1 Introduktion

Dette dokument beskriver specifikationen for den 2. obligatoriske handin i faget Softwaretest.
Den skal udarbejdes i grupper og afleveres og godkendes som beskrevet på Blackboard under Gruppetilmelding.

Dokumentet falder i 3 dele.

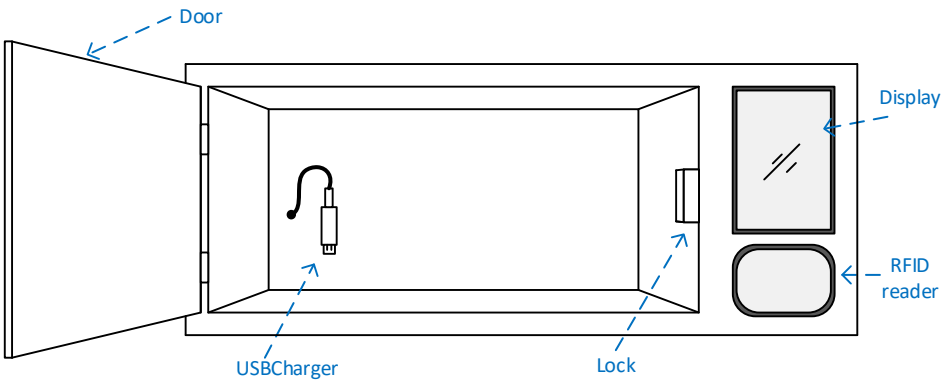
1. Første del (afsnit 2 Ladeskab til mobiltelefon) er specifikation og information til et ladeskab for mobiltelefoner, som er det system, der skal designes, implementeres og unittestes.
2. Anden del (afsnit 3 Krav til afleveringen til Handin 2) beskriver kravene til gennemførelse af opgaven, hvordan der skal arbejdes, hvad der skal afleveres og hvad afleveringen skal indeholde.
3. Tredje del (afsnit 4 Resurser) fortæller om de bilag, der kan give ideer og supplement til udførelse af opgaven og yderligere specifikation af dele af systemet.

Kravene er afspejlet i de Rubrics, der er oprettet til afleveringen. Rubrics bruges til vejledende evaluering, og pointtallet fra dem er ikke en facitliste til om man får godkendt sin aflevering.

2 Ladeskab til mobiltelefon

Denne opgave omhandler design, implementering og test af software til et ladeskab til en mobiltelefon, som skal opstilles i omklædningsrummet i en svømmehal eller lignende. Formålet med ladeskabet er, at en bruger kan tilslutte sin mobiltelefon til opladning i ladeskabet, efterlade sin telefon her og senere afhente den igen. Som identifikation tænkes en RFID tag der hænger på badebåndet eller nøglen til et tøjskab.

En principskitse af ladeskabet er vist herunder:



Figur 1: Principskitse af ladeskabet

Ladeskabet tænkes brugt som følger (det antages at skabet ikke er i brug):

1. Brugeren åbner lågen på ladeskabet
2. Brugeren tilkobler sin mobiltelefon til ladekablet.
3. Brugeren lukker lågen på ladeskabet.
4. Brugeren holder sit RFID tag op til systemets RFID-læser.
5. Systemet aflæser RFID-tagget. Hvis ladekablet er forbundet, låser systemet lågen på ladeskabet, og låsningen logges. Skabet er nu *optaget*. Opladning påbegyndes.

Brugeren kan nu forlade ladeskabet og komme tilbage senere. Herefter fortsætter den tiltænkte brug som følger:

6. Brugeren kommer tilbage til ladeskabet.
7. Brugeren holder sit RFID tag op til systemets RFID-læser.
8. Systemet aflæser RFID-tagget. Hvis RFID er identisk med det, der blev brugt til at låse skabet med, stoppes opladning, ladeskabets låge låses op og oplåsningen logges.
9. Brugeren åbner ladeskabet, fjerner ladekablet fra sin telefon og tager telefonen ud af ladeskabet.
10. Brugeren lukker skabet. Skabet er nu *ledigt*.

Når låsning/oplåsning finder sted, skal begivenheden logges til en fil inkl. timestamp (timer, minutter og sekunder), det aktuelle RFID samt en beskrivende tekst for begivenheden.

Opladningsdelen benytter sig af en USB chip, der kan tænde/slukke spændingen på ladestikket og måle den aktuelle ladestrøm, samt konstatere om der er en telefon forbundet til stikket.

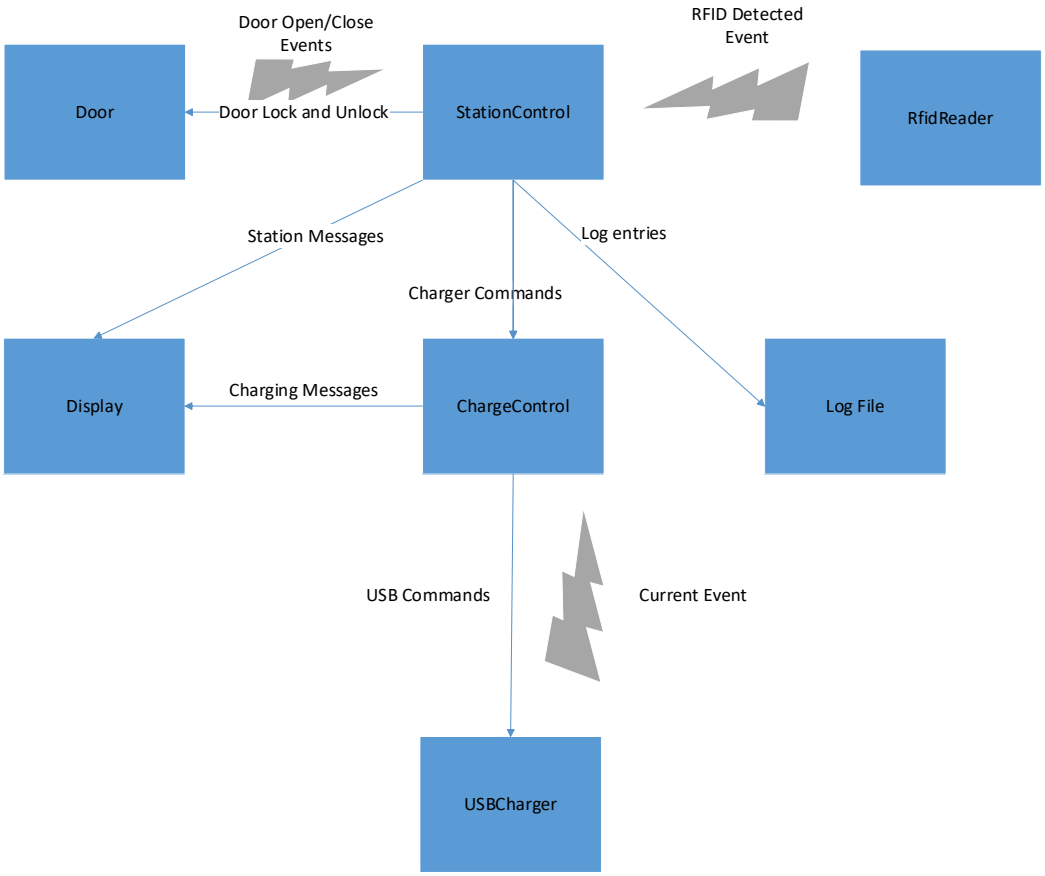
Opladningsdelen skal holde øje med, om opladningen er færdig, eller om der en fejlsituation.

Ud fra målingen af ladestrømmen fra USB chippen kan følgende situationer opremses:

Målt ladestrøm	**Betydning**
0 mA	Der er ingen forbindelse til en telefon, eller ladning er ikke startet. Displayet viser ikke noget om ladning.
0 mA < ladestrøm ≤ 5 mA	Opladningen er tilendebragt, og USB ladningen kan frakobles. Displayet viser, at telefonen er fuldt opladet.
5 mA < ladestrøm ≤ 500 mA	Opladningen foregår normalt. Displayet viser, at ladning foregår.
ladestrøm > 500 mA	Der er noget galt, fx en kortslutning. USB ladningen skal straks frakobles. Displayet viser en fejlmeddelelse.

Displayet har 2 områder, der kan anvendes til henholdsvis brugerinstruktioner og –meddelelser, og til status/meddelelser for opladningsdelen.

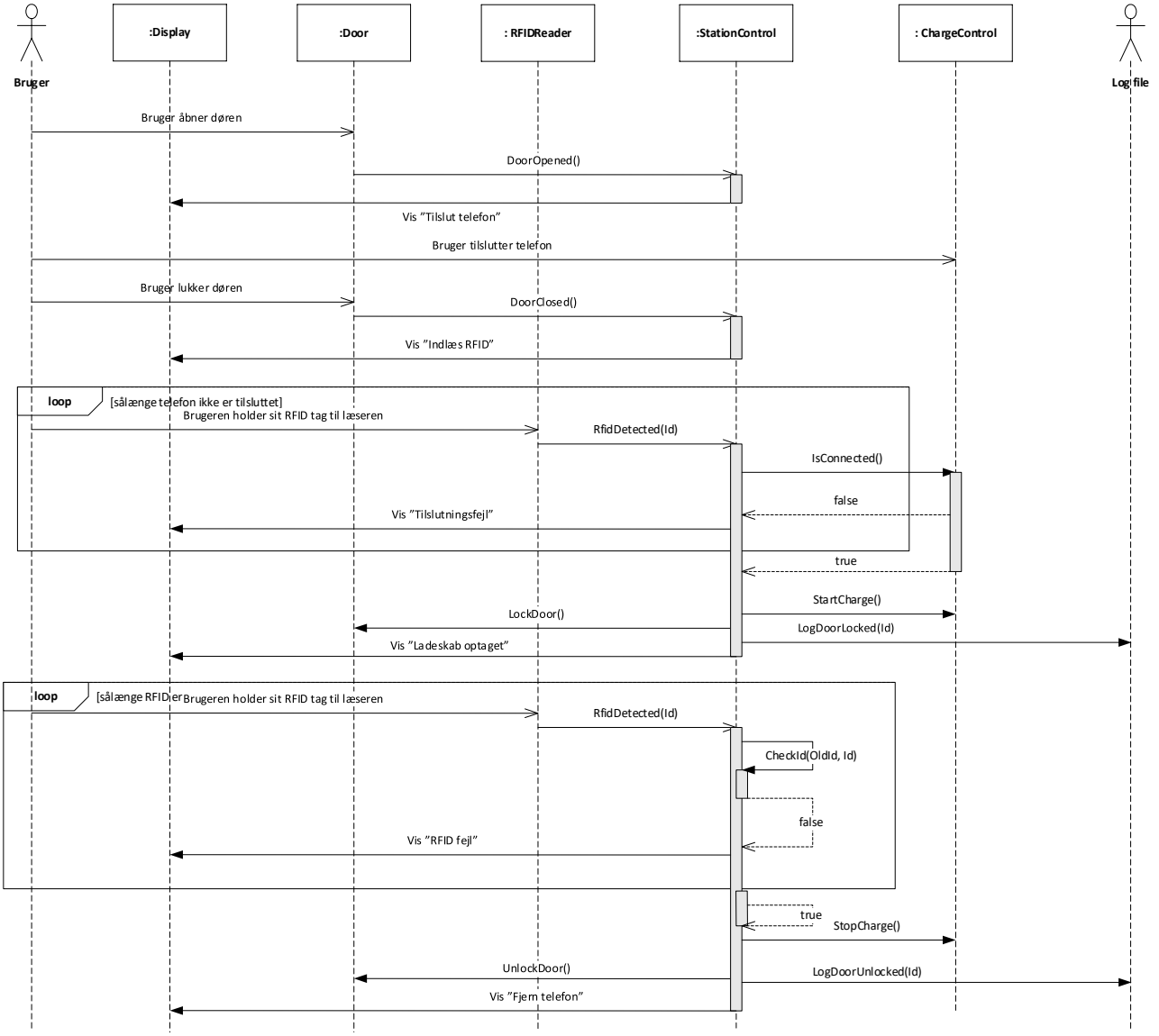
Et løseligt udkast til et design (*ikke* et UML klassediagram) er givet nedenfor:



Figur 2: Designskitse

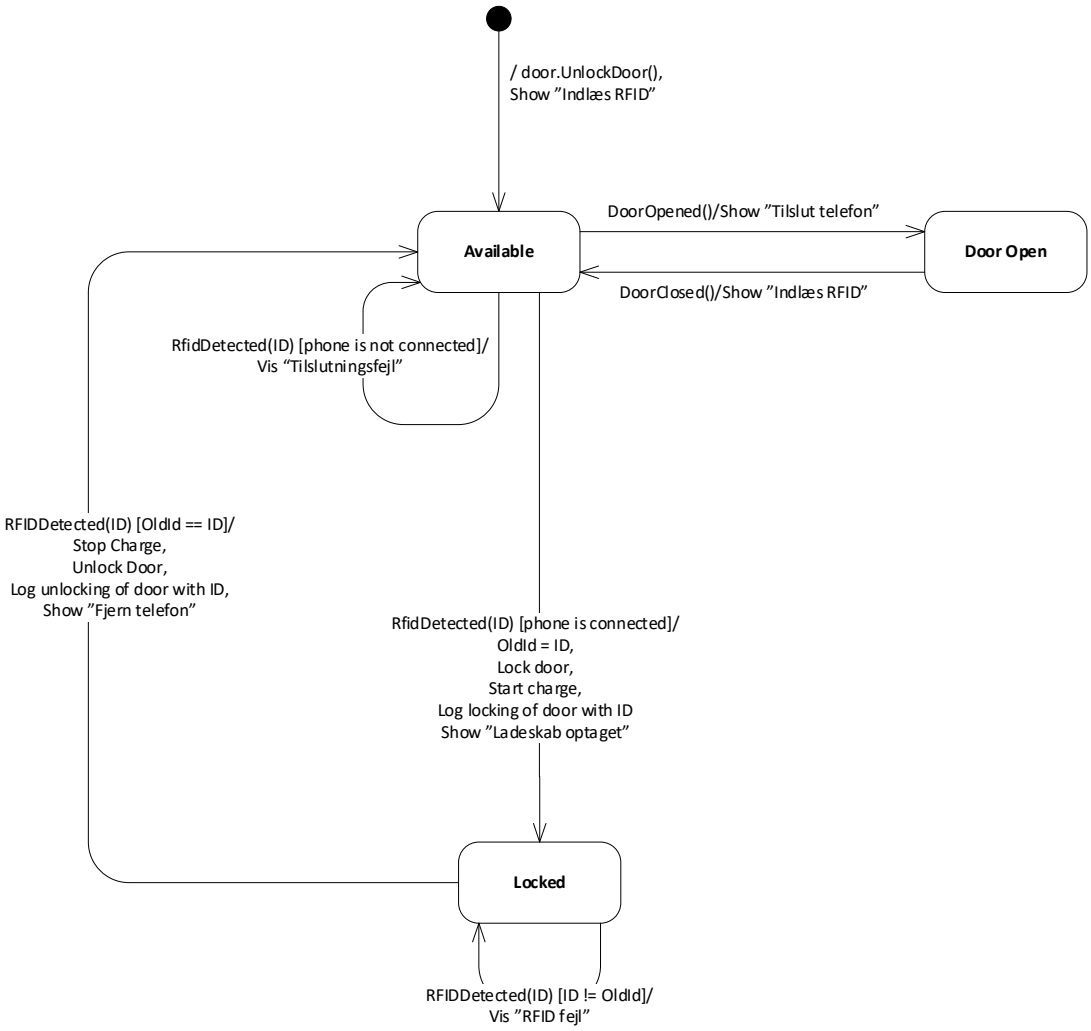
Klasserne `Door`, `Display`, `RFIDReader`, `USBCharger` på Figur 2 repræsenterer komponenter af samme navne på Figur 1. `Log File` kan ikke ses, men er indbygget i systemet, og kan inspiceres med software, der ikke er en del af denne opgave. Klassen `StationControl` er controller-klassen for systemet. `ChargeControl` er controller-klassen for opladningen.

Klassernes interaktion er vist på Figur 3. Bemærk, at dele af kommunikationen er synkron (metodekald) mens andre er asynkron (events).



Figur 3: Sekvensdiagram for anvendelse af ladeskabet

Klassen `StationControl`'s virkemåde kan beskrives vha. et UML tilstandsdiagram som følger:



Figur 4: Tilstandsmaskine for klassen StationControl

Et løseligt udkast til dele af implementeringen af klassen `StationControl` er givet i filen `StationControl.Handout.cs`.

En `USBCharger` simulator til brug for inspiration, integration og inkludering i den endelig app er givet i filerne `IUsbCharger.cs` og `UsbChargerSimulator.cs`., se afsnit 4.

3 Krav til afleveringen til Handin 2

I skal designe, implementere, unit teste og dokumentere et Ladeskabssystem som ovenfor beskrevet, idet I selvfølgelig ikke kan integrere med hardware der ikke findes.

Kravene til selve systemet står i ovenstående afsnit

De formelle krav til jeres aflevering fremgår nedenfor.

Følgende resultater af jeres arbejde skal afleveres på Blackboard, GitHub og Jenkins serveren:

1. Én (1) journal i PDF format på Blackboard, der beskriver dit design og refleksioner ifølge nedenstående retningslinjer.
2. Én (1) Visual Studio solution, i ét GitHub repository, der indeholder implementation og unittests, organiseret i passende class-library projekter samt en applikation i et console application projekt, ifølge nedenstående retningslinjer
3. Ét (1) Jenkins projekt, der er tilknyttet GitHub repositoriet, som udfører alle unit tests, viser testresultaterne og beregner og viser coverage for unit testene, ifølge nedenstående retningslinjer.

Der skal kun afleveres journalen på Blackboard, som et PDF dokument, som hoveddokument. Meget store diagrammer kan af hensyn til læsbarhed afleveres som bilag, fx direkte i Visio format. Check om diagrammer er læselige.

INGEN ZIP FILER!

Kode **skal** afleveres på GitHub og Jenkins jobs **skal** afleveres på Jenkins serveren.

Undtagelser til nedenstående krav skal godkendes af mig, og skal resultere i en løsning der er mindst ligeså god, som hvis man brugte det, der er omtalt i lektionerne og kravene. Derudover skal det kunne lade sig gøre at bedømme løsningerne uden for stort besvær – vi skal rette ca. 30 store afleveringer!

3.1 Krav til journalen og designet

Journalen skal være 5-15 sider lang. Den skal indeholde:

1. Gruppenummer
2. Gruppens medlemmer med studienumre
3. URL for GitHub-repositoriet
4. URL for Jenkins-jobbet
5. Klasse-, sekvens- og andre nyttige diagrammer med forklaringer, som beskriver jeres testbare design, opbygningen af jeres løsning og dens opførsel
6. Jeres design skal tage højde for den ikke eksisterende hardware og andre svært kontrollerbare afhængigheder og indkapsle dem, således at der kan testes gennem fakes
7. En refleksion over jeres valgte design (hvorfor, fordele og ulemper, ikke en beskrivelse, den gav I ovenfor)
8. En beskrivelse og refleksion over hvordan I fordelte arbejdet imellem jer (hvordan, hvorfor, fordele og ulemper)
9. En refleksion over hvordan arbejdet gik med at bruge et fælles repository og et continuous integration system (observationer, fordele og ulemper)

Fif til designet:

- "Events are your friends". Events er en stærk måde til at binde ting sammen med løs kobling, men det skal selvfølgelig heller ikke overdrives
- Med en passende anvendelse af events, og evt. timers, er det ikke nødvendigt at lave et multi-threaded design, men det er tilladt, hvis det bliver testet ordentligt
- Brug de udleverede resurser (se nedenfor) til at lade jer inspirere

3.2 Krav til Visual Studio solution, GitHub repo og arbejdsmetode

1. Der skal være mindst 3 projekter samlet i netop én Visual Studio solution:
    a. et klassebibliotek med "core" funktionaliteten i form af alle de nødvendige klasser til at implementere Ladeskabssystemet
    b. et klassebibliotek med unit test af ovenstående klasser
    c. et console app projekt, der bruger klasserne til at implementere en simpel app, der demonstrerer hvordan de skal sættes sammen og et simpelt forløb. Mere avancerede apps eller et GUI, er også tilladt, men giver ikke bedre bedømmelse
2. Som udgangspunkt forudsættes at der bruges .Net Framework for alle projekter. Hvis I vælger andet (.Net Core), er det jeres eget ansvar, at det også virker i forhold til GitHub og de build, test og coverage muligheder, der stilles til rådighed på Jenkins serveren
3. Jeres GitHub repository skal være offentligt
4. Commit-historikken på jeres repo skal vise, at I har fordelt arbejdet imellem jer, således at det er tydeligt, at der er bidrag fra alle gruppens medlemmer
5. Commit-historikken på jeres repo skal vise, at I har prøvet at arbejde med Continuous Integration, med en høj frekvens af commits og push'es

Fif til test:

- I kan få nytte af alt, hvad I har lært indtil nu. Brug mine anbefalinger
- Undersøg hvad NUnit og NSubstitute kan gøre for jer i forbindelse med exceptions (hvis aktuelt), events, Asserts og argumentmatching til at skrive nogle gode og stærke tests, med et minimum af arbejde – andet end at sætte sig ind i mulighederne

3.3 Krav til Jenkins anvendelsen

1. Det skal være ét Jenkins projekt/job, der kombinerer unit test og coverage beregning, og sørger for at resultaterne af begge er synlige på projektets hovedside på Jenkins
2. I skal bruge vores Jenkins server på ci3.ase.au.dk:8080
3. Jeres projekt skal have et navn, der tydeliggør hvilken gruppe der har lavet det og at det drejer sig om handin 2
4. Jeres build-historik skal vise, at I har arbejdet efter Continuous Integration under hele forløbet, så man kan se en interessant test-trend kurve. Sæt derfor jeres repo og Jenkins job op så tidligt som muligt

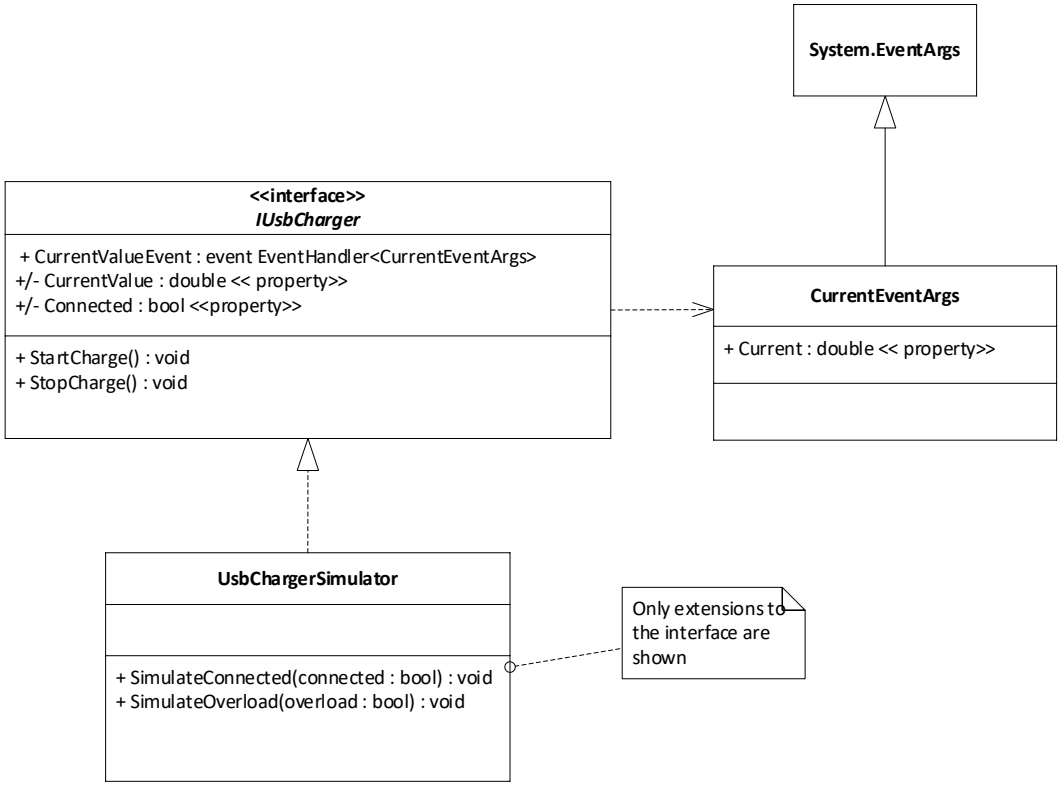
4 Resurser

Til inspiration findes på Blackboard filen `StationControl.cs`, der viser skelettet til implementering af en tilstandsmaskine, med illustration af en enkelt transition trigger. Betragt det som legacy kode, der ikke er lavet helt i henhold til de principper, I lærer her på kurset.

Derudover er der en implementation af en USB Charger simulator, der er indkapslet i et fornuftigt interface til en hardwaredriver for en USB chip, der kan nogle af de ting, der skal bruges.

Den kan bruges som den er, måske bortset fra, at den skal flyttes eller kopieres ind i jeres eget namespace.

Den simulerer opladningsprocessen for en telefon, fordelt over 20 minutter.



Figur 5 Klassediagram for hjælpekode til USB Chargeren

I filen `IUsbCharger.cs` finder man følgende definition på event args og interface:

```
public class CurrentEventArgs : EventArgs
{
    // Value in mA (milliAmpere)
    public double Current { set; get; }
}

public interface IUsbCharger
{
    // Event triggered on new current value
    event EventHandler<CurrentEventArgs> CurrentValueEvent;

    // Direct access to the current current value
    double CurrentValue { get; }

    // Require connection status of the phone
    bool Connected { get; }

    // Start charging
    void StartCharge();
    // Stop charging
    void StopCharge();
}
```

`CurrentEventArgs` og `IUsbCharger` **skal** I bruge i jeres design og kode.

Derudover ligger der en fil `UsbChargerSimulator.cs` med kode for en `UsbChargerSimulator`, der kan simulere opladning, forbindelse og overload.

Denne kan bruges i den samlede app, og til at lave eksperimenter med i forhold til jeres egen kode. Men da den simulerer en opladning på 20 minutter, er den ikke umiddelbart egnet til test.

Der er også en fil `TestUsbChargerSimulator.cs` med unit test af `UsbChargerSimulator`. Denne kan man lade sig inspirere af.