SWT Hand In 3
Gruppe 20

Aarhus University School of Engineering

Air Traffic Monitoring

25-04-2018

Studienummer	Navn	Email-adresse
201607589	Jakob Bonde Nielsen	au567214@post.au.dk
201607110	Kasper Juul Hermansen	au557919@post.au.dk
201605114	Stefanie Ruaya Nielson	au554093@post.au.dk

Link til Git: `https://github.com/TeamTyve/ATM`

Link til Jenkins - UnitTest: `http://ci3.ase.au.dk:8080/job/TeamTyveATMUnitTest/`

Link til Jenkins - integrationstests: `http://ci3.ase.au.dk:8080/job/TeamTyveATMIntegrationTest/`

Link til Jenkins - Coverage: `http://ci3.ase.au.dk:8080/job/TeamTyveATMCoverage/`

Indholdsfortegnelse

Design 1

1.1 Formål

Formålet med denne opgave er at lave et sytem der kan monitorere og præsentere flytraffik over et givent område. Der vil også blive kigget på om flyene i området flyver for tæt. Informationer om flight-tracks vil blive indhentet fra en TransponderReceiver, givet i strings. Dernæst objektifiseres disse strings og objekterne vil blive bearbejdet ved at tjekke om de flyver i et givent område samt om de flyver for tæt. Alle disse informationer vil blive præsenteret ved hjælp af et konsol-output. Hele systemet vil blive testet ved hjælp af både unit tests samt integrationstests.

1.2 Design

Link til Github: `https://github.com/TeamTyve/ATM`

Der er i opgaven givet en .dll fil der benyttes til at indhente flight-tracks og præsentere disse i strings. Der implementeres derfor et Track Observation System (herefter også benævnt som TOS), som benyttes til at objektifisere de indhentede strings. Formålet med at få objektifiseret disse strings, er at man får gjort de indhentede strings meget mere overskuelige når de omdannes til objekter, samt at det bliver muligt at arbejde på objekterne.
De indhentede data, og dermed objekterne er fly, som består af henholdsvis et tag (navnet på flyet), x- og y-koordianter, højde i luften, hastighed, retning i grader samt et timestamp.

Når flyene er blevet objektifiseret vil der blive checket for nogle forskellige ting. Først vil der blive checket for, om et fly er i et givent område - kaldet AirSpace. Hvis flyet er i AirSpace, vil flyets informationer blive udskrevet til konsollen. Dernæst vil der blive checket på om de fly, der er i airspace, flyver for tæt på hinanden. Hvis dette skulle være tilfældet, vil der udskrives en advarsel på konsollen, omkring hvilke fly der flyver for tæt på hinanden, samt hvad tid advarslen fremkom.

Klassediagram - Vedhæftet som bilag bagerst i dokumentet
På klassediagrammet ses hvordan systemet er opbygget. Det ses hvordan klassen TrackObservationSystem (TOS) er "hovedmodulet", der er forbundet til de andre klasser. Klassen "TrackingSystem"benyttes til at modtage en TransponderReceiver fra .dll filen. Denne TransponderReceiver overføres herefter til TOS. TOS er også koblet til TrackRepository der persisterer Tracks. TrackRepository benyttes til at udregne ting som hastighed, retning, distance etc. Det er denne klasse som i sidste ende objektifiserer de strings der bliver indhendtet fra TrackingSystem og over i TOS.
Klassen "AirSpace"benyttes til at tjekke om et fly er inde i AirSpace.
Derudover indeholder systemet også klasser som Seperation, SeperationAlertRepository samt SeperationAlert. Disse klasser benyttes til at tjekke om de fly, der er inde i AirSpace, flyver for tæt på hinanden.
Klasserne Output, LogHelper, ConsoleLogger og EventLogger er alle klasser der er med til at få printet dataen til konsollen.
Grunden til at der findes både ConsoleLogger samt EventLogger, er at EventLogger persisterer de events, der kommer, ned i en fil og ConsoleLoggeren skriver dataen ud på konsollen.

Sekvensdiagram - Vedhæftet som bilag bagerst i dokumentet
På sekvensdiagrammet ses det hvordan klasserne interagerer med hinanden. Det ses her hvordan TrackingSystem starter med at modtage en ITransponderReceiver fra klassen TransponderrReceiver der ligger i .dll filen. Herefter oprettes et nyt TrackObservationSystem der medtager denne ITransponderReceiver som parameter.Nu indhendtes der dermed data om indkommende fly. I sekvensdiagrammet gåes der herefter ind i en "alt". Så snart der er noget TransponderData klar vil denne data bearbejdes - Dataen, som indhendtes i strings, vil blive objektifiseret og der vil blive checket om de er i Airspace, samt om de flyver for tæt på andre fly inde i AirSpace. Disse data vil blive udskrevet hvis ConsoleOut er lig med true vil flyene i AirSpace blive udskrevet til konsollen.
Der er mulighed for at der sker et SeperationEvent. Hvis flyene er for tæt på hinanden ved check via funktionen "CheckSeperation()"vil der blive raiset et SeperationEvent. Dette vil resultere i, som der også ses på diagrammet, at en warning udskrives i konsollen.

Test 2

2.1 Unit tests

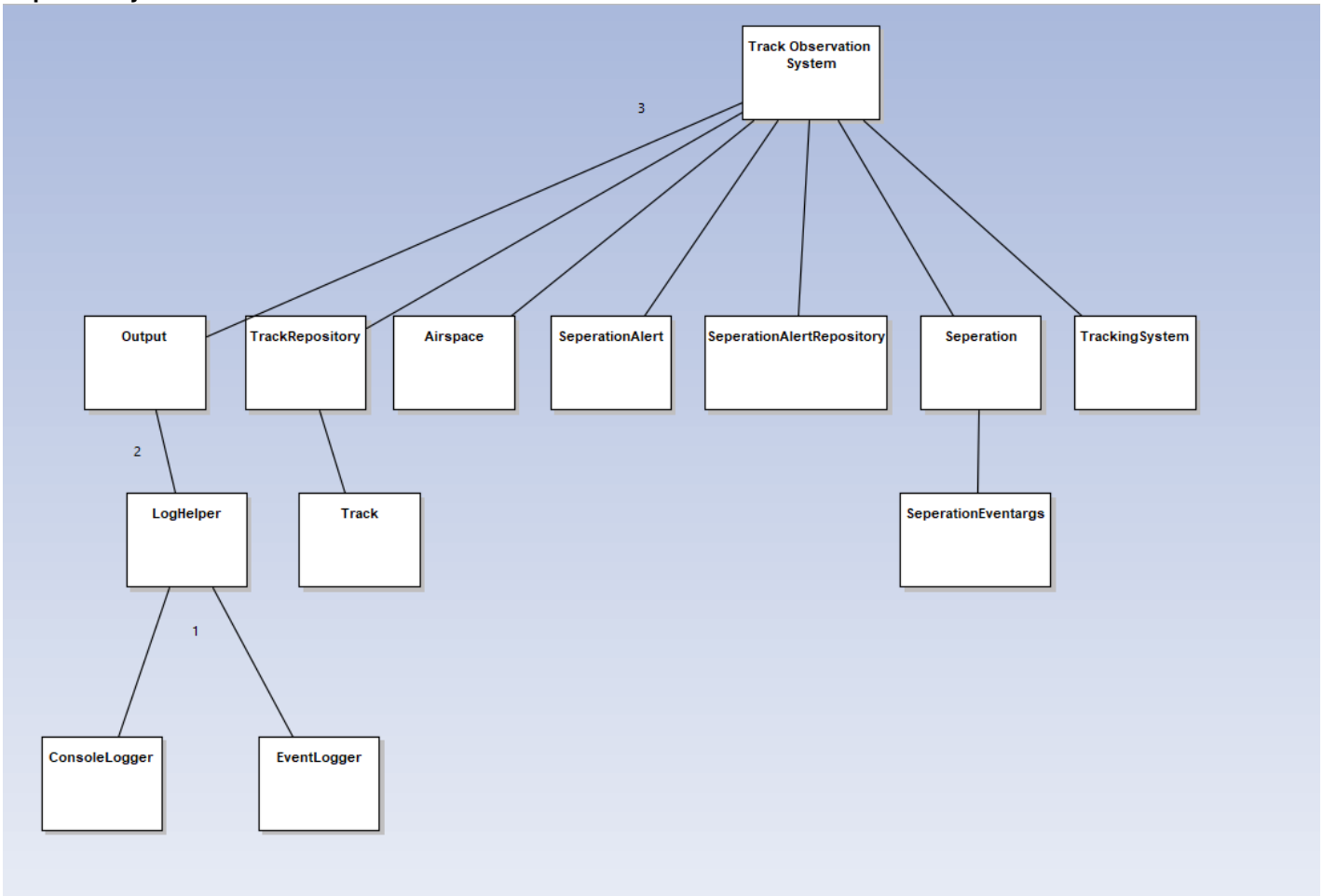
Link til Jenkins - Unit tests: `http://ci3.ase.au.dk:8080/job/TeamTyveATMUnitTest/`

Der er valgt ikke at teste ConsoleLogger.cs, da denne skriver direkte til konsollen, og det er ikke muligt at erstatte statiske metoder. Det vil derfor ikke være muligt at teste om Console.WriteLine() udskriver det korrekte.
Ydermere er der få andre klasser der ikke er blevet unit testet. Dette skyldes at disse klasser for eksempel kun indeholder entiteter, eller ikke har nogen logik i sig. Derfor vil det ikke give nogen videre værdi at teste disse.

2.2 Integrationstests

Link til Jenkins - Integrationstests: `http://ci3.ase.au.dk:8080/job/TeamTyveATMIntegrationTest/`

Dependency Tree



Ovenstående ses dependency-tree for systemet. Der er i alt tre steps, hvor der er valgt at teste det nederste lag først, og derfra bevæge sig op. Altså er bottom up valgt som integrationsstrategi

Denne strategi er valgt grundet at der derved er færre komponenter der skal stubbes, samt at det har virket intuitivt at bruge denne strategi.
Generelt set går bottom up testing ud på at starte fra bunden i det tilhørende dependency tree. De nederste klasser, og derved de klasser der afhænger af mindst vil blive testet først. Hvorfra der hierakisk vil blive testet op igennem

træet, indtil alle forbindelser imellem klasserne er blevet tests. Der vil altså blive testet fra submodulerne til hovedmodul(erne). En af fordelene ved bottom up strategien er at det kan eliminere behovet for anvendelse af stubbe i og med at der ikke er nogle højerestående dependencies at tage højde for. Dog er en af ulemperne ved strategien, at man får testet den overordnet funktionalitet sent i processen, og der derfor er risiko for at man skal lave noget af systemet om, senere end nødvendigt.

Integrationsplan

Step	Step 1	Step 2	Step 3
ConsoleLogger	S	S	S
EventLogger	X	X	X
LogHelper	D	X	X
Track		X	X
SeperationEventArgs		X	X
Output		D	X
TrackRepository		D	X
AirSpace			X
SeperationAlert			X
SeperationAlertRepository			X
Separation		D	X
TrackingSystem			X
TrackObservationSystem			D

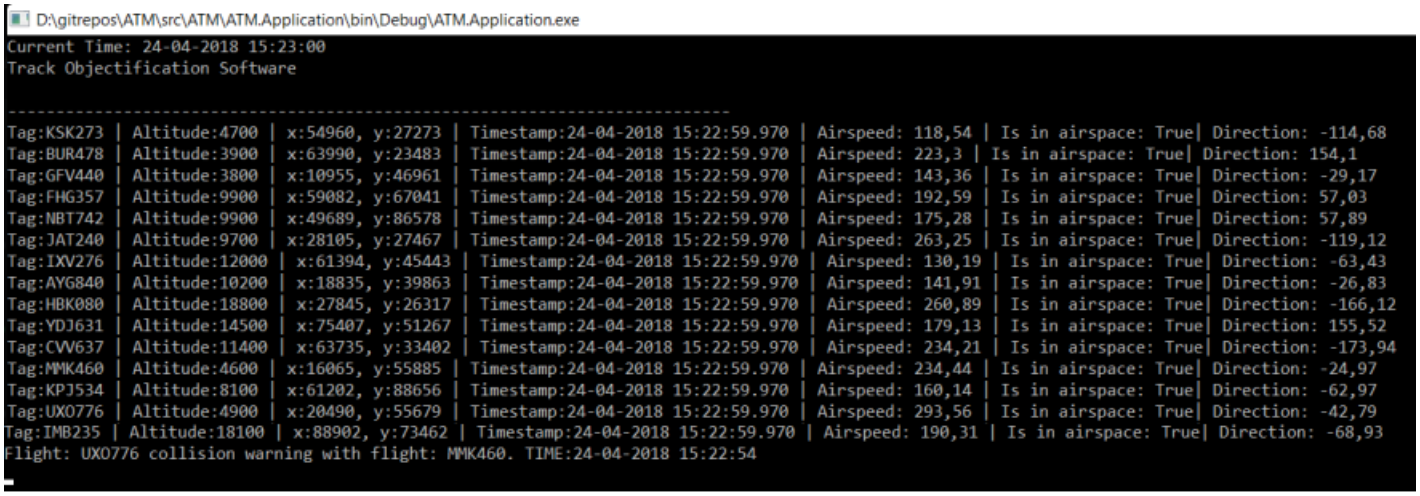
Integrationsplanen viser hvordan de forskellige klasser er inkluderet eller stubbet under de forskellige steps i integrationstesten. Integrationsplanen er lavet med udgangspunkt i dependency træet.

Resultater 3

3.1 Resultater

Link til Jenkins - Coverage: `http://ci3.ase.au.dk:8080/job/TeamTyveATMCoverage/`
Som det ses på nedenstående billede virker systemet som forventet. De tracks der ligger inde i AirSpace bliver udskrevet på konsollen med alle de ønskede informationer.

De warnings der kommer, når to fly flyver for tæt, bliver udskrevet nederst i konsollen.



3.2 Diskussion

Til denne hand in var der ikke givet nogle specifikke retningslinjer for hvordan opgaven skulle løses. Dette betød også, at der var stor frihed til hvordan systemet skulle designes. Der blev valgt et design, der gjorde at Track Observation System (TOS) - klassen var meget afhængig af de andre klasser, som det også fremgår af det udarbejdede dependency tree. Ud fra dette, kan det også ses at det er et ret "horisontalt"system, i og med at TOS indhenter strings og objektifiserer disse. Når dette er gjort vil der arbejdes direkte på objekterne igennem de andre klasser, i stedet for at bruge en form for pipes and filters arkitektur, hvor der arbejdes i en vandret orden, og objekterne bearbejdes hen ad vejen i stedet for det hele på én gang.

Dette blev i starten valgt, grundet at flere ting arbejdede på én gang. Der blev blandt andet tjekket på om flyet var i AirSpace, men også om flyet fløj for tæt på andre fly. Set i bakspejlet har det formegentlig været mere testbart at benytte en pipes and filters arkitektur, for at undgå at TOS fik så mange dependencies.

Gruppen fik dog løst det på anden måde og fik et tilfredsstillende resultat.

Der blev benyttet en CI server på Jenkins til at undersøge hvornår et build kunne merges til masteren. Dette blev dog ikke benyttet af gruppen, da det meste af systemet blev lavet da medlemmerne sad sammen fysisk på skolen.Det ville formegentlig have givet mere værdi i et større team, hvor der arbejdes mere uafhængigt af hinanden.

3.3 Konklusion

Alt i alt virker systemet som forventet på trods af at de måske ikke har været den optimale løsning der er blevet brugt. Set i bakspejlet kunne dette dog være løst ved at have brugt mere tid på design og arkitektur i starten af forløbet. Gruppen har erfaret at det er vigtigt inden at der startes med implementering, at der skal laves en god arkitektur, som også har fokus på at gøre programmet testbart. Gruppen har også erfaret at arkitekturen dermed har stor indvirkning på hvor let det er at lave integrationstest. Med en struktur hvor en klasse har for mange afhængigheder, som f.eks. i dette tilfælde med Track Observation System klassen, er det svært at udfører en overskuelig intergrationstest.