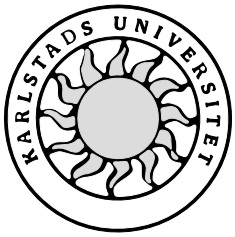


Datavetenskap

Markus Lindberg och Larserik Elander Jansson

Implementation av ett distribuerat processövervakningssystem

Examensarbete, C-nivå

2005:07

Implementation av ett distribuerat processövervakningssystem

Markus Lindberg och Larserik Elander Jansson

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Markus Lindberg

Larserik Elander Jansson

Godkänd, 2005-06-01

Handledare: Thijs Holleboom

Examinator: Donald Ross

Sammanfattning

Kritiska systemprocesser måste fungera i alla lägen. Rapporten tar upp och beskriver utvecklingen av ett distribuerat processövervakningssystem. Övervakningssystemet skyddar processerna mot driftsstörningar och andra oförutsägbara händelser. Siemens BT kommer använda slutresultatet till att öka tillförlitligheten av känsliga system i sin organisation. Rapporten går från att beskriva behovet av processövervakning, till analys av programvaran, dess användargränssnitt och allmänna tekniker för till exempel designmönster, kommunikation och datalagring. Säkerhetsaspekter och kryptering diskuteras för att bibehålla konfidentialiteten av informationsflödet mellan systemets komponenter. Design och implementation har dokumenterats i detalj med diagram och kodexempel. Slutligen presenteras resultatet av programvaran tillsammans med Siemens åsikter.

Implementation of a distributed process watchdog system

Abstract

Critical system processes have to run without interference. This report discusses the development of a distributed process watchdog system. A watchdog system protects processes from disturbances and other unpredictable events. Siemens BT will use the result of this project to enhance the reliability of sensitive systems in their organization. The report describes the need of a watchdog system, and it also provides an analysis of the developed software. The interface and general techniques for design patterns, communication and data storage is discussed. Also, different aspects of security and encryption have been taken into consideration to preserve a confidential flow of information between all parts of the system. Design and implementation have been documented in detail, including diagrams and code snapshots. The result of the project is presented together with Siemens opinions.

Innehållsförteckning

1 Inledning 1

2 Bakgrund 3

2.1 Diskussion av projektet 5
- 2.1.1 Produktbeskrivning
- 2.1.2 Kravspecifikation
- 2.1.3 Målsättning

2.2 Befintliga system 9
- 2.2.1 Active Directory
- 2.2.2 Watchdog-O-matic

2.3 Sammanfattning 10

3 Analys 11

3.1 Användargränssnitt 11

3.2 Modellering 13

3.3 Datalagring 14

3.4 Windows Management Instrumentation 15

3.5 Kommunikation 16

3.6 Designmönster 17
- 3.6.1 Singleton
- 3.6.2 Command
- 3.6.3 Facade
- 3.6.4 Model-View-Controller

3.7 Säkerhet 21
- 3.7.1 Kryptering

3.8 Sammanfattning 23

4 Design och implementation 25

4.1 Utvecklingsmiljö 25
- 4.1.1 Programspråk
- 4.1.2 Integrated Development Environment
- 4.1.3 Versionshantering

4.2 DWS Agent 27
- 4.2.1 Klassen Observer
- 4.2.2 Klassen ProcessHandler
- 4.2.3 Klassen DataManager
- 4.2.4 Klassen LogManager

4.2.5 Klassen NetworkManager
4.2.6 Klassen MainView
4.3 DWS Message Service 36
4.3.1 Klassen MessageService
4.3.2 Klassen MessageRouter
4.3.3 Windows Service
4.4 Data Service 40
4.4.1 Klassen DataService
4.5 Network Service 44
4.5.1 Klassen NetworkService
4.5.2 Klassen ConnectionState
4.5.3 Klassen DataBuffer
4.6 Paketstrukturer 48
4.6.1 Trap
4.6.2 Request
4.6.3 Response
4.7 Trådning 51
4.7.1 Hantering av trådar
4.7.2 Thread-safe
4.8 Kryptering 52
4.8.1 Implementation av kryptering
4.8.2 Implementation av dekryptering
5 Funktionell systembeskrivning 54
5.1 DWS Agent 54
5.1.1 Installation
5.1.2 Användarmanual
6 Slutsats och resultat 61
Referenser 62
Förkortningar 63
A DWS Agent, Användarfall 65
B DWS Message Service, Användarfall 66
C Klassdiagram DWS Agent 67
D Klassdiagram DWS Message Service 68

Figurförteckning

Figur 2-1: Systemtopologi 4
Figur 2-2: DWS Systemarkitektur 5
Figur 2-3: Redundant processkontroll 6
Figur 3-1: Användarmodell vs. Programmodell 11
Figur 3-2: Gränsnitt med metafor och trädststrukur 12
Figur 3-3: Tab prototyp 13
Figur 3-4: DWS Meddelandearkitektur 16
Figur 3-5: Designnmönstret Facade 19
Figur 3-6: Designmönstret MVC 20
Figur 4-1: DWS Agent, reducerat klassdiagram 27
Figur 4-2: DWS Message Service, reducerat klassdiagram 36
Figur 4-3: Trap-paket 49
Figur 4-4: Request/set paket 50
Figur 4-5: Response-paket 50
Figur 5-1: Skärmdump DWS Agent, installation 54
Figur 5-2: Skärmdump DWS Agent, Panelen General 55
Figur 5-3: Skärmdump DWS Agent, Settings 56
Figur 5-4: Skärmdump DWS Agent, Event log 57
Figur 5-5: Skärmdump DWS Agent, Processes 58
Figur 5-6: Skärmdump DWS Agent, processinställningar 59
Figur 5-7: Skärmdump DWS Agent, lägg till en process 60

1 Inledning

Idag är datorer en vardag för alla. Datorer används i det mesta och ett beroende har skapats. Ett beroende som vi själva har skapat utifrån teknikens framfart. En dator eller enskild applikation som inte fungerar som den skall påverkar oss i allra högst grad. Applikationer måste fungera för att garantera de tjänster som lovats. Det finns dock applikationer som är mer känsliga än andra för driftstörningar, som till exempel realtidsbaserade system. Det är dock även allmänt känt att inte alla applikationer alltid är fullt ut pålitliga. Alla har vi träffat på någon applikation som krashat eller inte fungerat som tänkt. Buggar ogillas av alla och utvecklare har till uppgift att eliminera och undvika att skapa buggar. Problemet är dock att programfel kan vara svåra att hitta. Det här projektet är inriktat på att skapa en programvara som övervakar andra applikationer i en distribuerad miljö, så att eventuella driftstörningar kan åtgärdas omgående.

Att utveckla en färdig produkt för att övervaka användarprocesser krävde mycket förarbete och noggrann analys av problemet. Ett möte med uppdragsgivare startade projektet där utvecklingen av ett distribuerat processövervakningssystem diskuterades. Även andra detaljer angående projektet och dess slutgiltiga produkt diskuterades, som till exempel vilken målgrupp som skulle installera, konfigurera och använda produkten. En noggrann undersökning av målgruppens datorvana samt programvaror som redan används hos företaget analyserades. Därefter dokumenterades en kravspecifikation av systemet som önskades och produktens användarfall skapades.

Kapitel 2 tar upp bakgrunden till problemet och vilka andra redan befintliga lösningar som finns på marknaden. Kapitlet beskriver även systemet som utvecklats och vilka applikationer som ingår samt vilka förhållanden de har med varandra. Nästkommande kapitel beskriver allmänna tekniker som krävs för att kunna lösa problemet. Tekniker som underlättar och krävs för att lösa problemet, men även underlättar för den slutgiltiga användaren. Kapitel 4 beskriver systemets design och implementation i detalj. Kapitlet beskriver implementerade komponenter, klasser och centrala och viktiga metoder. Valet av programspråk, utvecklingsmiljö och tekniker som används för att underlätta revisionshantering är även här

beskrivna. Kapitel 5 beskriver installation, hur användargränsittet fungerar och vilka operationer som kan utföras på applikationen.

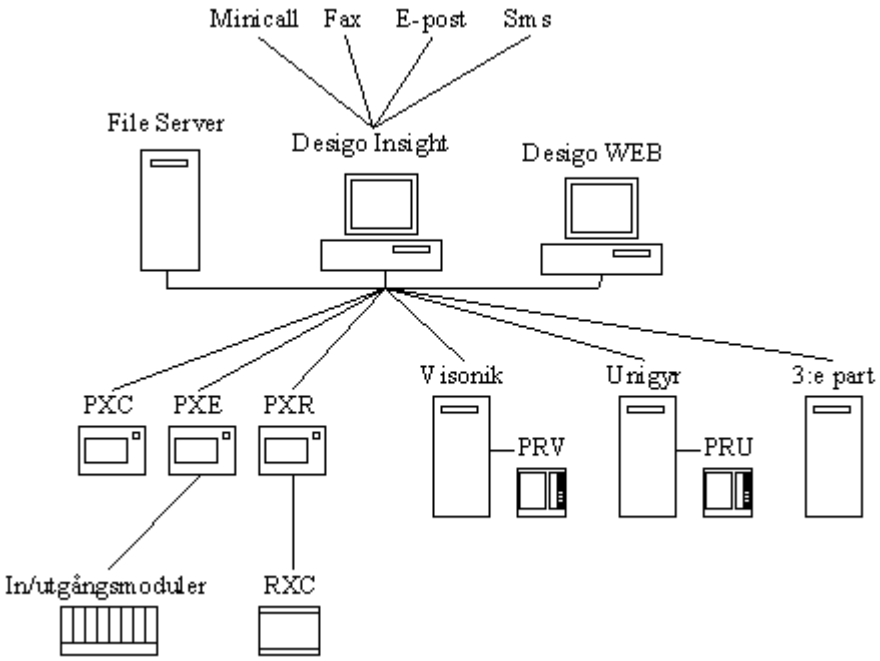
.

2 Bakgrund

Uppdragsgivare är Siemens Building Technologies som ingår i Siemenskoncernen. Siemenskoncernen omsatte totalt 74,2 mdr EURO under affärsåret 2002/2003. Företaget hade då 417 000 anställda världen över. Siemens Building Technologies utgör ett eget affärsområde inom Siemenskoncernen med affärsområdesledningen i Zürich, Schweiz. Affärsområdet omfattar verksamheterna styr-, regler-, säkerhets- och brandlarmsanläggningar. Företagets största utveckling ligger i Schweiz, men mindre utvecklingsprojekt kan även förekomma i andra länder. Sverige som är långt framme i sin utveckling är ett viktigt land för hela koncernen när det gäller produktutvecklingens framfart, feedback och kundönskemål [1].

Siemens Building Technologies består av två olika verksamheter, HVAC Products och Siemens Building Automation. HVAC Products erbjuder ett komplett produktprogram för byggnadsautomation vilket garanterar kvalitet, funktion och kontinuitet. Siemens Building Automation huvuduppgift är byggnadsautomationssystem för integrering av byggnadens tekniska installationer samt översyn av fastigheters drift för att ge förslag på energibesparande åtgärder och komfortförbättringar [2]. Systemen är anpassade till att uppnå bästa möjliga driftsekonomi och komfort i alla typer av byggnader under hela byggnadens livslängd. Användarvänlighet, öppenhet och konstandseffektivitet är viktiga nyckelord för Siemens Building Automation utveckling. Fram till 80-talet existerade få datoriserade styr- och reglersystem. Systemen kunde inte kommunicera och vidarebefordra information till dess övervakare. Behovet samt teknikens framfart gav upphov till att på senare år börja utveckla datoriserade styrsystem samt övervakningssystem. Tekniken har sen dess växt explosionsartat och flera generationer av datoriserade system har utvecklats.

Siemens Building Automation nuvarande datoriserade system och dess topologi illustreras i figur 2-1.



Figur 2-1: Systemtopologi

I det nedersta lagret längst ner i kedjan, finns mindre intelligenta, ej programmeringsbara enheter. Enheterna utför sin uppgift och vidarebefordrar information till lagret ovan. I mitten av kedjan finns mer avancerad programmeringsbara enheter, till exempel PXC. De kommunicerar med underliggande system och behandlar informationen efter vad som implementerats. På det översta lagret finns övervakningssystemet. Med dessa applikationer kan alla underliggande system övervakas och styras. Genom kommunikation med hjälp av olika protokoll, kan andra system än Siemens Building Technologies egna integreras och övervakas. Öppenhet som är en av Siemens Building Technologies är viktigt då kunden ofta endast har ett övervakningssystem till alla styr- och reglersystem, oberoende av leverantör.

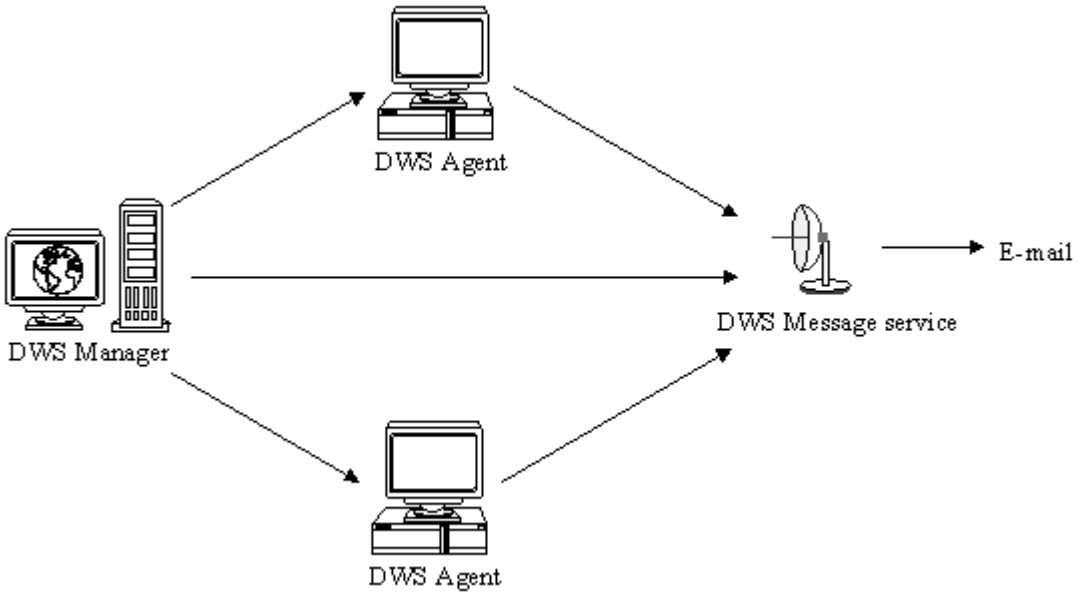
Övervakningssystem högst upp i topologin heter DESIGO™ INSIGHT, och är Siemens Building Technologies flaggskepp. DESIGO™ INSIGHT är ett övervakningssystem, med huvudapplikationer som till exempel larmutforskare, larmdirigerare, trendhanterare, händelsehanterare mm. Trendhanteringen är en viktig del för Siemens Building Technologies kunder. Den utgörs av en databas där uppmätta mätvärden sparas undan vid ett önskat intervall. Kunder, som till exempel mataffärer, måste kunna bestyrka att alla mätvärden, i alla frysdiskar eller kyldiskar, under alla dygnets timmar, är under en bestämd temperatur. Siemens Building Technologies har dock ett problem. Det har visat sig att en av processerna som ligger som grund för hela DESIGO™ INSIGHT funktionalitet kan låsa sig efter en viss tid. Processen förbrukar minne, det vill säga den lider av minnesläckage. Minnesläckagen

leder till att processen låser sig. Låsningen leder till att kunderna inte kan erhålla några mätvärden under den tid som applikationen låst sig. Buggen har visat sig vara svår att åtgärda. Problemet är mycket allvarligt då DESIGO™ INSIGHT är spindeln i nätet. Om DESIGO™ INSIGHT applikationen inte fungerar som den ska, förloras det viktiga informationsutbytet mellan trendhanteringen och underliggande system. Ett sätt att lösa problemet är att ha ett system som övervakar DESIGO™ INSIGHT kritiska process samt att starta om den vid eventuell låsning. Uppdraget är att utveckla ett sådant system.

2.1 Diskussion av projektet

Produkten som har utvecklats heter Distributed Watchdog System, DWS, och är en programvara för att övervaka och åtgärda driftstörningar av användarprocesser i en Windows®-miljö. Genom att skydda processer mot krascher, låsningar och minnesläckage ökas tillförlitligheten och kontinuiteten för systemet. Projektet omfattar tre huvudapplikationer, se även systemarkitektur i figur 2-2:

- DWS Agent
- DWS Manager
- DWS Message Service



Figur 2-2: DWS Systemarkitektur

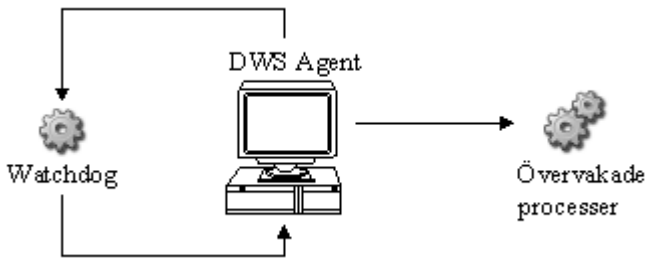
2.1.1 Produktbeskrivning

DWS Agenter kan distribueras över ett flertal datorer som var för sig övervakar sina lokala processer. Agenten kontrollerar kontinuerligt om någon av de utvalda processerna kraschat, låst sig eller fått någon annan driftstörning. Användaren kan på processnivå ställa in vilka åtgärder som ska tas vid krasch eller låsning och om meddelandetjänsten ska meddela kontaktpersoner eller inte.

Meddelandetjänsterna tar hand om inkommande händelser från agenterna och meddelar rätt kontaktpersoner efter ett konfigurerbart schema. Meddelanden kan ske under ett antal olika former, bland annat e-post och i framtiden eventuellt SMS. Schemaläggningen bestämmer vem som ska meddelas, och på vilket sätt, under givna tidsintervall. Händelser kan till exempel vara processkrascher, som inträffar hos en agent.

Agenterna övervakas av en DWS Manager för att kunna centralisera administrationen av det distribuerade systemet. Managern kan administrera utvalda agenter genom ett sammankopplande gränssnitt där man på distans kan se status över respektive agent och dess övervakade processer, samt möjlighet att utföra åtgärder som till exempel stoppa eller starta om processer. Förutom att kontrollera distribuerade agenter har managern hand om att administrera tillgängliga meddelandetjänster i systemet.

Systemet hanterar ett stort antal tekniker som redundant processkontroll, rollbaserade kontaktpersoner och schemalagda händelseutskick via valbart media. Med redundant processkontroll menas att övervakade processer har fler än en övervakare. Processer övervakas av en DWS Agent, som även övervakar sig själv, se figur 2-3. Processkontrollen sker även implicit genom DWS Manager som passivt övervakar de distribuerade agenterna.



Figur 2-3: Redundant processkontroll

På grund av det känsliga informationsutbytet mellan agent, manager och meddelandetjänst har ett antal åtgärder tagits för att säkerställa systemet mot intrång och felaktigt utnyttjande. All kommunikation är krypterad och en manager måste identifieras innan den kan ansluta till en agent. Detaljer om produkten diskuteras senare i kapitel 3 och 4.

2.1.2 Kravspecifikation

Följande krav på produkten beslutades gemensamt med uppdragsgivaren. Utifrån kravspecifikationen har användarfall skapats, se Appendix A och B.

2.1.2.1 Allmänt

- Produktens driftsmiljö skall vara Windows®, version 2000 och senare.
- Användargränsnittet skall vara på engelska för internationell gångbarhet.
- Mjukvara för att kunna installera agent, manager och meddelandetjänst ska finnas tillgänglig på CD vid leverans.
- All kommunikation ska ske över TCP/IP.

2.1.2.2 Agent

- Agenten ska ansvara för att övervaka utvalda processer och kunna återställa dessa vid eventuella driftsstörningar.
- Driftsstörningar omfattar programkrasch och låsning.
- Alla användarprocesser ska kunna övervakas och startas om av agenten.
- Agenten ska logga alla händelser i Windows händelsehanterare.
- Agenten ska kunna meddela meddelandetjänsten vid driftstörningar.
- Användaren ska välja om meddelandetjänsten ska meddelas vid en driftstörning eller inte.

2.1.2.3 Meddelandetjänst

- Meddelandetjänsten ska kunna hantera kontaktpersoner som meddelas vid driftsstörning.
- Meddelandetjänsten ska kunna schemalägga hur och när en kontaktperson ska meddelas.
- En kontaktperson ska kunna meddelas via e-post, med detaljerad information om programfelet (Orsak, tid, åtgärd, resultat).
- Inställning på processnivå om kontaktperson ska meddelas.

- Meddelandetjänsten ska kunna hantera e-post.

2.1.2.4 Manager

- Managern ska kunna ansluta till nya agenter.
- Managern ska kunna övervaka alla anslutna agenter och visa dess övervakande processers status.
- Manager ska kunna ansluta till nya meddelandetjänster.
- Managern ska kunna administrera meddelandetjänsten.

2.1.2.5 Säkerhet

- Nödvändiga åtgärder måste tas för att säkerställa informationsflödet mellan agent, manager och meddelandetjänst.

2.1.2.6 Dokumentation

- Dokumentation skall vara på engelska.

2.1.2.7 Önskemål

- Produkten kan utvecklas i .NET, men är ej ett krav.
- Meddelandetjänsten ska kunna hantera sms.
- Möjlighet att från managern kunna starta om alla övervakade processer hos anslutna agenter.
- Meddelandetjänsten ska logga alla händelser.
- Agenten ska ansvara för att övervaka minnesanvändningen för utvalda processer och kunna återställa dessa.

2.1.3 Målsättning

Målet är att skapa en fungerande lösning för Siemens Building Technologies återkommande problem där den kritiska applikationen slutar fungera. Lösning blir ett distribuerad agent-/manager-system som hanterar eventuella driftstörningar i form av krasch och låsning. Applikationen skulle medföra minskade utgifter i form av ingenjörstimmar som i slutändan

resulterar att företagets resultat förbättras. Stor vikt kommer att läggas på generalisering, skalbarhet och möjlighet att bygga vidare på produkten. Målsättningen är att produkten blir väl använd inom Siemens koncernen och att den även blir användbar även i andra sammanhang än problemet som Siemens Buildning Tecknologies har idag.

2.2 Befintliga system

Behovet av att kunna övervaka och administrera datorer och dess status har funnits en längre tid. Därav finns det redan många olika system på marknaden med möjlighet att övervaka och administrera datorer under Windows miljö. Systemen går från mycket enkla applikationer till mer komplexa och avancerade system. Exempel på sådana system kommer att nämnas och diskuteras varför Siemens inte valt några av dessa.

2.2.1 Active Directory

Actice Directory är en integrerad, distribuerad service som är inkluderad i Microsoft Windows Server 2003 och Microsoft Windows 2000 Server [3]. Applikationen innehåller en databas med information om användare och datorer på nätverket. Den gör det lätt för systemadministratörer att övervaka och administrera hela nätverket med dess objekt. Applikationen är mycket komplex och kraftfull och kan tilldela olika rättigheter för användare och datorer. Applikation har dock inte de funktionaliteter Siemens Building Technologies önskar. Stöd såsom att automatiskt starta om kraschade processer existerar inte i Active Directory. Däremot finns möjligheten att manuellt övervaka processer och starta om dem manuellt. En annan nackdel med Active Directory är komplexiteten. Applikationen är för stor för Siemens Building Technologies behov och medför därmed också en alldeles för stor kostnad. En tredje nackdel med Active Directory är att den endast kan administrera datorer på sitt eget nätverk. Siemens Building Technologies placerar ut datorer runt hela Sverige och därmed inom olika företags lokala nätverk. Siemens Buildning Technologies har inte, och kommer inte att få mer rättigheter inom andras företags nätverk än vad de har idag. De rättigheter som skulle behövas för att använda sig av Active Directory kommer aldrig Siemens Buildning Automation att få tillgång till. Därmed passar inte Active Directory som applikation för att lösa problemet.

2.2.2 Watchdog-O-matic

Kwakkelflaps programvara är utvecklad för att övervaka kritiska processer [4]. Applikationen har stöd för att automatiskt starta upp kraschade processer. Den har även stöd för att övervaka användarprocessernas tillstånd, samt starta om dem om ett onormalt tillstånd inträffat. Watchdog-O-Matic stödjer dock inte alla Siemens Building Automations krav. Applikationen är inte någon agent/manager applikation. Därmed finns det ingen möjlighet att centralt kunna övervaka valda användarprocesser hos datorer utplacerade på nätverket. Därmed stödjer inte Watchdog-O-Matic alla krav. Siemens Building Technologies har även provat denna programvara och är inte nöjda med den befintliga funktionaliteten. Watchdog-O-Matic är en enkel programvara som installeras lokalt på varje dator.

2.3 Sammanfattning

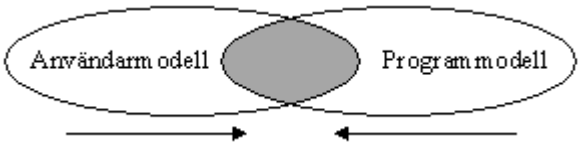
Siemenskoncernen har ett av Europas ledande bolag inom styr- och reglerteknik. Företaget har problem med att applikationen DESIGO™ INSIGHT låser sig. På uppdrag från Siemens Building Technologies ska produkten Distributed Watchdog System utvecklas. Produkten ska kunna övervaka valda användarprocesser i utvalda datorer. DWS kommer leda till att DESIGO™ INSIGHT kan prestera till fullo över en längre tid. Det finns många system som är framtagna för att övervaka och administrera datorer, men Siemens Building Technologies har ändå inte lyckats finna någon passande applikation som löser problemet. Även om produkten DWS är anpassad för Siemens Building Technologies specifika krav är den framtagen för att kunna användas i ett generellt syfte. Eftersom produkten är distribuerad har många säkerhetsfrågor uppstått.

3 Analys

En viktig del i projektet är att analysera alla dess delar och problem. I detta kapitel kommer allmänna tekniker som krävdes för att lösa uppgiften att diskuteras och förklaras. Tekniker som underlättar implementationen men även för den slutgiltiga användaren.

3.1 Användargränssnitt

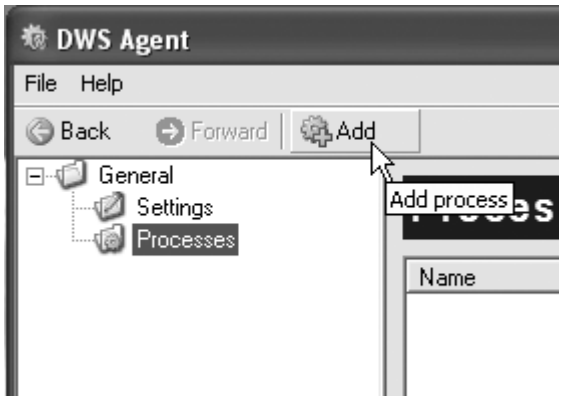
Målet med DWS användargrässnitt är ett mycket enkelt gränssnitt. Ett gränssnitt som slutanvändarna känner sig trygg med och inte blir avskräckt av. Avancerad och allt för krånglig funktionalitet har undvikits. Syftet är att användbarhetsgraden ska bli så hög som möjligt. En hög användbarhetsgrad minskar inlärningstiden och ökar användarens tillfredställelse. Målet är att minska avståndet mellan användarmodellen och programmodellen, se figur 3-1.



Figur 3-1: Användarmodell vs. Programmodell

Användarmodellen är den funktionalitet som användaren förväntar sig av applikationen. Programmodellen representerar hur applikationen fungerar. Användbarhetsgraden ökar desto närmare användarmodellen och programmodellen närmar sig varandra, eftersom programmet fungerar precis som användaren förväntar sig. För att uppnå detta har vi använt oss av metaforer i gränssnittet. En metafor kan till exempel vara en bild, med en viss funktionalitet, i ett gränssnitt som användaren känner igen från det verkliga livet. Exempel på en metafor i gränssnittet är plustecknet på knappen som används när man ska lägga till en process. Plustecknet har en viss symbolik som användaren kan härleda från det verkliga livet, se figur 3-2. Affordance är också en teknik som används för att öka användbarhetsgraden. Det är en

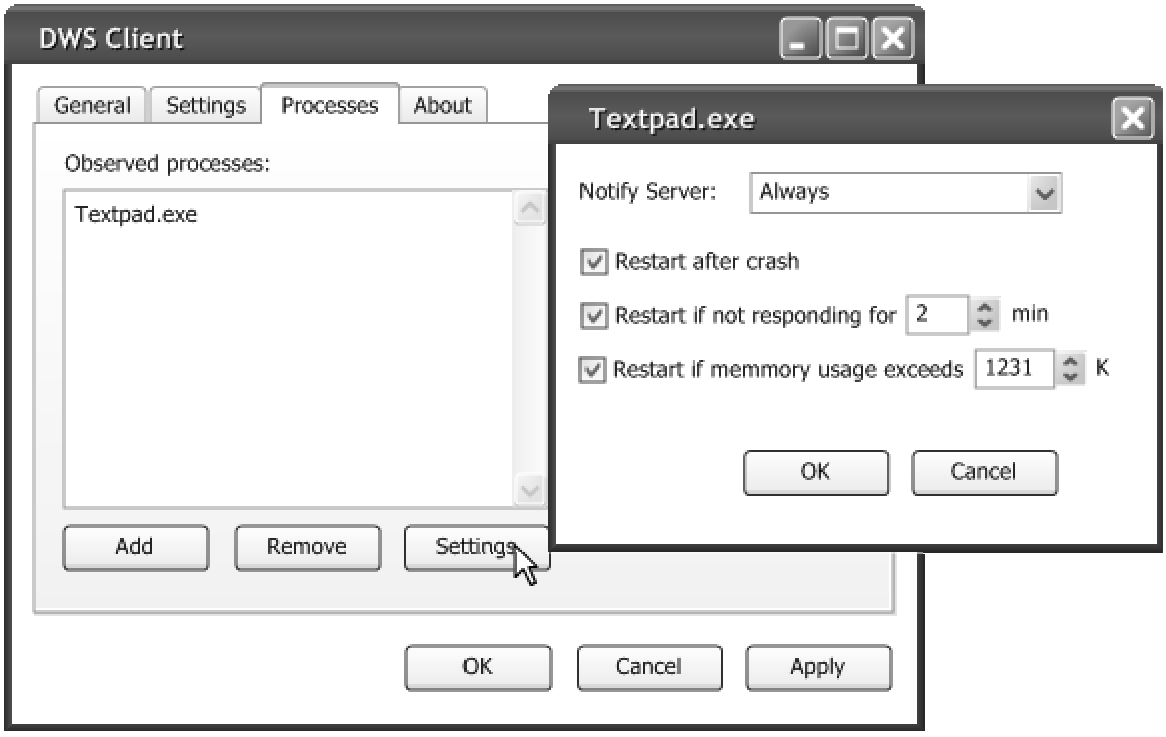
inbjudan till användning av funktionalitet. Bara genom att titta på gränssnittet, vet användaren hur man ska gå tillväga. Vi använder oss av Windows standardkomponenter vars affordance är väl utprovade. Affordance och metaforer är inte endast nyckeln till ett framgångsrikt användargränssitt. Det är också viktigt att en bra dialog med slutanvändaren. Att utvecklarna tycker designen är bra resulterar inte alltid i att slutanvändaren tycker den är bra. Uppdragsgivaren har på ett tidigt stadium utvärderat designen och kommit med förslag och ändringar. Förslagen kommer att resultera i en bättre produkt i slutändan.



Figur 3-2: Gränsnitt med metafor och trädststrukur

Vid utveckling av den första prototypen fanns valet mellan att använda ett träd eller ett fliksystem för att navigera mellan olika paneler. Båda lösningarna är inte obekanta för slutanvändaren. Tillsammans med uppdragsgivaren valdes dock att använda ett träd för att växla mellan olika paneler. Trädet har fördelar, då träd kan representera hierarkisk data. Uppdragsgivaren ansåg också att ett träd såg bättre ut designmässigt samt att produkten fick en mer professionell känsla, jämfört med tabbar. Trädet ger också en bra översikt över alla paneler. För att byta panel i det högra fältet klickar man helt enkelt på den nod som man vill se information om. Trädet är också mer dynamiskt än ett fliksystem. Att lägga till noder i ett träd anses vara helt naturligt medan att ändra på fliksystemets uppbyggnad kan uppfattas som förvirrande. En annan anledning till att träd valdes som navigeringslösning är att det inte leder till att fönster överlappas, i undantag till när en ny process ska läggas till för att övervakas. När en användare vill ändra på en observerad process inställningar klickar den på processen i trädet för att växla till dess inställningspanel. Motsvarande funktionalitet med en tabblösning illustreras i figur 3-3. Detta skulle medföra ett nytt fönster för att ändra på processens inställningar samt att försvåra navigeringen mellan olika processers inställningar. Det är också viktigt att få ett konsekvent utseende. Designen inom en applikation måste ha ett konsekvent utseende mellan olika paneler. I och med att produkten omfattas av tre olika applikationer är

det också viktigt att designen mellan dessa tre är konsekvent. Designen kommer då att leda till att de tre applikationerna knyts samman till en enda produkt utseendemässigt. Den slutgiltiga designen finns dokumenterad i kapitel 5.



Figur 3-3: Tab prototyp

3.2 Modellering

UML diagram har legat till grund för utvecklingen. Diagrammen är ett mycket kraftfullt hjälpmedel vid utveckling av en komplex applikation. De hjälper till att visualisera viktiga aspekter av systemet. UML hjälper till med att fånga upp uppdragsgivarens krav. Diagrammen underlättar om uppdragsgivaren ändrar något krav under utvecklingens gång, då UML diagrammen är ett gemensamt kommunikationsspråk mellan utvecklare och uppdragsgivare. Modulering med UML diagram används flitigt hos ett flertal utvecklingsmetodiker, som till exempel Unified Process [9]. Det är viktigt att framhäva att Unified Process inte har använts fullt ut. Unified Process är en god utvecklingsmetodik men skulle för oss mer stjälpa än hjälpa. Anledningen till detta är först och främst att erfarenheten av utvecklingsmetodiken inte är så god, samt att utvecklingsmetodiken passar bättre in på större och mer komplexa projekt. Uppdragsgivaren har heller inte ställt några specifika krav på vilken utvecklingsmetodik som ska följas. Därmed finns en stor valfrihet av hur

utvecklingen skall framskrida. Valda delar av olika utvecklingsmetodiker använts. Uppdragsgivarens krav moduleras med hjälp av användarfall. Även klassdiagram har använts under utvecklingen. Parprogrammering, som är en del av eXtreme Programming, har använts i utvecklingen av DWS [10]. Att programmera i par underlättar kodningen på flera olika sätt. Bland annat bidrar det till delning av kunskap, det vill säga ett pars erfarenhet kan utnyttjas på ett effektivare sätt, än om inte parprogrammering används. Koden är heller inte uppdelad i olika ägande delar, dvs. alla har rätt till att ändra i koden utan att be någon om lov. Det är dock viktigt att alla berörda får reda på om någon ändring har gjorts.

3.3 Datalagring

Att lagra data förekommer i så gott som alla applikationer och är en viktig del i hur slutprodukten kan prestera. Vissa tillämpningar ställer högre krav än andra på säkerhet, enkelhet och prestanda vid lagring av information. Andra tillämpningar begränsas av yttre faktorer, till exempel information från andra system eller att de måste anpassas för andra system. Val av datalagring kan även påverkas av ekonomiska faktorer då många av dagens mer avancerade lösningar för datalagring har kostnader som överstiger applikationen som utvecklas. Bland dagens möjligheter finns till exempel databaser, katalogtjänster och olika formaterad data lagrad direkt i filsystemet.

Databaser är ett vanligt sätt att lagra större mängder information och tillhandahåller många fördelar inom säkerhet och skalbarhet så väl som prestanda. Informationen är lätt att uppdatera, både lokalt och över ett nätverk, och den mesta logiken för lagringen är flyttad från den egna tillämpningen till databasen. Den största nackdelen för att använda databas som lagringssystem är de höga kostnaderna till en tredje part. I dagsläget finns det dock ett antal större databaser med öppen källkod som har i stort sett all funktionalitet av de större kommersiella systemen.

En katalogtjänst är en typ av databas speciellt framtagen för sökning och läsning av hierarkisk information. Likt gulasidorna i en telefonbok kategoriseras information upp efter sin identitet. Om man till exempel vill ha tag i en järnhandlare letar man först upp telefonboken, sen letar man efter industri och transport, sen bestämmer man vilken typ av underområde man är ute efter i det här fallet byggindustri, och så vidare.

Till sist kan lagring skötas helt själv av den egna tillämpningen. Det görs oftast genom att data formateras, antingen fritt eller efter någon given standard, och sparas manuellt i

filsystemet. I mindre applikationer kan den här metoden vara att föredra då den är enkel och förhållandevis snabb. Applikationen blir inte heller beroende av någon tredje parts programvara, och man slipper strukturell design av databasen. En nackdel kan vara att mycket av logiken för datahantering måste implementeras för hand, som till exempel sökning av olika slag. I de flesta modernare programspråk finns det däremot stöd för att underlätta hanteringen av formaterad information, där bland XML.

I agenten är informationen formaterad i XML. Den största motiveringen till det designbeslutet var att inte skapa beroende av någon yttre datakälla. För den lilla informationsmängd som DWS behöver kunna hantera är det onödig komplexitet. Även kommunikationen mellan enheterna i DWS distribuerade nät kommer delvis att ske i XML, och det underlättar att formatera den lagrade informationen därefter. Alternativa lösningar kan vara Microsoft Access, vars datafiler kan användas fristående och anslutas till via en standardiserad datahanterare som till exempel ODBC eller dess efterföljare OLEDB. Den lösningen används i DWS meddelandetjänst då den ställer högre krav på datalagring. Microsoft .NET Framework har funktionalitet för databaser lagrat direkt i minnet. Det kallas DataSet och kan lagra tabeller, relationer och vyer av olika slag. DataSet har dessutom stöd för att spara ner både struktur och data som XML. Det är alltså möjligt att använda DataSet-klassen med både Microsoft Access och XML. Katalogtjänster var givetvis inte aktuellt eftersom den information som ska lagras varken är hierarkisk eller tillräckligt omfattande för att motivera den typen av lagring.

3.4 Windows Management Instrumentation

Microsoft har sedan Windows 2000 inkluderat en särskild databas i sina operativsystem som innehåller information om hårdvara, mjukvara och status över systemet. Informationen kan man komma åt via ett gränssnitt WMI, Windows Management Instrumentation, som är Microsofts implementation av WBEM, Web-Based Enterprise Management. WBEM är en standard från Distributed Management Task Force, DMTF, och specificerar hur systeminformation görs åtkomlig i ett nätverk [8].

Exempel på information i WMI är tillgängliga nätverkskort i systemet, version nummer på Windows och status över exekverande processer. Det sistnämnda är väldigt användbart i DWS för att hämta information om övervakade processer. Ramverket i Microsoft .NET har också klasser för att komma åt liknande information om processer, men är mer begränsade.

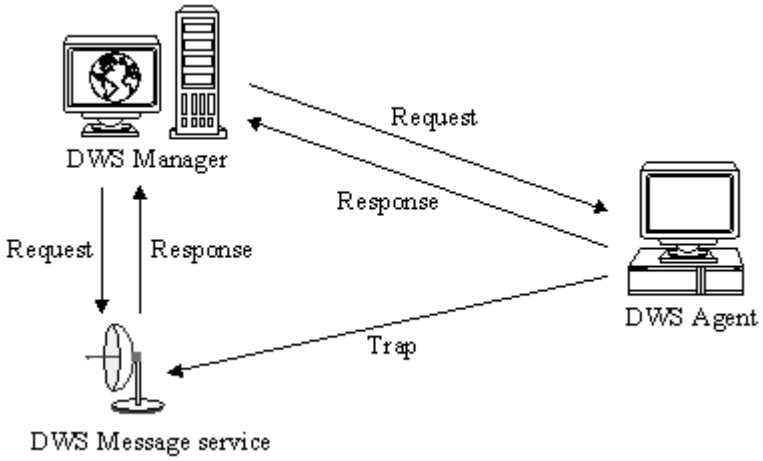
Informationen i WMI är tillgänglig genom ett speciellt framtaget språk WQL (WMI Query Language) som är direkt framtaget från ANSI SQL. Syntaxen är i princip den samma och WQL-frågorna returnerar objekt från WMI-databasen. I Microsoft .NET finns klasser för att ta hand om den här typen av anrop.

Jämfört med .NET är det avsevärt mer komplicerat att ta fram information om processer med WMI. Även en mindre förlust i prestanda kan upplevas, inte minst första gången en fråga ställs till WMI-databasen eftersom WMI-processen måste startas. DWS använder både WMI och .NET för att hämta information om övervakade processer.

3.5 Kommunikation

Datakommunikation är ett stort område som kräver noggrann eftertanke vid design för att ett system ska klara de krav som ställts på applikationen. Val av protokoll är viktigt för att en applikation ska få rätt kvalité på de tjänster den är ämnad för. För en agent- / manager-applikation som DWS finns det ett antal alternativ för hur kommunikationen kan skötas.

Antingen kan agenten rapportera in status till managern på givna intervall. Det kräver att alla agenter vet om adressen till varje manager som ska hantera respektive agent. En nackdel är att mycket onödig information kommer skickas. Ett bättre sätt är att låta managern begära status från de agenter som den för tillfället är intresserad av. Agenten måste dock ha möjlighet att skicka brådskande meddelanden till meddelandetjänst. DWS meddelandearkitektur illustreras i figur 3-4.



Figur 3-4: DWS Meddelandearkitektur

Den observante läsaren ser snart att arkitekturen i DWS liknar SNMP, Simple Network Management Protocol [7]. Det är inte av en händelse att DWS har lånat upplägget av informationsflödet från SNMP. SNMP är en standard för att övervaka och administrera noder på Internet och har funnits sedan 1980-talet. Precis som i figur 3-4 använder SNMP request och response-meddelanden för att kommunicera med noder i nätet. Ett tredje meddelande, kallat trap, används av noderna för att kunna meddela managers om händelser samtidigt som de inträffar. Trap-meddelanden ger lägre overhead i nätet eftersom en nod talar om när det finns information tillgänglig istället för att en manager ska behöva fråga med jämna mellan rum om något har hänt. En annan fördel med att använda trap-meddelanden är att det inte blir någon fördröjning mellan en händelse och då en mottagare får information om den inträffade händelsen. Med request-/response-meddelande är det alltid en tidslucka mellan varje request. Om något skulle inträffa strax efter en request kan det dröja lång tid tills nästa request då en mottagare får svar.

Till skillnad mot SNMP som transporterar meddelanden över UDP, sker transporten i DWS över TCP. Det underlättar implementationen då meddelanden inte behöver sparas undan tills mottagaren bekräftat meddelandet som mottaget. Bekräftelser och omskickningar sköts istället av transportlagret.

3.6 Designmönster

Ett designmönster är en uppsättning återanvändbara, effektiva och generella designlösningar. Målet med designmönster är att återanvända väl beprövade lösningar för problem som ofta är förekommande. Ett designmönster består av en abstraktion av ett generellt problem, en beskrivning av lösning och en beskrivning av vilka konsekvenser som den har. Ett designmönster är oftast programspråksoberoende, men det finns även mönster som är anpassade för ett visst programspråk eller ramverk, som till exempel .NET. Genom att använda sig av ett känt mönster blir problemlösningen lättare samt lösning enklare och elegantare [6]. Ett dokumenterat mönster förmedlar kunskap mellan experter och noviser. Det är bättre att lära sig av andras erfarenhet än av sina egna misstag.

Det finns tre huvudgrupper inom designmönster:

- Skapande designmönster
- Strukturerade designmönster
- Beteende designmönster

Skapande designmönster hanterar instansiering av objekt och hur objekten konfigureras. Strukturerande designmönster berör hur objekt sätts samman till en större struktur. Beteende designmönster hanterar algoritmer och tilldelning av ansvar mellan objekt och dynamiskt samspel mellan objekt.

DWS använder sig av ett flertal kända designmönster och några av dessa kommer att diskuteras i detta kapitel.

3.6.1 Singleton

Designmönstret singleton hör hemma i gruppen skapande designmönster och är ett av de mest välkända designmönster. Designmönstret ser till att endast en instans är möjlig av en klass. En matematisk definition på singleton är en mängd med endast ett element. Om mängden innehåller fler än ett element bryter mängden från singleton definitionen. Designmönstret singleton garanterar att en klass endast har en instans och förser resten av applikationen med en global accesspunkt för att ge tillgång till den. Klasser som ofta använder sig av singleton är de som har ansvarsområdet datahantering, loggning och kommunikation. Med hjälp av designmönstret singleton kan applikationens robusthet förbättras.

Designmönstret singleton kan implementeras på olika sätt. Man kan implementera en singleton-klass trådsäker eller icke trådsäker. Valet av implementationssätt anpassas efter det behov som applikationen kräver. Målet är att dock att alltid upprätthålla singleton-konceptet.

DWS använder singleton vid ett flertal tillfällen, som till exempel den klass som har hand om loggning, samt den som har hand om datahantering. Agent innehåller flera egendefinierade trådar. En flertrådig applikation innebär att nödvändiga åtgärder har vidtagits för att säkerställa att endast ett objekt existerar.

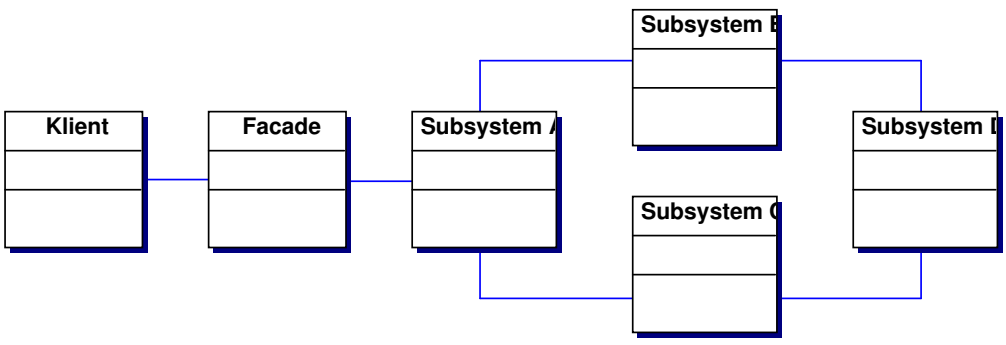
3.6.2 Command

Vanligen när man vill exekvera en funktion är att anropa funktionen. Det finns dock tillfällen då man vill kontrollera när och i vilken ordning funktionen anropas i relation till övriga händelser i programmet. Mönstret Command tar hand om det problemet genom att lagra information om funktionen och hur den ska anropas i ett objekt som kan sparas för att senare anropas av andra delar av programmet när det är mer lägligt.

I användargränssnittsprogrammering används ofta Command-mönstret, till exempel när man vill koppla kod till en extern händelse så som en knapptryckning. Andra tillfällen där mönstret är intressant är när man behöver exekvera en viss funktion efter en resurs har gjorts tillgänglig men innan den stängts. Ett bra exempel på det är vid anrop till databaser, då en anslutning måste upprättas innan användaren kan ställa frågor men där det är viktigt att stänga anslutningen efter anropets slut. Hur Command kan implementeras syns tydligt i metoden LendReader i kapitel 4.5.1.5.

3.6.3 Facade

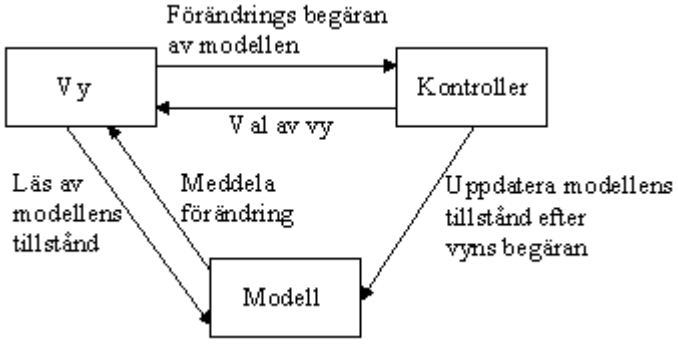
Facade skall alltid användas då en annan produkt eller annat programspråk används, som till exempel SQL eller operativsystemsanrop. Designmönstret har till uppgift att förenkla gränssnittet mot ett eller flera subsystem, se figur 3-5. Facade skapar en klass som representerar ett gränssitt mot subsystemet. Den döljer subsystemets komplexitet och kapslar in det på ett strukturerat sätt. Inkapslingen kan innehålla en eller flera klasser. Facade klassen blir den enda klass som anropar subsystemet. Den håller reda på vilka objekt som är ansvariga för vad och utför operationerna på dessa. Detta medför att facaden är den enda klass som påverkas om subsystemet byts ut till ett annat eller om dess definitioner ändras. Det övriga överliggande systemet blir således oberoende av subsystemets publika gränssitt. Denna uppdelning innebär att det blir enklare att implementera systemet. Det minskar även kopplingen, eftersom att det överliggande systemet inte blir kopplat till subsystemet, vilket alltid är ett mål. Komponenterna Data Service och Network Service använder facade, läs kapitel 4.4 och 4.5.



Figur 3-5: Designnmönstret Facade

3.6.4 Model-View-Controller

Designmönstret Model-View-Controller, MVC, hör hemma i gruppen strukturerade designmönster. MVC ligger dock på en högre abstraktnivå än de övriga designmönster som beskrivits. DWS Agent använder sig av detta designmönster. Designmönstret ger en tydlig uppdelning mellan tre olika problemområden, nämligen vy, modell och kontroller. Designmönstret illustreras i figur 3-1.



Figur 3-6: Designmönstret MVC

Modellen representerar kärnan av problemet. Den är spindeln i nätet och innehåller affärslogiken. Modellen brukar oftast moduleras och implementeras först av de tre ansvarsområdena. Den innersta kärnan av modellen känner inte till de mekanismer som exponerar dess tillstånd för resten av omvärlden. Genom att modellen löser ut events får alla tillhörande vyer reda på om dess tillstånd har ändrats.

Vyn är det användaren ser och speglar modellens tillstånd. Vyn känner till både modellen och kontrollern. Genom att fånga upp modellens utlösta events, får den reda på dess tillståndsförändringar. Om vyn, användaren, vill ändra på modellens tillstånd måste vyns begäran gå via kontrollern.

Kontrollern känner till både vyn och modellen. Den tar emot begäran av förändring från vyn samt kontrollerar modellens förvillkor. Om förvillkoren är uppfyllda anropar kontrollern modellen och förändringen genomförs. Kontrollern kan även förändra vyn, till exempel om användaren loggat in som administratör eller som en vanlig användare. Vyn och kontrollern kan kombineras till ett ansvarsområde om behovet finns.

Det finns både fördelar och nackdelar med att använda sig av MVC. En av nackdelarna är att det skapas mycket trafik mellan vy och modell vid små tillämpningar, samt att MVC kan vara omständligt vid små projekt. En annan nackdel är att det kan vara svårt att särskilja rollerna mellan ansvarsområdena. Fördelen med MVC är att man kan få ner kopplingsgraden.

Kopplingsgraden är ett mått på vilka beroenden olika komponenter har emellan varandra. Detta är en viktig del, då onödig stor kopplingsgrad ökar risken för att projektet misslyckas. En annan fördel är att utvecklingen kan ske parallellt. Användargränssnittet kan utvecklas av ett team, samtidigt som modellen utvecklas av ett annat. Andra fördelar med MVC är att det hjälper till att inkapsla funktionalitet, samt att det skapar en dynamisk bindning mellan vy och modell. Med dynamisk bindning är det lätt att byta ut någon av de tre komponenterna, utan att de påverkar någon annan. DWS Agent använder sig av MVC då fördelarna väger upp nackdelarna.

3.7 Säkerhet

Den viktigaste säkerhetsaspekten man måste tänka på i ett distribuerat processövervakningssystem är att ingen obehörig person kan ta kontroll över en agent via kommunikationskanalen till dess manager, och där igenom få tillgång till kommandon för att störa det övervakade systemet. Lokal säkerhet på agent respektive manager hanteras av det underliggande operativsystemet då ytterligare ett säkerhetslager inte ses nödvändigt eftersom systemet inte ger mer rättigheter än den inloggade användaren skulle ha från början.

Attacker mot systemet i form av överbelastning, DoS (Denial of Service), skulle kunna slå ut antingen kommunikationen för en enskild agent och på så sett hindra den från att meddela eventuella fel, eller kommunikationen för en meddelandetjänst och hindra alla agenter möjligheten att meddela viktig information åt systemansvariga. I en sådan händelse är agenten fristående och får förlita sig på sin egen förmåga att åtgärda fel i systemet. Att lägga den funktionaliteten i agenten och inte låta andra delar i systemet initiera alla åtgärder är viktigt, inte bara för att klara av attacker utan även för att skydda sig mot andra nätverksfel.

3.7.1 Kryptering

En viktig säkerhetsaspekt för att säkerställa kommunikationskanalen mellan DWS Agent, DWS Manager och DWS Message service är att kryptera trafiken mellan enheterna. Känslig information skickas mellan enheterna och kryptering gör det avsevärt svårare att avlyssna trafiken. Det finns två olika krypteringparadigmer, symmetrisk- och asymmetrisk kryptering. Fram till början av 70-talet fanns endast symmetrisk kryptering. Krypteringsmetoden går ut på att alla inblandade parter som kommunicerar med varandra äger samma nyckel. Samma

nyckel används vid kryptering respektive dekryptering. När mottagaren tar emot informationen, dekrypterar den det med samma nyckel som sändaren. Nyckelns längd beror på vilken krypteringsalgoritm man använder. Vissa krypteringsalgoritmer tillåter även programmeraren/användaren att välja nyckels längd. Fördelen med symmetrisk kryptering är att metoden är snabb. Nackdelen är distributionen av nyckeln, samt att nyckeln existerar mer än på ett ställe. Ju fler ställen nyckeln existerar på, desto större sannolikhet att den kommer på villovägar. Det asymmetriska krypteringsparadigmet skiljer sig åt genom att det finns två skilda nycklar. Den ena nyckeln är en privat nyckel som endast en part har och den andra en public nyckel som alla har tillgång till. Vid kryptering av information krypterar sändaren med mottagarens publika nyckel. Mottagaren tar sedan emot informationen och dekrypterar med sin privata nyckel. Asymmetrisk kryptering öppnar även upp andra möjligheter som inte kommer att diskuteras i rapporten.

Vid en driftstörning av en övervakad process skickas information från DWS Agent till DWS Message service. Den kritiska processen kan vara en brandvägg eller annan applikation som säkerställer konfidentialiteten. För att undvika att ingen obehörig får kunskap om att den kritiska processen inte fungerar som den ska, krypteras kommunikationsflödet mellan enheterna. Krypteringen är dock inte endast till för att försvåra avlyssning av informationsflödet mellan parterna. Den är även till för att ingen obehörig ska kunna styra DWS Agenten eller DWS Message service.

Data Encryption Standard, DES, är en symmetrisk kryptering som publicerades för första gången 1977. Den uppdaterades 1993 för kommersiellt bruk. DES nyckellängd består av 64 bitar, varav 8 bitar som är till för paritet. Den effektiva nyckellängden består således av 56 bitar. DES anses inte vara säker, då data som krypterats med algoritmen kan dekrypteras genom att testa alla möjliga varianter av nycklar.

DWS använder sig av triple-DES (3DES). Tripple-DES anses vara säkrare och använder sig av tre oberoende nycklar. Vid kryptering, krypteras informationen med första nyckeln, dekrypteras med andra nyckeln och till sist krypteras med den tredje. Algoritmen är också relativt lätt att implementera. Valet av krypteringsparadigm beror endast på att det är relativt lätt att implementera triple-DES och att det finns stöd för algoritmen i .NET.

3.8 Sammanfattning

DWS använder ett antal kända designmönster. Mönstren har underlättat utvecklingen, då de är väl beprövade och dokumenterade. Designen av en programvara är en viktig del och ligger till grundutvecklingen. Att använda sig av ett designmönster underlättar problemlösning och skapar goda förutsättningar för att projektet lyckas.

Vid utveckling och design av användargrässnittet är det viktigt att användaren sätts i första hand. Ett lättförståligt gränssnitt ökar användarens tillfredsställelse samt underlättar för användaren att sätta sig in i programvaran. Målet med DWS användargränssitt är det ska vara lättförståligt, snyggt, samtidigt som det har ett professionellt utseende.

Modulering med hjälp av UML-diagram har underlättat att visualisera viktiga detaljer av systemet som utvecklats. Parprogrammering används genom hela projektet. Att programmera i par underlättat implementationen. Ingen speciell utvecklingsmetodik har används, då delar av flera olika metodiker har tillämpats.

Windows Management Instrumentation underlättar implementationen, då databasen innehåller information om systemet. WMI ger en mindre förlust i prestanda, dock endast första gången WMI-databasen används efter en omstart av systemet.

Design av hur kommunikation mellan olika enheter har noga analyserats samt olika metoder har diskuterats. DWS har lånat upplägget av SNMP. SNMP är ett protokoll som är en standard för att övervaka och administrera noder. En stor skillnad mellan SNMP och DWS är valet av protokoll. TCP kommer att användas då det underlättar implementationen.

DWS systemet skall övervaka kritiska processer som är viktiga för Siemens Building Technologies. Det är viktigt att agenten är fristående och inte beroende av någon yttre applikationen. Säkerhetsaspekter har diskuterats och informationsflödet mellan enheterna krypteras för att försvåra att känslig information hamnar i fel händer.

4 Design och implementation

I det här kapitlet är DWS design och implementation beskrivet. Val av utvecklingsmiljö samt viktiga designbeslut redogörs. Tyngpunkten ligger dock på centrala implementationsdetaljer för DWS Agent och DWS Message Service. DWS Manager är i dagsläget inte slutfört och kommer därför inte redogöras.

4.1 Utvecklingsmiljö

Vid utveckling är valet av utvecklingsmiljö och programspråk en viktig del och kan påverka resultatet av projektet. Även ett bra versionhanteringssystem underlättar utvecklingen avsevärt, då flera personer är inblandade i utvecklingen.

4.1.1 Programspråk

DWS är implementerat i programspråket C#. .NET-platformen stödjer även ett flertal andra språkdialekter som till exempel Visual Basic, C++, J#. För att göra det möjligt måste ett dialektregelverk finnas som specificerar hur datatyper och andra språkelement får och kan användas. Regelverket har fått namnet ”Common Language Specification”, CLS [7].

C# är Microsofts senaste programmeringsspråk och är speciellt framtaget för .Net och dess klasser. Språket härstammar från många olika programspråk, där det bästa från varje språk är tagit. C# är ett modernt, typsäkert och objektorienterat språk. Det är designat för att skapa robusta och bestående komponenter i det verkliga livet. C# är ett språk som nybörjaren lättare tar till sig, jämfört med C++. Pekare kan användas men då måste koden vara inom ett block som markeras som osäkert (Un-Safe-Block). C# tillåter heller inte multipla arv, likt Java, däremot stöds multipla implementationer av Interfaces. Grundsyntaxen är lik C/C++, men på en högre nivå liknar C# mer Java. Likt Java så ärver alla objekt från klassen Object. En stor skillnad mellan Java och C# är att C# använder sig av .NET:s ramverk och Java använder sig

av Java:s. C# har även fler primitiva datatyper än Java. En annan skillnad mellan Java och C# är att de primitiva datatyperna i Java inte ärver från basklassen Object som de gör i C#.

C# har, som Java, en tjänst som automatisk frigör minne som inte längre används, en så kallat skräphantering (Garbage collector). Tjänsten sköter helt och hållet när minne ska deallokeras. Den automatiska deallokeringen underlättar för programmeraren samt ökar robustheten hos applikationen. .NET Common Laonguage Runtime sköter skräphanteringen och är gemensam för alla .NET applikationer [7].

Vid kompilering av kod, producerar inte en C#-kompilator en CPU-specifik kod. Likt Java blir resultatet av en kompilator en mellankod. Syftet med det är att göra program oberoende av programmeringsspråk. Koden kallas för Microsoft Intermediate Language, MSIL. När väl koden ska exekveras används en virtuell maskin som är anpassad för en viss plattform. Det finns möjlighet att direkt programmera applikationerna i MSIL, men är inte något att rekommendera.

4.1.2 Integrated Development Environment

Det finns ett antal olika utvecklingsmiljöer för .NET men den största och mest använda är Visual Studio.NET från Microsoft. Den tillhandahåller en enda miljö för alla typer av .NET projekt oavsett om det är Windows-applikationer, Windows-tjänster eller webbtillämpningar. Förutom en bra projektlösning klarar den avancerad felsökning (debugging) som underlättar utvecklingen. En gränssnittsdesigner för att snabbt ta fram grafiska användargränssnitt, finns tillgänglig och som så många andra utvecklingsmiljöer har även Visual Studio IntelliSense, det vill säga en funktionalitet som föreslår tillgängliga funktioner och parametrar. IntelliSence snabbar upp kodningen avsevärt.

En stor nackdel med Visual Studio är att dess licenskostnader är höga. I utvecklingen av DWS användes till fördel den studentlicens som fanns tillgänglig på Karlstads universitet.

4.1.3 Versionshantering

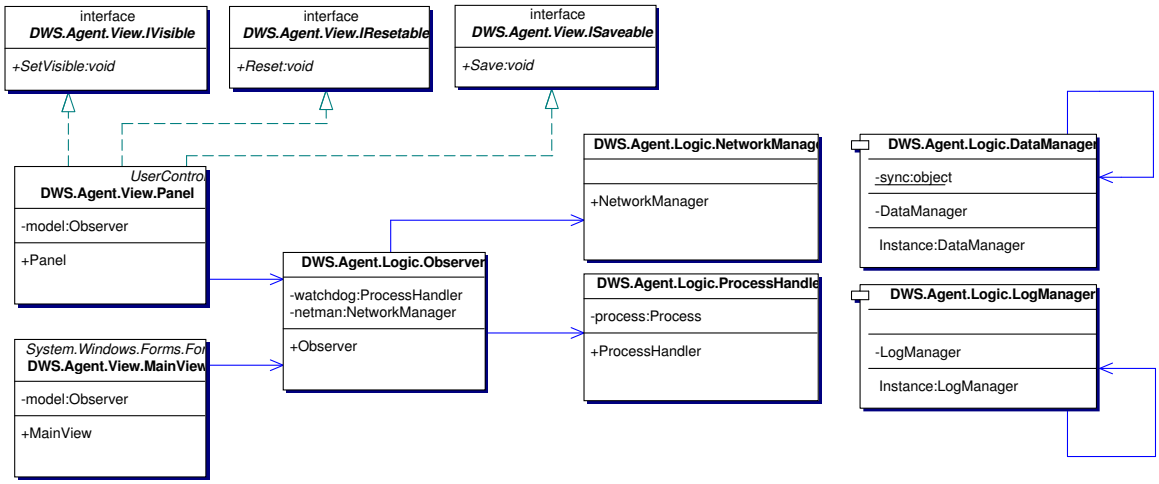
Versionshantering är ett viktigt ämne för alla utvecklingsprojekt, och ett verktyg för detta bör alltid användas oavsett storlek på projektet. Det ger utvecklarna möjlighet att samtidigt kunna arbeta med samma källkod utan att lägga ner tid på att sammanställa sina enskilda ändringar. Revisionshantering gör det också möjligt att gå tillbaka i tiden och till exempel få ett utdrag på källkoden precis som den var vid en tidigare version.

Det versionshanteringssystem som använts i projektet heter Subversion (SVN) och är en konkurrent till det ledande Concurrent Version System (CVS) inom öppen källkod [5]. Subversion har många fördelar över CVS bland annat atomär incheckning (commit) istället för per fil. Det vill säga att om en fil inte kan checkas in går ingenting in. Kataloger och namnbyten har egna versioner, så det är möjligt att ta bort och ändra strukturen på källkoden, men samtidigt ha kvar ändringarna i tidigare revisioner. Subversion gör det möjligt att se incheckade dokument från ett projekt direkt på webben, och alla utvecklare meddelas via e-post när någon ändring gjorts i projektet.

Till subversion finns ett klientverktyg som är integrerat i Windows standardmenyer. Verktyget heter Tortoise SVN, och gör det mycket lätt för flera utvecklare att arbeta med projektet på olika datorer. Genom Tortoise kan man bland annat checka ut dokument, uppdatera material för att få de senaste ändringarna och checka in sina dokument.

4.2 DWS Agent

Ett reducerat klassdiagram för DWS Agent illustreras i figur 4-1. Ett fullständigt klassdiagram och dess funktioner och datamedlemmar finns i bilaga C.



Figur 4-1: DWS Agent, reducerat klassdiagram

4.2.1 Klassen Observer

Klassen Observer har en central roll i applikationen och därmed är det mycket viktigt att optimera dess prestanda. Den måste vara robust och stabil och för att garantera applikationens funktionalitet, men samtidigt inte använda sig av för mycket resurser av systemet.

Klassen har hand om alla anrop från användargränssnittet, då användargränssnittet representerar både vyn och kontrollern. Observer representerar således modellen i designmönstret MVC. Vid en eventuell ändring av klassmedlemmar uppdaterar klassen det relaterade gränssnittet. Den uppdaterar även klassen DataManager som är till för att omstart av agent kan genomföras, utan att mista väsentlig data som till exempel agentens namn och vilka processer som övervakas. Några viktiga och väsentliga metoder är beskrivna i kommande underkapitel.

4.2.1.1 Metoderna Scan

Vid start av applikation startar klassen en egen tråd som övervakar alla processer som användaren vill övervaka, samt den process som övervakar DWS Agent, DWSWatchdog. Tråden stoppas endast då applikationen avslutas. Det finns två viktiga överlagrade metoder som tråden använder sig av. Metoderna ser ut som nedan:

```
private void Scan() {
  while(true) {
    lock(processHandlers) {
      foreach(ProcessHandler ph in this.processHandlers.Values) {
        this.Scan(ph);
      }
      this.Scan(this.watchdog);
    }
    Thread.Sleep(10000);
  }
}

private void Scan(ProcessHandler ph) {
  ph.Refresh();
  if(ph.IsResponding) {
    ph.Responding();
  } else {
    ph.NotResponding();
  }
  if(ph.HasExceededMemoryLimit) {
    ph.ExceededMemoryLimit();
  }
}
```

Metoderna meddelar om den övervakade processen inte svarar eller om den har överstigit den maximala tillåtna nivån av minnesanvändning. Om någon av dessa händelser inträffar meddelas detta till den specifika ProcessHandler-klassen som representerar processen. Den eventuella åtgärden beslutas dock inte här.

Orsaken till att ha en pollande tråd som frågar alla processer om dess status är att ramverket .NET inte har ett event som meddelar om en process inte längre svarar. Ett pollande system är således den enda lösningen på problemet. Lägg märke till att ingen koll görs om processen har avslutats. Anledningen till att ingen sådan koll görs är att .NET stödjer denna funktionalitet. Tråden exekverar endast var 10:e sekund. Den skulle kunna exekvera oftare, men var 10:e sekund räcker för att täcka uppdragsgivarens krav och behov. Tiden kan minskas, men skulle endast resultera onödiga systemresurser.

4.2.1.2 Metoden OnTrap

Metoden OnTrap används för att alla övervakade processer skall kunna meddela Observer klassen om dess driftstörningar. Metoden tar emot information som till exempel namnet på processen, driftstörningens orsak, åtgärd och om åtgärden lyckats. Metoden skapar ett trap-paket med motsvarande information. Här skapas även en tidstämpel för driftstörningen. Paketet vidarebefordras till NetworkManager för att sedan sändas vidare. Metoden ser ut som nedan.

```
private void OnTrap(int pid, string name,
                    Cause cause, Measure measure, bool success) {
  Packet p = new Packet(PacketType.Trap);
  p.Header["Name"] = this.agentName;
  p.Header["Location"] = this.agentLocation;
  p.Header["Description"] = this.agentDescription;
  p.Header["Timestamp"] = DateTime.Now.ToString();
  p.Data["Pid"] = pid.ToString();
  p.Data["Process"] = name.ToString();
  p.Data["Cause"] = cause.ToString();
  p.Data["Measure"] = measure.ToString();
  p.Data["Result"] = success.ToString();
  this.netman.SendTrapMessage(p);
}
```

4.2.2 Klassen ProcessHandler

ProcessHandler klassen representerar en observerad användarprocess. För att kunna kontrollera en process används komponenten Process som finns i ramverket .NET. För en detaljerad komponentbeskrivning, se kapitel 4.2.2.4. ProcessHandler svarar på frågor från

Observer-klassen angående processen som den representerar. Vid en eventuell omstart eller annan händelse kan Observer-klassen meddelas, om användaren valt detta. Vid eventuella ändringar av viktiga datamedlemmar, som till exempel ändring av processens identifieringsnummer, meddelas detta till klassen Observer så att användargränssnitten kan uppdateras. ProcessHandler innehåller även den timer som startas om processen inte svarar. Alla åtgärder för en driftstörning beslutas i denna klass.

4.2.2.1 Metoden Start

Metoden Start används vid flera olika tillfällen. Den används för att starta om en process som använder för mycket minne, eller en process som inte längre svarar. Samma metod används vid start av en process som kraschat. För att starta en process krävs det att man skapar en instans av process-klassen, samt initierar den med startargument och sökväg, se implementation nedan.

```
private void Start() {
  if(!this.process.HasExited) {
    this.process.Exited -= new EventHandler(this.process_Exited);
    this.process.Kill();
  }
  this.process = new Process(); //Ny Instans
  this.process.StartInfo.FileName = this.name; //Namn initiering
  this.process.StartInfo.Arguments = this.sArg; //Start argument
  this.process.Start(); //Startar processen
  this.pid = this.process.Id;
  this.process.Exited += new EventHandler(this.process_Exited);
  this.process.EnableRaisingEvents = true;
  this.OnPidChanged(this.pid);
  this.OnDataChanged();
}
```

4.2.2.2 Metoden OnTrap

Metoden OnTrap anropas alltid då en driftstörning inträffat. Metoden tar emot vilken orsak händelsen gäller, vilken åtgärd som vidtagits samt om åtgärden var lyckad eller inte. Om användaren ställt in att meddelandetjänsten skall meddelas triggas ett event. Eventet fångas upp av klassen Observer som sedan vidarebefordrar informationen.

```
private void OnTrap(Cause c, Measure m, Result r) {
   if(this.notify == Notify.Always ||
      (this.notify == Notify.OnFailure && r == Result.Failure)) {
      if(this.TrapItem != null) {
         this.TrapItem(this.pid, this.name, cause, measure, result);
      }
   }
}
```

4.2.2.3 Metoden OnLogg

Även metoden OnLogg anropas vid övervakade processers driftstörningar. Den tar emot samma argument som OnTrap. Istället för att trigga ett event, skriver den ner händelsen direkt till klassen LogManager.

```
private void OnLog(Cause cause, Measure measure, Result result) {
   LogManager.Instance.AddLogEntry(this.name, cause, measure, result);
}
```

4.2.2.4 Komponenten Process

Komponenten Process hittas i ramverket .NET under System.Diagnostics. Den är ämnad för att få tillgång till en process som exekveras på en dator. Komponenten är ett bra verktyg för att starta, stoppa, kontrollera och övervaka applikationer. Med hjälp av den kan en lista fås av alla processer som exekverar på datorn. Efter att en komponent initierats mot en exekverande enskild process, kan den användas för att få detaljerad information om processen. Exempel på information som kan fås ur komponenten är:

- Mängden trådar processen innehåller
- Total minnesanvändning
- Identifikationsnummer
- Start tid
- Om den har avslutats
- Om den svarar

Efter att Process-komponenten initierats används cacheminne för att spara undan information. Komponenten garanterar inte att informationen uppdateras. Med hjälp av Refresh-metoden uppdateras all information.

4.2.3 Klassen DataManager

Klassen DataManager följer designmönstret Singleton, se kapitel 4.2.3.1 för implementation. Den har till uppgift att spara undan all nödvändig information för DWS Agent. Detta för att omstart skall kunna ske utan att viktig information försvinner. Informationen sparas undan formaterat i XML. DataManager använder sig av DataSet som är en komponent i .NET. DataSet innehåller i sin tur en samling av klassen DataTable. DWS Agent använder sig av två tabeller. En tabell som innehåller agentens allmänna konfigurationsinställningar som till exempel agentens namn, beskrivning, adress till meddelande servicen etc. Den andra tabellen innehåller information angående alla övervakade processer. DataSet har färdiga metoder för att spara undan och läsa in XML dokument.

```
this.dataset.WriteXml("Data.xml");
this.dataset.ReadXml("Data.xml");
```

För att läsa en XML-fil krävs även ett XML-schema eller DTD-fil. I denna fil definieras strukturen på XML-dokumentet. DWS Agent använder sig av ett XML-schema som också är en inbäddad resurs-fil. För att läsa in resurs-filen krävs kodraderna nedan.

```
Assembly executingAssembly = Assembly.GetExecutingAssembly();
string schema = String.Format("{0}.{1}",
          executingAssembly.GetName().Name, "Resources.datastore.xml");
this.dataset.ReadXmlSchema(
          executingAssembly.GetManifestResourceStream(schema));
```

Fördelen med att ha en inbäddad resurs-fil är att filen kompileras in i binärfilen. Den slipper således följa med som en separat fil i installationen.

4.2.3.1 Kodexempel trådsäker singleton implementation

För att upprätthålla singletonkonceptet görs en dubbelkoll samt en låsning, Double-Check Locking [7]. Denna implementation är trådsäker, det vill säga att implementationen kan garantera att två separata trådar skapar en instans samtidigt. Volatile används för att garanterar att tilldelningen till instans-variabeln fullbordas innan variabeln blir tillgänglig.

Implementationen låser även en Sync-variabel istället för att låsa hela klassen DataManager, för att undvika att dödläge uppstår.

```
public class DataManager {
  static volatile DataManager instance = null;
  static object sync = new object();

  public static DataManager Instance {
    get {
      if(instance == null) {
        lock(sync) {
          if(instance == null) {
            instance = new DataManager();
          }
        }
      }
      return instance;
    }
  }
}
```

4.2.4 Klassen LogManager

Klassen LogManager följer designmönstret singleton. Den har till uppgift att skriva ner och sparar undan alla driftstörningar angående de övervakade processerna. Ramverket .NET har en komponent, EventLog, som är knutet mot Windows egna logghanteringssystem. Logghanteringssystemet har tre färdigdefinierade tabeller för att skriva ner händelser i, applikation, säkerhet och system. DWS Agent skapar dock en egen tabell för att inte blanda sig i operativsystemets egen logg. Tabellen skapas enligt nedan.

```
EventLog.CreateEventSource("DWS Agent", "DWSAgent");
```

För att skriva ner en händelse till loggen måste en initiering ske till rätt tabell. Efter det är det lätt att skriva ner, hämta och radera poster.

```
this.eventLog = new EventLog();
this.eventLog.Log = "DWSAgent"; //Initiering till tabell
this.eventLog.Source = "DWS Agent";

//Nedskivning av post
this.eventLog.WriteEntry(
  string.Format("Process: {0}\nCause: {1}\nMeasure: {2}\nResult: {3}",
                name, cause, measure, result), type, eventId);

//Hämtning av alla poster
EventLogEntryCollection entries = this.eventLog.Entries;

//Radering av alla poster
this.eventLog.Clear();
```

4.2.5 Klassen NetworkManager

Klassen NetworkManager ser till att inställd information vidarebefordras till DWS Message service. Klassen skall i framtiden även svara på begäran från DWS Manager. Den använder sig av komponenten Network Service för att kommunicera med nätverket.

Alla inkommande paket från klassen Observer märks med ett sekvensnummer och buffras i en Hashtabell, med sekvensnumret som nyckel. Hashtabellen skrivs ner till fil för att garantera att inga paket går förlorade även om applikationen avslutas. Den är även till för att buffra paketen vid kommunikationsproblem mellan DWS Agent och Message Service. Sekvensnumret är till för att få en kvittens på att ett enskilt paket verkligen kommit fram. Klassen ansvarar även för säkerställningen av informationsflödet mellan enheterna, kryptering. Paketen serialiseras/deserialiseras och krypteras med metoderna:

```
private byte[] SerializeMessage(Packet p)
private Packet DeserializeMessage(byte[] data)
```

Läs mer om krypteringsimplementationen i kapitel 4.8.

4.2.5.1 Metoden SendTrapMessage

Metoden tar emot paket från klassen Observer. Den märker paketet med ett sekvensnummer, lägger in paketet i hashtabellen och sparar ner till fil. En lock görs för att förhindra att olika trådar ändrar i hashtabellen samtidigt. Den anropar metoden Connect() som ansluter till inställd Message Service.

```
public void SendTrapMessage(Packet p) {
 lock(this.trapTable) {
     p.SequenceNumber = this.NextSequenceNumber;
     this.trapTable[p.SequenceNumber] = p;
     this.WriteTrap();
  }
  this.Connect();
}
```

4.2.6 Klassen MainView

Klassen MainView ärver från Windows.Forms och är DWS Agentens användargränssnitt, vy i MVC. Klassen består av ett navigeringsträd, meny, knapprad samt ett flertal användarkontroller. Menyraden används för att kunna minimera, stänga av applikationen samt

lägga till en användarprocess. Knapparna används för att kunna navigera tillbaka till en användarkontroll, likt Internet Explorer, samt även en knapp som gör det möjligt att även härifrån lägga till en användarprocess. Trädet används för att navigera till en ny användarkontroll, samt lägga till och ta bort processer. Det består av en rot-nod som endast är till för att gruppera de övriga noderna. Under rot-noden finns tre barn-noder:

- Settings
- Event Log
- Processes

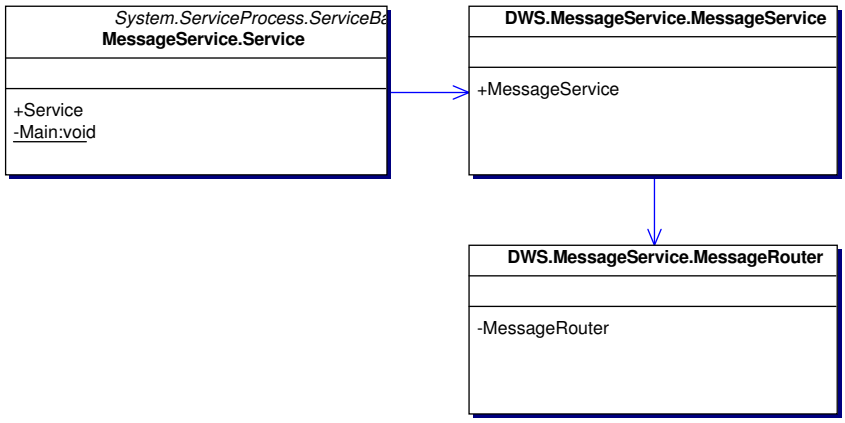
Via noden ”Settings” kan användaren ställa in all nödvändig information om DWS Agent, till exempel vilken port den ska lyssna på för inkommande anrop. Genom noden ”Event Log” kan en logg utforskas där det står i klartext alla händelser som inträffat. Process-noden visar alla övervakade processer. Under denna nod finns de processer som är övervakade. Där kan inställningar göras angående vilka åtgärder DWS Agent ska utföra, beroende på händelse. För en utförligare beskrivning, se kap 5.1.

Användarkontrollerna implementerar i sin tur från interfacen ISaveable och IResetable. Interface:t ISaveable används för att alla ändrade inställningar skall kunna sparas undan och IResetable för att alla användarkontroller ska uppdateras med aktuella inställningar.

4.3 DWS Message Service

Message Service är den meddelandetjänst i DWS som agenterna ansluter till för att leverera information till kontaktpersoner. Meddelandetjänsten är implementerad som en Windows Service och innebär att den startas tillsammans med operativsystemet utan att någon specifik användare måste vara inloggad. Vidare används en Microsoft Access-databas med konfigurerbara regler för att avgöra till vilka kontaktpersoner inkommande meddelande ska vidarebefordras.

Ett reducerat klassdiagram för DWS Message Service illustreras i figur 4-2. Ett fullständigt klassdiagram och dess funktioner och datamedlemmar finns i bilaga D.



Figur 4-2: DWS Message Service, reducerat klassdiagram

4.3.1 Klassen MessageService

Den här klassen, med samma namn som komponenten, har hand om den grundläggande funktionaliteten för att ta emot och se till att meddelanden blir omhändertagna av MessageRouter-klassen. Den använder sig av komponenten Network Service för att kommunicera med nätverket. Både MessageRouter och Network Service kommer att beskrivas i kommande kapitel.

4.3.1.1 Metoden ClientConnected

Den här metoden är kopplad till ett event i Network Service som körs varje gång en ny klient har anslutit till meddelandetjänsten. Allt metoden gör att efter den har anslutit är att börja lyssna efter data.

```
void ClientConnected(ConnectionState client) {
  this.net.Receive(client);
}
```

4.3.1.2 Metoden PacketRecieved

När ett paket är mottaget från en klient levereras det via ett event från Network Service och tas om hand om i metoden PacketRecieved. Vad som görs är att paketet dekrypteras och sedan deserialiseras. Med deserialisering menas att paketobjektet återskapas från en byte-array. Oavsett om hanteringen av ett paket lyckas kopplas anslutningen mot klienten ner.

```
void PacketReceived(ConnectionState client, byte[] data) {
  try {
    using(Stream s = new MemoryStream(data)) {
      TripleDESCryptoServiceProvider tdes =
          new TripleDESCryptoServiceProvider();
      using(CryptoStream csr =
          new CryptoStream(s, tdes.CreateDecryptor(MessageService.Key,
          MessageService.IV), CryptoStreamMode.Read)) {
        IFormatter f = new BinaryFormatter();
        MessageRouter.Deliver((Packet)f.Deserialize(csr));
      }
    }
  } finally {
    this.net.Disconnect(client);
  }
}
```

Efter paketet har dekrypterats och deserialiserats skickas det vidare till metoden Deliver i MessageRouter-klassen. Vad som sedan händer med paketet beskrivs i nästa sektion.

4.3.2 Klassen MessageRouter

MessageRouter är en helt statisk klass med metoder för att leverera meddelanden till rätt destination beroende på en uppsättning regler. Anledningen till att klassen är statisk är att den inte har något tillstånd. Varje regel har ett antal kontaktpersoner som ansvarar för meddelanden som matchar den regeln. För varje regel som matchar vidarebefordras informationen till rätt kontaktperson, dock max en gång även om kontaktpersonen skulle finnas med i flera matchande regler.

4.3.2.1 Metoden Deliver

Deliver metoden ansvarar för att leverera ett meddelande till ett antal e-postadresser beroende på vilka regler som matchar paketet. Först anropas metoden SelectRules som returnerar de regler som matchar ett paket varpå metoden SelectEmailAddresses anropas för att hämta de e-postadresser som är associerade med de givna reglerna. Hur ett paket matchas beskrivs i kapitel 4.3.2.4. Koden för Deliver ser ut som följer.

```
public static void Deliver(Packet p) {
  ArrayList emails =
       MessageRouter.SelectEmailAddresses(MessageRouter.SelectRules(p));
  MailMessage message = MessageRouter.FormatMessage(p);
  SmtpMail.SmtpServer = (string)DataService.GetObject(
DataService.CreateCommand("GetSmtpServer", CommandType.StoredProcedure));
  foreach(string email in emails) {
    message.To = email;
    SmtpMail.Send(message);
  }
}
```

4.3.2.2 Metoden SelectRules

SelectRules itererar över de regler som finns konfigurerade i databasen och avgör om alla villkor för en given regel stämmer med paketet p.

```
private static ArrayList SelectRules(Packet p) {
  ArrayList rules = new ArrayList();
  foreach(DataRow r in
     DataService.CreateTable(DataService.CreateCommand("GetRules",
       CommandType.StoredProcedure)).Rows) {
    bool applied = true;
    IDbCommand command = DataService.CreateCommand("GetRuleConditions",
                                          CommandType.StoredProcedure);
    command.Parameters.Add(
            DataService.CreateParameter("RuleId", r["Id"].ToString()));
    foreach(DataRow c in DataService.CreateTable(command).Rows) {
    if(!MessageRouter.Compare(Int32.Parse(c["ComparisonId"].ToString()),
     c["Value"].ToString(), p.Header[c["Type"].ToString()].ToString())) {
        applied = false;
      }
    }

    if(applied) {
      rules.Add(r["Id"].ToString());
    }
  }

  return rules;
}
```

4.3.2.3 Metoden SelectEmailAddresses

För att hämta e-postadresserna till de kontaktpersoner som hör till en uppsättning regler används metoden SelectEmailAddresses i MessageRouter. Att notera är att en e-postadress endast kan returneras en gång.

```
private static ArrayList SelectEmailAddresses(ArrayList rules) {
  ArrayList emails = new ArrayList();
  foreach(string rule in rules) {
    IDbCommand command = DataService.CreateCommand("GetRuleEmails",
                                          CommandType.StoredProcedure);
    command.Parameters.Add(DataService.CreateParameter("RuleId", rule));
    DataTable ruleemails = DataService.CreateTable(command);
    foreach(DataRow row in ruleemails.Rows) {
      string email = row["Email"].ToString();
      if(!emails.Contains(email)) {
        emails.Add(email);
      }
    }
  }

  return emails;
}
```

4.3.2.4 Meddelande Matchning

Hur ett meddelande matchas mot en viss regel bestäms av ett antal villkor som kan konfigureras av användaren av DWS. Villkoren kan vara plats av den agent som skickar meddelandet, namn på agenten eller en beskrivning som användaren fyllt i. Ett exempel på en regel kan vara att alla meddelanden som skickas från en agent som befinner sig i Karlstad och har namn som börjar på KAU ska vidarebefordras till en given e-postadress.

För att matcha paket med olika noggrannhet används metoden Compare i MessageRouter-klassen. Den kan till exempel matcha en sträng beroende på om den börjar med, slutar med, innehåller eller är exakt lika med en annan sträng.

```
private static bool Compare(int type, string v, string p) {
  switch(type) {
    case 1: return p.Equals(v);
    case 2: return p.IndexOf(v) > -1;
    case 3: return p.StartsWith(v);
    case 4: return p.EndsWith(v);
    default: return false;
  }
}
```

4.3.3 Windows Service

Windows services är ett sätt att skapa applikationer som inte är knutna till någon speciell användare utan som exekverar i bakgrunden av systemet. En Windows service kan startas automatiskt tillsammans med operativsystemet, kan även pausas, startas om och avslutas via administratörsverktygen i Windows men har inget eget användargränssnitt. Den här typen av applikationer lämpar sig särskilt bra till servrar och andra program som är ämnade att köras konstant utan att påverkas av andra användare i systemet.

För att få köra en Windows service måste den installeras på systemet vilket kan göras med hjälp av standardverktyget InstallUtil.exe. Det medför att en Windows service inte kan debuggas eller köras från Visual Studio bara genom att trycka F5 eller F11 som för andra projekt. Istället måste services installeras och startas manuellt, varpå det sedan är möjligt att ansluta en debugger.

En klass som implementeras som en Windows service måste ärva från System.ServiceProcess.ServiceBase och måste därifrån överlagra ett antal virtuella metoder. De metoder som ska överlagras är:

```
protected override void OnStart(string[] args);
protected override void OnStop();
```

Förutom metoderna ovan måste även i main anropa metoden ServiceBase.Run och ange vilken service som ska köras.

4.4 Data Service

För kommunikation mot databas används tjänsten Data Service som ger ett enkelt och säkert gränssnitt där onödig och komplex funktionalitet abstraheras för användaren. Komponenten är som en fasad mot databasen och följer designmönstret facade. Den funktionalitet som komponenten ansvarar för är vanligt förekommande i dataintensiva applikationer där koden för datahantering upprepas ofta. Det är därför viktigt att modulera dess operationer till en central tjänst där koden kan återanvändas från alla delar i programmet. Användaren behöver i många fall inte heller tänka på att skriva komplicerade felhanteringsblock för att garantera att alla databasanslutningar stängs på ett korrekt sätt.

4.4.1 Klassen DataService

DataService är en klass med endast statiska medlemmar och vars konstruktor är deklarerad som privat. Det går alltså inte att skapa några objekt av typen DataService. Anledningen till att DataService inte är någon singleton är att den inte har något tillstånd där skapande måste kontrolleras. En tumregel brukar vara att skapa en singleton klassen kommer ha ett tillstånd, för övriga klasser, med metoder som till exempel Math.Round, är det mer naturligt att deklarera dem som statiska.

4.4.1.1 Egenskapen ConnectionString

En anslutningssträng används för att ange hur och vilken databas som ska anslutas. Strängarna ser olika ut beroende på vilken databas man vill ansluta till och vilken mellanvara som används. För att ansluta till Microsoft Access via OLEDB ser anslutningssträngen ut så här:

```
Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Database.mdb;
User Id=admin; Password=*****;
```

Innan metoden GetConnection kan användas måste användaren sätta en giltig anslutningssträng. Det görs genom följande egenskap:

```
public static string ConnectionString { get; set; }
```

Värdet kan ändras i efterhand och påverkar då endast efterkommande anrop till Data Service.

4.4.1.2 Metoden GetConnection

GetConnection används för att skapa en anslutning mot den databas angiven i anslutningssträngen. Det är inte tänkt att användaren av Data Service ska använda GetConnection direkt, utan den anropas implicit av övriga metoder när det behövs. Den är fortfarande deklarerad som publik eftersom avancerade användare kan tänkas vilja förbigå de metoder som finns i Data Service och arbeta direkt mot anslutningen. I koden för GetConnection ser vi hur metoden returnerar typen IDbConnection, som är ett interface i .NET för att representera en anslutning.

```
public static IDbConnection GetConnection() {
   OleDbConnection connection =
        new OleDbConnection(DataService.ConnectionString);
   connection.Open();
   return connection;
}
```

Till skillnad mot till exempel Java, har inte .NET någon generell databasklass utan det finns olika klasser för olika typer av databaser. I exemplet ovan används OleDbConnection eftersom OLEDB är den mellanvara som används i DWS. Andra anslutningsklasser som finns är till exempel SqlConnection för MS SQL Server. Oavsett vilken klass man använder har de däremot likadana gränssnitt eftersom alla databasrelaterade klasser i .NET implementerar gemensamma interface. För att gränssnittet i Data Service ska bli generellt oavsett vilken databas som används följer alla parametrar och returtyper i komponenten de generella interfacen som implementeras av respektive klass.

4.4.1.3 Metoden FreeConnection

När en databasanslutning har skapats måste den stängas på ett korrekt sätt. I Data Service sköts detta automatiskt utan att användaren behöver tänka på det, men komponenten tillhandahåller även metoden FreeConnection för att explicit kunna stänga en anslutning. Det kan vara nödvändigt om en användare själv anropar GetConnection utan att använda de hjälpfunktioner som finns i Data Service.

FreeConnection har följande signatur:

```
public static void FreeConnection(IDbConnection connection);
```

4.4.1.4 Metoden GetObject

För att hämta ett enstaka värde från databasen används metoden GetObject. Den returnerar värdet från den första kolumnen i den första raden i resultatet av en SQL-fråga. Metoden är definierad nedan.

```
public static object GetObject(string statement,
                               CommandFormatter formatter) {
  using(OleDbConnection connection = (OleDbConnection)GetConnection()) {
    OleDbCommand command = new OleDbCommand(statement, connection);
    return FormatCommand(formatter, command).ExecuteScalar();
  }
}
```

GetObject tar två parametrar, statement och formatter. Statement är den SQL-fråga som användaren vill ställa till databasen. Formatter är ett delegat till en extern funktion som kan ta hand om mer detaljerad formatering av SQL-frågan. Det är användbart när man vill anropa lagrade procedurer i en databas och behöver ange in och ut parametrar. Delegaten tar emot ett

kommando och returnerar en modifierad version. Deklarationen av CommandFormatter ser ut så här:

```
public delegate IDbCommand CommandFormatter(IDbCommand command);
```

Nästa steg i GetObject är att ett nytt kommando skapas utifrån den angivna SQL-frågan och en databasanslutning som hämtas från metoden GetConnection. Eftersom anslutningen hämtas i ett speciellt block med nyckelordet using kommer frånkoppling av anslutningen ske automatiskt när blocket lämnas. Metoden returnerar genom att kommandot formateras med den medskickade delegaten varpå ExecuteScalar anropas.

4.4.1.5 Metoden LendReader

Metoden LendReader är till för att köra egendefinierad kod som arbetar på en öppen databasanslutning. För att göra det möjligt utan att lämna ut en anslutning till användare av Data Service tar metoden hjälp av ett designmönster, kommando, beskrivet i kapitel 3.6.3. Det innebär att användarens kod körs efter anslutningen är skapad, men innan metoden returnerar.

```
public static object LendReader(string statement,
                                CommandFormatter formatter,
                                BorrowObject borrower) {
  using(OleDbConnection connection = (OleDbConnection)GetConnection()) {
    OleDbCommand command = new OleDbCommand(statement, connection);
    return borrower(FormatCommand(formatter, command).ExecuteReader());
  }
}
```

Parametrarna är de samma som för GetObject förutom den sista parametern borrower som anger vilken funktion som ska utföra arbetet på det resultat SQL-frågan returnerar. Borrower är ett delegat av typen BorrowReader som har följande utseende.

```
public delegate object BorrowReader(IDataReader reader);
```

Metoden ser ungefär ut som GetObject, den skapar ett kommando från SQL-frågan i statement och formatterar kommandot med den angivna formateraren. Istället för att anropa ExecuteScalar som returnerar värdet i den raden i den första kolumnen, anropas ExecuteReader som lämnar en läsare till den öppna databasen. En läsare kan iterera emellan raderna i resultatet. Till slut exekveras användaren kod genom att läsaren skickas till borrower vars returvärde returneras av LendReader.

4.4.1.6 Metoden CreateTable

För att returnera ett resultat när databasen är frånkopplad kan man använda ett så kallat DataTable för att spara data i. Metoden tar en IDbCommand som parameter, och istället för att anropa någon av Execute-metoderna som i till exempel GetObject i kapitel 4.4.1.4, skapas en DataAdapter vars uppgift är att fylla ett DataTable med resultet från command-objektet.

```
  public static DataTable CreateTable(IDbCommand command) {
    using(OleDbConnection connection = (OleDbConnection)GetConnection()) {
      DataTable table = new DataTable();
      command.Connection = connection;
      OleDbDataAdapter adapter =
                            new OleDbDataAdapter((OleDbCommand)command);
      adapter.Fill(table);
      return table;
    }
  }
```

4.4.1.7 Metoden ExecuteCommand

Det finns SQL-kommandon som ändrar databasen, så kallade non-queries. Det kan till exempel vara för att sätta in, uppdatera eller ändra data. För det ändamålet finns metoden ExecuteCommand som tar ett uttryck och en formaterare som parametrar och utför kommandot. Metoden har följande utseende.

```
  public static void ExecuteCommand(string statement, CommandFormatter
formatter) {
    using(OleDbConnection connection = (OleDbConnection)GetConnection()) {
      OleDbCommand command = new OleDbCommand(statement, connection);
      FormatCommand(formatter, command).ExecuteNonQuery();
    }
  }
```

4.5 Network Service

Network Service är en tjänst som tillhandahåller ett gränssnitt för att kommunicera med andra datorer på ett nätverk. All funktionalitet som rör nätverket är abstraherat i den här komponenten efter designmönstret facade. Network Service tillåter också användaren att ge kommando, se kapitel 3.1 om designmönster, som körs för givna händelser. Till exempel när en ny klient har anslutit eller när ett paket har blivit mottaget. Network Service komponenten är uppdelad i tre klasser som beskrivs var för sig i kommande sektioner. Idéer har tagits från Richard Blum [8] gällande asynkron nätverksprogrammering.

4.5.1 Klassen NetworkService

NetworkService är huvudklassen i komponenten och alla anrop mot nätverket går genom den. Den har två viktiga delar. En lyssnare som är en socket dedikerad till att enbart lyssna efter nya inkommande anslutningar. När en anslutning accepterats skapas en ny socket och ConnectionState-objekt som läggs undan i en lista över anslutna klienter. En liknande lista finns över alla ConnectionState-objekt för utåtgående anslutningar. Värt att notera är att all kommunikation i Network Service sker över TCP.

4.5.1.1 Metoden Listen

För att initiera lyssnaren anropas metoden Listen som tar följande parametrar.

```
public void Listen(EndPoint ep);
```

Metoden tar en EndPoint som argument, vilket i .NET representerar en nätverksadress. Argumentet anger på vilken adress som socketen ska lyssna. En viktig sak att notera är att metoden är asynkron. Det innebär att ett anrop till Listen inte blockerar programflödet och väntar på en anslutande klient. Metoden kommer att starta upp en ny tråd som lyssnar på anslutningar och sedan returnerar direkt. När en klient ansluter kommer den asynkrona metoden att exekvera en så kallad callback-funktion som tar hand om anslutningen. Det sker dock i bakgrunden vilket gör att användaren av Network Service inte behöver tänka på det. Alla metoder i gränssnittet för Network Service är asynkrona. Listen metoden har två events som användaren kan använda för att exekvera egen kod då en klient ansluter.

```
public event ConnectionHandler AfterAccept;
public event ConnectionHandler FailedAccept;
```

4.5.1.2 Metoden Connect

Förutom att tillåta klienter att ansluta till en Network Service kan komponenten också ansluta till andra noder på nätverket. Det görs genom metoden Connect.

```
public void Connect(EndPoint ep);
```

Metoden tar precis som Listen emot en EndPoint men här anger den adressen dit användaren vill ansluta. Som nämnt är även Connect en asynkron metod som returnerar direkt efter den

anropats. För att användaren ska veta om det gick bra att ansluta eller inte finns det två events för att hantera just detta.

```
public event ConnectionHandler AfterConnect;
public event ConnectionHandler FailedConnect;
```

AfterConnect och FailedConnect kommer anropas när en anslutning upprättats respektive misslyckats. Här har användaren möjlighet att påverka vad som görs i varje fall. Sättet att få kod att exekvera mitt i en händelse in ett annat objekt kallas ett kommando. För mer om designmönster se kapitel 3.1.

4.5.1.3 Metoderna Receive och Send

Oavsett om Network Service har anslutit till en annan nod, eller om en klient anslutit till lyssnaren finns det två metoder för att kommunicera med den anslutna socketen.

```
public void Receive(ConnectionState state);
public void Send(ConnectionState state, byte[] data);
```

Den första metoden säger åt socketen för det medföljande ConnectionState-objektet att lyssna efter inkommande data, den andra metoden skickar data från sitt argument till medföljande ConnectionState-objekt. De två metoderna genererar varsitt event när data är mottaget respektive skickat.

```
public event PacketHandler PacketReceived;
public event PacketHandler PacketSent;
```

Det ger användaren möjlighet att på ett enkelt sätt ta hand om inkommande trafik, och avgöra exakt vad som har skickats.

4.5.1.4 Metoden Disconnect

När användaren vill koppla ifrån en anslutning, oavsett om anslutningen är inkommande eller utgående, används metoden Disconnect.

```
public void Disconnect(ConnectionState state);
```

Metoden kommer ta bort state-objektet från listan över uppkopplade anslutningar var på den kopplar ner socketen som finns angiven i state-objektet. Till sist körs ett event där användaren har möjlighet att exekvera egen kod som tar hand om från kopplingen.

Det event som går igång heter AfterDisconnect och har följande signatur.

```
public event ConnectionHandler AfterDisconnect;
```

4.5.1.5 Implementationstekniker

Eftersom kommunikationen sker över TCP, det vill säga förbindelseorienterad kommunikation, skickas data i strömmar mellan sändare och mottagare. Till skillnad mot UDP och meddelandeorienterad kommunikation där ett paket skickas klart och tydligt i varje datagram är det inte lika lätt att urskilja paketgränser med TCP. För att mottagaren ska veta var i strömmen ett paket börjar och slutar finns det tre olika tekniker.

Det enklaste sättet, men samtidigt det mest begränsade, är att alltid ha en fast paketstorlek. Med en given storlek vet mottagaren precis hur många byte den måste läsa från strömmen. Fördelen med fast paketstorlek är att det är snabbt och ingen extra information måste läggas till. I de flesta fall däremot är inte alla paket lika stora. Storleken måste då vara satt tillräckligt hög för att klara de största paketen, samtidigt som de mindre paketen måste fyllas upp.

Ett alternativ till att låsa paketstorleken är att ha någon form av avgränsare mellan paket i strömmen. En avgränsare kan vara en unik sekvens av byte som mottagaren lyssnar efter för att veta när ett paket slutar och ett annat paket börjar. Även om den här tekniken tillåter variabel paketlängd måste man vara noga med att avgränsaren inte ser likadan ut som någon del av det riktiga paketet. Det skulle få mottagaren att dela paketen innan alla data lästs och förmodligen göra informationen oläsbar.

Den teknik som används i Network Service tar hand om problemen från båda tidigare alternativ. Sändaren måste lägga till extra information om paketlängden innan han skickar varje paket. Mottagaren läser sedan de första byten som innehåller längden av efterkommande paket, och vet sedan hur många byte som måste läsas.

4.5.2 Klassen ConnectionState

ConnectionState är en klass vars instanser representerar en anslutning i NetworkService. Den har hand om det underliggande socket-objektet för varje anslutning, och innehåller också buffertar för inkommande och utgående data. Den mest intressanta egenskapen som användaren av Network Service kan behöva använda är:

```
public Socket Socket {
   get {
      return this.socket;
   }
}
```

Förutom socket och databuffertar håller ConnectionState även reda på om det är ett pakethuvud med information om hur många byte kommande paket är, eller om det är det riktiga paketet som tas emot. Anledningen till den lösningen beskrevs i kapitel 4.6.1.5.

4.5.3 Klassen DataBuffer

Den här klassen representerar en buffert som lagrar data i form av en byte-array. Dess storlek kan modifieras, och klassen har information om hur mycket data i bufferten som till exempel har skickas till transportlagret. Användaren av Network Service har möjlighet att se aktuella buffertar för respektive anslutning, men det är oftast inte intressant då innehållet automatiskt följer med som argument till de event där datan berörs. Om bufferten vill användas kan den kommas åt genom egenskapen Buffer:

```
public byte[] Buffer { get; }
```

4.6 Paketstrukturer

Det finns tre olika paket som skickas mellan applikationerna.

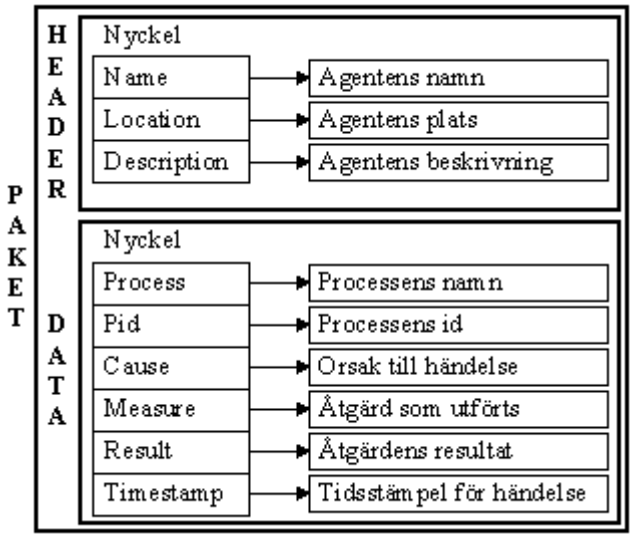
- Trap
- Request
- Response

Alla paket består av två delar, header och data. Beståndsdelarna får dock inte blandas ihop med transportlagrets, då paketstrukturens design ligger på applikationslagret. Innehållet i

header och data skiljer sig åt, beroende vilket paket som de tillhör. Delarna består av hashtabeller. Paketen skulle lika gärna kunna vara uppbyggda med XML, men valet av att använda hashtabeller grundar sig i att det underlättat implementationen. En detaljerad beskrivning av alla paket och dess paketstruktur finns beskrivet i kapitel 4.7.1-4.7.3.

4.6.1 Trap

I ett trap-paket ingår alltid information om vilken agent som har skickat paketet. Detta för att meddelandetjänsten skall kunna vidarebefordra rätt information till ansvarig person. Ett trap-meddelande och dess struktur illustreras i figur 4-3.

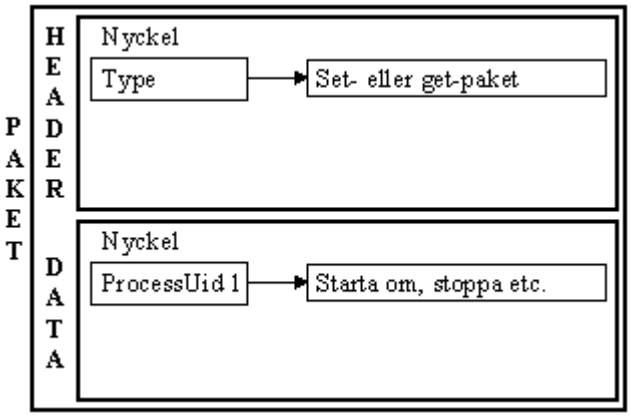


Figur 4-3: Trap-paket

Paketet består av all nödvändig information för att användaren i slutändan skall kunna tolka driftstörningen som uppstått och om felet kvarstår eller inte. I paketet ingår vilket fel som orsakat driftstörningen, åtgärden, samt om åtgärden var lyckad eller inte. Det finns även en tidstämpel, så att användaren kan få reda på när det inträffat.

4.6.2 Request

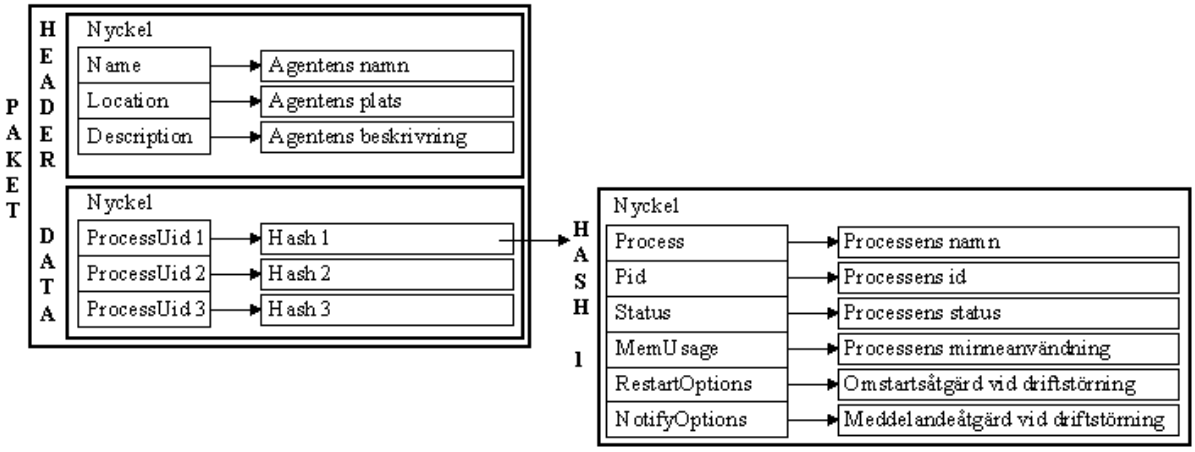
Ett request paket kan vara av två typer, *get*-paket och *set*-paket. Ett *get*-paket innehåller en förfrågning av DWS Agentens övervakade processers tillstånd. *Set*-paketet består av en förfrågning av åtgärd, som till exempel manuell omstart av en övervakad process. I figur 4-4 illustreras hur ett *set*-paket är strukturerat.



Figur 4-4: Request/set paket

4.6.3 Response

Ett response paket innehåller en likadan header som i ett trap paket. Data-delen skiljer sig dock åt jämfört med ett trap-paket. Data innehåller information om alla övervakade processer och dess tillstånd, identifikationsnummer, namn, status, minnesanvändning etc. Ett response paket och dess struktur finns illustrerat i figur 4-5.



Figur 4-5: Response-paket

4.7 Trådning

Trådar används på flera ställen i DWS och är nödvändigt för att enheterna (agent, manager och meddelandetjänst) ska kunna utföra flera saker samtidigt utan att låsa andra funktioner. De specifika trådarna som förekommer beskrivs i kapitel för respektive del. I följande sektioner kommer en generell beskrivning om trådhantering i .NET, samt hur man skriver trådsäker kod.

4.7.1 Hantering av trådar

För att initiera en ny tråd krävs ett objekt av typen Thread vars konstruktor tar en delegat som pekar ut var tråden ska börja exekvera.

```
Thread thread = new Thread(new ThreadStart(EntryPoint));
```

Delegaten ThreadStart förväntar sig en funktion utan parametrar som returnerar void. För att köra igång exekveringen av tråden anropas metoden Start i klassen Thread.

```
Thread.Start();
```

Vidare finns det en mängd metoder för att hantera en tråd, så som tillfälligt pausa exekveringen, låta tråden sova eller avbryta tråden helt och hållet. Exempel på dessa metoder är i samma ordning som beskriven ovan:

```
thread.Suspend();     //thread suspenderas
Thread.Sleep(1000);   //Aktuell tråd sover i 1000 ms
thread.Abort();       //thread termineras
```

4.7.2 Thread-safe

I program med flera trådar kan det lätt uppstå problem när de olika trådarna jobbar med samma data samtidigt. Med begreppet thread-safe menar man ett program vars trådar kan interagera utan oönskade konsekvenser. Det finns olika metoder för att trådar ska kunna kommunicera, samordna operationer och invänta varandra. Ett sätt är att använda semaforer, dock finns ett annat alternativ i C# som heter lock. Det är ett nyckelord som kan låsa kritiska delar i ett program så att endast en tråd får tillträde samtidigt. Ett exempel på hur lock används beskrivs nedan.

```
lock(this) {
   ... //Kritisk sektion
}
```

En parameter skickas med till lock vilken bestämmer att inga andra trådar som exekverar kan låsa samma objekt. I det här fallet är det en referens till det egna objektet, men kan vara vilket objekt som helst av typen object. Om en annan tråd skulle komma till uttrycket ovan och this redan är låst kommer tråden att vänta tills låset upphört och sedan själv skaffa sig ett lås och gå in i lock-blocket. När en tråd lämnar blocket släpps låset automatiskt [7].

Det finns en direkt översättning av lock i C# mot .NETs klassbibliotek. Klassen Monitor som ligger under System.Threading kan användas för att göra exakt samma sak. Det kompilatorn gör när den stöter på ett lock-uttryck är att översätta det till följande kod.

```
try {
  Monitor.Enter(this);
  ... //Kritisk sektion
} finally {
  Monitor.Exit(this);
}
```

Det är viktigt att notera try-finally-blocken i exemplet ovan. Genom att placera Monitor.Exit i ett finally-block garanteras att tråden släpper låset även om ett exception skulle inträffa i den kritiska sektionen.

4.8 Kryptering

För kryptering finns redan ett antal färdiga algoritmer som kan användas i .NET ramverk. Många av de vanligare krypteringsalgoritmerna är implementerade men DWS har tagit nytta av Triple-DES som är lätt att använda [8].

För att kryptera eller dekryptera information behövs dels en privat nyckel, samt en initieringsvektor. Vissa meddelanden, som till exempel e-post, har en fördefinierad header som alltid ser likadan ut och kan på så sätt identifieras även då de blivit krypterade. Initieringsvektorn används för att skydda sig mot det problemet genom att kasta om de första byten i ett meddelande.

För att de olika enheterna i DWS ska kunna läsa varandras meddelanden krävs att de använder samma nyckel. Nyckeln och initieringsvektorn är lagrade som statiska datamedlemmar i de ändpunkter där informationsutbytet tar plats.

4.8.1 Implementation av kryptering

För att kryptera en byte-array och spara resultatet i en MemoryStream s ser implementationen ut så här. Exemplet är inte komplett men visar hur krypteringen går till. Allt som skrivs till strömmen csw kommer krypteras och hamna i s.

```
using(MemoryStream s = new MemoryStream()) {
  TripleDESCryptoServiceProvider tdes =
     new TripleDESCryptoServiceProvider();
  using(CryptoStream csw =
     new CryptoStream(s, tdes.CreateEncryptor(NetworkManager.Key,
                         NetworkManager.IV), CryptoStreamMode.Write)) {
    csw.Write( ... );
    csw.FlushFinalBlock();
  }
}
```

4.8.2 Implementation av dekryptering

Följande kodexempel visar hur man dekrypterar information från en MemoryStream. Allt som läses från strömmen csr kommer dekrypteras från s och ges till användaren.

```
using(MemoryStream s = new MemoryStream(data)) {
  TripleDESCryptoServiceProvider tdes =
     new TripleDESCryptoServiceProvider();
  using(CryptoStream csr =
     new CryptoStream(s, tdes.CreateDecryptor(NetworkManager.Key,
                         NetworkManager.IV), CryptoStreamMode.Read)) {
    csr.Read( ... );
  }
}
```

5 Funktionell systembeskrivning

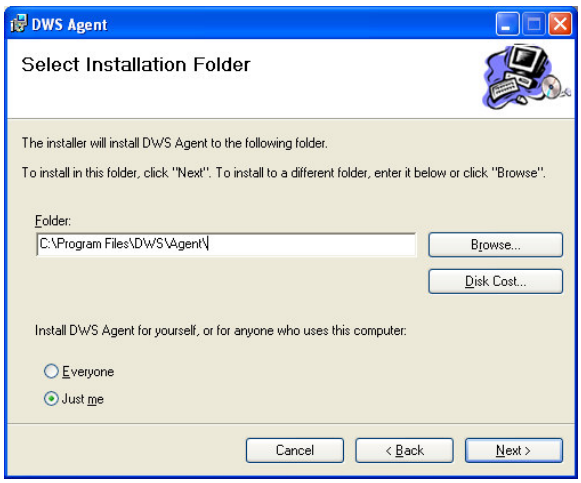
I kommande underkapitel kommer en beskrivning på hur DWS Agent installeras, konfigureras och används. DWS Message Service har inget användargränssnitt och därmed inte dokumenterad i detta kapitel. DWS Manager är inte färdigställd och därmed heller inte dokumenterad. Innan någon applikation installeras skall .NET ramverk version 1.1 installeras.

5.1 DWS Agent

I kommande underkapitel kan installation och användarmanual av DWS Agent läsas. En liknande installation gäller även för DWS Message Service.

5.1.1 Installation

Se till att ramverket .NET är installerat före installationen. Vid insättning av CD-skivan startar installationsprogrammet upp automatiskt. Följ installationsanvisningarna och läs noga igenom licensavtalet. Ändring av sökvägen för installationen är möjlig, se figur 5-1.



Figur 5-1: Skärmdump DWS Agent, installation

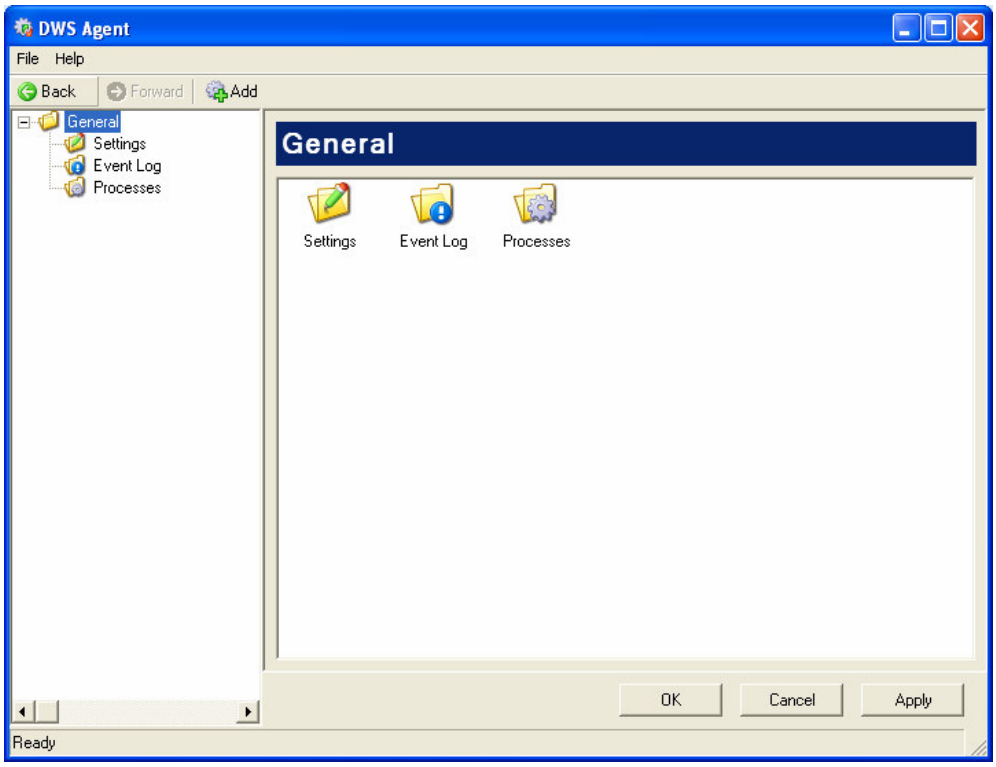
En genväg till DWS Agent skapas automatiskt på skrivbordet samt en på startmenyn.

5.1.2 Användarmanual

Starta DWS Agent genom att till exempel dubbelklicka på genvägen som skapats på skrivbordet under installationen. Vid start av applikationen läggs en ikon i systemlådan. Ikonen visar att agenten är i drift. För att stänga av applikationen kan menyvalet Shutdown under högerklicksmenyn på ikonen i systemlådan väljas. Ett alternativt sätt att stänga av applikationen är genom menyraden i applikationen. Klicka på File och välj sedan Shutdown. Vid klick på minimerings- eller avsluta-knappen minimeras applikationen och kan endast öppnas igen genom högerklicksmenyn på ikonen i systemlådan, Open.

5.1.2.1 Panelen General

Panelen General är den panel som visas vid start av applikationen. Panelen innehåller endast tre ikoner: Setting, Event log och Processes. Genom att dubbelklicka på någon av ikonerna kan navigering till vald panel ske, se figur 5-2. Navigera även till annan panel via trädet som finns placerat till vänster. Knapparna back och forward möjliggör att navigera tillbaka till tidigare besökta paneler.

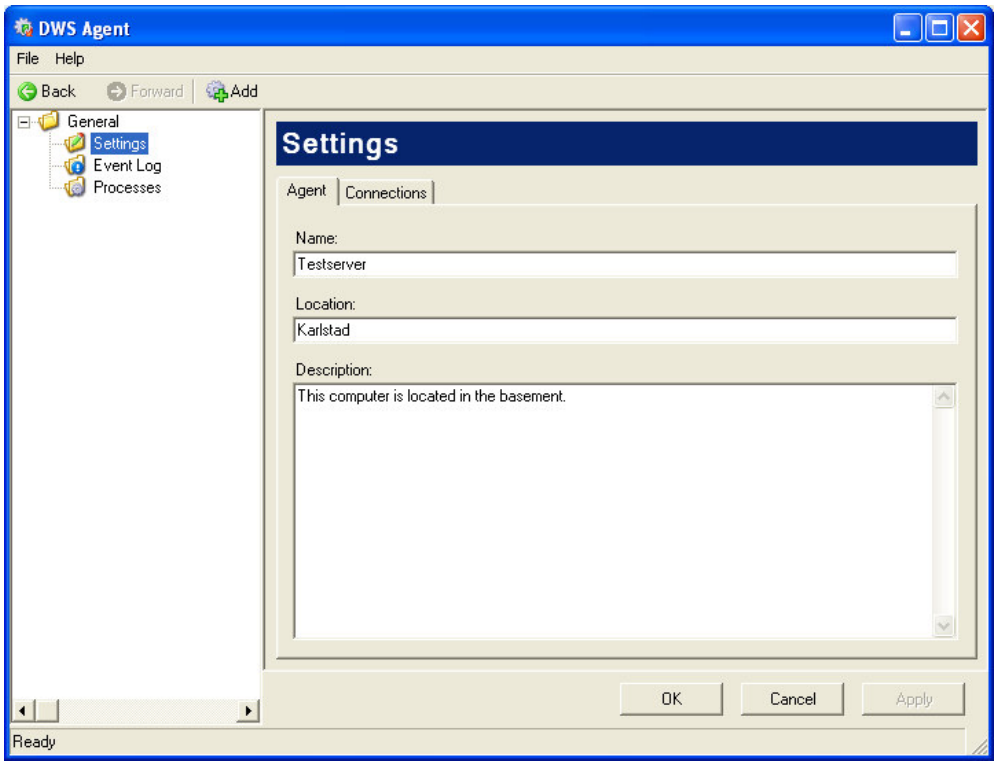


Figur 5-2: Skärmdump DWS Agent, Panelen General

5.1.2.2 Panelen Settings

I panelen Settings konfigureras allmänna inställningar för agenten. Under fliken Agent skall agentens namn, lokalisering samt beskrivning konfigureras, se figur 5-3. Var noga med att konfigurera dessa inställningar. Inställningar påverkar dock inte DWS Agentens funktionalitet lokalt. De är endast till för att routa e-post utskicken till rätt person, läs kapitel 4.3. Agentens namn, lokalisering och beskrivning bifogas även i e-postbrevet.

Under fliken Connections kan adressen och porten till meddelandetjänsten ställas in. Observera att default inställningen är konfigurerad till adress 127.0.0.1 och port 8085. Under samma flik anges även vilken port som agenten skall lyssna på för inkommande anslutningar. Inkommande anslutningar kan endast ske med DWS Manager.

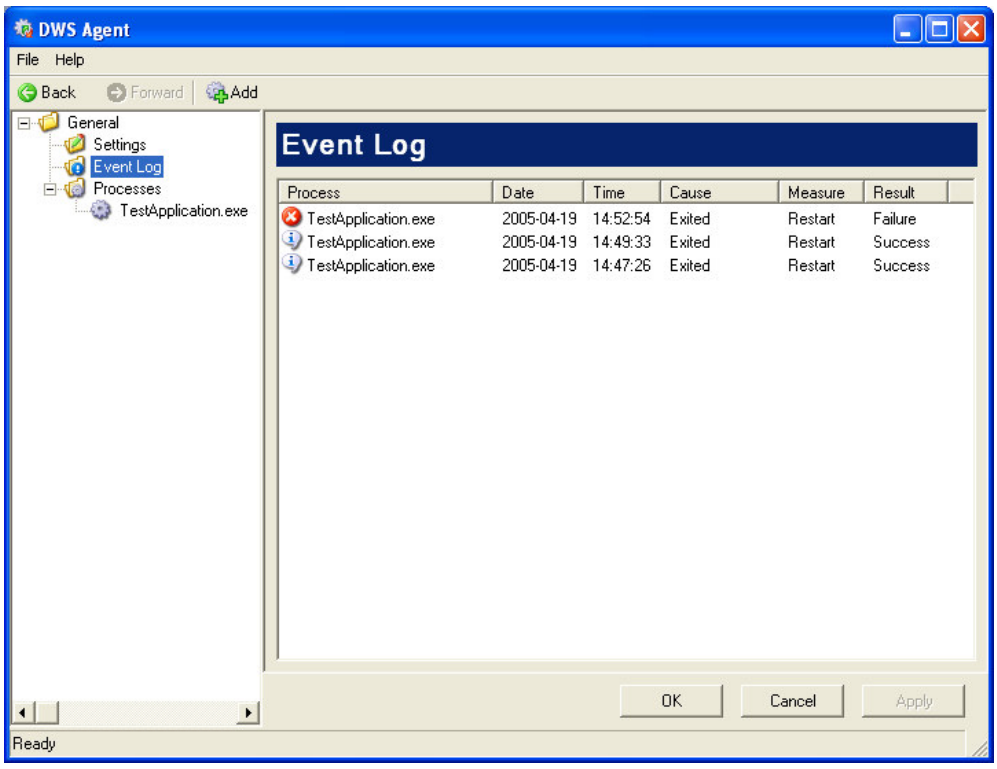


Figur 5-3: Skärmdump DWS Agent, Settings

5.1.2.3 Panelen Event view

Panelen Event view visar en logg för alla driftstörningar. Den första kolumnen anger namnet på den övervakade processen. Den andra och tredje kolumnen består av en tidstämpel på när driftstörningen inträffat. De resterande tre kolumnerna anger orsaken till driftstörningen, åtgärden för felet och om applikationen genomfört åtgärden framgångsrikt eller inte. Om åtgärden misslyckats är ikonen röd med ett kryss, se figur 5-4 översta rad. Däremot om åtgärden lyckats (driftstörningen åtgärdats) är ikonen vit.

Det är enkelt att radera loggens innehåll. Högerklicka på panelen och välj sen Clear all Events för att radera innehållet. Ett alternativt sätt att radera loggen är att högerklicka på noden Event Log i trädet och välja menyvalet Clear all Events. Observera att innehållet kommer att permanentiskt försvinna. Om loggen vill utforskas kan samma information hittas under operativsystemets egen loggutforskare under administratörsverktyg.



Figur 5-4: Skärmdump DWS Agent, Event log

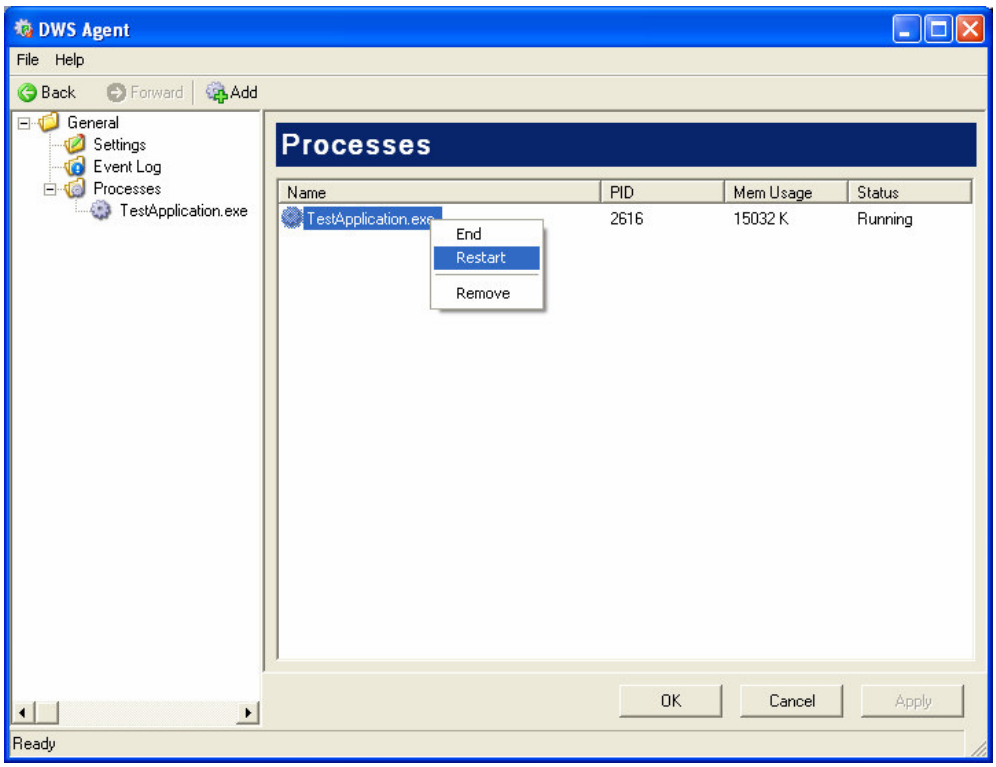
5.1.2.4 Panelen Processes

Panelen Processes visar alla övervakade processer. Första kolumnen består av processens namn. Förutom processens namn finns processens unika id, PID, processens totala minnesanvändning och status hittas. Denna information uppdateras kontinuerligt. Den unika identifieraren PID, är ett unikt nummer som operativsystemet tilldelar en startad process. Identifieraren ändras vid en eventuell omstart av processen. En process kan anta tre olika tillstånd:

- Exekverar
- Avslutad
- Svarar inte

Dubbelklicka på en process för att komma vidare till DWS Agentens inställningar, alternativt klicka på processnoden i trädet.

Genom att högerklicka på en process kommer man åt tre olika menyval: avsluta, starta om och ta bort övervakad process, se figur 5-5. Klicka på önskat menyval och DWS Agent utför operationen omgående. Samma meny kan nås via högerklick på processnoden i navigeringsträdet. Menyvalet Remove avlägsnar en övervakad process från agenten.



Figur 5-5: Skärmdump DWS Agent, Processes

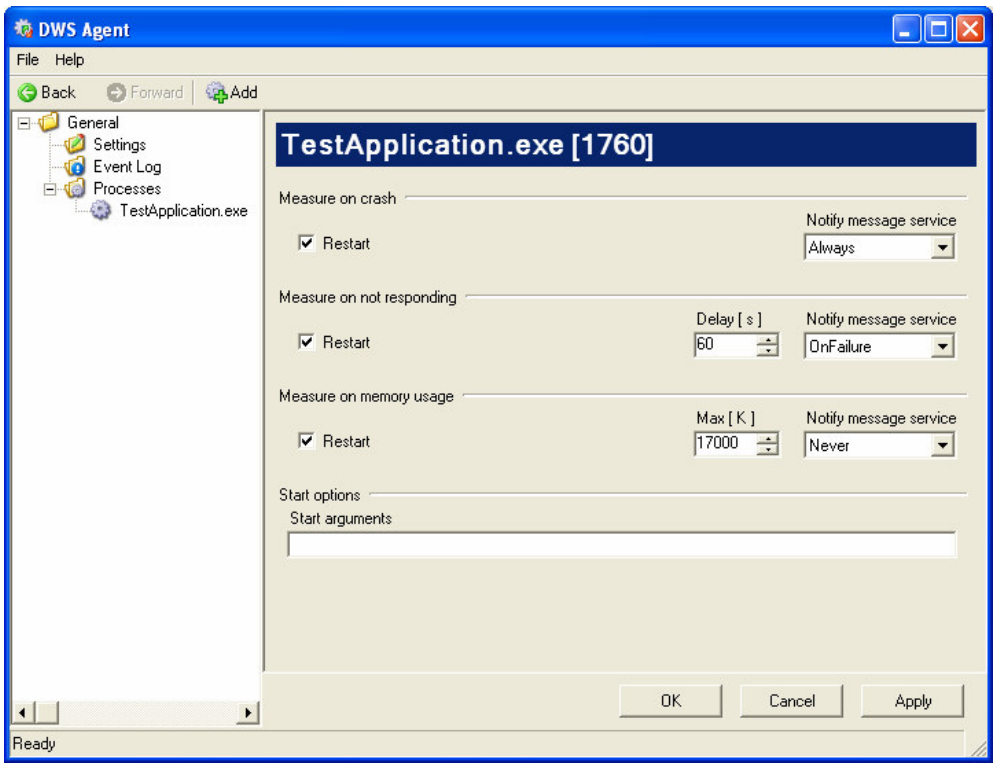
5.1.2.5 Panelen Process

För varje process finns en egen processpanel. Här kan alla processspecifika åtgärder för driftstörning konfigureras. Åtgärd för krasch, om inte processen svarar eller gräns på minnesanvändning är separerade så att olika åtgärder kan tas om så vill. Startargument för omstart av process är även möjligt att konfigurera här.

Det finns möjlighet att konfigurera när inställd meddelandetjänsten ska meddelas. Tre olika fall är möjligt vid en driftstörning.

- Alltid
- Åtgärd misslyckad
- Aldrig

Om alltid är valt kommer meddelandetjänsten meddelas oberoende om åtgärden för en driftstörning lyckats eller inte. Däremot om användaren endast vill meddelas om DWS Agent inte lyckats med att åtgärda driftstörningen skall OnFailure väljas. Alternativet Never medför att meddelandetjänsten aldrig meddelas vid driftstörningar.



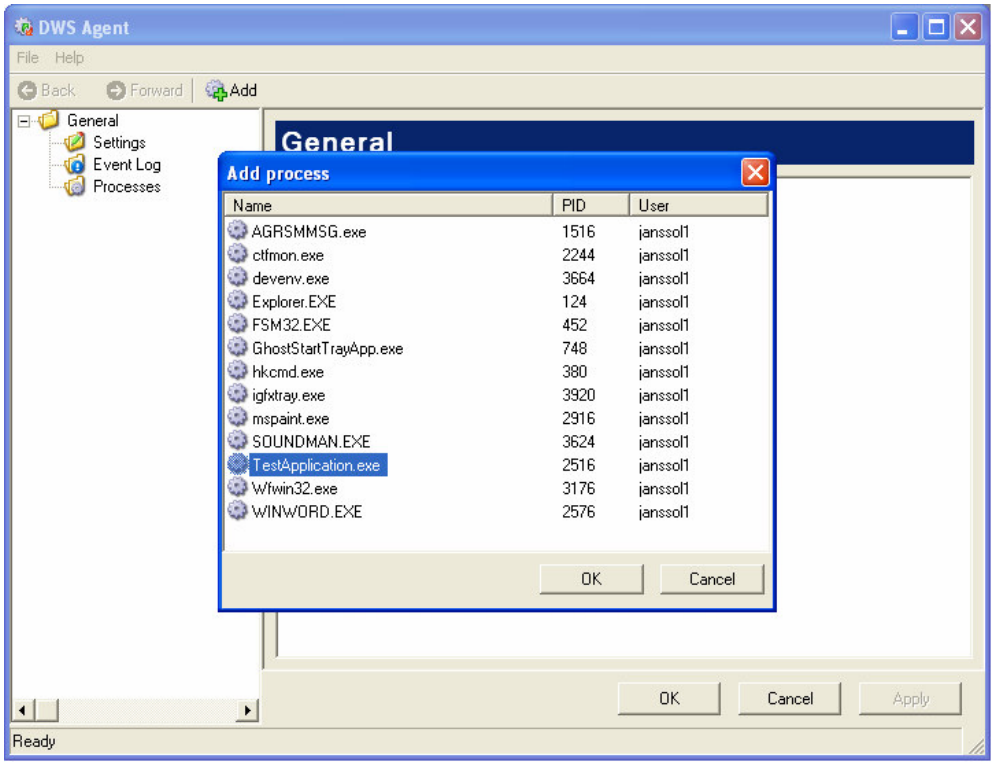
Figur 5-6: Skärmdump DWS Agent, processinställningar

5.1.2.6 Lägg till en process

Att lägga till en process för övervakning är mycket enkelt. Det kan göras på flera sätt.

- Klicka på knappen Add på knappraden under menyraden.
- Högerklicka på noden Processes och välj Add.
- Klicka på File och klicka sedan på menyvalet Add process.

Ett nytt fönster med en lista på alla oövervakade processer kommer fram, se figur 5-7. Välj den process som skall övervakas och klicka på knappen ok. Processen är nu övervakad. Klicka på motsvarande process-nod för att konfigurera önskade åtgärder vid driftstörning.



Figur 5-7: Skärmdump DWS Agent, lägg till en process

6 Slutsats och resultat

Utvecklingen av Distributed Watchdog System har överlag gått bra. Siemens fick tidigt inblick i programvaran och dess gränssnitt och har utvärderat dess funktionalitet allt efter som vilket lett till en bättre produkt. Även de utvecklingsmetoder som använts, först och främst par-programmering, har varit till stor hjälp för att upptäcka buggar och undvika slarvfel.

Tyvärr är inte alla delar i övervakningssystemet färdiga. Det är en hel del kvar att göra på managern innan den kan användas. Det beror till mesta del på att övriga delar tog mer tid än planerat samt att projektet från början vara så pass stort. Både agenten och meddelandetjänsten fungerar däremot utmärkt och är det är tillräckligt för att lösa Siemens problem med DESIGO™ INSIGHT.

Siemens är mycket nöjda med det som presterats och kommer använda lösningen hos sina kunder. De har till och med kommit med nya önskemål om funktionalitet och tillägg för programvaran. Såvida projektet kommer vidareutvecklas är i dagsläget oklart.

Referenser

[1] Siemens, *http://www.siemens.com*, 2005-01-20

[2] Siemend Buildning Technologies, *http://www.landisstaefa.com*, 2005-01-20

[3] Microsoft, *http://www.microsoft.com/windowsserver2003/technologies/directory/activedirectory/default.mspx*, 2005-02-21

[4] Watchdog-O-Matic, *http://www.kwakkelflap.com/*, 2005-02-02

[5] Polarion, *http://www.subversion.tigris.org*, 2005-02-02

[6] Steven John Metsker, *Design Pattern in C#*, Addison-Wesley, 2004

[7] Eric Gunnerson, *A Programmers Introduction to C#*, Apress, 2nd edition, June 2001

[8] Richard Blum, *C# Networking Programming*, Sybex, 2002

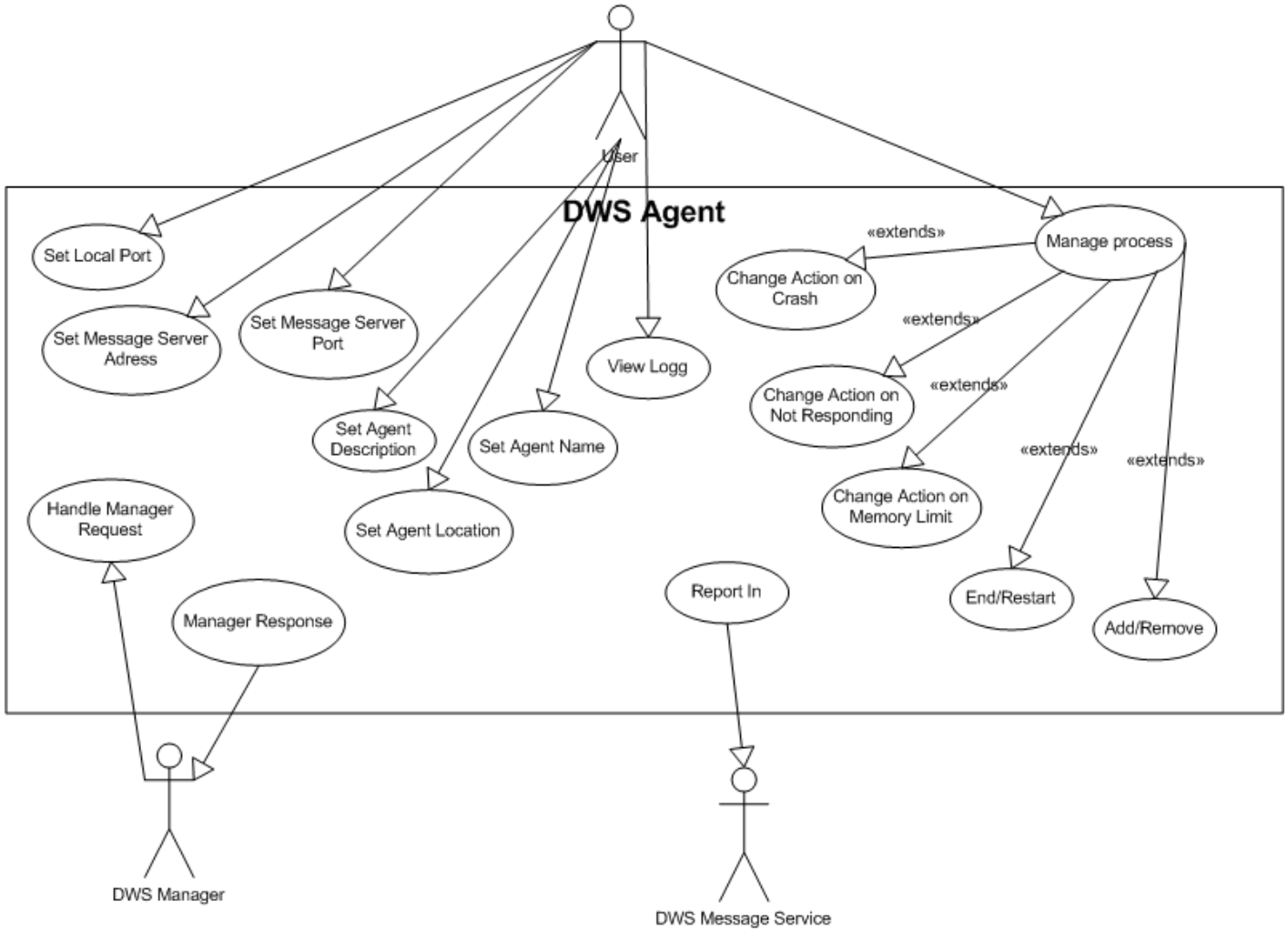
[9] Kendal Scott, *The unified process explained*, Addison-Wesley, 2001

[10] Kent Beck, *Extreme programming explained,* Addison-Wesley, 2001

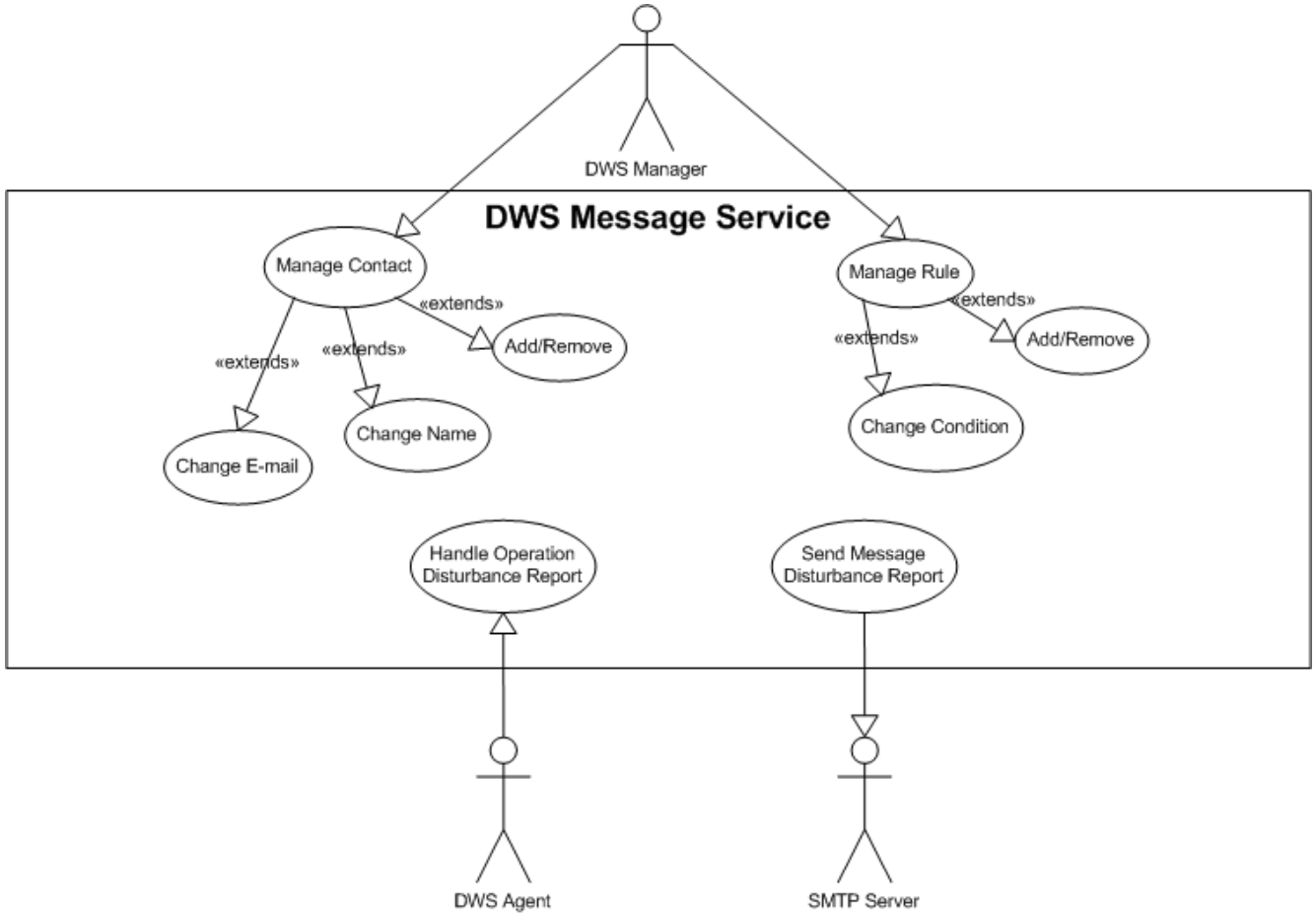
Förkortningar

DWS	Distributed Watchdog System
UML	Unified Modeling Language
XML	Extensible Markup Language
DTD	Definition Type Description
ODBC	Open Database Connectivity
OLEDB	Object Link Embedding Database
DMTF	Distributed Management Task Force
WMI	Windows Management Instrumentation
WBEM	Web-Based Enterprise Management
WQL	WMI Query Language
SNMP	Simple Network Management Protocol
SQL	Simple Query Language
MVC	Model-View-Controller
DoS	Denial of Service
DES	Data Encryption Standard
CLS	Common Language Specification
CLR	Common Laonguage Runtime
MSIL	Microsoft Intermediate Language
SVN	Subversion
CVS	Concurrent Version System

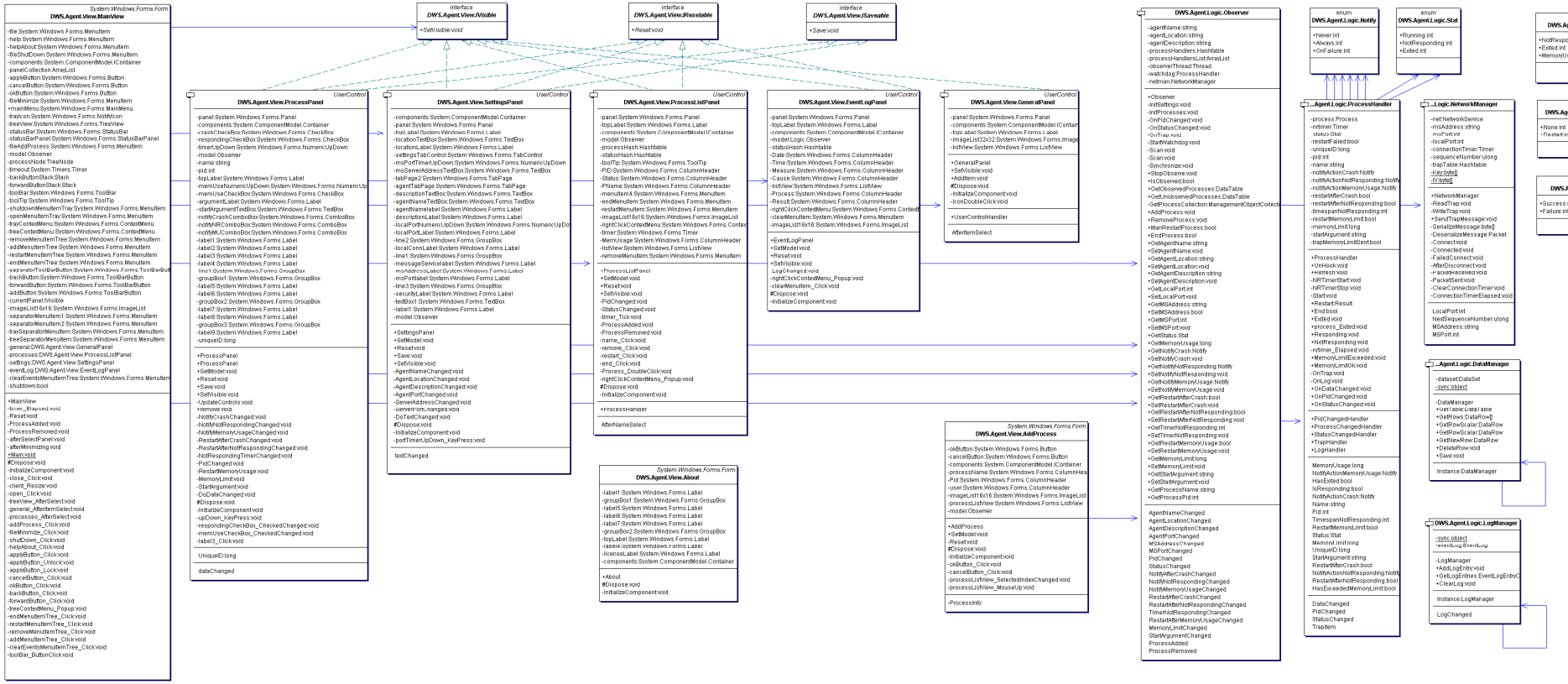
A DWS Agent, Användarfall



B DWS Message Service, Användarfall



C Klassdiagram DWS Agent



D Klassdiagram DWS Message Service