ATM GENAFLEVERING

I4SWT Gruppe 24

Navn	Studienummer	Email
Morten Bech	201604785	au568436@uni.au.dk
Alparslan Esen	201405877	au522394@uni.au.dk
Tri Nguyen	201610974	au564065@uni.au.dk
Berat Kaya	201610971	au572020@uni.au.dk

Github:

https://github.com/Macradon/AirTrafficMonitoring

Jenkins:

http://ci3.ase.au.dk:8080/job/ATM%20I4SWT24/

APRIL 25, 2019

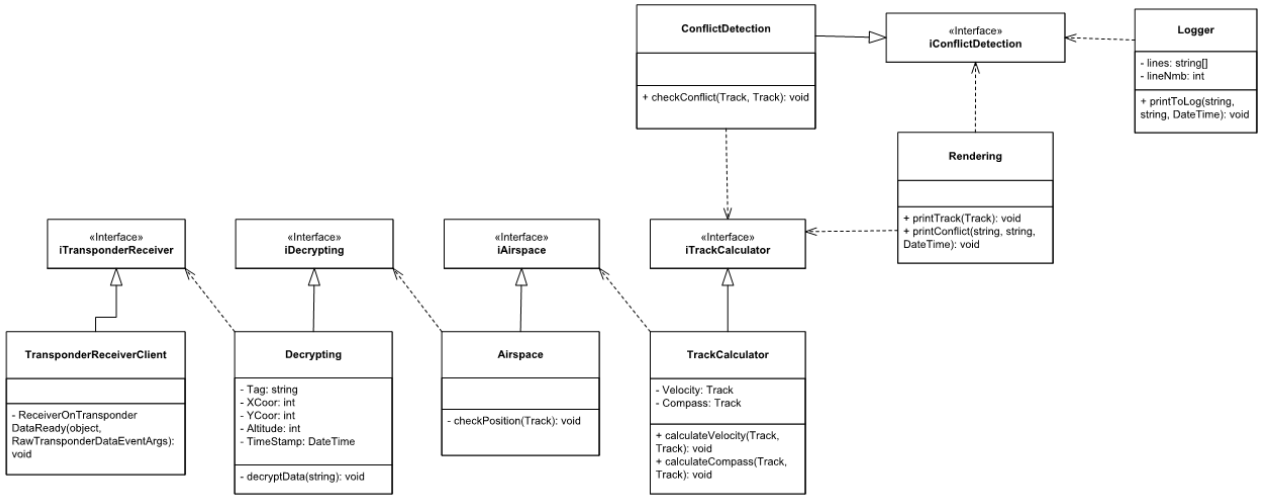
Table of Contents

Introduktion

Denne rapport vil give et indblik i vores opgave, overvejelser, valg, og arbejdsform. Dette vil hjælpe med at forstå udførelsen af opgaven, og vise de overordnede tanker der har været gruppen i hænde i løbet af opgavens udførsel. Da dette er en genaflevering vil der blive nævnt lidt til hvad vi har gjort før, og hvad vi har gjort i løbet af genafleveringsperioden.

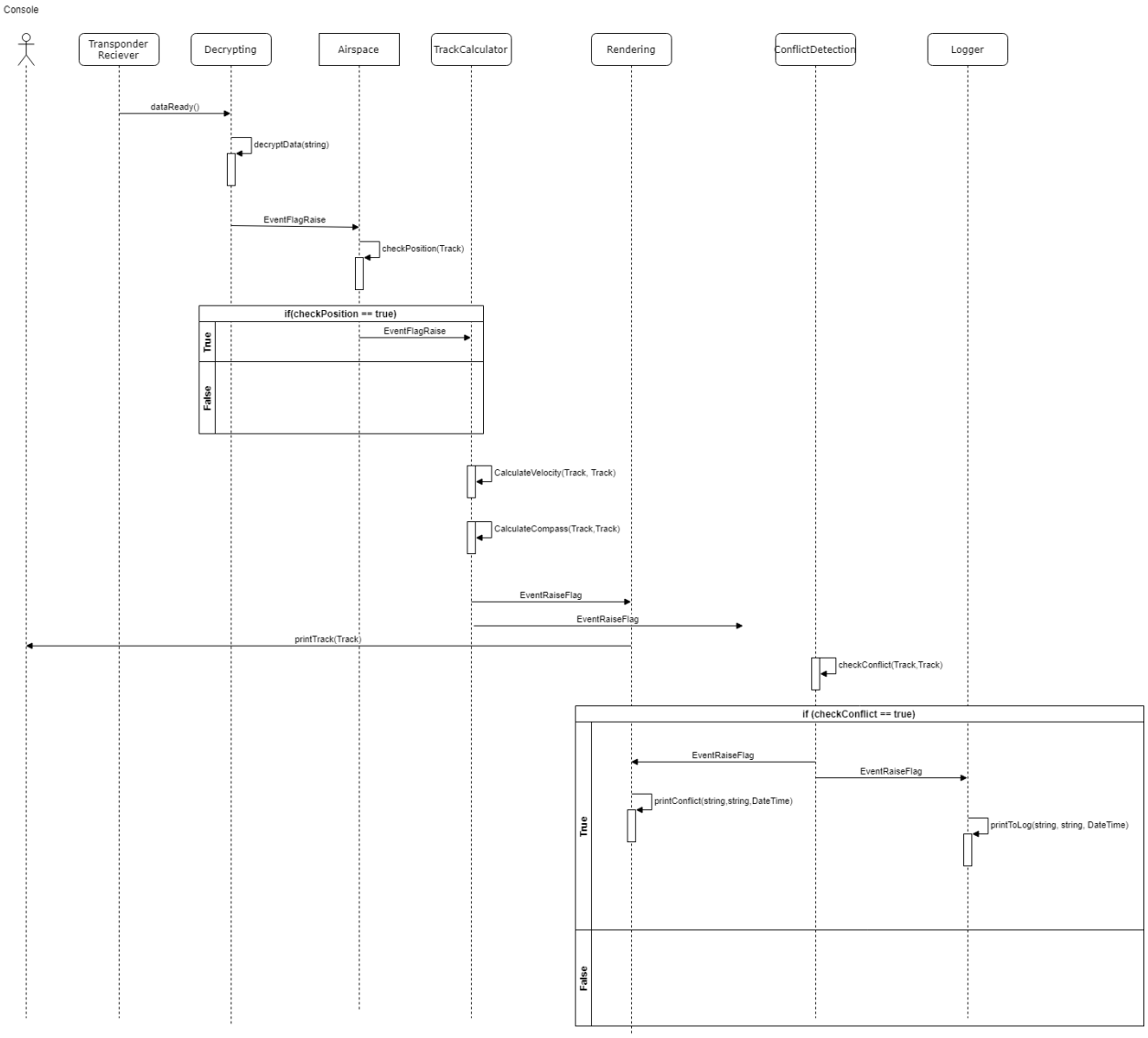
Design

Til selve designet af koderne er der implementeret et klassediagram og et sekvensdiagram. Vi besluttede os i modsætning til tidligere iterations design(Monolith) der havde al for høj kobling, fordi alle klasserne skulle have kendskab til hinanden. For at undgå dette, har vi kodet, således at der ikke er en "central" klasse der arbejdede ud fra de andre, men mere som en kæde af klasser(Observer pattern). Dette kan ses på klassediagrammet for neden.



Figur 1 Klassediagram over ATM

Nu, hvor vi har en god ide om hvilke klasser, attributter og metoder samt klassernes forhold til hinanden begyndte vi at tænke over, hvordan funktionaliteten skulle køre fremad. Efter nogle diskussioner besluttede vi os, at alle klasserne ikke behøver for at kende hinanden, da det resultere i høj kobling. Det problem har vi løst, som vi vil beskrive yderligere. Nedenfor ses et sekvensdiagram over vores program.



Figur 2 SekvensDiagram over ATM

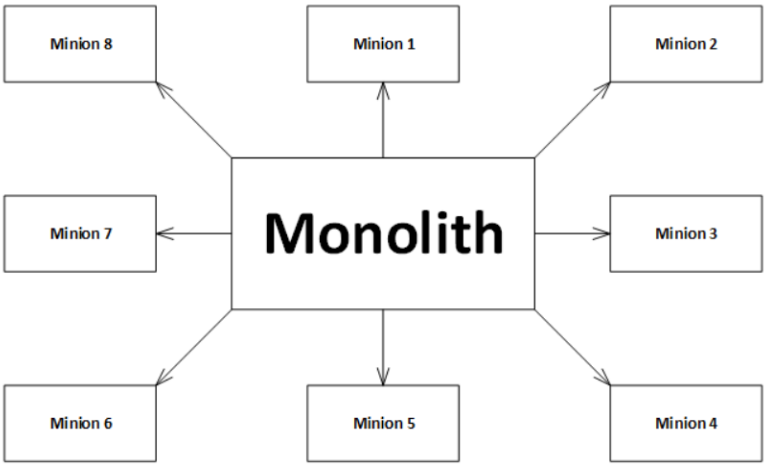
Vores kode blev designet efter observer pattern, hvor vi har vores publisher klasse, som notify'er til "observerne", altså de klasser der skal reagere på eventet, når der sker noget(event). Det hjælper til at Data flowet kører i sin retning, hvor dependency kører i den anden retning. Det kan ses på klassediagrammet Figur 1.
Vi endte med at lave koden i forhold til et observer pattern, dette er fordi vi valgte at bruge et mere event-driven system til implementeringen af vores klasser når det er de skulle arve fra andre klasser og ikke nødvendigvis have kendskab til de øvrige klasser.

Implementering

Applikationen er implementeret i Visual Studio, hvor vi har brugt github til versionskontrollering og kontinuerlig integration. Vi har brugt meget peer-programming i starten hvorefter vi havde etableret hvilke klasser og metoder der skulle laves delte det op og arbejdede parallelt. Derefter hen ad slutningen af implementeringen af applikationen gik tilbage til peer-programming, dette er grundet den måde vi har været vant til at arbejde på tidligere samt ønsket om at diskutere løsninger til implementationen.

Fejlhåndtering

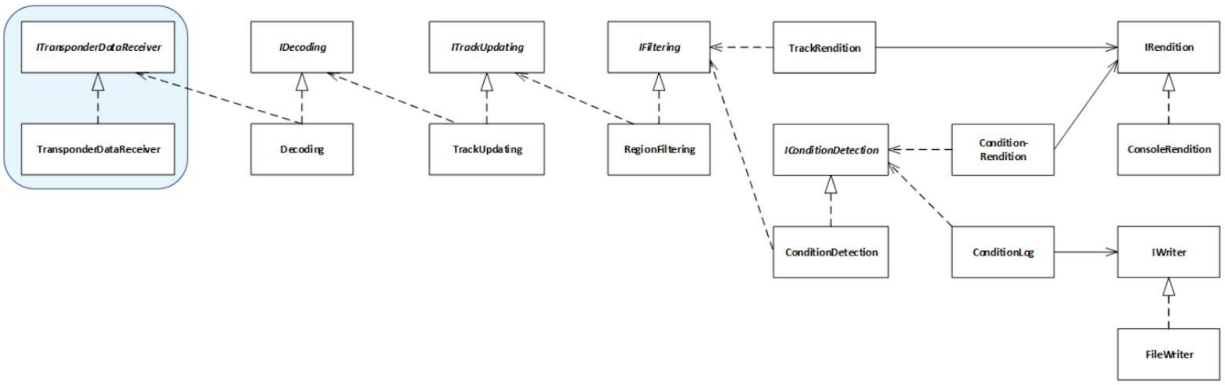
Vi har brugt ufattelig meget tid på at formater vores kode til genafleveringen, således at vi overholder de krav der blev stillet. Som sagt har vi ændret stort set alle steder i koden, hvor vi tidligere formede vores program efter Monolith. Se Figur 3. Som det også kan ses på figuren, så får vi alt for høj kobling, da Monolith klassen skal have kendskab til alle sine minions.



Figur 3 Monolith-Design Pattern fra undervisningens slides.

Løsning

Til genafleveringen har vi designet vores kode efter observer pattern, hvor, hver klasse vha. et eventflag notifier til de andre klasser, at der sker et event nu og derfra kalder den så på den metode der skal udføres. Som det kan ses,bliver relationen mellem klasserne mere overkommelig, da hver klasse ikke skal have kendskab til hinanden. Vha. denne design form undgår vi høj kobling.



Figur 4 Lav-Koblin design forslag fra undervisningens slides.

Arbejdsfordeling

Arbejdsfordelingen under genafleveringen har været meget mere struktureret, vi har delt implementeringen op mellem os og arbejdet parallelt. De flere iterationer som koden har gennemgået har været med til at gruppen har gået frem og tilbage mellem kode og design og vi har generelt sørget for at der altid har været nogen i gang med at kode selve funktionaliteten.

Test

Til vores tests, har vi lavet en række test cases, der skal testes om inputtet og outputtet til klasserne er det der forventes. Det der menes er at, f.eks. hvis vi tager "Rendering" klassen, så tester vi om denne klasse får noget Input, fra en anden klasse og om outputtet af "Rendering" klassen er som det vi forventer. I den første aflevering, lavede vi test-cases for vores funktioner, hvilket vi fandt ud af ikke var det der skulle testes. Derfor har vi i denne genaflevering opstillet de nye test-cases, men da vi ikke kunne formå at få programmet til at kører som vi vil, har vi ikke kunne teste.

Ideen med testene når det var vi havde nået så langt var at kigge på hvad vores klasser fik som input og se hvad de skulle smide ud som output og lave vores tests fokuseret på dette. Vi ville teste metoderne ved at bruge blackbox testen frem for at bare sammenligne med nogle variable vi havde erklæret under act delen af Unit Testen.

Mulig Unit Test

Da koden er blevet opdateret er der ikke blevet lavet unit tests. Ifølge klassediagrammet skal de fleste funktioner kunne videresende data til en ny klasse ved hjælp af events, hvor der skal hæves et flag, som så bliver detektet og handlet af den associerede klasse. Vi har sørget for at holde lav kobling på klasserne, så dem der er længst til venstre på klassediagrammet har en lavere dependency end dem helt til højre. Dvs. at de klasser med høj dependency kræver en større vej at teste, da det hele fungerer som en blackbox, og vi kun tjekker hvad vi sender ind som input og hvad der kommer ud igen som output. Hvis vi for eksempel vil lave test på calculateVelocity-funktionen, så vil vi lave en fake TransponderReceiver-datastreng, der vil køre igennem en masse funktioner, som vi "ikke kan se". Og derudfra vil vi tjekke om vi modtager det forventede output fra blackboxen.

Konklusion

I denne opgave skulle vi skrive et ATM program, samt lave unit tests for de klasser som modtog noget input og derefter tests for om outputtet var det forventede. Der blev skitseret et klassediagram og sekvensdiagram, som skal vise, hvordan programmet er opbygget og vise det sekventielle forløb. Vi har som sagt IKKE kunne færdiggøre opgaven til tiden, hvilket vi beklager rigtig meget.

Under den første aflevering var vores implementering meget vægtet til en main klasse, det ville vi som gruppe undgå til genafleveringen. Dette medførte vores nuværende implementering efter et par revideringer ser sådan ud som den gør. Vi som gruppe fik problemer med events, som skulle bruges i forhold til implementeringen af resten af funktionaliteten.

Vores Jenkins builder automatisk når det er vi commiter noget til vores Git, men siden vi ikke har været i stand til at få skrevet vores ønskede tests ind som kode og rettet dem vi allerede havde til. Så kan Jenkins ikke gennemføre dets build og fejler.

Som konklusion vil vi som gruppe sige at vores energi tidligt i opgaven gik meget på at lave en implementering der mere eller mindre matchede det diagram der blev udgivet til undervisningen på figur 4.

Opgaven er meget tæt på at være færdig. Vi havde et problem med event håndteringen i blandt 2 klasser, som vi brugte ufattelig meget tid på at rette, men som vi ikke kunne løse. Vi håber inderligt at vi kan få en chance/tid til at genaflevere denne opgave, da vi er meget tæt på at være færdige. Vi har som sagt under Test afsnittet forklaret, hvordan vi havde tænkt os at lave test casene, for genafleveringen.