SWT2 – Air Traffic Monitoring

Frederik Fuglsang Midtgaard	AU593459	Github: Batmancarlo
Troels Østergaard Bleicken	AU586685	Github: TroelsBleicken
Morten Rosenquist	AU588570	Github: MortenLyngRosenquist
Christopher Bernold	AU574151	Github: Bernold1

Github Solution: https://github.com/swt-gr9/ATMS-gr9/
Jenkins Server: http://ci3.ase.au.dk:8080/job/TeamSWT9-ATMS-CoverageTest/

[0]Billede: https://upload.wikimedia.org/wikipedia/en/9/9b/Aarhus_University_seal.svg

Indhold

Indledning

For enhver lufthavn er overvågning af luftrummet en kritisk opgave. Den store mængde af fly gør det til en stort set umulig opgave for et menneske. Derfor er det oplagt at lave et automatiseret, computerstyret system, der varetager denne opgave. Formålet med denne aflevering er at designe og teste en implementering af et lignende system.

1 Design

I dette afsnit vil vi uddybe designet af systemet, samt tankerne bag designet.

1.1 Tilblivelsen af designet

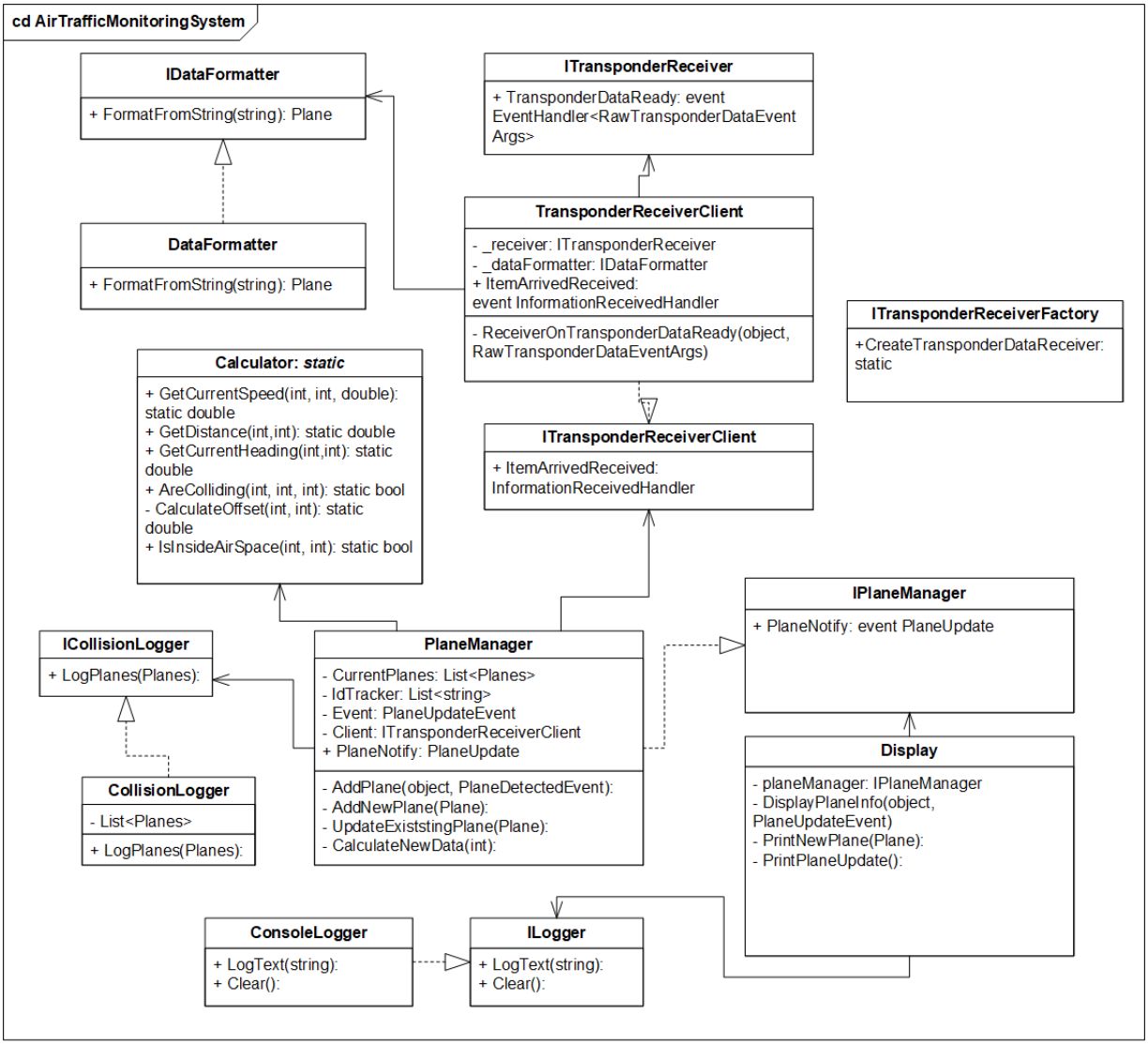
I starten af design fasen, gennemgik gruppen kravene til systemet, og opskrev de forskellige ting som skulle til før det input systemet fik, kunne fremstilles som det ønskede output på display. Herefter blev flere forskellige overordnede designs diskuteret, og det blev endeligt konkluderet, at det smarteste design var et samlebånds design, hvor hver led på samlebåndet havde et ansvar, hvorefter det næste led viderearbejdede dataet. Det blev også besluttet at sammenkoble samlebånds delene med events og eventhandlere, for at sænke koblingen i systemet.

1.2 Klasser

Herunder ses en tabel over de forskellige klasser for systemet for systemet. Systemet er delt op i flere under klasser, som hver har en specifik opgave for systemets funktionalitet. Klasserne er som følgende:

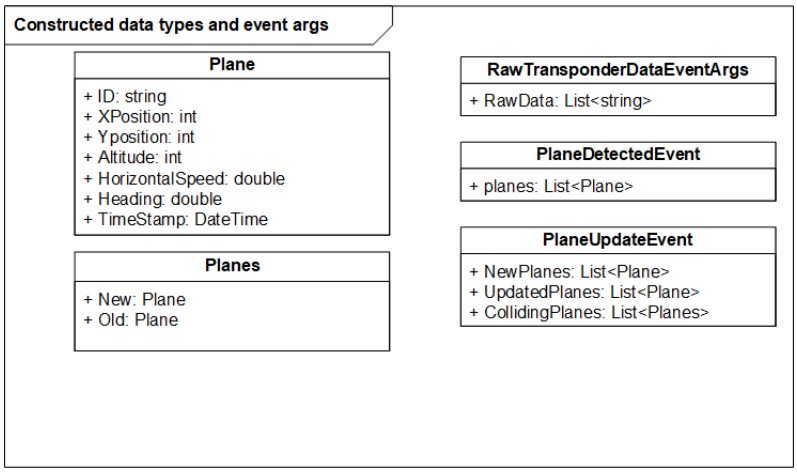
Klasse	**Ansvar**
Plane	Data klasse som indeholder de forskellige værdier for et fly, eks. ID, altitude og fart
TransponderRecieverClient	Klasse som håndterer event når nye transponderdata modtages, kalder DataFormatter og laver nyt event med formateret data
DataFormatter	Klasse som formattere transponderdata string til en Plane klasse
PlaneManager	Reagere på event fra TransponderRecieverClient, kalder funktioner i calculater, og laver nyt event med opdateret liste over fly
Calculator	Udregner ekstra værdier for fly, eks. fart og retning
Display	Reagere på event fra PlaneManager, og viser modtaget information på konsollen
AirTrafficMonitoringSystem	Laver instanser af alle klasser og starter systemet.
Plane	Data klasse som indeholder de forskellige værdier for et fly, eks. ID, altitude og fart
ConsoleLogger	Skriver tekst i konsolen
CollisionLogger	Skriver kollisionsinformationer til en .txt fil

Et klassediagram for systemet kan ses på figur 1. Klassen der samler alle klasserne er ikke tilføjet på klassediagrammet. Men den kender til alle interfaces.



Figur 1: Klassediagram for systemet

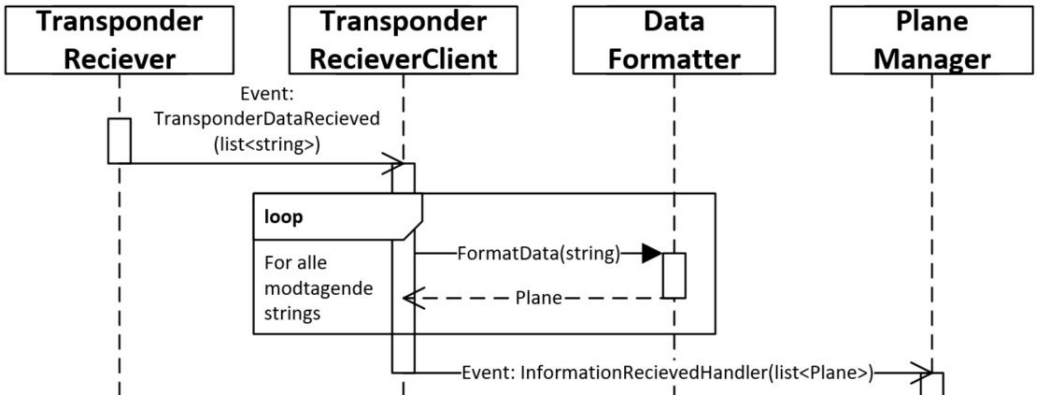
På klasse diagrammet kan der ses forskellige datatyper og event args. De kan yderligere beskrevet på figur 2



Figur 2: Datatyper og event args

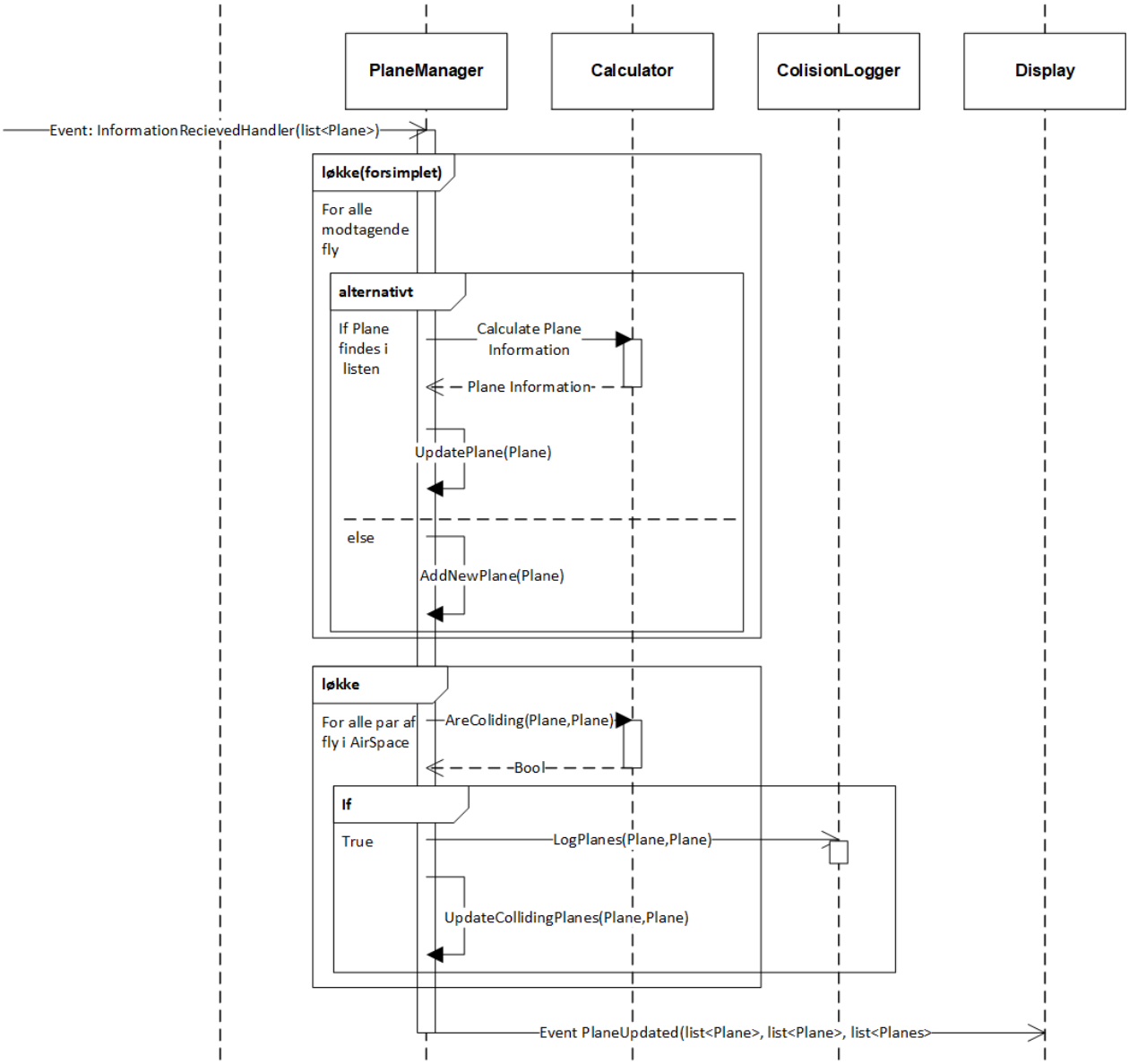
1.3 Sekvensdiagram

På følgende figurer ses systemets funktionalitet beskrevet ved sekvensdiagrammer. Systemet består hovedsageligt af 3 eventhandlere, som hver bruger nogle hjælpe klasser til at udfører deres funktionalitet. Når en eventhandler er færdig, laver den et nyt event, som den næste eventhandler kan reagere på. Det samlede sekvens diagram for systemet kan findes i bilagene, men for at overskueliggøre det, er det her delt op i 3 mindre sekvensdiagrammer, for hver eventhandler. Bemærk at sekvens diagrammerne er forsimplede i forhold til den reelle kode, ligeledes for at gøre dem mere overskuelige.



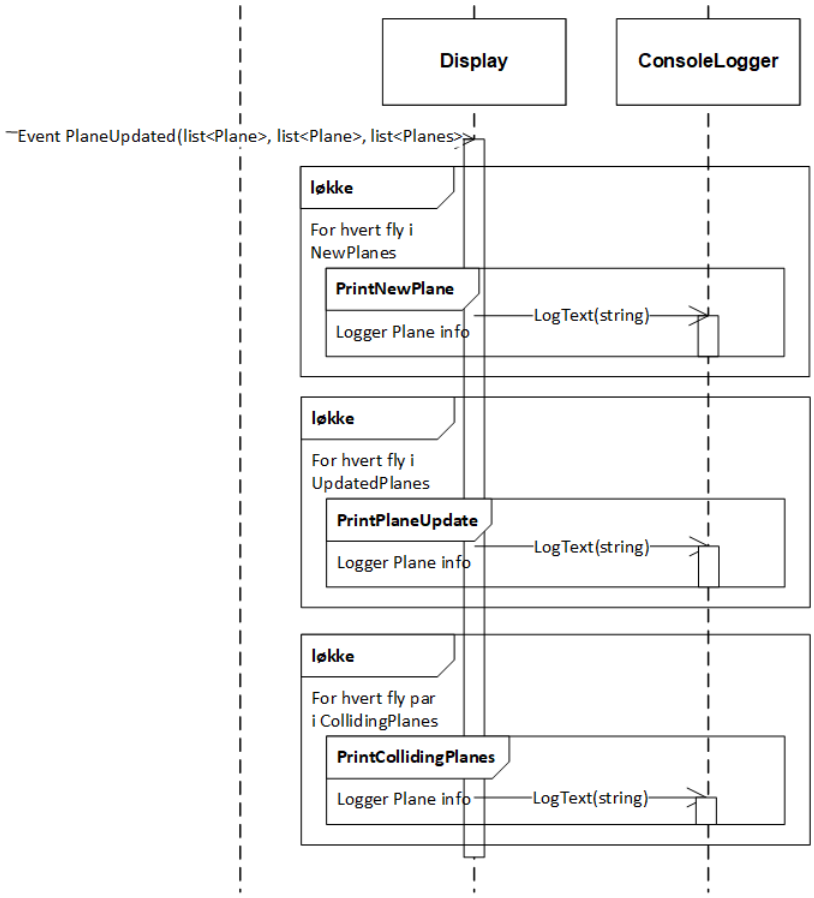
Figur 3: Sekvens Diagram for eventhandler, som håndtere modtagelsen af transponder strings

På figur 3 ses sekvensdiagrammet for håndteringen af det event som dannes, når transponder data modtages. Her formateres dataet med dataformatter til Plane klasser, og der sendes et nyt event, med en liste over nye planes.



Figur 4: Sekvens Diagram for eventhandler Plane Manager, som udregner værdier for Planes, og tilføjer fly til den rigtige liste

På figur 4 er sekvensdiagrammet for eventhandleren, som bearbejder det modtagne fly data. Flyene tilføjes lister alt efter om de er nye eller om de allerede er registrerede. Hvis de er registrerede, udregnes hastighed og retning. Der checkes i øvrigt for, om der er opstået seperation conditions (kaldet colliding i systemet). Disse gemmes i en 3. liste, og skrives til en logfil. Endeligt sendes event med de 3 lister. Bemærk at sekvens diagrammet er forsimplet, og eksempelvis mangler enkelte af udregnings funktionerne.



Figur 5: Sekvens Diagram for eventhandler Display, som sørger for at udskrive information om Planes til konsollen

På figur 5 ses eventhandleren for de bearbejdede fly lister. Denne eventhandler sørger kun for at udskrive dataet til konsollen, hvilket gøres med ConsoleLogger klassen. Bemærk at de forskellige print funktioner kalder LogText mange gange, med de forskellige værdier for et plane.

1.4 Designovervejelser

Events

Som det afspejles i klasse- og sekvensdiagrammet er systemet meget eventbaseret. Det er lavet fordi, det giver et godt logisk flow, og let at teste.

Dependency Injection (DI)

Alle dependencies i systemet er lavet ved brug af DI. Begrundelsen er, at det bliver rigtigt let at teste med NUnit og NSubstitute, og det giver en mere modificerbar og opdelt kode.

Ansvarsfordeling af klasser

Der er brugt i designet at fordele anvarsområderne af klasserne så vidt muligt, som det også kan ses på klassediagrammet. Men klassen PlaneManager, der modtager et event fra TRclient, bruger Calculator, CollisionLogger og sender et event til Display'et. Den har rigtig høj kobling, og et par ansvarsområder. Der kunne i designovervejelsen ikke ses en måde, man kunne undgå den klasse, der kobler dele af systemet sammen.

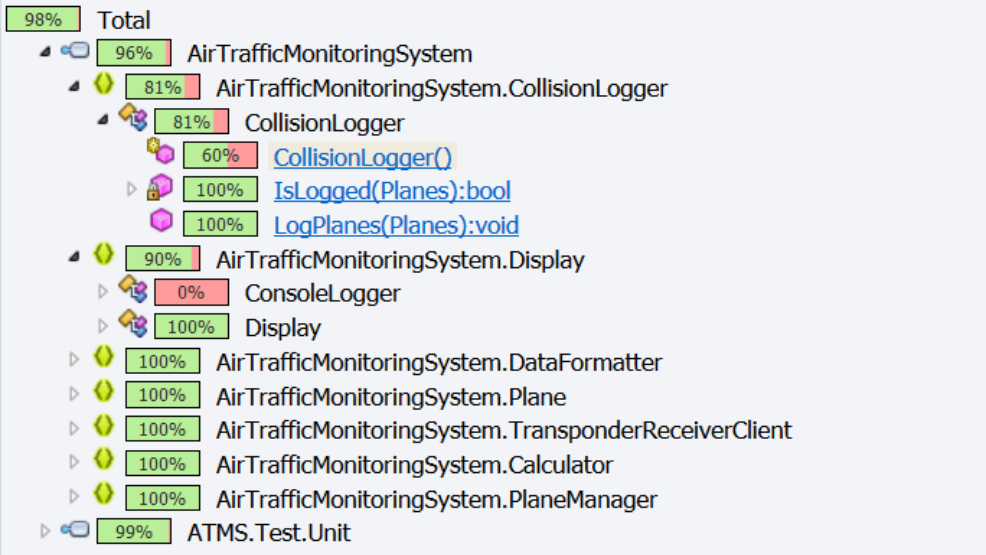
Calculator

Systemets lommeregner klasse "Calculator" er implementeret som en statisk klasse. Siden lommeregneren kun skal returnerer data baseret på parameterinputs i metoder, er der ingen grund til at oprette et objekt. Lommeregneren er også implementeret som en "dum" eller "uvidende" lommeregner. Det vil sige, at den får rene tal som inputs og derfor ikke kender til plane klassen.

2 Arbejdsproces

Til at starte med blev der lavet et overordnet design over systemet. Derefter tog alle medlemmer i gruppen en klasse. Den klasse lavede hvert enkelt gruppemedlem med tilhørende test. Når både klassen og tests var færdige, tog man en ny klasse. Grunden til at det er blevet gjort på den måde, er at det øgede gruppens arbejdseffektivitet. Samtidigt blev der skrevet gode tests til alle klasse med det samme. Ligeledes havde alle medlemmer af gruppen et dybt indblik i en lille del af systemet, sådan at gruppens medlemmer hver er eksperter i dele af systemet.

I starten af opgaven blev den fælles Github solution koblet på en Jenkins Server for at have Continous Integration(CI) etableret. Der er fælles mening om i gruppen, at CI har været et godt værktøj igennem opgaven. Der har været mulighed for at se den fælles udvikling af systemet hele tiden, også selvom man ikke sidder sammen. Udover de Unit Test man kunne se på Jenkins serveren, så er Coverage testen også blevet brugt. Den er blevet brugt til at se om der er dele af koden som gruppen har glemt at teste. På figur 6 kan der ses et udsnit af Coverage Reporten for systemet.



Figur 6: Coverage Report

Som det kan ses på figuren, er der ikke 100% coverage for systemet. Denne mangel kommer hovedsagligt fra CollisionLogger og ConsoleLogger. ConsoleLogger udskriver til konsollen (Console.WriteLine()) og kan derfor ikke køres i et test miljø. Derfor testes metodekaldet ConsoleLogger.LogText med NSubstitute, som viser, at metoden bliver kaldt med de rigtige parametre.
CollisionLogger når ikke 100% på grund af følgende linjer i dens constructor:

```
Path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) + @"\LogFile.txt";
            if (!File.Exists(Path))
            {
                var stream = File.Create(Path);
                stream.Close();
            }
```

Vi mistænker, at Jenkins ikke sletter filen efter dens skabelse, og at testen derfor ikke kører koden, efter filen er skabt.

3 Konklusion

Igennem arbejdet med opgaven, er har gruppen gjort sig gode erfaringer med software test, git, continous integration og ikke mindst det at arbjede som et hold på et software projekt. Projektets produkt er et funktionelt system, som lever op til de krav der var sat for systemet. Systemet kunne effektiveres og optimeres, så det ville fungerer hurtigere, men pga. systemetes begrænsede størrelse, og at der i arbejdet har været fokuseret på tests og design, er dette ikke gjort. Opdelingen af softwaren, og det faktum at hvert stykke software har været gennemtestes individuelt, har klart ført til at systemet var lettere at integrere, hvorfor der heller ingen problemer med integration har været. Ligeledes har valget med brug af events i systemet underbygget dette. Brugen af Jenkins, har medvirket til at hele gruppen hele tiden har haft overblik over, hvor langt systemet er, og om alt har fungeret som ønsket. Erfaringer gjort i forbindelse med projektet, vil blive ført videre, og kan let overføres til eksempelvis semesterprojekt, eller andre større opgaver.