AIR TRAFFIC MONITORING

I4SWT Mandatory exercise

Team 10

Mathias Mølgaard – 201509795 – 201509795@post.au.dk
Lasse Torp Jacobsen – 201708878 – 201708878@post.au.dk
Jeppe Christensen – 201706538 – 201706538@post.au.dk
Kathrine Kahns Hille – 201708081 – 201708081@post.au.dk

Jenkins: http://ci3.ase.au.dk:8080/job/SWT_Gruppe10_AirTrafficMonitoring/
Github: https://github.com/LasseTorp/SWT_Gruppe10_AirTraficMonitoring.git

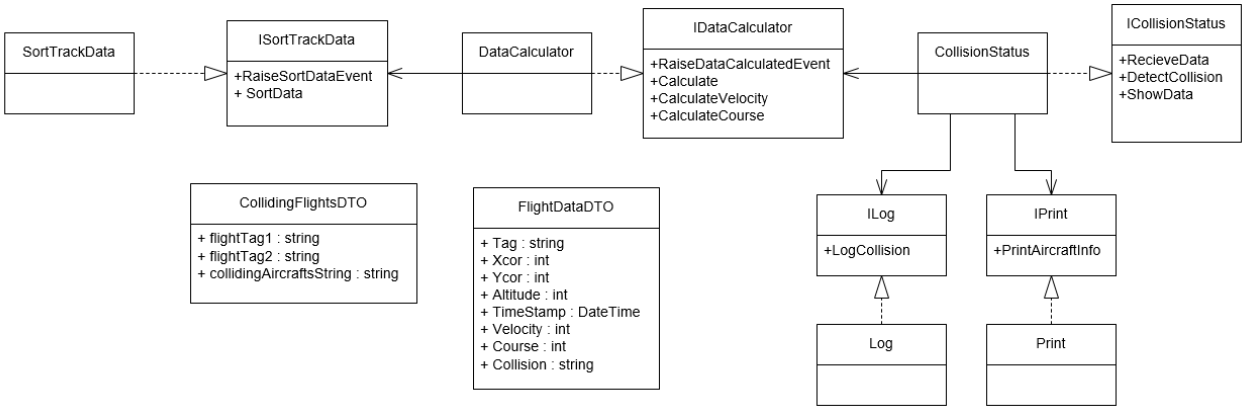
Indholdsfortegnelse

Software Design

Gennem designfasen var målet at skabe en løs kobling. Dette for at skabe et system, som overholder principper såsom single responsibility, open-closed m.fl. for at opnå et testbart system. I starten af designfasen havde gruppen lavet et klassediagram og et design, der byggede på observer- og strategypattern, som tidligere blev anvendt på 3. semester. Dog blev dette ikke det endelige design, da gruppen ved grundigere gennemlæsning af opgaven samt undervisningsmateriale på 4. Semester fandt frem til, at programmet i sidste ende ville blive mere enkelt og dermed også mere testbart, hvis designet i stedet byggede på pipelineprincippet.
Ud fra ovenstående beslutning blev der designet et pipelinesystem. Dette gjorde at en klasse bliver "vækket" når dataene er klar, og "vækker" den næste klasse som skal bruge dataene, når den først nævnte klasse er færdig med sin bearbejdelse af den pågældende data. På denne måde tilkobles klassen, der har foretaget beregningen til klassen, der skal bruge beregningen. Dette bliver så praktisk set et observerpattern, da en klasse, der behøver data bliver 'notificeret' når den nødvendige data er færdigbehandlet og sendt videre. Måden der bliver notificeret på er, at der bliver oprettet og "raised" et event når behandlingen af data er færdiggjort, så den næste klasse i programmet ved at dennes nædvendige information er klar, så denne kan påbegynde sin opgave. For at bestemme rækkefølgen på den data, der skal anvendes til forskellige bestemmelser samt beregninger i programmet forskellige klasser gennem programmet, er der blevet anvendt events. Det er gennem disse events, at klasserne kan notificere videre i systemet, når en beregning eller bestemmelse af data er foretaget, og den færdigbehandlede data nu kan anvendes i "næste" klasse.
Der er yderligere blevet anvendt tripple-I der omfatter, Identify, Interface og Inject. Dette da der skal identificeres afhængigheder, samt at koblingerne så skal gøres løsere ved brug af interfaces. Ved brug af triple-I bliver programmet i sidste ende yderligere mere testbart, da vi så kan afkoble enkelte klasser fra resten af systemet. De enkelte klasser er bruges i testprogrammet, da de er det enheder, der skal testes. Når disse er mere løst koblet til resten af programmet er de let testbare og man har mulighed for at kontrollerer hvilke afhængigheder den pågældende enhed bruger, hvilket i testprogrammet skal være den "fake"-afhængighed gruppemedlemmerne har oprettet til en specifik test.

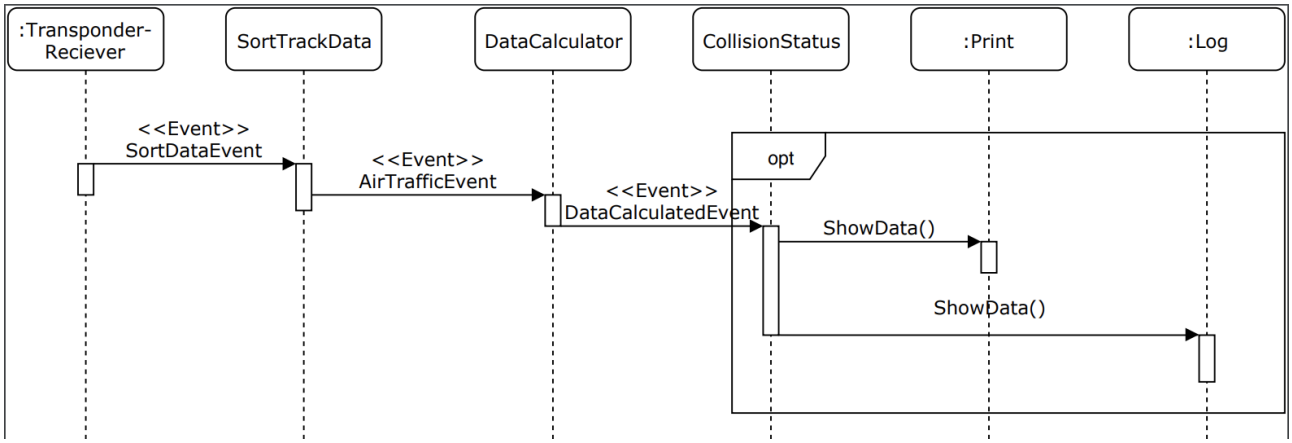
Ydermere bruges der gennem programmet constructor injection hvor der ved oprettelsen af et objekt, injiceres en instans af den klasse der skal bruges. Dette gør det også muligt at kontrollere afhængighederne når klassen skal testes.
Slutteligt har vi tilstræbt os efter at efterleve design princippet omhandlende single responsibility, dette også for at gøre det nemmere at teste klassernes idet hver enkel klasse så kun varetager en opgave, og dermed kun har én funktionalitet, der skal testes.

Klassediagram



Den første klasse som modtager data fra DLL-filen er SortTrackData. Her decodes den data som modtagers i programmet. Data for de enkelte fly modtages som strings, hvor hver information er opdelt af et semikolon. I klassen SortTrackData deles den modtagne information op, så hvert enkelt flys information deles ud i en FlightDataDTO, som lægges i en ny liste. Denne nye liste sendes videre via event til klassen DataCalculator, hvor hastigheden og kursen beregnes, og efterfølgende lægges ind i de enkelte DTO objekter. Herefter sendes den opdaterede liste af DTO-objekter videre via event til klassen CollisionStatus, hvor der beregnes om nogle fly er ved at kollidere. Ved tilfælde af kollision benyttes klassen log, hvorfra der logges til en fil hvilke fly der er ved at kollidere og tidspunktet. Der logges kun en gang pr potentiel kollision. Klassen print benyttes hele tiden til at printe de forskellige flys oplysninger.

Sekvensdiagram



Sekvensdiagrammet viser hvordan der fra de forskellige klasser bliver oprettet og raised events, som så gør "næste" klasse opmærksom på, at den information som denne har behov for at gå videre med bearbejdning af information nu er klar. Derefter påbegynder pågældende klasse sin behandling af den modtagne information og opretter og raiser så et event der notificerer videre i systemet.

Implementering og test af design

Gruppen startede, som beskrevet i øverste afsnit, med at lave klassediagrammer og vælge det design, der virkede mest effektivt og mest testbart. Dette klassediagram gjorde den kode, der skulle til at skabes mere overskuelig. Derudover gav det alle gruppemedlemmer en god forståelse og et godt udgangspunkt for at kunne påbegynde kodningen. Planlægningen af koden blev prioriteret i starten af projektet, hvilket viste sig at give pote gennem arbejdet. Den omfattende planlægning lettede kodearbejdet fra start til slut og designet, der blev valgt fra start blev fulgt gennem hele arbejdet med koden.

I det design, der blev valgt fra start, blev det tilstræbt at hver enkel klasse kun skulle varetage en specifik opgave i forhold til programmets samlede funktionalitet. Da hver enkel klasse havde klare opgaver og afgrænsninger blev det enkelt at dele kodeopgaver ud til forskellige gruppemedlemmer. Uddelingen af kodeopgaver foregik ved fælles diskussion. Der blev snakket lidt rundt i gruppen om, hvor hvert gruppemedlem følte, at man kunne bidrage mest men samtidig også hvor man kunne få en overkommelig udfordring. Alle bød derfor ind på hvilken del de kunne tænke sig at arbejde med, og det forløb gnidningsfrit, da gruppen består af medlemmer med forskellige kompetencer, der dermed dækkede de kompetencer, der skulle til for at løse opgaven.

Ved uddeling af kodeopgaver lettede GitHub arbejdet. GitHub har gennem kodearbejdet gjort det enkelt for de forskellige gruppemedlemmer at have hvert sit ansvar, som indebar at skulle kode en klasse og løbende teste den samme klasse. Da der fra start var udarbejdet både klasse- og sekvensdiagram blev det nemt for hvert enkelt gruppemedlem at holde overblik over sammenhængen og funktionaliteten af hele koden og dermed også vide hvad det pågældende gruppemedlems individuelle opgave bestod i.
Denne måde at uddele kodeopgaver på blev valgt, da denne virkede som den mest praktiske og mest effektive måde at gøre det på. Dette da der så ikke var flere gruppemedlemmer, der skulle arbejde i de samme klasser samtidigt og der så i høj grad ikke vil opstå merge-conflict ved opdatering af koden, når hver enkelt medlem havde committed og pushed nyligt færdiggjort arbejde. Pipeline princippet var i høj grad det, der gjorde det enkelt at dele arbejdsopgaver ud samt at arbejde synkront med koden, da koblingen i programmet blev hold løs. Dermed blev det også muliggjort at teste de enkelte individuelle klasser uden uafhængigt af hvor langt de andre gruppemedlemmer var med deres kodearbejde.

Ved test af koden, blev der oprettet testklasser, der testede de enkelte klasser i programmet. I hver testklasse er det vigtigt at foretage tests der dækker de mulige outcomes, der kan være fra de forskellige metoder. Dette så man kan forsøge at sikre, at koden fungerer som ønsket samt at der ikke sker noget uforudset, når koden kører. Det er en rigtig god måde til at opdage hvorvidt koden dækker, altså om den beregner det man ønsker og om der er farezoner i forhold til, hvis der kommer ”forkert” information ind i systemet.
Der blev oprettet test til både at tjekke at det var den rigtige information, der kom ind i klasserne samt til at tjekke at bearbejdningen af informationen skete på korrekt vis og gav de ønskede resultater. Til at teste blev der anvendt NSubstitue og skabt fakes til at teste de forskellige enheder under test (Unit Under Test, UUT). Ved problemer med dækningen af den klasse, der ønskedes testet eller ved problemer med at skrive en tilstrækkelig test blev der konfereret i gruppen. På denne måde har alle gruppemedlemmer oprettet og kodet andres tests, men samtidig også været ind over andre gruppemedlemmers test.
Det viste sig også at pipeline designet gav pote i forhold til test, da det var enkelt at teste ud fra de forskellige events og de enkelte klassers metoder og funktionalitet.

Løbende og efter testene var blevet kodet og kørt, blev Jenkins anvendt til at tjekke hvorvidt alle kodelinjer i programmet var dækket af test, så gruppen have mulighed for at konkludere hvorvidt der skulle skrives flere tests. Dette var et godt værktøj og gjorde løbende, at der blev ændret ting i testen så vi forbedrede coverageprocenten. Til slut havde vi en coverage procent på 98%, hvolket også er beskrevet yderligere i afsnittet herunder. Dette kunne konkluderes at være det gruppen maksimalt kunne opnå og derfor var dette et tilfredsstillende resultat og gruppen skrev ikke yderligere test til systemet.

Gennem arbejdet med projektet er det tydeligt, at gruppen har forbedret sig på både det kodetekniske og i forhold til samarbejde i gruppen. Gruppen har i høj grad lært om events og brugen af disse, samt hvordan det kan lette processen i forhold til både kodearbejdet på det primære system og testarbejdet. Derudover har gruppen tilegnet sig et større overblik over testbarhed og hvordan man i høj grad kan designe sig frem til god testbarhed.
Det står meget klart for gruppen efter dette gruppearbejde hvor meget et godt design kan betyde for hele gruppearbejdet, både i forhold til de enkelte gruppemedlemmers overblik over koden samt selve funktionaliteten af de forskellige klasser. Dermed blev det også meget enkelt at arbejde sammen, da forskellige gruppemedlemmers ansvar var klart defineret og dette gav klare linjer. Det har givet et godt overblik over hvordan man skaber en effektiv og produktiv proces i forhold til at designe, implementere og teste kode i et større projekt end gruppen har lavet forud.

Help from CI-servers

GitHub har været et yderst brugbart værktøj, da GitHub har gjort det muligt at arbejde individuelt og parallelt i gruppearbejdet. Dette parallele arbejde har gjort det nemt og effektivt at færdiggøre gruppearbejdet.
Gennem arbejdet med Handin 2 har det været en stor hjælp at anvende Jenkins. Jenkins har givet muligheden for at få adgang til en coverage-rapport. Denne har gjort det tydeligt hvilke dele af koden, der er blevet testet. Coveragerapporten beskriver i detaljer hvilken del af koden, der er blevet dækket af test. Det er hele ned på linjeniveau man får en rapport omkring hvorvidt den enkelte linje bliver dækket af de skrevne tests. Denne rapportering gør det muligt at arbejde målrettet og effektivt med test af den allerede eksisterende kode og dermed opnå en tilfredsstillende coverage. En tilfredsstillende coverage opnås ved at 100% koden er dækket af

test. Gruppens coverage er 98%, hvilket også er fuldt acceptabelt, da de manglende 2% er på baggrund af, at programklassen, der indeholder Main-metoden, ikke testes, hvilket den heller ikke skal. Denne er dog mulig at fjerne fra coveragerapporten, hvilket så havde givet 100%, men dette er ikke lykkedes for gruppen, og dermed forbliver coverage procenten 98%.
Jenkins har yderligere gjort det nemmere at holde overblik over de forskellige tests, da det var muligt, at teste samtlige tests samtidigt. Hvis der i testene opstod fejl, ville der komme en fejlmelding, der så informerer om hvor i koden denne fejl er opstået.