Handin 3 – ATM Del 2

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 Software Metrics build job:
http://ci3.ase.au.dk:8080/job/AAA_Software_metrics/configure

URL til Static Analysis job: **http://ci3.ase.au.dk:8080/job/AAA_static_analysis/**

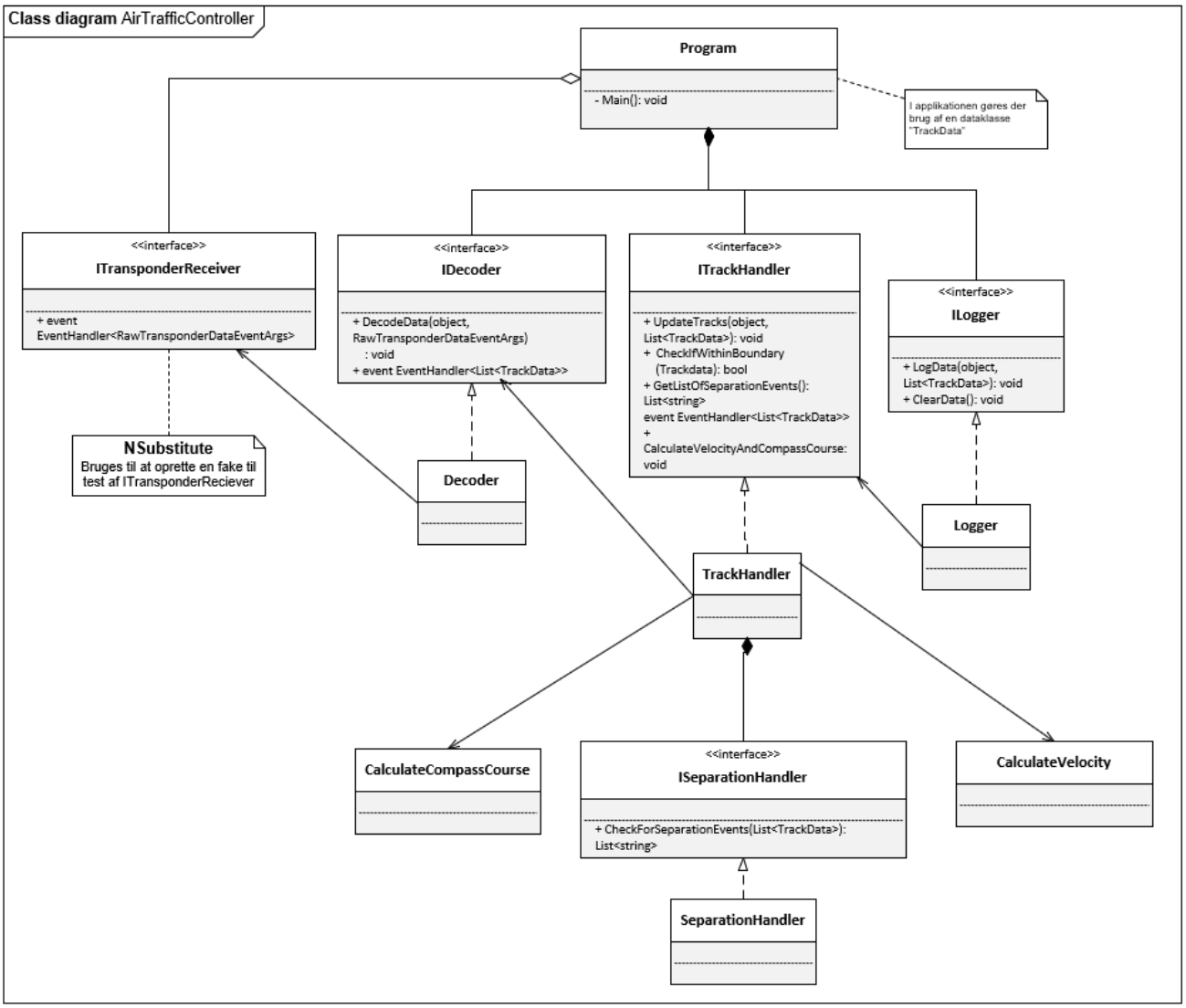
URL til Jenkins coverage: **http://ci3.ase.au.dk:8080/job/SWT2_AAA_dotCover/**

URL til Integration build job: **http://ci3.ase.au.dk:8080/job/AAA_IntegrationTest/**

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

Software Design

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



Figur 1 - Klassediagram for AirTrafficController

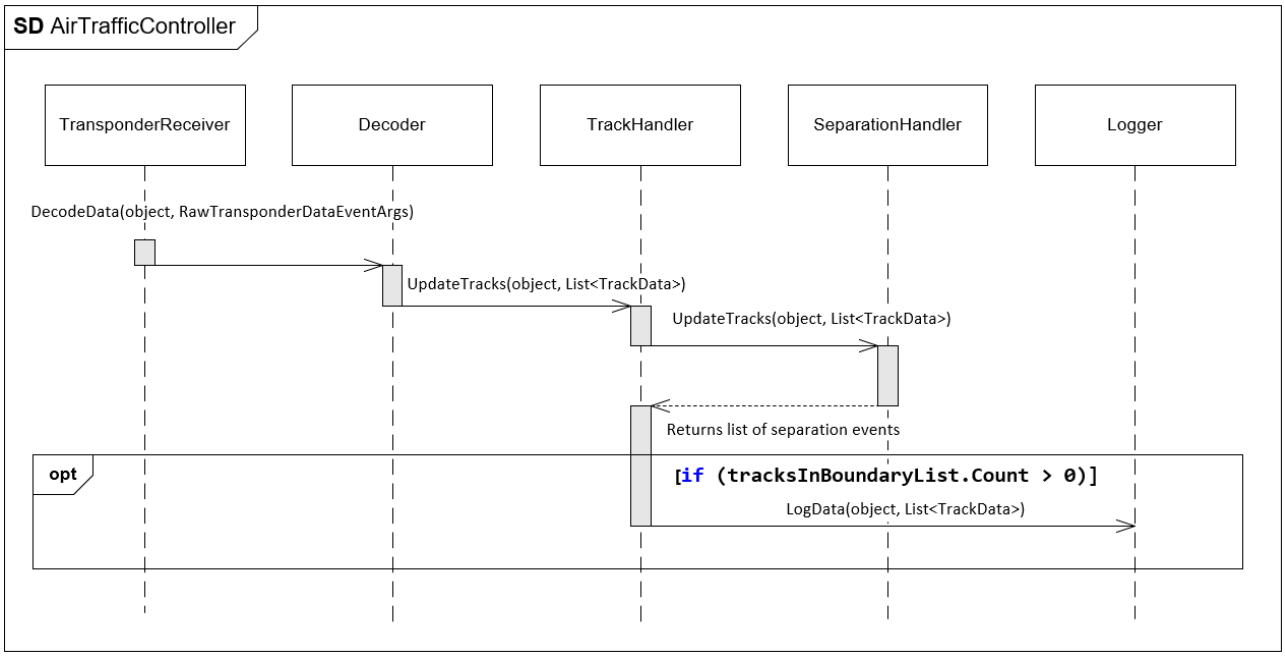
På overstående figur 1 kan klassediagrammet for AirTrafficController ses. Klassen Program bruges til at knytte underklasserne sammen ved at instantiere dem. Program består af en række interfaces, herunder IDecoder, ILogger, ITrackHandler, samt ITransponderReciever. ITrackHandler består yderligere af ISeparationHandler interfacet og udregningsklasser, CalculateCompassCourse og CalculateVelocity, som henholdsvis står for udregning af kompasretningen og hastigheden.

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 klasse 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 fake klasser.

I forhold til den tidligere version af denne hand in, er softwaredesignet ændret fra blæksprutte design til pipeline design. Den primære grund til dette er, at det er nemmere at fejlfinde i forhold til integrationstest, og generelt set er det nemmere at udføre en integrationstest. Af den grund interagerer alle underklasser nu via events. I den forbindelse er både klassediagrammet og sekvensdiagrammet ændret til at stemme overens med det nye softwaredesign. Her ses det først og fremmest på klassediagrammet at fx Decoder lytter på ITransponderReciever. Dette er kun en af de flere events der er kommet på.

For at vise klassernes interaktioner nedad en tidslinje, er der udover klassediagrammet blevet udarbejdet et sekvensdiagram. Her ses det hvordan pipeline designet kommer til udtryk ved at underklasserne interager mellem sig, og ikke længere én klasse som interager med alle underklasser, som vi tidligere har gjort. Dermed kan vi under integrationstesten nemt se, hvor det er på sekvenslinjen, det går galt.



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 Decoder, som håndterer eventet. Decoder behandler det modtagne data, hvorefter det sendes videre til TrackHandler, som på samme måde håndterer det modtagne event. Det sendes videre til SepationHandler, hvorefter den returner listen af fly som er for tæt på hinanden, hvis der altså er nogen. Derefter sendes det

videre til Logger, som printer listen af fly indenfor airspacet. Hvis listen af fly indenfor airspacet er tom, bliver der ikke printet noget.

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. Da vi besluttede at ændre vores softwaredesign til pipeline design, gjorde vi det også i fællesskab, således at alle var enige om designet, og derfra kunne vi implementere de nye krav og tests, som der skulle bruges.

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.

Vi har her anden gang i projektet haft meget bedre succes med Continuous Integration. Den primære grund til dette, har været vi har lavet selve designet og gjort klar til de forskellige features oppe på skolen. Efter dette har vi så kunne uddele de forskellige features til gruppe medlemmer. Det har gjort til at vi næsten ikke har skulle bruge noget tid på irriterende merge errors, da de forskellige features laves i forskellige filer. Det har gjort oplevelsen med CI meget bedre, da man med merge errors ofte kan føle der er meget irriterende spildtid, og det kan nemt skabe en dårlig stemning, da man kan begynde at skyde skylden for forskellige ting på hinanden. Men da vi primært har valgt at uddele de forskellige features, og brugt Trello, som er et online task board, har vi kunne holde styr på vi heller ikke arbejdede på de samme features.

Enhedstests

Som nævnt er testene delt op i klasser for deres respektive klasse, og yderligere er der testmetoder i disse klasser, som tester klassens funktionalitet. Vi har lagt vægt i at få testet systemets 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 Program-klasse. Alt i alt har der været fokus på at få lavet testene og det har været formålet med opgaven.

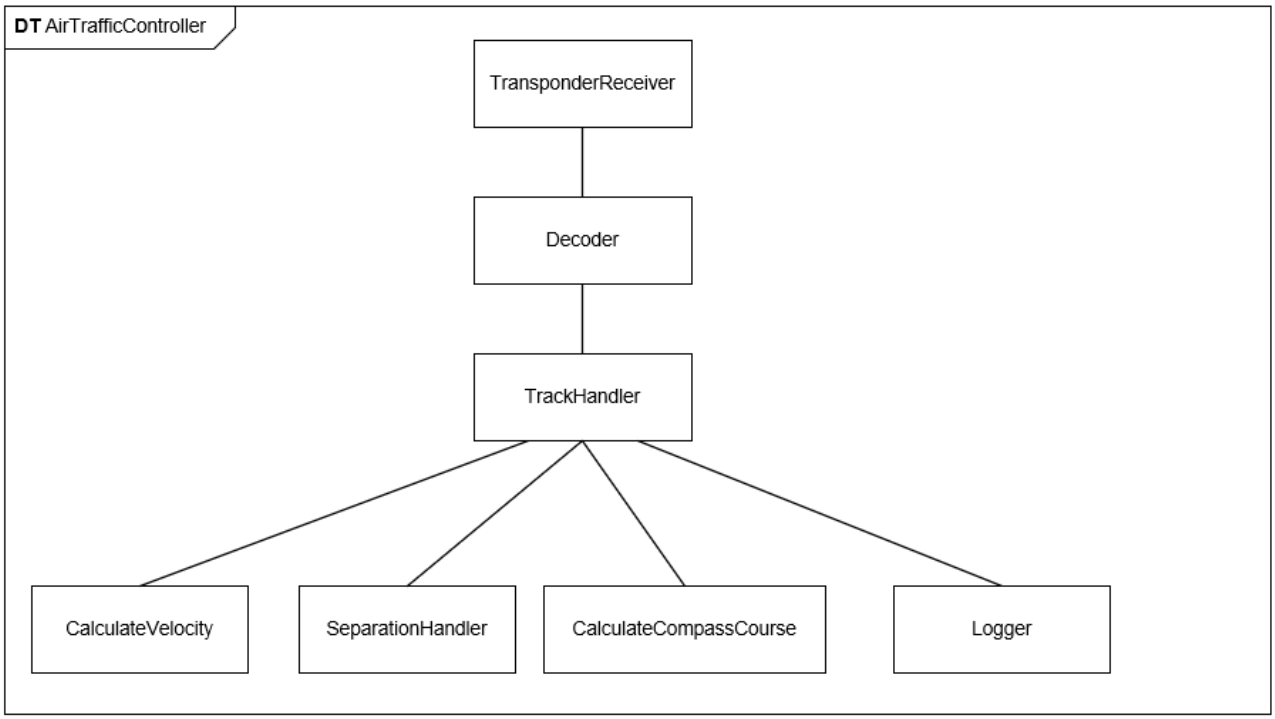
Det skal dog også lige siges at en række metoder i Loggeren er blevet testet manuelt, da disse metoder skriver til konsollen. Det ligger udenfor projektets omfang at teste det automatisk.

Generelt set er det svært at opnå 100% code coverage. Fx har vi Program klassen som instantierer alle underklasserne. Selvom dens opgave er kun at gøre dette, så påvirker det test coverage rapporten, da programmet ikke kan se forskel på hvad der er relevant at teste og hvad der ikke er.

Dependency tree

I forbindelse med integrationstest er der først udarbejdet et dependency tree. Dette ses på Figur 3, og er første skridt i integrationstestplanlægningen. Program er der hvor main er og altså der hvor alle de andre klasser bliver initialiseret, og derfor er den ikke med i dependency tree. TransponderReceiver laver et event, som Decoder lytter på. Decoder afhænger derfor af TransponderReceiver. TrackHandler håndterer trackdata, som er blevet decodet fra Decoder, hvorfor TrackHandler afhænger af Decoder. Til sidst har vi det nederste lag i vores dependency tree. Her har vi de fire klasser, som alle afhænger af TrackHandler.

Eftersom klasserne kommunikerer med events, kender klassen til hinanden begge veje, og dette er ikke tegnet på Figur 3 for overskuelighedens skyld. Det betyder eksempelvis, at TrackHandler og Logger begge kender til hinanden. Det samme gælder for alle forbindelserne, da alle kommunikerer med events grundet vores nye pipeline design.



Figur 3 – dependency tree for AirTrafficController

Integrationsplan

Vi har valgt at bruge bottom up integration, da den lignede en meget fin metode at bruge til dette system ud fra vores dependency tree. En anden grund til at bottom up er blevet brugt er, at det er nemt at dække over alle ens interfaces, samt at strategien for at gribe en integrationstest an udgør, at vi næsten ikke behøver gøre brug af stubs. Det er nogle af de gode egenskaber der er ved bottom up. Den har selvfølgelig også nogle ulemper. Eftersom man netop starter fra bunden, er der en del services, der skal tjekkes fra bunden. Det kan godt komme til at koste nogle kræfter at få opsat, men i sidste ende, sammenlignet med andre strategier, besluttede vi os for at vælge bottom up, og det virkede også godt. Integrationsplanen ses på Figur 4, hvor første step er bindeleddet mellem TrackHandler og de fire underklasser til denne, ligesom det ses på figur 3. I denne integrationstest er Decoder en stub og det er dens integration med de andre fire klasser, som der testes. Dette er det første step.

Step	TransponderReceiver	Decoder	TrackHandler	CalculateVelocity	SeperationHandler	CalculateCompassCourse	Logger
1		S	T	X	X	X	X
2	S	T	X	X	X	X	X
3	T	X	X	X	X	X	X

Figur 4 - integrationsplan

Konklusion

Der er i forlængelse af tidligere aflevering blevet implementeret de resterende metoder og krav til systemet, samtidig med at alle unit test vedligeholdes. Der er blevet brugt forskellige testmetoder til at få testet systemet ordentligt og herefter brugt continuous integration således at systemet blev testet hele tiden undervejs. Der er blevet forsøgt at få en god opfyldelse af coverage rapporten og dette har ført til et godt stykke arbejde. Dog er der som sagt nogle test hvor der er udført manuelle test, hvilket naturligvis påvirker test coverage, men dog ikke med en betydelig procent. Til sidst er der udarbejdet en integrationstest for at teste om alle klasserne kan interager med hinanden på korrekt vis.