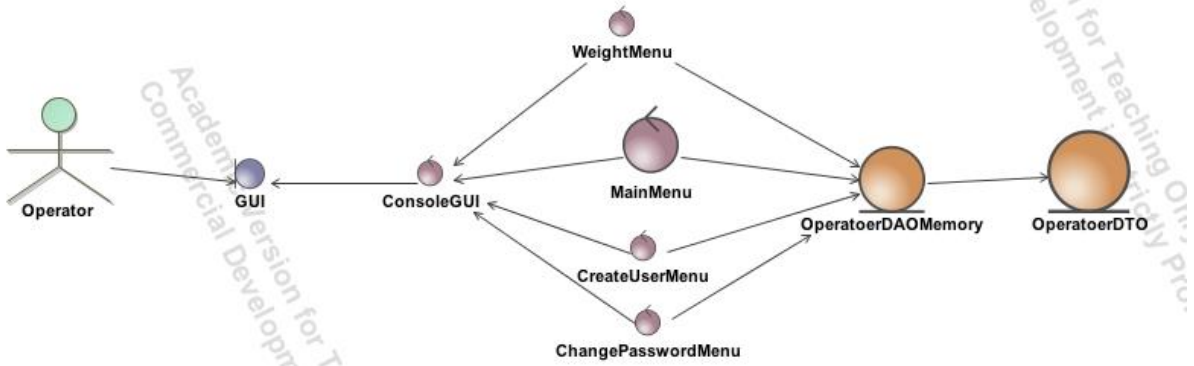

35: Ask brutto weight
36: Brutto weight
37: Answer
opt
[Change Password]
38: Change password
39: Ask password
40: Password
41: Ask new password
42: New password
43: Repeat password
44: new password
opt
[Log Out]
45: Log out



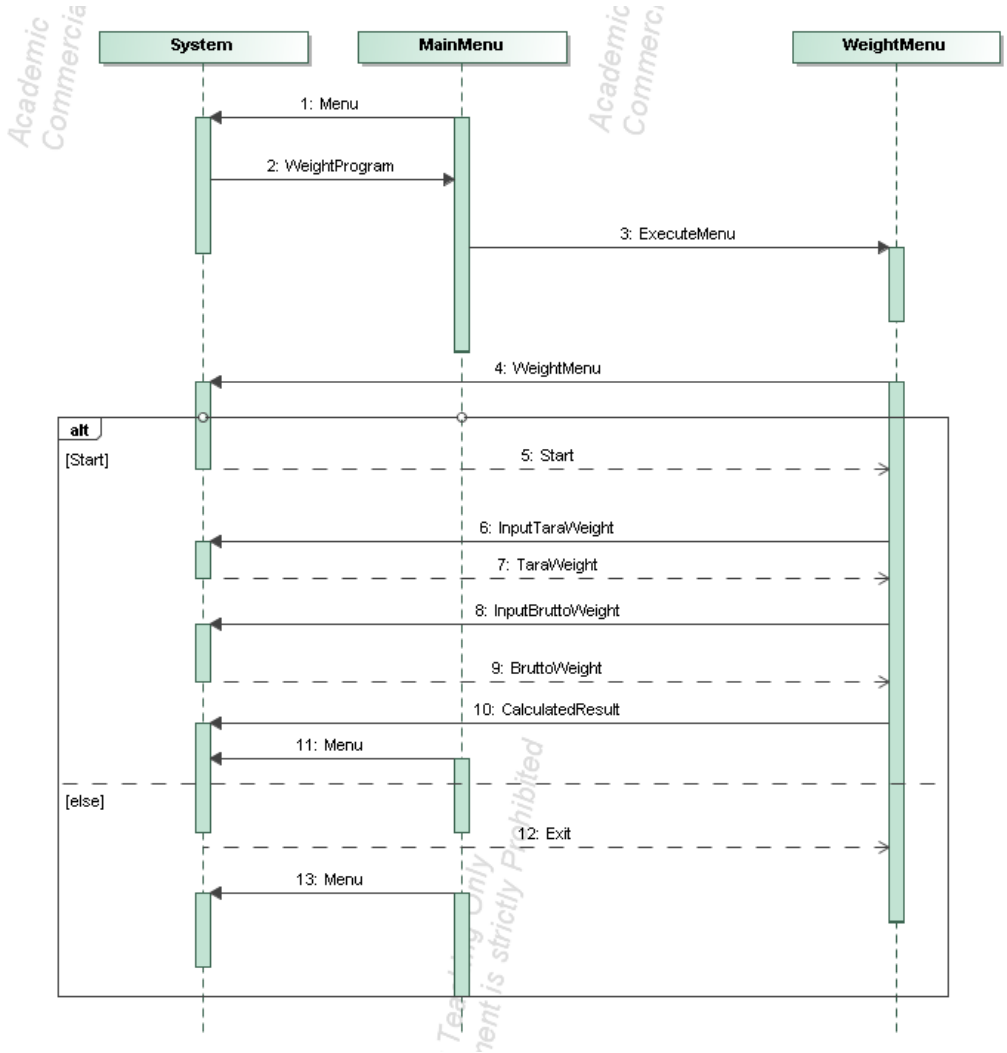
The system sequence diagram was a very good way to discover another aspect of the user's communication with the program, and it made a proper foundation for the future programming work. It also gave a great insight in the aspects we needed to take care of when our work continued with the design sequence models.

Before we could proceed with the design sequence diagrams, we needed to have a better overview of the actual classes we wanted to have in the programme. This needed to conform with the three-layer-model in order to fulfill the requirements. Hence the BCE-model, to give us a picture of how the program should communicate within itself, and that the program had the right amount of controllers as well as data containers, so no class would sit with too much responsibility.

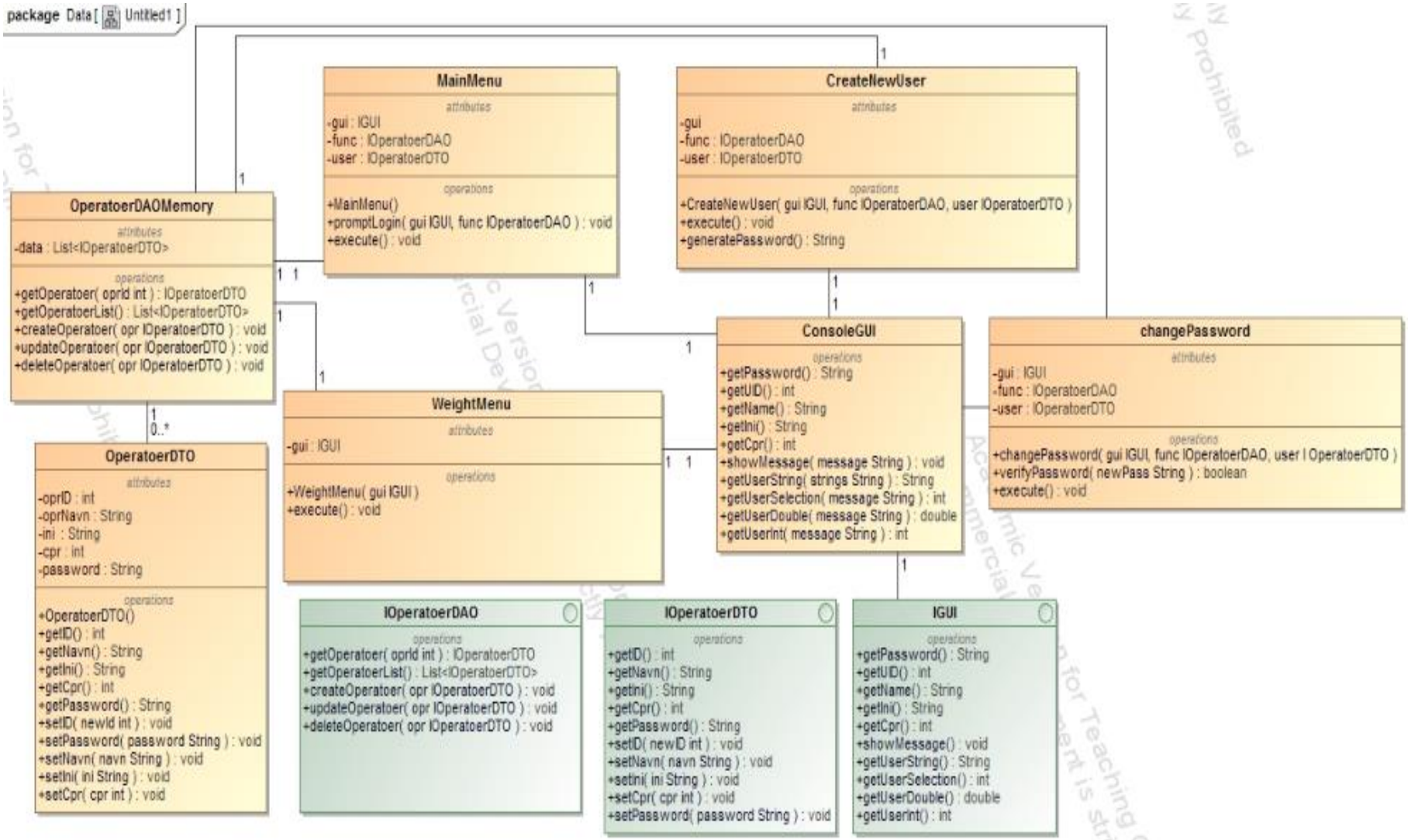
The BCE-model can be seen below:



In our BCE-model, we have two entities, the OperatoerDTO is the data container, while OperatoerDTOMemory is the functional entity.
Now we could proceed to the communication within the system. What happens in the system when a user wants to use a certain function. We did this by making design sequence diagrams for each use case. This would make the implementation so much easier, because the dsd's would give us a great insight in the actual code we needed to write. The dsd for the "use scale" can be seen down below:



The rest of the dsd's can be seen in the appendix.
For the "change password" dsd, see appendix 4. For the "create user" dsd, see appendix 5. And finally for the "log in" dsd, see appendix 6.
The only thing left now was to put all these aspects together into the class diagram. It can be seen under here:


package Data [Untitled1]
MainMenu
attributes
-gui : IGUI
-func : IOperatoerDAO
-user : IOperatoerDTO
operations
+MainMenu()
+promptLogin(gui IGUI, func IOperatoerDAO) : void
+execute() : void
CreateNewUser
attributes
-gui
-func : IOperatoerDAO
-user : IOperatoerDTO
operations
+CreateNewUser(gui IGUI, func IOperatoerDAO, user IOperatoerDTO)
+execute() : void
+generatePassword() : String
OperatoerDAOMemory
attributes
-data : List<IOperatoerDTO>
operations
+getOperatoer(oprId int) : IOperatoerDTO
+getOperatoerList() : List<IOperatoerDTO>
+createOperatoer(opr IOperatoerDTO) : void
+updateOperatoer(opr IOperatoerDTO) : void
+deleteOperatoer(opr IOperatoerDTO) : void
ConsoleGUI
operations
+getPassword() : String
+getUID() : int
+getName() : String
+getIni() : String
+getCpr() : int
+showMessage(message String) : void
+getUserString(strings String) : String
+getUserSelection(message String) : int
+getUserDouble(message String) : double
+getUserInt(message String) : int
changePassword
attributes
-gui : IGUI
-func : IOperatoerDAO
-user : IOperatoerDTO
operations
+changePassword(gui IGUI, func IOperatoerDAO, user I OperatoerDTO)
+verifyPassword(newPass String) : boolean
+execute() : void
WeightMenu
attributes
-gui : IGUI
operations
+WeightMenu(gui IGUI)
+execute() : void
OperatoerDTO
attributes
-oprID : int
-oprNavn : String
-ini : String
-cpr : int
-password : String
operations
+OperatoerDTO()
+getID() : int
+getNavn() : String
+getIni() : String
+getCpr() : int
+getPassword() : String
+setID(newId int) : void
+setPassword(password String) : void
+setNavn(navn String) : void
+setIni(ini String) : void
+setCpr(cpr int) : void
IOperatoerDAO
operations
+getOperatoer(oprId int) : IOperatoerDTO
+getOperatoerList() : List<IOperatoerDTO>
+createOperatoer(opr IOperatoerDTO) : void
+updateOperatoer(opr IOperatoerDTO) : void
+deleteOperatoer(opr IOperatoerDTO) : void
IOperatoerDTO
operations
+getID() : int
+getNavn() : String
+getIni() : String
+getCpr() : int
+getPassword() : String
+setID(newID int) : void
+setNavn(navn String) : void
+setIni(ini String) : void
+setCpr(cpr int) : void
+setPassword(password String) : void
IGUI
operations
+getPassword() : String
+getUID() : int
+getName() : String
+getIni() : String
+getCpr() : int
+showMessage() : void
+getUserString() : String
+getUserSelection() : int
+getUserDouble() : double
+getUserInt() : int

Implementation

Three-layer model and GRASP

One of the requirements for this program, was to make the design based on the 3 layer model, which made us split the program in 3 parts: Interface, functionality and data. By using the 3 layer model, we are making low coupling on the program (GRASP), which makes it easier to maintain and to switch out the different parts of the program. That, as well as edit parts of the program in cases where it is needed for future assignments. By using the 3 layer model we've implemented some parts of GRASP already, and we've made sure the program had high cohesion. We've not giving the classes too much to maintain, but made sure that all classes have an equal amount of work.

We've also implemented controllers between the interface layer and the functionality layer, which will maintain the program's flow, and what to fetch from the functionality layer and bring to the interface layer. Those controllers communicates with the GUI(which is just a console window for now) solely through the IGUI interface. We did this to make it easy to change the GUI in the future without having to rewrite our use case logic.

Using the three layer model also has other benefits, mainly that it will be easier to integrate a database later on because of the low-coupling that comes with it. Right now the code only stores the changes in-memory, which means that any changes will reset once the program is restarted, but if we create a class which implements the IOperatoerDAO interface, then it is trivial to make the program use a database or any other kind of storage.

We have made the shufflebag to utilize when a user needs a password. The shufflebag insures that all the conditions for making a password is met when auto generated, this way a user gets the right password complexity. The shufflebag is also used in assigning ID's to newly created accounts as it ensures that no number is ever picked twice.

In project we use 3 different controllers, because we have 3 individual menus. First is the main menu, this class controls logging in and the whole startscreen of the program, so it's the one with the most responsibility. Next in line is the createUser controller, which deals with the the user accounts and passwords. Lastly we have the weight menu, which controls the operations of the weight itself. We decided to split them up between the classes to split the responsibility and to better support extensibility.

Test

Here is a testscript for an administrator that wants to use the weight. Next to it, the administrator wants to add a new user. The administrator id is 10:

```
Indtast brugerID:
10
Indtast password:
admin
Du har nu følgende muligheder:
0.      Ændre password
1.      Start vægt programmet
2.      Logud
3.      Opret ny bruger

1
Du har nu følgende muligheder:
0.      Start vægt
1.      exit

0
Indtast tara-vægten i kg:
0,3
Indtast brutto-vægten i kg:
0,9
Netto-vægten er: 0.6000000000000001 kg
Du har nu følgende muligheder:
0.      Start vægt
1.      exit
```

```
3.      Opret ny bruger

3
Indtast navn:
Test
Indtast initialer:
TT
Indtast CPR:
1234567890
Genererer kodeord af 8 symboler
Ny account skabt med ID: 57 og adgangskode: +T93eV_o
Du har nu følgende muligheder:
0.      Ændre password
1.      Start vægt programmet
2.      Logud
3.      Opret ny bruger

2
Indtast brugerID:
57
Indtast password:
+T93eV_o
Du har nu følgende muligheder:
0.      Ændre password
1.      Start vægt programmet
2.      Logud
```

As you can see, we have created a new user (a precondition for this is to be logged in on an admin user), and tried to log in with the given ID and PassWord, in order to confirm that the user was a normal user, and not an admin, as well as the userID and password was functional.

Conclusion

Overall conclusion

To sum it all up, our project has successfully fulfilled the demands and requirements given to us, and we are satisfied with what we've accomplished.We've manage to create a successful user interface for future usage in the coming projects. We have gone through the different processes needed to deliver our product, and are quite satisfied with the result.

Product oriented conclusion

As mentioned above, we've made a successful user interface which is designed for users and an admin to maintain the system. The functions of the program is following: Create user (Admin only option), change password and find netto weight. To find the netto weight of a given object, you have to use the object we want the weight of minus the tare weight. We've managed to accomplish all the requirements listed, and made the program ready for future usages.

Appendix

Appendix 1

Use Case ID:	2
Use Case Name:	Log in
Actors:	Operator
Description:	The operator attempts to log in
Preconditions:	The operator has an account already
Postconditions:	The operator is logged in with his account
Normal Flow:	1. Application starts 2. User is presented with login screen 3. User chooses to log in 4. User is asked for his username 5. User inputs username 6. User is asked for his password 7. User inputs his password 8. System checks login credentials a. Either the username or password did not match, the user is sent back to the login screen with an error message b. Login credentials match the admin login i. User is logged into the admin account 9. User is logged into his account
Alternative flow:	User inputs login credentials matching to the admin account: 1. User inputs login credentials for admin account 2. User is logged into the admin account
Exceptions:	

Appendix 2

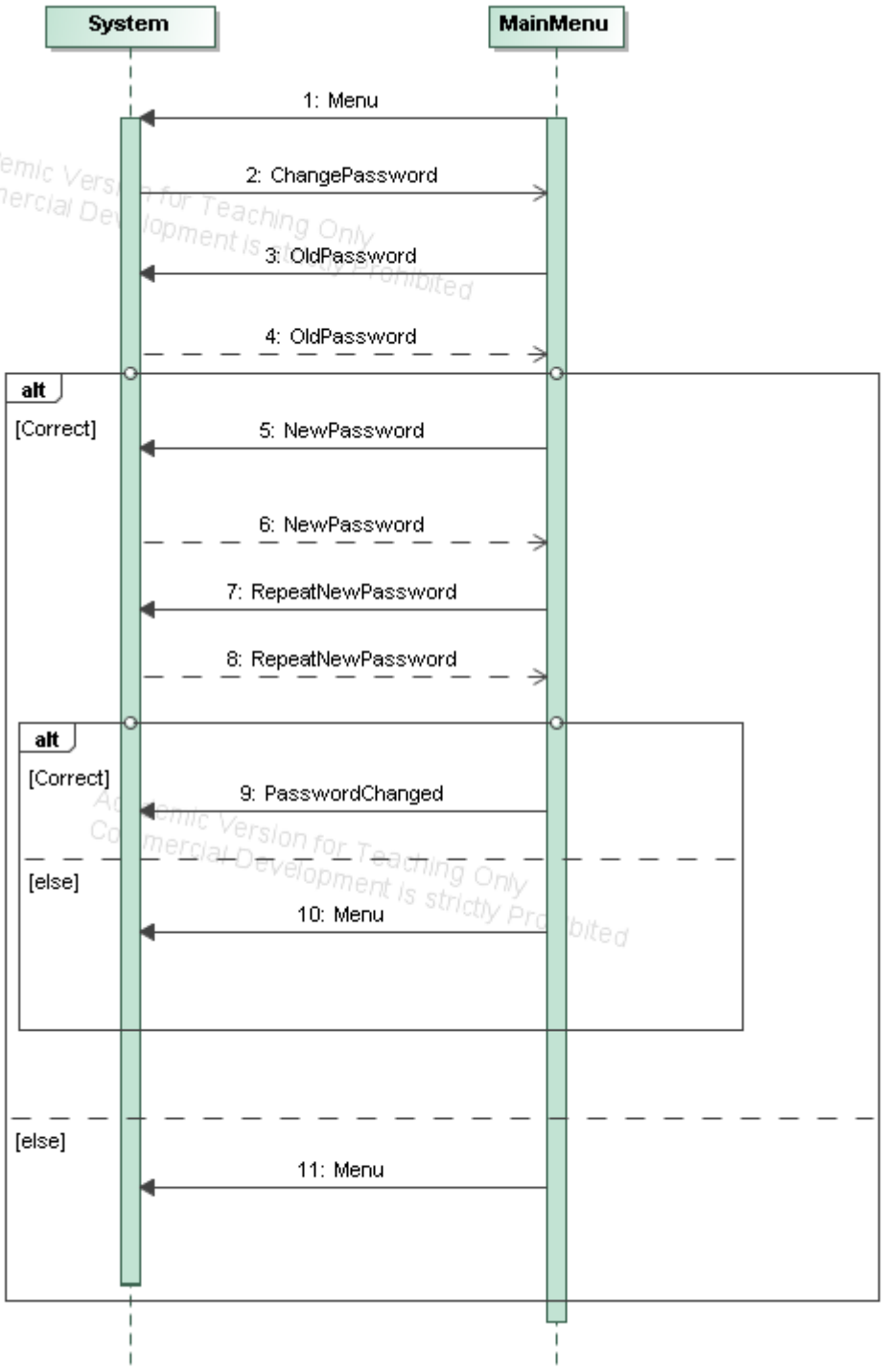
Use Case ID:	3
Use Case Name:	Change password
Actors:	Operator
Description:	The user attempts to change his password to another one
Preconditions:	1. The user already has an account 2. The user is logged in to his account
Postconditions:	The password is changed

Normal Flow:	1. User chooses to change password 2. User is asked for his old password 3. User inputs his old password a. Password incorrect, user is sent back to main screen 4. User is asked for a new password 5. User inputs a new password 6. User is asked to verify the new password 7. User verifies password . Password not verified, user is asked for a new password 8. Password is changed 9. User is sent back to main screen
Alternative flow:	
Exceptions:	

Appendix 3

Use Case ID:	4
Use Case Name:	Use Scale
Actors:	Operator
Description:	User uses the scale application
Preconditions:	User is logged in
Postconditions:	Weight is calculated
Normal Flow:	1. User chooses to use the weight application 2. User is sent to scale screen 3. User is asked for tara weight 4. User inputs tara weight a. Invalid input i. User is asked for tara weight 5. User is asked for brutto weight 6. User inputs brutto weight . Invalid input . User is asked for brutto weight 7. System prints netto weight 8. User is sent back to main screen
Alternative flow:	
Exceptions:	

Appendix 4


System
MainMenu
1: Menu
2: ChangePassword
3: OldPassword
4: OldPassword
alt
[Correct]
5: NewPassword
6: NewPassword
7: RepeatNewPassword
8: RepeatNewPassword
alt
[Correct]
9: PasswordChanged
[else]
10: Menu
[else]
11: Menu

Appendix 5