CDIO 2
02324 - Videregående Programmering

Gruppe Nr. 25
Afleveringsfrist: lørdag d. 18/03-2017 kl. 05:00

Denne rapport afleveres via CampusNet (der skrives ikke under).
Denne rapport indeholder Side 19 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	4	3	2	0	6	15
Jimmy	0.5	0	16	4	0	20.5
Lasse	2	2	3	0	10	17
Mehdi	1	4	2	0	8	15
Yanki	0	0	12	6	1	19
Total	8	6	40	3,5	18	86.5

Indhold

Resume

Denne rapport omhandler udarbejdelsen af en vægt simulator med en grafisk brugergrænseflade. Først gennemgår rapporten kravene for hvad der forventes af det færdige produkt. Dernæst kommer et analyseafsnit, som omhandler analyse og design af programmer, hvor der vil forklares om forskellige modeller vi har lavet for at få et overblik over hvordan programmet skal bygges op. Efterfølgende kommer implementations afsnittet, som dækker over selve udarbejdelsen af koden til programmet og tankerne omkring. Det næste rapporten indeholder er et testafsnit, som verificerer og vurderer programmet, for at se om det virker som det skal, og hvor solidt det er bygget op. Til sidst i rapporten findes konklusion, der opsummerer resultatet for projekter. Selve formålet med rapporten er, at give en indsigt i, hvordan vi har håndteret opgave, ud fra den udleverede opgavebeskrivelse.

Indledning

I dette projekt har vi fået til opgave at færdigimplementere et en vægtsimulator med en grafisk brugergrænseflade, som både kan kontrolleres via brugergrænsefladen, men også over TCP netværket. Vi har valgt at færdigimplementere det manglende kode i Java, så det stemte overens med det udleverede skelet. Ud fra den udleverede opgavebeskrivelse har vi specificeret kravene, så funktionaliteten i programmet opfylder det forventede resultat. Rapporten vil i sin helhed gennemgå overvejelser og argumentation for vores færdig implementerede kode, som skal give en forståelse for læser omkring tanker og idéer bag implementationen.

Kravspecifikationer

Funktionalitet

F1. Ved tryk på 'Exit' skal vægten slukkes (ved simulator skal GUI'en lukkes ned)
F2. Ved tryk på 'Zero' skal vægten nulstilles.
F3. Ved tryk på 'Tara' skal vægten tareres.
F4. Ved tryk på '[->' skal nuværende besked sendes til observere
F5. S: Send stabil afvejning
F6. T: Tarér vægt
F7. D: Skriv i vægtens display
F8. DW: Slet vægtens display
F9. P111 Skriv max. 30 tegn i sekundært display
F10. RM208 Skriv i display, afvent indtastning fra bruger.
F11. K. Skift vægtens knap tilstand.
F12. B: sæt ny bruttovægt
F13. Q afslut simulering (slukker vægt)
F14. Programmet skal simulere Mettler BKK vægten.
F15. Subjekt (vægten) skal kunne notificere observers over en socket
F16. Programmet skal lytte på port 8000.
F17. Vægten skal kunne modtage input fra bruger og samtidig modtage kommandoer over netværket.

Usability

F18. En bruger skal kunne interagere med systemet igennem en vægt og simulator
F19. Input skal foregå via vægt (eller simulator)

Reliability

F20. Alle ressourcer skal lukke korrekt ned når programmet afsluttes.

Implementation constraints

NF1. Programmet skal skrives i Java

Konfigurationsstyring

System specifikationer

Windows

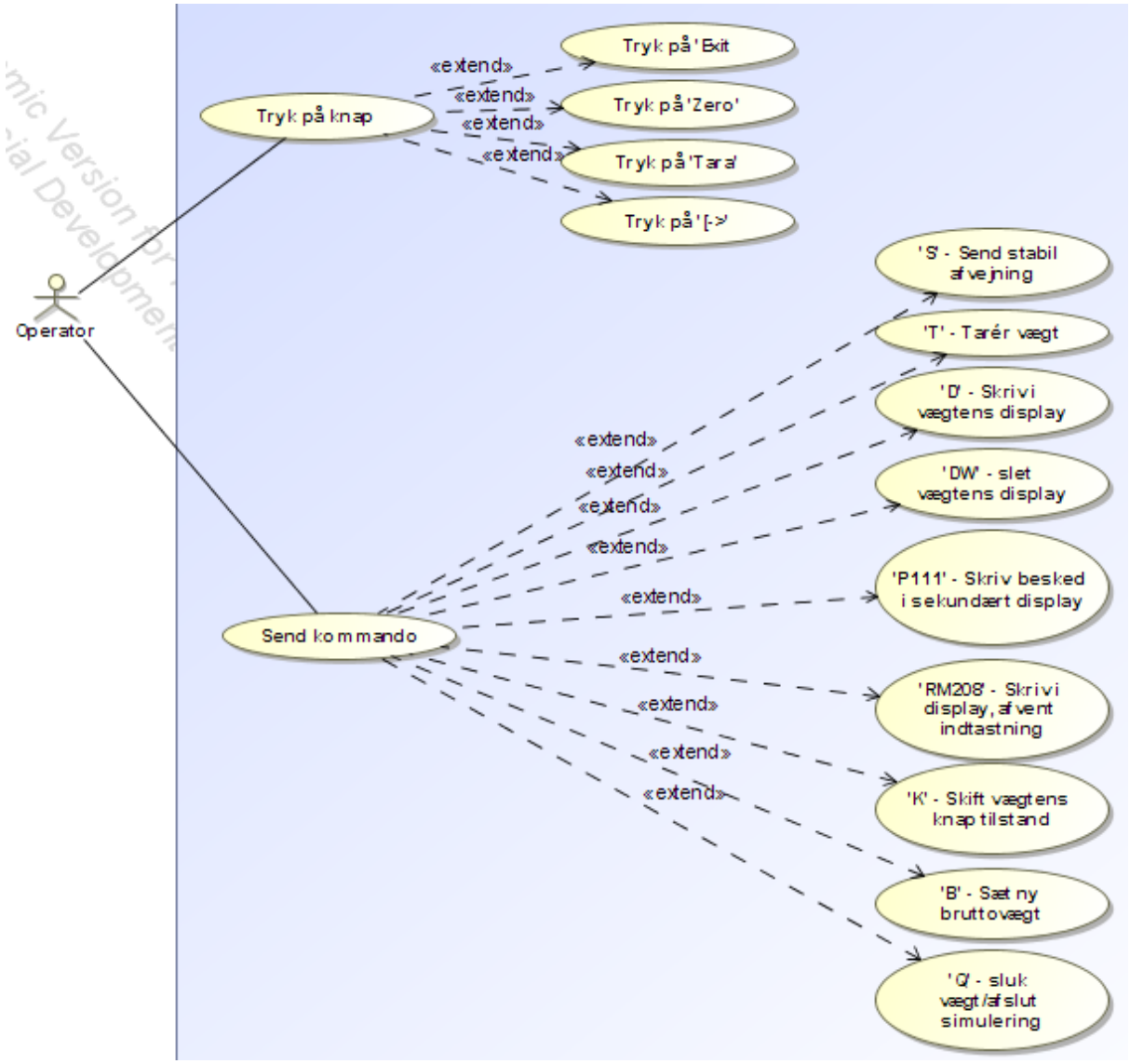
- Version 7 eller nyere.

Vejledning til import af Git-repository i Eclipse

1. Åben din browser
2. Gå til denne webside https://github.com/Gruppe25DTU/CDIO2
3. Kopier klonings URL til højre (https://github.com/Gruppe25DTU/CDIO2.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 diagram



På ovenstående diagram ses de aktioner som operatoren kan foretage. Operatoren har kan enten trykke på en knap på vægten eller sende en kommando over netværket. En eksempel på et knap-tryk kan være at operatoren er færdig med at bruge vægten og derfor trykker på 'Exit', hvorefter at vægten slukkes. Ligeledes kan Operatoren sende kommandoen 'Q' for at slukke vægten.

Use case beskrivelser

Følgende use cases er bygget på ovenstående use case diagram. Beskrivelserne giver gruppen en dybere forståelse af hvordan de forskellige scenarioer udfolder sig, samt hvilke skridt systemet skal tage i disse scenarioer. Beskrivelserne kan bruges til at foretage yderligere evalueringer om processen, samt om det er logisk at foretage grupperinger af disse handlinger på baggrund af GARSP princippet.

Use Case Name:	Tryk på 'Exit'
ID:	01
Actors	Operator
Pre Condition	Vægt er tændt
Post Condition	Vægt er slukket
Flow	Operator trykker på 'Exit' Vægt(/simulator) slukkes
Alternative Flow	
Exceptional Flow	

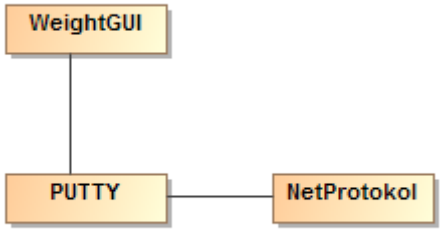
Use Case Name:	Tryk på 'Zero'
ID:	02
Actors	Operator
Pre Condition	Vægt er tændt
Post Condition	Vægten i displayet er 0.
Flow	Operator trykker på 'Zero' Displayet viser 0 kg
Alternative Flow	
Exceptional Flow	

Use Case Name:	Tryk på ’Tara’
ID:	03
Actors	Operator
Pre Condition	Vægt er tændt.
Post Condition	Vægten i displayet er 0.
Flow	Vægt på display er større end 0. Operator trykker på ’Tara’ Displayet viser 0 kg
Alternative Flow	Vægt på display er 0. Operator trykker på ’Tara’ Displayet viser 0 kg
Exceptional Flow	

Use Case Name:	Tryk på ’[->’
ID:	04
Actors	Operator
Pre Condition	Vægt er tændt.
Post Condition	Besked er sendt til netværket.
Flow	Operator indtaster besked i display Operator trykker på ’[->’ Vægt sender besked til netværket.
Alternative Flow	
Exceptional Flow	
Use Case Name:	Send kommando ’S’
ID:	05
Actors	Operator
Pre Condition	Vægt er tændt.
Post Condition	Kommando udført
Flow	Operator indtaster kommando ’S’ 1 Operator trykker på ’[->’

	2 Vægt sender nettovægten til netværket
Alternative Flow	Operator indtaster kommando 'T' 1. Operator trykker på '[->' 2. Vægt Tareres. Operator indtaster kommando 'D' 1 Operator trykker på '[->' 2 Operator skriver i vægtens display. Operator indtaster kommando 'DW' 1 Operator trykker på '[->' 2 Vægt sletter tekst i displayet Operator indtaster kommando 'P111' 1 Operator trykker på '[->' 2 Operator skriver max 30. tegn i sekundært display Operator indtaster kommando 'RM20 8' 1 Operator trykker på '[->' 2 Vægt skriver i display og afventer indtastning fra operator. Operator indtaster kommando 'K' 1 Operator trykker på '[->' 2 Vægtens knap-tilstand skiftes. Operator indtaster kommando 'B' 1 Operator trykker på '[->' 2 Sætter bruttovægt (i simulator) til det indtastede beløb Operator indtaster kommando 'Q' 1 Operator trykker på '[->' 2 Vægt slukkes.
Exceptional Flow	

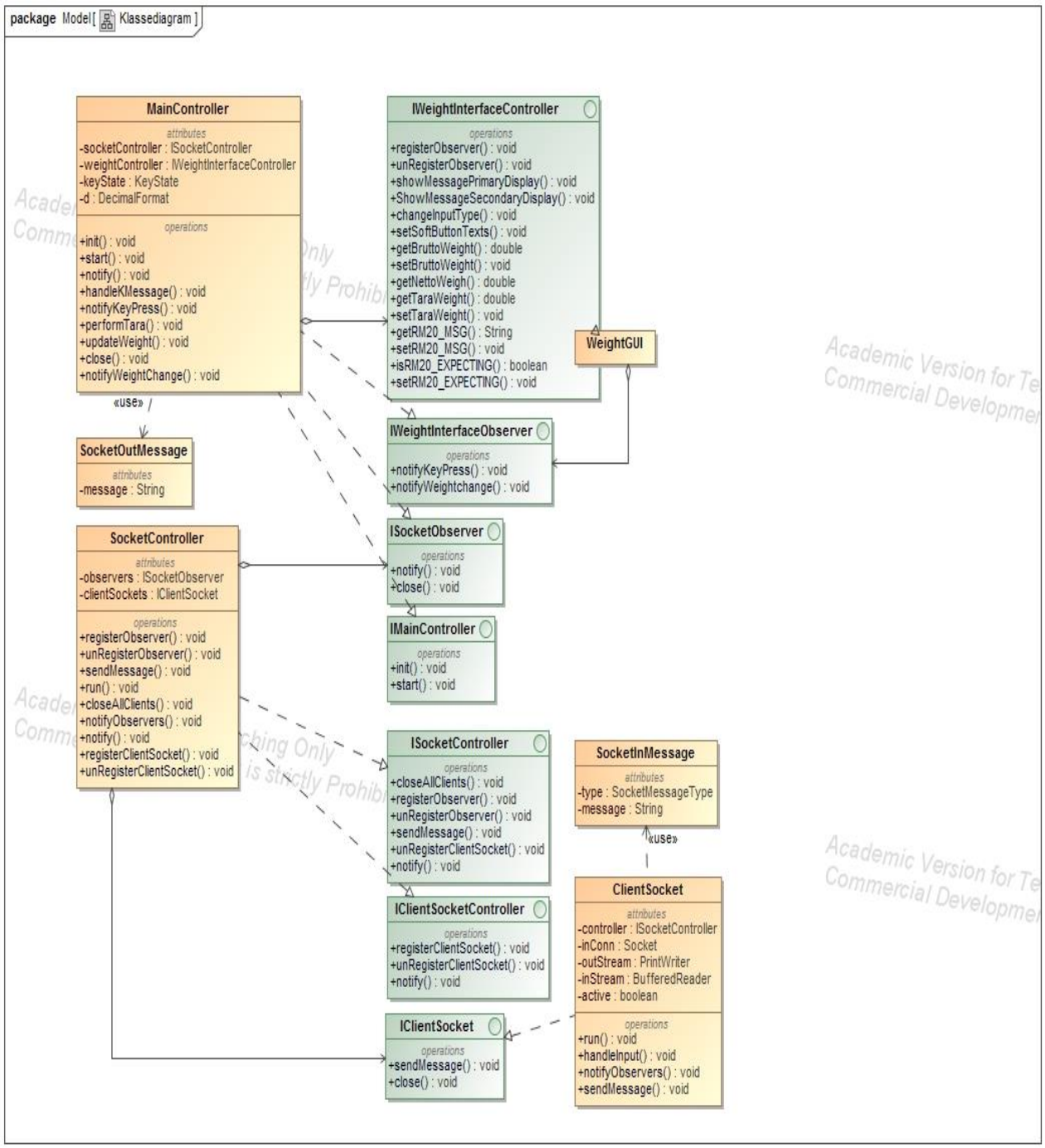
Domænemodel



Målet med denne domænemodel er at få et overblik over de elementer, som kommer til at indgå i vores projekt. Den følgende model indeholder ikke nogle detaljer om implementationen af vores program, modellen viser groft den overordnet struktur af hvordan programmet kommer til at se ud. Denne Domænemodel er meget simpel da de program vi lave ikke er så kompliceret, her er den vægt vi bruger en Vægt-GUI, den måde vi kan kommunikere med vores vægt er igennem en PuTTY, hvor vi også har nogle kommandoer vi kalder for Netværksprotokoller.

Design

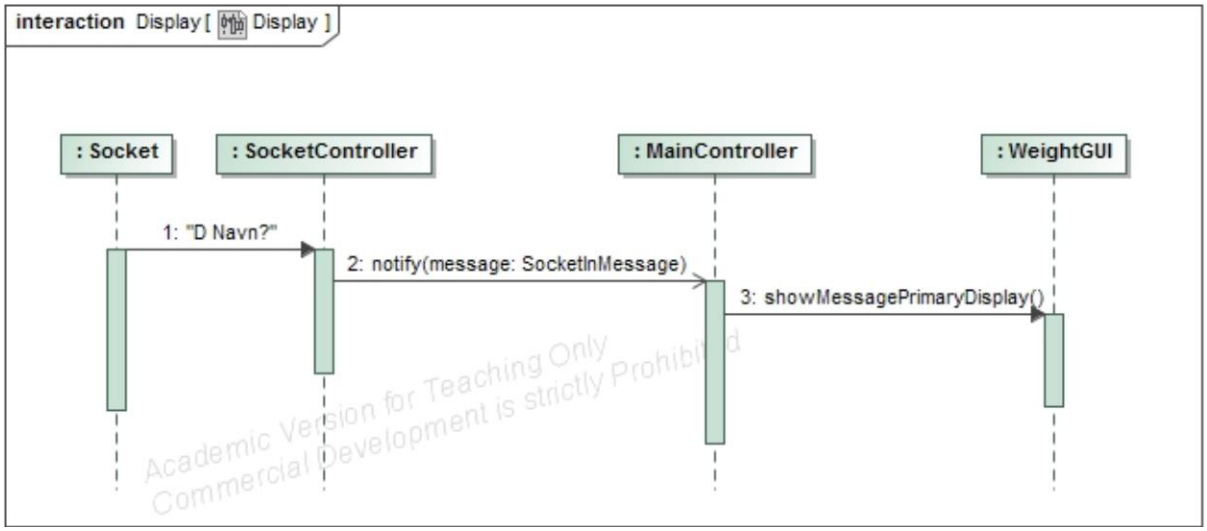
Klassediagram



I modsætning til en domænemodel, har vi her lavet et klassediagram der indeholder detaljer om de enkelte klasser. I kan se her at vi har en ISocketController, der holder øje med kommunikation på programmets socket og IWeightInterfaceController, der holder øje med ændringer i den simulerede brugergrænseflade til vægten. MainControlleren implementerer både ISocketObserver, IWeightInterfaceObserver og IMainController og kan derfor registrere sig hos de to controllers og modtage beskeder når der sker ændringer på socket'en og på brugergrænsefladen. Da MainControlleren både har en instans af

ISocketObserver og IWeightInterfaceObserver, skal registrere sig selv hos de to controllers og herefter starte de to controllers i hver deres tråd. Når de to controllers herefter har nyt input, notify'er de MainControlleren.

Design sekvensdiagram



Et Design sekvensdiagram (DSD) er et diagram der viser hvordan kommunikationen fungerer i systemet, vi har lavet nogle DSD'er over Netværksprotokollerne til dette system, og valgt at beskrive et af dem. Dette diagram viser hvad der sker i Netværksprotokol - D, med denne protokol skal vægten udskrive den sendte besked i displayet og svarer med en bekræftelse. Som vi kan se på diagrammet bliver der sendt en besked fra en Socket til en SocketController, her vil besked blive kontrolleret og sendt videre til MainController som derefter vil vise beskeden i vores WeightGUI.
(DSD for resten af kommandoerne kan ses i bilag)

Implementation

Observer Pattern

Software objekt mønsteret ”observer pattern” er et mønster der består af at subjekt og observers. Disse ’observers’ er afhængige af subjektet, idet at subjektet notificerer dem om nogle ændringer i dens tilstand. Subjektet notificerer normalt sine observers ved et kald af deres metoder. Observer mønstret er i dette program blevet brugt til, at distribuere information fra vægten til de observers der har tilsluttet sig. Dette betyder at når der foretages en ændring i vægtens tilstand i form af tryk på knap eller tarering, så vil vægten automatisk informerer observers’ne. Dette gør at observers’ne ikke skal sende en forespørgsel til vægten, for at vide om at tilstandsændring har fundet sted. I selve programmet fungerer klassen MainController som en observer til vægten. Det er i denne klasse at logikken bag vægten foregår, samt opdatering af vægtens display

Interfaces

I dette projekt fungerer interface’sne som døren til et større netværk. De bruges bl.a. til at koble observere til subjektets observer liste, samt at koble 2 forskellige stykker kode sammen. Her GUI’ens funktioner samt kommunikation mellem de forskellige pakker. Dette gør at koden kan ændres i en klasse uden at have effekt på andre klasser. Brugen af interfaces har gjort at klasser ikke importerer andre klasser, men implementere disse klassers interface.

Thread

En thread er en selvstændig “sti” af udførslen af noget kode. Når flere threads udfør en threads “sti” gennem samme stykke kode vil de adskille sig fra hinanden. Som et eksempel kan vi bruge en if-else sætning, her vil en thread udføre if delen imens den anden thread vil udføre else delen. Når flere threads udføre byte-kode sekvens i samme program, kaldes det for multiplethreading. Grunden til at vi i dette program har brugt multiplethreading, er for at få et bedre og hurtigere program. Et multiplethreaded GUI program er en meget godt, fordi at når GUI’en er multiplethreaded kan den modtage beskeder imens den udføre en anden besked den har fået. Når et program er threaded kan det også køre programmet hurtigere en et der ikke er threaded.
Vi har to threads i vores program, som kører indtil programmet slutter, her vil den ene thread konstant lytte efter inputs fra Gui’en (Når man trykker på en knappe) og derefter sende disse beskeder videre til vores main controller og så videre til socket laget. Vores anden thread lytter konstant efter nye forbindelser nede i Socket controlleren. når den så danner en forbindelse, bliver denne forbindelse knyttet til et nyt socket objekt, som lægges i et nyt ClientSocket objekt. derefter startes ClientSocket’s run metode i sin egen thread, hvor den udelukkende lytter efter inputs fra sin socket, og derefter sender kommandoen (hvis det er en gyldig kommando) videre op i lagene, op til Gui’en. Da flere ClientSockets kan sender deres kommandoer op på samme tid, og de alle benytter den samme metode notify() i socketControlleren, gjorde vi denne metode synchronized, det betyder at kun en thread kan benytte metoden ad gangen.

Socket

Socket er et endepunkt af en to vejs kommunikations link mellem to programmer der køre på et netværk. En socket klasse bliver brugt til at repræsentere forbindelsen mellem et klient program og et server

program. I dette program bliver sockets brugt til at repræsentere forbindelsen mellem en vægt og en bruger. Da en socket er en 2 vejs kommunikation, har det været er problem at tilføje flere observers, da når en observer har koblet sig på socketen blokerer den forbindelsen. Dette betyder at kun denne observer kan sende kommandoer til vægten over netværket, mens at andre observere, der har koblet sig på socketen senere, ikke kan. Hertil har vi etableret en kø, hvor de obsevere der kobler sig på socketen, bliver sat ind. Dette har medført at de observere, der har deres forbindelse blokeret, kan modtage notificeringer om en tilstand skift i vægten.

Undtagelser

Undtagelser (Exceptions) er defineret som event der forekommer under kørsel af kode. Et eksempel er division med 0. Undtagelser bruges ikke kun steder hvor der kan opstå aritmetiske undtagelser, men alle steder hvor fejl kan opstå. Her bruges udtrykket ”try-catch” til at indfange disse undtagelser, så snart de opstår. Undtagelserne kan herfra blive håndteret ud fra den grad af alvorlighed de har. I dette projekt bruger vi exceptions, når vi skriver til netværket eller læser noget fra socket’en. Dette medfører, at hvis der skulle være en fejl i den meddelelse der modtages, så går programmet ikke ned.

Concurrency

Concurrency er et udtryk, der beskriver et programmet der kan foretage flere ting på samme tid. Det vil sige at programmet samtidig kan modtage kommandoer fra netværket og udføre disse, samt modtage input fra et keyboard. Det er i dette program blevet brugt i vægten. Denne kan samtidig modtage input fra netværket og input fra knapperne på vægten. I denne sammenhæng bruges synkronisering til at dele adgangen til input fra vægten mellem de observere der er. I programmet bruges disse synkroniserede metoder alle steder hvor client-threads kan kommunikere med klassen SocketController’s thread. Samtidig er socketcontorllerens to HashSet wrapped i et syncronized Set. Dvs. uanset hvor mange clientsocket-threads der er i baggrunden, så er det kun én som kan snakke med SocketControlleren af gangen og alt der sker derefter kunne ligeså vel antage er der kun er én socketthread.
Problemet med synkronisering ligger i at, hvis clientsocket-thread optager SocketController.notify() indtil kommandoen er udført, så kan alle de andre clientsockets ikke give input. En løsning til dette have været at bruge en synkroniseret kø, som SocketControlleren kan gemme alle kommandoer i, og sende dem til MainControlleren i samme rækkefølge. Dette medfører at clientSockets kun optager og blokerer andre clientSockets i den tid det tager at gemme kommandoen i den synkroniserede kø.
Da både SocketControlleren og alle clientSockets synkroniserede funktioner er entrådede, behøver vi ikke at teste concurrency. Havde dette været tilfældet, skulle et framework som multithreadedTDC være anvendt.

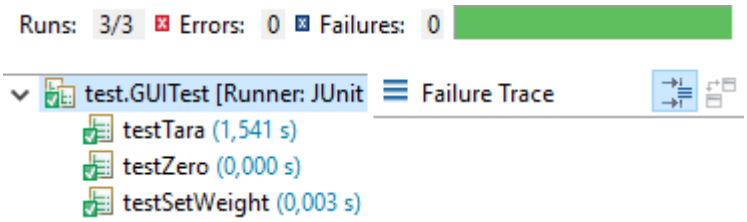
Test

JUnit.

Vi har testet programmet via JUnit tests for at se om kommandoerne gør som de skal. Vi har hertil oprettet 3 JUnit test cases. GUITest, SingleThreadTest, og WeightTest. Disse 3 cases tester henholdsvis GUI'en, den thread vi har oprettet samt vægten.

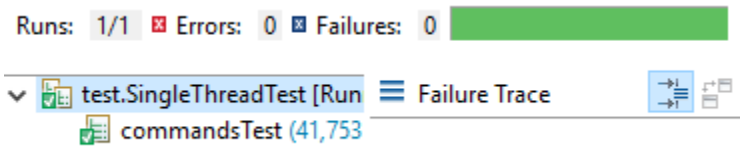
GUITest

GUITest indeholder funktionerne testSetWeight, testTara, og testZero. Disse 3 tester hver, som navnet fortæller, om GUI'en funktioner virker som planlagt. Som set på billedet til højre fungere disse funktioner optimalt.



SingleThreadTest

SingleThreadTest tester de forskellige kommandoer, som vægten kan udføre. Herunder P111, RM20, DW, RM20Answer, B, D, S, T. Disse test bekræfter at funktionerne er implementeret rigtigt i programmet. Som det ses på billedet herunder fungere alle funktionerne optimalt.



WeightTest

Denne case opretter objekter som de 2 foregående cases bruger. Casen opretter en socket på port 8000 og IP: 127.0.0.1 (localhost). Den lukker samtidig forbindelsen igen når testene er ovre.

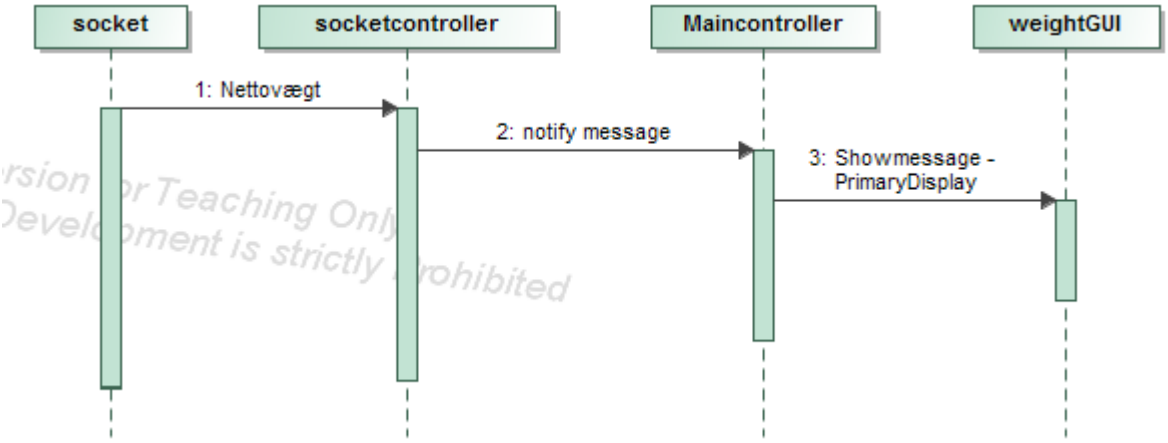
Konklusion

Vi har færdigimplementeret vægt simulationsprogrammet så funktionaliteten stemmer overens med de ønskede krav. Resultatet er blevet et funktionsdygtigt og solidt program, som opfylder de krav opgavebeskrivelsen stillede.

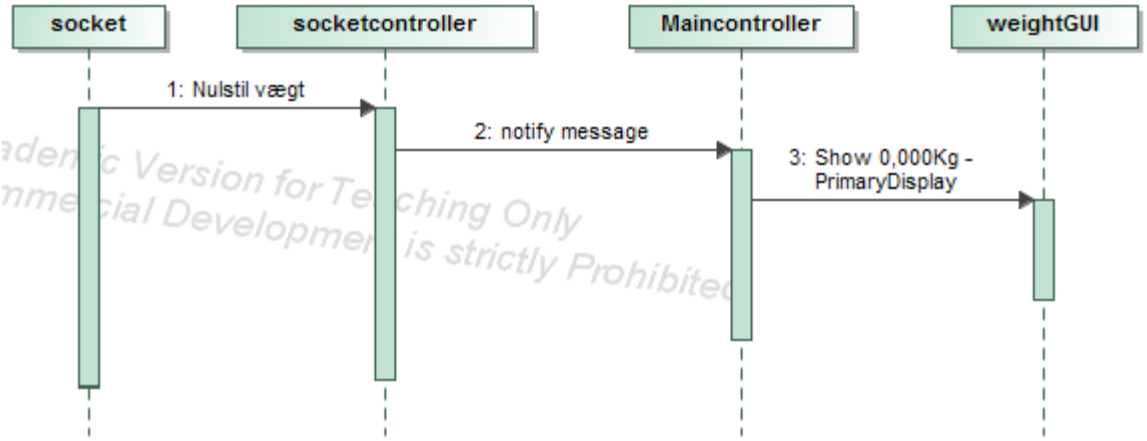
Bilag:

DSD

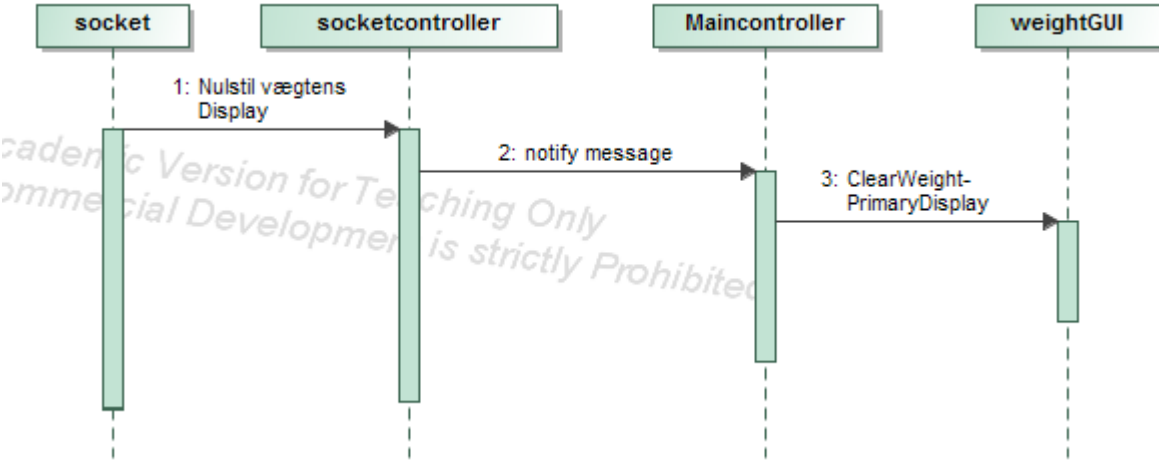
Kommando: 's'



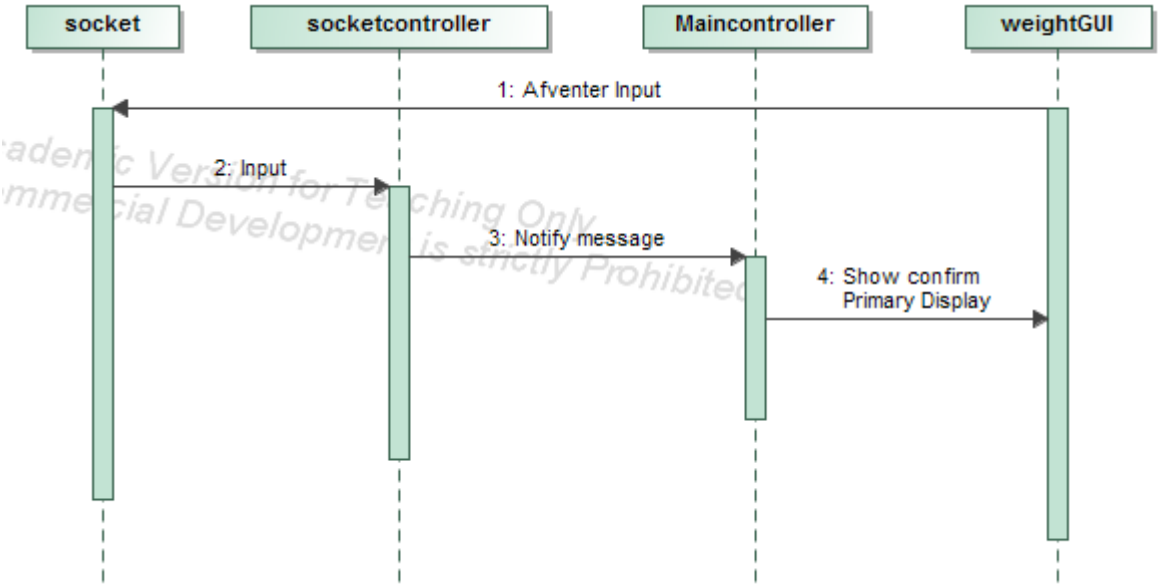
Kommando: 'T'



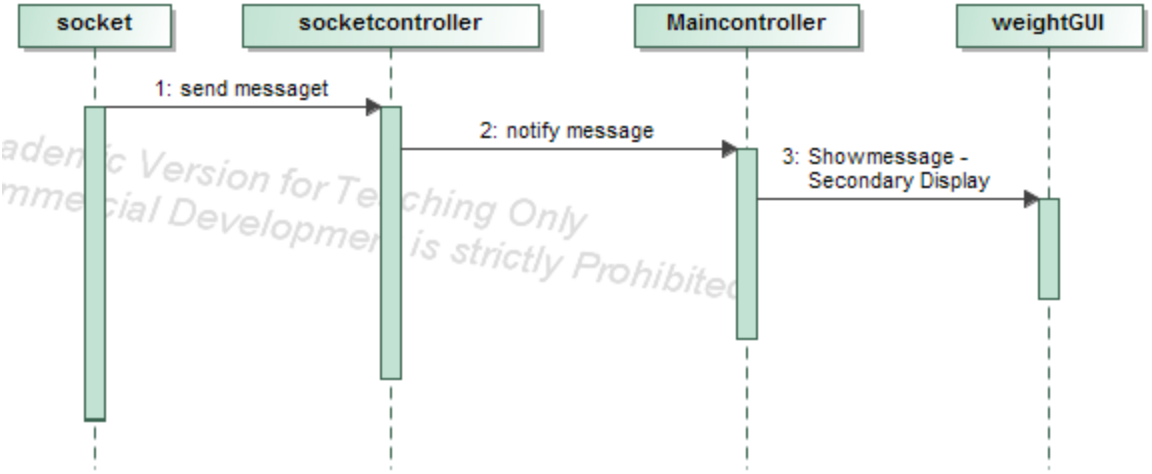
Kommando: 'DW'



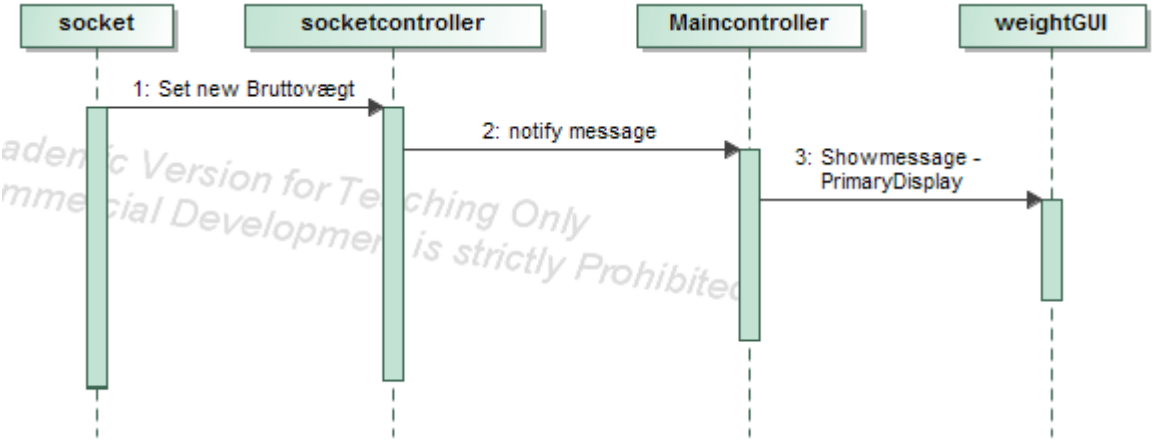
Kommando: 'RM20 8'



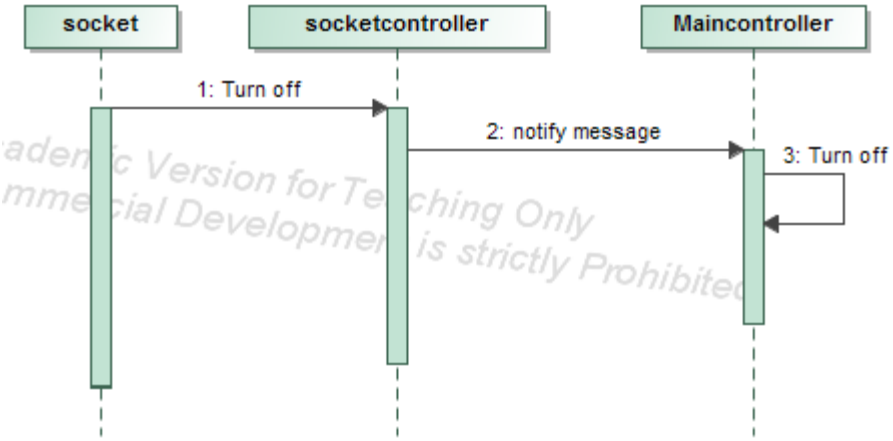
Kommando: 'P111'



Kommando: 'B'



Kommando: 'Q'



Kommando: 'K'