HÖGSKOLAN
SKÖVDE
Institutionen för kommunikation och information

Utformning av en kommunikations- och dataöverföringsarkitektur till IBDn

Utformning av en kommunikations- och dataöverföringsarkitektur som är anpassad till SAAB:s Intelligent Behavior Detector

Fannie Stenemo

Examensarbete i datalogi med inriktning mot dataspelsutveckling
30 hp C-nivå, vårterminen 2010

Utformning av en kommunikations- och dataöverföringsarkitektur

Examensrapport inlämnad av Fannie Stenemo till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information. Arbetet har handletts av Alexander Karlsson.

2010-06-09

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: ____

Utformning av en kommunikations- och dataöverföringsarkitektur

Fannie Stenemo

a07fanst@student.his.se

Handledare: Alexander Karlsson

Sammanfattning

Denna rapport undersöker möjligheten att bryta ner ett stort program till mindre och enkaler moduler och skapa en dataöverförings- och kommunikationsarkitektur mellan dem. Dataöverföringen skall vara dynamisk för att olika moduler skall under exekvering ha möjlighet att byta modul den vill ha data ifrån samt centralisera koden och kommunikationen skall optimera antalet meddelandeöverföringar. Arbetet resulterar i två arkitekturer som uppfyller tidigare satta målen i ett parallellt system, arbetet analyseras sedan analytiskt och styrkor och svagheter undersöks. Arkitekturerna medför att många problem med att utveckla stora system undviks och detta skapar fler möjliga kopplingar, mindre komplexa moduler samt centralisering av funktionalitet i programmet.

Nyckelord: Arkitektur, Designmönster, kommunikation, dataöverföring, parallellt system.

Innehållsförteckning

1 Introduktion ... 1
2 Bakgrund ... 2
2.1 Mjukvaruarkitektur ... 2
2.2 Intelligent Behavior Detector ... 3
2.2.1 Exempel ... 4
2.2.2 Generalisering av systemet ... 5
3 Problemformulering ... 6
3.1 Problem ... 6
3.1.1 Delmål 1 – Designmönster och designval ... 7
3.1.2 Delmål 2 - Prototyp ... 7
3.2 Utvärderingsmetod ... 8
4 Genomförande ... 10
4.1 Prototyp ... 10
4.1.1 Designval ... 10
4.1.2 Händelsehantering ... 12
4.1.3 Datahantering ... 14
4.2 Prototyp ... 15
4.2.1 Händelsehantering ... 15
4.2.2 Datahantering ... 15
5 Resultatanalys ... 17
5.1 Händelsehantering – Antal meddelanden ... 17
5.1.1 Scenario ... 17
5.1.2 Undantag ... 18
5.2 Händelsehantering – Relevanta meddelanden ... 19
5.3 Datahantering - Dynamisk sammankoppling ... 20
5.4 Datahantering - Centralisering av funktionalitet ... 21
5.4.1 Koden ... 21
5.4.2 Trådar ... 22
5.4.3 Tidskomplexitet ... 23
5.5 Analys Sammanfattning ... 25
5.5.1 Händelsehantering ... 25
5.5.2 Dataöverföring ... 25
6 Slutsats ... 26

6.1 Sammanfattning ... 26
6.2 Diskussion ... 26
6.3 Framtida arbete ... 27
Referenser ... 29

1 Introduktion

Datorerna har ändrats mycket och dagens datorer har inte bara en processor utan ofta två eller flera som hjälps åt med arbetet. Dagens program, speciellt stora system som behöver den extra processorkraften, blir i större och större utsträckning anpassade för att exekveras parallellt. För att bygga ett parallellt system behöver utvecklarna främst tänka på att synkronisera och kommunicera mellan olika processer (Ben-Ari, 2006). Detta eftersom två processer som exekveras samtidigt inte skall använda och ändra i samma data i det gemensamma minnet. Dessutom skall de kommunicera med varandra för att dela information.

Programmet som arbetet riktar sig mot är SAAB:s produkt Intelligent Behavior Detector vilket är ett analyssystem som får in data om objekt och analyserar dessa objekts beteenden över tid för att finna olika mönster. Detta skall användas av t.ex. kustbevakningen för att varna när båtar kommer för nära land, håller på att krocka eller beter sig på annat sätt avvikande. Produkten delas upp i mindre moduler för att sedan kommunicera med varandra och föra över data, detta för att minska komplexitet m.m. i systemet. Modulerna exekveras parallellt och detta ställer stora krav på den interna kommunikationen mellan modulerna, eftersom det skall klara av allt från att göra uträkningar på en stor mängd objekt till att kommunicera mellan olika delar av programmet.

Detta arbete kommer att fokusera på att skapa en förbättrad kommunikations- och dataöverföringsarkitektur till Intelligent Behavior Detector. Först analyseras den äldre arkitekturen och kriterier för den nya arkitekturen sätts upp. De stora kraven på arkitekturen är att antalet meddelandeöverföringar skall minska i systemet, att koden skall centraliseras och att dataöverföringen skall vara dynamiskt. Designmönster som är anpassade för dessa kriterierna samt för IBDn undersöks och utvärderas, och även designval som kan användas analyseras. Här upptäcktes det att det fanns några stora skillnader mellan de olika kriterierna och det valdes därför att det skall skapas två separata arkitekturer, en för händelsehantering och en för dataöverföring.

De två arkitekturerna tas fram genom att slå samman flera olika designmönster och designval. Designmönster väljs utifrån hur väl de är anpassade för problemen samt hur väl de passar ihop med andra designmönster. De skapade arkitekturerna analyseras och utvärderas för att se att kraven på arkitekturerna är uppfyllda. Analysen visar hur de olika kriterierna är uppfyllda genom att analysera olika scenarion så som skickade händelser i systemet och vad som har förändrats i koden samt i trådarnas beräkningar vid olika scenarion. Dessa analyser visar att kraven som sätts upp i kapitel 3.1 uppfylls.

2 Bakgrund

Detta kapitel börjar med att ge en inblick i hur mjukvaruarkitekturer byggs upp genom olika designmönster. Sedan beskrivs SAAB:s Intelligent Behavior Detector programvara vilken är mjukvaran som skall vidareutvecklas under detta arbete.

2.1 Mjukvaruarkitektur

Det är en stor process att bygga ett program och genom att spendera mycket tid i början på planering och utveckling av en arkitektur och val av designmönster kan många fel, som annars kan leda till problem i slutskedet, undvikas.

Designmönster används vid utvecklingen av arkitekturer och visar lösningar på vanligt förekomna problem, det är en mall för det specifika problemet och ger sambandet mellan objekt på en abstrakt nivå. Utvecklare använder ofta designmönster utan att själva veta om det, de är gömda i nästintill all mjukvara (Zalewski, 2001). Designmönster är den samlade kunskapen från många tidigare utvecklare som är förfinad, testad och genomarbetad allt för att fånga upp en bra design för ett specifikt problem. Det finns många olika kategorier av designmönster (MacDonald et al., 2002) men i detta arbete används enbart samlingsnamnet designmönster för dem alla.

Definition 1 – Designmönster

Designmönster är ett samlingsnamn för designlösningar med hög abstraktionsnivå som beskriver sammanhang, problem, lösningar och konsekvenser av att använda den specifika designen (Gamma et al., 1994).

Att använda ett designmönster kan lösa ett antal vanligt förekommande problem men när storleken och komplexiteten på program ökar behöver även designen av dem följa efter och detta realiseras genom att utveckla en arkitektur. En arkitektur ger en grafisk representation av systemet på en abstrakt nivå, genom dess komponenter, dess kopplingar samt användandet av designmönster. Detta ger en överblick samt en möjlighet till utvärdering av arkitekturens styrkor och svagheter (Garlan & Shaw, 1993).

Det finns många svårigheter med att utforma en bra arkitektur, svårigheterna ligger i att välja element, specificera elementens interaktion och deras begränsningar på ett sätt som gör att de uppfyller ställda kriterier på arkitekturen. Det finns inga klara regler eller rätta sätt att bygga upp en mjukvaruarkitektur, utan det kräver en blandning av kunskap och känsla och kan genomföras på oändligt många olika vis. Arkitekturen bedöms genom dess slutgiltiga egenskaper utifrån satta kriterier och mål och representeras på olika sätt beroende på vad som skall betonas. Det är t.ex. vanligt med text till grafiken för att förklara arkitekturen mer konkret. Design och arkitektur förväxlas ibland, skillnaden är abstraktionsnivån: Implementationen visar alla detaljer, designen visar få detaljer och en arkitektur visar minimalt med detaljer och är den högsta nivån av abstraktion (Eden & Kazman, 2003). Arkitekturen hjälper utvecklingen av programvaran genom att den sätter samman alla delar och ger en abstrakt design av programmet som går att analysera.

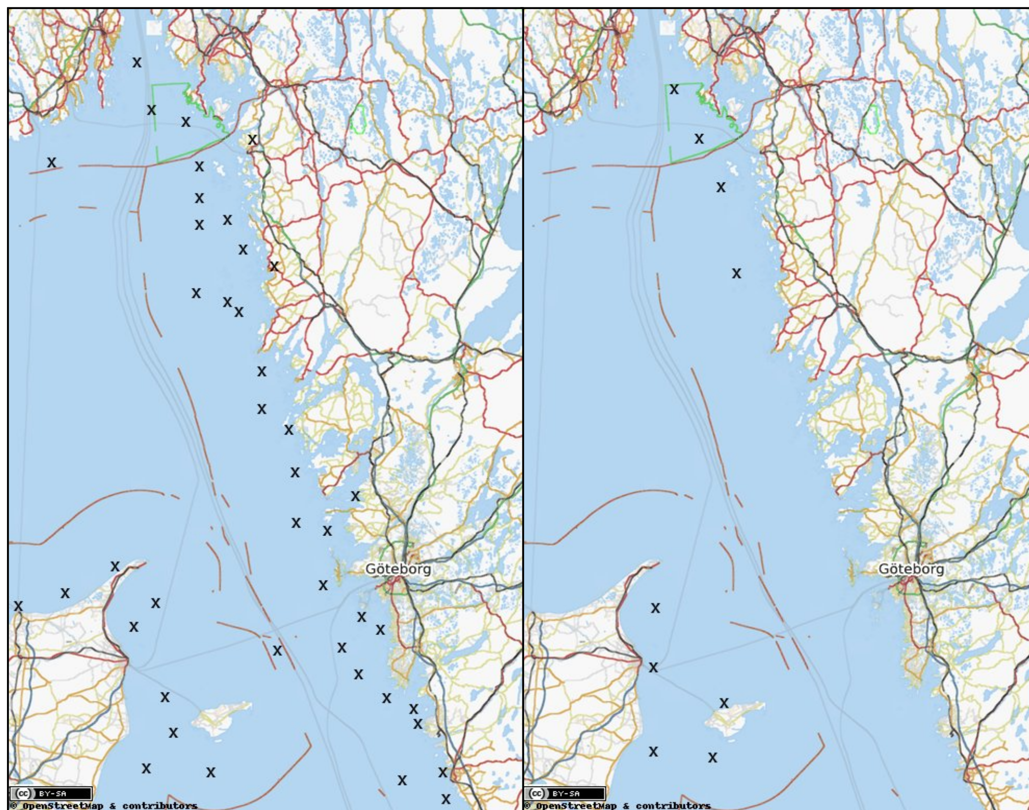
Definition 2 – Arkitektur

Mjukvaruarkitektur är ett programs struktur, vilket är en sammanslagning av systemets komponenter, deras kopplingar, designmönster och de designval som används på dessa komponenter. (Zalewski, 2001; Eden & Kazman, 2003; Garlan & Shaw, 1993).

Det som menas med designval är de val som är avgörande för utvecklingen oavsett vilka designmönster som används. Det kan t.ex vara valet mellan att automatiskt skicka data mot att begära data vilket skall ske oavsett vilket designmönster som sedan väljs.

2.2 Intelligent Behavior Detector

För att få en bra förståelse för detta arbete krävs en inblick i programvaran som kommunikations- och dataöverföringsarkitekturen skall appliceras på. Programvaran heter Intelligent Behavior Detector(IBD) och är utvecklad av SAAB Microwave Systems i Skövde. IBDn analyserar beteenden hos en stor mängd objekt över tid och informerar eller larmar när fördefinierade beteenden eller datadrivna avvikelser i beteenden upptäcks. Detta användas för att förenkla beslutsfattandet för användaren. Användaren kan fokusera på de detekterade objekten istället för att fokusera på att finna dem se, Figur 1 (SAAB group, 2009).



Figur 1 T.v. visas alla objekt IBDn har vetskap om.
T.h. visas de objekt IBDn funnit som matchar olika fördefinerade beteenden.

Människor är generellt bra på att se mönster och dra slutsatser från dem, men det människor inte klarar bra är att fokusera på många olika objekt över tid, detta kan däremot datorn hjälpa till med (Höppner et al., 1999). IBDn används därför främst för att upptäcka och visa objekt med särskilda beteenden, användaren behöver sedan enbart fokusera på dessa objekt och värdera informationen för att eventuellt agera på den. Data om objekten kommer in från olika fysiska sensorer eller simuleringar.

Användaren specificerar de fördefinierade beteendena genom att starta en eller flera olika så kallade beteendemoduler och kombinera ihop dessa så att IBDn ska känna igen de komplexa beteenden som användaren vill bli informerad om. IBDn är

uppbyggd av många olika typer av beteendemoduler som samarbetar, de stora kategorierna är: hjälpmoduler, grafiska moduler, enkla och komplexa beteendemoduler. Hjälpmodulerna utför t.ex. sortering av objekt, extrahering av särskilda typer av objekt eller sparar data i databasen. Grafiska moduler visar objekt, alarm och resultat från andra beteendemoduler i ett grafiskt gränssnitt. Enkla och komplexa beteendemoduler beräknar allt från kollision, sammanslagning och splittring av objekt till analys av båttyp och beteendeanalys. Exempel på en beteendemodul är en hastighetsmodul som får in två värden, ett minimivärde och ett maxvärde från användaren och larmar för alla objekt som har en hastighet utanför dessa gränser.

Definition 3 – Modul

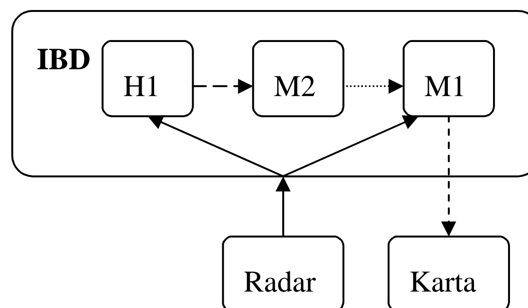
En modul är en beteendemodul, grafiskmodul eller hjälpmodul vilket i sig är en process i IBDn som utför beräkningar på objekt för att finna ett specifikt beteende och informera eller larma användaren om detta. Det går att kombinera ihop flera moduler för att framkalla mer komplexa beteenden (SAAB group, 2009).

2.2.1 Exempel

Här följer ett konkret exempel på hur IBDn ska användas för att övervaka Göteborgs hamn med omnejd, det går även att följa i Figur 2. En användare sitter framför IBDn och får in alla objekt via radar till sin dator. Användaren startar två moduler: M1 som visar alla objekt på skärmen, M2 som larmar för alla objekt av en i förväg bestämd typ, t.ex. fiskebåtar, som överstiger en hastighetsbegränsningen.

IBDn startar M1 först, den hämtar en lista med alla objekt från sensorerna och visar alla dessa objekt på rätt position i en kartapplikation. Modul2 begär information från en hjälpmodul H1, som i sin tur går igenom listan med alla objekt och tar ut alla fiskebåtar till en egen lista. Listan med fiskebåtar skickas från hjälpmodulen till M2 som går igenom listan och kontrollerar att ingen fiskebåt överstiger hastighetsbegränsningen. Om någon fiskebåt gör det skickas ett larm till M1som visas upp tillsammans med alla objekt i kartapplikationen. Användaren ser därmed när en fiskebåt inte följer angiven hastighetsbegränsning och kan agera på detta.

Detta är hur IBDn skall fungera men i början av arbetet gick det inte att skicka data mellan olika moduler och i den äldre arkitekturen bestod modul2 av funktionen ta ut fiskebåtar och sedan ta ut dem med hög hastighet. Detta gör modulen väldigt specifik och enbart användbar i ett specifikt tillfälle istället för att vara användbar i många tillfällen med olika hjälpmoduler kopplade till sig vilket den nya arkitekturen är.



Figur 2 Här visas IBDn och dess moduler och kopplingar ifrån exemplet över.

2.2.2 Generalisering av systemet

För att kunna bygga upp en bra arkitektur för kommunikationen och dataöverföringen i IBDn måste det tydliggöras vilken sorts system det är som skall användas. Detta avgör både vilka designval och designmönster som skall användas vid utvecklingen av arkitekturen, samt vilka liknande system som kan dra nytta av den färdiga arkitekturen. Här nedan tas några av de viktigaste aspekterna på systemet upp.

IBDn är ett parallellt system, vilket betyder att olika delar av systemet har möjlighet att exekveras samtidigt i en dator på olika processer. Detta ger mer kraft till programmet samt utnyttjar i många fall outnyttjad processorkraft (Hariharan & Aluru, 2005). Utveckling av parallella system är väldigt lik vanlig programutveckling, skillnaden ligger främst i att kommunicera och synkronisera mellan de olika modulerna som exekveras i olika processer. Arbetet kommer enbart att fokusera på kommunikationen i parallella system och inte synkroniseringen, vilket går att finna information om i andra rapporter. Det finns synkron och asynkron meddelandeöverföring. Synkron överföring kräver att processerna i fråga synkroniseras, sändaren blockeras därmed tills mottagaren är i takt och på liknande sätt blockeras mottagaren tills sändaren är i takt. Asynkron överföring behöver ej synkroniseras och sändaren blockeras därmed inte, sändaren använder sig av en buffer som temporär lagring av meddelandet tills mottagaren efterfrågar det. Är meddelandebufferten tom blockeras mottagaren tills ett meddelande läggs in för hämtning (Ben-Air, 2006).

IBDn är ett reaktivt system vilket innebär att systemets beteende är starkt beroende av resultatet av interna eller externa händelser. IBDn får hela tiden indata av sensorerna och beroende på dess innehåll förändras programmet och ger anpassad utdata (Barroca & Henriques, 1998). IBDn är däremot inte ett realtidssystem, eftersom systemet inte har strikta regler på hur lång tid som får gå innan utdata måste produceras (Ben-Air, 2006).

3 Problemformulering

Detta kapitel fokuserar på problemet samt utvärderingsmetoden. Problem kapitlet beskriver problemet och detta delas sedan in i två delmål. Metod kapitlet beskriver metoder för att utvärdera att problemet är uppfyllt.

3.1 Problem

Det finns idag problem med IBDns kommunikation och dataöverföring som begränsar programvaran. Mycket upprepningar och inkonsekvens i systemet, statiskt och alla moduler får vetenskap om alla händelser och får sedan välja vad de vill agera på.

IBDn har hög coupling mellan modulerna, coupling är ett mått på hur mycket komponenterna i ett program känner till och kommunicerar med varandra (Bell, 2000). Hög coupling anses vara en dålig egenskap i ett system eftersom det ger moduler som är svåra att återanvända och förändringar i en modul kräver ofta förändringar i andra moduler (Pressman, 2000).

En annan nackdel är att det inte är effektivt att skicka alla händelser till alla moduler och när det blir mycket händelser i systemet kommer detta att bli en flaskhals. Varje modul bör enbart få en delmängd av alla händelserna. En tredje nackdel är att dataöverföring inte kan ske mellan alla modulerna vilket gör att många moduler är uppbyggda av liknande moment men utan möjlighet att föra vidare data. Detta medför att många moduler upprepar samma eller liknande beräkningar. Detta strider mot regeln ”Don’t repeate Yourself” (Hunt & Thomas, 1999) vilken säger att koden aldrig skall upprepas på olika ställen i programmet, eftersom förändringar bara ska behöva göras på ett ställe. Detta gör även att dubbelberäkningar undviks och programmets flexibilitet ökar.

Utifrån dessa utgångspunkter har ett antal kriterier på kommunikationen och dataöverföringen satts upp. En bra arkitektur bör uppfylla följande minimikriterier:

1 Kommunikationen

- 1.1 Antalet meddelandeöverföringar i form av händelser skall minskas i IBDn
- 1.2 Modulerna ska få färre händelser som de inte är intresserade av

2 Dataöverföring

- 2.1 Dynamisk sammankoppling av moduler för flexibilitet: Kopplingarna mellan modulerna skall byggas upp utifrån i ett gränssnitt av användaren och modulen vet inte vid skapandet vem den ska få data ifrån.
- 2.2 Centralisering av funktionalitet för att minska komplexitet: funktionalitet som används likadant i den äldre arkitekturen centraliseras till en plats för att undvika dubbelberäkningar, inkonsekvens samt ge lägre komplexitet

Det finns även gömda krav i systemet som är direkt anknutna till systemets uppbyggnad så som att det skall vara parallellt, anpassat för storskalighet samt vara reaktivt. Dessa är dock ej mätbara i samma mån och är enbart sekundära mål i detta arbete, dock väldigt viktiga att vara medveten om. Det är även viktigt att förklara några uttryck ifrån kriterierna: Dynamisk sammankoppling av moduler innebär att kopplingen inte skall vara statisk och oföränderlig utan vid start skall användaren välja koppling. Det skall även vara möjligt att förändra kopplingen under exekveringen. Undvika inkonsekvens betyder att systemet skall vara konsekvent och om en del av koden förändras skall detta inte behövas göras på andra ställen i koden.

Problemet som detta arbete ska fokusera på är att hitta en kombination av designmönster som uppfyller de kriterier som satts upp för att motverkar de brister IBDn har idag. Designmönster och arkitektur är förklarat tydligare i kapitel 2.1, systemets grundegenskaper är beskrivna i kapitel 2.2.2.

3.1.1 Delmål 1 – Designmönster och designval

För att utforma en bra arkitektur utifrån systemets kriterier behövs information om olika arkitekturs byggstenar: designmönster och designval, det är dessa som bygger upp arkitekturen. Det finns väldigt många designmönster och en sammanfattning av relevanta designmönster ska genomföras, detta för att det ger en övergripande bild av vilka mönster som kan användas.

Designmönstren bedöms efter de problem de löser och jämförs mot kriterierna för att visa på styrkor och svagheter. Designmönster kan vara relevanta för studien av flera olika anledningar men först och främst sorteras de som är centrala för utvecklingen ut och de som har egenskaper som passar samman med systemets egenskaper. Det kan t.ex vara att det är kommunikation som är anpassat för storskalighet eller en sammankopplande mönster som minskar coupling i systemet. Dock skall även andra designmönster, som är mindre anpassade för systemet, också undersökas för att eventuellt kunna använda dess styrkor. Olika designval undersöks även och väljs ut utifrån systemets kriterier tillsammans med valda designmönstren. Det kan t.ex. vara valet om överföringen skall vara synkron eller asynkron.

3.1.2 Delmål 2 - Prototyp

Det går inte att lösa IBDns kommunikations och dataöverförings problem genom att enbart använda sig av ett designmönster utan det kommer att behövas en kombination av flera. Utifrån sammanställningen av designmönster i delmål 1 väljs därför de mest lämpliga designmönstren samt designvalen ut för att kombineras ihop till arkitekturer (MacDonald et al., 2002). Designmönster kombineras samman enligt de problem de skall lösa så att alla kriterier täcks av minst ett designmönster eller designval, detta kommer ge ett par olika arkitekturs förslag som kan utvecklas vidare (Gamma et al., 1994). Arkitekturen kommer att delas upp i två eller flera arkitekturer om behov finns, en uppdelning innebär att arkitekturerna blir mindre komplexa eftersom de täcker in färre kriterier och detta ger möjlighet till nya typer av arkitekturer. Det skapas en eller flera förslag på arkitekturer som sedan utvärderas mot nuvarande arkitektur samt mot varandra och den modell som uppfyller mest kriterier kommer att väljas för vidareutveckling.

Den valda arkitekturen implementeras i Java och använder sig utav IBDns grundmiljö. Detta gör att arkitekturen kan använda sig av verklig data och det skapar en mer realistisk utvärdering av prototypen. Prototypen skall analyseras och verifieras för att visa att den uppfyller kriterierna beskrivna i kapitel 3.1 samt för att visa på styrkor och svagheter med arkitekturen.

3.2 Utvärderingsmetod

När prototypen är klar skall den verifieras mot kriterierna för att se att de är uppfyllda samt att systemet fungerar som det är tänkt. Det finns några olika metoder för utvärderingen (Bell, 2000; Pressman, 2000; Timothy & Yair, 2009), här nedan följer två av dessa metoder som används i detta arbete:

- Experimentellt – Utföra specifika tester på prototypen för att mäta skillnaden mellan den nya och den äldre arkitekturen.
- Analytiskt – Beräkna och analysera arkitekturens förmågor i olika avseenden mellan den nya och den äldre arkitekturen.

Det är dessa två metoder som kommer att använda för att verifiera att arkitekturen uppfyller kriterierna ifrån kravspecifikationen ifrån kapitel 3.1 Varje krav verifieras var för sig med ett eller flera tester och de flesta av metoderna jämför den nya och den äldre arkitekturen med varandra för att se skillnad. Här följer kraven samt metoderna som används för att verifiera dem:

- *Antalet meddelandeöverföringar i form av händelser skall minskas i IBDn*

Kravet är väldigt självförklarande: antalet meddelanden som skickas över systemet skall räknas och att det skall vara mindre i den nya arkitekturen jämfört med den äldre. Detta genomförs *experimentellt* eftersom prototypen kan få verklig data i nutid eller ifrån en databas vilket ger mer korrekt och realistiskt resultat jämfört med att analysera olika scenarion. Resultatet analyseras sedan och mer generella formler skapas för arkitekturens meddelandeöverföring. För att ge mer information om i vilka fall som systemet har förbättrats skall ett antal olika test genomföras med varierande mängd objekt som analyseras samt varierande mängd moduler som är igång. Detta kommer visa när meddelandeöverföringen är optimal: med många eller med få objekt och moduler samt vilken förändring det är jämfört med den äldre arkitekturen. Detta ger oss även en insikt i vilket sorts system arkitekturen är optimalt till.

- *Modulerna ska få färre händelser som de inte är intresserade av*

Detta krav är väldigt likt föregående men här behandlas antalet relevanta meddelanden. Vad som räknas med relevanta avgör varje modul men alla meddelanden som en modul tar emot och som sedan inte behandlas utan enbart kastas är inte relevanta meddelanden. Detta undersöks *experimentellt* genom att räkna de meddelanden som når modulerna och vilka som är relevanta. Detta sker experimentellt eftersom det är enkelt att se om meddelandena används eller inte. Detta säger inte allt om systemet och därför *analyseras* även resultatet och detta är för att det är svårt att utföra experiment som visar just de relevanta händelserna. Båda teknikerna används därmed och dessa komplimenterar varandra och detta visar på styrkor och svagheter med arkitekturen.

- *Dynamisk sammankoppling av moduler för flexibilitet: Kopplingarna mellan modulerna skall byggas upp utifrån i ett gränssnitt av användaren och modulen vet inte vid skapandet vem den ska få data ifrån.*

Detta krav betyder att ingen koppling mellan två moduler är statisk utan kan ändras under exekveringen samt att varje modul kan kopplas samman med flera andra moduler av användaren genom minimal förändring i koden. Detta genomförs *experimentellt* genom ett enkelt test där ett antal moduler och deras kopplingar testas med olika kopplingstyper för att se om de kan föra över data. Fungerar det för en

kopplingstyp antas systemet fungera för övriga kopplingar av samma typ enligt ekvivalens klass testning (Ben-Air, 2006).

Den andra delen av verifieringen ska visa ansträngningen det krävs för användaren att skapa en koppling mellan två moduler har minskat. Detta krav mäts *analytiskt* genom att mäta hur mycket förändring i gränssnittet det behövs för att skapa en koppling mellan olika kombinationer av moduler. Det går inte att jämföra förbättringen mellan den nya och den äldre arkitekturen eftersom där krävs det individuell kodning för varje sammankoppling i den äldre. Om det går att skapa en koppling från gränssnitt är kravet uppfyllt och kopplingen flexibel.

- *Centralisering av funktionalitet för att minska komplexitet: funktionalitet som används likadant i den äldre arkitekturen centraliseras till en plats för att minimera dubbelberäkningar, inkonsekvens samt ge bättre tidskomplexitet.*

Centraliseringen innebär att det skall undvikas att liknande funktionalitet finns i flera olika moduler, dessa funktioner skall brytas ut och skapa egna moduler som kommunicerar vidare sitt resultat. Detta innebär inte att alla likadana funktioner skall brytas ut utan enbart de större funktionerna som gör mer avancerade beräkningar samt har ett eget syfte. Utvecklaren kan själv bedöma vad som skall delas upp men bedömningen under detta arbete är att om funktionen har en helt annan uppgift än modulens grundproblem skall den brytas ut. Om det t.ex finns en modul som läser in alla objekt och tar ut objekten som inte skickat uppdatering på x sekunder och sedan beräknar dess nuvarande position kan det delas upp till två olika moduler med två olika uppgifter.

Detta krav verifieras *analytiskt* genom att ta ut en testgrupp av ett antal olika moduler och metodiskt gå igenom dess kod och anteckna den funktionaliteten som återupprepas i den nya jämfört med den äldre arkitekturen. Det viktiga är inte om hur många rader kod som återupprepas utan hur många liknande funktionalitet som återupprepas. Upprepningen i koden och under exekveringen i trådarna skall *analyseras* och även tidskoplexiteten skall beräknas med hjälp av Ordo (Weiss, 2006). Ordo är ett begrepp som mäter hur tung och komplex en term är och säger hur många steg som i värsta fall behövs för att nå resultatet, vilket är ett sätt att beskriva hur effektiv en algoritm är. Tidskoplexiteten beräknas för varje modul från testgruppen och jämförs sedan mellan den nya och den äldre arkitekturen, detta för att visa att komplexiteten har minskat vid en centralisering och uppdelning av funktionaliteten. Detta analyseras eftersom det är svårt att visa med experiment och tester att komplexiteten har minskat i systemet.

4 Genomförande

Detta kapitel beskriver hur arkitekturen och prototypen har realiserats samt designval. Detta är uppdelat i de två kategorierna: arkitektur samt prototyp med underrubrikerna händelsehantering och datahantering. Dessa förklarar tillsammans det genomförda arbetet.

4.1 Prototyp

För att lösa kommunikations- och dataöverföringsproblemet formas det en arkitektur som sedan skall realiseras i en applikation och även utvärderas. Arkitekturen som skall skapas ska ha två grundegenskaper: att dynamiskt skicka data mellan olika moduler i systemet och att hantera händelser så att de enbart kommer till de moduler som är intresserade av dem. Det valdes att skapa två olika arkitekturer, en för händelser och en för dataöverföring som sedan skall arbeta tillsammans. Detta valdes eftersom det fanns stora skillnader i vilka designval och designmönster som är optimala för de olika arkitekturerna. För att ta fram en arkitektur behövs det först kunskap om designmönstren samt designvalen. Utifrån designmönstren och designvalen utformades en arkitektur som bäst uppfyllde alla kriterier i kapitel 3.1, samt hade bäst potential att användas till IBDns system. Här sammanfattas vad som valdes samt varför och varför olika alternativ valdes bort.

4.1.1 Designval

Här följer en summering av de designval som undersöktes samt vilka som är bäst lämpade till den nya arkitekturen. Designvalen relaterar till hur designmönstren skall användas, och används tillsammans för att nå bästa resultat. Designvalen förklaras även tillsammans med arkitekturerna i nästa kapitel.

I Figur 3 sammanställs de designval som undersökts, valet av designval samt skillnaderna mellan de två olika arkitekturerna. Det är designval 1 och 2 som skiljer arkitekturerna åt i tabellen, dock realiseras även övriga designval något olika mellan arkitekturerna, bl.a. p.g.a. detta. Nedanför förklaras vad designvalen innebär samt varför det specifika designvalet valdes.

Designval:	**Händelser:**	**Data:**
1) Skicka eller Hämta objekt	Skicka	Hämta
2) Hanterare eller Självorganisering	Hanterare	Självorganisering + delvis Hanterare
3) Spara objekt i Modul eller Databas	Modul	Modul
4) Uppdatera eller Byt ut	Byt ut	Byt ut

Figur 3 Tabell som visar Arkitekturens designval och dess val.

Designval 1: Skicka eller hämta objekt (push vs pull)
Detta avser vem som har ansvar för att föra vidare samt spara objektet.
Skicka: Modulen som skapar en händelse/data skickar vidare det och de lyssnande modulerna sparar det.
Hämta: Modulen som skapar en händelse/data sparar det och de lyssnande modulerna hämtar det när de är i behov av det.

För händelsehanteringen valdes att skicka objekten till de lyssnande modulerna. Detta valdes eftersom händelserna då sparas i de moduler som är intresserade av dem, vilket

gör att varje modul vet att den har fått alla händelser som är skickade fram till kör tillfället. Detta är att föredra eftersom alla händelser skall behandlas.
För Datahanteringen valdes att hämta objekten från skaparen. Detta valdes eftersom varje modul enbart vill ha det senaste tillgängliga objektet och är inte intresserade av att veta allt som skett sedan sista uppdateringen. Genom att modulerna hämtar objektet enbart när de vill ha det sker ingen onödig dataöverföring vilket skulle ha skett i annat fall.

Designval 2: Hanterare eller självorganisering
En hanterare organiserar kommunikationen mellan alla moduler, alla modulerna känner enbart till hanteraren. Självorganisering innebär att varje modul har vetskap om de moduler den skall kommunicera med, ingen övergripande modul som gör detta. Detta är tätt sammanknutet med designmönstret Mediator (Gamma et al., 1994) vilket förklaras närmare i kapitel 4.1.2, men en hanterare kan även implementeras med andra designmönster.

För händelsehanteringen valdes att använda en hanterare för organiseringen. Detta valdes för att alla moduler skall vara sammankopplade utan att vara beroende av varandra, hanteraren får in en händelse, bearbetar den och skickar den till de moduler som skall ha den. Alla moduler som vill skicka en händelse skickar det till hanteraren som sedan tar hand om allt därifrån. Alla som vill ha ett specifikt händelse säger till hanteraren vilka händelser den vill ha och får sedan enbart dessa. Detta minskar coupling eftersom alla enbart känner till hanteraren och inte varandra. Detta gör även varje modul mindre komplex vilket är att föredra.
För datahanteringen valdes självorganisering, dock behövs även en hanterare för att dynamiskt ta hand om tillkomst och förändringar i överföringen. Detta valdes eftersom kommunikationen skall vara dynamisk sammankopplad och när kopplingen är etablerad förändras den enbart sällan vilket gör att en statisk koppling mellan modulerna är att föredra eftersom det är onödigt att gå genom en hanterare med mycket data när det är främst statiska kopplingar efter upp starten. Hanteraren kopplar samman moduler samt varnar om koppling bryts, i övrigt kommunicerar de sammankoppla modulerna med varandra.

Designval 3: Spara i modul eller databas
De objekt som andra moduler ska få tillgång till sparas antingen i en databas i en central modul eller i varje modul.

För händelsehanteringen valdes att spara händelserna i varje modul som är intresserad av objektet, detta eftersom tillsammans med designval 1 gör det att varje modul får alla händelser utan att själva aktivt fråga eller behandla dem. För Datahanteringen valdes också att spara alla objekt i modulen som skapade objektet, detta eftersom det skulle blir en mycket stor tabell i en databas vilket inte är optimalt. Samt att det är onödigt att belasta en databas med all dataöverföring när systemet har relativt statiska kopplingar under en normal exekvering efter uppstarten.

Designval 4: Uppdatera eller byt ut
Uppdatera modulernas objektkopia med ny data som tillkommit sedan förra uppdateringen eller hämta om hela objektet varje gång.

För händelsehanteringen var detta inte ett val eftersom varje händelse är unikt och ingen uppdatering är därför möjlig. För Datahanteringen valdes att hämta om objekt, detta eftersom det är mycket som förändras och därför kommer det kosta mindre att hämta om objektet än att beräkna skillnaden och skicka över enbart den datan.

4.1.2 Händelsehantering

I den äldre arkitekturen filtreras inte händelser utan varje modul hämtar en lista med alla moduler som lyssnar på händelser och skickar händelsen till alla dem. Varje modul får var för sig sedan behandla händelserna och se om de är viktiga eller inte. Detta skall förändras så att varje modul enbart får de specifika händelser som den är intresserad av och inte de händelser som inte angår dem. Alla moduler kan skapa flera olika händelser.

För att uppfylla kriterierna måste alla händelser filtreras för att se vilka den skall till, för att detta skall ske smidigt och utan dubbelberäkningar eller dubbel kod sker det i en extern modul, en hanterare. Hanteraren implementeras med Mediator Designmönster (Gamma et al., 1994) vilket är en hanterar för kommunikationen mellan grupper av objekt och alla objekt kommunicerar med mediatorn istället för med varandra. Hanteraren används för att undvika att varje modul skall behöva implementera en filtrerings funktion samt skapar hanteraren abstrakta kopplingar mellan modulerna vilket minskar coupling i systemet. Hanteraren används också för att undvika att tillkomst av moduler ej registreras i tid, att avbrutna moduler fortsätter användas osv.

Hanteraren implementeras med singelton designmönster (Gamma et al., 1994) vilken försäkrar att en klass enbart har en instans och ger en global åtkomst till klassen. Detta är ett mycket bra designmönster som är bra att använda på de moduler som det enbart skall finnas en kopia av i projektet. Det finns inga större problem med designmönstret förutom att det enbart skall användas på de specifika klasser som skall vara globala och unika. Varje modul som vill ha vetskap om en eller flera specifika händelser skickar en förfrågan till hanteraren om att kopplas samman med dessa händelser. Varje modul skapar sedan sina händelser och skickar dem till hanteraren som filtrerar dem och skickar dem enbart vidare till de moduler som förfrågat om information ifrån dem. Händelsen skickas till de intresserade modulerna och sparas lokalt i dem så att en global lista undviks.

Händelser implementeras med Immutable designmönster (Haggar, 1999) vilket innebär att objekt är helt oföränderliga. De går inte att skriva till objekt efter skapandet utan enbart läsa från dem. Genom att göra alla händelser oföränderliga är de även trådsäkra. Detta är väldigt bra designmönster till den nya arkitekturen eftersom en händelse ej skall ändras. Problemen med designmönstret är att det snabbt blir en stor mängd objekt eftersom man måste skapa ett nytt varje gång man vill förändra något. Java tar automatiskt bort objekt utan referenser och därmed är det inte ett större problem i detta fallet.

Hanteraren har ansvaret för händelsehanteringen och använder sig av Publisher/Subscriber designmönster (Schmidt & O'Ryan, 2003) för att filtrera händelserna. Varje modul som skapar ett objekt håller en lista med alla andra moduler som är intresserade av detta objekt och meddelar dem när objektet förändras. De intresserade modulerna får då hämta det uppdaterade objektet. Genom att varje modul säger vilken händelse de vill ha info från sorterar hanteraren dessa i en observeringslista. När en händelse kommer in till hanteraren undersöks det om händelsen matchar en händelse i observeringslista och gör den det skickas händelsen till alla moduler som är sammankopplade med den händelsen i listan.

För att en modul skall kunna få information från flera olika händelser utan att behöva skicka väldigt många förfrågningar till hanteraren valdes det att skapa en abstrakt händelse som består av logiska uttryck (Di Stefano, 2007). Detta minimerar även

filtreringen och mängden abstrakta händelser. Den abstrakta händelsen har samma variabler som en vanlig händelse plus en extra som säger hur dessa variabler får kopplas samman. Den abstrakta händelsen innehåller även logiska uttryck med de logiska operationerna: and, or, not och * vilka kopplar samman uttrycket, * innebär att allt är tillåtet. Eftersom det finns två olika typer av händelser, en abstrakt och en konkret behöver match() funktionen översätta dem så att det går att jämföra detta visas här nedan i ett exempel.

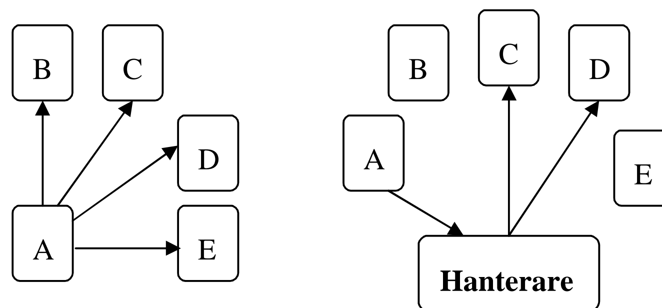
Det finns två olika händelser, det abstrakta är ett BDSubscribeEvent och det konkreta är ett BDEvent. Det abstrakta händelsen innehåller logik och det match funktionen gör är att se om dessa två matchar varandra.

BDSubscribeEvent()	BDEvent()
CreatorName = "not Speed33"	CreatorName = "Speed1"
ClassName = "Speed or Direction"	ClassName = "Speed"
EventType = "BDSystem"	EventType = "BDalarm"
Description = "*"	Description = "Speed: 33 m/s"
EventName = "*"	EventName = "Speed_To_High"
Connection = " (ClassName or eventType) and CreatorName"	

Om dessa två händelser skulle skickas in till match() funktionen skulle varje variable för sig beräknas för att ge ett sant eller falskt resultat. CreatorName får inte vara Speed33, vilket det inte är och det blir då sant. ClassName skall vara Speed eller Direction, händelsen är Speed och blir därför sann. EventType ska vara BDSystem vilket blir falskt. Description får vara vad som helst och lika så EventName. Connection säger nu att för att hela uttrycket skall vara sant måste dem uppfyllas. Klassnamn eller EventType skall vara sant samt CreatorName.

Match() funktionen måste vara effektiv för att överväga kraften det går åt till att skicka meddelanden till alla, dock är inte effektiviteten det enda viktiga eftersom en viktig del med omorganisationen är att antalet skickade meddelande skall minska.

En jämförelse mellan den nya och den äldre arkitekturen visas i Figur 4. I den äldre arkitekturen skickas en händelse till alla moduler, oavsett om de är intresserade av det eller inte. I den nya arkitekturen skickas det genom en hanterare och enbart till de intresserade modulerna.



Figur 4 T.v. är den äldre arkitekturen, A skickar händelser till alla
T.h. är den nya arkitekturen, med en hanterare som skickar händelsen till rätt moduler.

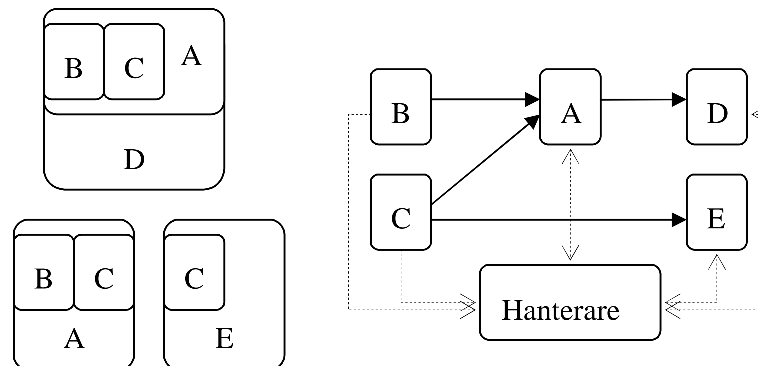
4.1.3 Datahantering

I den äldre arkitekturen går det inte att skicka data mellan modulerna utan istället består en modul av alla de delar som behövs för att lösa problemet. Detta innebär att det blir mycket inkonsekvent kod. Programvaran skall förbättras genom att skapa en dynamisk arkitektur som skickar data mellan modulerna.

Datahanteringen implementeras med pipeline designmönster (Buschman et al., 1996). Detta används för att koppla samman modulerna till en kedja där den förstes utdata är nästas indata. Dessa vägar kan även dela sig så två olika moduler kan ta den förstes utdata. Genom förgreningar undviks upprepande kod, designmönstret ger även en tydlig körordning för modulerna. Genom att använda pipeline skapas en statisk kedja av moduler där första modulen uppdaterar sitt utdataobjekt och övriga moduler får sedan hämta datan från den. Det stora designalternativet var mellan pipline och Blackboard designmönster (Corkill, 2003; Buschman et al., 1996) vilken använder sig av en databas som alla moduler i iterationer förändrar i. Det brukar vara ett bra designmönster för parallella system dock valdes detta bort eftersom det behövs en fördefinierad ordning på modulerna samt att det inte helt uppfyller kraven på att systemet inte skall ha återupprepad kod eller beräkningar.

Det största problemet med pipline är att kedjan måste specificeras vilket gör att den blir mycket statisk. För att datakopplingarna skall dynamiskt gå att sätta samman och förändra och för att detta skall vara möjligt behövs en hanterare, här används samma hanterare som för händelsehanteringen. Hanteraren kopplar samman två moduler och låter sedan dem sköta dataöverföringen själva. Detta sker eftersom dataöverföringen är väldigt statisk efter uppkopplingen och endast i få fall förändras kopplingarna under exekveringen och då är det onödigt att belasta en hanterare med överföringarna som annars kan bli en flaskhals. Det är enkelt att bygga samman olika moduler och skicka informationen utan att dubbelberäkningar genomförs med den nya arkitekturen. Den har även dynamisk sammankoppling som kan ändras under exekveringen genom att skicka en ny förfrågan till hanteraren om koppling till en annan modul.

En jämförelse mellan den nya och den äldre arkitekturen visas i Figur 5. I den äldre arkitekturen består varje modul av flera andra moduler och detta ger upprepningar i beräkningar och i koden. I den nya arkitektur skickas en uppkoppling till hanteraren vid start av varje modul som sedan ger tillbaka en koppling till önskade moduler (streckade linjer). Data hämtas sedan från föregående modul i ledet och leder framåt till ett eller flera olika beräkningar (vanliga linjer). Till exempel i Figur 5 startas modul B först och skickar sin koppling till Hanteraren. När modul A startas frågar den Hanteraren om kopplingsmöjligheter och får då en koppling till modul B.



Figur 5 T.v. är start strukturen, mycket upprepning av kod
T.h. är mitt förslag, med ett dataflöde mellan modulerna

4.2 Prototyp

För att ha möjlighet att utvärdera den nya arkitekturen samt hitta svagheter implementeras arkitekturen i Java som en vidareutveckling av den äldre arkitekturen. Det finns många centrala delar i koden som främst finns implementerade i hanteraren och här nedan följer några av dessa för djupare förståelse.

4.2.1 Händelsehantering

Händelsehanteringen har några centrala funktioner:

- SubscribeList vilket är en privat lista med alla abstrakta händelser och vilka moduler som är intresserade av dessa abstrakta händelser.
- SubscribeToEvent() vilket är en funktion som lägger till moduler i SubscribeList tillsammans med korrekt abstrakt händelse
- DispatchEvent() vilken används för att vidarebefordra en händelse till dem som är intresserade. Den går igenom alla händelser i SubscribeList och jämför med den konkreta händelsen i en match() funktion. Är händelserna lika skickas händelsen vidare till dem.
- Match() vilket är en privat funktion som tar en abstrakt och en konkret händelse och beräknar om de är lika. Detta sker med logik genom att varje abstrakt händelse delas upp och jämförs med den konkreta. Detta löstes med hjälp av en parser som förde över ett logiskt uttryck till en sann/falsk variable.

För att match funktionen skall klara av mer än enbart att jämföra med ett värde implementerades en parser som först tar den abstrakta händelsens uttryck och jämför med det konkreta för att skapa en sann/falsk sträng. För att visa på ett exempel kan vi återanvända tidigare exempel:

```
BDSubscribeEvent()
  CreatorName = "not Speed33"
  ClassName = "Speed or Direction"
  EventType = "BDSystem"
  Connection = " (ClassName or eventType) and CreatorName"

BDEvent()
  CreatorName = "Speed1"
  ClassName = "Speed"
  EventType = "BDalarm"
```

När dessa skall matchas börjar parsern att läsa in ”not speed” och eftersom not, and, or samt * är specialtecken behandlas inte dessa utan används som de är. Då jämförs Speed33 med Speed1 och detta uttryck blir falskt och resultatet från första iterationen blir ”not false”. Match() använder sedan en logikparser för att logiken skall bli korrekt vilken t.ex. säger att not framför false blir true och logikparsern beräknar först alla uttryck innanför parenteser osv. Detta ger oss en boolesk variabel som resultat, i vårt fall blir det True.

4.2.2 Datahantering

Datahanteringen sker genom klassen BDConnection vilken den sparar en modul referens, ett variabel nr och en typ av data som skall skickas. BDConnection har även en central funktion som är getData(), vilken hämtar en kopia av det senaste sparade objektet hos modulen den har en referens till. Funktionen är även synkroniserad för att den skall bli trådsäker.

```
public Class BDConnection {
        private Module module;
        private int variabel;
        private type ObjectType;
        public Object getData(){
            return module.getOutData(variabel, ObjectType);
        }
```

Genom att modulen har den här kopplingen är det väldigt enkelt att hämta data från en modul till en annan. För att skapa en BDConnection behöver däremot hanteraren involveras. Hanteraren har vetenskap om vilka moduler som är skapade och dessas utdata vilket är sparad i en lista.

Modulen som vill ha data skickar en förfrågan till hanteraren med modulens namn och variable nummer som den vill ha data från, vilka alternativ som är möjliga implementeras i ett grafiskt gränssnitt. Hanteraren kontrollerar namnet mot sin lista och skickar sedan tillbaka en BDConnection med referens till modulen. Den intresserade modulen kan sedan med hjälp av BDConnection anropa getData() för att hämta data från den modulen. Detta gör arkitekturen mer effektiv eftersom modulen inte vill veta om alla uppdateringar utan bara den sist uppdaterade objektet.

5 Resultatanalys

Det producerade systemet skall utvärderas för ett finna dess styrkor och svagheter samt verifiera att systemet uppfyller kraven satta i kapitel 3.1. Här nedan följer mina resultat och analys av varje krav samt en diskussion kring dem. Kapitlet är uppdelat efter varje krav som ska uppfyllas.

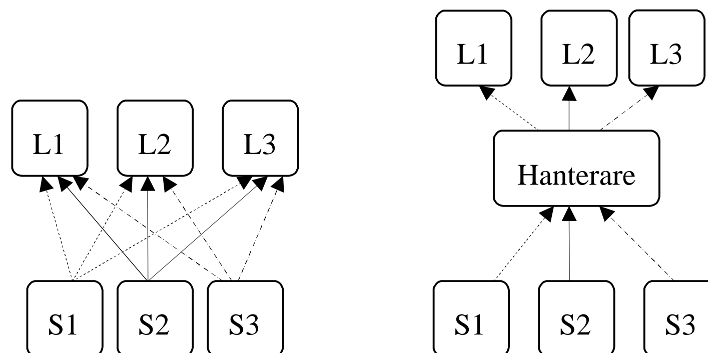
5.1 Händelsehantering – Antal meddelanden

Kravet som skall utvärderas är: *Antalet meddelandeöverföringar i form av händelser skall minskas i IBDn.*

Först sätts miljön upp och det finns två olika typer av moduler: Lyssnande moduler och Skapande moduler, detta visar vilken roll modulen har i arkitekturen. De skapande modulerna skapar händelser och skickas vidare dessa, de lyssnande modulerna tar emot händelser och bearbetar dem. En modul kan vara både lyssnande och skapande men i mina exempel är de antingen det ena eller det andra. Varje modul i mina kommande exempel skickar enbart en händelse var, vilket är våra så kallade ”unika händelser”, detta innebär att antalet unika händelser är lika många som antalet skapande moduler. Unika händelser används för att mäta antalet händelser när system blir större och en modul kan skapa flera unika händelser osv. Det utförs i både den äldre och den nya arkitekturen ett par meddelandeöverföringar för att koppla samman modulerna, dessa räknas det inte på eftersom de är ett specialfall och uppkommer enbart en gång under en exekvering.

5.1.1 Scenario

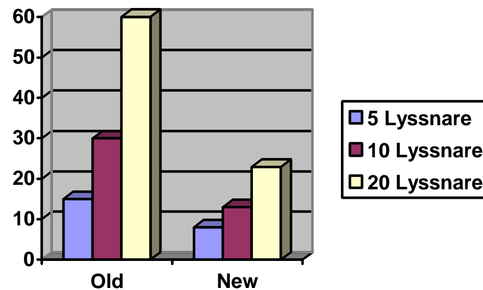
För att visa på ett realistiskt fall av hur händelsehanteringen kommer att användas undersöks tre skickande moduler: S1 skickar alarm när en hastighet överskrids, S2 skickar alarm när en riktning är felaktig och S3 skickar alarm när två objekt följs åt. Det finns även tre lyssnande moduler som beräknar olika saker på de olika händelserna. L1 tar in alla hastighetsalarm och visar upp dessa båtar i ett grafiskt gränssnitt, L2 tar in riktningsalarm och ser om dessa båtar är i de zonerna som man inte får ha denna riktning i och L3 tar in alla objekt som följer varandra och räknar ut avståndet mellan dem. Detta visas i Figur 6.



Figur 6 Här visas de olika modulernas skickade händleser t.v. är den äldre arkitekturen och t.h. är den nya arkitekturen.

I den äldre arkitekturen skickar varje modul ett meddelande till varje lyssnande modul, detta ger totalt nio skickade meddelanden. I den nya arkitekturen skickar varje modul ett meddelande till hanteraren som sedan för vidare alarmet till den eller de moduler som lyssnar på detta meddelande, detta ger oss enbart sex meddelanden.

Figur 7 visar de tre skickande modulerna med olika antal lyssnande moduler som var och en enbart tar emot en viss händelse.



Figur 7 Här visas skickade händelser med olika antal lyssnande moduler.

Detta scenario visar att antalet meddelandeöverföringar har minskat i systemet vid jämförelse mellan den äldre och den nya arkitekturen och det visar också att stora system vinner mer på den nya arkitekturen än små system med enbart få lyssnande moduler. Resultatet i diagrammet är uträknat ifrån följande formel:

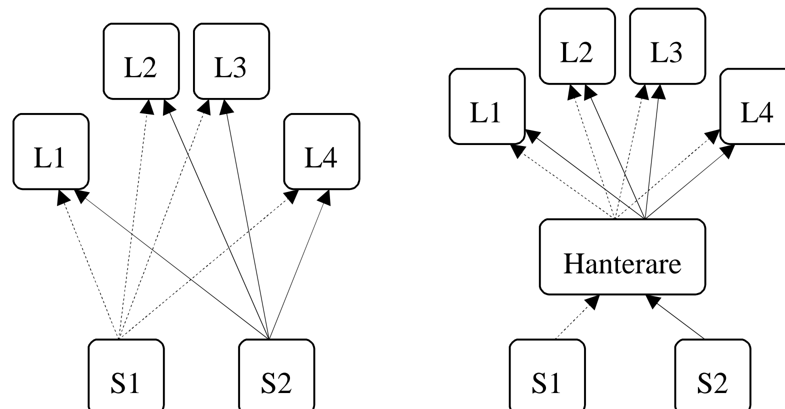
Formel 1: För den äldre arkitekturen är formeln: *Antal Unika Händelser * Antal Lyssnande Moduler*.

Formel 2: För den nya arkitekturen är formeln: *Antal Unika Händelser + Antalet Lyssnande Moduler * antalet händelser de lyssnar på.*

Formlerna kan exemplifieras på Figur 6: den äldre arkitekturen har tre unika händelser och tre lyssnande moduler och det ger oss 3*3 = 9 händelser. Den nya arkitektruen har också tre unika händelser och tre lyssnande moduler men varje modul lyssnar enbart på en händelse vilket ger 3+3*1 = 6 händelser.

5.1.2 Undantag

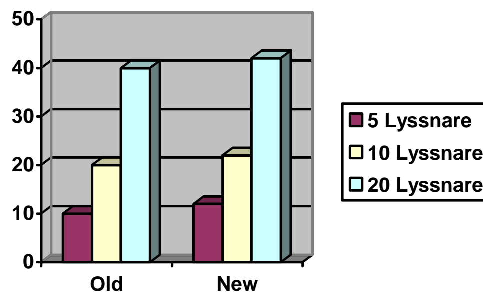
För att se skillnad mot föregående scenario testars fallet där alla moduler vill ha alla händelser och inget urval. Det finns två skapande moduler S1 och S2 som skickar ut händelser till fyra lyssnande moduler: L1, L2, L3 och L4. Detta visas i Figur 8.



Figur 8 Här visas hur händelser skickas.
t.v är den äldre arkitekturen och t.h. är den nya arkitekturen.

I Den äldre arkitekturen skickar varje skapande modul ett meddelande till varje lyssnande modul vilket ger totalt åtta meddelanden. I den nya arkitekturen skickar

varje skapande modul ett meddelande till hanteraren som sedan skickar ett meddelande till varje lyssnande modul vilket ger totalt tio meddelanden. Antalet meddelanden har därmed inte minskat i den nya arkitekturen utan istället ökat. Detta sker eftersom hanteraren först måste få händelserna för att sedan vidarebefordra dem. Försämringen är en konstant på 2 överföringar vilket är 1x Unika händelser. I Figur 9 visas försämringen i ett större system med fler lyssnare. Detta räknas ut enligt formel 1 och 2 i kapitel 5.1.1. Den äldre arkitekturen ger oss för 20 lyssnande moduler 2*20 = 40 och den nya ger oss 2+20*2 = 42 meddelandeöverföringar. Detta visar att skillnaden är en konstant på 2 överföringar även med fler lyssnande moduler vilket är endast en minimal försämring mot den äldre arkitekturen.



Figur 9 Här visas skickade händelser med olika antal lyssnande moduler.

Detta resultat visar att den nya arkitekturen inte i alla fall uppfyller kravet att antalet meddelanden skall minskas utan när alla meddelanden skall vidarebefordras ökar antalet meddelanden. Detta visar också att skillnaden är minimal med ett stort system men mer påtagligt vid ett litet. Resultatet visar att systemet inte lämpar sig för att föra vidare alla händelser utan är sämre på detta än den äldre arkitekturen. Systemet kommer dock inte att användas på detta viset, det är inte meningen att alla moduler skall få alla händelser utan. Detta är inget misstag utan övervägning mellan ny funktionalitet och optimering och till IBDn är den nya arkitekturen optimerad.

5.2 Händelsehantering – Relevanta meddelanden

Kravet som skall utvärderas är: M*odulerna ska få färre händelser som de inte är intresserade av.*

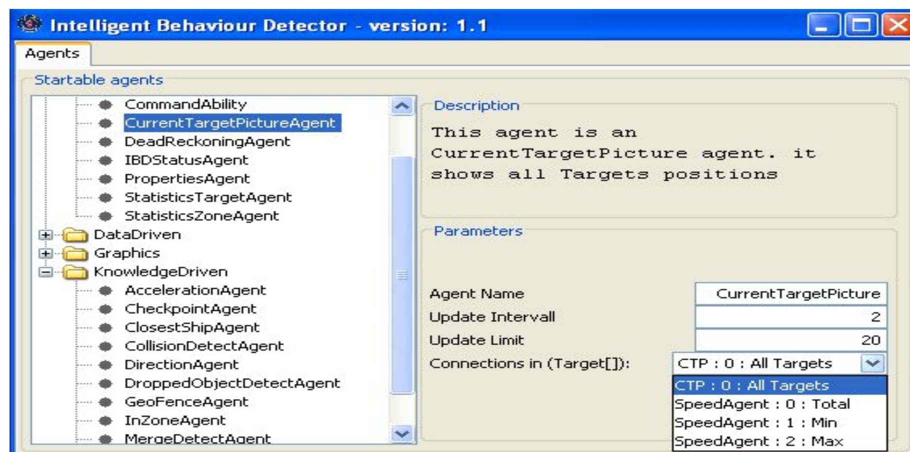
Genom realiseringen av logik parsern kan varje modul själv bestämma vilka händelser den vill få information om och övriga händelser får den ingen information om alls. Detta innebär att kravet inte bara är uppfyllt utan att varje modul enbart får de händelser som den är intresserad av. Systemet har testats för att se att modulerna enbart får de händelser som den prenumererar på, vilket sker korrekt. Det är möjligt att prenumerera på flera händelser samt slå samman händelser till enbart väldigt specifika sådana. Detta gör att antalet oviktiga händelser som modulen får är upp till skaparen av modulen och hur noggrant det eller de abstrakta händelserna som modulen skapar är. Det är möjligt att se till att modulerna enbart får de mest viktiga händelserna och inga oviktiga. Som exempel finns det en ZoneCounter vilken räknar båtar i ett område över en längre tid, den modulen vill enbart få alarm när en båt åker in i just denna specifika zon. Modulen skapar en abstrakt händelse som säger att EventName = EnteringZone och EventDescription innehåller strängen ”Zone33”. Detta ger den enbart de viktiga alarmen o inga oviktiga.

5.3 Datahantering - Dynamisk sammankoppling

Kravet som skall utvärderas är: *Dynamisk sammankoppling av moduler för flexibilitet: Kopplingarna mellan modulerna skall byggas upp utifrån i ett gränssnitt av användaren och modulen vet inte vid skapandet vem den ska få data ifrån.*

Det är svårt att jämföra detta krav eftersom den äldre strukturen inte har möjlighet att skicka data mellan modulerna. Att kopplingarna kan förändras och är dynamiska verifieras genom att starta flera moduler med flera olika listor med Targets som indata. Detta har genomförts systematiskt med alla hittills skapade moduler som vill ha in data genom att koppla samman dem med några olika indata från andra modulerna, resultatet är att det går utan några problem.

När en modul tappar en koppling, vilket sker när modulen den tar data ifrån avslutas går kopplingen över till default. Vad som sker då är individuellt mellan modulerna, vissa beräknar på default target listan som innehåller alla targets medan andra inte gör något förens den får en korrekt koppling igen. Detta tillsammans visar att kopplingen är dynamisk och anpassa efter vad som sker. Det är även möjligt att koppla om en startad modul dock tillåter inte gränssnittet detta i nuläget. Den dynamiska sammankopplingen ger oss stor flexibilitet genom att varje modul själv kan välja vilken data modulen skall arbeta på.



Figur 10 Här visas gränssnittet och rullistan med möjliga kopplingar.

Det som krävs av en modul för att få data från en annan modul är att i gränssnittet ställa in modulens namn samt vilken nr på listan om det finns flera. Eftersom alla startade moduler sparar sin ut data i en lista som alla moduler sedan kan komma åt är det enkelt att få veta vilka alternativ som finns. I gränssnittet ser du därför en rullista med alternativen på varje position där du kan ta in en Target lista. Som exempel startar vi en Hastighetsmodul vid namn SpeedLimit som ger ut tre olika listor med targets: Min, Max och Total. När sedan en ny modul skall startas kan den ta in Alla targets eller en utav dessa listor från gränssnittet, detta visas i Figur 10.

Det som krävdes i den äldre arkitekturen för att föra över data mellan modulerna är en hårdkodad referens till den andra modulen och detta gjorde att kopplingen var statisk, att modulen alltid måste vara aktiv mm. Vi uppfyller härmed kravet på en dynamisk arkitektur genom att vi verifierar att en modul kan ta in data från många olika moduler samt att när en modul i ledet avslutas förändras kopplingen och användargränssnittet väljer hur modulerna skall kopplas samman.

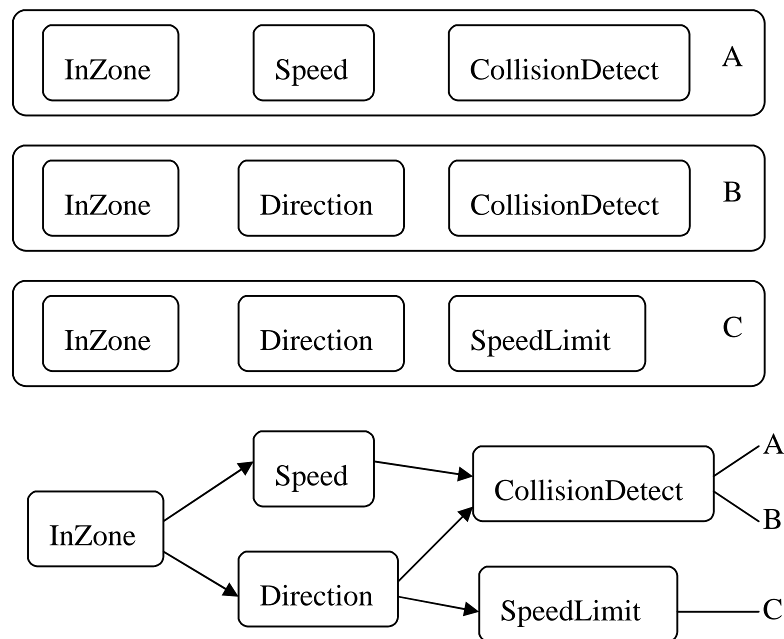
5.4 Datahantering - Centralisering av funktionalitet

Kravet som skall utvärderas är: *Centralisering av funktionalitet för att minska komplexitet: funktionalitet som används likadant i den äldre arkitekturen centraliseras till en plats för att minimera dubbelberäkningar, inkonsekvens samt ge bättre tidskomplexitet*

Detta krav utvärderas analytiskt genom att jämföra den äldre och den nya arkitekturen. Det är svårt att analysera alla alternativ av olika moduler och därför väljs att enbart analysera en delmängd av alla moduler.

5.4.1 Koden

Här nedan kommer ett exempel på tre moduler som finns i den äldre arkitekturen och som jämförs med hur de ser ut i den nya arkitekturen. Det som undersöks här är koden och upprepningar i våra klasser:



Figur 11 Skillnad i antal moduler mellan nya och äldre arkitekturen.
De tre översta tillhör äldre arkitekturen och de fem sammanlänkade tillhör den nya.

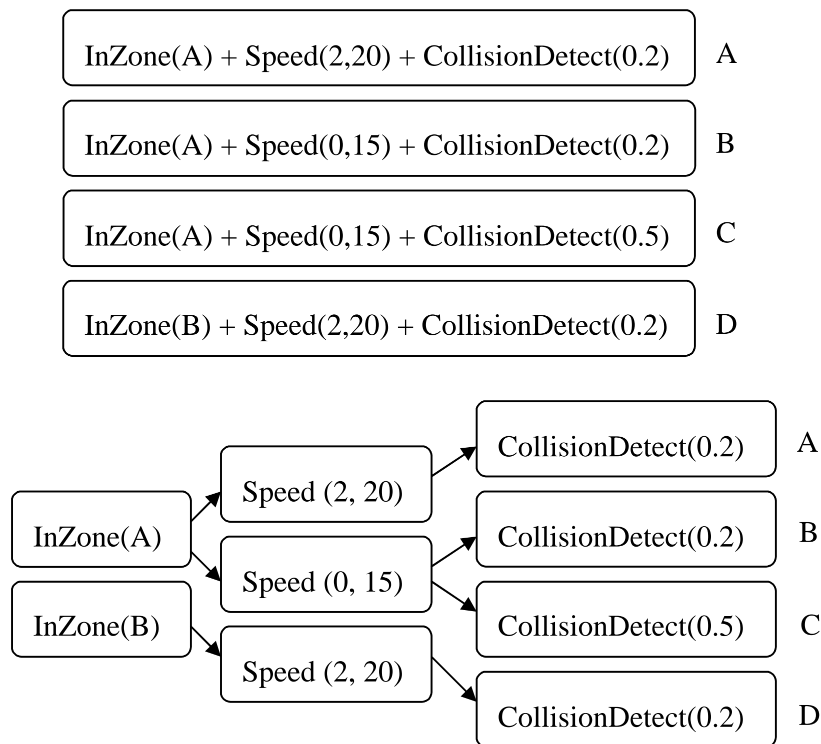
Figur 11 visar tre moduler A, B & C. I den äldre arkitekturen består dessa av olika funktionalitet i dem vilket är delvis upprepande. De tar alla ut data från en zon, beräknar sedan hastighet eller riktning på datan och sedan utförs kollisionsdetektering eller en hastighetsberäkning. Det är väldigt mycket lika funktionalitet i dessa moduler och InZone funktionaliteten ligger på tre olika ställen i olika moduler, Direction i två och CollisionDetect i två. Det är inkonsekvens i systemet vilket visas genom att om vi skall t.ex optimera CollisionDetect koden behövs den skrivas om i koden på två ställen osv. Detta gör att samma kod finns dubbletter av på andra ställen i systemet.

Upprepningarna i den äldre arkitekturen i Figur 11 undersöks och resultatet är att det är totalt fyra delar av nio möjliga som innehåller upprepande kod i den äldre arkitekturen. De upprepande funktionerna bryter vi ut till den nya arkitekturen och skapar fem individuella moduler vilka har en specifik egenskap var och dessa moduler skickar sedan data mellan varandra. Detta skapar fem moduler och ingen upprepad kod och ingen inkonsekvent kod utan om du ändrar i Direction modulen vet du att det

alltid är den som exekveras. Detta visar att koden har centraliserats samt att detta tar bort problemet med inkonsekvent kod (Hunt & Thomas, 1999).

5.4.2 Trådar

När programmet exekveras utförs olika trådar på olika processorer och detta kan också ge upprepad kod. Varje modul exekveras på en tråd och det är svårt att jämföra trådarna med varandra eftersom den nya arkitekturen är mycket mindre och fler, därför räknar vi funktionalitet även här. Några olika instanser av modul A från föregående kapitel undersöks vilken består av delarna: InZone, Speed & CollisionDetect. Modulen undersöks med olika indata för att få en stor variation och här nedan följer ett exempel med fyra olika inställningar och hur detta översätts till den nya arkitekturen, Figur 12.



Figur 12 Skillnad i antal trådar skapade mellan nya och äldre arkitekturen.
De fyra översta tillhör äldre arkitekturen och de nio sammanlänkade tillhör den nya.

Det skapas enbart fyra trådar i den äldre arkitekturen men dessa trådar innehåller var och en tre funktionaliteter med olika eller likadana värden. Vi ser enkelt att det är två upprepade InZone(A), en Upprepad Speed(0,15) och en upprepad Speed(2,20) samt två upprepade CollisionDetect(0.2). Detta ger oss sex upprepningar i trådarna av tolv totalt.

Sedan undersöks den nya arkitekturen vilken enbart skapar en InZone för A och en InZone B vilket tar bort två upprepningar. Det skapas tre Speed trådar men två är dubbletter. Dubbletter skall undvikas i möjlig mån men detta är korrekt eftersom de två dubbletterna av Speed(2,20) har olika indata. En instans beräknar speed på Targets ifrån InZone(A) och en räknar speed på targets från InZone(B). Detta gör dem till två olika instanser med olika indata och olika utdata. Dock sparar den nya arkitekturen en dubbelberäkning genom Speed(0,15). Det skapas fyra instanser av CollisionDetect vilken är sist i ledet vilket är lika många som i den äldre arkitekturen. Detta gör att

inga beräkningar sparas här, men det blir heller inte mer upprepningar än i den äldre arkitekturen. I den nya arkitekturen exekveras det 9 funktionaliteter vilket är att jämföra med att det exekveras 12 i den äldre arkitekturen.

Det finns vissa nackdelar med dessa upprepningar av liknande trådar vilket skulle gå att fixa genom att skapa mer komplexa moduler så att t.ex. Speed tar in data från x antal olika Target Listor och ger ut lika många Target Listor. Detta skulle eventuellt kunna optimera vissa beräkningar dock kommer ungefär lika mycket beräkningar att ske oavsett. Därför valdes det att göra flera enklare moduler istället för få komplexa.

5.4.3 Tidskomplexitet

Tidskomplexiteten på olika moduler undersöks för att visa att komplexiteten har minskat (Bell, 2000). Det går att mäta komplexiteten på flera olika sätt (Coskun & Grabowski, 2005) men det jag har fokuserar på är funktionens storlek och tiden det tar att utföra dem vilket beräknas matematiskt med Ordo. Ordo beräknar max antal operationer som krävs för att gå igenom en funktion. Här visas en förenklad pseudokod för de tre olika funktionerna som utgör modulernas grund: InZone, Speed och CollisionDetect. Både InZone och Speed har en tidskomplexitet på O(n) medan CollisionDetect har en tidskomplexitet på $O(n^2)$.

```
InZone:
TargetLista AllaTargets
För alla Target ifrån AllaTargets
    Ifall Target är i Zone
        Spara Target i ny lista.

Speed:
TargetLista AllaTargets
För alla Target ifrån AllaTargets
    Ifall Target har hastighet över x
        Spara Target i ny lista.

CollisionDetect:
TargetLista AllaTargets
För alla Target1 i AllaTargets
    För alla Target2 i AllaTargets
        Ifall Target1 är för nära Target2
            Skicka ett alarm
```

Tidkomplexiteten för modulerna A, B och C i Figur 12 vilken finns i föregående kapitel jämförs. Det valdes att inte jämföra modul D eftersom den ligger utanför dem andra och innehåller inga dubbleringar. Första modulen vilket är InZone tar in n antal objekt och ger ut n_2 objekt, andra modulen i ledet vilket är Speed tar in n_2 objekt och ger ut n_3 objekt och tredje modulen CollisionDetect tar in n_3 objekt och ger ut n_4.

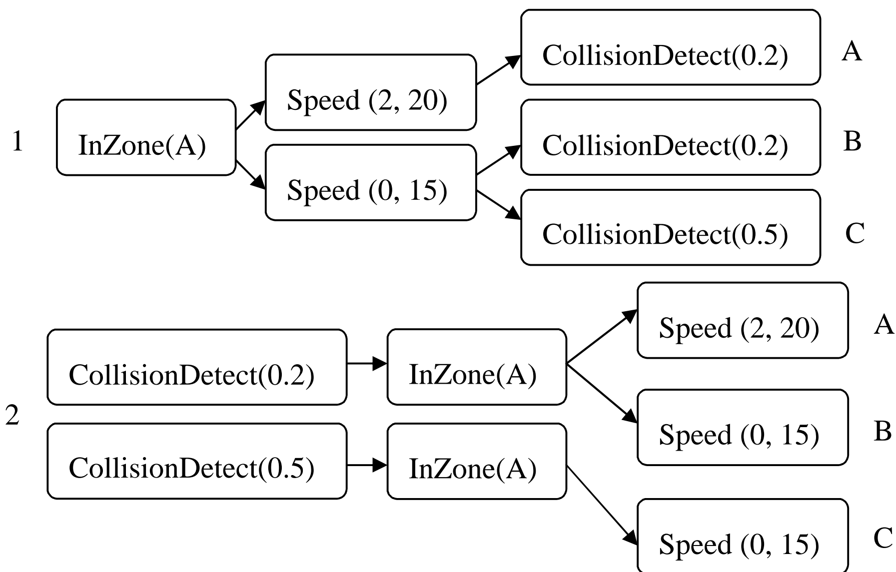
Den äldre arkitekturen får en beräknad tidskomplexitet på $O(3n_1 + 3n_2 + 3n_3^2)$ och den nya arkitekturen har tidskomplexitet på $O(n_1 + 2n_2 + 3n_3^2)$. Detta beräknas genom att ersätta varje funktion av speed, InZone och CollisionDetect med dess Ordo värde gånger antal gånger den exekveras. Den nya arkitekturen får lägre tidskomplexitet eftersom den sparar in ett antal beräkningar som är lika, detta går även att se i Figur 12.

Här följer ett exempel på hur tidskomplexiteten för den äldre arkitekturen beräknas. Det sker genom att InZone exekveras tre gånger, en för A, B och C, och har O(n) vilket ger O(3n). Speed exekveras sedan också tre gånger, har O(n) och tar in n_2 objekt, vilket ger $O(3n_2)$ och CollisionDetect exekveras tre gånger har $O(n^2)$ och tar in n_3 objekt vilket ger $O(3n_3^2)$. Dessa tre adderas och resultatet blir $O(3n_1 + 3n_2 + 3n_3^2)$.

Ordo är beräknat antal operationer det i värsta fall behövs för att uppnå målet och i detta fall blir båda $O(3n^2)$ när man sammanfattar dem eftersom det är den största variabeln som avgör. Detta visar att trots att den nya arkitekturen får ett aningen lägre tidskomplexitet är det inte stor skillnad mellan dem.

I den nya arkitekturen går det att enkelt ändra ordning på modulerna för att anpassa dem och förändra tidskomplexiteten så det passar ditt system och detta möjliggör ett sätt att optimera systemet. Den tyngsta funktionen CollisionDetect ligger sist i våra exempel och har därmed flest instanser vilket kan vara ansträngande för systemet dock får den in färre objekt än den skulle fått om den låg tidigare i kedjan. Beroende på hur många Targets som finns i listan samt hur många objekt som InZone och speed gallrar bort kommer CollisionDetect att vara optimal på olika positioner i kedjan. CollisionDetect i form av den tyngsta beräkningen skall få in så lite objekt som möjligt samt upprepas så få gånger som möjligt, sedan avgör tillfället vilken position som är optimal.

För att förtydliga kommer här ett exempel. Vi jämför fortfarande tidskomplexiteten för modul A, B och C i figur 12, dock enbart för den nya arkitekturen. Detta jämförs med två olika ordningar på modulerna, en som tidigare och en med CollisionDetect modulen först och Speedmodulen i slutet, dessa visas i Figur 13.



Figur 13 Två olika ordningar på modulerna i den nya arkitekturen.

I vårt exempel innehåller Target listan 1000 objekt, Speed modulen gallrar bort 50 % och InZone modulen gallrar bort 75 % och CollisionDetect modulen gallrar bort 90 %. Här nedan följer beräkningarna för båda ordningarna på modulerna.

1: InZone – Speed – CollisionDetect	$O(n + 2n_2 + 3n_3^2) = O(3n^2)$
$n = 1000$, $n_2 = (n*0.25) = 250$, $n_3 = (n_2*0.5) = 125$, $n_4 = (n_3*0.1) = 12.5$	
Beräkning: $1000 + 2*250 + 3*(125^2) = 1000 + 500 + 3*15625 = 48\ 375$ operationer	
2: CollisionDetect – InZone – Speed	$O(2n^2 + 2n_2 + 2n_3) = O(2n^2)$
$n = 1000$, $n_2 = (n*0.1) = 100$, $n_3 = (n_2*0.25) = 25$, $n_4 = (n_3*0.5) = 12.5$	
Beräkning: $2*1000^2 + 2*100 + 2*25 = 2*1000\ 000 + 200 + 50 = 2\ 000\ 250$ operationer	

Som ni ser på beräkningarna är det stor skillnad på antal operationer som utförs beroende på ordningen modulerna startar i. Exempel 1 gav oss ca 50 000 operationer medan exempel 2 vilken hade den tunga CollisionDetect modulen i början gav oss ca 2 000 000 operationer. Här vägde inte antalet CollisionDetect trådar så stor roll som antalet objekt som modulen får in och skall beräkna på. Det är viktigt att vara medveten om att komplexiteten kan förändras enbart genom att placerar modulerna i en annan ordning. Detta exempel visar tydligt att bara för att tidskomplexiteten är lägre för exempel 2 behöver den inte vara optimal för det specifika systemet. En stor fördel med den nya arkitekturen är att användaren kan ändra ordning på modulerna för att minimera tidskomplexiteten i systemet eftersom resultatet blir ekvivalent oavsett vilken ordning modulerna exekveras i.

5.5 Analys Sammanfattning

Eftersom det är ett väldigt omfattande analys kapitel kommer här ett sammanfattande stycke vilket liksom tidigare delas upp i händelsehantering och dataöverföring.

5.5.1 Händelsehantering

Kraven för händelsehanteringen kan sammanfattas till att antalet skickade händelser skall minskas samt att varje modul enbart skall få de händelser den är intresserad av.

Dataöverföringen i systemet har minskat i de scenarion där varje modul enbart är intresserade av några händelser. Den optimeras mer jämfört med den äldre arkitekturen ju mer moduler som kommunicerar. Om varje modul däremot vill ha alla händelser i systemet ökar antalet meddelandeöverföringar, vilket inte är bra. Detta är dock inte ett problem för IBDn eftersom alla moduler inte kommer vilja ha alla händelser, det är så systemet är uppbyggt, men det kan vara bra att tänka på i andra system. Varje modul får enbart de händelser som modulen specificerar i den abstrakta händelsen vilket gör att även andra kriteriet är uppfyllt, sedan är det upp till användaren att specificera ett så korrekt abstrakt event som möjligt för att enbart få de viktiga händelserna.

5.5.2 Dataöverföring

Kravet för dataöverföringen kan sammanfattas till att moduler skall dynamiskt kopplas samman i ett gränssnitt för att byta data samt att funktionalitet skall centraliseras för att minimera inkonsekvens.

Den dynamiska sammankopplingen av moduler sker i gränssnittet genom att varje modul definierar vem den vill få data ifrån och tappar modulen koppling går den in i ett default läge där ingen ny data kommer in. Centraliseringen av koden analyserades och det framkom att inkonsekvensen i koden enbart är minimal och går att undvika helt genom att skapa många små moduler. Centraliseringen medförde även att det blev färre dubbelberäkningar i trådarna genom att samma funktionalitet beräknas på ett ställe och skickas vidare. Dock undviks inte all eftersom två moduler kan dela vissa indataobjekt men ta dem från olika ställen och då uppkommer dubbelberäkningar på dessa objekten. Arkitekturen minimerar antalet dubbelberäkningar och i vissa fall undviks de helt. En intressant upptäckt vid analysen av centraliseringen var att tidskomplexiteten går att påverka mycket genom att ändra exekverings ordningen på modulerna. Om de mest komplexa och tyngsta beräkningarna placeras i slutet optimeras antalet beräkningar, vilket är väldigt enkelt att genomföra i den nya arkitekturen eftersom alla funktionaliteter är frilagda i egna moduler.

6 Slutsats

Kapitlet börjar med en sammanfattning av det genomförda arbetet och resultatet av det producerade systemet. Det övergår sedan i en diskussion kring arbete och slutligen ett kapitel om framtida arbete med arkitekturen.

6.1 Sammanfattning

Arbetets mål är att finna en arkitektur som uppfyller de ställda kraven på kommunikationen och dataöverföringen ifrån kapitel 3.1. Detta har resulterat i två arkitekturer, en var för de två stora delproblemen: kommunikation och dataöverföring.

Kommunikationen genomförs med händelser som skickas till en hanterare och som sedan hanteraren jämför med olika abstrakta händelser för att hitta matchande. Om händelserna matchar skickas den konkreta händelsen vidare till de moduler som är förknippade med den abstrakta händelsen. Modulerna skickar abstrakta händelser till hanteraren och anmäler genom dessa att de är intresserade av händelser som matchar den händelsen.

Dataöverföringen är realiserad genom att varje modul som vill ha indata från en annan modul får den modulens namn och listnummer om det finns flera listor från gränssnittet. En förfrågan skickas sedan till hanteraren om att koppla upp sig till denna och är allt korrekt skapas en koppling mellan dem med en referens till modulen i fråga. Varje modul som kan ge utdata registrerar sig hos hanteraren för att sedan ha möjlighet att kopplas ihop med valfri annan modul.

Analysen av kommunikationsarkitekturen visar att arkitekturen optimerar antalet skickade meddelanden i systemet. Detta sker eftersom hanteraren väljer ut enbart de relevanta händelserna och skickar vidare dessa. Undantaget är när alla moduler är intresserade av väldigt många händelser var, då blir det fler meddelande skickade över systemet än tidigare. Det blir en konstant försämring med lika många meddelanden som är unika. Detta i sig är inget problem för IBDn eftersom den inte kommer att användas på det viset, men det kan vara avgörande för andra liknande system. Analysen visar även att den nya arkitekturen är bättre anpassad till ett storskaligt system än den äldre.

Analysen av dataöverföringsarkitekturen visar att systemet blivit dynamiskt och flexibelt samt att funktionalitet har centraliserats. Upprepningar i systemet har minskat men undviktes inte helt i och med att varje modul kan ha flera instanser med lika värden fast som beräknar på olika mängd data, inkonsekvensen är dock borta. Komplexiteten har inte försämrats men ej optimerats heller, dock finns möjligheten att förändra ordningen och minska komplexiteten beroende på ditt specifika system. Varje modul har enbart vetskap om hanteraren samt de moduler den är sammankopplad med vilket håller antalet beroenden lågt.

6.2 Diskussion

Det nya systemet har många fördelar så som utbyggbart, anpassat för storskalighet, konsekvent kod mm. Genom att IBDn har brutits ner till att bestå av många enklare moduler istället för ett fåtal väldigt komplexa och att det finns möjlighet att kommunicera mellan dem skapar detta många nya möjligheter. Modulerna är enklare än tidigare eftersom de skall kunna användas i fler olika sammanhang och tillsammans med många andra moduler. Trots att varje modul blir mer enkel har systemet nu möjlighet att länka samman flera enkla moduler för att finna väldigt

specifika beteenden. Detta ger även användaren samt utvecklaren mer möjligheter att enkelt förändra moduler efter specifika krav genom att slå samman egenskaper från flera moduler.

Samtidigt som det blir enklare för en användare att skapa sitt specifika beteende istället för ett färdigkonstruerat innebär även arkitekturförändringen att det behövs fler steg för att nå dit, och det i sig innebär fler möjligheter att göra fel. Det kommer även kräva att alla moduler tydligt beskriver vad de gör för att de skall användas vid rätt tillfällen osv. IBDn har utvecklats till att utav fyra moduler kunna skapa inte mindre än 64 unika kombinationer av sammankopplingar av modulerna, vilket sedan även kan skapas med olika instanser av samma moduler. Detta ger användaren stor frihet och mycket makt att skapa egna kombinationer och specifika kedjor av moduler.

Många har tidigare slagit samman olika designmönster och skapat arkitekturer för att få fram nya egenskaper (Sharifi et al., 2010; Rimac et al., 2008) och detta är ett liknande arbete med en egen nisch mot IBDn. Arkitekturen är anpassad för IBDn men likande system kan också använda sig av arkitekturerna och få nytta av dess positiva egenskaper. Genom att tillämpa min arkitektur i t.ex. ett händelsesystem i ett spel kan man få fram de positiva egenskaperna som att enbart de som vill få händelser får dem. Detta kan realiseras vid att t.ex. enbart alla i samma geografiska zon får händelser samt vill t.ex objekt enbart få objekthändelser medans varelser vill få alla, med hjälp av logikparsern kan det specificeras väldigt exakt vem som vill ha kunskap om vad. Det är dock viktigt att tänka på tidskomplexiteten hos modulerna och hur detta påverkar systemet, samt att om alla händelser skall skickas vidare till alla moduler är detta troligen inte optimalt för det systemet.

6.3 Framtida arbete

De färdiga arkitekturerna uppfyller kraven dock vet jag om ett par förbättringar som jag hade velat få med men som inte hanns med under tidsrymden för examensarbetet. Här nedan följer några av dessa. Det finns många förbättringar sammankopplade med optimering av systemet för storskalighet eftersom mitt examensarbete och arkitekturen öppnade upp för många nya möjligheter.

En av de viktigaste optimeringarna av arkitekturen jag rekommenderar är att använda designmönstret Readers/Writers för att optimera synkroniseringen mellan modulerna. Detta är ett designmönster som möjliggör att flera moduler kan läsa data ifrån en modul samtidigt men när den datan skall förändras kan ingen läsa eller skriva samtidigt. Detta gör systemet trådsäkert och mer optimerat än hur arkitekturen fungerar idag när alla moduler går in i den kritiska sektionen var för sig. Detta är en optimering som gör att systemet kommer klara fler datakopplingar.

Logikparsern kan behöva optimeras eftersom den kommer att exekveras väldigt många gånger och kan annars bli en flaskhals. Den går att optimera genom t.ex. parsa texten till ett tal istället för en sträng osv. Det finns även en möjlighet att logik parsern skall jämföra en hel sträng mot en händelse istället för en abstrakt händelse med en konkret händelse, detta beror på hur systemet kommer vidareutvecklas. Anledningen att logik parsern realiseras som en sträng är för att händelser har sedan tidigare i IBDn varit uppbyggda av Strängar och enums samt att strängar är ett bra sätt att få in data från gränssnittet.

Det går även att se andra arbeten i samband med utvecklingen av händelsehanteringen så som att händelser skall starta kedjor av händelser, en modul startar olika moduler beroende på dess indata som i sin tur kan starta nya moduler. I nuläget är modulerna

programmerade att uppdateras med ett visst intervall men genom att förändra detta till att de uppdateras när en viss händelse har skett går systemet att både optimeras genom att alla moduler är sovande tills händelsen sker samt att detta ger många nya möjligheter för användaren av systemet.

Tidigare nämnda problem är främst mindre tillrättningar och vidareutvecklingar av den utvecklade arkitekturen men det finns även större projekt i nära anslutning till mitt examensjobb. Detta beror främst på att genom att ha möjlighet att koppla samman moduler uppkommer nya utmaningar med IBDn så som att optimera startade moduler och ta hand om kopplingarna optimalt. En lösning på problemet om att det kommer behövas fler led av moduler vilket gör systemet svårare att använda är att vidareutveckla IBDn så att den kan gruppera ett antal moduler samman för mer övergripande funktioner. Detta medför även att systemet måste undvika startandet av dubbletter i olika grupperingar utan istället kopplar samman dem med tidigare startade moduler. Det stora blir sedan att optimera ordningen av modulerna som exekveras så att de blir så tidsoptimala som möjligt i enhet med resultaten från kapitel 5.4.3 om tidskomplexitet och dess påverkning på systemet. Genom att förändra ordningen på exekverande moduler kan systemet förbättras avsevärt. Det går även att koppla samman moduler på flera andra vis än enbart ordning, de kan t.ex. utföras union och snitt på två listor för att skapa en ny lista. Den resulterande listan kan i vissa fall ge samma resultat som t.ex. en kedja av de två modulerna men genom att göra ett snitt istället går listorna att även använda till andra kopplingar.

Arkitekturen går att använda i många olika system och kan anpassas till de flesta händelsehanteringssystem eller reaktiva dataflöden. Detta kan ge stora vidareutvecklingsmöjligheter så som att anpassa systemet för dataspel där olika uppdrags ges och beroende på vilken väg som tars samt vilka saker som uppfyllts under varje uppdrag kan olika vägar i mellan uppdragen visa sig samt hemliga uppdrag. Detta skulle skapa mer levande story och uppdragskopplingar.

Referenser

Barroca L. & Henriques P. (1998) A framework and patterns for the specification of reactive systems. *Information and Software Technology* 40(3) pp 135-42

Bell D. (2000) *Software Engineering, a programming approach*, 3rd edition. Addison-Wesley

Ben-Air M. (2006) Principles *of concurrent and distributed programming*, 2nd edition. Addison-Wesley

Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. (1996) Pattern-Oriented Software Architecture: A System Of Patterns. John Wiley & Sons

Corkill D. (2003) Collaborating Software**,** Blackboard and MultiAgent Systems & the Future. *Proceedings of the International Lisp Conference*.

Coskun E., Grabowski M. (2005) Software complexity and its impacts in embedded intelligent real-time systems. *The Journal of Systems and Software* 78(1) pp.128–145

Di Stefano A. Pappalardo G., Santoro C., Tramontana E. (2007) A framework for the design and automated implementation of communication aspects in multi-agent systems. *Journal of Network and Computer Applications* 30 (1) pp 1136–52

Eden A. & Kazman R. (2003) Architecture, Design, Implementation. *International conference of Software engineering – ICSE,* 2003 Portland

Gamma E., Helm R., Johnson R., Vlissides J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley

Garlan D. & Shaw M. (1993) “An Introduction to Software Architecture” in *Advances in Software Engineering and Knowledge Engineering,* Volume 1. World Scientific Publishing Company

Haggar P. (1999) Practical Java Programming language Guide. Addison-Wesley

Hariharan B. & Aluru S. (2005) Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods. *Parallel Computing* 31(3-4) pp 311-31

Hsueh N., Shen W., Yang Z., Yang D. (2008) Applying UML and software simulation for process definition, verification, and validation. *Information and Software Technology* 50(9-10) pp 897-911

Hunt A. & Thomas D. (1999) *The Pragmatic Programmer*. Addison-Wesley

Höppner F., Klawonn F., Kruse R., Runkler T. (1999) Fuzzy cluster analysis. Wiley

MacDonald S., Anvik J., Bromling S., Schaeffer J., Szafron D., Tan K. (2002) From patterns to frameworks to parallel programs. *Parallel Computing*, *28(12) pp. 1663-1683*

Pressman R. (2000) *Software Engineering, a practitioner’s approach*, 5ft edition. McGraw-Hill

Rimac I., Borst S. & Walid A.(2008) *Peer-Assisted Content Distribution Networks: Performance Gains and Server Capacity Savings*. Bell Labs Technical Journal 13(3) pp. 59–70

SAAB group. (2009) *IBD Productspecifikcation 1301*, internal documentation at SAAB

Schmidt D. & O'Ryan C. (2003) Patterns and performance of distributed real-time and embedded publisher/subscriber architectures. *The Journal of Systems and Software*. 66(3) pp 213-223

Sharifi M., Mirtaheri S., Khaneghah E. (2010) *A dynamic framework for integrated management of all types of resources in P2P systems*. J Supercomput 52 pp. 149–170

Timothy E. &Yair L. (2009) Towards a Guide for Novice Researchers on Research Methodology: Review and Proposed Methods. *Issues in Informing Science and Information Technology* 6(1) pp 323-37

Weiss M. (2006) *Data Structures and Algoritm Analysis in C++, 3erd Edition.* Pearson Higher Education

Zalewski J. (2001) Real-time software architectures and design patterns. *Annual Reviews in Control* 25(1) pp.133-146