Handin 2 – ATM Del 1

Gruppe nummer: Gruppe 17

Navn	Studienummer	Email
Anders Fibiger	201611672	201611672@post.au.dk
Viktor Søndergaard	201610466	201610466@post.au.dk
Arni Medic	201611674	201611674@post.au.dk
Aram Al-Sabti	201406080	201406080@post.au.dk

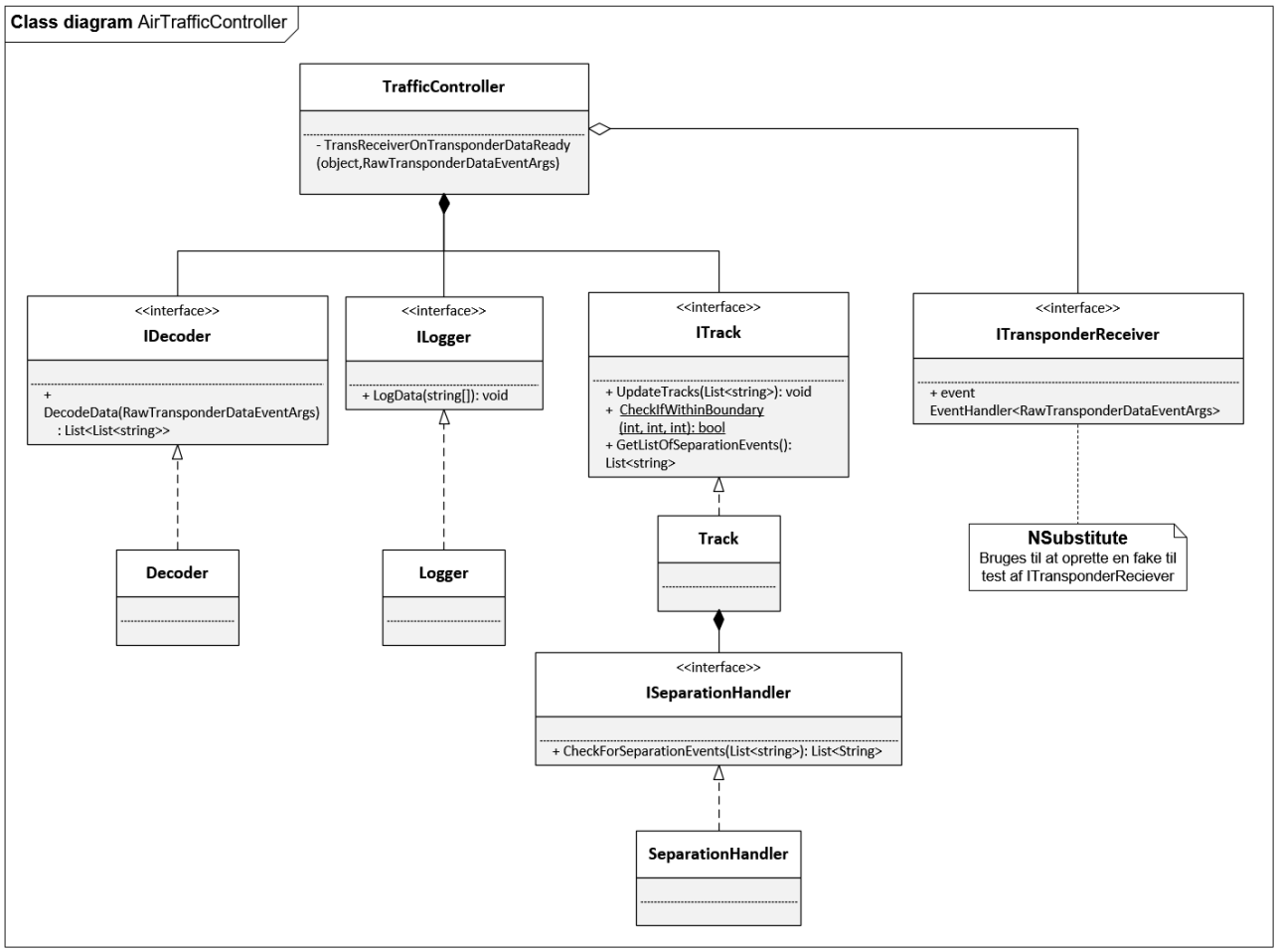
URL til Jenkins build job: http://ci3.ase.au.dk:8080/job/SWT2_AAA/

URL til Jenkins coverage: http://ci3.ase.au.dk:8080/job/SWT2_AAA_dotCover/

URL til GitHub repository: https://github.com/andersfibiger/SWT2

Software Design

Der er på baggrund af systembeskrivelsen er der udarbejdet et design af AirTrafficController.



Figur 1 - Klassediagram for AirTrafficController

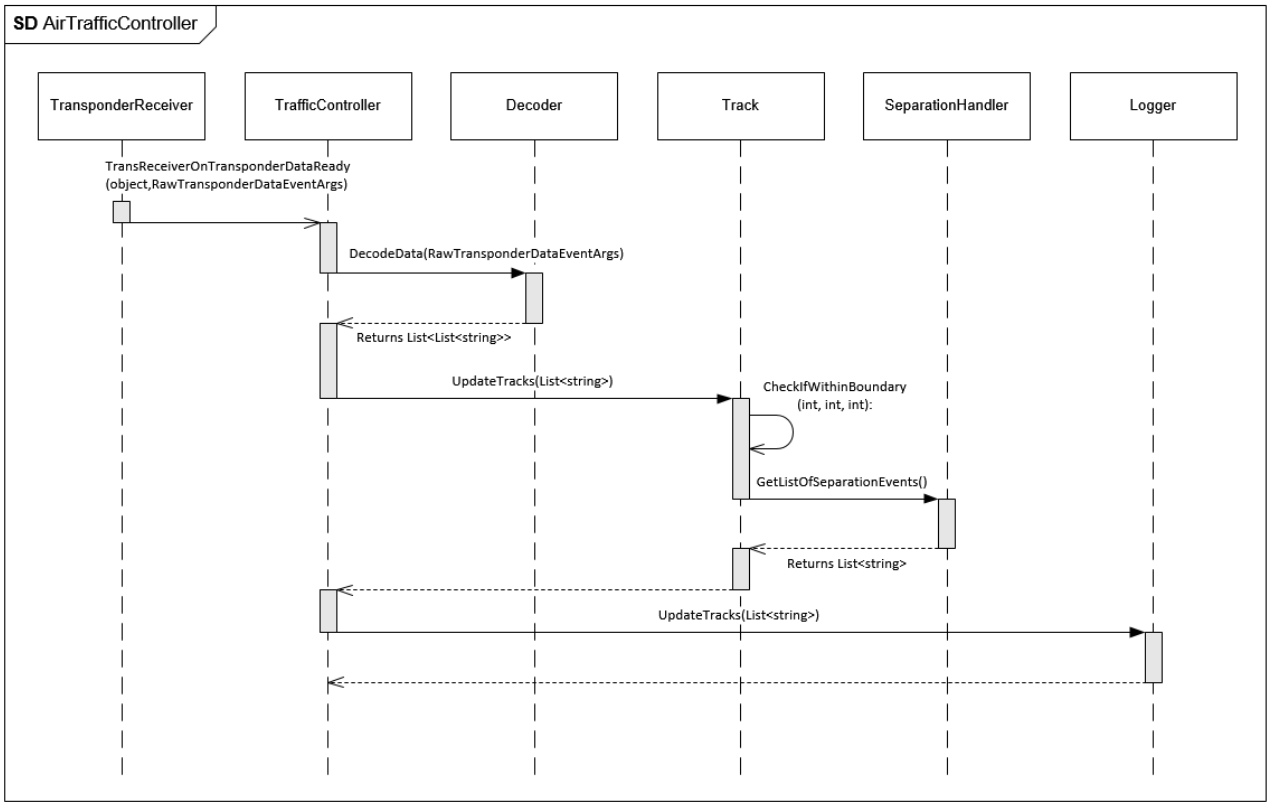
På overstående figur 1 kan klassediagrammet for AirTrafficController ses. Klassen TrafficController bruges til at knytte underklasserne sammen på en struktureret måde, samt at benytte klassen til at subscribe på eventet fra TransponderReviever. TrafficController består af en række interfaces, herunder IDecoder, ILogger, ITrack, samt ITransponderReciever. ITrack består yderligere af ISeparationHandler interfacet.

At der gøres brug af interfaces, gør først og fremmest at koblingen er mindre mellem baseklassen og den konkrete klasse. Derudover er det er smart at benytte i forbindelse med unit.test, idet vi kan implementere en fake i stedet for at ændre den reelle klasse. Da vi netop gør brug af interfaces, er baseklassen ligeglad med de konkrete klasser. Dette betyder, at der kan gøres brug af dependency injection, ved at konstruere baseklassen med de fake afhængigheder.

I forhold til AirTrafficController-systemet er der valgt at lægges vægt ITransponderReciever, i forhold til at gøre brug dependency injection. Det er netop fordi klassen har ukontrolleret adfærd (output), hvilket betyder at en fake er nødvendig. Her er der valgt at benytte frameworket

NSubstitute. Vi har valgt at benytte dette framework, da det er nemt at substituere en ”ægte” klasse med denne fake, samt at der ikke er behov for at explicit implementere vores fakeklasser.

For at vise klassernes interaktioner nedad en tidslinje, er der udover klassediagrammet blevet udarbejdet et sekvensdiagram.



Figur 2 - Sekvensdiagram for AirTrafficController

På overstående figur 2 kan sekvensdiagrammet for AirTrafficController ses. Her ses det at eventet opstår fra TransponderReciever og derefter sendes videre til TrafficController, som håndterer eventet. Herfra kaldes Decoder, som behandler det modtagne data på ønsket vis. Når den proces er færdig, overtager TrafficController igen og kalder Track-klassen med kommando om at opdatere listen af Tracks. Da Track kun måler indenfor et bestemt område tjekker den for om de modtagne Tracks er indenfor området. Hvis flyene så er for tæt på, hvilket er bestemt af nogle specifikke begrænsninger, så bliver eventet for SeperationHandler kaldt. Denne klasse returnerer listen for de fly som er i farezonen. Til sidst returneres der til TrafficController, som sørger for at printe den opdaterede liste ud via Logger-klassen.

Arbejdsfordeling

Vi har lavet softwarearkitekturen på studiet og lavet en del pair programmering sammen. Det har vi valgt at gøre for at alle har den fulde forståelse af systemet. Dette gjorde vi i starten, således at alle var med på hvad opgaven gik ud på. Herefter delte vi de forskellige test klasser ud. Vi delte det lidt ud efter lyst, men sørgede for at hver klasse var i sin egen fil. I starten havde vi lavet en testklasse, men vi fandt ud af, at det var meget smartere at dele hver klasse op i en test for denne klasse. Derved kunne man spare meget tid ved at dele opgaverne ud, således at en til to personer sad på en klasse og lavede tests for denne. Alt i alt var arbejdsfordelingen meget fin og var en effektiv måde at få udviklet systemet på. Ved hjælp af github og CI serveren, gjorde det også måden vi arbejdede på meget nemmere fordi vi netop fik testet systemet hele tiden. Vi blev også opmærksomme på, hvis der mangel på tests ved hjælp af CI.

CI Server

Continuous Integration har til tider været en stor hjælp når vi har arbejdet hjemmefra, men hvis man skal kunne bruge det effektivt, skal man også have nogle regler sat op og have inddelt projektet ordentligt til at starte med. Hvis man ikke gør det, ender man med at bruge meget tid på at merge, og genstarte Visual Studio når den brokker sig over diverse ting.

Så for at undgå disse merge og Visual Studie errors, har vi valgt at vi arbejder på hver vores fil når det gælder test, fordi ellers ender vi med at spilde mere tid på at skulle ordne fejl, end vi vinder på at bruge CI. Overordnet har det hjulpet på vores projekt, da vi alle sammen har kunne arbejde på det samtidig, og alle har kunne bidrage til det. Men selvom vi har brugt CI valgte vi ofte at pair programmere når vi sad oppe på studiet, især til at starte med da vi ikke havde inddelt projektet ordentligt. Derudover var det godt at bruge til at se, hvordan ens test var. Vi havde en CI server til unit test, hvormed den byggede hurtigere, og en til coverage, så vi kunne se hvordan koden var dækket ind. Coverage kunne bruges til at se, hvor man manglede at lave tests, hvilket viste sig at være meget effektivt, så man kunne se hvor man skulle have lavet tests næste gang.

Når vi skal bruge CI til næste gang ved vi med sikkerhed at man lige skal have sat det ordentligt op til at starte med, og have nogle faste regler.

Enhedstests

Som nævnt er testene delt op i klasser for deres respektive klasse, og yderligere er der testmetoder i disse klasser, som tester klassen funktionalitet. Vi har lagt vægt i at få testet systemet funktionaliteter. Dette har medført at vi har haft fokus på at teste hver gang, at vi har haft et nyt modul færdigt. Til dette brugte vi også CI og den coverage report, der blev genereret, til at vurdere hvor godt funktionaliteten virkede.

I vores tests har vi gjort brug af forskellige testmetoder. Dette indebærer for eksempel Boundary Value Analisys (BVA) til at teste om grænseværdierne for vores tracking område overholder specifikationerne.

Der er blevet overvejet bruget af stubs og mocks, hvoraf valget faldt på NSubstitute frameworket, da denne gør arbejdet for os, og vi ikke skal implementere vores egne fake klasser. På den måde har vi kun lavet fakes og brugt NSubstitute med dem til at erstatte vores afhængigheder med fakes. I den forbindelse har vi også brugt principperne om dependency injection. Specifikt har vi brugt constructor injection i vores

TrafficController. Alt i alt har der været fokus på at få lavet testene og det har været formålet med opgaven.

Konklusion

Der er blevet arbejdet primært med tests for systemet. Dette afspejler sig i vores program, da dette ikke er blevet fuldt implementeret endnu. Der er blevet brugt forskellige testmetoder til at få testet systemet ordentligt og herefter brugt continuous integration således at systemet blev testet hele tiden. Der er blevet forsøgt at få en god opfyldelse af coverage rapporten og dette har ført til et godt stykke arbejde.