**F21 - ST4SWT**

**Obligatorisk handin 2**

**Chargingstation**

**Afleveret:** 25-03-2021

**Afleveret af**: Gruppe 14

**Deltagere i afleveringen**

|  |  |  |
| --- | --- | --- |
| Studienummer | Navn | Studieretning |
| 201907690 | Lasse Gjetrup Eskildsen | ST |
| 201611717 | Jacob Jungersen Hansen | ST |
| 201905378 | Signe Gram Sand | ST |
| 201908197 | Lars Mølgaard Pedersen | ST |

**Kontaktperson**

|  |  |  |  |
| --- | --- | --- | --- |
| Studienummer | Navn | E-mail | Studieretning |
| 201908197 | Lars Pedersen | 201908197@post.au.dk | ST |

**URL Jenkins:**

<http://ci3.ase.au.dk:8080/job/Team14ObligatoriskOpgave2/>

**URL GitHub:**

<https://github.com/JacobJH/LadeskabTeam14>

Indhold

[Arkitektur og design: 1](#_Toc67568365)

[Arkitektur for chargingstation: 1](#_Toc67568366)

[Design for chargingsystem: 2](#_Toc67568367)

[Overvejelser ift. arkitektur og design: 4](#_Toc67568368)

[Strukturering af github: 4](#_Toc67568369)

[Opdeling af ansvar: 5](#_Toc67568370)

[Refleksion over arbejdsfordeling: 5](#_Toc67568371)

[Refleksion over brug af Jenkins og continuos integration: 6](#_Toc67568372)

# 

# Arkitektur og design:

## Arkitektur for chargingstation:

Vi har tager udgangspunkt i sekvensdiagrammet og statemachinen til at identificere de nødvendige blokke i vores system. Ud fra sekvensdiagrammet har vi også kunne undersøge hvilken relation der vil være passende imellem de forskellige blokke.

Nedenfor ses det udarbejdede klassediagram som beskriver alle blokke og deres relation til andre blokke. Allerede her har vi valgt at planlægge hvilke blokke der med fordel kunne implementere et interface - her tænkes specielt på de klasser som skal håndtere funktionalitet for hardware, da dette ikke vil være tilgængeligt og derfor skal simuleres.



Figur 1 Arkitetur for chargingstation system

## Design for chargingsystem:

Ud fra det udarbejde klassediagram i arkitekturfasen har vi lavet et færdigt klassediagram som beskriver designet for vores system. Vi har anvendt specielt sekvensdiagrammet til at sikre at alle de nødvendige metoder fremgår og derudover har vi designet systemet således at vi kan kommunikere imellem boundary og controller-laget med event. Her har vi brugt figur 2: Designskitse fra opgavebeskrivelsen samt vores klassediagram fra arkitekturfasen til at fastlægge hvilke events der vil være nødvendige.

Stationkontrol skal kende til statussen for hvorvidt USBChargeSimulator er tilkoblet med en mobil. For at holde koblingen lav imellem klasserne så vælger vi at oprette en metode i ChargingContol som kan holde styr på statussen af USBChargingSimulator og derved bliver denne information tilgængelig for Stationcontrol. 

Figur 2 Klassediagram for Chargingstation systemet

## Overvejelser ift. arkitektur og design:

* Der er implementeret interfaces til boundary klasserne, som repræsenterer den manglende hardware. Dette gør det muligt at anvende fakes, og derudover giver det mulighed for fremtidigt at kunne implementere hardware-klasser
* Programmets arkitektur vil blive bygget op i 3 lag, efter boundary control entity (BCE) modellen. Dvs et lag til boundary-klasser, et lag til controller klasser og et lag til entities. Dette gøres for at sikre et naturligt flow af data samt give et godt udgangspunkt for at kunne designe systemet med en lav kobling. Den lave kobling vil give en stor fordel når det kommer til testbarheden af systemet.
* Vi designer systemet så der indgår et Observer Pattern og til dette anvendes der events. Det vil sikre en løs kobling imellem klasserne.

Vi anvender events følgende steder:

* + Håndtering af lukning samt åbning af døren
  + Håndtering af RFID-læseren
  + Udmelding omkring strømtrækniveauet på telefonen

Boundary analyse:

For at kunne foretage en ordentlig test som dækker alle tænkelige scenarier for metoden NewChargeHandler() laves der en boundary analyse. Vi gruppere alle inputs, så der kan dannes ækvivalent klasser hvori alle input vil resultere i det samme output. Mellem hver af disse ækvivalentklasser findes der nu boundary values. Disse kan bruges til at bestemme hvilke værdier vi bør teste på. Derved er det heller ikke nødvendigt at teste på samtlige værdier i hver ækvivalent klasse da det som udgangspunkt er nok at teste på værdierne tæt på de kendte boundary values.

Fordi vi vælger kun at kigge på input og output er dette en god måde at sikre at testene der bliver lavet er black box test og ikke whitebox test.

# Strukturering af github:

Da hele systemet bliver forbundet til en Jenkins server, har vi valgt at arbejde med branches når vi implementere ”nye features”. Det betyder i praksis at når vi starter en ny feature branch hver gang vi starter en ny dags arbejde / lektion. Det betyder at vi ikke behøver at få alt uploadet på Jenkins før det er skrevet færdigt og er testet. Når vores feature branch er færdig eller når dagen er slut, vil vi merge branchen med vores Main-branch, og derved få Jenkins til at køre Coverage test, interaktionstest og unit test på hele systemet.

Fordelen ved dette er at vi kun sender kode til Jenkins Serveren som allerede er testet. Da vi sidder 4 mennesker og arbejder i GitHub kan det nogle gange være nødvendigt at gemme noget kode som ikke helt er færdigt, hvis de andre skal kunne se det. Det vil dog være unødvendigt at få dette op på jenkinsserveren, da det ikke nødvendigvis er testet færdig. Derfor fungere det godt at have andre branches at arbejde på.

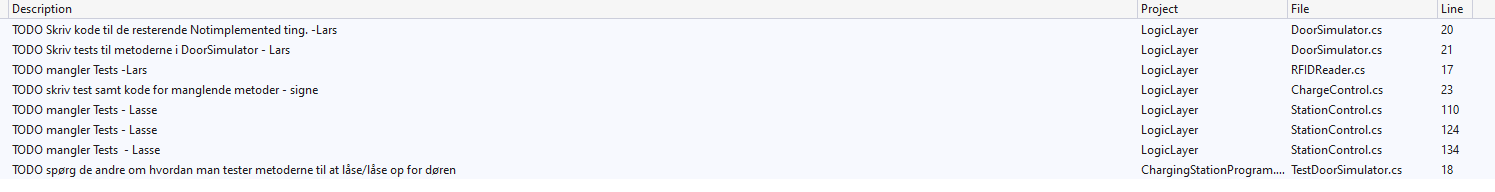
# Opdeling af ansvar:

Udgangspunktet for uddelegeringen af ansvaret bygger på et ønske om at skrive test først og derefter kode klasserne. Derfor vælges det at uddele hver boundary klasse og dertilhørende test og interface til et gruppemedlem. På den måde får alle mulighed for at skrive test til en klasse, som de derefter kan implementere. Efter dette vil vi sammen skrive testene til kontroller klasserne, da disse har behov for interaktionstest såvel som State baseret test, og det vil derfor give god mening at lave disse når boundary klasserne og deres interfaces er lavet.

Fordelen ved at dele opgaverne ud på denne måde er at det sikres at vi ikke ved et uheld kommer til at skrive kode oveni hinanden og derved har lavet dobbeltarbejde. Derudover vil det gøre det lettere at skrive klasserne, hvis man selv har skrevet testen som hører til, i modsætning til hvis man skulle implementere klasser ud fra test som man ikke selv har skrevet.

Vi ønsker at øve os i testdriven Development og derfor er vores vigtigste prioritet at testene skrives først og koden derefter. Denne måde at opleve arbejdet på sikrer at det er muligt.

Begrænsningerne ved denne måde er blandt andet at nogle af gruppemedlemmerne muligvis vil ende med at større arbejde end andre, fordi de enten koder hurtigere eller har fået mindre klasser som ansvarsområder.

Ud over at uddele opgaverne som ovennævnt, så har vi også anvendt //TO DO funktionen i Visual Studio. Derved har vi sammen kunne planlægge hvem der skal lave hvilke opgaver og på den måde har vi også kunne holde styr på hvor langt alle andre er med deres opgaver. 

Figur 3 Eksempel på brugen af // TO DO for en given arbejdsperiode

# Refleksion over arbejdsfordeling:

Opdelingen af arbejdet, så alle havde ansvaret for deres egen boundary-klasse var meget god, da dette betød at vi kunne arbejde forskudt når vi hver især havde tid. Dette har gjort det meget effektivt at kode, og det har været nemt at holde styr på hvad man selv har lavet og mangler at lave. Dette betød dog også at nogen havde mere arbejde end andre, da tests af visse klasser var mere omfattende end andre, men forskelligheden i arbejdsbyrde var ikke meget urimelig.

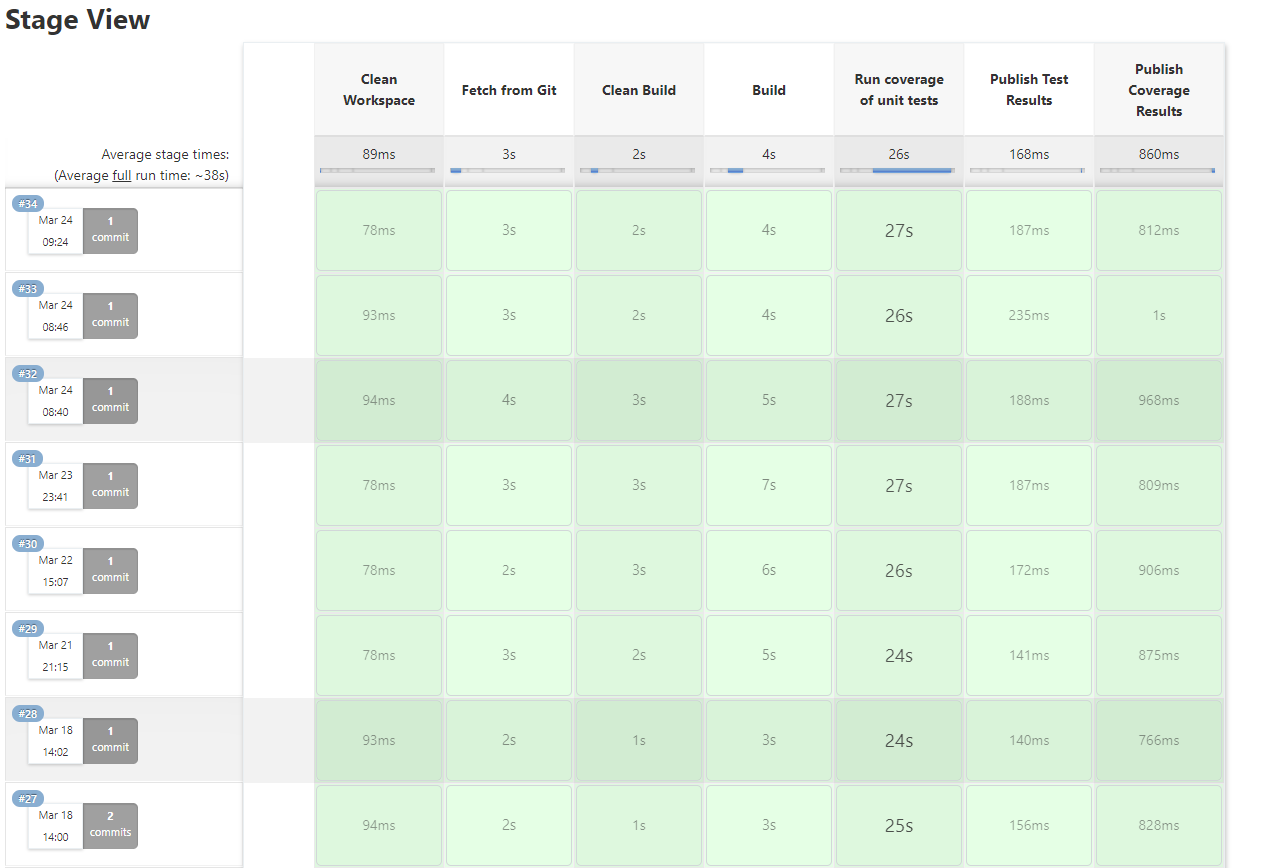
Ved at bruge //TO DO funktionen hjalp det os med at sikre at alle nåede i mål med deres opgaver, men det er stadigvæk en udfordring med denne arbejdsfordeling af sikre at alt er lavet. Man kunne derfor til en anden gang overveje om der måske skulle være en som havde det overordnede ansvar for at holde styr på at alle dele af systemet var blevet implementeret.

En anden ulempe ved dette system er, at hvis der er fejl og mangler fra nogens side, så er det svært at opdage, da vi hver især kun har kendskab til det vi selv havde lavet. En anden gang kunne vi gøre således at man fik ansvaret for at lave sin egen klasse/test og derefter tjekke om det en andens ansvarsområde er lavet acceptabelt.

# Refleksion over brug af Jenkins og continuos integration:

Det har generelt fungeret godt for os at bruge Jenkins og Continues integration. Det har gjort det meget overskueligt at se om der er fejl i vores test. I den forbindelse så har vores opdelingen i branches for hver iteration været rigtig smarte, og det har gjort at vores Jenkins Server har givet et pænt overblik og ikke har medtaget alle commits undervejs hvor koden ikke har været helt færdig.

Som det ses på nedenstående figur så har vi generelt ikke haft de store probelmer med at testene fejler, hvilket skyldes at vi kun laver commits til Main branche når vi ved at koden fungere.



Figur 4 Udklip fra Jenkins