MobileChargingSystem

ChargeSystem

SWTF24

Gruppe 23

Omid Makki – au715512

Adi Saric – au721885

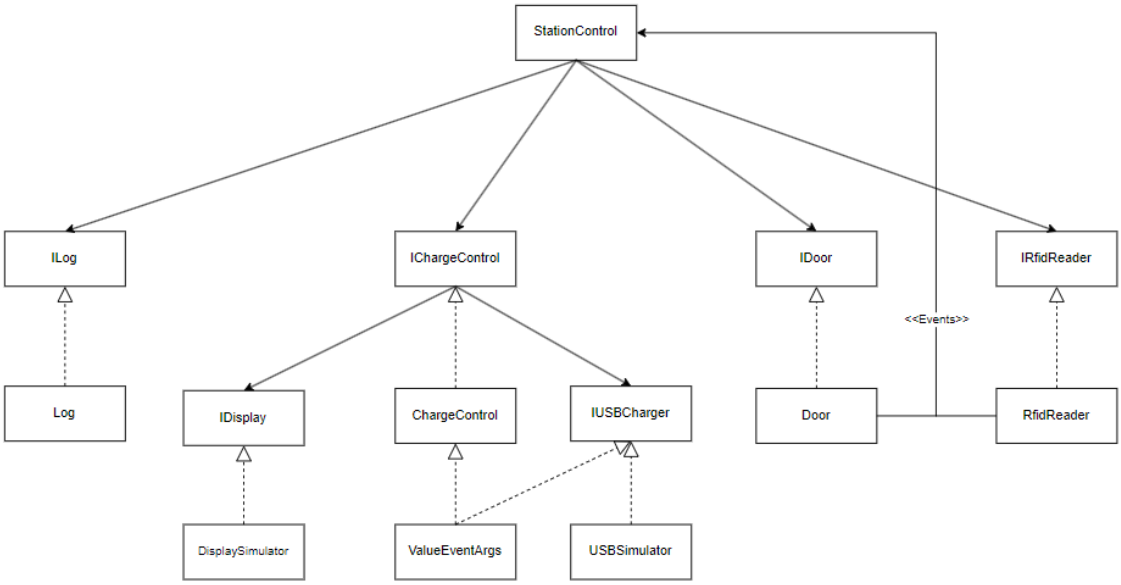
Lukas Vinge – au720575

Repository: https://gitlab.au.dk/gruppe23/handin2.git

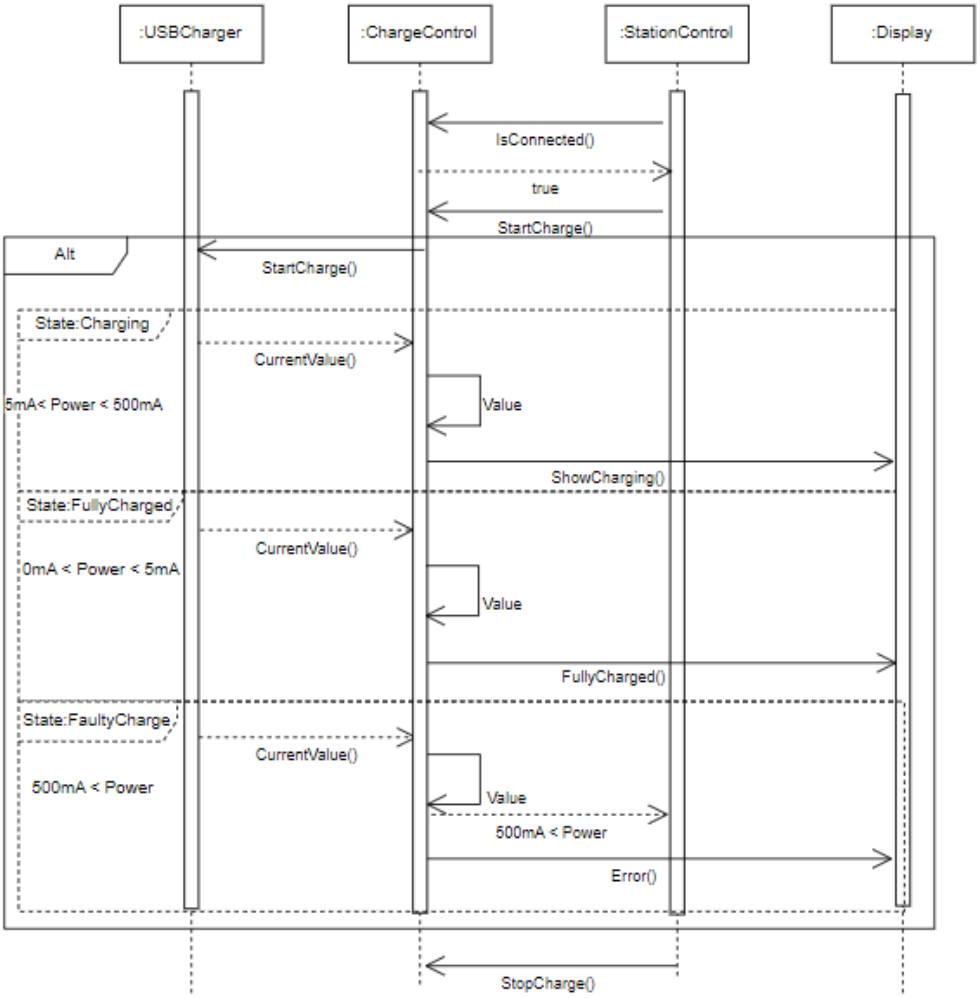
Diagrammer

Klassediagram:

I nedenstående klassediagram ses det hvordan det overordnede design af MobileChargingStation ser ud.



Sekvensdiagram:



Derudover er der lavet et sekvensdiagram, hvor der fokus på hvilket ting der sker under opladning. Her er klasserne USBCharger, ChargeControl, StationControl og Display. Her ses der at når strømmen er mellem 5mA og 500mA, skal Display vise ShowCharging(). Derudover skal Display vise FullyCharged() når strømmen er mellem 0mA og 5mA. Til sidst skal Display vise Error() når strømmen er over 500mA.

Simulering af Mobile charging station

Opaven går ud på at lave test af et hardware system, derfor er det afgørende at kunne teste systemet, selvom hardwaren ikke er tilgængelig i denne del af udviklingsfasen.

I designet af mobilechargingstation tages der højde for de udfordringer der kan opstå når hardwarekomponenterne mangler. Derfor implementerer vi simulatorer for boundary klasserne, som står for at simulere hardwaren og dennes interaktion med omverden.

Hvert komponent indeholder nødvendige funktioner og simuleret adfær for at kunne udføre gode og brugbare test disse inkluderer UsbSimulater, Doorsim, DisplaySimulator og RfidReaderSim.

Denne tilgang til systemet indebærer periodisk brug af white box testing, dette retfærdiggøres da vi tester med boundary klasser.

UsbSimulator

UsbSimulator koden er udleveret som en del af opgaven.

Denne klasse bruges til at simulere UsbCharger. Den simulerer tilslutning ved hjælp af SimulateConnected, og simulerer Overload ved hjælp af SimulateOverload. Vi kan styre opladerens input med kald af disse metoder, her kan vi teste hvordan systemet reagerer.

RfidReaderSim

Koden til RfidReader simulerer en Rfid læser der er tilkoblet systemet. Det er designet sådan at man kan scanne sit Rfid. Dette trigger så et event som giver indeholder et ID der oplåser Mobile Charging Station.

DoorSim

Denne klasse simulere døren ind til mobile charging station. Den har metoder der simulerer at døren oplukkes DoorOpened, og lukkes igen med DoorClosed.

Så er der opstillet en eventhandler med DoorClosedEvent og DoorOpenedEvent.

DisplaySimulator

Dette er en simulator til at teste på displayet. Denne simulator præsenterer instruktioner og meddelelser på displayet, som brugeren ville kunne se, dette gøres med Console.WriteLine.

Dette vises ved displayet output.

Design

Layered Structure

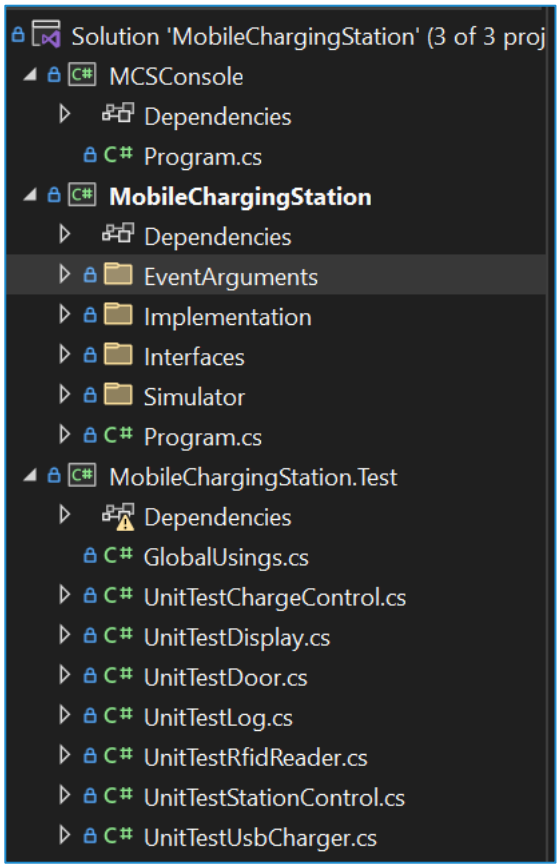
Vi har i vores kode opdelt det i 4 lag.

Interfaces, Implementering, EventArguments og Simulering.

Disse lag er placeret i hver deres project. Dette giver en god og klar opdeling af de opgaver vi havde for vores system. Det gjorde det lettere at udvikle systemet enkeltvis, og derefter fejlfinde på systemet.

Vi har brugt interfaces til at definere grænsefladerne i mellem komponenterne af systemet, dette gjorde processen i kodningen en del nemmere.

For at gøre koden mere overskuelig har vi gjort chargeControl til at styre opladningen og StationControl håndtere alle tilstande i systemet, samt interaktionen af de simulerede hardware komponenter i systemet.



Anvendte Testteknikker

- White Box
- Black Box
- Fakes
- Dependency Injection
- Coverage
- Boundary Values

Tests

White box testing

Vi har implementeret klasser der simulerer boundary klasserne. Dette gør at vi kan simulere os til at åbne og lukke døren, tilslutte og overbelaste laderen, samt scanne Rfid.

Disse ting kan ikke påvirkes af koden

Black Box Testing

Når vi indkapsler klasserne og sørger for at have private Set-metoder, så er klasserne sig som black box, hvor indre tilstande ikke er kendt for brugeren. Da de har private enums er vi nødsaget til at få et output for at teste de forskellige tilstande.

Dependency Injection

I C# muliggører vi udskiftning af implementeringer ved at bruge interfaces. Dette tillader os at udvikle flere forskellige implementeringer af samme funktionalitet.

Både stationControl og ChargeControl benytter sig af dependency injection ved at modtage deres afhængigheder gennem constructoren. Dette gør at vi kan benytte os af fakes eller stubs vha. NSubstitute.

Coverage

Vi har benyttet report generator via. Gitlab til at udregne vores Coverage.

Den laver en pipeline på vores kode og tester hvor meget af koden der er dækket.

Fakes

Vi benytter NSubstitue til at oprette, dette tillader os at efterligne adfæren i vores system. Her kan man sige at Simulator klasserne udgør en form for fakes.

Et eksempel på fakes er når vi f.eks. sætter UsbCharger til true.

Boundary Values

I ChargeControl har vi burgt Boundary Values til at analysere. Dette har testet på de værdier som HandleCurrentValueEvent har med de forskellige strømværdier. Vi har testet fire scenarier.

- Current = 0
- Current: 0mA < Power < 5mA
- Current: 5mA< Power < 500mA
- Current: 500mA < Power

Arbejdsfordeling

Vi har forsøgt at lave en paralleliseret arbejdsfordelingm vi har sat vores CI-pipeline op og det betyder at vi havde mulighed for at lave denne form for arbejdsfordeling. Vi har sikret os i design processen at alle var med omkring så der var en fælles forståelse for projektets udførelse.

Hvordan?

Først har vi oprettet vores vores Gitlab gruppe. Herefter har vi lavet både klasse- og sekvensdiagram så hele gruppen har samme forståelse for udførelsen af opgaven. Dette er derefter så blevet arbejdet på sammen men med hvert sit fokuspunkt.

Vi har kontinuerligt opdateret vores project ved hjælp af push til Gitlab, og herefter sparet med hinanden om hvilke løsninger der var til de forskellige dele af projektet.

Vi har arbejdet ret sent med projektet og har derfor haft en stor arbejdsmængde hen mod deadline, dette er ikke oplagt og bliver nok ikke fremgangsmåden fremover.

Hvorfor?

Grunden til denne tilgang var at vi ville sørge for at alle havde en god forståelse hele vejen rundt om projektet. Derfor startede vi også ud med at danne os et samlet overblik i designfasen, og herefter er gået lidt individuelt til værks. Til slut har vi så samlet stykkerne sammen.

Dette har gjort vi kunne finde løsningen til test og generelt opgaven sammen i fællesskab.

Fordele/Ulemper

Fordele

- Struktureret
- God kommunikation omkring individuelle fejl og mangler
- Arbejdet sikrede alle fik en forståelse for opgaven

Ulemper

- Ingen tidsplan, derfor presset til sidst
- Individuelle commits og push til gitlab kunne “ødelægge” koden

Proces

Arbejdet med et fælles repository og Continous Integration (CI) har både negative og positive sider.

Her er hvad vi har erfaret ved arbejdsprocessen.

1. Godt samarbejde ved fælles repository

Brugen af det fælles repository har gjort at alle har haft mulighed for at dele kodestykkerne med hinanden. Dette har gjort det nemt at kommunikere og hjælpe hinanden i løbet af processen.

2. Consistency

Vi har ved brug af en fælles repository nemt kunne holde den samme consistency i koden, det har gjort det nemt at se hvilke ændringer der har påvirket koden, både godt og skidt. Dette gjorde fejlfinding meget mindre besværligt.

Fordele og Ulemper ved CI

Fordele

- CI forbedrer kvaliteten af udviklingsprocessen, da det hjælper til at finde fejlene hurtigt i processen.
- Hurtig feedback, den hjælper også med at finde fejl hurtigt.
- Ci gør koden nemmere at holde sikker og uden fejl, dette gør arbejdet nemmere.

Ulemper

- Opsætningen af CI-Pipeline var bøvlet
- Der skal konstant opdateres, da man er nødsaget til at have det nyeste udkast af koden
- Processen kan blive kompleks hvis opgaven er større og inkluderer flere gruppemedlemmer