Software Test - Air Traffic Monitoring Part 2
Gruppe 21

Chris Wilhjelm Olesen, 201205932, AU453533
Kasper Gnutzmann Andersen,201607263, AU569735
Mette Christine Jacobsen, 201610470, AU565978
Mikkel Bleeg Carstensen, 201611667, AU557112

November 2018

Formål:

Formålet med denne opgave er at stifte bekendtskab med planlægning af integrationstest samt implementering heraf.

Indhold

1 Introduktion

I dette opgave har man skulle lave en monitor, der skulle tracke flyaktivitet, indenfor et givet luftrum. Her skulle man kunne se om flyene var på vej i kollision.

2 Arbejdsfordeling

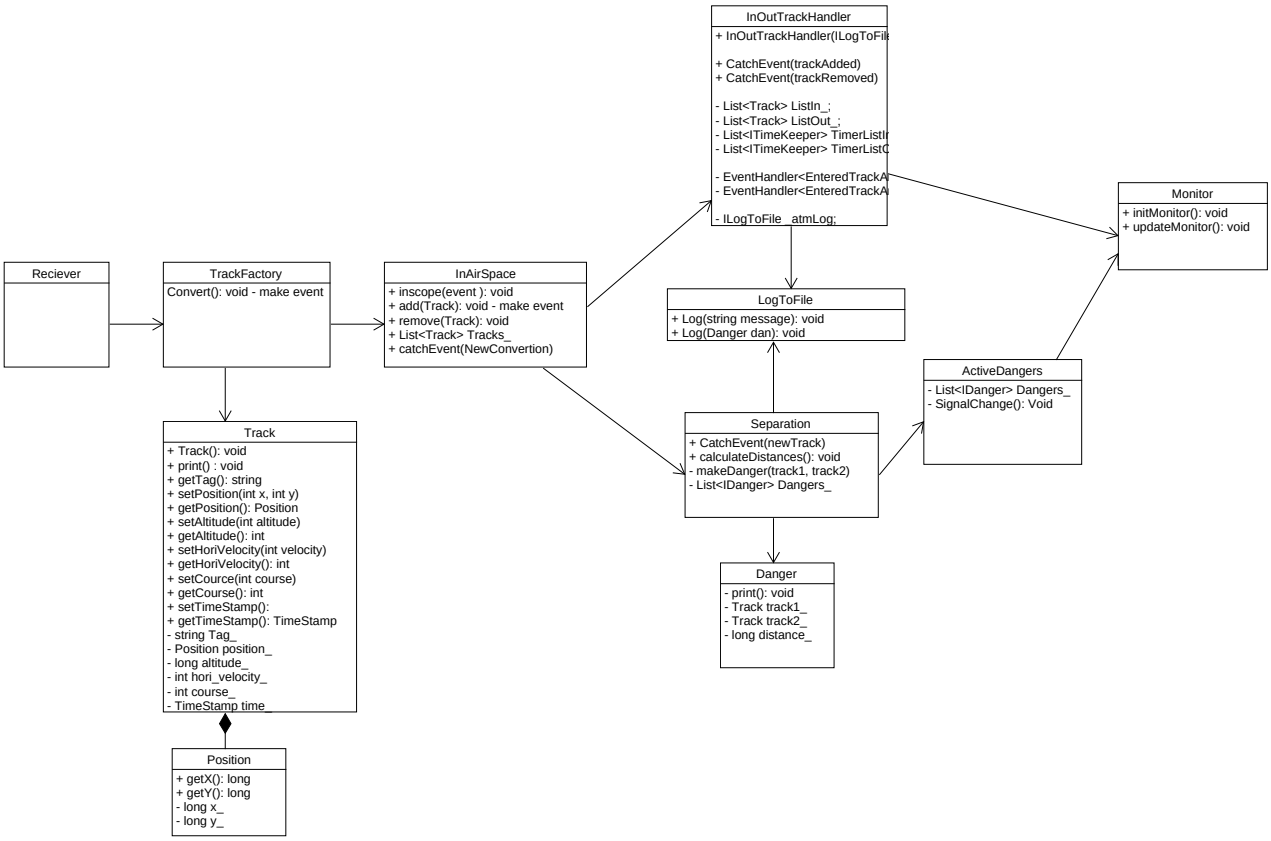
I gruppen er der valgt at fordele arbejdsopgaverne i klasser og tests, hvor hver person har haft en klasse som skulle implementeres, hvorefter den skulle testes. Dette gjorde, at personen som skulle teste klassen, havde bedre kendskab og større viden, omkring hvad der skulle testes og hvad der skulle undersøges samt uddybes. Dette valg blev taget, da det gav bedre mulighed for at opnå 100% Coverage.
Ansvaret for de forskellige klasser blev ikke opdelt efter et bestemt formål. Opgaverne blev delt ud på de forskellige personer, hvor et medlem i gruppen kunne have ansvar for flere klasser, dog ikke så krævende, og en anden person kunne få en klasse, som kunne inkludere mere krævende implementation.
Efter implementationen blev der lavet tests til de gældende klasser, dette blev opdelt i, at det medlem af gruppen som havde lavet klassen, skulle teste klassen.
På trods af at gruppen var opdelt hver for sig, var der dog altid mulighed for at få reviewet sin kode af de forskellige medlemmer, samt få hjælp til at lave tests.
Ved nogle klasser er der oprettet interfaces som test objekter, det er blevet gjort i klasserne LogToFile og Separation.

3 Design

Under dette afsnit vel designet af gruppens projekt samt de overvejelser der er gjort af disse.

3.1 Klassediagram

På bagrund af opgaven er der blevet udviklet et klasse diagram, til at danne et overblik over de forskellige blokke, gruppen forventer skal indgå i implementeringen af applikationen. Dette klasse diagram ses i figur 1

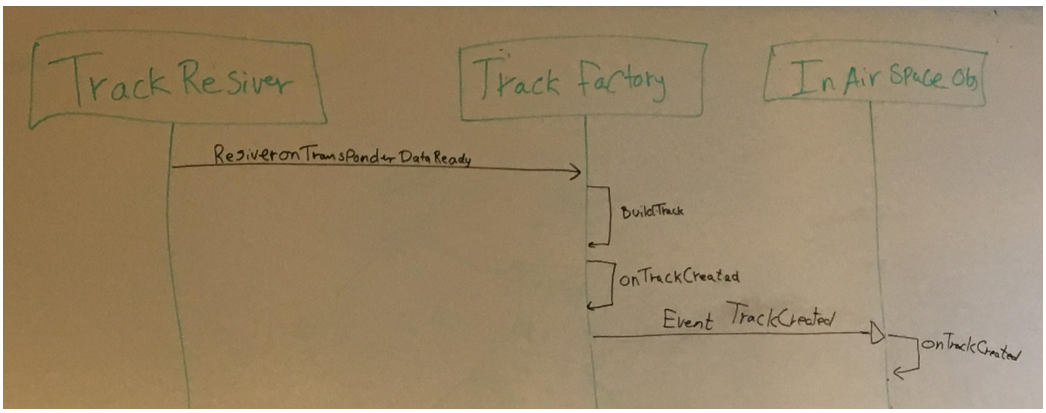


Figur 1: Klasse diagram

Som det ses på figur 1 er de fleste klasser forbundet med association, eftersom der anvender events til at sende data imellem klasserne er der egentlig tale om en bidirectional association, da den ene klasse subscriber på den anden, som sender data retur. Der er altså tegnet association i dataflow retningen.

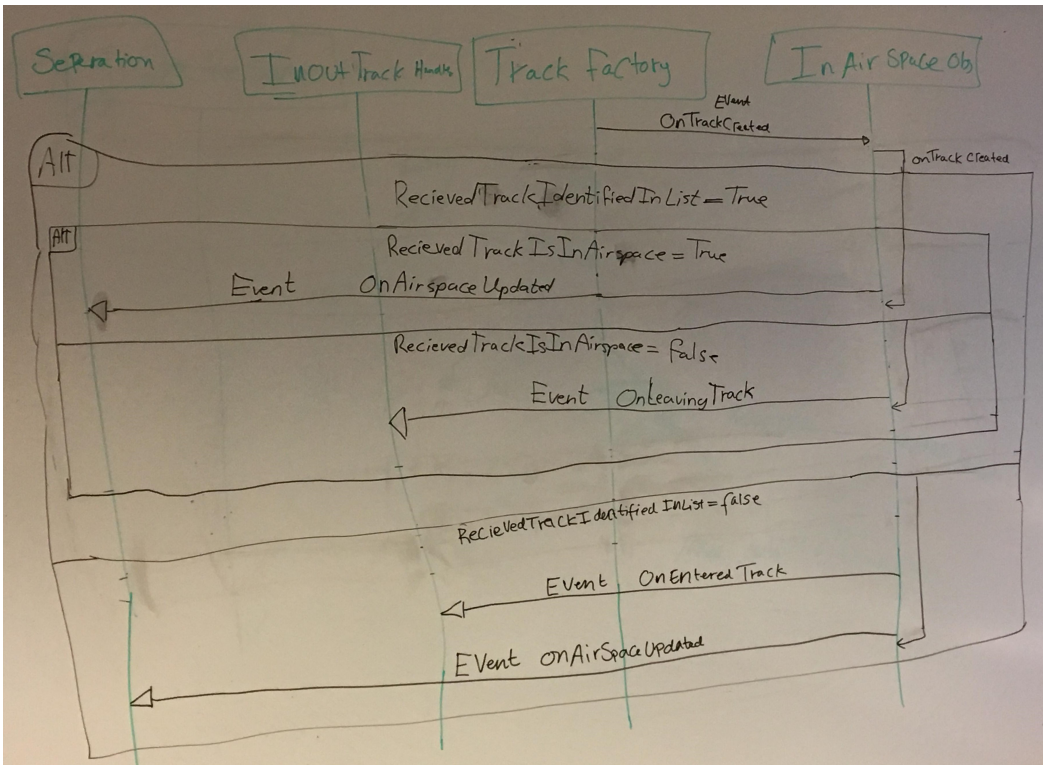
3.2 Sekvens Diagram

Her er de enkelte klasser beskrevet med sekvens diagrammer, uddybning ses under hvert diagram.



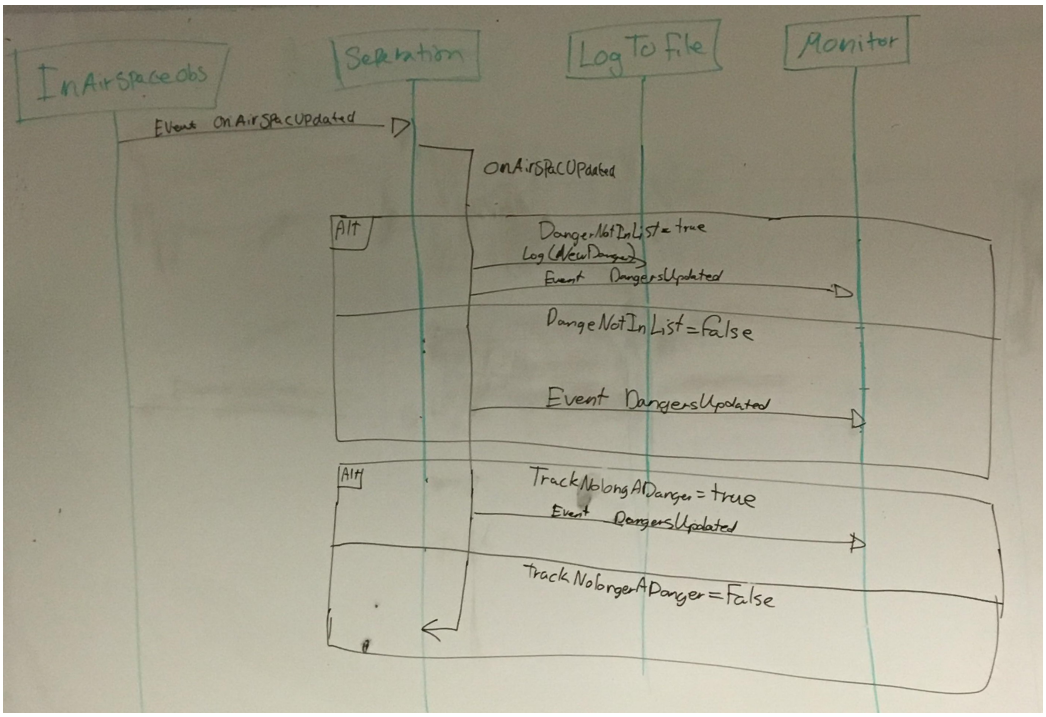
Figur 2: Sekvensdiagram af klassen TrackFactory

Som det ses på figur 2 trigger ReceiverOnTransponderDataReadyEvent eventhandleren for TrackFactory-klassen. For hver tekst-streng der modtages, som defineres oprettes et Track objekt som sendes videre via TrackCreated eventet, som trigger eventhandleren for InAirSpaceObserver-klassen. - Disse to klasser var tidligere samtlet under én klasse, men blev refaktoreret for at adskille klassernes ansvar.



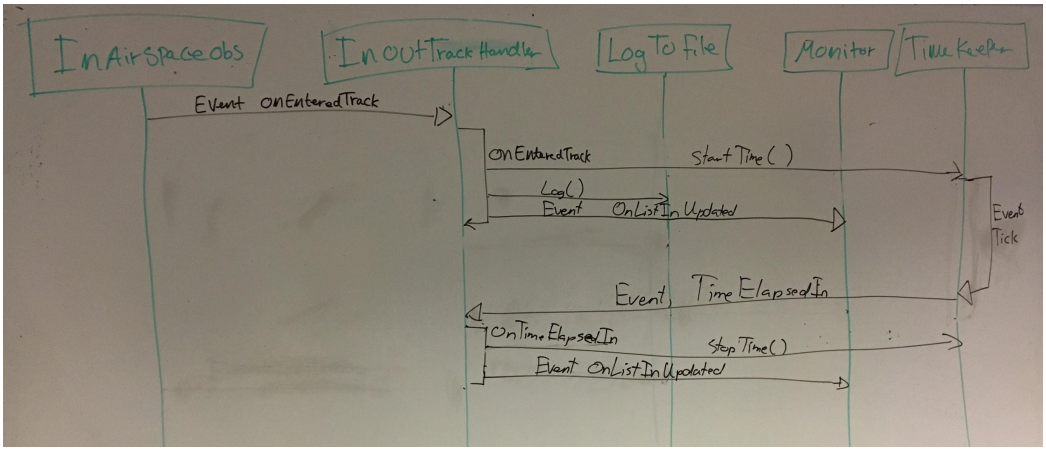
Figur 3: Sekvensdiagram af klassen InAirspace

Som det ses på figur 3 videresender InAirSpaceObserver tre typer af events, EnteredTrack, LeavingTrack og AirSpaceUpdated. Når eventhandleren for TrackCreated eventet kaldes afgør klassen om det nyoprettede tracks tag allerede er at finde i listen for tracks der overvåges, og om hvorvidt tracket er inden for det overvågede air space. Er Tracket allerede på listen listen, men uden for rammerne, må flyet der repræsenteres lige have forladt air space'et, og der sendes et leavingTrack event, med et opdateret track, samt listen af tracks, hvor det forladende fly er fjernet i et AirSpaceUpdated event. Er tracket på den overvågede liste og inden for rammerne, er det et track der skal opdateres, hastighed og kurs opdateres, den gamle instance af tracket fjernes fra listen og det nye tilføjes, hvorefter der sendes et AirSpaceUpdated event. Er tracket ikke på listen, men inden for rammerne er der tale om et nyt fly der kommer ind i air space'et, det nye track tilføjes listen og der sendes et EnteredTrack og AirSpaceUpdated event, men respektivt det nye track og den opdaterede liste. - Er begge valideringer false kasseres tracket.



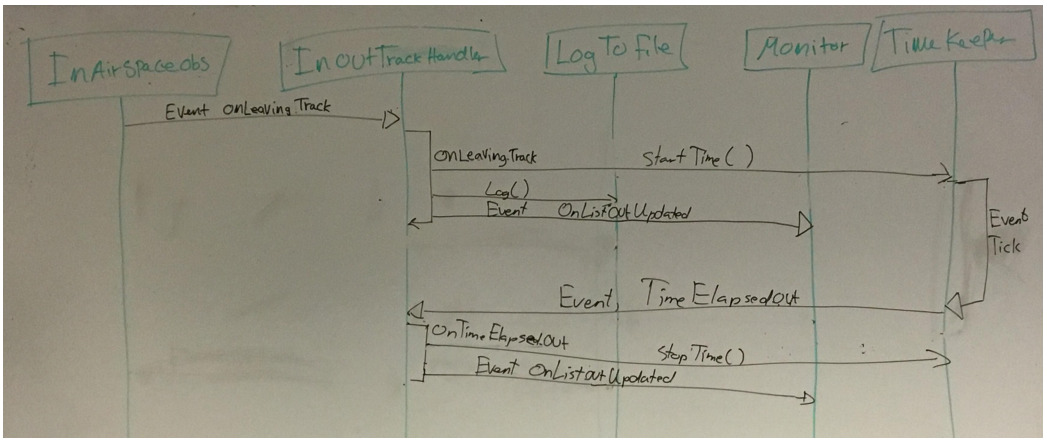
Figur 4: Sekvensdiagram for klassen separation

Klassen Separation håndtere om hvorvidt der er opstået et nyt separationEvent, i koden kaldet Danger. Dette gøres ved at det nye track sammenlignes med alle andre track i luftrummet, og hvis to er for tætte på hinanden oprettes en danger som gemmes i en liste af aktive dangers. Separation har også til ansvar at fjerne dangers fra listen, når en danger ikke længere af aktiv. Hver gang listen af aktive dangers er blevet opdateret skal der laves et event til monitoren. som fortæller listen er ændret.



Figur 5: Sekvensdiagram af klassen InOutTrackHandler, når nyt track kommer ind i luftrummet.

Klassen InOutTrackHandler håndter de events der kommer fra InAirSpaceObservator. SD ses i figur 5 og figur 6 Hvis der kommer et OnEnteredTrack event, vil InOutTrackHandler klassen starte en timer der håndtere tidtagningen for eventet. Hernæst vil den tilføje tracket til en liste som klassen indeholder, listIn, samt kalde eventer OnListInUpdated, der sender et event til Monitor om at der er kommet et nyt track ind i airspacet. Her efter bliver dette track skrevet til loggen, med tag, timestamp og enter, dette ses som SD på figur 5. Når der er gået 5 sekunder vil eventet TimeElapsedIn bliver kaldt, der søger for at timeren stoppes, samt at tracket hvis tid er overskredet 5 sek, slettes fra listen. Kommer der derimod er OnLeavingTrack event, vil klassen næsten gøre det samme. Dog er det events som OnListOutUpdated og TimeElapsedOut, og lister som ListOut der bruges i stedet. Dette SD ses i figur 6.



Figur 6: Sekvensdiagram af klassen InOutTrackHandler, når track forlader luftrummet.

3.3 Refleksion over Design

Gruppen har valgt at lave et pipeline baseret design med udgangspunkt i forrige projekt. Dette er tildels lykkes. Der er oprettet flere klasser end sidste projekt, som har reduceret ansvaret af nogle af klasserne.Derudover er der anvendt events til at sende data imellem de fleste klasser for at få mindre kobling mellem klasserne. Klassediagrammet er blevet mere beskrivende og har færre forbindelser end forrige projekt. Desuden er der igennem sekvens diagrammerne, som ikke helt overholder standarden, beskrevet under hvilke forhold klasserne kommunikerer.

4 Unit tests

Unit test bruges til at undersøge om en given klasses funktioner virker efter hensigten. Er en klasse afhængig af andre, eller benytter den dem, vil disse klasser i testen blive lavet til stubbe, fakes eller moks. H

4.1 Substitution: Stubs og Mocks

Foruden boundary value analysis, er klasserne forsøgt afkoblet fra hinanden således disse tests helt separat. Hvor der har været dependensies mellem klasser er forsøgt anvendt stubs eller mocks, disse er f. eks. oprettet ved brug af NSubstitute.

```
private ITrackFactory fakeTrackFactory_;
private ITrackOpticsProvider fakeOpticsProvider_;
private InAirSpaceObserver _uut;

[SetUp]
[Author("Kasper Andersen")]
0 references | Kasper Andersen, 3 days ago | 1 author, 2 changes
public void Setup()
{
    fakeOpticsProvider_ = Substitute.For<ITrackOpticsProvider>();

    fakeTrackFactory_ = Substitute.For<ITrackFactory>();

    // Inject the fake TDR
    _uut = new InAirSpaceObserver(fakeOpticsProvider_);

    fakeTrackFactory_.TrackCreated += _uut.OnTrackCreated;

}
```

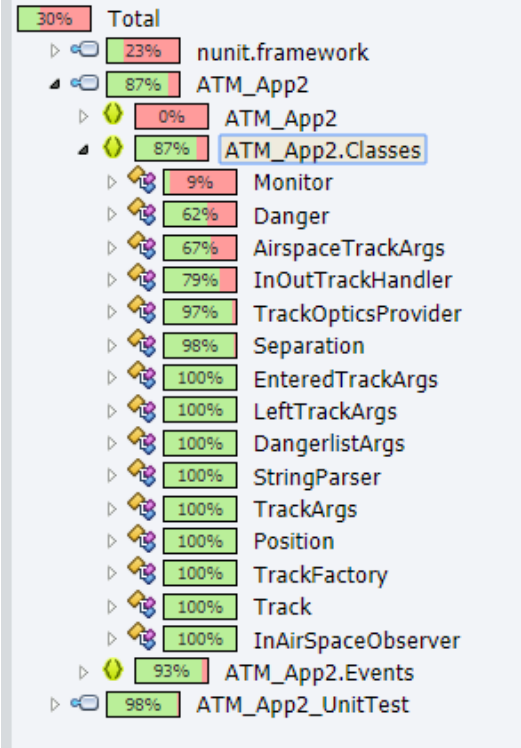
```
Track[] _testTracks = new Track[]
{
    new Track("ATR423", new Position(12000, 12000), 14000, 0, 0, "20151006213456789"),
    new Track("ATR423", new Position(14000, 12000), 14000, 0, 0, "20151006213456790")
};


fakeOpticsProvider_.GetTrackCourse(_testTracks[1], _testTracks[0]).Returns(90);
fakeOpticsProvider_.GetTrackVelocity(_testTracks[1], _testTracks[0]).Returns(2000);


List<Track> newAirspace = new List<Track>();

_uut.AirspaceUpdated += (o, args) => { newAirspace = args.TracksInAirSpace; };


foreach (var track in _testTracks)
{
    fakeTrackFactory_.TrackCreated
        += Raise.EventWith(this, new TrackArgs(track));
}

Assert.That(newAirspace[0] != _testTracks[0]);
Assert.That(newAirspace[0].hori_velocity_, Is.EqualTo(2000));
Assert.That(newAirspace[0].course_, Is.EqualTo(90));
```

Figur 7: InAirSpaceObserver, AirSpaceUpdated unit test med stub og mock klasser

Et eksempel på brug af stubs og mocks ved unit test i unit testen af InAirSpaceObservers håndtering af modtagelse af to tracks med samme tag som set i fig. 7. Da begge er inden for air space skal dette resultere i et AirSpaceUpdated event. For at isolere klassen er klasserne som InAirSpaceObserver er afhængig af blevet fake'et. Således kan TrackCreated eventet rejses med rette test data fra fakeTrackFactory. Desuden kan resultat af kaldet af GetTrackCourse() og GetTrackVelocity() simuleres, da denne test ikke skal afhænge af funktionaliteten af TrackOpticsProvider-klassen, men om InAirSpaceObserver anvender svaret herfra korrekt.

4.2 Coverage test

Fra Unit test projektet på Jenkins er følgende coverage rapport hentet, link til job ses i sektion 7.

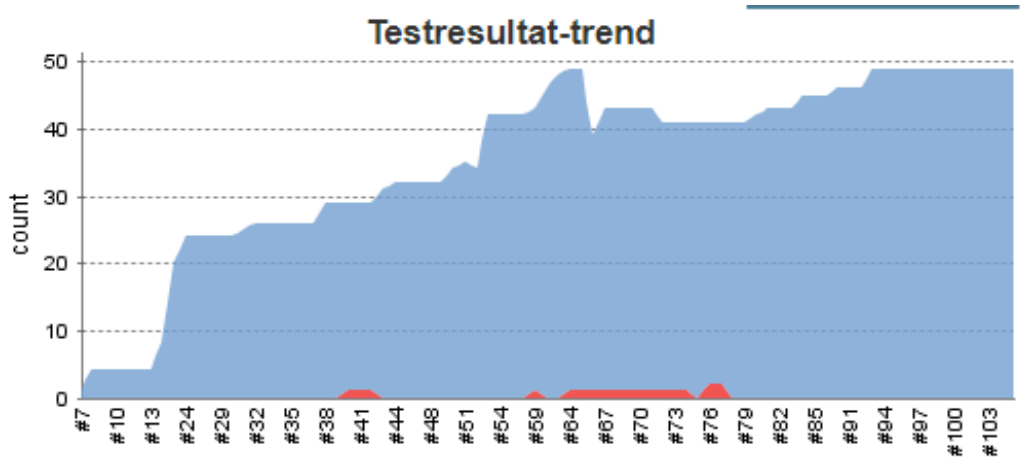


Figur 8: Coverage af unit test for klasserne.

Af figur 8 ses at der er blevet testet på 87% af det implementerede koden i de forskellige klasser. De fleste klasser har opnået en coverage på 100%. Dette viser at det meste kode er blevet testet, mindst én gang. Dette er dog ikke en garanti for der ikke kan være fejl i koden, det reducere dog sandsynligheden for det. Der er ikke testet ret meget i Monitor, da denne klasse ikke er helt implementeret. Danger har en overload implementeret som ikke anvendes i programmet, og derfor er der ikke prioriteret en test af denne overload.

4.3 Refleksion over unit test

Det er valgt at lave unit test på så mange af projektet klasser som muligt. Dette gør at man ved det senere arbejde med integrationstesten, kan sikre at det er linket imellem klasserne der er fejl i, og ikke de enkelte funktioner i klasser.
Der er klasser som Track, Posision og TimeKeeper, som ikke vil blive unit testet. Track og Positions er simple datastrukturer og de blev desuden unit testet ved sidste aflevering. TimeKeeper-klassen er mere kompleks og involvere et realtídskomponent, og den blev derfor stubbet så længe som muligt i processen.



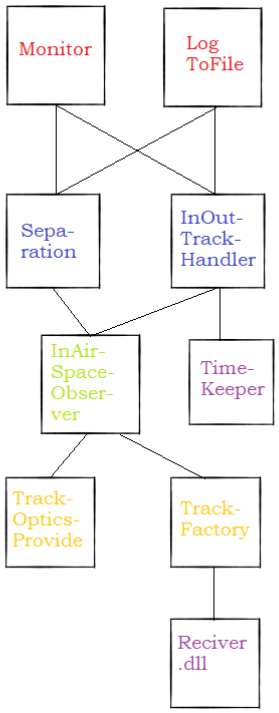
Figur 9: Oversigt at antal af unittest samt succes og fejlede

På figur 9 fremgår det hvordan testresultat trendensen har været for projektet. Det ses at langt de fleste pushes har kunne bygge, men at der er nogle få der har voldt problemer. Dette kan skyldes at der fx er blevet ændret i navngivningen for klassen, men man har glemt det i Interfacet. Hernæst kan man have pushet, uden at have ud kommenteret nogle funktioner eller test der ikke var helt funktionelle. Dette vil altsammen have været med til at give fejl i tests. Som er udtrykt ved de røde felter.

4.4 Dependency Tree

I denne opgave er der blevet lavet et dependency tree, for at give et større overblik over integrationstesten.
Formålet med dette dependency tree, er at lave en bottom-up integration.
Grunden til dette valg er, at det er lettere at undersøge, hvor der er fejl i koden, da man tager små moduler ad gangen og bygger mere på. I løbet af dette er det lettere at opdage hvilke klasser der ligger fejl i, da man ikke kan komme videre til næste niveau, før at alle test er klaret.
En ulempe ved denne metode er, at det ikke er muligt at lave en mulig prototype af programmet, da det skal bygges helt op, før man kan teste det.
Det kan også være kritisk med de øverste moduler, som kontrollerer flow i systemet, bliver testet til sidst og dette kan give nogle fejl indimellem. Dette er dog ikke kritisk for vores system, da det meste af vores flow kommer længere nede i systemet, så derfor bliver det testet før Monitor klassen.
For at danne et overblik over dette dependency tree, er de forskellige lag farvekodet, ses i 10.

1. Dette er stubbende til træet, da disse ender ikke er afhængige af andre klasser.
2. Dette inkluderer kun InAirspace, dog bliver denne testet af forskellige instancer, da den både inkluderer TrackFactory og OpticsProvider.
3. I dette step, er det linket fra InAirSpaceObservator til Seperation og InOutTrackHanlder der testes. For at gøre dette er de tidligere testet klasser TrackFactory og opticsProvider også inkulderet, da det er events triggeret i disse der styre udfaldet for de andre.
4. Monitor klassen er den overordnede klasse i systemet, da denne holder styr på de samlede objekter i nedenstående klasser.
5. Disse klasser minder om stubbe, dog bliver disse først testet til sidst, da TimeKeeper er en timer, som bliver faket indtil de sidste test, for at reducere test tiden på de forskellige klasser. Reciever.dll er rigtige fly, som kommer ind i systemet. Disse bliver også faket indtil de sidste test, dette er også for at reducere tiden for at teste alle klasserne.



Figur 10: Dependency Tree diagram over systemet, Step 1 ses som de orange farvet kasser, step 2 som den grønne, step 3 som de blå, step 4 som de røde og til sidst step 5 som de lilla.

4.5 Integrations Planlægning

step #	Track-Factory	Track-Optics-Provider	InAir-Space-Observer	Separa-tion	InOut-Track	LogTo-File	Monitor	Time-Keeper	Tran-sponder-Receiver
1	**T**	**T**							**S**
2	**X**	**X**	**T**						**S**
3	**S**	**X**	**X**	**T**	**T**			**S**	**S**
4	**X**	**X**	**X**	**X**	**X**	**T**	**T**	**S**	**S**
5	**X**	**X**	**X**	**X**	**X**	**X**	**T**	**X**	**X**

Tabel 1: Integration plan i steps

I tabel 1 er vores plan for en Buttom Up test af vores system, da vi mener denne test strategi egner sig godt til test af en pipeline struktur. Første step er test af de lavest koplede klasser, dette er allerede udført gennem vores udførte unit tests.
Herefter tester vi integrationen mellem TrackFactory- og InAirSpaceObserver-klassen (TrackOpticsProvider inkluderet som kompositionselement). Derefter kobles Separation, InOut og LogToFile på testen, for at teste dette lag af pipelinen. Herefter kan hele systemet testes med Monitor-klassen som driver.
Til sidst koples de egentlige TransponderReceiver og TimeKeeper klasser på og testes, da de indtil nu er blevet stubbet.

4.6 Integrations test

```
33      [Test]
        0 references | Mette, 10 hours ago | 1 author, 1 change
34      public void OnEnteredTrack_TestEvntcalledOnes()
35      {
36          List<Track> newList = new List<Track>();
37          Track inserTrack = new Track("MCJ523", new Position(15000, 13000), 12000, 10, 34, "2016111912343892");
38
39          int _evenCounter = 0;
40
41          _uut.listInUpdated += (o, args) =>
42          {
43              _evenCounter++;
44              newList = args.listEntered;
45          };
46
47          fakeAirSpace_.EnteredTrack += Raise.EventWith(this, new TrackArgs(inserTrack));
48
49          Assert.That(_evenCounter, Is.EqualTo(1));
50
51      }
```

Figur 11: Kode udklip fra unit test af InOutTrackHandlers event OnEnteredTrack.

```
49      [Test]
        0 references | ChrisWOle, 3 hours ago | 1 author, 1 change
50      public void OnEnteredTrack_TestEvntcalledOnes()
51      {
52          List<Track> newList = new List<Track>();
53          Track inserTrack = new Track("MCJ523", new Position(15000, 13000), 12000, 10, 34, "20180304124520412");
54
55          int _evenCounter = 0;
56
57          InOutHandler_.listInUpdated += (o, args) =>
58          {
59              _evenCounter++;
60              newList = args.listEntered;
61          };
62
63          fakeTrackFactory_.TrackCreated += Raise.EventWith(this, new TrackArgs(inserTrack));
64
65          Assert.That(_evenCounter, Is.EqualTo(1));
66          Assert.That(_evenCounter, Is.EqualTo(newList.Count));
67          Assert.That(newList[0] == inserTrack, Is.True);
68      }
```

Figur 12: Kode udklip fra Integrations test step 3 af event OnEnteredTrack .

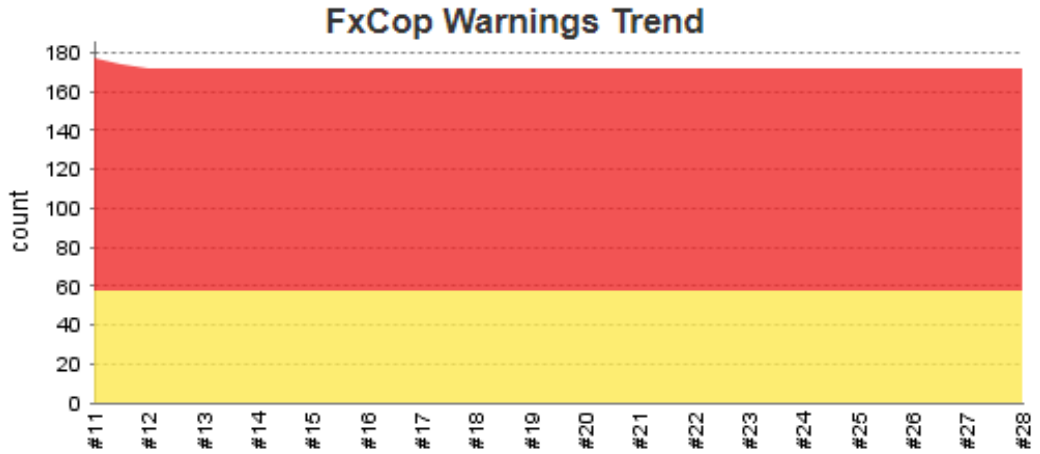
I figur 11 og figur 12 ses den reelle forskel på Unit test (figur 11) og integrations testen (figur 12) for en given funktion. Her ses det at det ikke længere er eventet der skal gribes der kaldes, men eventet der starter hele lavinen. Derved får man testet sin integration helt fra bunden, som jo er målet med en bottom up integrations test. En relation testes ved at der ikke længere anvendes fakes omkring den relation der skal testes, men at de reelle klasse anvendes.

4.7 Refleksion over integrationstest

Inden integrationstesten blev påbegyndt, blev der lavet Unit test til så mange af klasserne som muligt. Dette gjorde at man kunne gå direkte til step 2 af integrationstestene. Integrationstestene er på sin vis ligesom unit testene. men i stedet for at bruge stube, fakes og muks, bruges de realle klasser. Derfor er det valgt at lave en bottom up integration, hvor det første step kan skiftes ud med en unit test. Det blev valgt at tage en integrationstest af gangen da det herved ville blive klart hvori eventuelle fejlen lå.
I forhold til til den lagte integrationsplan, så nåede vi til Step 3. Der var problemer med håndtering af Leaving-Track event. Dette kan både være i event handleren, men problemet kan også ligge hos senderen. Grundet tidsrammen er buggen ikke blevet indentificeret, men det begrænser søgningen til meget specifikke sektioner af kilde koden.

5 Statisk analyse

Her ses på hvor godt koden er, med henblik på navngivning, vedligehold, præstation osv.



Figur 13: Oversigt af advarsler fra statisk analyse.

Af figur 13 Kan det ses at projektet indeholder rigtig mange advarsler, for at se hvad advarslerne er for, er der lavet en oversigt efter kategori.

FxCop Warnings - Namespace ATM_App2/ATM_App2/Classes

Summary

Total	High Priority	Normal Priority	Low Priority
129	79	50	0

Details

Files | Categories | Warnings | Origin | Details | High | Normal

Category	Total	Distribution
Microsoft.Design	48	
Microsoft.Globalization	9	
Microsoft.Maintainability	1	
Microsoft.Naming	53	
Microsoft.Performance	6	
Microsoft.Usage	12	
Total	129	

Figur 14: Advarsler inddelt efter kategori i de implementerede klasser.

Efter inddellingen efter kategori som ses på figur 14 Kan det ses at de fleste advarsler opstår pga navngivning, dette var forventet, da der ofte anvendes navne med meget lille variation, og at syntaks for metoder og attributter ikke overholdes. Næst er pga design, hvilket ikke er alt for heldigt. Hvori disse fejl typisk ligger har ikke været i fokus for gruppen, da testene var problematiske.

6 Konklusion

Kom man i mål med af få testet alt og nå en Coverage Rapport på 100%? Hertil må svaret være nej. Det er stadig klasser der mangler at blive implementeret(fx. Monitor), funktioner og klasser der mangler at blive unit testet (fx. TimeKeeper) og flere steps af integrationstesten mangler. Det er under integrationstestet blevet klart at visse funktioner ikke virkede som de gav udtryk for under unit testene. Der har dog ikke været tid til at rette disse, men det giver et rigtig godt billede af, hvor vigtige disse test er. For selv om funktionerne virker som unit test, ses det i dette tilfælde at der er en fejl, der først kommer til udtryk når alle sammenhængende skal spille sammen.

7 Formelle

Gruppen Jenkins findes under:
SWT_Grp21_atm2:
http://ci3.ase.au.dk:8080/job/SWT_Grp21_atm2/
SWT_Grp21_atm2_Integration:
http://ci3.ase.au.dk:8080/job/SWT_Grp21_atm2_Integration/
SWT_Grp21_atm2_StaticAnalysis:
http://ci3.ase.au.dk:8080/job/SWT_Grp21_atm2_StaticAnalysis/
Gruppens Git:
https://github.com/chriswole/SWT_gr21_StaticA1.git