ST4SWT-02 Software test - Mobile Charging Station

Afleveret af gruppe 16:
Nicolas Warrer - 201703321
Søren Paulsen - SP93586
Frederik Laursen - 202004530

25. Marts 2022

Contents

1 GitHub og Jenkins

GitHub:
`https://github.com/FrederikLaursenSW/SWT_Assign2`

Jenkins:
`http://ci3.ase.au.dk:8080/job/team16F22_SWT_assignment2/`

2 System beskrivelse

Systemet består af 2 kontrol klasser ChargeControl og StationControl. StationControl er den overordnet kontrol klasse, som indeholder en chargeController. Til disse 2 kontrol klasser hører der 3 komponent klasser. Door, RfidReader og UsbChargeSimulator, hvor UsbChargeSimulator er en simulator klasse. Door giver besked til stationControl om døren er åben eller lukket via. events. RfidReader sender besked om den scannet Rfid værdi til stationControl via events. UsbChargeSimulator simulerer en opladning og sender den nuværende current værdi til ChargeControl. ChargeControl styrer selve opladningen og kan udfra current værdien sende forskellige beskeder ud på displayet.

3 Software design

På figur 1 ses et sekvensdiagram, for den del af systemet, der ikke er beskrevet vha. det udleverede sekvensdiagram. Her ses det, at når StationControl kalder StartCharge() til en ChargeController, så kaldes en StartCharge() funktion til en UsbChargeSimulator. UsbChargeSimulatoren starter en timer og kalder en eventhandler, der invoker et CurrentValueEvent. Idet controlleren modtager et CurrentValueEvent, så kaldes controllerens HandleChargingEvent(), der kalder CurrentChanged(). Herefter evalueres den modtagede current værdi, og der laves Console.Writelines i hver af de 3 alternativer.

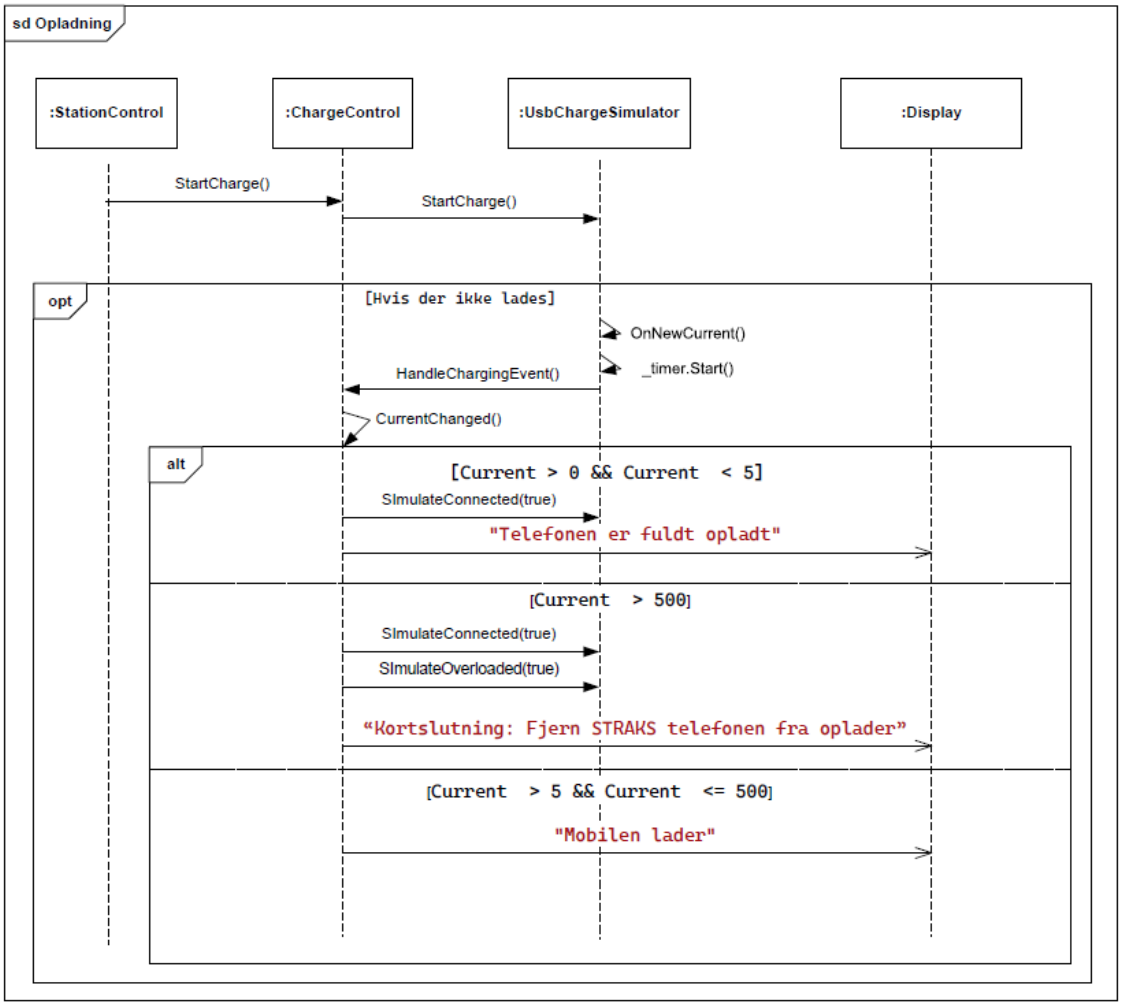

Fig. 1: *Diagram: Sekvensdiagram ChargeControls opladnings*

På figur 2 ses systemets klassediagram. Diagrammet kan ses tydeligere i den visio fil, der er vedlagt som bilag. Det ses på diagrammet, at der anvendes interfaces til komponentklasserne Door, UsbChargerSimulator og RfidReaderen, så der kan anvendes fakes til at teste.

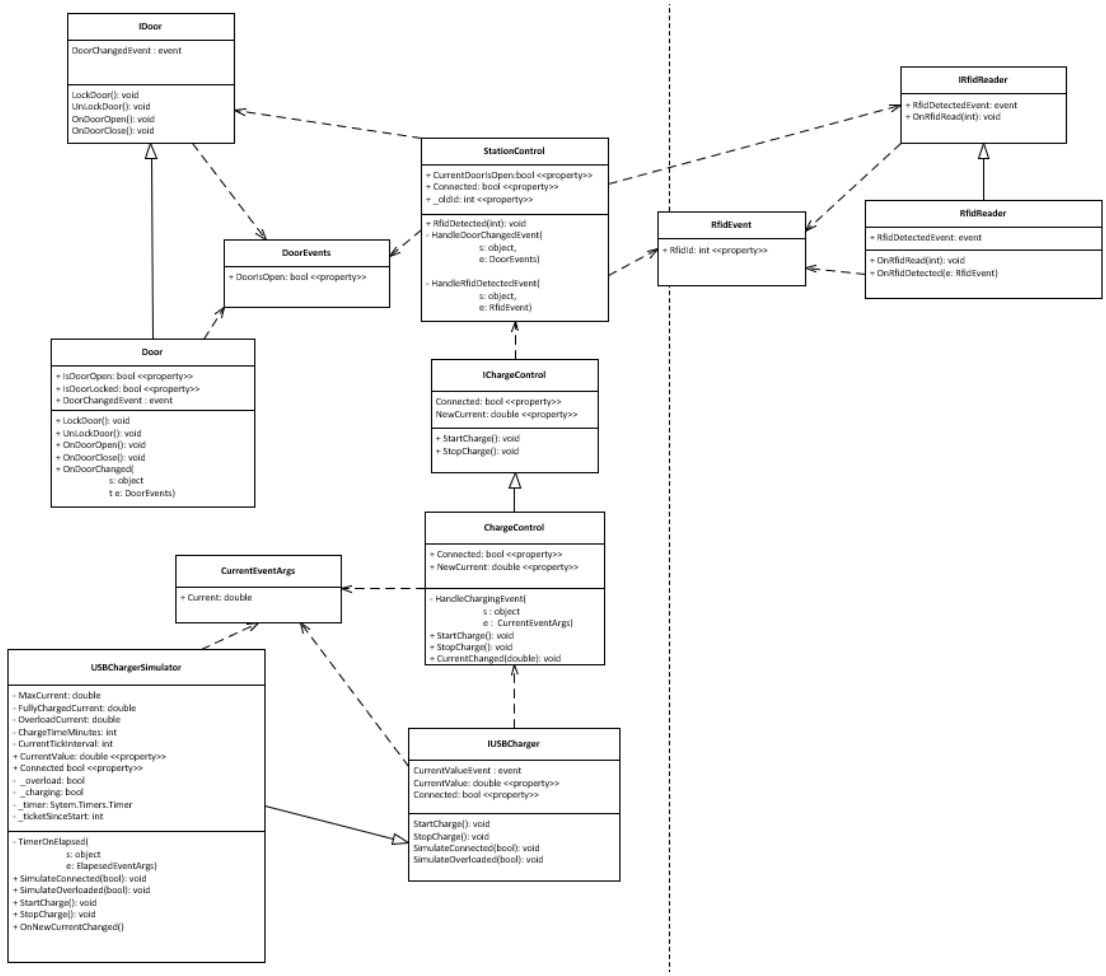

Fig. 2: *Diagram: Klassediagram ChargeControl*

4 Designreflektioner

Gruppen har udarbejdet designet ud fra det materiale der udarbejdet til assignment 2, samt de design principper der anvendes når der skal designes "for testabilty". Her tænkes på lav kobling og single responsibility. Igennem hele designfasen har gruppen haft fokus på muligheden for at de-koble klasser, og dermed bibeholde lav kobling. Dette er sket ved anvendelse af interfaces, dependency injektion og events. Høj grad af de-kobling i designet, gør det muligt at kontrollere afhængigheder under test. Det betyder, at der under test kan anvendes fakes i stedet for den "oprindelige" klasse. For at sikre lav kobling, har gruppen generelt udarbejdet designet efter Triple-I (Identify, Interface, inject), og jævnligt introduceret interfaces for at kunne kontrollere afhængigheder og dermed isolere den del af koden der ønskes testet. Derudover har gruppen anvendt dependency injection. Ved anvendelse af dependency injection, kan en given klasse nemt erstattes med en fake, uden der skal ændres i det kode der ønskes testet.

5 Test af klasser

Rapporten indeholder en beskrivelse af hvordan I delte arbejdet med klasser og deres test og hvorfor I gjorde det på den måde

I vores solution har vi anvendt 3 projekter, der hver især har et klart ansvarsområde. I SWT_Assign2 projektet ligger det udleverede program, der har til formål simulere en simpel console-app til en bruger. I LadeskabsTest projektet, der er et NUnit test-projekt, ligger unit-tests af de 2 controllers ChargeControl og StationControl. Yderligere ligger der i dette projekt test af hver af de 3 komponenter (Door, RfidReader og UsbCharger-Simulator), der kan generere events. Slutteligt har vi placeret alle fakes, interfaces og modelklasser i et ClassLibrary projekt, der er gjort tilgængeligt for de to andre projekter igennem dependencies. Der eksisterer følgende model klasser i ClassLibrary projektet:

- StationControl
- ChargeControl
- Door
- RfidReader
- UsbCharSimulator

Hver af disse klasser implementerer et interface, der gør det muligt at lave dependency injection med de fakes, som der bliver brug for. Der eksisterer følgende interfaces i ClassLibrary:

- IChargeControl
- IDoor
- IRfidReader
- IUsbCharger

De 3 "komponent-klasser", der skal kunne generere events, fungerer som event sources, og er implementeret vha. C# events. For at gøre koden så læsbar som muligt, så er disse events placeret i de tilhørende interfaces, og implementeret i model klasserne. Der oprettes altså eksempelvis et DoorEvents, der indeholder en public bool DoorIsOpen, som ændres i dør-klassens OnDoorOpen() og OnDoorClose() funktioner. Idet der forekommer en ændring, så publiceres/opdateret der til eventet vha. invoke. Det er af denne årsag, essentielt, at event sourcen testes før de to controllers, hvorfor der er lavet test på de 3 event-sources (Door, RfidReader og UsbChargerSimulator). I disse tests bliver det undersøgt, om det fornævnte event overhovedet bliver rejst, på de forventede tidspunkter med korrekt data.
Idet vi har testet de 3 "komponent-klasser" kunne vi konstatere, at de forventede events bliver rejst. Herefter ønskes det at teste de to controllers StationControl og Charger-Control. Her ønskes det testet om de modtager de forventede events og håndterer disse korrekte. For at gøre dette anvendes der 3 fake klasser og en simulator. For Door og RfidReader er der ikke manuelt oprettet fake klasser, men der er i stedet anvendt NSubstitute til at autogenere fakes, da NSubstitute blandt andet selv håndterer hvornår der skal anvendes stubs og mocks. Yderligere anvendes faken til at rejse eventet. For UsbChargerSimulatoren, der skal emulere et strømtræk er selve simulatoren brugt til at teste uut ChargerControl. For at holde testen af de to controllers så simple som muligt, så er der anvendt den interne logfile vha. writer = File.AppendText(LogFile) ved testen af StationControl, og der er anvendt simple Console.Writelines til displayet ved test af begge controllers, hvorfor der ikke ikke eksisterer en Display og LogFile klasse.

Der anvendt en boundary value analysis til at identificere hvor mange test cases der var nødvendige for at teste alle equivalence partitioner. Eksempelvis er det testet for ChargeControlleren, om "Current" er korrekt, når der bliver rejst et CurrentEventArg. Her identificerede vi partionerne efter den opgivne tabel i handoutet, og lavede derfor samlet 10 test, for at sikre at alle grænseværdier for hver partion blev testet.
Gruppen lavede fælles opstart, hvor designet og UML-diagrammet blev diskuteret og fastlagt. Herefter blev der oprettet 3 projekter i en solution, og de udleverede klasser blev inkluderet. Efterfølgende blev der oprettet tomme klasser som placeholders for de klasser der selv skulle udvikles. Gruppen lavede herefter eventet til Door klassen sammen, så vi var enige om hvordan der korrekt skulle attaches og updates. Da gruppen havde skabt enighed om strukturen og designet splittede vi arbejdet ud, så hver person var ansvarlig for at færdiggøre en komponentklasse og dennes test. Yderligere arbejdede vi siddeløbende 1 mand på test af StationControl klassen og 2 mand sammen på ChargeControl klassen. Ved at opsplitte arbejdet på denne måde oplevede vi meget få mergeconflicter i Git.

6 Anvendelse af CI og Coverage rapport

Da gruppen havde opsplittet ansvaret for testen af de 3 komponentklasser og de 2 controllers, så blev der meget sjældent arbejdet samtidigt på tests i den samme klasse. Af denne årsag var det muligt for gruppen at teste forskellige tests samtidigt vha. resharper. Idet en test ikke fejlede, så kunne et gruppemedlem lave en pull og push request, der resulterede i at jenkins kørte. En enkelt gang blev der pushet tests der ikke virkede til CI serveren, hvorved at testen og coverage rapporten fejlede. Dette var et tydeligt eksempel på hvorfor det er en god ide at køre testen med resharper i visual studio før der pushes. Dette forsagede, at de andre gruppemedlemmer ikke kunne se coverage-rapporten indtil den fejlede test blev rettet. Selv om testen køres på continuous integration serveren, så vurderes det altså til at være en god ide at køre testen lokalt i visual studio først, idet en fejlet test kan forsinke resten af gruppemedlemmernes arbejde, da der ikke er adgang til coverage rapporten.