IAE

En plattform för automatiserad analys och flexibel informationsinhämtning

LUNDS UNIVERSITET
Lunds Tekniska Högskola

LTH Ingenjörsskolan vid Campus Helsingborg
Datateknik

Examensarbete:
Daniel Ardby (dardby@gmail.com)
Marcus Klang (m-klang@bostream.nu)

Skapad med LaTeX den 16 juni 2010

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2010

Sammanfattning

DISA är ett internationellt företag i gjuterimaskinbranschen som har en framtidsmål att kunna övervaka sina 250 olika gjuterimaskinstyper som finns ute hos deras kunder. Från maskinerna kan massvis med information hämtas och utifrån denna information kan man utvärdera hur bra de fungerar. Informationen lagras i form av mätvärden hos kunden för framtida bruk men används i nuläget inte till något. Företaget vill kunna nyttja all den information som lagras om maskinerna för att kunna erbjuda bättre kundservice i form av snabbare reservdelsleveranser. Detta minskar risken för att kunders maskiner ska drabbas av totalt stillestånd.

Ovanstående ambitioner ledde till utvecklingen av en plattform för flexibel informationsinhämtning och automatiserad analys med grafisk presentation. Utvecklingen gjordes på förfrågan av DISA Industries A/S för att fylla deras behov av ett verktyg som kan varna ifall deras kunders gjuterimaskiner håller på att gå i sönder och som kan överblicka maskinernas maskinstatistik.

Informationsinhämtning handlar om att på ett så flexibelt och generellt sätt som möjligt hämta hem den maskininformation som finns lagrad, till ett system som bearbetar den genom analys, för att sedan förbereda den för presentation i ett användargränssnitt.

Automatiserad analys är den komponent som analyserar maskinernas mätvärden och varnar om dessa ligger utanför förutbestämda gränsvärden. En maskin får t.ex. inte ha en oljetemperatur för hydraulikpumpen som överstiger ett visst värde för att maskinen ska fungera normalt.

Grafisk presentation handlar om att presentera analyserad och inhämtad maskininformation i ett användargränssnitt på ett så optimalt sätt som möjligt för företagets servicetekniker. I denna del av arbetet utnyttjas den senaste tekniken för utveckling av grafiska Windows-applikationer, *WPF*.

Arbetet har resulterat i en komplett plattform för flexibel informationsinhämtning och automatiserad analys kallad IAE.

Nyckelord: DISA, Informationsinhämtning, Automatiserad analys, Maskingstatstik, Presentation, WPF

Abstract

DISA is an international company in the foundry industry that has a goal of being able to monitor their 250 different casting machines worldwide. A massive amount of information can be retrieved from the machines and by evaluating this information you can get an indication of how well the machines operates. The information is stored as meassured values at the customer site but is not used any further. The company wants to make use of all this stored information to offer better customer service by optimizing their spare part deliveries. Thereby preventing customers' machines from a total standstill.

Because of this, a platform that supports flexible information retrieval and automatic analysis for graphical presentation was needed to be developed. The development was carried out at request by DISA Instustries A/S to fulfill their need of a tool that can send out alerts if their customers' casting machines are about to be worn out and give an overview that presents the machine statistics.

Information retrieval is all about designing the most flexible and generic way to retrieve the stored machine information and deliver it to a system. The system manages the information by analyzing and organizing it for presentation in a graphical user interface.

Automatic analysis is the component that analyzes the values of the machines and warns if these are out of predetermined bounds. E.g. a machine is not allowed to have a hydraulic pump oil temperature that exceeds a certain value under normal working conditions.

Graphical presentation deals with the process of presenting analyzed and retrieved machine information in an optimal user interface for the DISA service technicians. In this section, the latest technology for developing graphical Windows applications, *WPF*, is utilized.

The work has resulted in a complete platform for flexible information retrieval and automatic analysis called IAE.

Keywords: DISA, Information retrieval, Automatic analysis, Machine statistics, Presentation, WPF

Förord

Föreliggande examensarbete har utförts i samarbete med DISA Industries A/S i Herlev under våren 2010. Arbetet omfattar mjukvaruutveckling inom området datateknik och motsvarar 22,5 högskolepoäng. Det har svarat för den avslutande delen av vår utbildning som högskoleingenjörer vid Ingenjörshögskolan i Helsingborg på Högskoleingenjörsprogrammet i Datateknik.

Utvecklingen av IAE-plattformen har varit mycket lärorik och det har varit spännande att få omsätta förvärvade teoretiska kunskaper på ett praktiskt och verkligt problem som behövde lösas.

Med detta förord vill vi ta tillfället i akt att rikta ett särskilt tack till vår handledare Nils Assarsson för den möjlighet han gav oss att få utföra examensarbetet på DISA. Vi vill också passa på att tacka vår examinator Mats Lilja för den ovillkorliga hjälp han gett oss under arbetets och utbildningens gång.

Helsingborg, Juni 2010.

Daniel Ardby
Marcus Klang

Innehåll

Konventioner

I rapporten har en del ord valts att formateras på sitt särskilda sätt beroende på vad de har för betydelse. I följande punktlista visas de formaterade ord som används i denna rapport:

- `Typewriter text` används för ord i utvecklingsmässiga sammanhang.
- *Betonad text* används för ord som bör uppmärksammas extra.
- **Fetmarkerad text** används för ord som är av större vikt för detta examensarbete.

1 Projekt

1.1 Inledning

1.1.1 Bakgrund

DISA är ett danskt företag inom gjuterimaskinbranschen som erbjuder ett komplett sortiment av järn- och aluminiumproduktionslösningar för den internationella gjuteriindustrin. Produktionslösningarna består utav leverans av gjuterimaskiner, s.k. DMM (DISA Moulding Machine), service för maskinerna samt installation av maskiner för hela fabrikers verksamhet. Lösningarna levereras till kunder över hela världen. Gemensamt för alla DISA:s produktionslösningar är ett världsomspännande kundservicenätverk som strävar efter att säkra tidsleveransen av reservdelar och alltid erbjuda teknisk support. Idag skickas servicetekniker ut till kunderna för att lösa uppkomna problem och serva företagets maskiner. Just nu ger man service efter felet har uppstått.

Företaget vill hänga med i teknikutvecklingen och har därför satt upp som målsättning att kunna utnyttja Internet för fjärrövervakning av sina kunders maskiner för att på så sätt erbjuda kunderna bättre service, kunna sätta in preventiva åtgärder innan en maskin går sönder, och för att kunna minska resandet för serviceteknikerna, vilket både är tidsbesparande och kostnadseffektivt. Målet är också att försäkra att man kan skicka ut reservdelar i tid så att man kan förhindra totalt stillestånd av kunders maskiner, vilket för kunden givetvis innebär stora kostnadsbesparingar. Gjuterimaskinerna som DISA tillverkar genererar en stor mängd information som i dagsläget bara sparas på plats men som inte används vidare, d.v.s. information som skulle kunna utnyttjas vid felsökningsanalys av gjuterimaskiner men som istället går förlorad.

DISA har startat upp ett projekt som heter RMS (Remote Monitoring System) för att uppnå sin målsättning. Syftet med RMS är att ta vara på den stora mängd information som sparas på plats vid maskinerna. Redan hösten 2009 startades ett delprojekt som engagerade studenter på Campus Helsingborg för att lösa första delen av RMS projektet, nämligen maskinövervakning med hjälp av signalsampling och lagring av signaldata som sedan ger möjlighet för analys i Danmark. DISA har som femårsplan att installera detta övervakningsprogram på över 200 maskiner.

Som en annan del i RMS projektet återfinns utvecklingen av ett användargränssnitt för presentation av gjuterimaskinmätvärden som serviceteknikerna kan använda sig av för att analysera en gjuterimaskins status. Detta användargränssitt har till uppgift att sammanställa all information som varje gjuterimaskin genererar.

1.1.2 Problemformulering

Problemställningen för examensarbetet är därför tillägnad just konstruktionen av användargränssnittet fast med fokus på den underliggande programarkitekturen och programstrukturen.

DISA använder i dagsläget ett avancerat program vid namn *DasyLab* för analys av maskinsignaler. Personalen är väl förtrogen med detta program eftersom det är huvudverktyget för diagnostisering av maskiners hälsa och programmet är tänkt att fortsätta användas under felanalys. Dock är det ett verktyg som lämpar sig bäst för mer djupgående och avancerad analys när något i en maskin verkligen är felaktigt och felet redan har identifierats som existerande. Det är alltså tämligen överarbetat att utnyttja verktyget om man bara ska säkerställa att maskinen fungerar normalt eller om man vill se ifall det existerar några problem överhuvudtaget. Med andra ord klarar sig DISA inte med programmet som enda utgörande del i deras felsökningssystem utan det saknas viss funktionalitet som kan ge en överskådlig överblick av maskiner och deras maskindata.

Företaget önskar sig följaktligen ett program som på ett enkelt och snabbt sätt tar reda på om det existerar något fel. Inte vad felet är utan endast att det finns ett fel som bör uppmärksammas och samtidigt varnar om detta. Med andra ord någon form av automatiserad och lättviktig föranalys av maskindata som utförs och som talar om ifall den mer djupgående analysen DasyLab erbjuder är nödvändig för maskinen i fråga. Programmet ska även bistå med maskinöverblick och fungera som första anhalten i felsökandet av maskiner.

Vidare har DISA som önskemål att funktionalitet ska finnas för presentation av maskinstatistik i form av olika diagram som t.ex. cirkeldiagram, stapeldiagram, tabeller m.m. De är också i behov av en funktion som exporterar högfrekvent samplad maskinsignaldata till DasyLab kompatibelt ASC-format så att serviceteknikerna kan fortsätta använda DasyLab för mer noggrann analys.

Därmed åtar sig examensarbetarna utmaningen att utveckla ett program utifrån ovan beskrivna problem. Som en sammanfattning av programmets problemområden och för att få en lättöverskådlig blick över vad som måste utvecklas presenteras följande grovindelning: *informationsinhämtning*, *organisering av informationen*, *automatisk analys av informationen*, *presentation av informationen* och *exportering av informationsresultat*. Problemområdena beskrivs på följande vis:

Informationsinhämtning innebär extrahering av maskindata från DISA:s centrala RMS server till programmet som ska utvecklas.

Organisering betyder att konfigurera presentationen av maskindata.

Analys handlar om funktioner som underlättar serviceteknikernas analysprocess av maskindata genom detektion av gränsvärdesöverskridande.

Presentation innebär visning av maskindata grafiskt på ett optimalt sätt för serviceteknikerna.

Exportering är funktionaliteten att vidarebefordra resultatet av den analyserade datan samt maskinstatistisk data till externt rapportgenerationsprogram. Även export av högfrekvent samplade signaler till ASC-format ingår i detta område.

Utifrån ovan presenterade problem blir examensarbetets problemformulering följande:

Hur implementerar man ett system som på ett flexibelt sätt hämtar data från ett databassystem och sedan utför automatiserad analys av datan för att slutligen göra en presentation av denna?

1.1.3 Frågeställningar

En av projektets arbetsmetoder bestod i att en del frågeställningar iakttogs vid projektets början. Detta gjordes dels för att avgränsa det valda ämnesområdet så att det inte blev allt för omfattande och dels för att underlätta för projektarbetarna så att dessa inte skulle halka av ämnet och komma på sidospår. Frågor ställdes som:

Hur implementerar man en flexibel informationsinhämtningsarkitektur? D.v.s. en arkitektur som är så pass generell att den kan hämta data från vilken källa som helst och leverera denna data till vilken kommunicerande godtycklig komponent som helst.

Hur sköter man delen som har med organisering att göra?

Hur implementerar man en analysmotor?

Vilka presentationssätt finns det att visa maskindata på?

Hur utvecklar man ett gränssnitt mot ett yttre system med uppgift att leverera resultatet från analysen och informationsorganiseringen?

Hur konstruerar man ett system där alla komponenter är byggda som självständiga enheter som fungerar ihop men sådana att man kan byta ut godtycklig komponent mot en annan, t.ex. en nyare version av komponenten, och ändå få systemet att bibehålla funktionalitet likt ett modulbaserat system?

Hur går det till under utvecklingsprocessen av ett användargränssnitts underliggande programarkitektur?

Hur strukturerar man koden i ett större programvarusystem?

Hur gör man upp en högnivå design?

Hur fungerar testningsfasen i utvecklingsprocessen?

Hur programmerar man grafiska användargränssnitt?

Med ovanstående frågor grundlades de riktlinjer projektet skulle följa, d.v.s. vad för ny kunskap projektet skulle bidra med och vilka nya lärdomar det skulle tillföra projektarbetarna.

Frågorna talar också om vad detta examensarbete inriktar sig på för problemområden och vad läsaren har att vänta sig och kommer få besvarat efter att ha läst igenom det.

1.1.4 Syfte

Examensarbetet har väsentligen två övergripande syften. Dels att hjälpa DISA med en del av det tidigare beskrivna RMS projektet, i företagets strävan på att förbättra sina servicetjänster för sina kunder. Dels att projektarbetarna ska införskaffa sig en gedigen kunskapbas i mjukvaruutveckling och utveckla den förmåga som krävs av en ingenjör i arbete ute på företag. Ingenjören får inte vara främmande för ny teknik och måste därför alltid vara öppen för att inhämta ny kunskap.

Syftet för de enskilda examensarbetarna är att projektet ska ge en fördjupning i hur man programmerar generella komponenter som fungerar som självständiga enheter och därmed kan användas tillsammans med godtyckligt valda interagerande enheter.

Vad som är viktigt för läsaren att förstå är att examensarbetet med dess problemformulering att utveckla ett användargränssnitt, syftar till att undersöka hur den underliggande programarkitekturen för ett användargränssnitt kan designas och att fokus *inte* ligger på användargränssnittet som sådant och hur man utformar det så användarvänligt som möjligt. Det får bli som uppgift till nästa gång.

Arbetet syftar också till att ge grundläggande kunskaper i programmeringsspråket *C#* och den relaterade plattformen *.NET* med tillhörande ramverk. Även *WPF* (Windows Presentation Foundation) och *ADO.NET* (Active Data Objects) ska ges möjlighet för utforskning.

Slutligen finns det ett sekundärt syfte med arbetet och det är att det också ska ge övning i teknisk rapportskrivning med tillhörande muntlig presentation. Dessutom är det meningen att redan förvärvade kunskaper inom utbildningen ska ges möjlighet för tillämpning på ett verkligt problem som behöver lösas.

1.1.5 Målsättning

Målet med examensarbetet är att utveckla ett program för presentation av gjuterimaskindata och som kan användas vid felsökning av gjuterimaskiner. Programmet ska hålla tillräckligt hög kvalitet för att DISA ska ha nytta av och vilja använda sig av programmet i sin verskamhet.

Programmet ska innehålla ett fungerande användargränssnitt som är optimalt anpassat för serviceteknikerna och som underlättar felsökning av gjuterimaskiner samt sammanställer status för maskinerna.

1.1.6 Avgränsningar

Projektets avgränsningar är att ett fullskaligt presentationsverktyg, **STUI** (Service Technician User Interface), för signalanalys av företagets gjuterimaskiner ska produceras. Fullskaligt i den meningen att komponenter från de fem delar som utgör den tidigare nämnda grovindelningen i avsnitt 1.1.2 på sidan 11, ska utvecklas. Rent konkret så innebär det att komponenterna består av följande saker:

- En komponent med programarkitektur för informationsinhämtning från ett databassystem. Det vill säga en komponent som består av en uppsättning SQL-frågor och logik som kan ta emot resultatet av frågorna.
- En komponent med logik som hanterar informationen som har hämtats och organiserar den till lämpligt format för presentation.
- En komponent som innehåller en uppsättning funktioner vilka bearbetar den inhämtade informationen genom analys. Analys som kontrollerar ifall värden håller sig inom förutbestämda gränser.
- Ett användargränssnitt som ger en överblick av gjuterimaskiners status och presenterar maskinstatistik.
- En komponent som kan användas av ett externt system för att komma åt den information som har analyserats och organiserats.

Ovanstående punkter utgör således det som är tänkt ska uträttas i detta projekt. Examensarbetarna lyckades slutföra informationsinhämtningen, analyseringen och enklare version av STUI-användargränssnittet till examensarbetets avslut.

Observera att även projektets frågeställningar fungerar som en del i avgränsningen.

1.2 Nulägesbeskrivning

Nedan ges en beskrivning av både nuläget i detta projekt samt hur långt man kommit i övrigt i RMS projektet. Som nulägesbeskrivning av det föreliggande projektet ges en förstudie av den teknik som anses aktuell för projektet. Förstudien är ett resultat av den litteratursökning som utfördes i projektet och visar på ett urval av den kunskapsinhämtning

som gjordes i början av projektet för att kunna arbeta med projektets problemformulering.

I nulägesbeskrivningen av RMS projektet ges det istället en förteckning med tillhörande förklaringar av de tekniska system, hårdvaru- som mjukvarusystem, som är i drift för tillfället. Detta för att dels läsaren ska få förståelse för och överblick över projektomgivningen men också för att projektarbetarna ska veta vilka system som finns tillgängliga och som måste tas i beaktande under utvecklingen.

1.2.1 Förstudie

Programmeringsspråket C# och plattformen .NET

Det primära programmeringsspråket för .NET är C#. Man kan säga att C# är ett programmeringsspråk som har dragit nytta av erfarenheterna från tidigare programmeringsspråk som C, C++, Java och Visual Basic genom att plocka godbitarna från vart och ett av dessa programmeringsspråk och samla dem i ett och samma språk. Från programmeringsspråket C tog man tillvara på den höga prestandan som språket medför, från C++ ärvde man den objektorienterade filosofin, från Java anammade man den höga typsäkerheten samt skräpinsamlaren (eng. garbage collector) medan man från Visual Basic drog lärdom av den snabba programutveckling språket möjliggjorde. Genom att plocka styrkorna från dessa språk och samla dem i C# så har man skapat ett programmeringsspråk som är idealiskt för utveckling av komponentbaserade, flerskiktade och distribuerade klientapplikationer för Windows.

Eftersom C# är ett objektorienterat programmeringsspråk finns det stöd för att definiera och arbeta med klasser. Som bekant definierar klasser nya typer som gör det möjligt att utvidga språket för att bättre åskådliggöra det problem man försöker lösa och på så sätt är C# ett språk som har oändliga modelleringsmöjligheter. Programmeringsspråket innehåller nyckelord för deklaration av nya klasser och tillhörande metoder samt för implementering av inkapsling, arv och polymorfism, vilka är de tre huvudingredienserna som utgör objektorienterad programmering.

I C# är det så att både definitionen och deklarationen av klassen återfinns i själva klassdeklarationen. Det är alltså inte som i C++ där klassdefinitionen ofta ligger i en separat headerfil. Utan det är mer som i Java, där allt som har med klassen att göra återfinns i samma fil i vilken klassen först deklareras i. Dock finns det så klart språkkonstruktioner i C#,

i form av nyckelordet `partial`, som tillåter att klasser delas upp i flera filer för mer organiserad struktur något som inte tillåts i Java. T.ex. kan det vara så att man i en fil bara vill ha sådant som användaren av klassen utnyttjar, d.v.s. publikt deklarerade medlemmar, medan man i en annan fil vill gömma sådant som inte är relevant för klassanvändaren som t.ex. de privat deklarerade medlemmarna.

Som vilket nytt modernt objektorienterande programmeringsspråk som helst så introducerar C# en hel del nyheter jämfört med de tidigare språken. I det följande kommer ett axplock av de nya funktionerna att presenteras.

Något som skiljer sig från de äldre programmeringsspråken är språkkonstruktionen *egenskap* (eng. *property*) som möjliggör för användaren av en klass, åtkomsten av det tillstånd som ett objekt befinner sig i, som om tillståndet vore ett medlemsfält trots att åtkomsten implementeras likt en accessormetod. Detta är idealiskt, eftersom användaren av en klass vill ha direkt åtkomst till objektets tillstånd och inte gå via metoder medan skaparen av klassen samtidigt vill dölja klassens interna tillstånd i medlemsfält och tillhandahålla indirekt åtkomst via en accessormetod. Med en egenskap uppfyller man båda målen; den erbjuder ett enkelt gränssnitt för klassanvändaren i det att den ser ut som ett medlemsfält samtidigt som den implementeras som en accessormetod vilket följer god objektorienterad programmeringsed med datagömning.

C# stöder inte multipell ärvning som C++ gör, utan en klass i C# kan endast ärva från en basklass. Däremot stöder språket *gränssnitt* precis som Java gör och en klass kan implementera flera sådana. Av konventionella skäl brukar man inleda gränssnittsnamn med stort I, som i `IDataResult`.

Strukturer är något som härstammar från C och C++ och finns därmed också i C#. Strukturerna i C# skiljer sig dock ganska mycket från sina äldre släktingar. I C++ fungerar en struktur exakt som en klass fast med skillnaden att standardåtkomst är public istället för private. I C# är strukturer av värdetyp, till skillnad mot klasser som är av referenstyp. D.v.s. strukturerna är av en datatyp som är begränsad och lättviktig som när den initieras ställer mindre krav på operativsystemet och tar upp mindre plats i primärminnet än en vanlig klass. En struktur är till för att beskriva enkla datatyper som bara precis består av data och inte någon annan logik. Strukturer i C# kan inte ärva från en klass eller ärvas vidare, men en struktur kan implementera gränssnitt.

En nyhet i C# jämfört med tidigare programmeringsspråk är språkkonst-

ruktionen `delegate`. Delegater möjliggör ett sätt att anropa metoder indirekt, likt det sätt som funktionspekare i C++ fungerar på. Men det finns en väsentlig fördel med delegater jämfört med funktionspekare och det är att de är typsäkra referenstyper (en form av klass) som kapslar in metoder med specifika signaturer och returtyper. Det finns olika varianter på delegater; anonyma delegater och delegater i samband med *Lambda*-uttryck. Delegater används ofta i samband med händelsehantering och som callbacks.

Språket har fullständigt stöd för integration med *XAML* (eXtensible Application Markup Language), vilket är en deklarativ syntax med taggar likt XML, och som används i programmering av Windows-applikationer med hjälp av WPF.

Från Java lärde man sig även hur användbart inline-dokumentation av kod är och man lade till stöd för detta för att förenkla skapandet av referensdokumentation till de program man skriver. Men man såg möjligheterna med XML och lät dokumentationen vara XML-baserad så att den enkelt skulle kunna formateras på olika sätt för t.ex. online-visning.

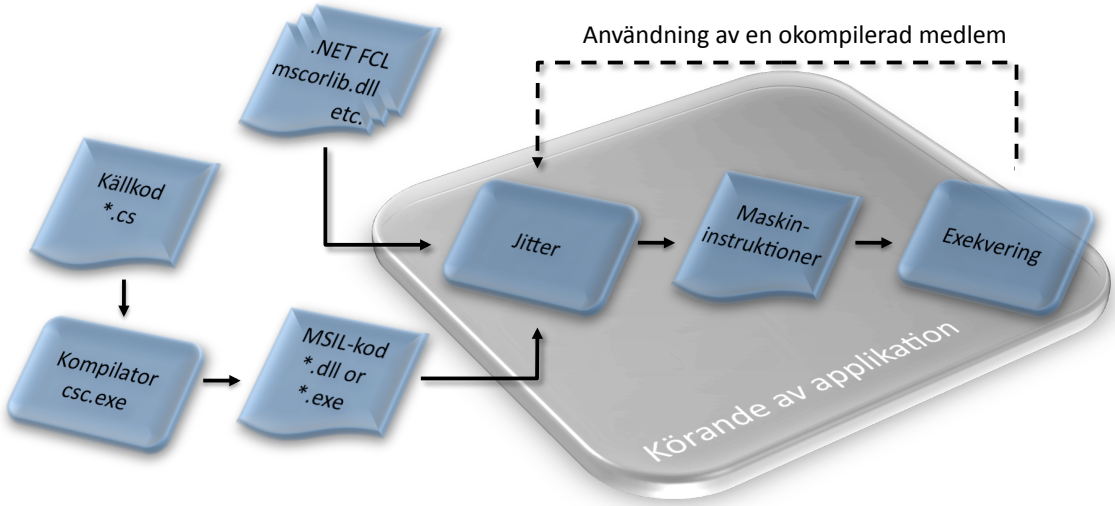
Trots all hittills presentation av C# utgör programmeringsspråket bara en del av ett mycket större ting, plattformen .NET. Plattformen .NET kan enkelt liknas vid en mjukvaruplattform som har placerats ovanpå det befintliga operativsystemet (d.v.s. den nativa plattformen) för en dator. Plattformen består av en samling som innefattar klienter, servrar och utvecklingsverktyg. T.ex. ingår det i plattformen ett ramverk som möjliggör skapande av praktiskt taget vad som helst; all typ av mjukvara som t.ex. server- och klientbaserade webbapplikationer, klientbaserade kontroller, Windows-applikationer (grafiska användargränssnitt) av typ Windows Forms och WPF, plattformsoberoende och webbläsaroberoende Internet-applikationer m.m.

Med ramverket följer det s.k. basklassbiblioteket (FCL) som är ett enormt omfattande bibliotek av redan färdigskrivna lösningar på återkommande programmeringsproblem. Som programmerare är tanken att man inte ska behöva återuppfinna hjulet varje gång man utvecklar och det är just vad basklassbiblioteket har konstruerats för så att man ska slippa återuppfinnandet och ha fundamentala programmeringslösningar tillgängliga på ett och samma ställe. Här finns hjälp för nästan allt man kan tänka sig; hämtning från datakällor, trådning, I/O till systemet, XML-avläsare, säkerhet, strånghantering, nätverkskommunikation m.m.

En huvudkomponent i .NET-ramverket är *Common Language Runtime* (*CLR*) vilket är en värdmiljö för körning av applikationer som är skrivna

för .NET på ett säkert sätt. Det är en motsvarighet till Javas virtuella maskin.

När man kompilerar källkoden som man har skrivit för .NET, så kompileras den inte direkt till maskininstruktioner som det görs i vissa andra programmeringsspråk, t.ex. C++, utan den kompileras till ett plattformsoberoende format, likt den *bytecode* Java använder sig av. Formatet kallas för *MSIL* (*Microsoft Intermediate Language*) eller ibland *CIL* (*Common Intermediate Language*) och sparas på hårddisken i en s.k. *sammanställningsfil* (eng. *assembly*) som antingen kan vara en DLL-fil eller en EXE-fil. När man sedan kör applikationen laddas MSIL-koden in i primärminnet och kompileras "on-the-fly" av en *jitter* till den aktuella plattformens maskininstruktioner. Jitter är ett annat uttryck för en *Just-In-Time*-kompilator. Att MSIL-koden kompileras till den aktuella plattformens maskininstruktioner innebär att det finns jitters för varje plattform. Till sist exekverar körmiljön, CLR, maskininstruktionerna. Hur kompileringsförfarandet går till illustreras av nedanstående figur 1.1.



Figur 1.1: *Kompilering i .NET*

I sammanställningsfilen ingår det mer än bara MSIL-kod. Den består också av *metadata* som beskriver varje klass, struktur, enumeration i sammanställningsfilen samt typens medlemmar som t.ex. egenskaper, metoder, medlemsfält osv. Metadatan innehåller information om signaturer och returtyper. Metadatan genereras av kompilatorn men som programmerare kan man även påverka denna m.h.a. språkkonstruktionen `Attribute`, vilken skrives precis före åtkomstdefinieraren för en dekla-

ration. Beskrivningen som metadatan för med sig är väldigt detaljrik och det gör sammanställningsfiler till helt självbeskrivande enheter. Det medför en fördel för dem som utnyttjar sammanställningsfilen i det att de behöver ingen extra information för att hitta de klasser, metoder, egenskaper m.m. som definieras i sammanställningsfilen, utan allt som behövs är filen i sig. Metadata stöder därför komponentorienterad programmering. Det finns egentligen två typer av metadata. Metadata som beskriver typer (typmetadata), vilken nämndes tidigare, och metadata som beskriver innehållet i en sammanställningsfil, det s.k. *manifestet*, eller sammanställningsfilen metadata. Manifestet beskriver innehållet i sammanställningsfilen genom att referera till typmetadatan. Men den refererar även till *resurser*, vilka är icke-körbara saker i ett program, som t.ex. bilder, ikoner, filer m.m. Det är alltså manifestet som gör sammanställningsfiler till helt självbeskrivande enheter.

I plattformen medföljer även det erkända utvecklingsverktyget från Microsoft, Visual Studio, vilket är en integrerad utvecklingsmiljö (eng. IDE) som underlättar utvecklandet genom att maximera produktiviteten i .NET m.h.a. olika inbyggda finesser. [3, 9]

Programmeringsspråket C# valdes som utvecklingsspråk i projektet av flera anledningar. Dels av alla ovannämnda goda kvaliteter som språket för med sig och dels för att det var naturligt ur utvecklingssynpunkt att fortsätta att utveckla i det programmeringsspråk som DACS[1] kodades i. Därmed undviks potentiella kompatibilitetsproblem.

ADO.NET

I det förra delavsnittet nämndes det att .NET ramverkets FCL innehåller stöd för hämtning från datakällor. Stödet för hämtning från datakällor riktar sig mot relationsdatabaser och stödet sammanfattas i en samling klasser som tillåter en att interagera med databaserna. Det är just detta paket, eller *namespace* som det på C#-språk heter, som går under namnet *ADO.NET*.

ADO.NET är uppbyggt i två alternativa dataarkitekturer; med det *frånkopplade* lagret och med det *uppkopplade* lagret. Varför man valt att göra denna uppdelningen är för att databasanslutningar betraktas som dyrbara och ohanterade resurser som man värnar om.

Med det frånkopplade lagret hämtas det data från en databas och sedan

[1] Vad DACS är för något kan läsas om på sidan 26.

sparas datan lokalt på datorn i en cache-liknande datastruktur som kallas `DataSet`. `DataSet` är en klass och är kärnkomponenten i det frånkopplade lagret i ADO.NET. I och med att man sparar ned datan lokalt kan man koppla ifrån anslutningen mot databasen och det sköter det frånkopplade lagret automatiskt. Datan som man sparat i en `DataSet` kan man sedan bearbeta i den lokala datorn på det sätt man vill och kontakt med databasen (d.v.s. uppkoppling av anslutningen) sker bara när man vill ändra dataposter eller ny data behöver hämtas. Med ett `DataSet` kan man alltså både läsa från och skriva till databasen.

Det frånkopplade lagret medför tydliga fördelar. En av fördelarna är så klart att databasen avlastas eftersom det sker mindre belastning av den då en anslutning kopplar från så fort den blir inaktiv (inte skickar eller tar emot data). Detta frigör resurser och lämnar plats för andra klienter att ansluta till databasen. En annan fördel är att applikationen man utvecklar och som använder sig av det frånkopplade lagret blir mer skalbar. D.v.s. applikationen kan hantera en större mängd anslutningar eftersom de återigen kopplar från så fort de blir inaktiva. Det är nämligen så att databasanslutningar är resurskrävande och flera tusentals uppkopplade och parallella sådana är svåra att upprätthålla för applikationen.

Med det uppkopplade lagret hämtas det data från en databas genom en `DataReader` som utgör kärnkomponenten för det uppkopplade lagret. Istället för att datan från databasen sparas lokalt i en cache så hämtas den stegvis från databasen, rad för rad, genom en anslutning som hela tiden hålls öppen, till det att man har gått igenom all data. `DataReader` ger en enkelriktad genomgång av data (från början till slut) och erbjuder bara läsåtkomst. `DataReader`-objekt är lättviktiga objekt och utnyttjandet av dem är det lättaste och snabbaste sättet att hämta data från en databas. Objekten passar utmärkt för tillfällen då man behöver behandla stora mängder data på ett så snabbt sätt som möjligt. Att göra detta med `DataSet` skulle bli minneskrävande eftersom det sparar datan i minnet. Vid tillfällen som dessa är det fördelaktigt att använda sig av det uppkopplade lagret.

Vid användning av det uppkopplade lagret måste man som användare explicit tala om när anslutningen ska upprättas och när den ska kopplas från. Det sköts alltså inte automatiskt som det gör i fallet med det frånkopplade lagret. [3, 9]

XML

XML står för *eXtensible Markup Language* och är ett uppmärkningsspråk likt HTML. Båda språken arbetar med data. Båda språken använder sig av *taggar* vars placering i koden har en innebörd. Båda språken regleras av specifikationer definierade av World Wide Web Consortium (W3C). Bägge språken har många likheter. Men det finns en väsentlig skillnad mellan de båda uppmärkningsspråken och det är att HTML används för att presentera data på webbsidor med fokus på hur data ser ut medan XML har som uppgift att "bära" data. Med det menas att XML ska lagra och transportera data på ett strukturerat sätt. Fokus för XML ligger på den semantiska betydelsen datan har. Medan många språk, programmeringsspråk som uppmärkningsspråk, verkligen uträttar något genom att de t.ex. bildar instruktioner för hur en dator ska arbeta som i fallet med programmeringsspråk eller beskriver hur data ska visas på en webbsida vilket HTML gör, så åstadkommer XML inte någonting.

Men vad det gör är att fungera som en standardiserad metod för att överföra information mellan olika system och applikationer. Som bekant så lagrar och behandlar datorsystem samt databaser sin data med sina egna specifika format, vilket gör dem inkompatibla med varandra när det gäller kommunikation. Men eftersom XML lagras i vanligt hederligt text format, vilket är ett mjukvaru- och hårdvaruoberoende sätt att lagra data på, kan data skapas och tolkas av olika system samt delas mellan systemen. P.g.a. detta har XML etablerat sig som ett universalformat för utbyte av digital information mellan system. [10]

Taggarna i XML kallas för *element* och ett XML-dokument består av ett stort träd av element där det första, vilket är det element som omfamnar alla andra element, kallas för *rot*-element. Det kan bara finnas ett sådant element i ett dokument men å andra sidan kan rot-elementet och dess *barn* (ett underelement till ett omgivande element som kallas för *förälder*) ha flera underelement. Enligt specifikationen som W3C definierar för XML, ska varje element ha en inledande tag och en sluttag (med undantag för element som är självavslutande då endast en tag behövs). Följande exempel beskriver hur ett XML-dokument kan vara uppbyggt.

XML Exempel

```xml
<bookshelf>
  <book category="Post Apocalyptic Horror">
    <title>The Stand</title>
    <author>Stephen King</author>
    <publication_date>
      <year>1991</year>
      <month>05</month>
    </publication_date>
    <publisher>Signet Book</publisher>
    <isbn nr="9780451169532" />
  </book>
  <book category="Spy novel">
    <title>The Little Drummer Girl</title>
    <author>John le Carré</author>
    <publication_date>
      <year>1983</year>
      <month>03</month>
    </publication_date>
    <publisher>Hodder and Stoughton</publisher>
    <isbn nr="0-340-32847-9" />
  </book>
</bookshelf>
```

I exemplet syns rot-elementet med dess beskrivande namn, `<bookshelf>`. Elementet är det enda yttersta elementet i dokumentet och det innehåller två barn: böckerna *The Stand* och *The Little Drummer Girl*. Varje bok tillhör en viss bokkategori och det beskrivs med hjälp av något som kallas för *attribut*. Alla element kan innehålla attribut och attributen består av en *nyckel* (`category`) och ett *värde* (`"Post Apocalyptic Horror"`). Sedan syns det att varje bok har flera barn där de flesta utav barnen regelrätt innehåller en inledande tag och en sluttag. Sluttag inleds alltid med tecknet `/`. Det är bara elementet `<isbn>` som skiljer sig genom att det antar ett självavslutande element. I och med det måste elementet ha sitt innehåll i ett attribut. Som synes i exemplet kan varje underelement innehålla flera underelement vilka i sin tur består av ytterligare underelement. Det finns ingen begränsning för hur långt ett träd kan nästla sig.

Ett utmärkande drag för XML är att det är utvidgbart och det motiverar varför dess namn innehåller ordet *extensible*. Med utvidgbart menas att användaren av språket själv kan definiera sina egna taggar. Som i det förra exemplet visades ett antal taggar. Samtliga taggar var egendefinierade. XML innehåller inga fördefinierade taggar. Alla taggar skapar man själv och betydelsen de får bestämmer man själv genom att ge taggarna en lämplig beskrivning i form av ett taggnamn. Det är detta som gör uppmärkningsspråket så genialt. Det öppnar dörrar för oändliga möjligheter vad gäller beskrivning av lagrad data. [3]

Windows Presentation Foundation

Windows Presentation Foundation brukar förkortas *WPF*. Det är en teknologinyhet med .NET version 3.5 och utgör ett av två sätt att skapa grafiska Windows-applikationer på. Det är den senaste tekniken man kan använda sig av för att utveckla grafiska användargränssnitt under Windows och har i och med det en rad fördelar jämfört med sin förfader. Det andra sättet att skapa applikationerna på (förfadern) är den betydligt äldre tekniken Windows Forms som kom med den första versionen av .NET, version 1.0, och som bygger på en teknik som använts i mer än 15 år. Det är faktiskt så att den tekniken introducerades redan med Windows 3.0 och den använder sig av två välkända delar av operativsystemet Windows: *User32* och *GDI/GDI+*. Det är User32 som bistår med det bekanta utseende som knappar, fönster, textrutor osv. i Windows har. Medan GDI:s uppgift är att ge uppritningsstöd för rendering av objekt, texter och bilder i Windows. Tillsammans utgör de huvuddelarna för att skapa användargränssnitt i Windows. Det har givetvis skett förbättringar och uppdateringar av teknologierna men oavsett hur mycket som görs med dem så kommer man inte ifrån de fundamentala begränsningar som finns i två komponenter som utvecklades för mer än 15 år sedan. De var helt enkelt inte byggda för dagens moderna datorarkitektur.

Med WPF slipper man många utav dessa begränsningar eftersom WPF använder sig av *DirectX*. DirectX är ett nyare API än User32 och GDI+ och används för hantering av multimedia under Windows-plattformen, men med tonvikt på grafik och video för utveckling av spel. Det utvecklades med fokus på att vara snabbt och utnyttjar grafisk hårdvaruacceleration från grafikkortet för att rendera komplexa texturer, 3D grafik och specialeffekter som exempelvis delvis genomskinlighet. WPF har t.ex. inte begränsningen som GDI+ har, med att högnivå grafikingredienser som

texturer och lutningar av grafikytor måste konverteras till instruktioner för varje pixel innan renderingen utförs av grafikkortet, vilket så klart resulterar i mycket långsammare rendering. Den har inte begränsningen eftersom DirectX med grafisk hårdvaruacceleration direkt kan tolka grafikingredienserna, vilket möjliggör för direkt rendering av grafikkortet. En indirekt följd av den grafiska hårdvaruaccelerationen är att WPF avlastar CPU:n med beräkningar och låter GPU:n (grafikprocessorn på grafikkortet) ta hand om så många utav dem som möjligt. Med andra ord blir det bättre prestanda.

Andra styrkor som WPF tillhandahåller är att det är *upplösningsoberoende*, grafiska komponenter som exempelvis knappar är utseendelösa, ordnandet av det grafiska gränssnittet är deklarativt (mer om detta längre fram), det finns stöd för multimedia, stöd för *stilar* och *mallar*. Att det är upplösningsoberoende innebär att oberoende av vilken upplösning användaren av Windows-applikationen har inställt på sin dator så skalar WPF upp respektive ned det grafiska gränssnittet för att anpassa sig till upplösningen så att de grafiska komponenterna fortfarande är proportionerliga gentemot varandra. Utseendelösheten av de grafiska komponenterna medför att man kan skräddarsy dem helt och hållet och det är endast funktionen hos komponenterna som är fast. Stilar och mallar tillåter utvecklaren att införa egna standardarder för hur vissa typer av komponenter ska se ut genom att utseendet skräddarsys.

När man utvecklar applikationer med hjälp av WPF programmerar man till stor del i ett deklarativt språk som heter *XAML* och som står för *eXtensible Application Markup Language*. XAML är en dialekt av det tidigare nämnda standardiserade språket XML. Att det är ett deklarativt språk innebär att det är uppbyggt av taggar som kallas för *element*, vilka kan liknas vid de taggar som återfinns i språk som *HTML* fast där standarden säger att varje element måste bestå av en inledande tag och en sluttag. Programmerandet i XAML utnyttjas vid själva skapandet av det grafiska gränssnittet under applikationsutvecklandet eftersom det med taggar enkelt kan åskådliggöras de grafiska komponenternas placering i gränssnittet. Det är nämligen så att man med ordningen taggarna inbördes emellan talar om var någonstans i gränssnittet som en komponent ska placeras. Detta sätt är mycket bekvämare och behändigare att utveckla grafiska användargränssnitt på än med det programmatiska sättet som förespråkas i Windows Forms.

Istället vänder man sig till vanlig hederlig kod, lämpligtvis med programmeringsspråket C#, för logiken bakom det grafiska användargränssnittet. Någonting måste ju hända när en knapp trycks in eller när ett val görs i en listvy. Det är logik i bakgrunden som sköter detta och den brukar benämnas *händelsehantering*. Händelsehanteringen fungerar på så sätt att de grafiska komponenterna eller *kontrollerna*, som de inom ämnet kallas för, utlöser s.k. händelser, vilka programmatiskt deklareras med ordet `event`, och dessa händelser kopplas sedan samman med *händelsehanterare* i vilka själva logiken för vad som ska hända vid ett knapptryck eller val i en listvy finns implementerad. [3, 4]

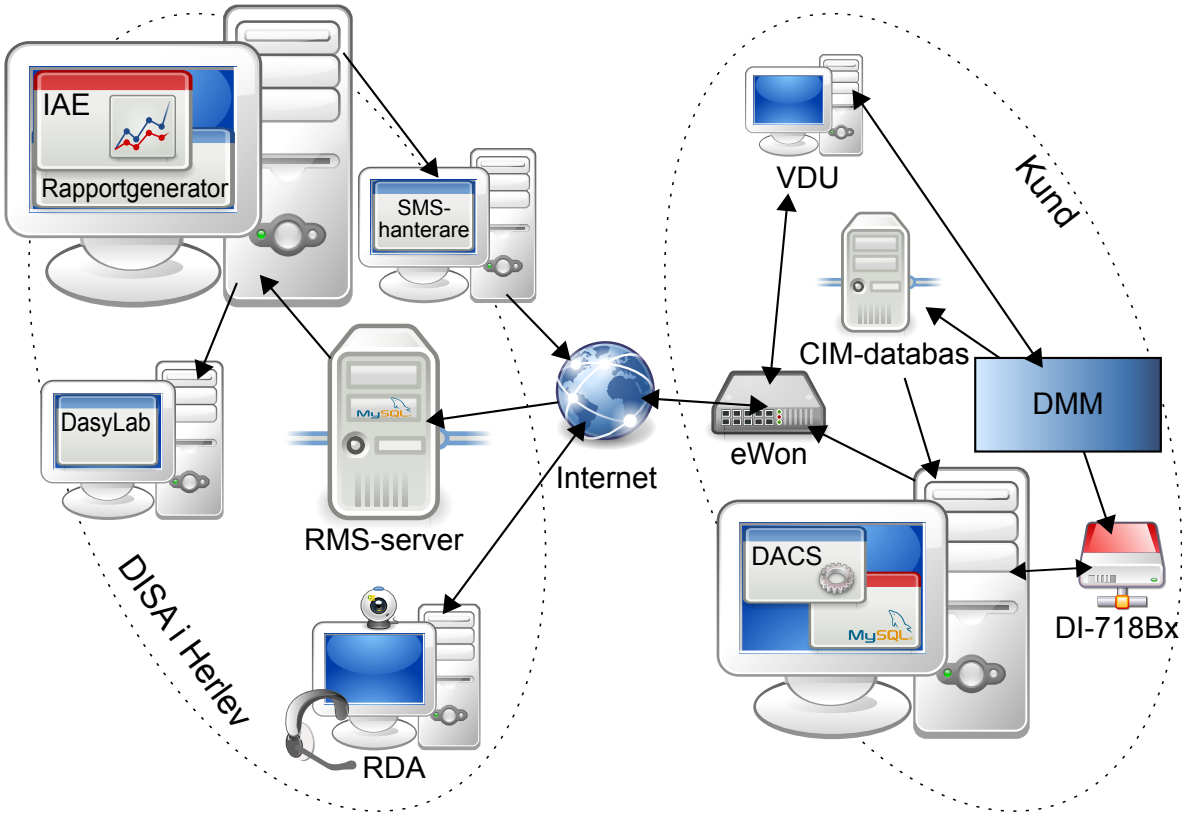
1.2.2 RMS-projektet

I figur 1.2 presenteras en överskådlig bild över miljön i RMS-projektet. Bilden visar hur systemen i projektet kommunicerar med varandra. Pilar som är dubbelriktade visar att trafiken går i båda riktningar medan enkelriktade pilar illustrerar simplextrafik. På bilden syns två lokala nätverk; dels DISA:s huvudkvarters nätverk i Herlev och dels godtycklig kunds nätverk. Nätverken knyts samman av Internet.

Observera att bilden är grovt förenklad och endast visar schematiskt hur det ser ut. T.ex. kan det saknas en del mellansteg i form av switchar för att fördela ut trafiken i respektive nätverk.

DACS

DACS står för *Data ACquisition Service* och är själva övervakningsprogrammet som nämndes avsnitt 1.1.1 på sidan 10. Programmet genomför övervakningen med hjälp utav signalsampling av olika maskinsignaler från en gjuterimaskin. Till sin hjälp har programmet samplingshårdvaran DI-718Bx från DataQ. Men mer om den i nästa delavsnitt. Efter att DACS har fångat upp signalerna under samplingssessionen lagrar den värdena i en MySQL-databas på en lokal dator ute hos kunden. Programmet tillåter att samplingssessioner initieras på begäran (On-Demand sampling). Dessutom finns det funktionalitet för schemaläggning av samplingssessioner. Övervakningsprogrammet är redan utvecklat och väntas på att tas i bruk. Dock stödjer övervakningen endast sampling av högfrekventa maskinsignaler i nuläget.



Figur 1.2: *Nuläget i RMS-projektet*

DataQ DI-718Bx

Det är en hårdvarukomponent från DataQ som sköter signalsampling av gjuterimaskinen. Varje gjuterimaskin ute hos företagets kunder är ihopkopplad med en sådan för övervakning. Den kan avläsa signaler från en maskin på upp till 16 st kanaler samtidigt. Samplingshastigheten är inställningsbar.

DMM

DMM står för *DISA Moulding Machine* och är alltså själva gjuterimaskinen. En gjuterimaskin från DISA kan tillverka allt mellan himmel och jord. Den kan t.ex. tillverka allt från bildelar som bromstrummor, motorblock och skivbromsar till kaminer och små detaljer som nycklar. Den gjuter i järn såväl som aluminium. Det finns över 200 olika modeller av maskiner i företagets sortiment, stora som små.

eWon

Detta är en router från företaget eWon[2] med säkerhetsfunktioner som erbjuder en säker anslutning och kommunikation mellan företagets RMS-server och företagets kunders maskindatabaser. Tanken är att överföring av varje kunds maskindata till RMS-servern ska ske genom denna hårdvarukomponent. För tillfället överförs inte informationen men det pågår utveckling inom området så det kan antagas att informationen finns på plats i RMS-servern.

VDU

Varje gjuterimaskin är utrustad med en VDU (Visual Display Unit). En VDU är gjuterimaskinens operatörspanel som kommunicerar med PLC-styrsystemet. Genom VDU kan operatören göra inställningar för att styra maskinen och reglera dess gång.

CIM-databas

Varje gjuterimaskin är dessutom utrustad med CIM-databas. Det är en databas som lagrar två typer av värden. Ena typen av värde är maskininställningsparametrar som konfigureras på VDU:n för att maskinen ska köras på ett visst sätt. T.ex. parametrar för att maskinen ska producera en viss form för en typ av produkt. Det är alltså lagrade värden av maskininställningsparametrar som har använts av maskinen under drift. Den andra typen av värde är mer av statistisk natur. T.ex. hur många formar har försökts att produceras, hur många lyckade formar har producerats, hur lång var väntetiden på att sand skulle beredas, hur lång var väntetiden på att järn skulle beredas, hur många nödstopp har skett etc. Hädanefter kommer dessa värden att gemensamt kallas för CIM-värden. Databasen använder en Sybase databashanterare och är installerad på en PC bredvid gjuterimaskinen.

[2]Företagets hemsida: `http://www.ewon.biz/`

RMS-server

RMS-servern utgör den centrala servern i RMS projektet och finns placerad i Danmark på DISA:s huvudkontor i Herlev. Servern samlar alla företagets verksamma maskiners data. Den innehåller en MySQL-databashanterare med en databas för varje gjuterimaskin innehållande signalinformation för högfrekventa och lågfrekventa maskinsignaler, CIM-värden, lagrade värden efter signalsampling m.m. Servern har också en databas för antalet kunder med kontaktinformation och maskininformation. RMS-servern är det system som STUI med hjälp av IAE (Information and Analysis Engine) kommer att inhämta information ifrån för organisering, analys och presentation.

DasyLab

Det är ett avancerat program för signalanalys. Det är som sagt serviceteknikernas huvudverktyg för diagnostisering av maskiners status och det är därför viktigt att STUI görs kompatibelt med detta verktyg. Med kompatibilitet menas att den information (främst högfrekventa maskinsignalers data) som hämtas från RMS-servern m.h.a. STUI måste kunna exporteras till ASC-format (formatet som DasyLab använder sig av) så att informationen kan analyseras i DasyLab.

RDA

Förkortningen står för *Remote Diagnostic Access* och beskriver DISA:s koncept för att fjärrfelsöka gjuterimaskiner som finns ute hos kunder från huvudkontoret i Herlev. RDA omfattar en godtycklig PC på vilken video och audio kommunikation kan ske m.h.a. headset och videoprogramvara. Servicetekniker kommunicerar med kunden och vägleder denne med instruktioner för hur en maskin ska diagnostiseras.

Rapportgenerator

En del i RMS-projektet som tillåter generering av rapporter utifrån informationen som finns lagrad i RMS-serverns databaser. Den omfattar ett användargränssnitt för serviceteknikerna i vilket de kan författa maskinstatusrapporter som sedan skickas ut till kund för information om hur bra deras maskin fungerar. Rapportgeneratorn existerar inte i nuläget men parallell utveckling med STUI sker inom området. Generatorn är ett externt system som IAE måste leverera information till. Det innefattar information som t.ex. produktionsstatistik och stopptider.

SMS-utskickare

Det är också en del i RMS-projektet. Utskickaren skickar ut rapporter likt de rapportgeneratorn genererar fast i mer avskalad form och direkt ut till förmän ute på produktionsgolvet. Det är ytterligare en del som inte finns i nuläget men som håller på att utvecklas. IAE måste därför också kunna kommunicera med denna komponent.

2 Utveckling

2.1 Översikt

I följande kapitel kommer det att diskuteras om hur utvecklingen i projektet gick till. Kapitlet omfattar framförallt avsnitt om följande komponenter:

Informationsmotorn är den komponent som hämtar information på ett enhetligt sätt från RMS (Remote Monitoring System) databasen.

Analysmotorn är den komponent som utför analys på hämtad information av ovanstående komponent i syfte för att undersöka så att t.ex. en hämtad signal håller sig inom givna gränsvärden.

Användargränssnittet är den komponent som presenterar den information som finns i RMS, resultat från analyser utförda av analysmotorn samt kan konfigurera reglerna[1] som skall gälla för existerande analyser.

Men det kommer även tas upp saker från projektet som t.ex. vilken utvecklingsmodell användes, vilka arbetsmetoder utnyttjades och hur fungerade testningen.

Examensarbetets titel syftar på en plattform som heter IAE som står för Information and Analysis Engine. Detta är ett samlingsnamn för informations- och analysmotorn. Att skapa ett samlingsnamn som inkluderar bägge motorerna har att göra med att de hör ihop och kompletterar varandra även om bara analysmotorn är beroende av informationsmotorn och inte tvärtom.

Under examensarbetets gång har examensarbetarna fortlöpande haft kontakt med deras handledare på DISA och har på så sätt införskaffat sig den information som behövts för att besvara frågor som dykt upp under examensarbetets gång. DISA har inte haft specifika krav på lösningen. DISA har utryckt en vilja att examensarbetarna skulle skapa en lösning som löser en del av det övergripande målet d.v.s. att skapa ett fungerande RMS-system. Att endast ha RMS som vägledare var för att främja innovation och skapa en lösning som innehåller funktioner som kanske DISA inte själv tänkt på.

[1] Analysmotorn analyserar information baserat på regler och parametrar. Dessa regler beskriver t.ex. gränssvärden. Mer i detalj om detta kommer under kaptiel 3.

2.2 Utvecklingsmodell

Under utvecklingen i projektet användes en egen typ av utvecklingsmodell. Denna var speciellt anpassad till projektet. Den egna utvecklingsmodellen valdes dels för att projektets fokus inte låg på att öva med att arbeta enligt befintliga och traditionella utvecklingsmodeller utan det låg på de punkter som togs upp under avsnitt 1.1.4 på sidan 14 och dels för att den passade projektet bättre än övriga modeller som fanns att tillgå.

Den använda utvecklingsmodellen bestod av ett antal steg som genomfördes i följande ordning:

1. Undersökning av problemet
2. Skapa sig en målbild
3. Konstruktion av högnivådesign
4. Utvärdering av högnivådesign
5. Implementering
6. Utvärdering av implementation
7. Kontinuerlig testning

Om man under utvärderingen av högnivådesignen kom fram till att den inte fungerade eller för övrigt inte ansågs lämplig, valde man att omkonstruera designen genom att börja om på nytt.

Ifall man under utvärderingen av implementeringen upptäckte att något fattades, kompletterade man implementationen med det som saknades. Om det istället upptäcktes att det inte gick att genomföra en implementation av en viss funktion enligt den påbörjade högnivådesignen, gick man tillbaka till designen och gjorde ändringar.

2.3 Arbetsmetoder

De arbetsmetoder som utnyttjades i projektet var följande:

- Dokumentation av kod
- Förundersökningar
- Frågeställningar
- Analys av nuläget
- Litteratursökning
- UML-design
- Utveckling
- Testning
- Skarpa tester i Danmark

En stor del av den information som krävdes för att utveckla produkten fanns endast på Internet. Examensarbetarna valde att använda en teknik som är beskriven i förstudien under avsnitt 1.2.1 på sidan 24 som heter WPF för konstruktion av användargränssnittet. Denna teknik är fortfarande i ett ungt stadium och även om den är ett fullgott alternativ för användargränssnittskonstruktion är den inte komplett. En stor del av problemmen har varit prestanda och detta verkar Microsoft veta om då prestanda har haft stort fokus under utvecklingen av nästa version av .NET Framework. Under tiden examensarbetet pågick blev .NET Framework 4.0 stabilt enligt Microsoft; den var i betastadium när arbetet påbörjades. Denna version av .Net ramverket innehåller de förbättringar gällande prestanda i WPF som tidigare nämndes men även många andra tillägg. Eftersom arbetet påbörjades i .NET Framework 3.5 då detta var stabilt så valdes det att inte byta till .NET Framework 4.0 p.g.a. risk för kompatibilitetsproblem, buggar och nyinlärning av den nya versionen av utvecklingsmiljön (IDE) Visual Studio. Det finns inget som hindrar ett byte vid vidareutveckling.

2.3.1 Dokumentation av kod

Implementationen av programmet präglades av noggrann koddokumentation. Det vill säga alla klasser, klassmedlemmar, namnutrymmen etc. dokumenterades med hjälp av dokumentationskommentarer medan de skrevs för att underlätta för användare av koden. För att göra det mer användarvänligt att läsa dokumentationen av all kod följer det med en automatiskt genererad dokumentationsfil av dokumentationskommentarerna. Denna fil erbjuder ett webbliknande gränssnitt för presentation av dokumentationen. Ett exempel på hur denna dokumentation ser ut kan hittas under avsnitt 3.3.2 på sidan 74.

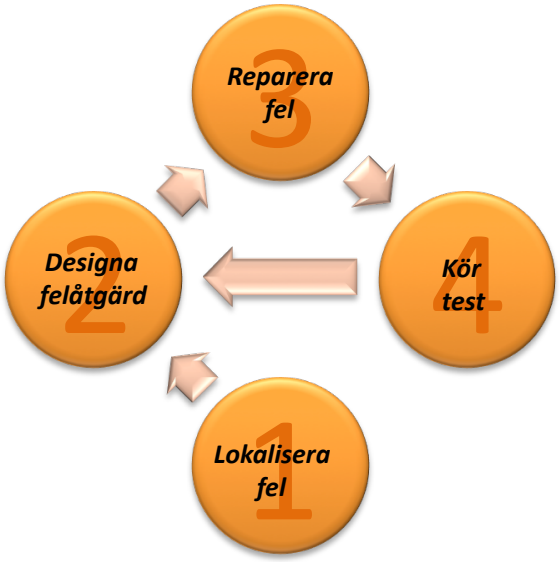
2.3.2 Testning

Testningsfasen utav utvecklingen i projektet delades upp i olika typer av tester, där ett test tillhörde en viss typ beroende på vilken programarkitekturnivå testet i fråga tillhörde. Den första typen av test som utfördes när ett programmeringsproblem höll på att lösas var *enhetstestning*. Enhetstestning befinner sig på den lägsta nivån av tester. Det är tester som testar den allra minsta byggstenen av ett programmeringsproblem. Den allra minsta byggstenen av ett programmeringproblem och den lägsta nivån i programarkitekturen är klassnivån. Det var på denna nivå som enhetstestning gjordes. Med andra ord gick det ut på att testa att de skrivna klasserna gjorde rätt saker.

För att utföra enhetstesterna utnyttjades ett verktyg som heter *NUnit*. NUnit är ett enhetstestningsramverk för alla .NET kompatibla programmeringsspråk. Det är ett verktyg som gör det möjligt för .NET-programmerare att få samma funktionalitet som *JUnit* erbjuder för Java-programmerare. Därför är det mycket likt JUnit och man konstruerar enhetstester i NUnit på ett väldigt snarlikt sätt som i JUnit. Med verktyget konstruerar man tester som testar en klass med dess tillstånd och medlemsmetoder, d.v.s. att dessa gör rätt saker. Testerna skrivs som enkla klassmetoder i en speciell testklass som har till uppgift att testa en speciell programmeringsfunktion. I testklassen finns även initieringsmetoder samt upprensningsmetoder som körs före respektive efter varje testmetod. Inför ett test kan det nämligen behöva göras en del tillståndsinställningar på objektet som man testar innan man är i rätt läge där omständigheterna är sådana som man vill att objektet ska testas i. Efter ett visst test kan det också behöva köras en metod som frigör de resurser som man har använt sig av under testet och det görs med

upprensningsmetoder. NUnit är ett mycket kompetent verktyg som i sin senaste inkarnation, version 2.5.5, ger stöd för .NET version 4.0. Mer om NUnit kan läsas på [7].

Man använde NUnit till att skriva testklasser som kontrollerade att en klass och dess metoder följde sina givna specifikationer. Det kunde t.ex. vara att en metod endast tog emot specificerad input och att en viss specificerad åtgärd vidtogs ifall felaktig input mottogs, som t.ex. att kasta ett exceptionellt undantag, eller att metoden lämnade ifrån sig förväntad output. Men ännu viktigare att objekttillståndet efter att metoden exekverat var det tänkta. Med hjälp av NUnit utförde man alltså *black-box*-testning där man endast iakttog input och output. Vad som fanns inuti, d.v.s. hur metoden var implementerad, brydde man sig inte om vid själva testningen. När enheter misslyckade med att passera enhetstesterna arbetade man enligt nedanstående debuggingprocess.



Figur 2.1: *Projektets debuggingprocess*

I figur 2.1 visas det att man först lokaliserade felet i metoden i fråga och sedan konstruerade en felåtgärd. Felet reparerades med åtgärden och testet kördes på nytt för att se att åtgärden rättade till felet. Om den inte gjorde det fick man återgå till steg två i processen.

När väl en enhet lyckades passera ett enhetstest, ansåg man enheten vara färdig och gick vidare med implementationen av nästa enhet. När samtliga enheter till slut hade testats färdigt var det dags att testa hela *komponenten*[2] som modellerade det aktuella programmeringsproblemet. Testet av komponenten utgör nästa typ av test i projektets testningsfas, *komponenttestet*. Komponenttestning finns på en nivå ovanför enhetstestning. Med detta test är det viktigt att kontrollera att enheterna samverkar på ett korrekt sätt och tillsammans löser komponentens uppgift det vill säga man testar på en högre nivå än klassnivå, där flera klasser är involverade i ett testfall, och där testklassen som beskriver testfallet inte är knuten till just en enskild klass som vid enhetstestning. Även till detta användes NUnit.

Den tredje typen av test som gjordes var *system-* och *integrationstest*. Integrationstest kontrollerar precis som dess namn antyder interaktionen mellan flera olika komponenter efter det att man har kopplat ihop dem. Det är ett övergripande test för att säkerställa att hela systemet fungerar som det ska och det utfördes ständigt indirekt vid användargränssnittsutvecklingen eftersom det vid nya funktioner av gränssnittet presenterade det underliggande systemets sammanlagda resultat (output).

Enhetstestning utfördes parallellt med utvecklingen, d.v.s. så fort en enhet implementerats testades den. Komponenttestning gjordes så fort en komponent var helt färdigställd men den utnyttjades även som *regressionstestning* allt eftersom nya funktioner tillades i systemet. Detta för att säkerställa att nya funktioner inte hade påverkat äldre funktioner.

Varför gjordes då indelningen av tester? Det var för att man enkelt skulle kunna lokalisera var fel uppstod. Genom att börja med att grundligt testa i små steg, som med enhetstestning, för att sedan metodiskt arbeta sig uppåt i programarkitekturhierarkin, verifierar man att del för del fungerar korrekt och att felet inte ligger där utan att det måste ligga vid den senast tillagda delen. Man ville helt enkelt eliminera risken att inte hitta var ett uppkommet fel låg, vilket är överhängande vid direkttestning av system.

[2]En komponent består av flera enheter som tillsammans samverkar för att lösa programmeringsproblemet/programmeringsuppgiften. Informationsmotorn är t.ex. en komponent.

2.4 Informationsmotorn

Ett sätt för att hämta in information på ett enhetligt sätt från olika datalagringskällor och informationsformat.

2.4.1 Undersökning av existerande information

Första steget med utvecklingen av informationsmotorn var att analysera vad för format och struktur all existerande informationen har. Nedan följer de frågorna som behövde besvaras:

1. Vilka datatyper finns?
2. Vilken typ av information behövs?
3. Hur mycket information handlar det om?
4. Hur snabbt måste systemet vara?
5. Vilka sätt behövs för att söka i informationen?

Examensarbetarna satte upp ett krav för systemet att det skulle klara av information som motsvarar ett kontinuerligt insamlande under tio år. Kravet gör att det handlar om flera gigabyte information d.v.s. i storleksklassen om tiotals miljoner databasrader. Detta ställer krav på att förenkla sökningar och minimera sökningar på rådata[3], eftersom en sökning kan ta en halv minut till en halvtimme beroende på sökningens komplexitet. Att bara läsa all rådata ger inte något av värde utan det är sammanställningar och statistisk analys som ger information som är av intresse för serviceteknikerna. Det som krävs är att skapa tabeller som sammanställer informationen och sedan ser till att innehållet hålls uppdaterat när ny information anländer från kunden.

[3]Icke sammanställd information i databasen, t.ex. produktionsdata för formar.

Informationen som existerar kan delas in i tre huvudsakliga grupper som är lagrade på varsit sätt:

CIM (Computer Integrated Manufacturing) är en databas, den beskrivs under avsnitt 1.2.2 på sidan 28.

LF (Low Frequency) omfattar lågfrekvent[4] samplade mätvärden för bland annat olika temperaturer. Den viktigaste LF-signalen är oljetemperaturen för hydralikpumpen eftersom denna komponent är en av de dyraste komponenterna i maskinen.

HF (High Frequency) omfattar högfrekvent[5] samplade mätvärden under t.ex. 2 minuter av en gjuterimaskins arbete.

All information kan ges tillbaka i form av ett flertal sammanhängande variabler i en uppsättning (eng. *tuple*) eller som det vardagligt kallas: en *rad*. Samtliga grupper av information har en relation till tid och dessutom så är en mycket stor del av informationen av numerisk natur. Då all information har relation till tid så kan den största prestandavinsten fås genom att man avgränsar med avseende på tid. Sammanställningar av främst medel, minimum, maximum av mätvärden är mycket intressant. Att använda t.ex. sammanställningar av minimum och maximum kan användas för att göra snabba undersökning om en signal håller sig inom givna gränsvärden. information innehåller bl.a. datatyper såsom heltal, reella tal, texter och datum/tid.

De tre olika typer av information som finns är lagrade på tre olika sätt och eftersom exakt vilka frågor som behöver ställas är okänt[6] så kräver detta en inhämtningsarkitektur som ger tillbaka informationen på ett enhetligt sätt samt kan anpassas till att tolka information lagrade på olika sätt. Då exakt vilka frågor som behöver ställas är okänt och inte går att fullständigt utreda kräver detta också att arkitekturen är utökningsbar. Av erfarenhet och undersökning på företaget vet examensarbetarna att det finns mycket mer information att hämta och det är otänkbart att den informationen lagras på ett annat sätt. I och med detta så har därför examensarbetarna tagit höjd för att stödja fler än tre sätt att hämta in information på och gjort konstruktionen helt generell.

[4] I sammanhanget handlar det om ett värde per minut t ex

[5] Det handlar om t ex 300 mätvärden per sekund eller snabbare.

[6] Har att göra med att serviceteknikerna på DISA aldrig har använt sig av verktyg som utför statistik på denna nivå och helt enkelt inte vet vilken form av statistik som kan vara intressant.

När valet av arkitektur gjordes så dök alternativet upp om att det hade varit en fördel att använda ADO.NET databasinhämtningsarkitektur då den erbjuder en komplett lösning för inhämtning via dess `DataReader` och ett sätt för att hämta information offline; `DataSet`. Informationen som finns i databasen ligger lagrad på ett sådant sätt att inte allt kan uttryckas med SQL-frågor eller är otroligt komplext och därmed också långsamt, så därför gav detta två alternativ:

1. Att utveckla en fullständig ADO.NET drivrutin som inte var tänkt att användas för icke SQL baserade databasmotorer.
2. Att bygga ett lager ovanpå ADO.NET.

Att bygga ett lager ovanpå ADO.NET var med hänsyn till komplexitet, minneseffektivitet och prestanda ett bättre val.

Då examensarbetet påbörjades fanns endast HF-mätvärden. LF-mätvärden och CIM-värden har lagts till under examensarbetets gång.

2.4.2 Koncept

För att lösa problemet med flexibel informationsinhämtning behövdes en målbild. En bild så att det går att visualisera sig vad som behöver byggas och hur detta skall byggas.

Första utkastet av informationsinhämtning

Det gällde att få olika typer av information, olika sätt att läsa in information på samt olika sätt att filtrera och göra sökningar på information att fungera. Exakt vilka funktioner som behövdes från början var svårt att veta och därför behövdes ett system som var flexibelt, utökbart men där komplexiteten hålls på en acceptabel nivå så att det är relativt enkelt att bygga ut.

Som tidigare nämnts kan all information hämtas som en uppsättning av sammanhängande variabler. Det kan finnas många uppsättningar. Några av variablerna i en uppsättning är primära, d.v.s. de är unika för den uppsättningen och kan användas för att hitta samma information igen.

Den bästa lösningen var att bevara den befintliga relationsstrukturen.

Examensarbetarna började med att försöka skapa en definition på vad det som är programmet skall kunna göra. Att skapa en slags matematik för detta gjorde det lättare att se hur strukturen kunde byggas upp.

För att hantera funktioner så var det tänkt att använda en modulär konstruktion där informationsinhämtningsfunktioner läggs i s.k. informationsdrivrutiner. Dessa drivrutiner kan innehålla många funktioner och de beskrivs utförligt senare i rapporten.

Första utkastet såg ut som följande, där varje funktion representerar en generell byggsten:

$$result = process(get(query(mapping())))$$

mapping är en funktion som kan berätta var informationen kan hittas. Då informationen var lagrad på ett relationellt sätt så innebar detta att peka ut vilken tabell och vilka kolumner som innehåller den information som eftersöks.

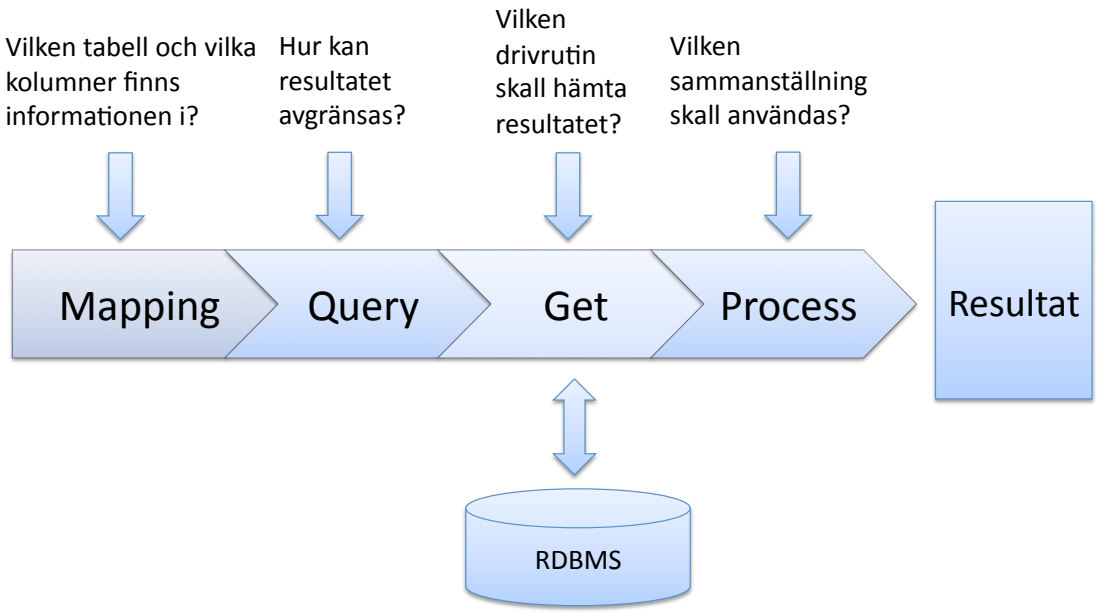
query är en frågefunktion som utifrån vad *mapping* ger tillbaka skapar en SQL-fråga som kan användas av *get* funktionen.

get är en informationsinhämtningsfunktion. Denna funktion kopplar upp mot t.ex. MySQL, kör SQL-frågan skapad av *query* och hämtar hem informationen.

process är en sammanställningsfunktion som kan beräkna t.ex. minimum, maximum och medelvärde av data hämtad av *get*.

Det var valbart att ha en sammanställningsfunktion eftersom viss information skulle kunna komma sammanställd direkt från databasen och då behövs ingen sammanställningsfunktion eftersom den redan är sammanställd. Informationsflödet kan man se som data som flödar från block till block, se figur 2.2.

Detta första utkast var mycket generellt men tillät mycket lite specialisering totalt sett och hade hög komplexitet. Denna lösning baserades på att så att säga koppla ihop tabeller och kolumner och dynamiskt skapa frågor och sedan eventuellt skapa funktioner som behandlar och gör beräkningar på informationen för att slutligen bli sammanställd. Det är något databasmotorn normalt har till uppgift att göra. Problemet var återkoppling; funktionerna behövde kommunicera med varandra för utbyte av information. Lägg därtill att kanske inte all information som kommer att hämtas i framtiden kommer från en SQL-databasmotor. Att



Figur 2.2: *Hur första utkastet av informationsflödet var tänkt att fungera*

lägga in återkoppling skulle teoretiskt vara möjligt men komplexiteten hade ökat och funktionerna hade inte blivit lätt utbytbara då de beror på varandra. Om man inte bygger in återkoppling utan låter varje funktion arbeta för sig själv vilket är tanken så behöver all information som sammanställs faktiskt hämtas från databasmotorn och detta är en långsam process. Det är bättre att låta databasmotorn göra det jobbet.

Det slutgiltiga utkastet av informationsinhämtning

Det slutliga konceptet som valdes för att konstruera en lösning på förenklades så pass mycket att det skapades en enda generell funktion. Det behövdes alltså ingen återkoppling. Denna funktion ger man argument till som sedan när den körs ger tillbaka information. Funktionen kan innehålla databehandling och tunga beräkningar samt kan optimeras internt på relevant sätt för att hämta information. Nedan ges funktionen:

$$result = get(argument_1, argument_2, \cdots, argument_n)$$

Argumenten var från början endast typerna datum/tid, heltal, reella tal. Senare i implementationsfasen så lades det även till stöd för text och resultat från en annan funktion så att systemet klarar av att bygga funktioner som kan göra sammanställning från annan information. Detta gör systemet mer flexibelt än vad första utkastet gjorde. Denna funktion kunde också anpassas till att fungera som en sammanställningsfunktion, d.v.s. ta data från en funktion och omvandla den till en annan form.

Utvärderingen av denna lösning är att specialisering tillåts inom varje funktion, komplexiteten sjunker eftersom det handlar om en enda generell funktion och inte fyra stycken olika som måste kopplas ihop. Vid implementationen blev det lättare att se vad det var som skulle implementeras eftersom målet var tydligare. Det blir också lättare att optimera så att mängden information som måste behandlas vid sammanställning minskas rejält eftersom detta kan byggas in i funktionen och kan vara specifikt för varje funktion. Sammanfattningsvis blir det enklare att implementera, lättare att optimera för hög prestanda och faktiskt mer generellt än det första utkastet även om ambitionen var att göra det mindre generellt så att det blev mer målspecifikt.

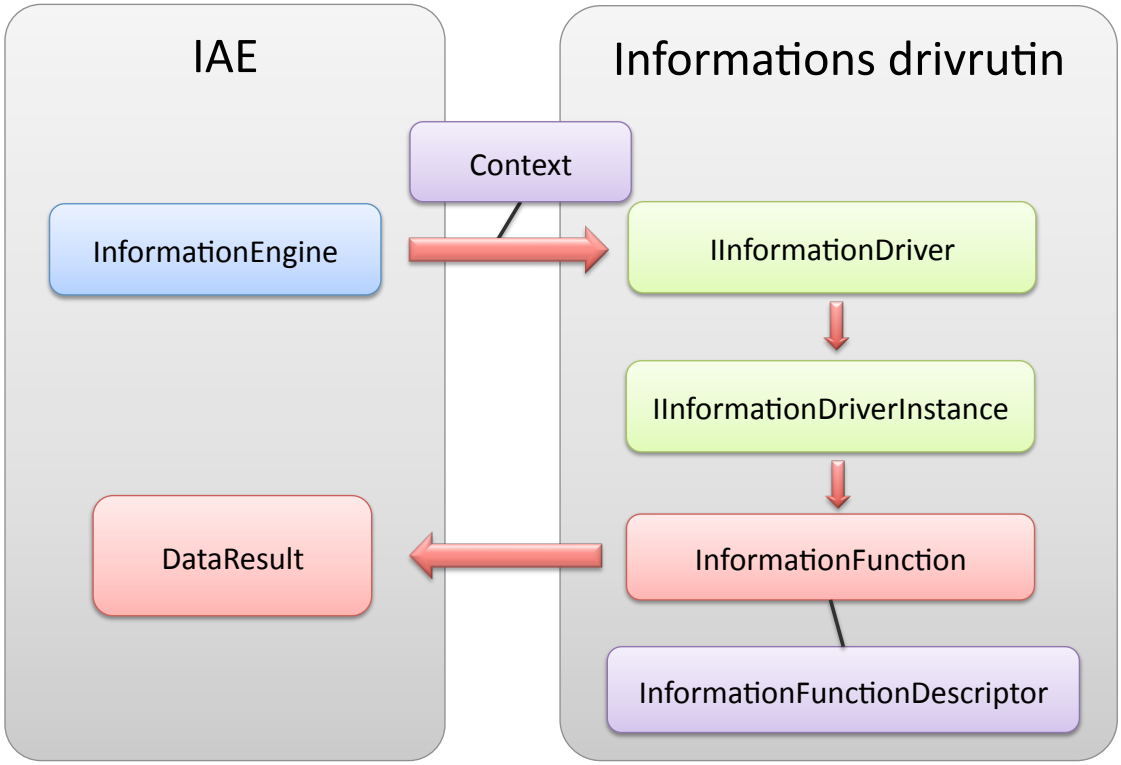
get har nu blivit det som hädanefter kommer att kallas för **informationsfunktion** i rapporten. Det är alltså en funktion som kan behandla och hämta in ny information baserat på ett fåtal definierade parametrar.

2.4.3 Hantering av funktioner

Med ett koncept på hur informationsinhämtningen skulle ske så behövdes också en infrastruktur för att hantera dessa funktioner. Tanken är att dessa funktioner skall vara modulära och ska inte behöva hårdkodas in i strukturen utan ska dynamiskt kunna laddas in. Det valdes då att använda en design pattern som heter *Plug-in*. Detta är ett vanligt design pattern för konstruktion av modulära programvaror vars funktionalitet kan utökas utan att själva huvudprogrammet, *värden*, behöver modifieras. Rent tekniskt innebär det att ett gränssnitt för kommunikation definieras och kod laddas in dynamiskt i värden som sedan kör koden i det externa programmet och på så sätt utökar funktionaliteten. [2]

Plug-in är en generell benämning på mjukvara som kan kopplas på inuti en värd. Eftersom denna lösning har använts på ett specifikt sätt valdes det att kalla alla dessa för *Informationsdrivrutiner* då de innehåller *informationsfunktioner* som driver informationsinhämtaren.

2.4.4 Högnivådesign



Figur 2.3: *En förenklad bild av högnivåstrukturen för informationsmotorn*
Blå färg är startpunkten, röd är aktiva enheter, lila är hjälpklasser och grön färg är gränssnitt som implementeras av en informationsdrivrutin.

InformationEngine är huvudklassen som hanterar alla informationsdrivrutiner. Den hanterar alla instanser av IInformationDriver som finns. Det finns en instans för varje drivrutin som existerar. Dessa laddas in vid start.

Context innebär som namnet säger, kontexten. Den innehåller information om vilken gjuterimaskin som är aktuell och vilken databas som information som ska behandlas finns i. Genom denna kan också drivrutinerna komma åt konfigurationen i programmet. I denna konfiguration finns t.ex. alla SQL-frågor som används en informationsdrivrutin vid namn "SQL Driver", mer om detta kommer under avsnitt 3.3.4 på sidan 79.

IInformationDriver är den klass som symboliserar en informationsdrivrutin. Den innehåller bland annat information om vad drivrutinen heter, vilken version det är och ett sätt för att skapa en `IInformationDriverInstance` som är en instans av drivrutinen.

IInformationDriverInstance är en instans av drivrutinen som kan användas för att skapa informationsfunktioner, d.v.s. `InformationFunction`. Ur denna drivrutinsinstans kan man hämta en lista över alla funktioner som finns i drivrutinen och en fullständig beskrivning av dem via `InformationFunctionDescriptor`

InformationFunction är en basklass för informationsfunktioner, innehåller ett standardiserat sätt för att skicka med parametrar och validera deras typer vilka kan fås via `InformationFunctionDescriptor` som beskriver funktionen. Alla informationsfunktioner ärver denna basklass i informationsdrivrutinen.

InformationFunctionDescriptor beskriver en specifik informationsfunktion. Den innehåller information om datatyp, namn och en beskrivning av varje parameter som funktionen behöver. Även information om vad funktionen ger ut för data beskrivs av klassen.

DataResult hanterar allt resultat given av en informationsfunktion. Figuren visar inte att det finns en direktkoppling till InformationFunction som levererar data. Det finns två sätt att driva en DataResult. Ett är via ett gränssnitt vid namn `IDataResult` och det andra är via ärvning. Valet till att ha ett gränssnitt var för att förenkla implementationen i informationsdrivrutinen.

Figur 2.3 beskriver hur vägen ser ut för att få information ur en informationsdrivrutin. Pilarna symboliserar exekveringsflödet för att skapa en `DataResult` som innehåller resultatet. Bilden är förenklad och visar inte t.ex. att parametrar läggs in i `InformationFunction` innan den körs. Den visar inte de två hjälpstrukturer: `Parameter` och `Output` som beskriver parametrar till resp. utdata från en informationsfunktion.

2.5 Analysmotor

Ett av önskemålen från DISA är att kunna göra en automatisk bedömning av en maskins tillstånd, hur pass bra den fungerar med andra ord. För att göra denna bedömning behövs det en strukturerad och inställningsbar analys. DISA har en vision att om 5 år kunna koppla upp minst 200 gjuterimaskiner och för att kunna få en överskådlig bild av hur maskinerna fungerar behövs automatisk analys.

2.5.1 Undersökning

En del frågor uppkom vid övervägandet av hur analysmotorn skulle byggas upp:

1. Hur kan HF-signaler analyseras?
2. Hur kan LF-signaler analyseras?
3. Vad för information ur CIM kan vara av vikt för analys?
4. Hur kan man specificera hur analysen skall gå till?
5. Hur kan resultaten användas i användargränssnittet?
6. Hur mycket information kan tänkas behövas analyseras?
7. Vilka analysmetoder skulle kunna vara av intresse?
8. Hur kan analyserna automatiseras?

LF-värden är kontinuerliga mätningar över lång tid, HF mätningar ger stötvis med detaljerade mätvärden över en relativt kort period t.ex. 2 minuter och ur CIM-databasen kan man få statistisk information såsom antal formar producerade per dag, vecka, månad och år. Även väntetider för sand, järn m.m. då gjuterimaskinen har behövt stanna upp för att vänta på materialen som behövs skall komma till rätt plats finns i CIM. Allt detta kan hämtas och representeras på olika sätt. Att hämta informationen på ett generellt sätt löstes med informationsmotorn men att sedan göra en analys som ger vettig feedback till serviceteknikerna är uppgiften som analysmotorn skall lösa.

HF-mätningar

Information från serviceteknikerna beskrev att vissa signaler har korrelation till varandra. Dessa signaler kan analyseras för huruvida de följer varandra och hur stor fördröjningen är, om den minskar, ökar eller är lika vilket är korrekt. Det är små förändringar det handlar om, men detta är ett typexempel på en analys som manuellt är mycket tidskrävande att göra och säger mycket om maskinens status.

LF-mätningar

LF-mätningar omfattar t.ex. mätningar av hydraulikpumpens oljetemperatur. Oljan är det som smörjer hydraulikpumpen och för att smörjegenskaperna skall vara som bäst så krävs det att oljetemperaturen är inom ett specifikt intervall. Oljetemperaturen regleras i form av att den värms upp och kyls ner. Oljetemperaturen förändras därför långsamt och går både upp och ner över lång tid beroende på vad som produceras. Denna oljetemperatur är speciellt intressant då hydraulikpumpen är en av de dyraste komponenterna i gjuterimaskinen. Om temperaturen skulle vara fel så kan pumpen förslitas onormalt mycket eller till och med skära sönder med ekonomiska följder som konsekvens. Detta eftersom en reparation kan vara så hög att det motsvarar en helt ny maskin. Att automatiskt undersöka om temperaturen är i farozonen, d.v.s. är utanför angivna gränser är mycket viktigt för att kunna förebygga dyra reparationer samt att ge en mycket bra fingervisning om vilket skick maskinen är i.

CIM-värden

Av erfarenhet vet DISA att maskinen går långsammare med tiden men exakta siffror saknas. Att maskinen går långsammare kan tyda på att någon del fungerar sämre än vad som är tänkt men då hastighetsförminskningen är så pass liten på kort tid är det något som är svårt att upptäcka förrän något fel har uppstått. Att kunna göra en automatisk analys på cykeltiderna[7] och ge en varning om dessa har ökat plötsligt är av intresse för att veta om maskinen fungerar bra. Det finns också annan information såsom fel som har uppstått i PLC-styrsystemet, detta är också mycket intressant för att veta så att allt är i ordning. Det handlar mycket om statistisk analys när det gäller CIM-värden såsom minimum, maximum, medel, trender och andra metoder för att beräkna ett värde som kan vara av relevans för gjuterimaskinens hälsa.

Analysering

Att utföra en analys kan ses som att omvandla information till ett format så att det blir överskådligt och begripligt. Just konceptet med transformering ligger till grund för idéen med att bygga upp en *analysmodell* som har strukturen av ett träd där varje nod är en informationsfunktion som ger information till en annan.

Dessa analysmodeller kan sparas ner och sedan automatiskt köras genom att använda analysmotorn i ett extern program som kör analysen regelbundet och sparar resultaten. Dessa resultat bör sparas på ett enhetligt sätt och kunna visas i ett användargränssnitt.

Prestanda och datamängd

Analysmotorn kan tänkas analysera stora mängder information, mycket större än vad som får plats i primärminnet. I och med detta måste det tas hänsyn till att all data som ska analyseras inte kan sparas i minnet utan måste hämtas vid behov.

[7]Cykeltid för en gjuterimaskin är tiden det tar för maskinen att tillverka en gjuteriform.

Användargränssnittet

Det skapades ett användargränssnittskoncept för hur analysinformationen skulle kunna visas i det fall då analysen visar på svaghet, se figur 2.4. Tänk på att detta var ett koncept för att få en bild över hur det *kan* se ut men inte hur det *behöver* se ut.

Tanken var först att resultatet från analysen skulle visas där t.ex. "Value out of bounds" är en beskrivning som analysfunktionen ger tillbaka. Detta meddelande skulle vara mänskligt läsbart och ska berätta om vad analysen kom fram till. Sedan skulle information om relevant graf som ska visas också följa med samt en referens på vad för data analysen är baserad på så att man kan undersöka var resultatet kom ifrån.

Figur 2.4: *Användargränssnittskoncept för hur analysresultat kan visas*

Framtiden

Att utveckla en komplett lösning för DISA är inte möjligt då omfattningen av detta är mycket större än examensarbetets tidsrymd och omfattning. I och med detta så har mycket fokus lagts för att göra det DISA vill möjligt istället för att faktiskt lösa allt vad automatisk analys innebär (eller kan tänkas innebära). Med denna analysmotor kan alla automatiska analyser göras men det kräver vidareutveckling i nära samarbete med DISA och med utförliga specifikationer från serviceteknikerna, som kan maskinerna i detalj, för att lyckas. Målet för analysmotorn var att ge DISA en grund som man sedan kan bygga ut, d.v.s. att lyckas med ett "proof of concept".

Sammanfattning

Det behövdes ett system som kan ta information från olika informationsfunktioner, behandla informationen på ett sätt så att det inte krävs att analysmotorn måste vara specialkonstruerad för varje bit information som kommer ut ur informationsfunktionerna. Sedan analyseras informationen och ett analysresultat skapas för att ge en fingervisning om huruvida maskinen fungerar rätt, konstigt eller fel på ett sätt som kan översättas så att det enkelt kan visas i ett användargränssnitt.

2.5.2 Koncept

Mycket likt uppbyggnaden av informationsmotorn så var första tanken att använda samma prototypteknik:

$$analysisresult = analyze(get_1(arg_1, arg_2, \cdots, arg_n), \cdots, get_n(\cdots))$$

analyze är analysfunktionen som tittar på resultatet från get_n

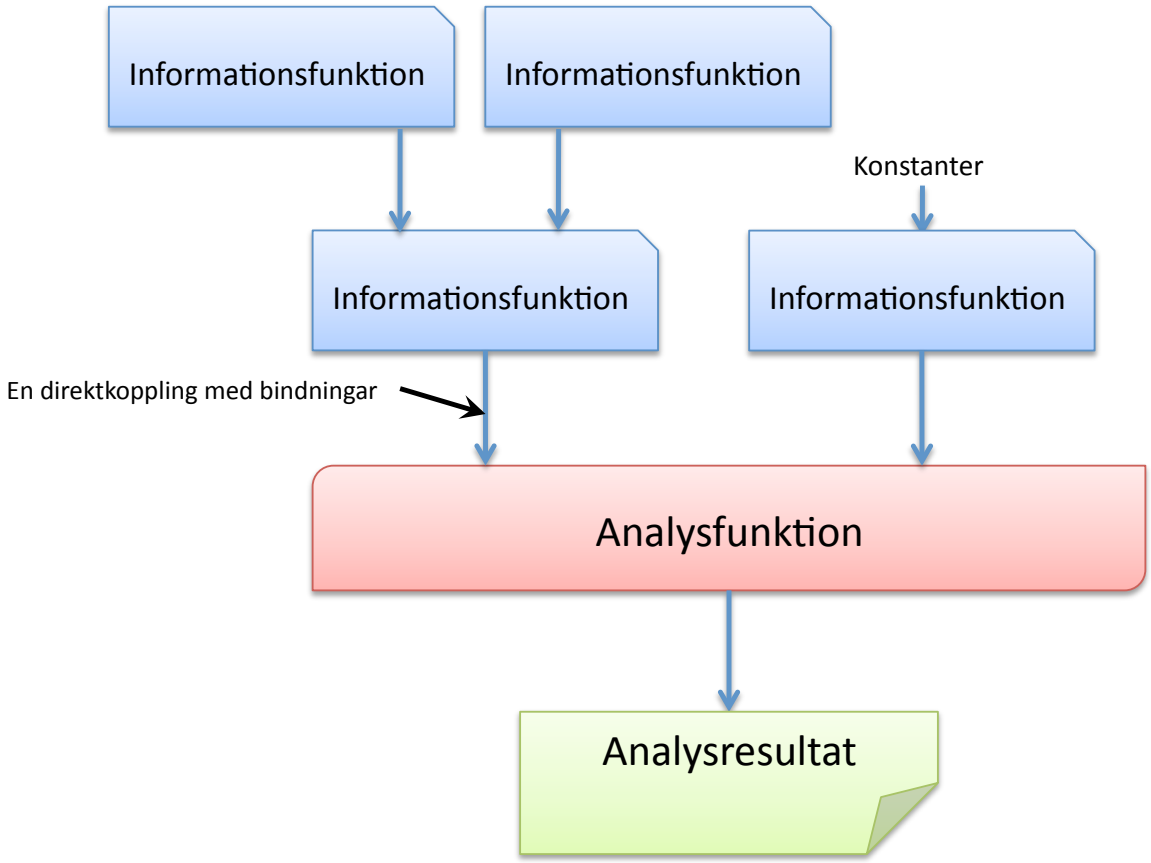
analysisresult är resultatet från en analysfunktion

get är informationsfunktioner som hämtar information

arg$_n$ indata till en informationsfunktion

Denna metod fungerar bra men den beskriver bara hur man kan koppla en uppsättning av specifika informationsfunktioner till en analysfunktion. För att göra systemet flexibelt och utbyggbart utan att behöva bygga nya byggstenar varje gång så behövs ett system där resultatet från en informationsfunktion kan transporteras till en annan funktions ingångar som i sin tur ger information baserat på de inparametrar som gavs.

Denna uppbyggnad blir mer av ett träd där varje nod kan ha n st noder kopplade till sig, se figur 2.5 på nästa sida.



Figur 2.5: *Ett generellt analysträd även kallat analysmodell*

Informationsfunktionen som hämtar information baserat på parametrar. Dessa kan kopplas dynamiskt utifrån en modellbeskrivning. Informationsfunktioner kan omvandla data och göra beräkningar på dem. Detta gör det möjligt att skapa mycket komplexa beräkningar enkelt.

Kopplingen mellan funktioner är det som gör att hela systemet fungerar. Denna koppling ser till att en informationsfunktion och analysfunktion får korrekt indata. Kopplingen är fullt definierad i modellbeskrivningen.

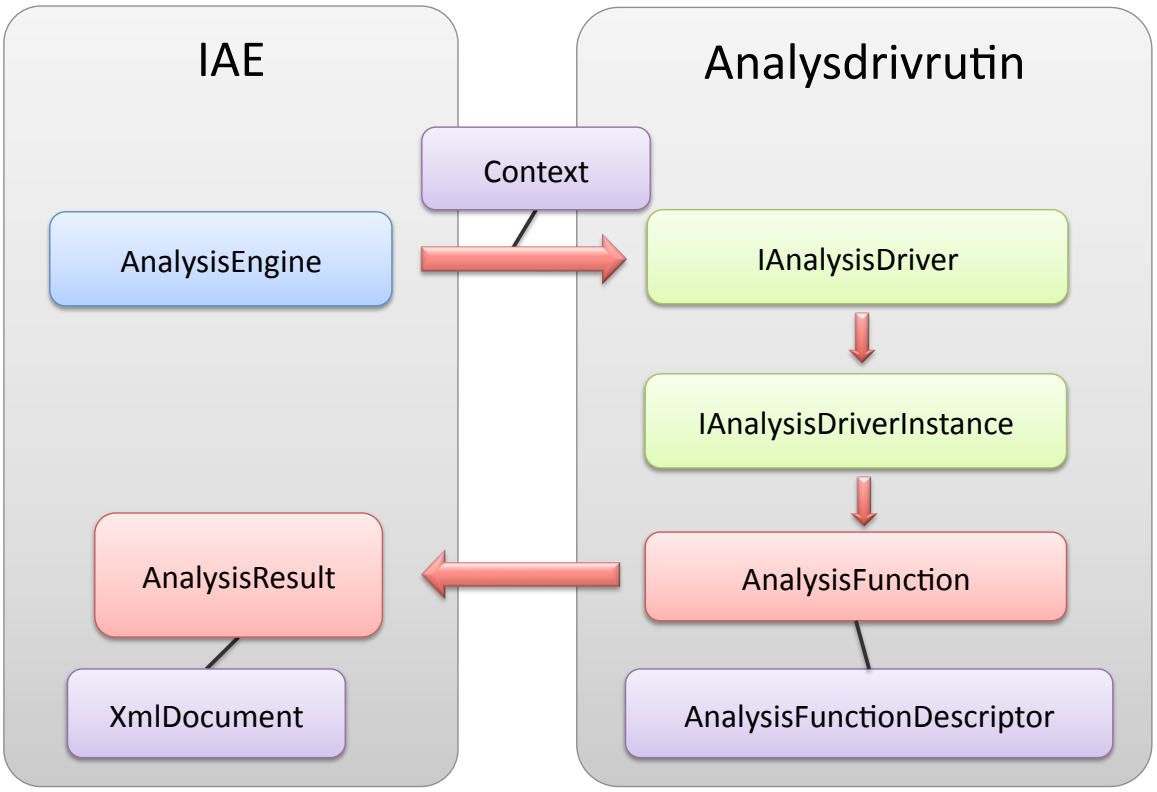
Analysfunktionen analyserar sedan resultatet som bildas m.h.a. trädet.

Analysresultatet beskriver resultatet på ett sätt som kan användas för att visa resultatet i användargränssnittet.

Kopplingarna omvandlas till klasser av typen `DataResult` när analysen väl skall ske. Denna klass används för att läsa data i framåtgående riktning.

2.5.3 Högnivådesign

Det kan vara så att läsaren har upptäckt att inget har nämnts om hur analysfunktionerna hanteras såsom hanteringen av informationsfunktionerna beskrevs under avsnitt 2.4.3.

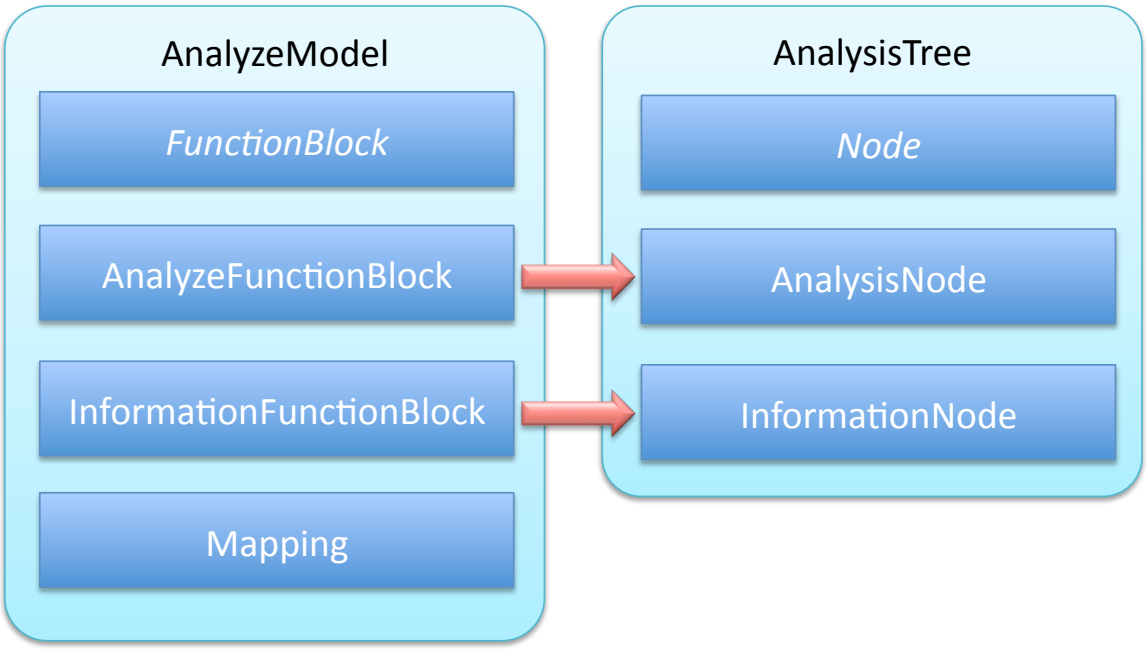


Figur 2.6: *Hur analysfunktioner används, jämför detta med figur 2.3 på sidan 44*

Anledningen är helt enkelt för att upplägget är i princip identiskt fast med utbytta namn, se figur 2.6. Det finns ingen skillnad mellan `IAnalysisDriver` och `IAnalysisDriverInstance` mot informationsmotorns motsvarigheter förutom det faktum att informationsfunktionen nu är en analysfunktion. Den största skillnaden för hantering i analysmotorn är resultatet. Detta resultat ska innehålla en instans av `XmlDocument`. Det är en klass som tillhör .NET ramverket och symboliserar ett XML-dokument. Detta XML-dokument beskriver resultatet av analysen.

Analysmodell och analysträdet

Det som skiljer informationsmotorn från analysmotorn är främst tillägget av en **analysmodell** och generation av ett *körbart* **analysträd**. Skillnaden mellan dessa två är att analysmodellen är en beskrivning av hur informationsfunktionerna skall sättas ihop, beskriver bland annat vilka konstanter som skall användas för parametrar, vilken utdata från en informationsfunktion som skall gå in i en annan o.s.v. Modellen är *beskrivande* och kan därför inte användas för att direkt göra en analys. Analysträdet skapas från analysmodellen och det är den som används för att starta en analys. Det finns en begränsning i analysmodellen och analysträdet och det är att all informationsinhämtning och omvandling endast får landa i en enda slutgiltig analysfunktion som gör själva analysen. Om det skulle vara så att fler behövs så kan man skapa flera modeller och därmed skapas fler analysträd och köra dem efter varandra.



Figur 2.7: *Analysmodellen och analysträdets uppbyggnad*

Se figur 2.7 för att se hur högnivådesignen av analysträdet ser ut. De röda pilarna visar motsvarande instanser av klasser mellan *beskrivande formen* och *körbar form*.

I figur 2.7 har man valt att kalla funktionerna för block resp. noder. Valet att kalla det för block var för att om man skulle se det i ett användargränssnitt så skulle det se ut som ett fyrkantigt block, alltså det är endast beskrivande. Noder här är aktiva och data flödar mellan dem.

AnalyzeModel symboliserar själva analysmodellen och utför omvandlingen från en **analysmodell** till ett **analysträd**.

AnalysisTree är själva analysträdet. Den utför själva analyseringen och innehåller det som behövs för informationsinhämtning. Den innehåller t.ex. det som behövs för att hämta hem all information via informationsfunktionerna.

FunctionBlock är en abstrakt basklass för hantering av blocken och innehåller basstrukturen för hur funktionerna skall användas. Till exempel definierar denna att det skall finnas mappningar mellan blocken.

InformationFunctionBlock representerar en informationsfunktion. Den innehåller bland annat vilka konstanter som valts och genom basklassen som beskrivs ovan hur den skall kopplas ihop med en annan funktion. Den innehåller också metoder för att skapa en instans av informationsfunktionen, dessa metoder används vid skapandet av `AnalysisTree`

AnalysisFunctionBlock är själva analysfunktionen och det får endast finnas en av denna i en analysmodell p.g.a. begränsningar som valts att göras för att minska komplexiteten i lösningen. En analysfunktion kan också ha konstanter och de beskrivs i klassen.

Mapping är den del som beskriver hur utdata från en informationsfunktion kan kopplas till en annan funktion. Kopplingen kan ske till både en informationsfunktion och en analysfunktion.

Node är motsvarigheten av `FunctionBlock` i analysträdet. Det är en klass som motsvarar en nod i ett träd och kan ha n st barn kopplade till sig.

InformationNode är motsvarigheten av `InformationFunctionBlock` fast i körbar form. Den besitter en körbar instans av informationsfunktionen.

AnalysisNode är motsvarigheten av `AnalysisFunctionBlock` fast precis som ovan i körbar form.

`FunctionBlock`, `InformationFunctionBlock`, `AnalysisFunctionBlock` och `Mapping` är alla nästlade klasser inuti `AnalyzeModel`, vilket också kan ses i figur 2.7. På samma sätt är `Node`, `InformationNode` och `AnalysisNode` nästlade klasser i `AnalysisTree`. Det valdes att göra på detta sätt eftersom de respektive grupper av klasser är en inre del av implementationen och om de används utanför ska det vara tydligt att de tillhör de inre mekanismerna som finns.

2.6 Presentation & användargränssnitt

2.6.1 Bakgrundsarbete

I samband med utvecklingen av användargränssnittet upptäcktes det att gränssnittet hängde sig efter att man hade beordrat det att utföra en viss uppgift, t.ex. att hämta in information från databasen om en speciell HF signal för presentation av denna. Användargränssnittet slutade helt att svara på input medan det höll på med uppgiften som det tidigare blivit ombedd att utföra ända till det att uppgiften blivit slutförd. Då började det att reagera på vad användaren bad det att göra. Som användare kunde man alltså inte förstora fönstret, bläddra bland menyer, mata in data i textrutor etc. så länge programmet var upptaget med en tilldelad uppgift. En uppgift som att hämta in information om HF signaler är en mycket långsam och prestandamässigt krävande uppgift (speciellt för databasen) eftersom det kan röra sig om flera hundratusentals databasrader. Databaskopplingen blir då snabbt en flaskhals och medför en helt oacceptabel väntetid för användaren.

Ganska snart insåg man att det hela handlade om i vilken sekvens instruktionerna exekverade. Alla instruktioner exekverade nämligen i en enda tillgänglig sekvens, både GUI-instruktioner och underliggande programfunktioner, då man egentligen hade behövt flera parallella exekveringssekvenser som kunde köra samtidigt. Lösningen på dilemmat var att utnyttja trådning. Som det till en början var konstruerat, kördes allting på en och samma tråd; *huvudtråden*. För att tillåta att GUI-instruktioner skulle kunna exekvera samtidigt som instruktioner från informationsmotorn eller analysmotorn krävdes det en uppdelning av instruktioner på flera trådar; en självständig tråd (huvudtråden) som kunde sköta GUI-instruktioner, och en ny tråd för varje ny uppgift som användaren tilldelade programmet m.h.a. användargränssnittet. På så vis blev tråden för användargränssnittet alltid tillgänglig för input från användaren.

.NET FCL innehåller ett helt komplett namnutrymme, `System.Threading`, med en mängd klasser och gränssnitt för hantering av multitrådad programmering. Men med flertrådad programmering stiger komplexiteten i programmeringen eftersom svårigheten med att hantera jämlöpande och samtidigt åtkomst till delade och skyddade resurser tillkommer. Det är t.ex. lätt hänt att ens program hamnar i ett dödläge där ingen tråd får tillgång till den skyddade resursen och då fastnar programmet i all evighet i det låsta läget. Därför valdes det att utnyttja en smidigare typ av lösning på trådproblemet; användningen av `BackgroundWorker`-klassen, vilken underlättar och förenklar uppgiften att skapa bakgrundsarbete vid användargränssnittsutveckling.

`BackgroundWorker` återfinns i namnutrymmet `System.ComponentModel`. Den medför en rad fördelar jämfört med att själv hantera trådarna. Bland annat har den stöd för notifikation av bakgrundsarbetets framåtskridande och fullständigt avbrytande av bakgrundsarbetet vid vilken tidpunkt som helst. Den ger alltså trådarna med bakgrundsarbete möjlighet att rapportera tillbaka status för hur långt arbetet har kommit, vilket man i sin tur kan använda som feedback till användaren genom en förloppsmätare i statusfältet samt att huvudtråden m.h.a. en användares input kan avbryta bakgrundsarbetet när som helst utifall det skulle bli för tidskrävande[8]. Men den absolut största fördelen med `BackgroundWorker` är att den erbjuder ett väldigt enkelt gränssnitt för att starta upp och avsluta trådar samt att hantera synkronisering mellan trådar utan att dessa samtidigt ändrar en skyddad resurs[9] så att något blir fel. En annan fördel med klassen är att den förhindrar att dödlägen (eng. deadlocks) uppstår eftersom den på ett korrekt sätt låser och låser upp skyddade resurser för trådar. På grund av de två senare fördelarna är användningen av `BackgroundWorker` det enklaste och säkraste sättet för flertrådad programmering och också huvudskälet till varför den valdes som lösning på trådproblemet.

Under utvecklingen av användargränssnittet använde man sig av `BackgroundWorker` genom att skapa en ny instans av klassen för varje bakgrundsarbete som skulle utföras och knyta deras händelser `DoWork`, `ProgressChanged` och `RunWorkerCompleted` till lämpliga händelsehanterare. Det är nämligen så att när man startar en `BackgroundWorker` med `BackgroundWorker.RunWorkerAsync()`, så väljer den en fri tråd att exekvera på från trådpolen som CLR hanterar, och avfyrar händelsen `DoWork`

[8]Båda funktionerna användes flitigt under användargränssnittsutvecklingen.

[9]Skyddade resurser som kan bli låsta i all evighet eller bli osynkroniserade i detta samhang är användargränssnittskontroller.

från denna tråd. Händelsen fångas upp av händelsehanteraren (metoden) som man tidigare associerade med händelsen och instruktionerna inuti händelsehanteraren fungerar som bakgrundsarbete. Därför förpassades alla databasanslutningar med hämtning av information från databasen och övriga beräkningar som behövde göras till händelsehanteraren för `BackgroundWorker`-händelsen `DoWork`. I och med detta tilläts allt bakomliggande användargränssnittsarbete att köras på separata trådar och användargränssnittet kunde hållas mottagligt för användarinteraktion. När slutligen bakgrundsarbetet slutförs, avfyrar `BackgroundWorker` händelsen `RunWorkerCompleted`, vilken fångas upp med den tidigare knutna händelsehanteraren vari de skyddade användargränssnittsresurserna (grafdiagram, listvyer, stapeldiagram etc.) kan uppdateras säkert. [3, 9, 4, 5]

2.6.2 Förundersökning av grafikkontroll

DISA uttryckte önskemål för ett presentationsverktyg för sina gjuterimaskiners signaler och maskindata. Signalerna var tänkta att visas i signalgrafer medan maskindata presenteras i lämpligt statistikdiagram. För detta ändamål inleddes en förundersökning med målet att undersöka om det fanns några befintliga grafiska kontroller som uppfyllde företagets önskemål på presentation och som kunde utnyttjas.

En hel uppsjö alternativ fanns att tillgå. Men dessvärre visade det sig att de flesta var avgiftsbelagda och endast en liten skara var gratisalternativ som var i stabila versioner och erbjöd det som eftersöktes. Med anledning av detta och att man ville spara utvecklingstid av en egenutvecklad grafisk diagramkontroll, valdes initialt Microsofts *.NET Windows Forms Chart Control* [6] för presentation av maskindata och maskinsignaler.

2.6.3 Maskinstatistik

Vid presentationen av en gjuterimaskins väntetider uppstod ett problem gällande uppdelningen av väntetider per tidsintervall. DISA hade som önskemål att kunna se väntetider för en maskin totalt sett per år, per månad, per vecka och per dag för de två senaste månaderna för att på så sätt få en överblick över varje maskins väntetider, vilket underlättar diagnostiseringen av en maskins status. Problemet var att DISA ville att väntetiderna skulle presenteras per år, per månad, per vecka och per dag medan informationen i databasen var lagrad per produktionsskift,

där varje produktionsskift inte bara höll sig inom en dag utan kunde sträcka sig över flera dagar och dessutom inte var lika långt varje gång det startade. Det innebar att det inte gick att göra direkta sökningar i databasen för att sedan presentera väntetiderna i användargränssnittet per tidsintervall utan någon form av uppdelning av väntetiderna per dag erfordrades.

För att lösa problemet fick man helt enkelt införa viktning, i vilken man beräknade hur stor andel väntetid som motsvarades av varje dag. Om t.ex. ett produktionsskift började den 22 september 2008 kl. 12:01 och avslutades den 23 september 2008 kl. 04:44 och hade en väntetid på järn som motsvarade 70 min, så varade produktionsskiftet i 16 h och 43 min varav 11 h och 59 min utgjordes av skiftets startdag och 4 h och 44 min utgjordes av skiftets slutdag:

$$11\text{h}59\text{min} + 4\text{h}44\text{min} = 16\text{h}43\text{min}.$$

När de olika dagarnas skifttid samt den totala skifttiden hade beräknats, räknade man om alla tider till sekunder:

$$\begin{aligned} 11\text{h} \cdot 60\text{min} \cdot 60\text{s} + 59\text{min} \cdot 60\text{s} &= 43\,140\text{s}, \\ 4\text{h} \cdot 60\text{min} \cdot 60\text{s} + 44\text{min} \cdot 60\text{s} &= 17\,040\text{s}, \\ 16\text{h} \cdot 60\text{min} \cdot 60\text{s} + 43\text{min} \cdot 60\text{s} &= 60\,180\text{s}. \end{aligned}$$

För att slutligen beräkna hur många minuter av de 70 minuterna som maskinen fick vänta på järn som tillhörde respektive dygn:

$$\begin{aligned} \frac{43\,140}{60\,180} \cdot 70\text{min} &\approx 50{,}18 \text{ min hörde till startdagen} \\ &\text{och} \\ \frac{17\,040}{60\,180} \cdot 70\text{min} &\approx 19{,}82 \text{ min tillhörde slutdagen.} \end{aligned}$$

Dessa väntetider adderades sedan till dygnens väntetider för järn. När dagsuppdelningen hade blivit klar var det sedan enkelt att sammanställa väntetiderna för resterande tidsintervall; år, månad och vecka.

2.7 Skapade utvecklingsverktyg

Då mycket information skulle komma på rätta platser och infrastrukturen för överföringen inte var utvecklad då IAE utvecklades så har ett utvecklingsverktyg som har använts av examensarbetarna skapats. Dessa är värda att nämna då de har behövts vid undersökning och tagit tid att utveckla.

2.7.1 RMS Synthesizer

För att kunna testa systemet fullt ut samt för att kunna hitta och lösa prestanda problem, testa SQL-frågor och ha värden att söka i så skapades ett utvecklingsverktyg som internt kallades för *RMS Synthesizer*.

Detta verktyg kan:

- Syntetisera LF signaler som kan användas för test av analys- och informationsmotor. Dessa LF signaler skapas med hjälp av kubisk interpolering och slumpgeneration.
- Kopiera över CIM databas specifikation dvs skapa tabeller med motsvarande specifikationer utifrån de cim moduler som behövs och kopiera över allt innehåll mellan två olika databas motorer nämnligen från Sybase SQL Anywhere till MySQL.
- Kopiera över HF signaler uppmätta med bakgrundsprocessen innan stöd för RMS var byggt.
- Skapa hela RMS databasstrukturen med alla procedurer, materialiserade vyer, tabeller och databaser.
- Skapa de sammanställningstabeller som behövs och initiera dem på rätt sätt.
- Har en uppbyggnad som möjliggör tung datahantering på ett enkelt sätt.

Verktyget har använts för att föra över de 800 MB exempeldata som praktiska tester är baserade på. Detta verktyg var och är också oumbärligt för att snabbt skapa en korrekt databas struktur för RMS databassystemet. Grunden som skrevs i detta programmet har också sedan används som bas för att skapa CIM värdes överföring i bakgrundsprocessen.

LF syntetisering

En av uppgifterna för RMS Synthesizer var som tidigare nämndes, att syntetisera LF signaler. LF signaler behövde genereras på syntetisk väg eftersom det var data som inte fanns tillgänglig för inhämtning och analysering av IAE till en början[10]. IAE behövde nämligen datan för att dels man skulle kunna verifiera att den delen av informationsmotorn som hämtar in LF signaler faktiskt fungerade som den skulle och dels för att analysmotorn skulle ha data att arbeta med som efterliknade de verkliga signalerna så att analysen skulle bli så korrekt som möjligt.

För att syntetisera LF signaler användes slumptalsgenerering i kombination med kubisk interpolering. Slumptalsgenerering utnyttjades för att erhålla olika värden som signalen kunde anta. D.v.s. genereringen användes för att uppnå oförutsägbarhet hos signalen. Man ville ju inte att signalen skulle bete sig statiskt, vilket en signal från en vanlig talgenerator som genererar tal enligt ett givet mönster skulle göra, utan att den skulle ha liv i sig likt en verklig maskinsignal.

"Livet" i signalen som slumptalsgenereringen medförde ville man behålla, men samtidigt ville man få den att se mindre dramatiskt slumpartad ut och mer mjuk i sina övergångar mellan olika tal. Detta för att signalen skulle te sig mer realistisk. En slumptalsgenerator kan nämligen emellanåt vara för häftig i sin generering av tal så att dess output förefaller sig onaturlig. T.ex. kan två tal hamna i varsin ände av slumptalsintervallet och då blir det en väldigt brant övergång mellan talen. För att uppnå en mer naturlig signal med mjukare övergångar som inte är så branta utnyttjades kubisk interpolering.

Interpolering i allmänhet är ett sätt för att mjuka ut en samling värden så att dess övergångar inte blir så branta genom att helt enkelt lägga till extra tal mellan de befintliga värdena. Den enklaste formen av interpolering är linjär interpolering, där interpoleringsfunktionen tar tre parametrar som input, parametrarna a, b och x, där de två första är värdena som det ska interpoleras mellan (d.v.s. läggas till ett nytt tal mellan) och där x är en parameter som kan anta ett värde mellan 0 och 1. Vad för tal funktionen lämnar ifrån sig beror på värdet av x. När x är 0 returnerar funktionen a och när x är ekvivalent med 1 lämnar den ifrån sig b. Alla värden mellan 0 och 1 på x gör så att funktionen ger ifrån sig ett tal vars värde hamnar mellan a och b.

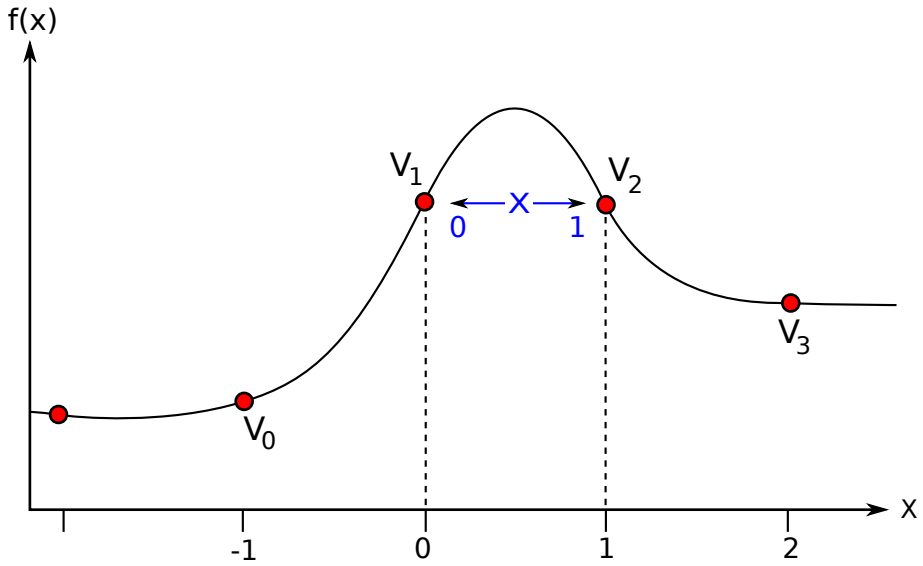
[10]Vid projektets början hade DACS bara stöd för att sampla HF signaler och därför var det de enda signaler som fanns lagrade.

Kubisk spline är en mer komplex form av interpolering som ger mjuka övergångar mellan punkter. Resultatet blir mjukare därför att interpoleringen inte enbart tar hänsyn till värdena a och b, vilket det ska interpoleras mellan, utan även intilliggande värden i samlingen. Den kubiska splinefunktionen som användes för LF syntetisering tog hänsyn till ett värde före a benämnt v_0 och ett värde efter b benämnt v_3. Värdena a och b gavs beteckningarna v_1 respektive v_2.

Namnet kubisk spline förklaras av att beräkningen som görs är av tredje ordningen:

$$f(x) = Px^3 + Qx^2 + Rx + S, \text{ där}$$

$P = (v_3 - v_2) - (v_0 - v_1)$, $Q = (v_0 - v_1) - P$, $R = v_2 - v_0$ och $S = v_1$, medan x fortfarande är densamma som för linjär interpolering. I följande figur illustreras det vilka värden som det tas hänsyn till vid kubisk interpolering.



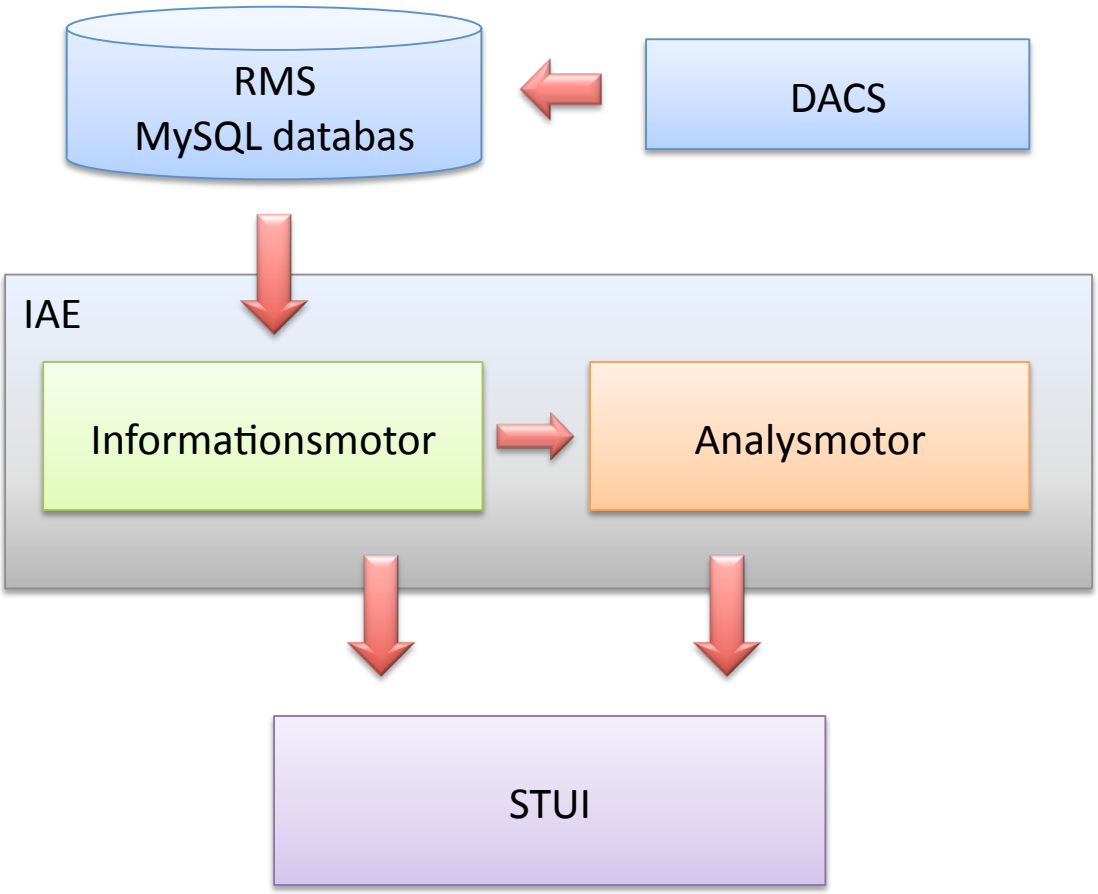
Figur 2.8: *Kubisk spline*

Men eftersom funktionen fortfarande bara ger ett tal mellan de slumpmässigt genererade värdena, applicerades den kubiska interpoleringsfunktionen fem gånger för att få fem extra tal mellan varje slumptalsgenererat värde. Detta för att få så realistisk signal som möjligt. Den kubiska splinefunktion som valts kan karakteriseras som en Catmull-Rum spline då $\tau = 1$ [1].

3

Resultat

3.1 Översikt



Figur 3.1: *En överblick av IAE och kringliggande system.*

Figur 3.1 illustrerar flödet av information från källan i DACS som direkt mäter på DMM-maskinen och vägen till STUI som använder IAE för presentation.

3.2 DACS

Som tidigare nämnts så är denna delen av central vikt för inhämtning av all information som skall analyseras. Det är denna som hämtar in all information från DMM. Denna process är viktig för alla examensarbeten som berör DISA men eftersom inget examensarbete egentligen handlar om bakgrundsprocessen, även om den berör alla så var det ingen som var intresserad att ta fullt ansvar för denne. Lägg därtill att inte alla var med och konstruerade den och därmed inte visste hur programmet var uppbyggt och helst inte ville sätta sig in i konstruktionen p.g.a. tidsbrist. I och med detta så har de 6 examensarbetare som är involverade med DISA valt att samarbeta om denna del och vidareutveckla olika delar som är av vikt för vederbörandes examensarbete.

För examensarbetarna av detta projekt innebar det att utveckla mätning av LF-signaler och säkerställa att HF signalerna kan hämtas och analyseras på ett enkelt och snabbt sätt. Det är också värt att nämna att examensarbeterna har utvecklat moduler och tekniker som läser in information som denna bakgrundsprocess genererar på ett effektivt och skalbart sätt. Även om import av CIM-värdes delen inte var utvecklat av examensarbetarna har basen för detta utvecklats av dem i RMS Synthesizern verktyget.

3.3 Informationsinhämtning

Som tidigare nämnts handlar informationsinhämtningen om att på ett flexibelt och generellt sätt hämta information som är modulärt, utbyggbart och skalningsbart.

3.3.1 Val av databasmotor

Det finns en uppsjö av databasmotorer att välja bland men det finns inte allt för många som inte kräver en licens. Då MySQL är en erkänd RDBMS, brett använd och har gott stöd för många plattformar samt enkelt kan användas i C# så valdes denna databasmotor.

3.3.2 Databasuppbyggnad

Det finns tre typer av källor för information: **HF-signaler**, **LF-signaler** och **CIM-värden**. Dessa tre hämtas in för varje maskin som använder RMS-systemet. För att minska komplexitet och underlätta backup och hantering av informationen som är insamlad så får varje maskin en egen databas i MySQL. Det finns alltså en databas per maskin som innehåller all information som berör just den maskinen. Innehållet i denna databas kan utökas utan problem; det kan läggas till tabeller och information utan att det skapar problem för informationsinhämtaren. Det är också så att det finns en central databas som kallas för `RMS`. Denna centrala databas innehåller information om vilka maskiner som finns, vilken databas som den aktuella maskinens data ligger i och information om serialnummer, var maskinen finns och ett namn på den.

När examensarbetet påbörjades fanns endast den databas som DACS använde. Den innehöll endast HF-signaler. Under arbetets gång har mätningar av LF-signaler och synkronisation av information som finns i CIM lagts till. HF-uppbyggnaden har ändrats, det fanns idéer redan från början att använda binär lagring, denna blev dock inte fastställd förrän precis innan examensarbetet påbörjades. För att få en känsla för vad det handlar om har detta beskrivits här under resultat. HF signalbiten fanns redan färdig vid påbörjandet av examensarbetet och det är endast för förklaring till analys en utförlig beskrivning har tagits med.

Materialiserade vyer

Av DISA så fick examensarbetarna en exempeldatabas av CIM-värden. Denna CIM-värdesdatabas innehöll 800 MB data, runt 8 miljoner databasrader totalt. Att göra sammanställningar av denna databas var mest intressant men det krävde att MySQL var tvungen att gå igenom all information; det finns inget snabbare om man t.ex. vill summera produktionsstatistik per dag för all data som finns. Detta gick inte så snabbt; tog runt en halv minut för en sammanställning. Väljer man små tidsrymder är det inga problem. Men ofta var det intressant att summera per dag, vecka, månad och år.

För att lösa detta behövdes sammanställningar göras. Det finns en enkel lösning för detta och det kallas för *Materialiserade vyer*. En materialiserad vy fungerar precis som en tabell som endast kan läsas fast innehållet är skapat av en SQL fråga. Istället för att köra frågan varje gång sparas sammanställningen och uppdateras då källinformationen ändras. MySQL som valdes att användas stödjer inte dessa materialiserade vyer så en alternativ lösning valdes. Det var att skapa en vanlig tabell och fylla den med information från SQL frågan och sedan lägga in triggers som reagerar då informationen ändras och uppdatera denna tabell [8]. När det i den följande texten skrivs att det löstes med materialiserade vyer så är det den ovan nämnda lösningen som det syftas till. Principen och funktionen är samma men implementationen skiljer sig, materialiserade vyer är enklare att skapa och ändra.

HF signaler

Denna biten utvecklades från början i bakgrundsprocessen (DACS) och signalerna lagras m.h.a. denna binärt i MySQL vilket innebär att varje databasrad eller tupel[1] kan innehålla t.ex. 128 st mätvärden från 16 st olika kanaler, detta ger med andra ord 2 048 mätvärden per rad. Det valdes att göra på detta sätt eftersom det är mer lagringseffektivt samt avsevärt mycket mer optimalt för överföring till ett program. Att lagra informationen på ett relationellt korrekt sätt var så pass ineffektivt att det övervägde fördelarna med att kunna söka på informationen med hjälp av MySQL:s kraftfulla sökfunktioner.

För att visa på varför det valdes att göra så här: Varje högfrekvent mätning är ca 2 minuter lång och är indelade i *sessioner*. Varje session är på minst 16 kanaler och innehåller 300 mätvärden per sekund. $2 \cdot 60 \cdot 300 \cdot 16 = 576\,000$ mätvärden per session. Varje mätvärde är ett 16 bitars heltal med tecken vilket innebär att varje mätvärde tar 2 byte. $576\,000 \cdot 2 = 1\,152\,000$ byte för en session. Lägg där till att det görs 3 mätningar per dag och 365 gånger om året: $1\,152\,000 \cdot 3 \cdot 365 = 1\,261\,440\,000$ byte eller $1\,261\,440\,000/1\,024^3 \approx 1.17$ GB. Det är inte så farligt egentligen men det visade sig att varje mätvärde kräver mer än bara själva mätvärdet, det måste vetas vilken signal det gäller, vilken session signalen tillhör och även en räknare för mätvärdet då det kan teoretiskt hända att mätningarna bryts itu. Varje mätvärde tog då 55 - 60 byte med alla tillägg som behövs av MySQL. Det är minst 27.5 gånger mer än vad som behövs. Lägg där till att tabellen behöver för ett år innehålla 631 miljoner rader och erfarenheter vid testning visade på att det redan är mycket långsamt för sökningar vid 5 miljoner rader och kopiering av innehållet tog flera minuter och det var bara 10 sessioner, alltså motsvarande 3 dagars samplingar.

Genom att lagra informationen binärt, 2 048 mätvärden per databasrad, så kunde man reducera antalet rader 2 048 gånger och tilläggen per mätvärde är bara ca 2.5 byte vilket är 22 gånger bättre. Nackdelen är att sökning i innehållet måste göras i ett program som har laddat hem och packat upp informationen. Värt att nämna är att reduktionen i storlek blir viktigare också i framtiden då DISA har uttryckt stöd för minst 64 kanaler som det skall mätas på. Lagringsmässigt är det lösbart men den overhead det blir för att hämta information gör systemet i princip obrukbart när man skall göra analys av större mängder information.

[1] En ordnad mängd, kallas ibland koordinat (Källa: `http://tyda.se/search?form=1&w=tuple&w_lang=&x=0&y=0`)

LF signaler

När bakgrundsprocessen konstruerades så fanns ingen uttryckt önskan om att kunna mäta långsamma signaler (LF). Dock visade det sig att dessa signaler är de signaler som egentligen talar om mest om hur gjuterimaskinen mår och var därför viktiga att mäta. Det fanns idéer på att mäta med hjälp av PLC men då examensarbeterna inte hade tid att utforska detta område så valdes en kompromiss som kunde lösas med existerande kunskap och hårdvara. Bakgrundsprocessen gjorde HF-mätningar med hjälp av en mätare som kommer från DataQ[2]. Via en ActiveX[3] kontroll som DataQ levererar med mätaren kan man hämta information från deras mätare.

LF signalerna hämtas från denna DataQ mätare och lagras på ett sätt som följer god databaskonstruktion eftersom det inte handlar om samma datamängder som vid HF signaler samt att dessa läggs till kontinuerligt och inte i korta tidsbegränsade skurar av data. Varje rad innehåller:

- En tidstämpel.
- Information om vilken signal det är.
- Det ursprungliga råa mätvärdet som ett 16 bitars heltal.
- Ett omvandlat mätvärde i sin rätta SI enhet, representerat som ett reellt tal.

Denna informationen kan sökas igenom och hämtas via enkla SQL-frågor.

Tanken är att mäta allt från en gång per sekund till en gång i timmen. Detta kan teoretiskt också skapa stora mängder information men i detta fall valdes det att skapa en materialiserad vy som sammanställer informationen på ett effektivt sätt.

LF signalerna mäts kontinuerligt så länge DACS processen körs, buffras upp med tidsstämplar då datan har anlänt och skickas till MySQL. Eftersom DataQ samplaren inte kan göra mätningar långsammare än 1 gång per sekund valdes det för längre tidsmätningar att summera alla mätningar och sedan dela med antalet mätningar. Detta blir alltså ett medelvärde över lång tid som lagras.

[2]Mer om dem kan läsas på `http://www.dataq.com/`.

[3]En teknik utvecklad av Microsoft för att köra tredjeparts mjukvara inuti en annan tillverkares mjukvara. Tekniken är huvudsakligen utvecklad för att en användare ska kunna interagera med denna kontroll.

CIM värden

DISA har delat upp innehållet i fyra delar:

- Information om produktion
- Väntetidsinformation som är sammanställd
- PLC-fel som har uppstått
- Mönsterinformation för formar som produceras

Examensarbetarna fick en exempeldatabas från DISA som innehöll 5 års produktionsdata. Det handlade om 800 MB information. Ur lagringssynpunkt är detta egentligen inte mycket men eftersom sammanställningar och statistisk analys och inte specifika värden var av intresse ur uppgiftens synvinkel, så blev det tungt att ladda in. Det tog ungefär en halv minut[4] för att sammanställa hela databasen. Inte så lång tid egentligen men med tanke på att det skall visas i ett användargränssnitt så hade det varit oacceptabelt lång tid.

Precis som LF signaler valdes det att skapa materialiserade vyer. Användandet av materialiserade vyer så som beskrivet under avsnitt 3.3.2 gjorde sammanställningar mycket effektiva. Den materialiserade vyn gjorde att det istället för en halv minut tog runt 50 - 90 millisekunder på en Core 2 Duo 2.2 GHz MacBook Pro under Windows 7 att finna den informationen som det söktes efter. Den materialiserade vyn konstruerades med hänsyn till att så mycket information som möjligt skulle kunna tas ur den och att det skulle kunna gå att koppla värden med andra tabeller utan att behöva söka igenom allt för mycket information. Vyn summerade produktionsdata ner till timmen, så det finns bara information i diskreta steg om 1 timme per steg.

Väntetidsinformation var redan sammanställd, exempeldatan given av DISA innehöll ca 1 månads information vilket inte var tillräckligt för långsiktig statistik men tillräckligt för att bygga en lösning som visar informationen.

[4]Mätningarna gjordes på en bärbar MacBook Pro, 2.2 GHz Core 2 Duo, 4 GB RAM, 2.5 tums hårddisk under Windows 7, 32 bit med MySQL Query Browser 1.2.14.

Modulär struktur

Som tidigare nämnts skulle informationsmotorn vara utökningsbar och flexibel. Detta kunde lösas med hjälp av Plug-in konceptet. Det finns flera anledningar till valet av just Plug-in konceptet, några av dem är:

- Vem som helst som har tillgång till specifikationen på hur gränssnittet ska se ut samt det attribut som används för att märka viktiga klasser kan utveckla Plug-ins utan att behöva veta hur hela programmet fungerar.
- Det är lätt att lägga till funktioner utan att behöva bygga om programmet.
- Arkitekturen tillåter att informationsmotorn används för mer än endast en specifik lösning då den kan utökas och är därmed framtidssäkert.

Tekniskt sett åstadkoms plug-in systemet med hjälp av *reflection* som innebär att man tittar på den metadata som finns lagrad i binären som bland annat beskriver vilka klasser som finns, vilka metoder de har o.s.v. Man söker efter ett specifikt `Attribute` som en eller flera klasser har. Detta *attribut* markerar på klassnivå vilken eller vilka som är huvudklasser d.v.s. den eller de klasser som skall följa uppsatta reglerna för ett Plug-in. Klassen förväntas följa ett definierat gränssnitt vid namn `IInformationDriver`. Detta gränssnittet definierar saker som vad drivrutinen heter, vilken version den innehar samt innehåller ett sätt för att skapa ett instans av drivrutinen. Vid skapning av denna drivrutin som även kallas informationsdrivrutin så undersöks och verifieras det att gränssnittet är korrekt.

Utdrag från kod för SQL-informationdrivrutinens deklaration

```
/// <summary>
/// The SQL Driver
/// </summary>
[InformationDriverAttribute(1, "SQLEngine")]
public class SQLDriver : IInformationDriver
{
    /// <summary>
    /// Description of what type of information the driver can get.
    /// </summary>
    public string Description
    {
        get { return "A driver that by executing SQL retrieves information"; }
    }

    /// <summary>
    /// Who wrote the driver?
    /// </summary>
    public string Author
    {
        get { return "MK, DA"; }
    }

    /// <summary>
    /// What is the version of the driver?
    /// </summary>
    public string Version
    {
        get { return "1.0"; }
    }

    /// <summary>
    /// The name of driver
    /// </summary>
    public string Name
    {
        get { return "SQL Retriever Module"; }
    }

    /// <summary>
    /// Create an instance of CIM driver
    /// </summary>
    /// <returns>A new CIM driver instance</returns>
    public IInformationDriverInstance CreateInstance()
    {
        return new SQLRetrieve();
    }

    /// <summary>
    /// The unique id that is used to get hold of the
    /// driver in the RetrievalEngine
    /// </summary>
    public string UniqueId
    {
        get { return "SQLEngine"; }
    }
}
```

Tidigare kan ses ett exempel på hur SQL-informationsdrivrutinen är deklarerad. Innan klassdeklarationen görs så står det

```
[InformationDriverAttribute(1, ‘‘SQLEngine’’)]
```

där siffran *1* betyder API version 1. I framtiden kan det hända att revisioner behöver göras på utseendet i gränssnittet och detta kan man då göra genom att öka siffran så att värden vet att denna drivrutin använder en annan API version. Namnet “SQLEngine” är det unika ID som används för att hitta drivrutinen.

Klassen består av en del metoder. Alla de som skrivs ovan nämnda exempel är definierat av gränssnittet `IInformationDriver`. De flesta är ganska självförklarande. `CreateInstance()` är kanske inte det. `CreateInstance()` skapar en instans av drivrutinen. Alla egenskaper och metoder i denna klass är definierat av gränssnittet `IInformationDriver`.

Från denna kan man skapa en instans av drivrutinen. Det finns alltså två stycken klasser, en som är drivrutinshuvudet och en som är en användbar instans av drivrutinen. Anledningen till att dessa två är separerade istället för att skapa en användarbar instans direkt är för att undvika att göra tunga inladdningar vid sökning av drivrutiner d.v.s. göra lat initiering[5] (eng. lazy initialization). Då en instans skapas skickas det med ett gränssnitt till informationsmotorn så att databasanslutningar kan skapas. Även andra detaljer som specifika inställningar skickas med och fullständig initiering görs.

[5]Ett sätt att fördröja fullständigt skapande av kostsamma objekt och istället skapa dem då de behövs fullständigt istället för att alltid skapa dem.

Utdrag ur kod för drivrutinsgränssnittet

```
/// <summary>
/// An instance of an generic driver
/// </summary>
/// <remarks>
/// Setup is always called before FunctionDescription,
/// FunctionList or Function. IDriverInstance is not threadsafe,
/// only one thread per instance is allowed. Multiple instances
/// are allowed.
/// </remarks>
/// <threadsafety static="true" instance="false"/>
/// <typeparam name="DescriptionType">The type that describes
/// the content of the driver</typeparam>
/// <typeparam name="FunctionType">The type that represents
/// a function</typeparam>
public interface IDriverInstance<FunctionType,DescriptionType>
 : IGenericFunction, IDisposable
{
    /// <summary>
    /// Checks if function exists
    /// </summary>
    /// <param name="name">Which function is it?</param>
    /// <returns>true if function exists</returns>
    bool FunctionExists(string name);

    /// <summary>
    /// Get the description of a particular function
    /// </summary>
    /// <param name="function">Which function is it?</param>
    /// <returns>The function description</returns>
    /// <remarks>Returns null if function does not exist.</remarks>
    DescriptionType FunctionDescription(string function);

    /// <summary>
    /// Get the list of functions in this driver.
    /// </summary>
    List<string> FunctionList { get; }

    /// <summary>
    /// Use a function in the driver.
    /// </summary>
    /// <param name="function">Which function is it?</param>
    /// <returns>The retrieve function that can retrieve data from
    /// the input parameters</returns>
    /// <remarks>Returns null if function does not exits.
    /// This function is not thread safe to be called from multiple threads.
    /// You will have to have a IDriverInstance per thread in order to
    /// use multiple functions from a driver at the same time.</remarks>
    FunctionType Function(string function);
}

/// <summary>
/// An instance of an Information Driver Instance
/// </summary>
public interface IInformationDriverInstance
    : IDriverInstance<InformationFunction, InformationFunctionDescriptor> { }
```

På förra sidan finns ett utdrag ur koden som visar hur gränssnittet för en generell drivrutin ser ut. Då informationsfunktioner och analysfunktioner är så lika så valdes det att göra en generell bas och återanvända samma struktur. Detta nämndes också under avsnitt 2.5.3. I utdraget finns fyra stycken metoder. Det mesta beskrivs i dokumentationen men nedan följer en översättning:

FunctionExists tar reda på om en specifik funktion existerar eller inte

FunctionDescriptor ger tillbaka en beskrivning av en specifik funktion. Denna beskrivare innehåller information, i informationsfunktionsfallet, vad funktionen ger tillbaka och vad för argument den tar in. Det finns två olika sådana här.

FunctionList ger en lista över alla funktionsnamn som finns i drivrutinen.

Function skapar en användbar funktion som man sedan kan köra och få ett resultat från.

Universell informationsinhämtning

All information som kunde hämtas kan representeras i tabellform och eftersom IAE är baserat på en relationell databasstruktur så var det logiskt att informationsinhämtningen faktiskt levereras på samma sätt. Istället för att göra en fullständig ADO.NET databasdrivrutin som var ett alternativ, valdes det att utveckla ett lättviktigt gränssnitt som kan leverera stora mängder information på ett universellt sätt och som kan dra nytta av den enkla och utökningsbara strukturen.

Detta kunde `DataResult` göra. Den hämtar information på samma sätt som information hämtas från en SQL-databas d.v.s. per rad tills det inte finns mer rader. Varje rad kan innehålla en eller flera kolumner som innehåller någon av följande datatyper:

- Sträng
- Heltal
- Reellt tal
- Tidsomfång t.ex. 1 dag 4 timmar 1 minut 4 sek. `TimeSpan` strukturen som finns i .NET användes för detta.
- Datum/Tid

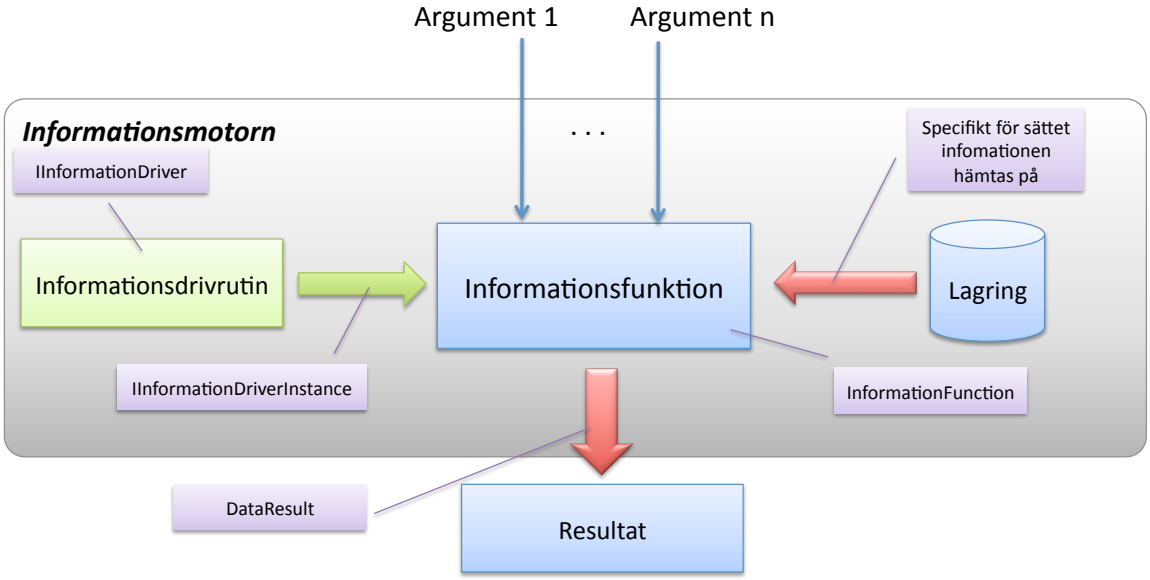
Information ur `DataResult` hämtas på följande sätt:

Pseudokod för inhämtning

```
1 Så länge data finns
2         Hämta kolumn n från resultatet
3         Hantera informationen
4 Rensa upp
```

Det finns också utökningar som omvandlar resultat till en ADO.NET datastruktur vid namn `DataTable`. Denna används i ett `DataSet`. Varje resultat innehåller också information om själva resultatet precis som i ADO.NET fast med tillägg av en beskrivning levererad av informationsdrivrutinen samt vilken funktion som användes för att skapa innehållet.

3.3.3 Hur hänger alla delar ihop?



Figur 3.2: *En illustration som visar hur flödet av information är samt hur en informationsfunktion initieras för användning.*

Blå färg innebär entiteter som har en funktion eller representerar en sak, röd är flödet av information, lila är hjälptexter som beskriver vilken klass som används vid vägen i fråga, grön färg är vägen för att initiera en informationsfunktion för användning.

Informationsdrivrutinen använder en del underklasser som har klara roller, mer om vad dessa delar gör har beskrivits tidigare under avsnitt 2.4.4 på sidan 44.

Argumenten ges från t.ex. användaren i ett användargränssnitt och sätts som parametrar till funktionen.

Resultat representeras i en implementerad `DataResult` som beskrevs tidigare.

Informationsmotorn är uppbyggd på ett enkelt sätt och är även enkel att använda. För att t.ex. välja en informationsfunktion, sätta dess parametrar och skriva ut all information så är koden undertill allt som behövs. Koden använder en hjälpmetod som finns i informationsmotorn, denna heter `Retrieve`.

Exempelkod för hur man kan använda informationsmotorn

```
//Initiera alla plug-ins
PlugIns.InitializeEngines();

//Skapa informationsmotorn
InformationEngine engine =
        new InformationEngine("configuration.xml");

//Ladda en informationsdrivrutinen "SQLEngine"
//och använd informationsfunktionen "GetLFSignal".
DataResult result = engine.Retrieve(
        "SQLEngine",   //Informationsdrivrutinen
        "GetLFSignal", //Informationsfunktionen
        "DMM_OIL",     //Vilken LF signal det är
        new DateTime(2008,1,1,0,0,0),     //2008-01-01 00:00:00
        new DateTime(2008,12,31,23,59,59) //2008-12-31 23:59:59
);

//Gå sedan igenom all information
while(result.Next())
{
        for(int i = 0; i < result.ColumnCount; i++)
        {
                Console.WriteLine(

                "Col[" + i + "] = "
                 + Convert.ToString(result[i])

                );
        }
}
```

Koden ovanför initierar alla informationsdrivrutiner, skapar en instans av informationsmotorklassen med konfigurationsfilen `configuration.xml`. Därefter hämtas resultatet av informationsfunktionen “GetLFSignal” och gås igenom och skrivs ut i terminalen. De två datumen avgränas resultatet, i detta fall är det hela år 2008.

3.3.4 Informationsdrivrutiner

För att kunna utvärdera systemets arkitektur och effektivitet så fanns ett behov att implementera två stycken informationsdrivrutiner. Dessa drivrutiner behövdes även av både **STUI** för presentation av sammanställd information och rapportgeneratorn som skapade rapporter baserade på sammanställd information som kommer från informationsmotorn.

De två informationsdrivrutiner som skapades var HF-informationsdrivrutinen och SQL-informationsdrivrutinen.

HF-informationsdrivrutinen innehåller funktioner som hämtar samplade HF-mätvärden som är lagrade binärt och översätter dessa till reella mätvärden. Den används för att kunna presentera bl.a. samplade HF-mätvärden i STUI och kunna exportera till ASC-filformatet.

SQL-informationsdrivrutinen är speciell, drivrutinens funktioner definieras av huvudkonfigurationen som är i XML. Det betyder att funktioner kan skapas dynamiskt utan att någon kod behöver skrivas, allt definieras i XML. Drivrutinen fungerar som en adapter för ADO.NET och använder `DataReader` internt. Denna drivrutin används för att hämta samplade LF-mätvärden och sammanställd CIM-data.

3.3.5 Utvärdering

Den lösning som skapats är generell och stödjer dynamisk körning och hantering av informationsfunktioner. Att lösningen är så pass generell skapar vissa problem såsom att funktioner som används måste kontrolleras att de faktiskt finns. Det måste också kontrolleras att den förväntade utdatan är av de typer som förväntas.

Man skulle kunna skapa en klass som bara innehöll dessa funktioner och så att säga "hårdkoda" dem. Det finns fördelar och nackdelar med detta. Fördelen är att det är mycket enklare. Nackdelen är att det kan bli komplicerat längre fram att utöka mängden funktioner eller rentav ändra i dem för att man kan ha valt att ta genvägar som ökar utvecklingshastigheten. Denna lösning ger möjlighet att till att hårdkoda valet av funktion, men dessa måste på något sätt kontrolleras att de finns annars kommer programmet som använder dem att krascha vid försök.

Den enskilt största anledningen till val av en så generell grund är för att kunna sätta upp analysmodeller dynamiskt i ett användargränssnitt. Att även kunna utöka funktionerna utan att behöva modifiera grunden är också en fördel då det blir enklare att underhålla koden. Det finns naturligt sätt mindre saker som kan gå fel i grunden.

3.3.6 Problem och lösningar

Under ett skarpt test i Danmark för att testa igenom informationsmotorn och DACS uppdagades det ett problem. Problemet bestod i att företagets kunders maskindatabaser kan skilja sig åt vad gäller strukturen för hur CIM-värden lagras. Det vill säga de motsvarande relationstabellerna för CIM-värden behöver inte se likadana ut för varje kunds maskindatabas. T.ex. kan de ha fler antal kolumner (attribut relationsspråkligt talat) alternativt sakna några kolumner som andra maskindatabaser har. Det kan t.o.m. vara så att hela tabeller inte finns hos den ene men hos den andre. Det var när testmaskinens CIM-databas inte innehöll samma datastruktur som den under utvecklingen använda CIM-databaskopian, som skillnaden upptäcktes.

Varför är det då ett problem? Det utgör ett problem därför att en informationsinhämtningskomponent inte på förhand kan veta hur en datakälla är uppbyggd och ställa de rätta frågorna. En inhämtningskomponent är programmerad för att kommunicera med en viss typ av datakälla eftersom den innehar ett antal unika frågor just för den aktuella datastrukturen. Skiljer sig strukturen betyder det att frågorna inte kommer att vara korrekta mot just den datakällan och programmet blir inkompatibelt.

För att lösa problemet implementerades en verifieringsmekanism i programmet som verifierar att den databas den kommunicerar med innehar den förväntade typen av datastruktur i form av tabeller och deras uppbyggnad. Verifieringsmekanismen sammanfattades i en komponent som döptes till **Verifier** och som beskrivs mer utförligt i nästa avsnitt.

3.3.7 Verifier

Ett enkelt program som utifrån en beskrivning i XML kan verifiera en databas struktur. Alltså att de tabeller och kolumner som förväntas finnas faktiskt existerar. Detta är användbart för att undersöka så att alla tabeller har skapats korrekt och så att import av data från t.ex. CIM har importerats på korrekt sätt.

Formatet ser ut så här:

Exempel på hur XML formatet ser ut

```
<specification>
  <rms>
    <table name="MachineList">
      <column name="serialId" type="integer" primary="1" canbenull="0" />
      <column name="friendlyName" type="string" canbenull="0" />
      <coulmn name="modelName" type="string" canbenull="0" />
      <column name="databaseName" type="string" canbenull="0" />
      <column name="country" type="string" canbenull="0" />
      <column name="location" type="string" canbenull="0" />
    </table>
  </rms>
  <machine>
    <table name="SampleLowFreq">
      <column name="ConfigurationID" type="integer" primary="1" />
      <column name="TimeStamp" type="datetime" primary="1" />
      <column name="RawValue" type="smallint" />
      <column name="MetricValue" type="double" />
    </table>
  </machine>
</specification>
```

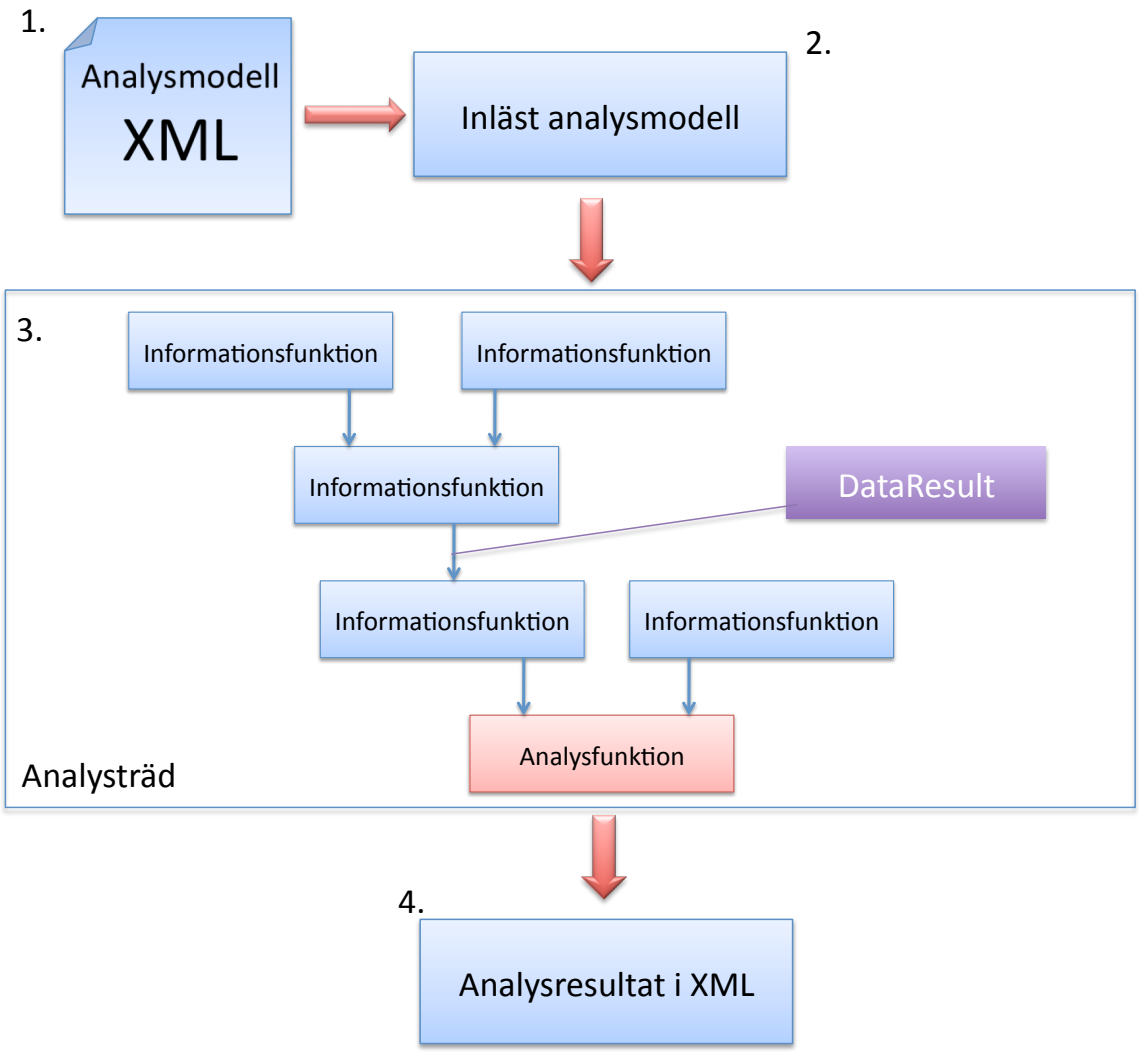
Som kan ses ovan finns två delar en för `rms` och en för `machine`. `rms` är själva huvuddatabasen som innehåller en lista över maskiner. Då börjar verifieraren att först gå igenom denna och sedan hämtar den varje maskindatabas och verifierar den databasstrukturen. Det kan finnas fler än en tabell i beskrivningen, detta exempel är nerkortat för att inte förbruka en för stort utrymme i rapporten. Varje kolumn har en datatyp t.ex. string, ett värde som beskriver om kolumnen får vara `NULL` eller inte (`canbenull`) och slutligen ett attribut som säger om nyckeln är primär eller inte.

Utifrån specifikationen ovan kontrollerar Verifier om databasen är korrekt uppsatt. Om databasen är korrekt så kommer inte de inställningar som är satta i IAE eller beroende projekt att krascha för databasfel.

3.4 Analysmotor

Denna del handlar om att kunna utföra automatiserad analys på ett flexibelt sätt.

3.4.1 Den stora strukturen



Figur 3.3: *Vägen igenom en analys*

En analysering består av 4 stycken steg som illustreras i figur 3.3 på föregående sida:

1. Ladda in Analysmodellens beskrivning som finns i XML, vilken läses in från huvudkonfigurationen.
2. Skapa en modell som innehåller alla inställningar för varje informationsfunktion, representeras av `AnalyzeModel`.
3. Skapa ett träd som är körbart och har alla informationsfunktioner skapade och kopplade, representerat av `AnalysisTree`.
4. Kör analysen och skapa ett analysresultat som beskrivs i XML.

3.4.2 Analysmodellen

Denna modell beskrivs i XML. Det kan finnas flera analysmodeller i huvudkonfigurationen. Varje modell har en enda slutanalysfunktion, se figur 3.4. Analysmodellen beskriver hur informationsfunktioner kopplas ihop och hur de olika variablerna omsätts mellan varandra.

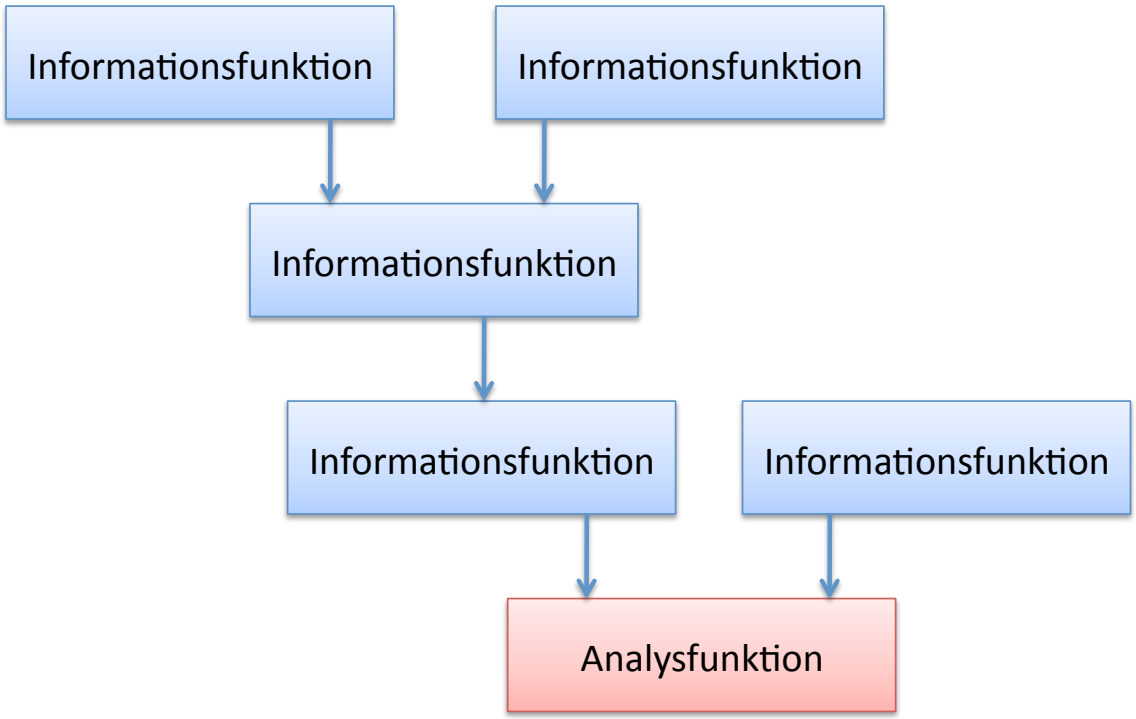
I figuren så visas hur en analys kan byggas upp. Den byggs upp med informationsfunktioner som kopplas till en eller flera andra informationsfunktioner. De första informationsfunktionerna kan antingen vara helt utan inparametrar eller med parametrar som är konstanter. Själva modellen ska alltid sluta med en enda analysfunktion som producerar ett resultat i XML.

3.4.3 Kopplingar mellan informationsfunktioner

Genom att titta i figur 3.4 så kan man se att det finns kopplingar mellan informationsfunktioner och även till slut till en analysfunktion. Det finns två olika typer av kopplingar:

Dataresultatkoppling är en koppling som förmedlar resultatet representerat som `DataResult` direkt till en annan informationsfunktions ingång. Denna koppling kan alltså förmedla mer än en typ av data och fler än ett värde.

Parameterkoppling innebär att ett värde förmedlas direkt från ett resultat.



Figur 3.4: *Ett generellt analysträd*

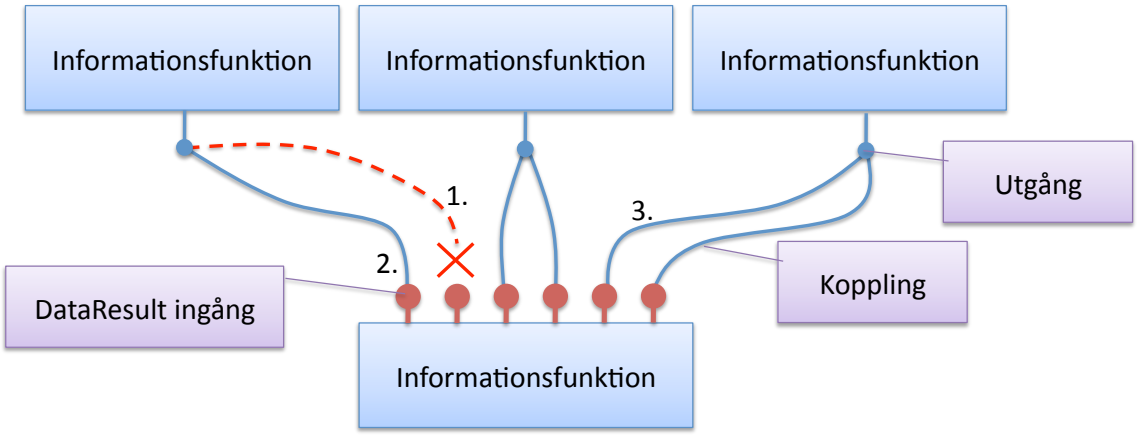
Båda typerna av koppling kräver att de översätts mellan varandra. Detta görs genom att allt resultat som produceras av en informationsfunktion samt alla inparametrar har namn, variablerna kopplas ihop med hjälp av dessa namn och kopplingen beskrivs i analysmodellen. En dataresultatkoppling är speciell då denna kan förmedla mer än en variabel. Denna koppling omsätter därför fler än en variabel och kan tänkas som många kopplade kablar inuti i ett rör om man tittar på strukturen utifrån.

Kopplingar mellan informationsfunktioner har några begränsningar:

- Endast en informationsfunktions resultat kan kopplas till en uppsättning parametrar i en annan informationsfunktion, d.v.s. du kan inte koppla resultatet från en informationsfunktion till två olika.
- Endast en enda dataresultatkoppling får existera från en informationsfunktion till en annan, se nr 1 i figur 3.5.
- Analysfunktionen är alltid unik i ett analysträd, det kan bara finnas en i trädets rot.

Anledningen till dessa begränsningar är att det får bara finnas en *drivande* funktion. Med *drivande* menas det att det bara får finnas en funktion som går igenom innehållet och *driver* resultatet framåt. I programmet så sätts resultaten ihop och flera körningar av informationsfunktioner t.ex. görs transparent från den sista del som faktiskt läser resultatet. Att kunna kombinera resultat på detta sättet är det som gör att analysmotorn fungerar utan att bli för komplex, detta gör också att informationsfunktionerna inte behöver veta att resultat sätts ihop.

I figur 3.5 så illustreras kopplingar. Illustrationen visar också på ett av problemen som nämnts tidigare med att två olika informationsfunktioner kopplas till en gemensam och för att detta ska fungera så används den cartesiska produkten av de ingående informationsfunktionernas resultat för att ge alla kombinationer av parametrarna.



Figur 3.5: *Kopplingar mellan informationsfunktioner*

1. Denna koppling tillåts inte. Anledningen till detta var för att det gick inte gick att definiera hur programmet skulle fungera vid detta fall.
2. Detta är en dataresultatkoppling som förmedlar ett resultat via `DataResult` direkt till en annan informationsfunktion som tar detta resultat som inparameter.
3. En vanlig parameterkoppling som innebär att en informationsfunktion behöver köras så många gånger som det finns kombinationer av parametrar.

För att visa den cartesiska produkten fungerar i vårt fall så visas ett exempel nedan:

$$\begin{aligned} a &= \{2001, 2002\} \\ b &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \\ a \times b &= \{2001, 1\}, \{2001, 2\}, \cdots, \{2001, 12\}, \cdots, \{2002, 12\} \end{aligned}$$

a och b kan vara två olika radresultat som kommer ut ur två `DataResult` klasser. Dessa två kombineras genom $a \times b$ som är själva cartesiska produkten och det skapas $2 \cdot 12 = 24$ kombinationer. Man kan se a och b som två olika resultat som både förmedlar 1 variabel var, principen fungerar också när resultaten innehåller mer än 1 variabel per uppsättning men då är det uppsättningarna som kombineras och inte de enstaka variablerna.

3.4.4 Analysresultat

Ett analysresultat beskrivs i XML. Resultatet av en analys delas upp i 4 delar:

- Analysfunktionsinformation
- Information om analysmodellen
- Konstanter som behövs för att återskapa resultatet
- Resultatet

Analysfunktionsinformationen innehåller t.ex. vilken analysfunktion som använts, vilka parametrar som har satts och deras värden. Analysmodellen innehåller den information som krävs för att koppla ihop analysmodellbeskrivningen med analysresultatet. Den tredje delen som är konstanter behövs för att kunna återskapa resultatet och kan även kallas det initiala tillståndet. Det är alltså de värden som behövs för att återskapa resultatet vid felsökning. Den sista biten är själva analysresultatet.

Ett analysresultat kan ha 4 olika tillstånd:

OK vilket betyder att analysen gick igenom och inget konstigt påträffades, detta läget innehåller minimal information eftersom det inte finns något att anmärka på.

QUESTIONABLE betyder att ett värde ligger i en osäkerhetszon och bör kontrolleras av en servicetekniker. Här följer det med information som kan hjälpa till vid analysen som t.ex. värden kring det området där oljetemperaturen går in i osäkerhetsområdet.

NOT_OK detta innebär att t.ex. ett mätvärde ligger utanför normala förhållande och garanterat bör tittas över då mätvärdet är avvikande.

FAILED kan inträffa då analysen misslyckas som t.ex. informationen som krävs kunde inte hämtas.

Till tillståndet kommer ett kort meddelande som beskriver hur analysen gick. Tillståndet och ett meddelande som beskriver analysen är det som sammanfattar analysen. Sedan kan analysfunktionerna beroende på hur de är implementerade även inkludera värden för att skapa en graf, listor över analysvärden och andra generella variabler som kan vara värt att nämna. Dessa resultat följer en definierad XML standard för att kunna visas i ett användargränsnitt på ett standardiserat sätt.

3.4.5 Utvärdering

Analysmotorn kan ta en beskrivning av information i en XML fil. XML filen beskriver analysträdet som utför analysen. Informationsfunktioner kan kopplas till varandra och därigenom behandla stora mängder information. En nackdel med denna uppbyggnad är att det som gör att analysen får mening ligger i XML filen, detta gör att programmet kan inte utvärdera om den information den får in är rätt eller fel, den får informationen i det format det begär men det kan vara vad som helst. Fördelen är att vilken typ av information som helst så länge den följer datatypen kan behandlas.

3.5 Presentation & användargränssnitt

3.5.1 Presentationsalternativ

Då Microsofts .NET Windows Forms Chart Control tillhandahöll ett väldigt brett sortiment av alternativ för att presentera data på, öppnade den många möjligheter att presentera en gjuterimaskins maskindata på. Kontrollen erbjöd bl.a. datapresentation i form av följande diagramtyper:

- Stapeldiagram och cylinderdiagram
- Liggande stapel- och cylinderdiagram
- Linjediagram med linjer mellan värdepunkter
- Kurvdiagram (graf)
- Steglinjediagram
- Punktdiagram och bubbeldiagram
- Cirkeldiagram och ringdiagram
- Ytdiagram
- Histogram
- Pyramiddiagram och trattdiagram

De flesta av diagrammen fanns dessutom i tredimensionella varianter, med olika sätt att gruppera och stapla serier[6] på och med olika sätt att kombinera serier på. Med andra ord fanns det en hel uppsjö av alternativ att presentera maskindata på.

Med så många alternativa presentationssätt av maskindata, var det inte den enklaste av uppgifter att välja presentationsalternativ. Men med hjälp av olika diagramprototyper kom man fram till det mest optimala sättet att presentera varje typ av maskindata på.

[6]En serie är en post av presenterad data i ett diagram. Det kan förekomma flera serier i ett och samma diagram.

Produktionsdata för en gjuterimaskin är en typ av maskindata. En gjuterimaskins produktionsdata omfattar sex kategorier: antalet gjuteriformar som lyckades respektive misslyckades med att bli producerade, bli gjutna och få sina järnkärnor korrekt placerade. Det mest passande och optimala sättet att visa dessa data på per år, per månad samt per dag[7] för serviceteknikerna, var att sammanställa informationen i ett stapeldiagram för varje tidsintervalltyp och där tiden för varje stapeldiagram placerades utmed x-axeln och antalet formar utmed y-axeln.

För att serviceteknikerna enklare ska kunna ställa antalet gjuteriformar som lyckades med att bli producerade i relation till hur många gjuteriformar som å andra sidan misslyckades med att bli producerade, så staplades datakategoriernas respektive serier på varandra. På så sätt kunde man också lätt urskilja hur stor del utav det totala antalet försök att producera gjuteriformar som antalet lyckade respektive misslyckade formar utgjorde. Likadant gjorde man med resterande kategorier. Dessutom grupperades alla datakategoriers serier för att dem skulle kunna presenteras per respektive tidsintervalltyp. Möjlighet att välja vilka kategorier som ska visas i diagrammet lades också till genom att serviceteknikerna kan aktivera eller avaktivera serier via diagrammets symbolförklaring[8].

[7]DISA hade som önskemål att kunna överblicka en maskins produktionsdata år för år, månad för månad samt dag för dag för en tidsperiod på två månader tillbaka.

[8]Med varje diagram följer en symbolförklaring vilken förklarar vad varje serie betecknar för statistik genom att visa vad varje serie har för färgkodning i diagrammet.

En annan typ av maskindata är en maskins väntetider. Det finns nämligen ett antal faktorer som påverkar effektiviteten för en gjuterimaskin och det är väntetider för följande saker:

- Sand
- Järn
- Järnkärnor
- Shake out
- Pattern change
- Operational stop
- Guard gate stop
- Emergency stop
- Shot Air Pressure
- ACE
- External Unit
- Active Feeding Unit
- Vacuum

Ju längre tid maskinen får vänta på varje sak desto sämre blir effektiviteten. Att beskåda väntetider för gjuterimaskiner är av vikt för serviceteknikernas felsökningsanalys eftersom de dels kan ge en vink om var i maskinen något är felaktigt och att de dels kan fungera som underlag för kunden för vad denne kan förbättra för att få en effektivare maskin. Båda aspekterna ingår för övrigt i företagets strävan på att erbjuda så bra kundservice som möjligt.

I lösningen för hur väntetider skulle presenteras, blev fördelningen av antalet diagram och hur axlarna kom att se ut (fortfarande tiden på x-axeln fast istället för antalet formar på y-axeln placerades där antal minuter för väntetid) densamma som för produktionsdata fast med skillnaden att väntetider valdes att presenteras i linjediagram. Linjediagrammen utformades så att i ett första läge visar dem den totala väntetiden[9] för en gjuterimaskin i form av en enda linjeserie och är serviceteknikern sedan intresserad av noggrannare information än så kan denne klicka på linjen för att få en mer detaljerad vy över gjuterimaskinens samtliga väntetider.

[9]Det vill säga alla väntetider summerade totalt sett för en gjuterimaskin.

I detta läge representerar varje väntetid varsin linjeserie för att man på en smidigt sätt ska kunna jämföra väntetider inbördes emellan. Funktionen för aktivering och avaktivering av visade serier i diagrammet, implementerades även i denna presentationslösning.

Gällande presentationen av samplade HF-signaler och LF-signaler var det redan givet att dessa skulle presenteras i linjediagram eftersom det var så servicetekniker beskådade signalerna under deras analys.

3.5.2 Problem och lösningar

Det första problemet som uppenbarade sig under utvecklingen av användargränssnittet var ett problem rörande uppritningen av högfrekvent samplade maskinsignaler i linjegrafer. När man försökte rita upp signalerna med den grafiska kontrollen som valdes i förundersökningen av grafikkontroll, Microsofts .NET Windows Forms Chart Control [6], visade det sig att det skulle gå mycket långsamt och att uppritandet skulle frysa hela det grafiska användargränssnittet under en oacceptabelt lång tid trots att arbetet med uppritandet hade delegerats till en egen tråd. Det innebar sådana prestandaproblem att programmet för en användare blev näst intill oanvändbart. Det var ett problem som på förhand inte hade förväntats och där det inte direkt gick att säga vad som orsakade problemet, eftersom många faktorer i programmet kunde utgöra själva flaskhalsen. Man blev helt enkelt tvungen att gå igenom hela programkedjan, från det att datan hämtas från databasen till det att den ritas upp i linjegrafen och alla dess abstraktionsnivåer på vägen.

För att ta reda på vad som orsakade de omfattande prestandaproblemen inleddes en undersökning. Undersökningen gick ut på att utesluta det ena efter det andra i programkedjan tills att man stod kvar med orsaken till problemet. Ganska snart upptäcktes det att programmet förbrukade oanade nivåer av primärminne och att det hela handlade om ett minneshanteringsproblem. Eftersom programmeringsspråket som användes var C#, vilken har en internt arbetande skräpinsamlare (eng. garbage collector), kunde det inte vara frågan om en minnesläcka som man kan råka ut för i programmeringsspråk som C eller C++. Istället sågs de interna datastrukturerna för mellanlagring av datan över, t.ex. att signalsamplen lagrades som `DataTable`-objekt. Mycket riktigt krävde `DataTable` mycket minne och andra enklare lagringsformer testades. Vanliga vektorer erbjöd ett minnesutnyttjande på mindre än en tiondel så mycket som `DataTable`.

Men det återstod fortfarande minneshanteringsproblem vid själva databindningen av data och grafkontrollen. Alltså trots bättre mellanlagring allokerade programmet flera hundra megabyte vid uppritandet. Minnesanvändningen hade sjunkit med endast en fjärdedel och resterande andelar minne krävdes att också de släpptes fria för att programmet skulle hamna på en acceptabel nivå. Det var alltså grafkontrollen själv som förbrukade dessa mängder minne när den fylldes med data. Man visste att kontrollen lagrade varje sampel i objekt av den minneskrävande typen `DataPoint` på den hanterade högen (eng. managed heap). Det innebär att eftersom en högfrekvent samplad maskinsignal läses av med en samplingshastighet på 300 sampel/s och en typisk samplingssession av HF-signaler har en varaktighet på 2 minuter, så utgörs en session ungefärligen av

$$300 \cdot 2 \cdot 60 = 36\,000 \text{ sampel},$$

och därtill lika många objekt av typen `DataPoint`. Det är en rejäl mängd minnesockuperande objekt.

Slutsatsen av undersökningen blev att grafkontrollen inte gick att använda vid presentationen av HF-signaler. Signalerna representeras av helt enkelt för många punkter i grafkontrollen. Den enda möjliga lösningen på problemet var att utveckla en egen kontroll i WPF för just graferna till de högfrekvent samplade maskinsignalerna. Vilket gjordes.

För att utveckla en egen kontroll krävdes det nyinlärning på området användarkontroller och uppritning av visuella komponenter. När utvecklingen av denna kontroll gjordes fick examensarbetarna uppleva den del av WPF som inte var så önskvärd prestandaproblem. Att rita upp över 1000 linjer var otroligt långsamt, detta hade att göra med att WPF sköter all omritning och sparar alla instruktioner för uppritningen. Att behöva rita om alla 1000 linjer varje gång något ändras i användargränssnittet är p.g.a. overhead inte så effektivt. Detta löstes genom att cacha uppritningen som bilder och då kunde upp till 100 000 linjer ritas upp men programmet känns långsamt vid uppritning så i framtiden behövs förenkling av linjerna för att hantera större mängder punkter.

4 Diskussion

4.1 Slutsatser

En generell struktur är otroligt tidskrävande att bygga upp. Den måste tänkas igenom, utvärderas och dokumenteras. IAE är en generell struktur med undantag för vissa mindre specialiseringar. Informationsmotorn har lyckats med att vara flexibel och utökbar, detta bevisas genom användning i användargränssnitt, av rapportgenerator och analysmotor. Under utvecklingen var det en mycket stor fördel att ha tänkt igenom högnivådesignen utförligt. Varje klass i IAE har en specifik funktion, detta visade sig mycket fördelaktigt under utveckling vid felsökning.

Läsaren kanske ställer sig frågan *varför så generellt?* Detta har att göra med möjligheten till utökbarhet samt reduktion av utvecklingstid i framtiden. Examensarbetarna kunde inte förskaffa sig all information som var nödvändig för att skapa en mer specifik lösning som uppfyllde de behov DISA hade samt enkelhet vid vidareutveckling och med anledning av detta valdes en generell lösning. Det finns stora fördelar med en generell lösning, man kan tänka sig följande: Vid en konstruktion av en generell lösning så tar det mest tid i början därefter sjunker utvecklingstiden radikalt och fortgår mycket mer tidseffektivt. En generell lösning kan också användas för fler problem än vad den från första början tänktes lösa vilket ytterligare utökar den enorma tidsvinst som ges för en generell lösning vid vidareutveckling. I och med detta blir det därför en produkt för framtiden och det var det som var målet.

4.2 Vidareutveckling

Med en struktur som är så pass flexibel och utökbar som IAE är så finns det massor av möjligheter till vidareutveckling. Några exempel är t.ex. att implementera fler informationsfunktioner som täcker upp det fullständiga behov DISA har. En valideringsmekansim utformas som kan verifiera att informationsfunktioner som behöver användas faktiskt existerar bland de informationsdrivrutiner som finns laddade. Analysträdet kan förfinas med bättre felhantering samt även här fler analysfunktioner som täcker DISA:s fullständiga behov.

När det kommer till hur informationen är lagrad fick examensarbetarna en känsla för hur MySQL fungerade när det hade miljontals rader data. MySQL är optimerat för att hitta specifik information men inte att på ett effektivt sätt sammanställa massvis av information och ändå vara lagringseffektivt. Med anledning av detta kan det vara intressant att utvärdera möjligheten att byta till en annan databasmotor som t.ex. PostgreSQL för att utvärdera dess effektivitet och undersöka möjligheten att ytterligare optimera lagring och sökning.

Användargränssnittet STUI var inte fokus i detta projekt utan det var den underliggande programarkitekturen för att driva ett sådant som var det. I och med detta så har därför inte utförliga användartester gjorts som kan ge svar på hur ett användarvänligt användargränssnitt kan utvecklas samt hur interaktionen mellan människa och dator fungerar. Att utföra användartester är ett naturligt steg vidare. Möjligheten att beskriva analysmodeller grafiskt och en fullständig WPF baserad grafkontroll är andra förslag på vidareutveckling.

Terminologi

.NET
En mjukvaruplattform. Plattformen består av en samling som innefattar klienter, servrar, ramverk, programmeringsspråk och utvecklingsverktyg. 14, 16, 18–20, 24, 34, 35, 53, 57, 88, 91

ADO.NET
En teknik för att kommunicera med databasmotorer och hämta hem information från dem. Namnet har ärvts från det gamla COM baserade Active Data Objects (ADO) men har väldigt lite gemensamt med detta att göra. 14, 20, 21, 40, 76, 79

API
Application Programming Interface, gränsnitt mellan mjukvaror för att de skall kunna kommunicera med varandra. 24, 73

ASC
Ett ASCII filformat som innehåller kolumnbaserad data som kan användas vid utbyte av information mellan mjukvaror. 11, 12, 29, 79

CIM
Computer Integrated Manufactoring, en databas som innehåller maskinparametrar, väntetider och produktionsinformation. 28, 29, 39, 40, 46, 48, 60, 65–67, 79–81

DACS
Data Acqusition Service, bakgrundsprocessen som inhämtar information från DMM. 26, 64, 66, 68, 69, 80

DLL
Dynamic Link Library, ett mjukvarubibliotek som innehåller körbar kod och kan inkluderas i körande processer dynamiskt.. 19

DMM
DISA Moulding Machine. Det är DISA:s förkortning på deras gjuterimaskiner. 10, 27, 64, 65

EXE
Executable, ett körbart program på Windows plattformen. 19

FCL
Framework Class Library. Det gemensamma basklassbiblioteket för .NET. Det innehåller stöd för det mesta i programmeringsväg. 18, 20, 57

GUI
Graphical User Interface, användargränssnitt. 56

HF
High Frequency, högfrekvent samplade mätvärden exempelvis 300 mätvärden per sekund eller snabbare. 39, 40, 46, 66, 69, 79, 91, 92

IAE
Information and Analysis Engine, den del som har hand om både informationsinhämtning och analys. 1, 3–5, 29, 30, 32, 60, 61, 64, 81, 94, 95

IDE
Integrated Development Enviroment, en utvecklingsmiljö t ex Visual Studio. 20, 34

LF
Low Frequency, lågfrekvent samplade mätvärden exempelvis 1 mätvärde per minut eller långsammare. 39, 40, 46, 47, 60–62, 65, 66, 69, 70, 79, 91

PLC
Programmable Logic Controller, ett programmerbart styrsystem lämpat för automation. 28, 48

Plug-in
En struktur för modulära mjukvaror. En plug-in är en programmodul som kan laddas in och köras i ett annat program. 43

RDBMS
Relational Database Management System eller på svenska relationsdatabashanterare. 65

RMS
Remote Monitoring System. 10–12, 14–16, 26, 28–30, 32, 66

SQL
Structured Query Language, ett programmeringsspråk utformat för databasmotorer i syfte att kunna välja ut och göra beräkningar på information. 15, 41, 44, 67, 69, 73, 76, 79

STUI
Service Technician User Interface, Serviceteknikernas användargränssnitt. 15, 29, 64, 79, 95

UML
Unified Modeling Language, ett modelleringsspråk för dataveteskapsområdet. 34

WPF
Windows Presentation Foundation, ett grafikramverk för uppritning av användargränssnitt. 3, 4, 14, 18, 24, 25, 34, 92, 95

XAML
eXtensible Application Markup Language, ett uppmärkningsspråk med deklarativ syntax med taggar likt XML som används i programmering av Windows-applikationer. 18, 25

XML
eXtensible Markup Language, ett utökbart uppmärkningsspråk som används för lagra och transportera information. 18, 22, 24, 25, 53, 79, 81, 83, 86, 87

Litteraturförteckning

[1] Christopher Twigg *Catmull-Rom splines* `http://www.cs.cmu.edu/~462/projects/assn2/assn2/catmullRom.pdf` (Mars. 2003)

[2] Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995, ISBN 0-201-63361-2

[3] Liberty, J. et al., *Programmering C# 3.0*, Pagina Förlags AB, Sundbyberg 2008, ISBN 978-91-636-0940-4.

[4] MacDonald, M., *Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5, Second Edition*, Springer-Verlag New York, Inc., New York 2008, ISBN 978-1-4302-0576-0.

[5] MSDN, (2010) *BackgroundWorker Class(System.ComponentModel)* ,`http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.aspx` (April. 2010)

[6] MSDN, (2010) *Samples Environment for Microsoft Chart Controls*, `http://code.msdn.microsoft.com/mschart` (April. 2010)

[7] NUnit Project, (2007) *NUnit - Home*, `http://www.nunit.org/` (Mars. 2010)

[8] Padron-McCarthy, T. et al., *Databasteknik*, Studentlitteratur 2005, ISBN 978-91-44-04449-1.

[9] Troelsen, A., *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Springer-Verlag New York, Inc., New York 2007, ISBN 978-1-59059-884-9.

[10] W3C School, (2010) *XML Introduction - What is XML?*, `http://www.w3schools.com/xml/xml_whatis.asp` (April. 2010)