SWT- Gruppe 8 -AirTrafficMonitor

Nicholas Ladefoged 201500609
Max Barly Jørgensen 201401694

April 2018

Link til github: https://github.com/maxdenstore/SWTHandin3_Gr8

Links til Jenkins:

1. unit tests og coverage http://ci3.ase.au.dk:8080/job/Air%20Traffic%20Monitoring%20-%20Group%208%20-%20Unit/

2. integrationstest :http://ci3.ase.au.dk:8080/job/Air%20Traffic%20Monitoring%20-%20Group%208%20-%20Integration/

3. Code metrics : http://ci3.ase.au.dk:8080/job/Air%20Traffic%20Monitoring%20-%20Group%208%20-%20CodeMetrics/

Contents

1 Introduktion

1.1 Introduktion til opgaven

Som forskrevet i opgaven vil vi udføre unit test, samt integrations test af AirTraficMonitor systemet. Der bruges på en DLL fil (TransponderReceiver) som er udgangspunktet for hele opgaven. Vi vil forklarer vores design og grundliggende implementeringsvalg udfra UML diagrammer som støttes med en Intergrationstest, som er baseret på dependencies identificeret i forbindelse med designet. Vi har valgt at udfører en del af arbejet over LiveShare, hvilket er grundlaget for at en user (Nicho1991) på github som udgangspunkt har flere commits, da liveshare gør at kun en master med projektet kørende på sin PC kan committe og pushe. Vi har i denne opgave forsøgt efter bedste evne at lave commits ofte med kommentar på, som kan ses på vores github.

2 Design

2.1 Aktivitets Diagram

Vi har valgt at dokumentere vores design ved hjælp af UML. Hertil har vi lavet et aktivitets diagram som har til formål at give et overblik over hvordan systemet fungerer, samt et tilhørende klasse diagram som beskriver de klasser der indgår i systemet. Diagrammet giver en overskuelighed over hvor forskellige klasser afhænger af hinanden, da diagrammet er delt op i swimlanes, der repræsentere hver enkelt klasse.

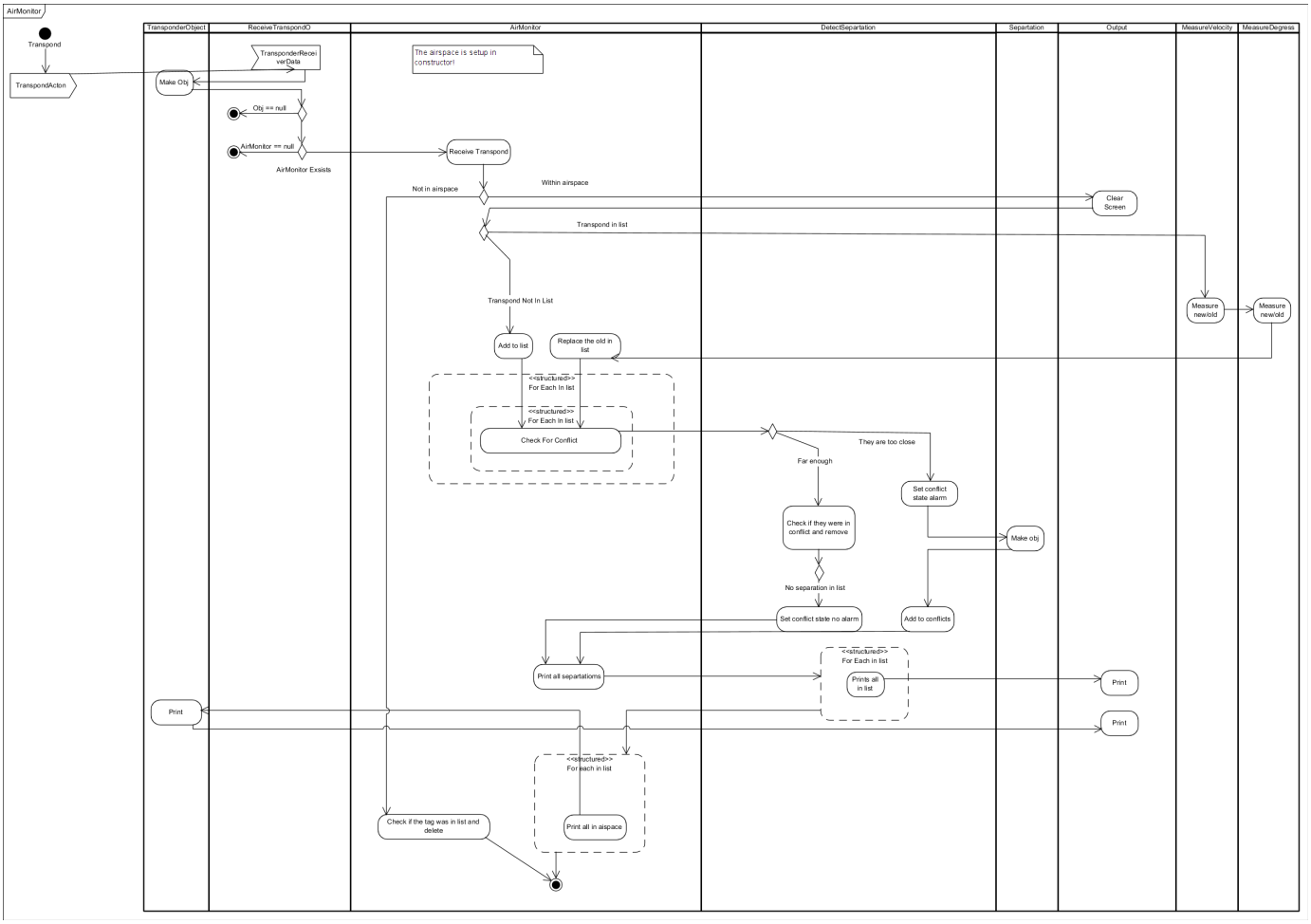

Figure 1: AktivitetsDiagram

Ved hjælp af figur 1 Får vi hermed et overblik over hvordan vores integrations test skal udføres, og i hvilke sammenhænge klasser bruger hinandens interfaces.

3 UnitTest

3.1 Unit testing

Unit test bestræber sig på at white-box teste klasser, således funktionaliteten i en klasse bliver testest i forhold til de krav der er stillet klassens funktionalitet.

Vi har opbygget tests for hver enkelt klasse (med en undtagelse) der skal forsøge at efterstræbe dette krav. målet for enhver klasse er at teste 100 procent af hvad en klasse indeholde, samt overholde en underliggende bva analyse. Det klart at virkeligheden ikke altid tillader 100 procent coverage af alt kode i form af vores unit tests. Vi har udført en analyse af præcist hvor meget vi har testest på vores jenkins server, og figur **??** viser hvordan virkeligheden ser ud her.

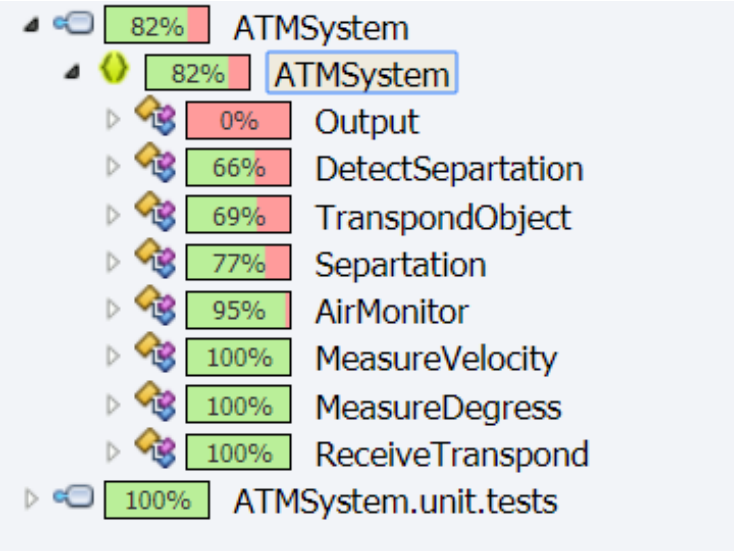

Figure 2: Code coverage

Det første der springer i øjene er 0 procent på output. Dette skyldes at output blot er en klasse der sender en streng videre afsted til consolen! Dette har vi af den grund vurderet, ikke har brug for test. De andre klasser der ikke har 100 procent skyldes private metoder, der skal testes i en sammenhæng, der bliver bedre testet i vores integrations test, som vi har vurderet som ok her.

4 IntegrationsTest

4.1 IT test

Vores integrationstest er udført baseret på vores dependency diagram, som kan ses på figur 3. Vores dependencies kan illusterer hvordan integrations testen kan udføres ved hjælp af afhængigheder blandt de forskellige klasser. Vi har benyttet top-down approach da, vi syntes dette gav mest mening ifht. vores dependencies for overskuelighedens skyld. Derfor er diagrammer vist således det tydeligt fremgår at øverste led agerer som driver i samtligt tilfælde.

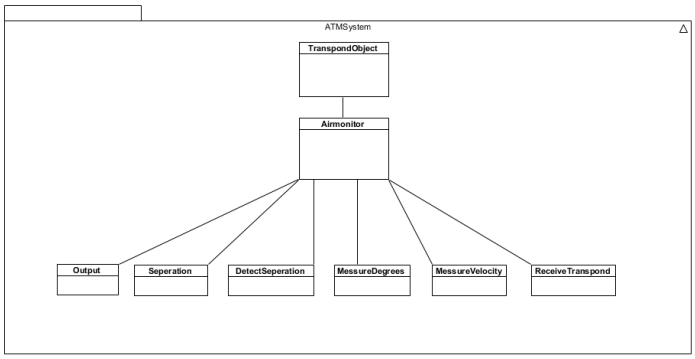

Figure 3: IntegrationsTest Illutstration

Da Airmonitor er hovedelementet i vores system, men er afhængig af input fra transpondObject klassen, benytter vi derfor transpondObject som driver, hvor vi tester gennem Airmonitor ned til hver enkelte klasse.

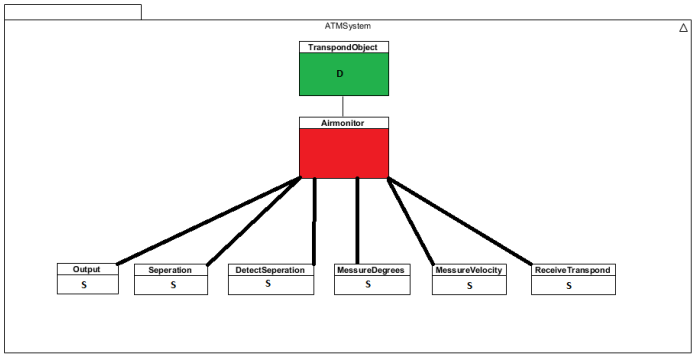

Figure 4: IntegrationsTest Illutstration

Figur 4 Testen er udført hvor transpondObject agerer som driver og Airmonitor er unit under test. Vi benytter de resterende klasser som stubs for at sikre Airmonitor virker som tiltænkt. Testen er udført ved brug af NSubstitude, hvor vi faker alle klasser der agerer som stubs. Det gør sig gældende at hver "stub vej" er testet individuelt for en test. Testene er udført ud fra vores aktivites diagram figur 1

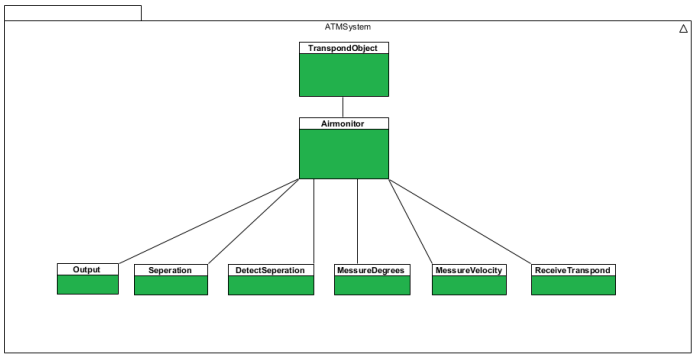

Figure 5: IntegrationsTest Illutstration

Figur 5 Efter testen er udført, kan vi tage udganspunkt i at de resterende klasser der agerede som stub er unit testet, derfor er integrationstesten færdig afviklet.

TranspondObject	Airmonitor	ReceiveTranspond	MessureDegrees	MessureVelocity	DetectSeperation	Seperation	Output
D	X		X	S	S		S
D	X		X	S	S		S
D	X		S	X	S		S
D	X		S	X	S		S
D	X		S	S	X	X	S
D	X		S	S	X	X	S
D	X		S	S	S		X
D	X		S	X	S		S
D	X		S	S	X	X	S
D	X		S	S	X	X	S
D	X		S	S	S		X
D	X		S	S	S		X
D	S	X					S
D	S	X					S

D = Driver

S = Stubbed/Mocked

X = Included

Figure 6: IntegrationsTest Tabel

Figur 6 viser oversigten over alle integrations test som er udført i denne opgave. Vi har som udgangspunkt udført testen baseret på både fejlscenarier, hvor vi forventer forkert output, samt tiltænkte scenarier, hvor vi tester ved brug af korrekte inputs.
Hermed får vi altså testest alle de ahhængigheder figur 2 viser, samt De forskellige scenarier som kan opstå, Heraf et eksempel hvor et fly er i konflikt med et andet, og så senere ikke længere er! Når vi har udfyldt vores tabel, har vi kigget kode og 2 igennem for at afsløre hvor vores interfaces ligger der skal testes i forbindelse med denne integrationstest.

5 Konklusion

5.1 Konklusion

Vi har i første omgang måtte genafleverer opgaven, dette har faktisk hjulpet os enormt meget til en bedre forståelse af hele opgaven. Vi har fuldstændigt ændret vores design, og startet forfra, stortset fra bunden. Vores nye design er en markant forbedring, hvor vi har haft SOLID principper, samt TDD i tankerne hele vejen gennem designfasen. Vi har omskrevet kode, samt struktureret koden, således opgaven virker meget mere overskuelig. Vores integrations tests, har også fået en væsentlig forbedring, da vi nu har opsat test tabel, samt et solid design hvor vi har overvejet grundigt hvordan vi skulle teste inden selve koden blev skrevet. Vores Unit Test og coverage er også markant forbedret, da vi nu har lettere ved at holde styr på de enkelte unit tests, hvor vi er gået fra at have en meget lang unit test, til en masse små opdelte test cases.