Air Traffic Monitoring
I4SWT Mandatory exercise

Skrevet af:
Gruppe 22
Nicolaj Christian Knudgaard Fisker : au534990 : au534990@uni.au.dk[1]
Frederik Salling Østergaard : au588968 : au588968@uni.au.dk[2]
Jesper Søgaard Kristensen : au302180 : au302180@uni.au.dk[3]
Rasmus Duedahl Vesterheim : au523693 : au523693@uni.au.dk[4]

Jenkins: http://ci3.ase.au.dk:8080/job/Team22ATMBuildCoverage/

GitHub: https://github.com/ScanfErBae/SWT_Assignment2_ATM

[1] Github navn: MightyIdefix
[2] Github navn: Salling008
[3] Github navn: MisterJelle
[4] Github navn: Jogans

Tilgang til opgaven

Opgaven er blevet gået til ved, at der i gruppen er blevet identificeret en række kernefunktioner, som skulle være løst for at opnå en korrekt adfærd. Disse kernefunktioner er efterfølgende blevet sat i forbindelse med et sekvensdiagram for at kunne overskue systemet. Denne opgave er blevet gjort i fællesskab for at opnå en fælles forståelse for opgaven.
Sammensætningen af adfærden i systemet har ændret sig markant gennem opgaveprocessen. I kraft af at der igennem forløbet er kommet flere opdagelser og erkendelser.

I begyndelsen havde vi tænkt os at lave et system, som var bygget op omkring en controller klasse. Denne controller klasse skulle stå for at modtage eventet fra Receiver Transponder. Herefter skulle den uddelegere samtlige opgaver. Der skulle bl.a. være en lommeregner og en klasse til at tjekke, om to fly er for tæt på hinanden. Efter at have lavet nogle udkast til klassediagram og sekvensdiagram gik det op for os, at kontroller-klassen ville komme til at have utroligt meget ansvar, hvis systemet bliver implementeret på denne måde.

Vi valgte derfor at snakke med Frank og Bjarke omkring vores tanker om designet, hvor vi fik bekræftet, at det ikke var en optimal måde at lave systemet på. Det ville yderligere være meget svært og anstrengende at teste en så omfattende controller-klasse. Efterfølgende gik vi i gang med at designe et nyt system, og vi blev enige om, at det var en god idé at lave størstedelen af systemet som en pipeline, hvor vi efterhånden vil få sorteret i alle flyene.

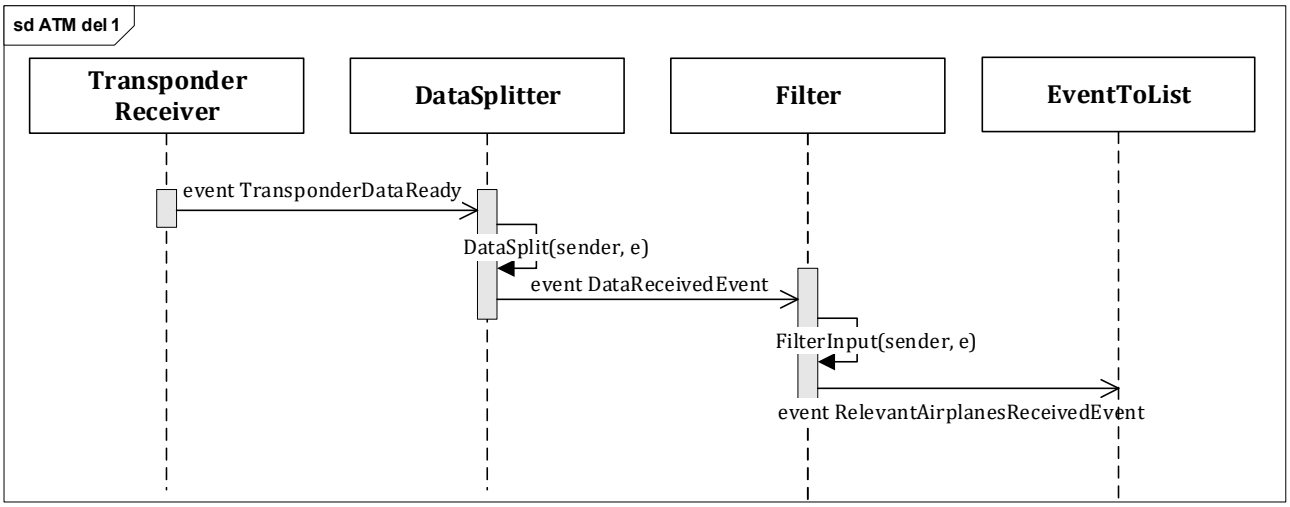
De klasser, der indgår i pipelinen, er blevet opdelt således, at de har en singulær funktionalitet. For på den måde at være meget testbare og ikke-komplekse. Dog er ulempen ved denne tilgang, at der forekommer flere viderebringelser af events i pipelinen, for at facilitere kommunikationen igennem hele pipelinestrukturen. Men denne ulempe er marginal, da de involverede test er lignende tidligere test af events, samt at fordelen ved ikke-kompleks kode er stor.

Dog er pipeline-strukturen ikke blevet ført igennem hele opgaven. I stedet har gruppen valgt at oprettet en Plane klasse, der indeholder properties for hver af de datapunkter, der er tilknyttet et fly. Det er en liste af disse fly objekter, som sendes rundt i ATM systemet.
Hermed bliver relevant information samlet i et enkelt fly objekt, således at der ikke skal sammenholdes data mellem forskellige lister af objekter.

Vi er altså endt med at have et pipeline-system, hvor vi imellem vores filtrering af flyene sender data rundt ved brug af events, mens vi til sidst i processen har valgt at have en højere kobling, da vi følte, at gav god mening.

Sekvensdiagrammer og kodens adfærd

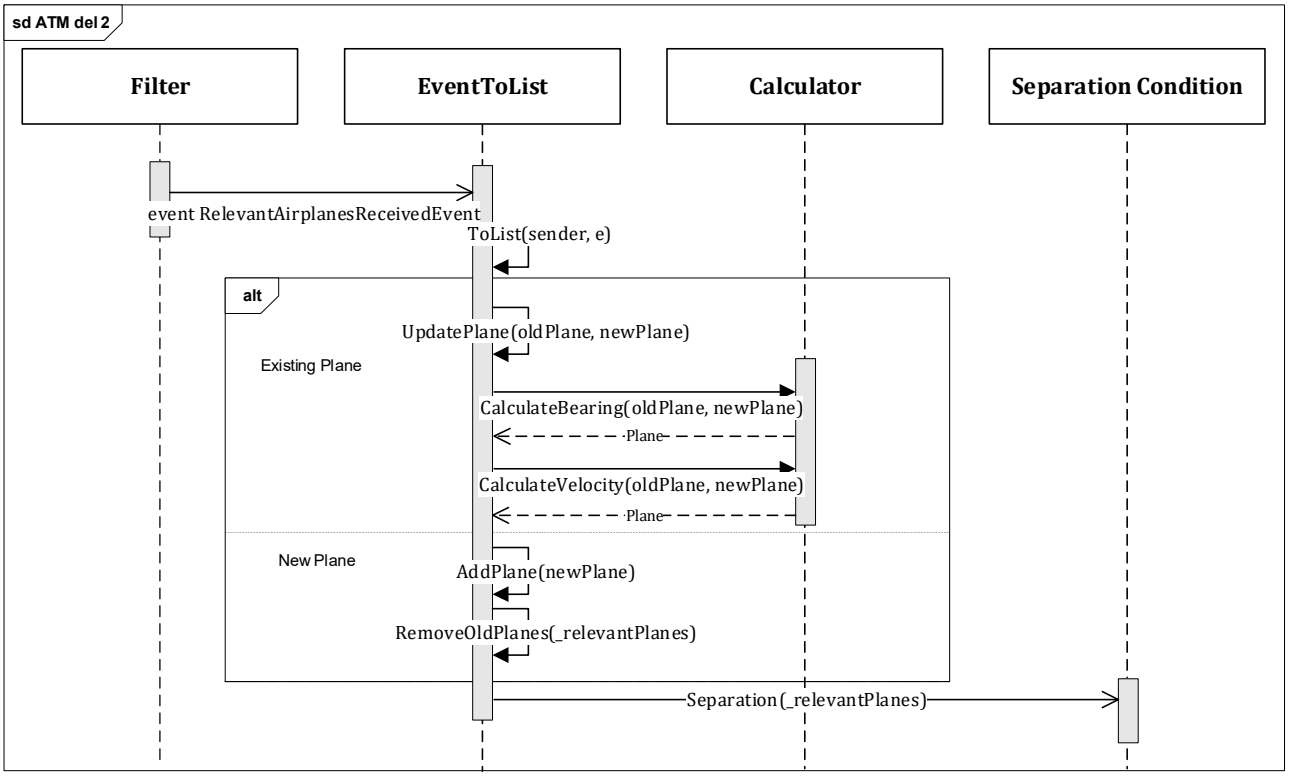
Nedenfor ses første del af projektets sekvensdiagram, der viser de tre første klassers adfærd. Denne opsplitning er blevet gjort at for kunne gøre sekvensdiagrammet mere overskueligt og samtidigt kunne vise de enkelte metodekald. Sekvensdiagrammerne er blevet opdelt over 3 diagrammer, der hver fokuserer på hver deres adfærd i systemet.
Samtlige sekvensdiagrammer er blevet oprettet således at der vises et aktiverende metodekald, samt oprindelsesklassen for dette metodekald. På samme måde vises et sekvensdiagrams afsluttende metodekald og i hvilken klasse dette kald ender.



Figur 1 Sekvens diagram over ATM del 1

DataSplitter klassen, der fremgår på figur 1, står for at modtage events fra Transponder Receiver. Når DataSplitter modtager et event, deles strengen op og der oprettes nye Plane objekter med de medsendte værdier. Disse fly bliver gemt i en liste, som til sidst bliver sendt videre i systemet som et nyt event.

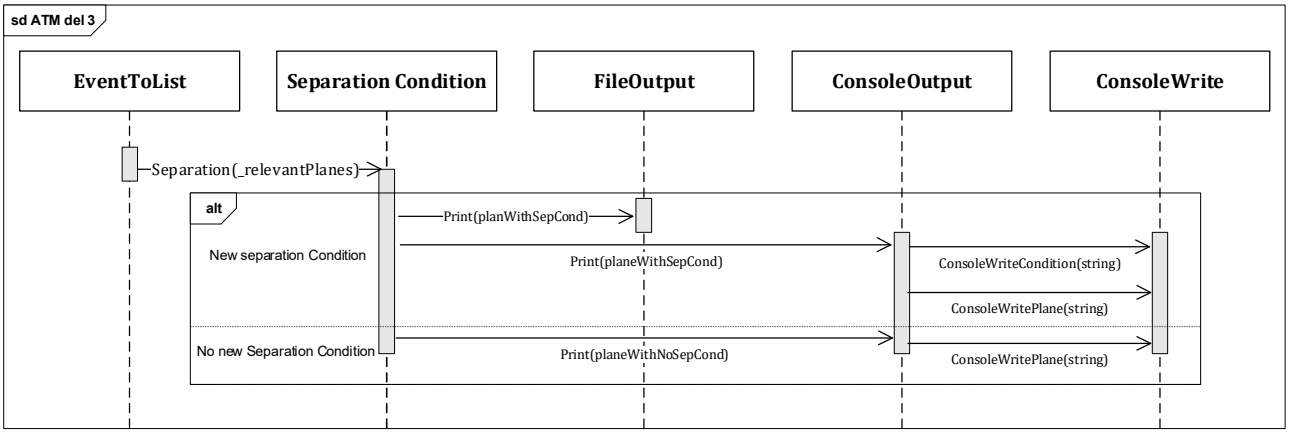
Herefter modtager vores Filter klasse de næste events. I Filter klassen sorterer flyene efter gyldighed, så vi kun kigger på de fly, som er i vores luftrum. De fly, der er i vores luftrum, bliver tilføjet til en liste, og når vi har løbet alle flyene fra eventet igennem, sender vi listen af fly videre via et nyt event.



Figur 2 Sekvens diagram over ATM del 2

På figur 2 ses sekvensdiagrammet der viser forholdet mellem Filter, EventToList, Calculator og Speration Condition. Vores EventToList klasse gemmer på en liste af de fly, vi fik fra det tidligere event. Denne liste bliver sammenlignet med flyene fra det seneste event og afhængigt af flyenes status, sker der forskellige ting. Til at starte med markerer vi alle fly i vores liste til at være som værende irrelevante.

Hvis vi modtager et nyt fly, så bliver det tilføjet til vores liste af fly og vi sætter et flyet til at være relevant. Hvis vi modtager et fly, som vi allerede havde i vores liste, så opdaterer vi flyet i listen og markerer det som værende relevant. Til sidst løber vi listen igennem og fjerner fly, som ikke længere er relevante. Til sidst sender EventToList listen af fly videre til vores Separation Condition klasse.

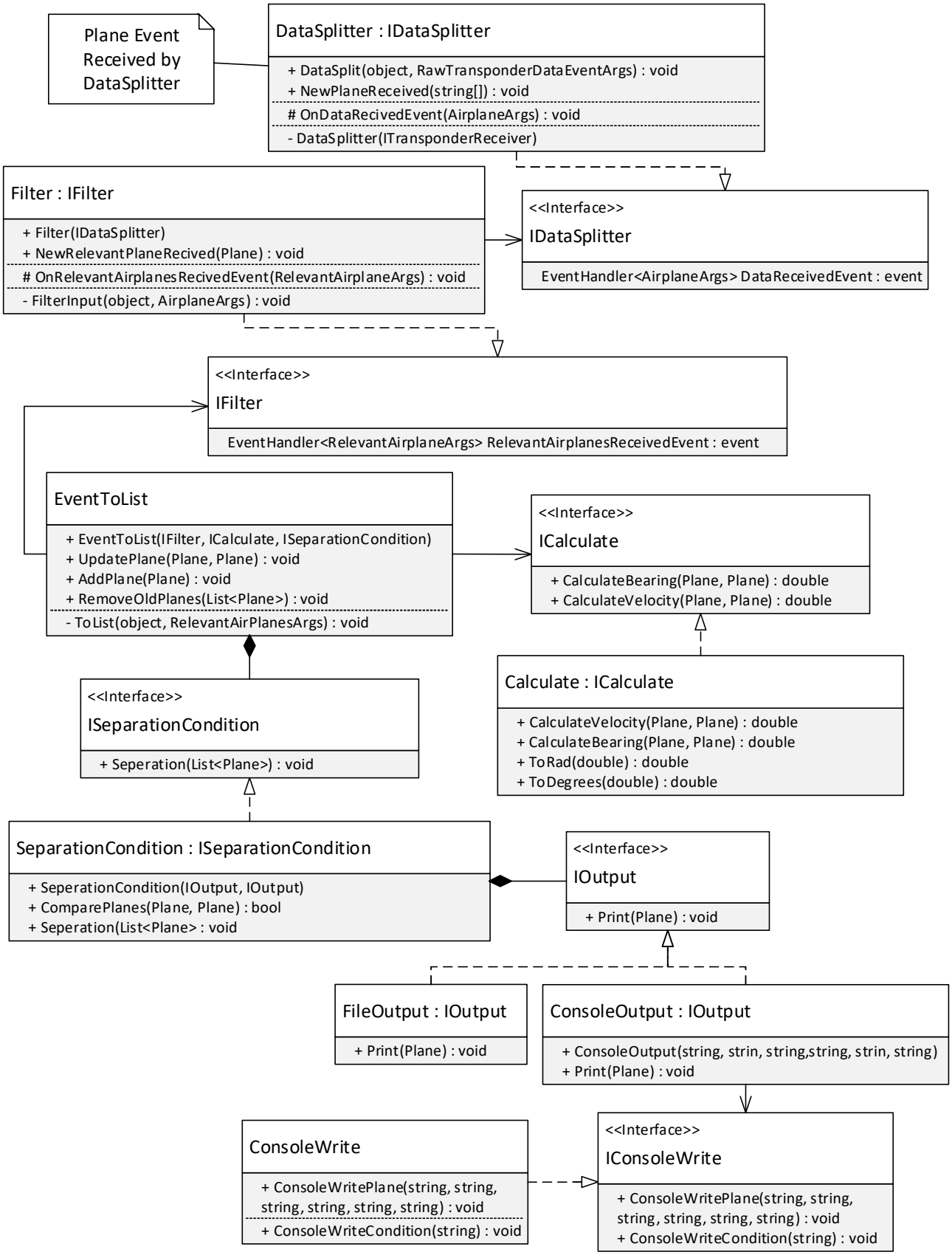


Figur 3 Sekvens diagram over ATM del 3

På figur 3 ses sekvensdiagrammet der viser forholdet mellem Controller, Separation Condition, FileOutput, ConsoleOutput og ConsoleWrite. I Separation Condition klassen løber vi listen af fly igennem og sammenligner deres koordinator for at se, om de flyver tæt på hinanden. Vi har implementeret dette ved at alle fly har en liste af strenge, hvori vi gemmer tags på den fly, som de er tætte på. Hvis vi opdager, at to fly er for tæt på hinanden, tjekker vi, om de allerede var tæt på hinanden ved sidste måling. Hvis det er første gang, de flyver for tæt på hinanden, tilføjer vi dem til hinandens liste af tags, hvorefter de bliver sendt videre til FileOutput klassen. Hvis de allerede står i hinandens liste af tags, sender vi dem ikke til FileOutput. Hvis to fly ikke er for tæt på hinanden, tjekker vi, om de står i hinandens liste af tags, og fjerner dem, hvis de er. Til sidst sender vi alle fly til ConsoleOutput klassen.

I ConsoleOutput klassen kaldes der forskellige udskrift adfærd alt afhængig om flyene er tætte på andre fly, da de udover den normal udskrift til consolen også skal skrives ud med en advarselsbesked, hvis de er for tæt på andre fly. I begge tilfælde sammensættes et fly objekts data til en string af data, som indeholder hele udskriften for det pågældende fly objekt, der sendes videre til ConsoleWrite klassen.
Klassens print metode modtager et plane ad gangen. Dette giver en begrænsning, da console clear ikke kan placeres i denne klasse. I stedet er console.clear blevet placeret i klassen EventToList, da denne klasse er den sidste der indeholder en liste af planes. Dermed kan der ved afslutning af listen cleares consolen.

ConsoleWrite klassen udskriver den modtagne streng ved console.write. Klassen er blevet dannet for at kunne indkapsle metoden for konsoludskrift, således at den udelukkende består af et metodekald, der er indbygget i C# standarden. Dermed kan en test af denne klasse være unødvendigt, da funktionaliteten af C# kernefunktioner er garanteret fra Microsoft.



Figur 4Klasse diagram over ATM

Struktur og Klassediagram

På figur 4 ses klassediagrammet for projektet, hvor den valgte struktur fremgår. En generel fremgangsmåde som er forsøgt implementeret har været at benytte interfaces mellem hver klasse.
Dette er gjort for at overholde SOLID principperne og nedbringe koblingen i systemet. Derudover har det været et godt udgangspunkt at kunne generere fakes til at teste koden.
Der er dog en undtagelse, hvor en klasse ikke har et interface. Dette sker ved klassen EventToList, der er overgangen mellem pipeline strukturen og planeobjekt strukturen, der skyldes at der ikke er dependensies af EventToList klasse i andre klasser.
Selve pipeline strukturen kan godt genfindes i klassediagrammet på forrige side, hvor en ligefrem rute fra et modtaget event til klassen EventToList fremgår.
Herefter ses klassen EventToList er afhængig af både Calculate Klassen og Separation Condition klassen.

How did the use of a CI server help you – did it help you at all? How/why not?

Continuous Integration (CI) er i dette projekt blevet benyttet ved Jenkins Server til at dokumenteret samt opnå overblik over projektet. Dokumentationen er tilstedekommet ved at oploade gentagne iterationer af koden. Hvor CI har været særlig brugbar i forbindelse med test og overskueligheden af disse.
Ved at benytte coverage funktionen har gruppen kunne få et kvantitativt overblik over hvilke kodedele, som endnu var uberørte, og ud fra dette kommunikere fremtidige arbejdsopgaver.
Derudover har grafen over de forskellige iterationer af kode kunne give et overblik over omfanget og antallet af test.
Det har været en fordel ved at der på baggrund af dette redskab kunne kommunikeres ud fra en fælles forståelse af problemet. Denne kommunikation kan gøres uden at sidde med koden åben.
En af ulemperne kan være at dette redskab kan blive det definerende metrik til at estimere om et projekt er blevet testet tilstrækkeligt. Det er meget belejligt at tænke at projektet med en 100 % coverage er lige som det skal være. Men en egentlig forståelse af kodens omfang og adfærd er nødvendig for at kunne teste koden tilstrækkeligt, og kan altså ikke erstattes af CI.
Dog er det meget behageligt at opnå en høj procentvis coverage. Der er ikke opnået i 100% coverage i vores system. Dette skyldes at Jenkins opfatter main som en klasse, som skal testes også selv om at Main ikke kan testes. Console.clear på linje 29 er udkommanderet, da Console.clear gjorde det svært at teste EventToList.

How did you divide the software classes between group members for implementation and test? Why did you divide it as you did?

Da der var overblik over programmets kerneopgaver, blev enkelte arbejdsopgaver fordelt mellem gruppemedlemmerne. Disse opgaver blev fordelt i forhold til medlemmernes umiddelbare interesse og forståelse af de enkelte dele. For de fleste af klasserne er der arbejdet to og to, så der var større forståelse bag koden og det var større sikkerhed på om koden kunne løse opgave. Nogle opgaver blev fordelt, så der var en enkelt person, der skulle lave klassen, men vi har dog været gode til at spørge hinanden efter hjælp. Nogle opgaver har dog været for svære og dermed skulle de delegeres til et gruppemedlem med større kompetencer.
Rækkefølgen for løsninger for kerneopgaverne er blevet sket ud fra om en standalone klasse kunne oprettes. Et eksempel på dette er klassen Calculate, hvor problemet bestod af omdanne koordinater og tid til hastighed og retning. Andre kerneopgaver, der kunne løses til at starte med, var at håndtere modtagelsen af events i starten af systemet, der endte ud med at hedde DataSplitter.

Andre klasser har været afhængig af forrige dele i systemet og en generel forståelse af kommunikationen i systmet. Dermed kunne disse klasser først færdiggøres senere, når disse ubekendte er fundet. Dette kan ses på Klassediagrammer på figur 4. Hvilket gav en naturlig rækkefølge for løsning af projektets første dele.

Undervejs har der været en tvivl om løsningen af problemet. Hvilket har medført, at det var naturligt med et større fokus på funktionaliteten af klasserne end at teste hen over vejs og lave unit test. Dette medførte at langt de fleste unit test er blevet implementeret efter at der hovedfunktionaliteten var kommet på plads. Vi følte, at det gav mere mening at teste noget der fungerede, end at finde ud af, hvordan man tester noget der ikke fungere. Denne opgave har dog gjort os klogere og vi har forstået. Vi fandt ud af, at continuous integration og test driven programming ville havde hjulpet os med at identificere, hvad der fejlagtiggjorde et build og hvilke skridt der skulle tages for at løse fejlen.

In general, describe anything good or bad you have learned from this exercise in the report.

Vi har lært at test driven programmering er mulig løsning, men er svært at praktisere. Det virker til at man skal kende løsningen på programmet for at kunne lave testene før implementeringen af programmet. Det var svært for os at lave dette, da vi fra starten havde lavet et design der ikke var optimalt for test driven programming.
Dog har det vist sig, at hvis man allerede fra implementeringen af design og kode medtænker unit test, kan denne del af processen gøres med naturligt og fandens meget lettere. End at skulle skabe miljøer med x antal fakes, for blot at undersøge en enkelt funktionalitet.

Vi har fået en bedre forståelse for og forudsætning for at kunne lave test fra starten. På den måde vil testene være objektive og teste programmet hen over vejs, så problemerne med programmet bliver mere tydelige og nemmere at finde. Af disse grunde skulle vi have været bedre til at lave test noget tidligere. Generelt er vi blevet bedre til at lave de forskellige former for test, blandt andet test til eventdriven programming. I forbindelse har vi lært hvad de forskellige gør og vigtigheden af at lave test først.

Vi har også lært, hvordan man laver og tester et større system. Heriblandt hvilke overvejelser der skal laves for at systemet fungere. Der er mange dele fra software design der skulle interageres for at de enkelte klasser kunne testes.