I4SWT

Software Technology Engineering

SWT - Hand In 2

Gruppe 10

Asger Busk Breinholm, 201807859
Andreas Stavning Erslev, 201406223
Mathias Holm Brændgaard, 201705103

26. oktober 2021

Indhold

1 Jenkins og GitHub

Jenkins link:
http://ci3.ase.au.dk:8080/job/Handin_Two_Gruppe10_vTwo/

Gamelt GitHub link (Forkert .gitignore):
https://github.com/SWTE2110/HandInTwo_SWT.git

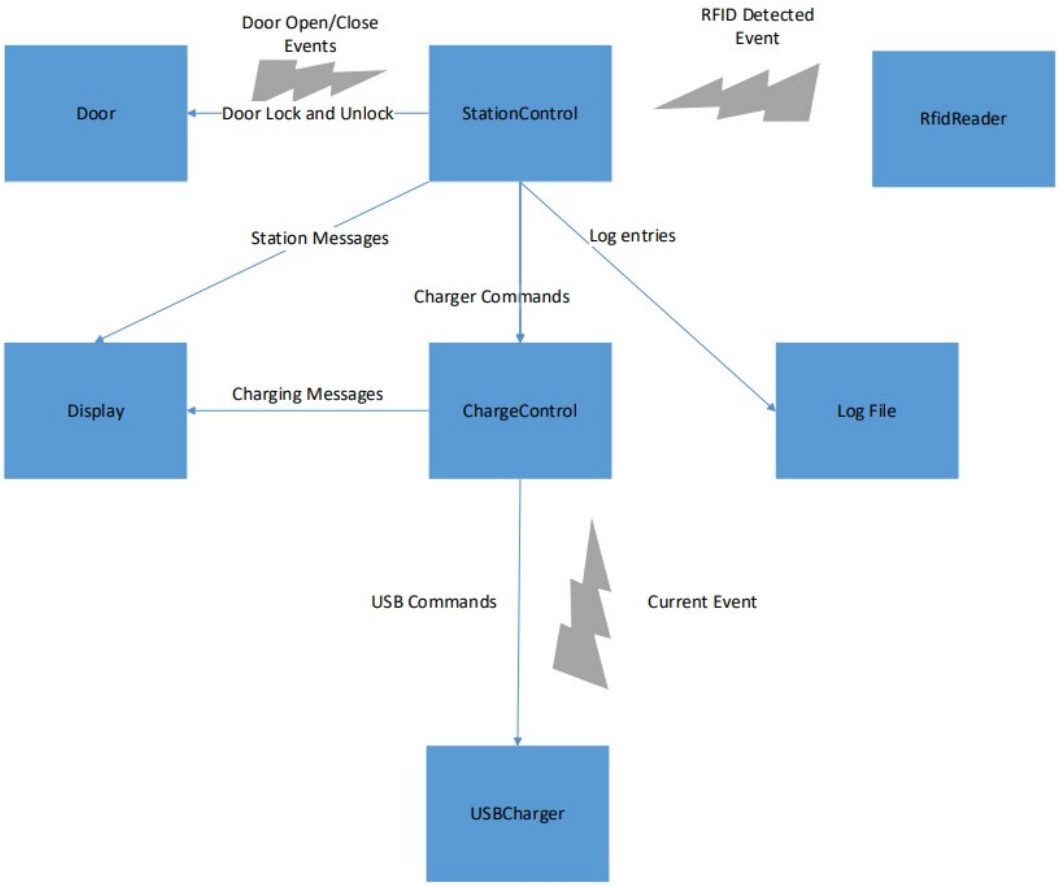
Github link:
https://github.com/SWTE2110/HandInTwo_v2

GitHub kontoer:
- Aerslev / AndreasErslev: Andreas | au588668
- SyntaxXeror | Asger | au581050
- DaloonOfDoom | Mathias | au518070

2 Software Design

OBS! I dette afsnit gøres der brug af klassediagram (Figur 3), der ses i Diagram-afsnittet.

I forbindelse med Software Design er der blevet brugt forskellige metoder til at sætte klasserne op. Ud fra analyse af Design-diagrammet, Figur 1 (Figur 2 i opgave beskrivelsen), er der blevet sat relationer op mellem klasserne. Her ses det, hvordan de forskellige klasser forbindes. Ud fra diagrammet ses også, hvordan der er event-interaktioner mellem tre klassepar. Et event mellem Door og StationControl; RfidReader og StationControl; og ChargeControl og USBCharger. Dette design kan bruges til at lave et simpelt UML-diagram, som dog kun består af overordnede klasser uden metoder.

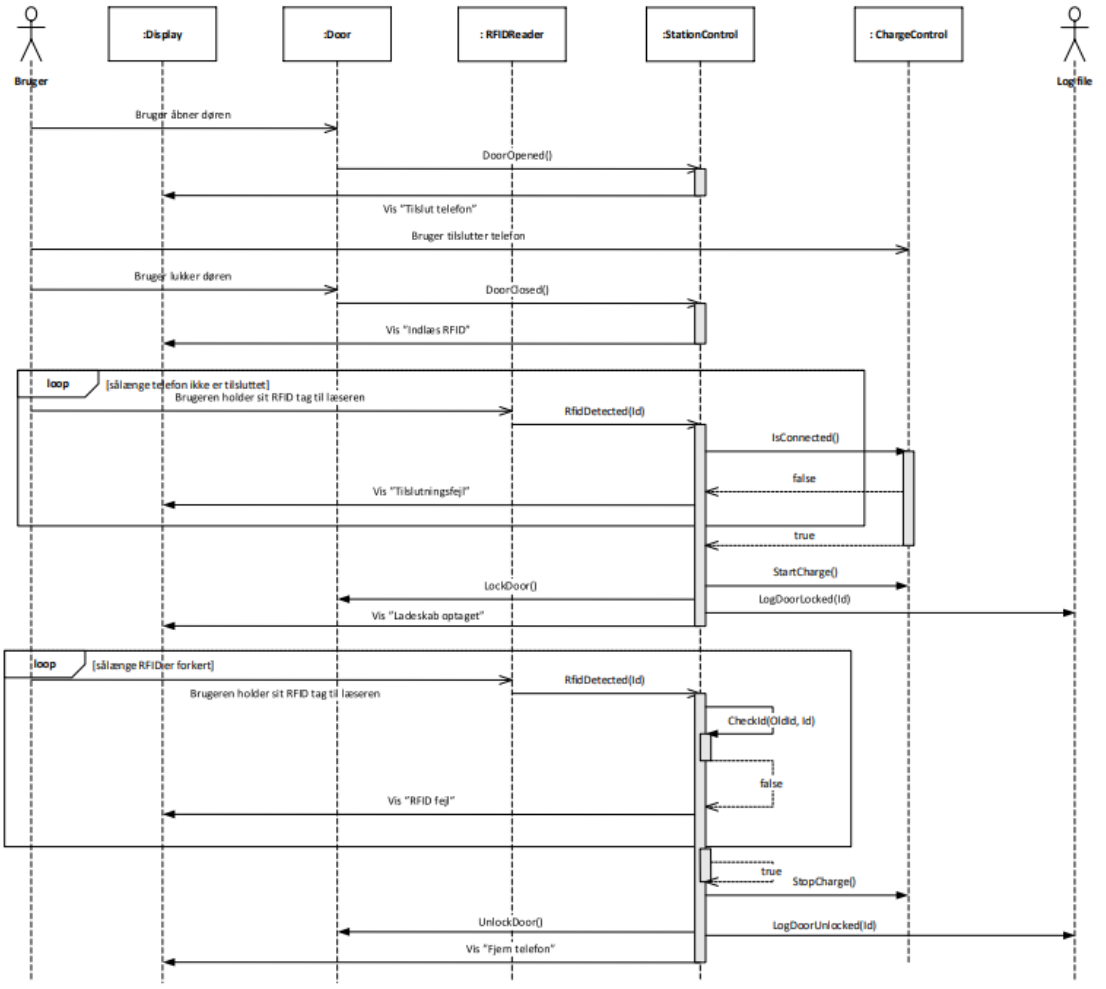


Figur 2: Designskitse

Figur 1: Designskitse (Figur 2)

I opgaven var der yderligere udleveret et sekvensdiagram, Figur 2, som kan bruges til at se de forskellige metoder, der bliver brugt mellem klasserne. Herudfra kan der dannes en idé om, hvordan klasserne interagerer ved brug af systemet. Da der allerede er opstillet et grundlæggende UML-diagram, er det hurtigt at tilføje metoder ud fra sekvensdiagrammet. På sekvens ses ikke

de events der bliver brugt, så her er det en fordel at have Designskitsen, som fortæller, at der skal være et event.



Figur 3: Sekvensdiagram for anvendelse af ladeskabet

Figur 2: Sekvens diagram (Figur 3)

Klasse diagrammet kan nu sættes op med interfaces og klasser. Ved nærmere analyse af klasserne ses behov for diverse attributter. Disse indskrives i klassen selv, da det ikke er nødvendigt for interfaces at kende dem. Relationerne mellem de forskellige interfaces kan nu også sættes op, så de beskriver den måde der kommunikeres på. Heraf ses det at "IRifdReader"kun har en relationer, der er et event, og derfor ikke har nogen tilbagevende relation til "StationControl". Selve diagrammet forklares yderligere i design afsnittet.

2.1 SOLID

Der er blevet gjort brug af SOLID-principperne i designet. Heraf ses f.eks. Single-responsibility principle (S), da alle klasserne er delt op, således at de kun har én specifik funktion. F.eks. er LogFile kun til at logføre informationer, og Display kun er til for at skrive beskeder ud til brugeren.

Der er også blevet brugt Open–closed principle (O), således at det ikke er nødvendigt at ændre i de eksisterende klasser for at ændre systemets adfærd. Der kan blot laves nye implementeringer af det pågældende interface. Dette gør det desuden lettere at teste systemet via dependency injection.

Der gøres ikke brug af Liskov Substitution-princippet, da der ikke nedarves fra nogen klasser (L).

Ved at sætte et interface op for hver klasse, der beskriver ligepræcis det, den klasse skal, ikke mere eller mindre, er der da også blevet gjort brug af interface-segregation principle (I).

Yderligere er der mellem ChargeControl og USBCharger blevet gjort brug af Dependency inversion principle (D), da der er et abstraktions niveau mellem de to klasser. Da ChargeControl er en High-level moduel, og USBCharger er et Low-Level, er det nødvendigt at skabe en sammenhæng, hvor ChargeControl ikke er afhængig af USBCharger. Dette gøres ved at sætte et abstraktions lag op mellem de to, som her er IUSBCharger.

Programmet overholder derved SOLID-principperne.

3 Arbejdsfordeling

Til start er der blevet lavet Software design, hvilket er blevet lavet samlet, da det er vigtigt, at alle har god forståelse af designet. Ved fælles analyse af designdiagram (Figur 1) og sekvensdiagram (Figur 2) er der blevet diskuteret frem og tilbage om, hvordan disse diagrammer kunne bruges bedst muligt. Ud fra diskussionen er der ad flere omgange blevet lavet det UML-klasse diagram, der ses under Diagram-afsnittet. Herefter er der blevet oprettet filer for både interfaces og klasser. Herefter har vi diskuteret hvad de forskellige metoder skulle kunne.

Efter alt er blevet diskuteret, er koden blevet opdelt i mindre grupper. Både den kode der skulle til i klassen, for at få den til at have den ønskede effekt, men også de test der skulle til. Undervejs i processen, er der blevet gjort status, hvor der, med nye øjne, reflekteres over den skrevne kode. Løbende blev der lavet Push til GitHub, hvilket gjorde det muligt for de andre gruppemedlemmer, hurtigt at integrer nyskreven kode til deres lokale projekt.

Grundet diverse fejl, var der problemer med at opsætte Jenkins og GitHub server. I denne forbindelse har alle gruppemedlemmer hjulpet med at løse dette.

Alle har været med til at skrive rapporten, dog på grund af sygdom under en møde gang, har Andreas været med på sidelinjen, i forbindelse med at færdiggøre de sidste detaljer i koden. Denne tid er da flittigt blevet brugt på at skrive rapport.

4 CI

I opgaven er der blevet lavet en Jenkins-server, samt lavet en WebHook, der forbinder Jenkins med et GitHub-repository. Dette sørger for, at der bliver lavet en test, hver gange der bliver lavet et push til GitHub-repositoriet. Jenkins-serveren er opsat med Test Coverage. Ved denne opsætning undersøges der, hvor meget af koden er blevet gennemløbet ved test.

På grund af forkert refereret GitHub-reposotory er der ikke blevet lavet test efter alle pushes. Herefter er der opstået problemer, da der manglede en NuGet package. Derforefter ville Test Coverage ikke køre, der der var fejl i opsætningen af Jenkins-pipeline.

I opgaven er der som nævnt blevet sat en Jenkins-server op. Denne skal bruges til at lave tests samt lave en Test Coverage-rapport, der vurderer, hvor meget af koden der er blevet testet. Imens koden bliver skrevet, er der brug for at udføre versionsstyring. Dette er gjort via en GitHub-server. For at teste koden så ofte som muligt bliver der sat en WebHook op mellem Jenkins-serveren og GitHub-serveren. Denne WebHook sørger for, at når der bliver lavet et Push til GitHub serveren, vil der automatik laves en test og laves en Test Coverage-rapport.

Ved oprettelse af GitHub-repositoriet blev der ikke oprettet en .gitignore til at starte med. Det gav store problemer i forhold til versionsstyring, hvilket gjorde, at vi måtte oprette et nyt repository, og overføre alt arbejde, der var lavet, dertil. Dette betyder, at noget af det forgående arbejdes versioner kun ligger på det tidligere GitHub-repository.

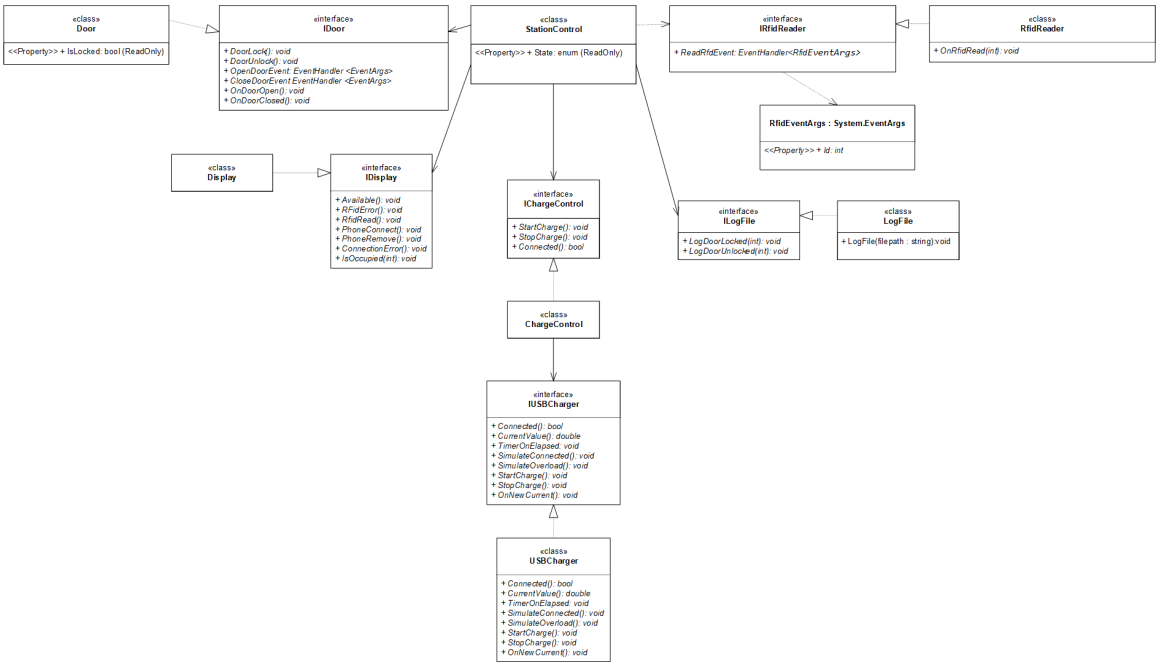
Vi løb ind i et problem i forbindelse med unit test of LogFile på CI-serveren. Her fik vi en fejl på serveren, som ikke opstod på vores egne computere. Dette skyldes, at LogFile's constructor kalder funktionen File.Create(string filepath), såfremt den pågældende fil ikke eksisterer. Dette var ikke et problem på de lokale enheder, da filen eksisterede fra en tidligere test. Da testen skulle køres på CI-serveren, blev vi derfor opmærksomme på, at noget var galt. Først troede vi, at det skyldtes, at flere tests blev kørt parallelt, således at flere programmer forsøgte at tilgå den samme fil samtidigt. Efter at ændre testmetoderne, så hver test brugte hver sin fil (testen blev kørt mere end én gang), blev problemet afhjulpet, men dette skyldes, at filerne allerede eksisterede på CI-serveren fra første gennemløb af det modificerede testprogram. Efter at have undersøgt File.Create nærmere fandt vi ud af, at den benyttede logging-metode, File.WriteAllText, selv opretter filen, hvis den ikke findes, afskaffede vi koden fra LogFile's constructor, og alle tests kører nu uden problemer.

Der ses i forbindelse med Test Coverage en dækning af programkoden på 100%, hvilket var målsætningen. I HandinTwo.Test ses imidlertid kun en dækning på 99% på grund af Test_OpenDoorEvent_Locked(), hvor en anonym funktion subscribes til et event, som aldrig bliver raised. Den anynome funktion bliver derfor aldrig kaldt, hvilket giver anledning til en coverage på under 100%. Dette var dog hensigten, da testen skulle vise, at eventet aldrig bliver raised.

5 Diagram

Som der er beskrevet i Design-afsnittet, er UML-klasse diagrammet blevet inspireret af figurene i opgavebeskrivelsen. På diagrammet ses relationerne mellem de forskellige klasser. Der ses, at StationControl indeholder referencer til de fleste interfaces på nær IUSBCharger. Dette skyldes at StationControl er bindeledet mellem klasserne. Klasserne giver tilsammen et indblik i, hvordan systemet skal fungere. En af grundene til at systemet er bygget op sådan, er SOLID-principperne, som også er beskrevet yderligere i Design-afsnitet.

Det blev besluttet, at private fields ikke skulle anføres i diagrammet. Derudover besluttede vi, at kun extentions til interfaces skulle anføres i de implementerende klasser. Vi mente, at selve naturen af interfaces gjorde det overflødigt at anføre interfacets indhold to steder. Ved relationen mellem StationControl og IRfidReader ses desuden en anden type relation end mellem SationControl og andre interfaces, da relationen mellem de to udelukkende består af et event.



Figur 3: Klasse Diagram for Ladeskab