

Beteckning: _____



Institutionen för matematik, natur- och datavetenskap

Metodutveckling för framställande av dynamisk systemkarta

Magnus Flodin
Juni 2007

Examensarbete, 10 poäng, C
Datavetenskap

Dataingenjörsprogrammet
Examinator/handledare: Jonas Boustedt
Medbedömare: Bengt Östberg

Metodutveckling för framställande av dynamisk systemkarta

av

Magnus Flodin

Institutionen för matematik, natur- och datavetenskap
Högskolan i Gävle

S-801 76 Gävle, Sweden

Email:

ndi04mfn@student.hig.se

Abstrakt

I komponentbaserade system är det viktigt att ha kontroll över vilka komponenter som har beroenden till varandra för att undvika att delar av systemet slutar fungera vid exempelvis en komponentuppdatering. I detta arbete undersöks hur komponenter kan identifieras, samt hur olika typer av beroenden kan spåras. Arbetet är utfört som ett uppdrag givet av Sandvik Systems Development (SSD), vilket är ett av Sandvik-koncernens två internationella IT-bolag. Målet med uppgiften är att utforma en modell som tar fram information för komponenterna, samt kartlägger dess beroenden på Windows-plattformen. Mycket information om ett system finns att hämta i olika typer av "statiska" metadata-filer som automatiskt genereras vid skapandet av en lösning i Visual Studio. En projektfil (*.proj) är ett exempel på en sådan fil, då den innehåller specifik information för ett visst projekt. En metod för att erhålla "dynamisk" komponentinformation är att programmatiskt ladda in binärfilen för en komponent och därigenom utvinna dess metadata. Mitt lösningsförslag tar vara på informationen som finns lagrad i de "statiska" metadata-filerna för att identifiera komponenter, samt för att spåra dess beroenden. Denna information kompletteras med information erhållen genom att ladda in komponenters binärfiler.

Nyckelord: Systemkarta, komponenthantering, komponentidentifiering, komponentberoendehantering, konfigurationshantering

Innehåll

1 Inledning	5
1.1 Problem	5
1.2 Syfte	5
1.3 Frågeställning	5
2 Teoretisk bakgrund	6
2.1 Kort om komponenter och komponentbaserade system	6
2.2 Identifiering av komponenter	6
2.3 Beroenden mellan komponenter	6
2.3.1 Statisk beroendehantering	7
2.3.2 Dynamisk beroendehantering	8
2.4 Introduktion till .NET Framework	9
2.4.1 Kompatibilitet mellan olika versioner inom .NET Framework	9
2.4.2 Omdirigering av applikationer	10
2.4.3 .NET assembly	10
2.5 Byggverktyg för .NET-lösningar	10
2.5.1 NAnt	10
2.5.2 MSBuild	11
3 Metod	12
3.1 Vilken typ av information efterfrågas?	12
3.2 Informationskällor i ett Microsoft Visual Studio-system	13
3.2.1 Konfigurationsfiler	13
3.2.2 Solution-filer (.sln)	13
3.2.3 Projektfiler (*.proj)	14
3.2.4 Enterprise Template Projects-filer (.etp)	16
3.2.5 Assembly Manifest	16
3.3 Befintliga analyseringsverktyg	17
3.3.1 NDepend	17
3.3.2 Dependency Walker	17
4 Lösningförslag och genomförande	18
4.1 Lösningsmodell	18
4.2 Utveckling av applikationen SystemScanner	19
4.3 Utveckling av komponenten FileFinder	19
4.4 Utveckling av komponenten SystemAnalyzer	20
4.4.1 Utveckling av basklassen Analyzer	20
4.4.2 Utveckling av klassen SolutionAnalyzer	20
4.4.3 Utveckling av klassen ProjectAnalyzer	21
4.4.4 Utveckling av klassen EtpAnalyzer	21
4.4.5 Utveckling av klassen AssemblyAnalyzer	21
4.4.6 Utveckling av klassen AnalyzerCreator	22
4.5 Utveckling av databasen SystemInfoDb	22
4.6 Utveckling av webbapplikationen SystemViewer	24
5 Resultat av genomförande	25
6 Diskussion	26
6.1 Diskussion av problem under implementationsfasen	26
6.1.1 Fler än ett exemplar av en solution-fil	26
6.2 Förslag till vidareutveckling av lösningsmodell	26
6.2.1 Uppkoppling mot mjukvaruhanteringsystem	26
6.2.2 Release-information för system	26
6.2.3 Schedulerad eller händelsekänslig start av applikation	27
6.2.4 Analysmodul för VB6-komponenter	27
7 Slutsatser	28
8 Referenser	29

9 Bilaga: Ett jämförande analys av två mjukvaruhanteringssystem	30
9.1 Visual SourceSafe vs. Team Foundation.....	30
9.1.1 Skillnader i arkitektur.....	30
9.1.2 Säkerhet och Projekträttigheter.....	30
9.1.3 Pålitlighet	31
9.1.4 Skalbarhet.....	31
9.1.5 Funktionella skillnader.....	31

1 Inledning

Sandvik Systems Development (SSD) är ett av Sandvik-koncernens två internationella IT-bolag som jobbar tätt tillsammans med koncernens affärsområden. SSD är Sandviks eget konsultbolag för processutveckling, IT-arkitektur, projektledning, samt utveckling, införande och förvaltning av system.

1.1 Problem

Inom SSD finns ett stort antal applikationer utvecklade för Windows-plattformen. Dessa applikationer spänner över flera versioner av utvecklingsmiljöer och ramverk, såsom VB-6, .Net 1.0, .Net 1.1 och .Net 2.0. Dessutom har olika applikationer beroenden till olika tredjepartsprodukter.

För att reducera risken för att en förändring av en komponent skall påverka något annat system negativt finns ett behov av att snabbt få en överblick över de beroenden som finns mellan olika system och komponenter. Till exempel skulle man kunna vara intresserad av att veta vilka applikationer som körs på .Net 1.1, eller vilka applikationer som använder sig av 3:e-partskomponenter av version 1.1.

1.2 Syfte

Denna studie avser att definiera en avgränsad informationsmodell som täcker de behov som finns, samt att ta fram en lösning som inhämtar så mycket information som möjligt automatiskt – till exempel genom att analysera källkod och olika typer av filer i SSD:s applikationer.

1.3 Frågeställning

- Vilken typ av information skall tas fram - när behövs den och till vad?
- Vilka metoder kan utvecklas för att finna informationen som söks?
 - Var kan informationen hittas?
 - Hur kan informationen inhämtas automatiskt?
- Hur ska information representeras?
- Hur kan informationen kommuniceras?

2 Teoretisk bakgrund

För att kunna ta fram en lösning som täcker in de problem som presenterats krävs först och främst en förståelse för vad som menas med en komponent, samt hur beroenden generellt yttrar sig i ett komponentbaserat system. I och med att systemet i detta fall finns på en Windows-plattform presenteras också en kort beskrivning av .NET Framework.

2.1 Kort om komponenter och komponentbaserade system

Komponentbaserad mjukvaruutveckling är ett delområde inom mjukvaruutveckling som syftar till att skapa fristående komponenter, vilka kan användas som byggstenar i ett system. Komponenter anses inneha en högre abstraktionsgrad jämfört med objekt, då de inte delar tillstånd eller kommunicerar genom att utväxla data via meddelanden.

Komponentbaserad mjukvaruutveckling är en ansats till mjukvaruutveckling som förlitar sig på återanvändning av mjukvara. Denna teknik uppkom till följd av att man inom objektorienterad utveckling misslyckats med att ge stöd för effektiv återanvändning - enskilda objektklasser anses vara för detaljerade och specifika. Komponenter är i jämförelse med objektklasser mer abstrakta och kan ses som fristående enheter som tillhandahåller tjänster [13].

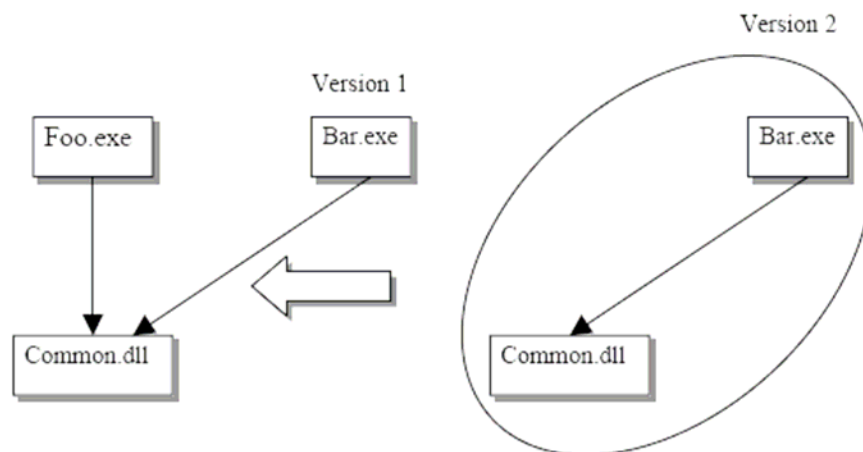
2.2 Identifiering av komponenter

Med att identifiera en komponent, avses här att ta fram nödvändig metadata¹ för vederbörande. Det skulle kunna vara information såsom namn, typ, versionsnummer, datum för senaste ändring etc för komponenten. Det finns olika sätt att ta fram den här typen av information beroende på systemet i fråga.

2.3 Beroenden mellan komponenter

I ett komponentbaserat system är vetskapen om komponenternas beroenden till varandra av stor vikt, inte minst vid systemuppdateringar. Vid en systemuppdatering är det mycket viktigt att i förväg kunna kartlägga vilka komponenter som direkt påverkas av en eventuell uppdatering, samt vilka tredjepartskomponenter som på grund av dess beroenden indirekt påverkas av uppdateringen. Innehållet under denna rubrik baseras till stor del på den datavetenskapliga artikeln [1], författad av Larsson och Crnkovic. I figur 1 illustreras ett exempel där applikationen *foo.exe* slutar fungera efter att komponenten *Common.dll* ersatts av en senare version.

¹ Data om data – en beskrivning av något slags data



Figur 1. Foo.exe slutar fungera när en ny version av Common.dll introduceras [2]²

Att hantera strukturer och beroenden är en viktig del i konfigurationshanteringen. När komponenter samlas in för hantering under exekvering är det viktigt att ha vetskap om komponenternas beroenden. Beroendeanalysen skulle kunna delas upp i två områden: statisk beroendehantering, samt dynamisk beroendehantering.

2.3.1 Statisk beroendehantering

Vid "Software Configuration Management" (SCM), fritt översatt till konfigurationshantering av mjukvara, kan beroenden hanteras av byggverktyget *Make*. Syftet med detta verktyg är att specificera dels beroenden mellan de olika delarna som används vid skapandeprocessen, och dels skapandeprocessen i sig. Specifikationen över beroendena kan antingen tas fram manuellt, eller med hjälp av särskilda verktyg. Ett exempel på ett sådant verktyg är *mkmf*, vilket går igenom de olika delarna och upptäcker dess beroenden. Dessa beroenden refererar explicit till andra "delar" som är inblandade i skapandeprocessen. Dessa beroenden behandlar oftast bara direkta relationer, och om en inkluderad fil i sin tur inkluderar en annan fil är det upp till utvecklaren att explicit definiera dessa filer. Det är alltså inte möjligt att rekursivt ta fram dessa beroenden.

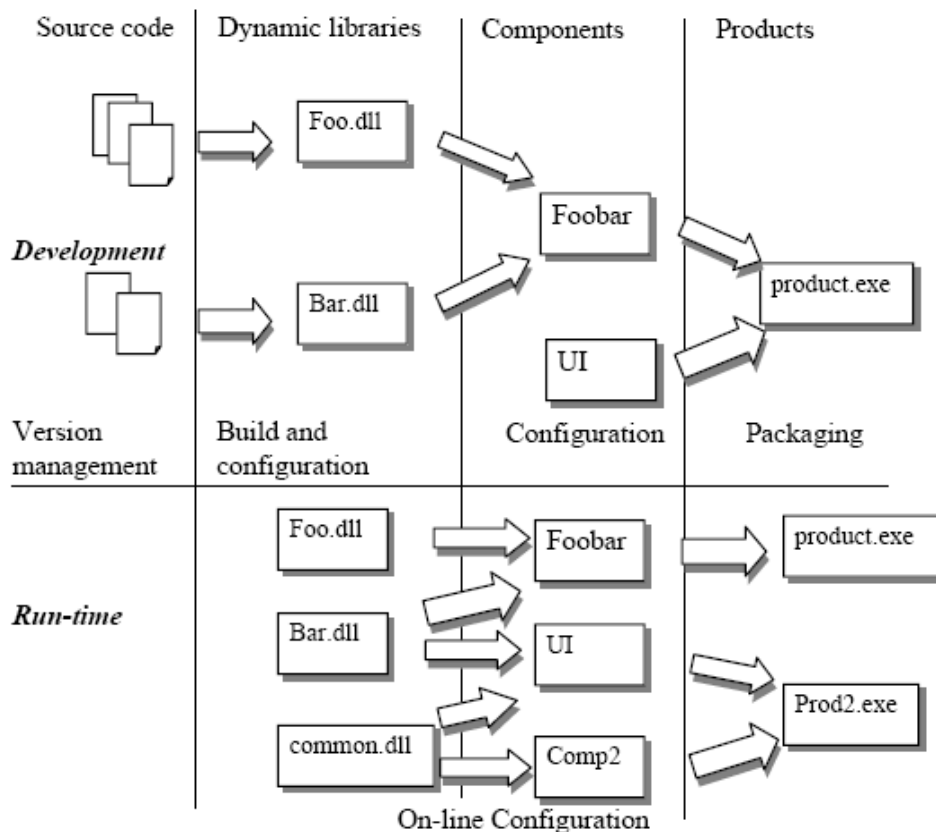
Make-filen kan även innehålla information om de verktyg som använts i skapandeprocessen. På så vis kan en återbyggnadsprocess startas om något verktyg ändrats. Det finns andra definitioner som är mer avancerade varianter av *Make*. Ett sådant exempel är Macros. Notera att beroenden inte innefattar versionshantering av applikationerna, då det antas vara utfört i förväg.

I och med att det ofta kan finnas ett antal implicita relationer och beroenden, kan en *Make*-fil inte vara heltäckande, men för det mesta räcker det till för att vi ska kunna konfigurera och bygga system.

² Med tillåtelse av upphovsmannen används figur 1 och 2.

2.3.2 Dynamisk beroendehantering

Frågan är om man kan använda sig av samma metodik för att kartlägga beroenden vid körning (run-time) som vid utveckling (compile-time). Komponenternas beroenden till varandra har en nära anknytning till hur de är grupperade. Dynamisk gruppering är en viktig funktion hos komponenter. Genom att kartlägga ett systems olika beroenden kan man lätt se vilka konsekvenser en eventuell komponentsättning skulle få. Figuren nedan (figur 2) illustrerar hur statisk respektive dynamisk mjukvaruhantering fungerar i ett komponentbaserat system.



Figur 2. Skildrar statisk vs. dynamisk mjukvaruhantering [2].

Det är för komponentbaserade system möjligt att spåra beroenden både implicit och explicit. De explicita beroendena realiseras genom adresser/pekare som pekar på komponenternas interface, och genom att söka igenom binärfilen för en komponent kan man hitta dessa beroenden. Denna teknik medför att det är möjligt att rekursivt söka igenom filer för att på så sätt ta fram beroenden mellan komponenter. Processen att utföra detta kan dock bli ganska komplicerad, eftersom referenserna bara visar vilka interface och metoder som används och inte till vilken komponent de hör till, vilket i detta fall är av intresse.

Dessa explicita beroenden täcker dock inte in samtliga beroenden. Det kan finnas beroenden mellan komponenter trots att de inte nödvändigtvis kommunicerar direkt med varandra. Ett exempel på ett sådant implicit beroende skulle kunna vara när två komponenter har gemensam data, där den ena komponenten förbereder data för den andra.

2.4 Introduktion till .NET Framework

Microsoft .NET Framework är en utvecklingsmiljö som gör det möjligt att kombinera ett flertal olika programmeringsspråk och bibliotek för att gemensamt skapa Windows-baserade applikationer [3].

Microsoft började utveckla .NET Framework under sent 90-tal, och i slutet av år 2000 släpptes den första betaversionen av .NET 1.0. I tabellen nedan presenteras de versioner av .NET Framework som i skrivande stund finns tillgängliga, samt den version av Microsoft Visual Studio i vilket ramverket inkluderats.

Tabell 1. Release-information för olika versioner av .NET-ramverket [4].

Version	Release-år	Kommentar
.NET Framework 1.0	2002	Del av Visual Studio .NET 2002
.NET Framework 1.1	2003	Del av Visual Studio .NET 2003
.NET Framework 2.0	2005	Del av Visual Studio 2005
.NET Framework 3.0	2006	-

2.4.1 Kompatibilitet mellan olika versioner inom .NET Framework

Microsoft .NET Framework strävar efter att bibehålla kompatibilitet både bakåt och framåt mellan de olika versionerna. Likväl kan dock en förändring som ökar säkerheten, korrektheten eller funktionaliteten åstadkomma kompatibilitetsproblem.

Med stöd för bakåtkompatibilitet i ett ramverk menas att en applikation, exempelvis skriven i .NET Framework version 1.0 även kan köras på version 1.1. Ramverket har ett mycket bra stöd för bakåtkompatibilitet. De flesta applikationer som fungerar på en viss version av .NET Framework kommer även att fungera för nästföljande version [5].

Framåtkompatibilitet i ett ramverk är då det motsatta, det vill säga att en applikation som är skriven för .NET Framework version 1.1 även ska fungera för version 1.0. Trots att .NET Framework stödjer framåtkompatibilitet kommer applikationer som använder specifika typer eller medlemmar från version 1.1 inte att fungera fullt ut på version 1.0. Detta bör dock inte ses som ett brott mot framåtkompatibiliteten eftersom applikationen aldrig kan förväntas fungera. Vill man att applikationen ska fungera fullt ut på olika versioner av .NET Framework ska man bara använda sig av typer och medlemmar av den av dessa tidigaste versionen [5].

2.4.1.1 Kompatibilitetsbrytande förändringar

En kompatibilitetsbrytande förändring kan antingen vara en binär- eller en funktionell förändring för typer eller medlemmar i .NET Framework, vilka medför bakåt- eller framåtinkompatibilitet. En bakåtbrytande förändring medför bakåtinkompatibilitet, medan en framåtbrytande förändring medför framåtinkompatibilitet. Säkerhetsförbättringar är det vanligaste motivet till bakåtbrytande förändringar, medan framåtbrytande förändringar generellt tas till för att åtgärda funktionella fel i

tidigare versioner av ramverket [5].

Med en binärt brytande förändring menas att signaturen för en typ eller medlem är ändrad. Det skulle kunna vara allt från ett namnbyte av en metodparameter till ett namnbyte av en metod eller ändring av returtyp. Binära förändringar leder ofta till bakåt- och framåtinkompatibilitet [5].

En förändring som medför att funktionaliteten hos en typ eller medlem ändras kallas för en funktionellt brytande förändring. Den här typen av förändringar är ofta svårare att upptäcka än en binärt brytande förändring. En funktionellt brytande förändring kan leda till bakåt-, framåt-, eller både bakåt- och framåtinkompatibilitet. Det är sällan en sådan funktionell förändring genomförs på .NET Framework förutom när ett kodningsfel behöver rättas till, eller när säkerheten och pålitligheten behöver förbättras [5].

2.4.2 Omdirigering av applikationer

Som standard kan en applikation endast köras på den version av .NET Framework, på vilken den kompilerades. Vill man att en applikation ska kunna köras på fler versioner av ramverket måste det anges i en konfigurationsfil. Att ange vilken version av .NET Framework en applikation ska använda kallas att omdirigera applikationen. En applikation kan bli omdirigerad till att antingen använda endast en specifik version av .NET-ramverket, eller att välja ut en ur en given samling kandidatversioner [5].

2.4.3 .NET assembly

En assembly är en PE-fil (Portable Executable). Med det menas att filen antingen är en process-assembly (EXE), eller en biblioteks-assembly (DLL). En process-assembly utgör en process som använder sig av klasser som finns definierade i en eller flera biblioteks-assembly'n [17].

2.5 Byggverktyg för .NET-lösningar

Tidigare i uppsatsen (sektion 2.3.1) har vid ett antal tillfällen byggverktyget *Make* nämnts, vilket inom datavetenskapen är ett välkänt verktyg som används för att automatisera bygget av stora applikationer. De filer som specificerar instruktionerna för *Make* kallas *Makefiles*. *Make* används framförallt för C/C++-projekt, men i princip kan det användas för nästan alla kompillerbara programspråk.

Kompilatorn är det elementära verktyget för att bygga en exekverbar applikation av källkod. *Make* är en separat enhet som meddelar kompilatorn vilka källkodsfiler som ska kompileras. Den spårar vilka filer som har ändrats sedan projektet senast byggdes, och kompilatorn kompilerar endast de komponenter som har beroenden till dessa filer [8].

2.5.1 NAnt

NAnt är ett byggverktyg utvecklat för .NET Framework. På hemsidan för *NAnt* [9], beskrivs produkten med följande citat: "In theory it is kind of like make without make's wrinkles. In practice it's a lot like Ant.". *Ant* och *NAnt* är två mjukvaruverktyg för automatiserad kompilering [10]. Båda dessa verktyg liknar *Make*, men *Ant* är

framförallt utvecklat för Java-projekt, medan *NAnt* riktar sig, som tidigare nämnts mot .NET-projekt. En märkbar skillnad mellan *Make* och *Ant/NAnt* är att de senare använder XML för att beskriva byggprocessen och dess beroenden, medan *Make* har sitt *Makefile*-format.

2.5.2 MSBuild

MSBuild är en byggplattform för Microsoft och Visual Studio. Verktygen för *MSBuild* finns inbyggda i .NET Framework 2.0, och finns därmed tillgängliga i Visual Studio 2005 (8.0). *MSBuild* hanterar *MSBuild*-projekt, vilka har liknande XML-syntax som *Ant* och *NAnt*. Trots att dess syntax är baserad på XML är dess grundläggande struktur jämförbar med tidiga Unixbaserade *Make* [11].

3 Metod

Det finns olika sätt att utvinna information ur ett system. I denna sektion följer bland annat en beskrivning av några informationskällor varifrån metadata kan erhållas. Först och främst bör man dock fastställa vilken information som är av intresse att ta fram.

3.1 Vilken typ av information efterfrågas?

En central fråga är vilken typ av information för system och komponenter som egentligen efterfrågas. Det är lätt hänt att man tar med för mycket information utan att motivera dess nytta. Vid samtal med den person på SSD som har det övergripande ansvaret för systemutvecklingen på Windows-plattformen diskuterades vilken typ av information som var motiverad att ta fram. Vi kom fram till att det vore värdefullt att få ut följande information:

- Namn på systemet
- Sökväg till systemets rot
- Vilket mjukvaruhanteringssystem som använts
- Datum för senaste release, samt näst senaste release
- Vilka komponenter/projekt systemet består av:
 - Namn
 - Sökväg
 - Typ av komponent (klient- eller webbkomponent)
 - Version av .NET-ramverket den skapats på
 - Beroenden till andra komponenter
 - Assemblyversion

Namn och sökväg till system och komponenter används för identifikation respektive lokalisering av vederbörande. Genom att SSD använder sig av två olika mjukvaruhanteringssystem skulle en upplysning om vilket av dessa system som använts för en lösning ge en bättre helhetsöversikt, samt underlätta lokalisering av lösningen. Se bilaga (sektion 9.1) för närmare information om dessa två mjukvaruhanteringssystem och vad som skiljer dem åt.

För att få en uppfattning om hur ofta en ny version av ett specifikt system släpps, samt för att hålla reda på när den senaste versionen släpptes skulle det vara värdefullt att lagra information om datum för senaste release, samt datum för näst senaste release av systemet. På så vis skulle denna information kunna ligga till grund för eventuella ställningstaganden om huruvida utvecklingsarbetet av ett system borde omdisponeras. Till exempel kanske man upptäcker att nya versioner av ett visst system släpps alldeles för sällan. Att släppa nya versioner inom kortare tidsintervall kanske kunde leda till ett bättre arbetsflöde, trots att färre uppdateringar/ändringar av systemet hinner göras mellan varje release.

Genom att för samtliga komponenter för varje enskilt system lagra information om vilken version av .NET Framework komponenterna är skapade på kan man utföra en viss typ av beroendeanalys. Man skulle då till exempel kunna spåra samtliga .NET 2.0-applikationer. Läger man dessutom till information om varje komponents typ, såsom klientapplikationer eller webbapplikationer kan man erbjuda denna analys

ytterligare en dimension. Då skulle man exempelvis kunna spåra samtliga .NET 1.1-webbapplikationer, eller alla klientapplikationer baserade på .NET 2.0.

Genom att för varje komponent lagra information om vilka andra komponenter denna har beroenden till skapas ett informationsunderlag för analys av komponentberoenden. Med hjälp av denna information kan vi se vilka komponenter som krävs för att en annan komponent ska fungera. Ett annat ändamål skulle kunna vara att spåra vilka applikationer som använder sig av en viss tredjepartskomponent.

3.2 Informationskällor i ett Microsoft Visual Studio-system

Här följer en beskrivning av var man kan finna metadata i ett system utvecklat i Microsoft Visual Studio .NET.

3.2.1 Konfigurationsfiler

En konfigurationsfil för en applikation är en XML-fil som innehåller information och inställningar specifikt för en applikation. Filen innehåller konfigurationsinställningar som CLR (Common Language Runtime) läser, och inställningar som själva applikationen kan läsa [6].

Namnet och lokaliseringen av konfigurationsfilen för en applikation beror på applikationens värd (eng. host). För en applikation vars värd är exekverbar (eng. executable host) finns konfigurationsfilen i samma katalog. Namnet på konfigurationsfilen är detsamma som för applikationen, men med tillägget ".config". Om exempelvis en applikation heter "myApp.exe", ska konfigurationsfilen ha namnet "myApp.exe.config" [6].

För en webbapplikation finns defaultinställningar för konfigurering specificerade i en fil med namnet *Machine.config*, vilken finns lokaliserad i katalogen "%SystemRoot%\Microsoft.NET\Framework\versionNumber\CONFIG\". Vårdens konfiguration ärvs av sub-siter och sub-applikationer. Om det finns en konfigurationsfil i sub-siten eller sub-applikationen kommer de ärvda värdena inte att användas, men de kan bli överlagrade och finns då tillgängliga för konfigurations-API'et. En konfigurationsfil för en webbapplikation har namnet "Web.config" [7].

3.2.2 Solution-filer (.sln)

Visual Studio skapar två separata filer när man skapar en ny lösning (eng. solution). Den ena är en *.suo*-fil (solution user options-fil), där användarspecifik information finns lagrad. Filen är dold och skulle till exempel kunna innehålla information om var användarens brytpunkter (eng. breakpoints) finns lokaliserade.

Den andra filen som skapas är en *.sln*-fil (solution-fil). Här lagras information om vilka projekt som lösningen består av, inställningar för mjukvaruhanteringssystem och andra globala inställningar som rör samtliga projekt i lösningen. Här finns till exempel information om vilket mjukvaruhanteringssystem som använts vid utvecklandet av lösningen. Följande figur (figur 3) visar strukturen för en solution-fil. Denna exempelfil är inte en fullständig solution-fil; endast de delar som är väsentliga för denna studie finns med.

```

Microsoft Visual Studio Solution File, Format Version 8.00
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Sandvik.WebServices.Authorization",
    "http://localhost/Sandvik.WebServices.Authorization/Sandvik.WebServices.Authorization.csproj",
    "{2F707FAF-D5A6-468D-9FA4-81A2B5DF630C}"
    ProjectSection(ProjectDependencies) = postProject
    EndProjectSection
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Sandvik.WebServices.Authorization.Test",
    "C:\Test\Sandvik.WebServices.Authorization.Test\Sandvik.WebServices.Authorization.Test.csproj",
    "{B4CBDA45-1CA2-4C81-8A68-3B30754FA7A5}"
    ProjectSection(ProjectDependencies) = postProject
    EndProjectSection
EndProject
Global
    GlobalSection(SourceCodeControl) = presolution
        SCCNumberOfProjects = 2
        SCCProjectName0 = \u0022$/Applications/AuthorizationService/source\u0022,\u002000YNEAAAA
        SCCLocalPath0 = .
        SCCProvider0 = MSSCCI:Microsoft\u0020visual\u0020sourceSafe
        CanCheckoutShared = false
        SolutionUniqueID = {9BD89619-6562-4B8A-BF8E-A03EB09F40D5}
        ...
    EndGlobalSection
EndGlobal

```

Figur 3. Figuren visar relevanta delar ur en solution-fil.

För varje listat projekt finns information om dess typ, dess namn, dess sökväg relativt lösningens rot, samt dess unika id (GUID). Projektets GUID (Globally Unique Identifier) används bland annat till att spåra beroenden. Vilken typ av projekt det handlar om representeras också av ett GUID. Värt att lägga märke till är att en C#-Windowsform och ett C#-bibliotek representeras av samma GUID, eftersom dessa projekttyper egentligen är desamma, förutom att dess inställningar för output-typ skiljer sig. I solution-filen lagras dessa attribut i den ordning de just presenterades.

En viss typ av projektberoenden kan också spåras i lösningsfilen. Dessa beroenden är inte de som skapas med projektreferenser, vilka deklarerats i projektfilerna (se sektion 3.2.3). I lösningsfilen definieras istället de beroenden som anger när ett visst projekt måste byggas före ett annat, utan att projekten nödvändigtvis direkt refererar till varandra. Den här typen av information finns lagrad inom en tagg (eng. tag) i lösningsfilen med namnet *ProjectSection* för *Visual Studio .NET 2003* och *Visual Studio 2005*. I *Visual Studio .NET 2002* däremot lagras den här typen av beroendeinformation inom en "tagg" med namnet *GlobalSection*.

Under taggen *Global* finns konfigurationsinställningar för projekten, samt information för mjukvaruhantering (eng. source control). I figuren ovan kan vi till exempel utläsa att mjukvaruhanteringssystemet *Microsoft Visual SourceSafe* använts (se medlem *ScProvider*).

3.2.3 Projektfiler (*.proj)

Varje projekt skapar ett antal filer för att lagra information om sig själv, så kallad metadata. En av dessa filer är en så kallad projektfil. Filändelsen för en projektfil är baserad på programspråket projektet är skapat i. Om det till exempel handlar om ett C#-projekt har projektfilen filändelsen *.csproj*, medan ett Visual Basic-projekt lagras med filändelsen *.vbproj*. Dess interna format är dock baserade på samma XML-schema.

En projektfil inleds med lite grundläggande projektinformation; bland annat vilken version av Visual Studio som använts, samt projektets GUID. Därefter följer information om bygginställningar, konfigurationsinställningar, samt referensinformation. Följande figur (figur 4) visar hur en projektfil är strukturerad. Denna exempelfil är inte en fullständig projektfil, utan endast de för denna studie betydande fragment finns med.

```

<visualStudioProject>
  <CSHARP
    ProjectType = "web"
    ProductVersion = "7.10.3077"
    SchemaVersion = "2.0"
    ProjectGuid = "{BB5CBDBC-7B38-41C6-B59B-A4C58E9C0F22}"
    ...
  >
  <Build>
    <Settings
      AssemblyName = "Sandvik.Applications.GSSFollowUp.web"
      OutputType = "Library"
      ...
    >
    ...
  </Settings>
  <References>
    <Reference
      Name = "System"
      AssemblyName = "System"
      HintPath = "..\..\..\WINNT\Microsoft.NET\Framework\v1.1.4322\System.dll"
    </Reference>
    <Reference
      Name = "Sandvik.Applications.GSSFollowUp.Report.Entities"
      Project = "{1D1EA13E-251F-4410-8F98-1128950CBAF3}"
      Package = "{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}"
    </Reference>
    <Reference
      Name = "log4net"
      AssemblyName = "log4net"
      HintPath = "bin\log4net.dll"
    </Reference>
  </References>
</Build>
<Files>
  <Include>
    <File
      RelPath = "AssemblyInfo.cs"
      SubType = "Code"
      BuildAction = "Compile"
    </File>
    <File
      RelPath = "default.aspx"
      SubType = "Form"
      BuildAction = "Content"
    </File>
    ...
  </Include>
</Files>
</visualStudioProject>

```

Figur 4. Figuren visar relevanta delar ur en projektfil.

I ovanstående exempel-projektfil kan vi utläsa att projektet är av typen "Web". Med detta menas att det aktuella projektet är en webbapplikation/webbkomponent. Ett projekt kan antingen vara av typen "Web", eller av typen "Local". Det senare påvisar att det aktuella projektet utgör en klientapplikation/klientkomponent.

Medlemmarna *ProductVersion* och *SchemaVersion* i projektfilen talar om vilken version av Visual Studio som använts vid skapandet av projektet. I projektfilen i exemplet ovan deklaras *ProductVersion* 7.1 ("7.10.3077"), och *SchemaVersion* 2.0. Detta motsvarar *Visual Studio .NET 2003*. För *Visual Studio .NET 2002* har *ProductVersion* värdet 7.0, och *SchemaVersion* värdet 1.0. Projektfiler för *Visual Studio 2005*-projekt ser lite annorlunda ut, då dessa är MSBuild-filer. Dessa filer följer ett annat XML-schema, men dess ändamål är detsamma.

Under bygginställningar (<Build><Settings>) finns bland annat information om assemblynamn och output-typ för projektet (se figur 4).

Vid referensinformationen listas alla assembly'n och projekt som det aktuella projektet refererar till. En referens till en assembly innehar medlemmarna *Name*, *AssemblyName* samt *HintPath*, där den senaste anger sökvägen till aktuellt assembly relativt projektfilens sökväg. En referens till ett annat projekt ser lite annorlunda ut. Här hittar man istället medlemmarna *Name*, *Project* och *Package*, där de två senare representeras av varsitt GUID.

Sist i projektfilen listas alla filer som ingår i projektet. För varje filreferens deklarerar *RelPath*, vilket motsvarar sökvägen till filen relativt projektfilens sökväg.

3.2.4 Enterprise Template Projects-filer (.etp)

I sektion 3.2.2 finns beskrivet hur solution-filer bland annat deklarerar vilka projekt som utgör lösningen, eller rättare sagt deklarerar där sökvägen till dess projektfiler. I *Visual Studio .NET 2003* kan man i solution-filen, förutom projektfiler även ”peka ut” Enterprise Template Projects-filer (ETP).

En ETP-fil skulle i det här fallet kunna ses som en extra länk mellan en solution-fil och dess projektfiler. I ETP-filen deklarerar en logisk delmängd av lösningens samtliga projekt. Figuren nedan (figur 5) visar hur delar av en ETP-fil kan se ut.

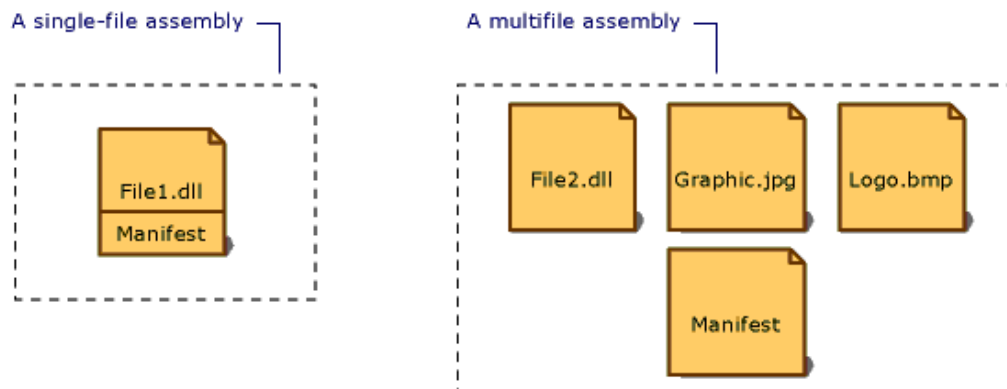
```
<?xml version="1.0"?>
<EFPROJECT>
  <GENERAL>
    ...
    <References>
      <Reference>
        <FILE>Microsoft.ApplicationBlocks.Data\Microsoft.ApplicationBlocks.Data.csproj</FILE>
        <GUIDPROJECTID>{30933672-466E-4F67-A111-ABF267539146}</GUIDPROJECTID>
      </Reference>
    </References>
  </GENERAL>
</EFPROJECT>
```

Figur 5. Figuren visar relevanta delar ur en ETP-fil.

Som man kan utläsa ur figuren ovan listas varje projekt med dess sökväg relativt ETP-filens sökväg, samt dess GUID inom en *Reference*-tagg. I detta exempel refererar ETP-filen endast ett projekt.

3.2.5 Assembly Manifest

Varje assembly, oavsett om den är statisk eller dynamisk, innehåller en samling data som beskriver hur dess element relaterar till varandra. Metadata för en assembly finns lagrat i något som kallas *assembly manifest*. Här finns bland annat all metadata som behövs för att definiera omfånget (eng. scope) för en assembly, samt för att bestämma referenser till resurser och klasser. Detta *assembly manifest* kan lagras antingen i en PE-fil (se sektion 2.4.3) tillsammans med *Microsoft Intermediate Language*-kod (MSIL), eller i en fristående PE-fil som endast innehåller *assembly manifest*-information. Figuren nedan (figur 6) illustrerar hur det kan se ut.



Figur 6. En illustration över hur manifestet kan lagras för en assembly.

För en assembly som består av endast en fil bakas manifestet in i PE-filen för att åstadkomma ett "single-file assembly". Om en assembly består av flera filer kan manifestet antingen vara fristående, eller bakas in i en av PE-filerna.

Ett manifest för en assembly har bland annat följande funktionalitet [12]:

- Enumererar (räknar upp) filerna som finns i en assembly
- Styr hur typer och resurser mappas till de filer som innehåller dess deklARATIONER och implementationer
- Enumererar de assembly'n som denna assembly har beroenden till
- Skapar en typ av sekundärt beteende mellan konsumenterna av en assembly och dess implementationsdetaljer

3.3 Befintliga analyseringsverktyg

Det finns idag olika verktyg på marknaden som analyserar system och kartlägger dess beroenden. Två exempel på sådana verktyg är *NDepend* samt *Dependency Walker*. *NDepend* är en kommersiell produkt, medan *Dependency Walker* kan laddas ner och användas kostnadsfritt.

3.3.1 NDepend

NDepend är ett verktyg för .NET-utvecklare med vilket man kan kontrollera komplexiteten, kvaliteten och utvecklingen av .NET-kod. Verktöget analyserar källkod och assembly'n och genererar utifrån denna information en rapport. Systemet visualiseras via ett grafiskt användargränssnitt [14].

NDepend har ett antal olika funktioner. Beroenden mellan assembly'n i en applikation kan till exempel presenteras i tabellform, eller i form av ett diagram [15].

3.3.2 Dependency Walker

Dependency Walker är ett gratisverktyg som scannar Windows-moduler (32- och 64-bit) och bygger ett hierarkiskt träd-diagram av de moduler som har beroenden till varandra. För varje upptäckt modul listas alla funktioner, och vilka av dessa som verkligen anropas från andra moduler. För varje enskild fil presenteras information såsom full sökväg till filen, basadress, versionsnummer, maskintyp, debuginformation med mera [16].

4 Lösningsförslag och genomförande

De två befintliga analyseringsverktyg som i korthet beskrivits i sektion 3.3 tar fram delar av den information som i denna studie efterfrågas, men dessa verktyg är i sig inte skräddarsydda utifrån de önskemål som finns. Ett lösningsalternativ skulle kunna vara att läsa in intressanta delar ur resultatet som ett externt verktyg genererat, och sedan komplettera med övrig önskad information via ett eget separat analyseringsverktyg.

I och med att analyseringsverktyget *Dependency Walker* är ett gratisverktyg kunde det laddas ned för en testkörning. Det visade sig att det var möjligt att spara analyseringsresultatet i textformat. Genom att pars³ denna resultatfil vore det möjligt att tillgodogöra sig resultatet. Detta tillvägagångssätt har dock uteslutits på grund av att det inte är tillåtet att integrera *Dependency Walker* med andra verktyg. För verktyget *NDepend* har jag inte satt mig in hur det är på det området, men eftersom det handlar om en kommersiell produkt samtidigt som en tämligen omfattande analys av dess resultat måste genomföras för att tillgodogöra sig informationen har jag valt att lösa uppgiften utan inblandning av externa verktyg.

Genom att skapa ett system som inhämtar information som finns lagrad i solution-, ETP- samt projektfiler kan en systemkarta med grundläggande komponentinformation och information om dess beroenden tas fram. En approach som torde uppfattas som naturlig för att genomföra detta är att utgå från en solution-fil, analysera den, och via dess projektlisning lokalisera dess projektfiler som i sin tur analyseras. Om ETP-filer används i lösningen initieras ytterligare ett steg till processkedjan. I det fallet analyseras först solution-filen där dess ETP-filer lokaliseras, vilka i sin tur pekar ut projektfilerna.

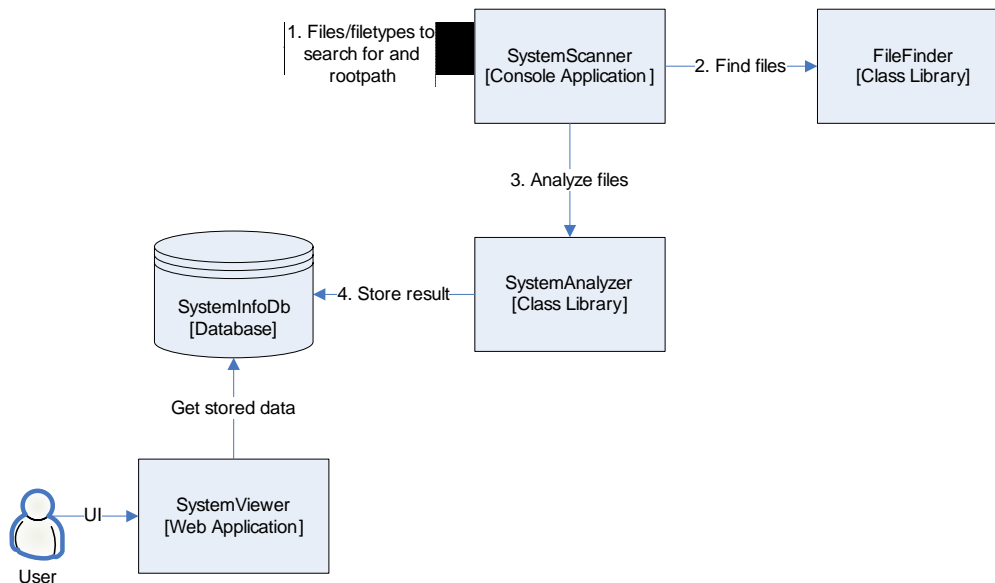
Om ett projekt har ett beroende till en assembly (eller flera) är det av intresse att veta vilken version av denna assembly projektet refererar till. Genom att ladda in denna assembly, vars sökväg publiceras i projektfilen kan den typen av information tas fram programmatiskt.

Lösningförslaget är implementerat med utvecklingsverktyget *Visual Studio 2005*.

4.1 Lösningmodell

För att lättare förstå vad kommande beskrivna komponenter/applikationer (sektion 4.2-4.6) har för syfte, samt hur de inbördes hänger ihop skapades följande modell (figur 7).

³ Tolka/analysera med hjälp av ett datorprogram



Figur 7. Översiktlig modell över lösningsförslagets utformning och parternas syfte.

I figuren skildras hur komponenter/applikationer samarbetar för att utföra önskad uppgift, och hur erhållen informationen hanteras. Här illustreras också arbetsflödet, samt hur en ”användare” kan tillgodogöra sig resultatet.

4.2 Utveckling av applikationen SystemScanner

Den styrande applikationen varifrån hela systemanalysen påbörjas kallas *SystemScanner*. Detta är en konsolapplikation som genom att anlitna övriga komponenter i lösningen åstadkommer önskat resultat. När applikationen startas anges den fil/filtyp som ska eftersökas, samt sökväg till den katalog varifrån sökningen ska utgå. Denna information används sedan av komponenten *FileFinder*, vilken beskrivs i nästa sektion (sektion 4.3). När *FileFinder* slutfört filsökningen anlitas komponenten *SystemAnalyzer* (se sektion 4.4). För denna komponent finns olika analysmoduler tillgängliga, och rätt analysmodul anlitas för de påträffade filerna.

4.3 Utveckling av komponenten FileFinder

Eftersom det resulterande systemet ska kunna analysera ett stort antal lösningar automatiskt krävs något som först söker igenom filsystemet för att lokalisera samtliga solution-filer.

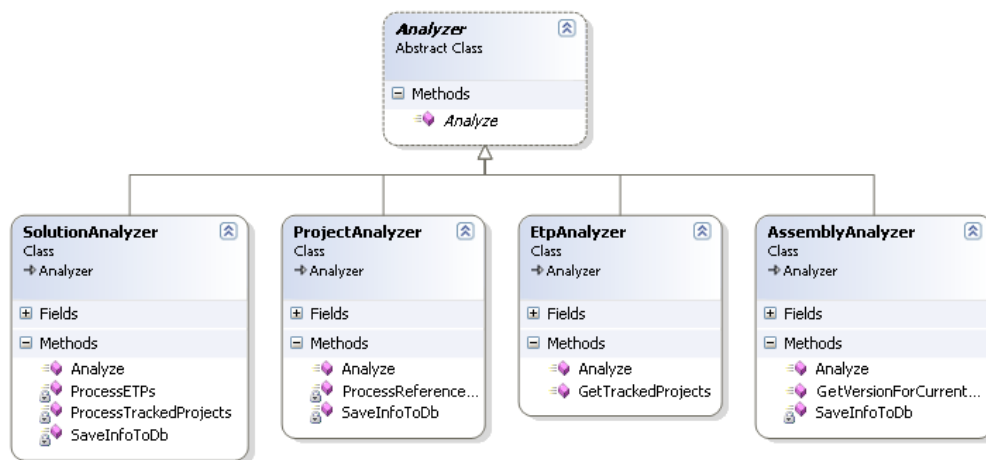
För att lösa detta delproblem skapades komponenten *FileFinder*, vilken söker efter en viss typ av fil med startpunkt i angiven rotkatalog. Denna modul söker igenom angiven katalog, varefter alla dess underkataloger genomsöks på samma sätt med hjälp av rekursiva anrop. Sökvägen till de funna filerna lagras i en dynamisk array. I detta specifika fall är det solution-filer (.sln) som eftersöks, men komponenten kan även användas för att söka efter specifika filer (filnamn inkl. filtyp), eller filer av godtycklig filtyp. I samband med att komponenten *FileFinder* anropas anges vilken fil/filtyp som ska eftersökas, samt från vilken katalog sökningen skall utgå.

4.4 Utveckling av komponenten SystemAnalyzer

Komponenten *SystemAnalyzer* är den centrala enheten i min lösning. Det är här de olika filtyperna analyseras, varefter den resulterande informationen lagras i en relationsdatabas.

4.4.1 Utveckling av basklassen Analyzer

SystemAnalyzer består av ett flertal olika klasser, däribland basklassen *Analyzer*. Detta är en abstrakt basklass som endast definierar metoden *Analyze*. Som argument tar metoden *Analyze* ett modulobjekt. Detta objekt innehar sökvägen till den fil som ska analyseras. Figuren nedan (figur 8) visar strukturen för hur den abstrakta basklassen *Analyzer* och dess subclasser är implementerade.



Figur 8. Klassdiagram som ger en översikt av analysmodulerna och dess struktur.

4.4.2 Utveckling av klassen SolutionAnalyzer

Klassen *SolutionAnalyzer* är en subclass till den abstrakta klassen *Analyzer*. Som namnet antyder är dess uppgift att analysera solution-filer. Genom att parse innehållet i en solution-fil utvinns önskad information genom att söka efter specifika taggar/medlemmar. *SolutionAnalyzer* läser dels in global information som gäller för hela lösningen, såsom id för lösningen, den version av Visual Studio som använts, samt vilket mjukvaruhanteringssystem som använts. Sedan inhämtas information om de projekt som utgör lösningen. Information såsom GUID, namn, relativ sökväg samt dess typ-GUID läses in för varje listat projekt.

När filen är analyserad lagras informationen i en relationsdatabas, *SystemInfoDb*, vilken beskrivs närmare i sektion 4.5. Här laddas global information för lösningen, grundläggande information för dess projekt, samt relationerna mellan lösningen och dess projekt in till databasen. Innan någon information laddas till databasen sker dock en kontroll om den analyserade filen är ändrad sedan en eventuellt tidigare utförd systemanalys. Skulle filen vara oförändrad sker ingen informationsöverföring till databasen.

Nästa steg blir att analysera varje funnet projekt, vilket sker i klassen *ProjectAnalyzer* (se sektion 4.4.3).

4.4.3 Utveckling av klassen ProjectAnalyzer

Klassen *ProjectAnalyzer* är en annan ärvande klass till den abstrakta basklassen *Analyzer*. Här analyseras projektfiler, vilket också i detta fall sker genom parsning av filerna. Generell information om projektet sovras, såsom typ av projekt (webbkomponent eller klientkomponent), samt vilken version av Visual Studio med vilket projektet är skapat. Även information om projektets assemblynamn och outputtyp (DLL eller EXE) erhålls, det vill säga namnet och typen för den komponent/applikation, vilken är produkten efter att projektet kompilerats.

När ovanstående är utfört inhämtas information för dess referenser. Vilken information som söks för varje referens beror på om det handlar om en referens till en assembly, eller en referens till ett annat projekt. För en assembly-referens hämtas information om dess namn, samt sökvägen till assembly'n relativt projektfilens sökväg. För en projektreferens å andra sidan hämtas information om dess namn och GUID.

Efter analysen laddas utvunnen information över till databasen. Här laddas projektspecifik information in, samt grundläggande information för vardera projekt/assembly som det aktuella projektet refererar till. Här lagras också information för relationerna mellan aktuellt projekt och dess projektreferenser. Relationerna mellan projektet och dess assembly-referenser hanteras i ett senare steg (se nästa stycke). På samma sätt som för *SolutionAnalyzer* (se sektion 4.4.2) sker ingen databaskoppling om källfilen, i detta fall projektfilen, inte är ändrad sedan en eventuellt tidigare utförd systemanalys.

Samtliga assembly-referenser analyseras var och en, vilket sker med hjälp av analysmodulen *AssemblyAnalyzer* (se sektion 4.4.5). Efter att versionsnummret för en assembly-referens hämtats (via *AssemblyAnalyzer*) lagras information om relationen mellan denna assembly och dess refererande projekt. Anledningen till att detta delmoment är utbrutet ur övrig databasförbindelse för denna klass (se föregående stycke) är att versionen för en assembly-referens måste bestämmas innan denna koppling kan fullbordas. Det är ju av stort intresse att hålla reda på vilken version av en viss assembly ett projekt refererar till.

4.4.4 Utveckling av klassen EtpAnalyzer

I den här lösningen har ETP-filen bara en uppgift, och det är att "peka ut" de projektfiler som där refereras. Analysen sker med samma metod som för de ovan beskrivna analysmodulerna (se sektion 4.4.2 samt 4.4.3), det vill säga genom att texten i filerna parsas och där specifika taggar/medlemmar eftersöks.

4.4.5 Utveckling av klassen AssemblyAnalyzer

Genom att ladda in en assembly kan man programmatiskt komma åt dess metadata som finns lagrat i manifestet. Detta är vad som sker vid analys i klassen *AssemblyAnalyzer*. Här förekommer ingen parsning av text, vilket sker i de tidigare presenterade analysmodulerna (se sektion 4.4.2, 4.4.3 samt 4.4.4). Här laddas istället en assembly in, varefter ett antal metoder anropas för att hämta önskad metadata för denna. Dessa metoder finns tillgängliga i en fördefinierad klass i .NET Framework:s klassbibliotek. Det finns mycket information lagrat i manifestet. För detta

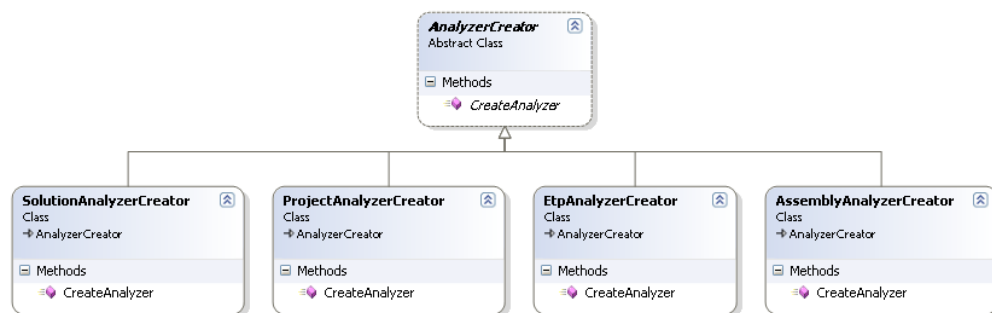
lösningsförslag erhålls versionen för aktuellt assembly, samt versionen för det .NET Framework som denna assembly skapats på.

Det kan av olika anledningar förekomma att det inte finns någon assembly lokaliserad vid angiven sökväg. I dessa fall indikeras detta i denna lösning genom att sätta versionssträngen till "N/A". På detta sätt påvisas att versionen för aktuellt assembly inte är fastställd.

Efter utförd analys laddas information om aktuellt assembly över till databasen. Ingen överföring sker dock om samma information redan finns lagrad, som ett resultat av tidigare genomförd systemanalys.

4.4.6 Utveckling av klassen AnalyzerCreator

Genom att använda designmönstret *Factory Method* skapas rätt typ av analysmodul för aktuell källa. Följande figur (figur 9) visar ett Visual Studio-genererat klassdiagram baserat på min kod.



Figur 9. Klassdiagram som illustrerar hur designmönstret *Factory Method* är implementerat.

Designmönstret *Factory Method* används för att skapa "virtuella konstruktörer" där en generell algoritm kan skapa objekt av en för den ännu odefinierad klass. Detta medför en klar fördel om man i framtiden skulle vilja utöka systemet med en ny typ av analysmodul. Exempelvis kan en ny typ av fil introduceras i en senare version av Visual Studio, vilken man är intresserad av att analysera. Tack vare tillämpningen av detta designmönster är det möjligt att tämligen enkelt utöka systemet genom att skapa en ny typ av analysmodul med tillhörande "Creator"!

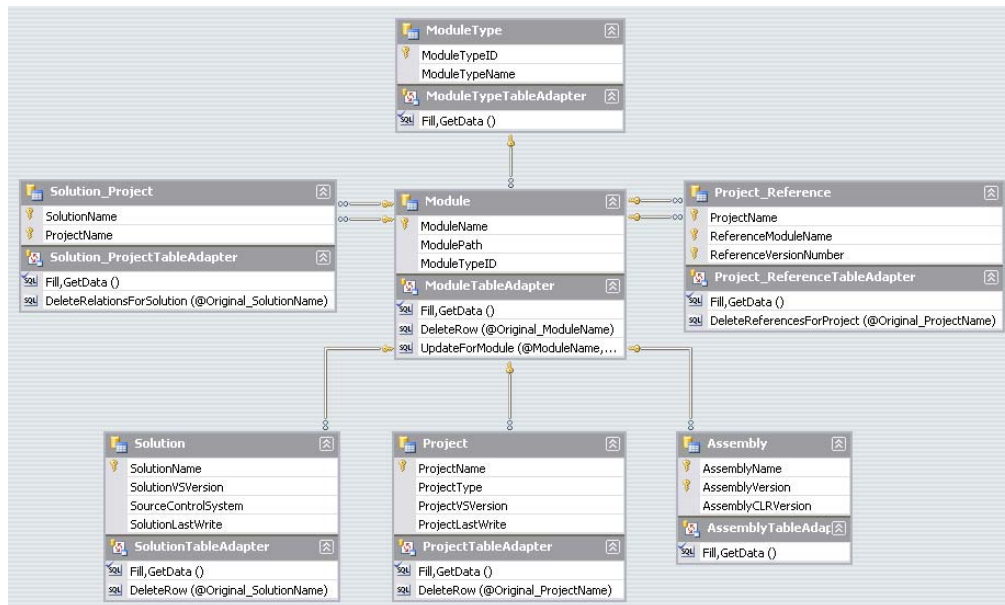
4.5 Utveckling av databasen SystemInfoDb

För att lagra informationen som *SystemAnalyzer* tar fram skapades relationsdatabasen *SystemInfoDb*, vilken är normaliserad till tredje normalform. Även databasen är designad med utvecklingsverktyget *Visual Studio 2005*. Genom att lagra resultatet för senast utförda systemanalys i en databas är det möjligt att presentera detta när som helst, utan att först behöva genomföra en ny analys.

Kopplingen mellan analysklasserna i *SystemAnalyzer* och databasen sker med hjälp av ett så kallat *DataSet*. Ett *DataSet* är en ADO.NET⁴-komponent som utgör ett utrymme

⁴ En samling komponenter för access av data och "data services" inom .NET Framework

i minnet och innehåller data inhämtat från angiven datakälla. För att kunna lagra data i, och hämta data från tabellerna i databasen via analysmodulerna skapades för detta *DataSet* så kallade *TableAdapter*:s för var och en av tabellerna i databasen. Via en specifik *TableAdapter* är det möjligt att direkt från applikationen ladda in data i den aktuella tabellen. Följande figur (figur 10) visar hur databasen, *SystemInfoDb* är uppbyggd. Här kan man också se att varje tabell har varsin *TableAdapter* kopplad till sig; exempelvis *ModuleTableAdapter* för tabellen *Module*.



Figur 10. Visar databasens tabeller och relationer (även tabellernas "TableAdapters")

Den centrala tabellen i databasen är tabellen *Module*. I denna lagras generell information för varje modulobjekt, såsom namn, sökväg, samt typ-id. Den senare är en främmande nyckel (eng. foreign key) som kopplar ihop denna tabell med tabellen *ModuleType*, där varje typ-id motsvaras av ett visst modultypsnamn. De modultyper som i denna lösning finns tillgängliga är typerna, vilka representeras av tabellerna *Solution*, *Projekt*, samt *Assembly*. Dessa tabeller är subtabeller (ärvande tabeller) till tabellen *Module*. Anledningen till att all modulinformation inte lagras direkt i tabellen *Module* är att man för de olika subtyperna vill lagra lite olika typ av information. Att samla samtliga fält från dessa subtabeller och lägga ihop dem med fälten i föräldratabellen *Module* skulle medföra många tomma fält i databasen, eftersom de flesta fält bara gäller för en viss modultyp. Exempelvis lagras information om projekttyp (*ProjectType*) endast för *Project*-instanser, och därmed skulle detta fält stå tomt för samtliga *Solution*- och *Assembly*-instanser. En annan fördel med denna indelning är att det medför bra stöd för utökning med framtida/nya modultyper.

I tabellerna *Solution* och *Project* finns ett fält med namnet *SolutionLastWrite* respektive *ProjectLastWrite*. Dessa fält lagrar datum/tid för senaste ändring av analyserad fil. Denna information används i komponenten *SystemAnalyzer* (sektion 4.4) för att fastställa om den analyserade filen (projekt- eller solution-fil) är modifierad sedan tidigare utförda systemanalyser. Skulle den inte vara det sker ingen uppdatering av informationen i databasen.

I tabellen *Solution_Project* lagras information om vilka projekt som ingår i olika lösningar (solutions). En lösning kan bestå av flera projekt, samtidigt som ett projekt kan finnas med i flera lösningar, varför denna relationstabell måste finnas.

Relationerna mellan ett projekt och dess referensmoduler (projekt eller assembly) lagras i relationstabellen *Project_Reference*. Denna tabell har bland annat ett fält med namnet *ReferenceVersionNumber*, vilket måste finnas med för att man ska kunna veta vilken version av en viss assembly-referens ett projekt refererar till. Eftersom en referensmodul även kan vara ett projekt, vilket saknar version markeras detta i tabellen genom att sätta dess version till "P". Ett alternativ till denna lösning vore att ersätta denna tabell med två separata tabeller anpassade till varsitt referensobjekt; en tabell för assembly-referenser, samt en annan för projektreferenser (utan versionsfält).

4.6 Utveckling av webbapplikationen SystemViewer

Ett lämpligt sätt att visualisera resultatet skulle vara att presentera det i en webbapplikation. Presentationsskiktets enda koppling till resterande system är dess koppling till databasen (*SystemInfoDb*). För denna uppgift skapades webb-applikationen *SystemViewer*. Denna applikation kommunicerar med/hämtar data från databasen med hjälp av en så kallad *ConnectionString*⁵. Resultatet visualiseras med hjälp av olika presentations-vyer, vilka är kopplade till varsin datakälla. En datakälla kan till exempel vara en tabell i en databas, eller en databas-vy som är skapad utifrån en, två eller flera tabeller.

Figuren nedan (figur 11) visar hur en presentation av insamlad information kan se ut i webbapplikationen.

	<u>Assembly name</u>	<u>Assembly version</u>
Select	Sandvik.Security.SecureValue.ValueAccesser.dll	1.0.1648.17564
Select	Sandvik.Web.Services.cTalk.dll	1.1.1.0
Select	SqlDataLib.dll	1.7.9.0
Select	TerminologyUpdater.dll	1.1.2.0
		1 2 3

<u>Project name</u>	<u>Type</u>	<u>Developed in</u>
GSL.Data	Client component	Visual Studio .NET 2003
Productivity.Analyzer.Data.ExternalProvider XX	Client component	Visual Studio .NET 2003

Figur 11. Visar en vy ur webbapplikationen.

I detta exempel presenteras tredjepartskomponenterna i den övre presentations-vyn, och de projekt/komponenter som använder sig av vald (markerad) tredjeparts-komponent i den nedre vyn.

⁵ Textsträng med anslutningsinformation till en specifik databas

5 Resultat av genomförande

Lösningförslaget har resulterat i ett verktyg, vilket identifierar system och dess applikationer/komponenter, samt spårar dess beroenden. En analys genomförs för de system som anträffas i angivet filträd genom att man för programmet anger filträdets rot. Det är också möjligt att genomföra analysen på flera del-filträd genom att ange sökvägar till dessa (se sektion 4.3).

Analysen sker dels genom parsning av solution-, ETP- samt projektfiler, och dels genom att ladda in assembly'n, varefter metadata programmatiskt inhämtas (se sektion 4.4).

Det av verktyget genererade analysresultatet lagras i en relationsdatabas. En webbapplikation kopplas till databasen där den hämtar resultatet, varefter detta presenteras för användaren (se sektion 4.5 samt 4.6).

6 Diskussion

Här följer en beskrivning av problem som uppkommit under implementeringen av lösningsmodellen, samt förslag på vidareutveckling av denna.

6.1 Diskussion av problem under implementationsfasen

Under implementationsfasen stötte jag på ett par oförutsedda problem. Här kommer en beskrivning av ett av de mer betydande problemen.

6.1.1 Fler än ett exemplar av en solution-fil

För en specifik lösning kunde det förekomma fler än en solution-fil lagrad i filträdet. I vissa fall handlade det om identiska kopior, vilka fanns lokaliserade på olika platser/i olika kataloger. Eftersom sökvägarna till projektfilerna, vilka anges i solution-filen är relativa solution-filens sökväg skapar detta ett problem. Skulle fel solution-fil väljas anges felaktiga sökvägar, vilket medför att projektfilerna inte lokaliseras.

För att undvika att läsa in ”felaktiga” solution-filer plockades de filer som förekom fler än en gång ut, varefter de placerades i en separat lista. Därefter får ”användaren” explicit ange vilka av dessa solution-filer som ska flyttas tillbaka till den ursprungliga listan, vilken innehåller de solution-filer som sedan ska analyseras.

6.2 Förslag till vidareutveckling av lösningsmodell

Här följer några förslag på hur det resulterande systemanalyseringsverktyget skulle kunna vidareutvecklas.

6.2.1 Uppkoppling mot mjukvaruhanteringssystem

Befintlig lösningsmodell söker efter filer att analysera i ett ”vanligt” filträd. Eftersom system och komponenter under utveckling oftast finns lagrade i en databas, vilken är dedikerad ett mjukvaruhanteringssystem finns behov att utveckla ett gränssnitt mellan analyseringsverktyget och mjukvaruhanteringssystemets databas.

6.2.2 Release-information för system

Något som i sektion 3.1 presenterats som eftersträvansvärd metadata för ett system är information om datum för senaste, samt näst senaste release. I detta läge har inte någon lösning på det problemet implementerats på grund av att tiden inte räckt till. En metod att erhålla denna information är att kommunicera med det mjukvaruhanteringssystem, vilket hanterar det aktuella systemet. Den här typen av information finns lagrad och åtkomlig i mjukvaruhanteringssystemet. I mjukvaruhanteringssystemet *Team Foundation* finns denna information lagrad i ett så kallat *ChangeSet* (se bilaga, sektion 9.1.5.1).

6.2.3 Schedulerad eller händelsekänslig start av applikation

Istället för att manuellt starta analyseringsapplikationen skulle det vara smidigt om systemanalysen kunde schemaläggas, eller ännu bättre om den kunde startas endast när uppdateringar/ändringar för ett system har registrerats. Ett steg som skulle effektivisera analysen ytterligare vore att endast scanna det/de system, för vilka ändringar registrerats. En idé för att åstadkomma detta är att låta mjukvaruhanterings-systemet signalera analysverktyget när ändringar för ett system är registrerat, samt att då också ange sökvägen till detta. Detta är möjligt i till exempel mjukvaruhanterings-systemet *Team Foundation* (se bilaga, sid 29). Vid registrering av en ändring kan man låta ett script exekveras. I detta script skulle man kunna starta konsolapplikationen *SystemScanner* (se sektion 4.2) och ange sökvägen till ändrat system som argument. Denna sökväg finns lagrad i det incheckade objektets *ChangeSet*.

6.2.4 Analysmodul för VB6-komponenter

I denna studie har system och komponenter utvecklade med olika versioner av .NET Framework behandlats. Innan .NET introducerades för mjukvaruutveckling på Windows-plattformen användes en utvecklingsteknologi med benämningen COM⁶. SSD har ett antal applikationer utvecklade i VB6 (Visual Basic 6.0), vilket är ett programspråk integrerat i utvecklingsverktyget *Visual Studio 6.0* (föregångare till *Visual Studio .NET*). VB6 är en del av COM-teknologin. För att inkludera VB6-komponenter i systemanalysen krävs att två nya analysmoduler utvecklas, vilka anpassas för dessa system. VB6-lösningar är uppbyggda enligt samma princip som de vilka redan hanterats i denna studie. Det som skiljer är filernas filändelser och dess interna struktur. Filer som motsvarar ".sln"-filer (solution-filer) har för VB6 filändelsen ".vbg". Dessa anger vilka projekt som ingår i lösningen. Projektfiler å sin sida har för VB6-projekt filändelsen ".vbp", där referenser och inkluderade filer anges, samt annan projektspecifik information.

Genom att inkludera ".vbg"-filer över sökta filtyper, samt att skapa två nya analysmoduler kan VB6-applikationer inkluderas i systemanalysen. Det som krävs är en analysmodul som parsar ".vbg"-filer, samt en annan som parsar ".vbp"-filer. Dessa analysmoduler kommer att vara likartade *SolutionAnalyzer* samt *ProjectAnalyzer* (se sektion 4.4.2 och 4.4.3), med skillnaden att dessa filer har en lite annorlunda struktur, samt att innehållets taggar/medlemmar kan ha andra namn.

Att ladda in en DLL- eller EXE-fil fungerar på samma sätt oavsett på vilken utvecklingsplattform komponenten/applikationen är skapad.

⁶ Component Object Model – föregångare till .NET Framework

7 Slutsatser

Vilken typ av information skall tas fram - när behövs den och till vad?

Vilken typ av komponent-/applikationspecifik information som är intressant att erhålla diskuterades med den person på SSD, vilken där har det övergripande ansvaret för systemutvecklingen på Windows-plattformen. Resultatet av denna diskussion, samt motiveringar till nyttan av önskad metadata presenteras i sektion 3.1 (sidan 11).

Vilka metoder kan utvecklas för att finna informationen som söks - var kan informationen hittas, samt hur kan den inhämtas automatiskt?

Olika typer av statiska metadata-filer, vilka automatiskt genereras vid mjukvaru-utveckling med verktyget Visual Studio innehåller mycket av den information som efterfrågas. Genom att parse dessa filer kan denna information erhållas. För att tillgodogöra sig dynamisk komponent-/applikationsinformation, såsom versionspecifik information används ett annat tillvägagångssätt. I fördefinierade klassbibliotek för .NET Framework finns ett antal metoder för att hämta metadata som finns lagrat i manifestet för en komponent eller applikation. Genom att ladda in en komponent/applikation kan alltså dess metadata programmatiskt utvinnas.

Hur ska information representeras?

Det är lämpligt att lagra utvunnen information efter en systemanalys i till exempel en relationsdatabas. Detta möjliggör bland annat att resultatet kan presenteras när som helst, utan att först behöva genomföra en ny analys. Inga större ombildningar sker normalt sett för de system som är under utveckling; grundstrukturen förblir ofta densamma, varefter uppdateringar genomförs. Efter en första heltäckande systemanalys behöver därefter endast fortlöpande uppdateringar och diverse tillägg laddas in i databasen.

Hur kan informationen kommuniceras?

Att presentera resultatet via en webbapplikation medför att informationen blir lättillgänglig för användarna. Om man istället skulle presentera resultatet via exempelvis en klientapplikation, krävs att användaren har denna applikation tillgänglig. Detta är inte nödvändigt för en webbapplikation, vilken finns lokaliserad centralt på en server. Presentationsskiktet kan kopplas till den databas som innehar den information som ska presenteras. På detta sätt separeras presentation och analys, vilket underlättar underhåll såsom uppdateringar och ändringar av systemet.

8 Referenser

- [1] Larsson, M. & Crnkovic, I. *Configuration Management for Component-Based Systems*, <http://www.mrtc.mdh.se/publications/0295.pdf> (2007-03-15)
- [2] Larsson, M. & Crnkovic, I. *New Challenges for Configuration Management*, <http://www.idt.mdh.se/~icc/articles/1999-Toulouse/CMChallenges-submitted.pdf> (2007-03-26)
- [3] *Microsoft .NET*, <http://www.microsoft.com/net/> (2007-03-13)
- [4] *.NET Framework*, http://en.wikipedia.org/wiki/.NET_Framework (2007-03-13)
- [5] <http://www.gotdotnet.com/team/changeinfo/default.aspx> (2007-03-13)
- [6] *Application configuration files*, [http://msdn2.microsoft.com/en-us/library/ms229689\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms229689(VS.80).aspx) (2007-04-19)
- [7] *ASP.NET Configuration Settings*, [http://msdn2.microsoft.com/en-us/library/b5ysx397\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/b5ysx397(VS.80).aspx) (07-04-24)
- [8] *make*, <http://en.wikipedia.org/wiki/Make> (07-04-19)
- [9] *NAnt: A .NET Build Tool*, <http://nant.sourceforge.net/> (07-04-19)
- [10] *Apache Ant*, http://en.wikipedia.org/wiki/Apache_Ant (07-04-19)
- [11] *MSBuild*, <http://en.wikipedia.org/wiki/MSBuild> (07-04-19)
- [12] *Assembly Manifest*, [http://msdn2.microsoft.com/en-us/library/1w45z383\(vs.80\).aspx](http://msdn2.microsoft.com/en-us/library/1w45z383(vs.80).aspx) (07-04-23)
- [13] *Component-based software engineering*, http://en.wikipedia.org/wiki/Component-based_software_engineering (07-04-26)
- [14] *NDepend – Understand and control the architecture of your .NET applications*, <http://www.ndepend.com/> (07-04-26)
- [15] *Ndepend – Getting started*, <http://www.ndepend.com/GettingStarted.aspx> (07-04-26)
- [16] *Dependency Walker 2.2*, <http://www.dependencywalker.com/> (07-04-26)
- [17] *.NET assembly*, http://en.wikipedia.org/wiki/.NET_assembly (07-05-21)

9 Bilaga: Ett jämförande analys av två mjukvaruhanteringssystem

Som en deluppgift given av SSD skulle en jämförande studie av två mjukvaruhanteringssystem, båda från Microsoft genomföras. Eftersom detta arbete inte direkt ansluter till mitt huvudprojekt har jag valt att placera denna undersökning som en bilaga.

9.1 Visual SourceSafe vs. Team Foundation

Visual SourceSafe och *Team Foundation* är två olika versionshanteringssystem för professionell mjukvaruutveckling. I denna rapport presenteras några viktiga skillnader mellan dessa två system gällande mjukvaruhantering (eng. source control).

Båda systemen möjliggör följande grundläggande uppgifter:

- Utveckla mer än en version av en produkt samtidigt
- Ändra en släppt version av en produkt utan att påverka andra versioner
- Snabbt återkalla en samling relaterade filer
- Fastställa vem som gjort en ändring och när
- Jämföra modifieringar av en fil
- Flytta ändringar från en version till en annan

Applikationer i *SourceSafe 2005* kan enkelt migreras till *Team Foundation Server* (kärnan i *Team System*). Båda systemen erbjuder dels en kommandorads-klient, och dels integrering i *Visual Studio 2005*. *Team Foundation* har inget separat användarinterface, vilket *SourceSafe* har.

9.1.1 Skillnader i arkitektur

SourceSafe Explorer och den plug-in som finns till *Visual Studio* läser från och skriver till en *Visual SourceSafe*-databas, vilket är en samling filer som vanligtvis lagras i en delad nätverkskatalog.

Team Foundation är ett "client-server"-system som använder sig av *.NET Web service* för att komma åt objekt som finns lagrade i en *SQL Server*-databas. Denna arkitektur erbjuder förbättrad prestanda och tillförlitlighet.

9.1.2 Säkerhet och Projekträttigheter

Användarrättigheter och tilldelningar i *Visual SourceSafe* som anges i dess administrationsprogram är fristående från delningsrättigheterna i *Windows* för *Visual SourceSafe*'s databaskatalog. Rättigheter och tilldelningar kan sättas för specifika *SourceSafe*-projekt eller individuella *SourceSafe*-användare, men alla *SourceSafe*-användare måste ha samma rättigheter till databaskatalogen. Detta medför att alla användare, oavsett deras projekträttigheter satta via administrationsprogrammet i *SourceSafe* har tillgång till hela *SourceSafe*-databasen.

I *Team Foundation* binds användarspecifika operationsrättigheter och projektnivårättigheter till *Windows* användarkonton. Säker verifiering av användarna

utförs med hjälp av Internet Information Services (IIS). Ingen separat åtkomst till SQL Server-databasen är nödvändig för individuella användare.

9.1.3 Pålitlighet

Eftersom *Visual SourceSafe* inte har någon serverkomponent är operationer som flyttar data från klienten till databasen "non-transactional". Med detta menas att en transaktion inte kan återgå till en tidigare nivå (eng. roll back) om ett problem uppstår (t.ex. avbrott i nätverksanslutning). Om ett problem skulle uppstå under en skrivoperation kan integriteten för den påverkade filen bli åsidosatt och information gå förlorad.

Team Foundation är en "client-server"-applikation där skrivoperationer i databasen utförs med hjälp av lagrade procedurer som inte utsätts för anslutningsproblem i nätverket. Dessutom genomförs speciella operationer genom att använda transaktioner, så att man kan återgå till en tidigare nivå vid ett eventuellt fel.

9.1.4 Skalbarhet

Team Foundation har stöd för arbetsgrupper med upp till 2000 användare, medan *SourceSafe* rekommenderas till arbetsgrupper med upp till 20 användare. Servrar för *Team Foundation* kan innehålla så mycket data som SQL Server tillåter, och din hårdvara kan hantera. En *SourceSafe*-databas å sin sida har en rekommenderad övre storleksbegränsning på 4 GB.

9.1.5 Funktionella skillnader

Den interna strukturen ser i princip ut på samma sätt för en *SourceSafe*-databas som för en *Team Foundation*-server. Både databaser och servrar har en hierarkisk struktur. Kataloger innehåller filer, och filer består av versioner som identifieras av nummer och datum/tid för skapandet.

9.1.5.1 Changesets

Team Foundation inför ett nytt koncept som inte finns i *Visual SourceSafe*, nämligen ett s.k. "changeset". Ett "changeset" är en logisk behållare där man i *Team Foundation* lagrar allt som relaterar till en enskild incheckningsoperation: ombearbetningar för filer och kataloger, länkar till relaterade objekt, incheckningsnoteringar, incheckningskommentar, samt övrig information som t.ex. vem som gjort ändringarna.

9.1.5.2 Sharing & Pinning

Team Foundation saknar ett likvärdigt kommando till "Share" och "Pin" som finns i *SourceSafe*. Vid migrering av *SourceSafe*-projekt till *Team Foundation*-projekt ersätts en "Pin" från en *SourceSafe*-databas med en label.

9.1.5.3 Skillnader i lagring av historik

Visual SourceSafe och *Team Foundation* lagrar historik på olika sätt på några punkter. Här följer några exempel:

Add & Create: När man i *Visual SourceSafe* lägger till en fil eller skapar en katalog

skapas utöver filen i sig även en version av föräldern. I historiken för föräldern lagras händelsen som "add", och i filens historik lagras händelsen som "create". För *Team Foundation* å andra sidan skapas endast en version av filen eller katalogen i sig när man lägger till en fil/katalog. Händelsen lagras då som "add" - här skapas alltså ingen version av föräldern.

Rename, Delete & Undelete: I *Visual SourceSafe* skapar dessa händelser en ny version av föräldern, medan det i *Team Foundation* skapas en ny version av objektet i sig.

Move: När man i *Visual SourceSafe* flyttar en katalog skapas nya versioner för föräldrarnas kataloger i både käll- och målkatalogen. Händelser lagras för varje förälder som man flyttar katalogen till eller från. Det skapas ingen version av katalogen K. Ett exempel: Om du flyttar katalogen K från källan S till målet D, skapas en ny version av D genom händelsen "Move \$K from S", och en ny version av S skapas genom händelsen "Moved \$K To D". I *Team Foundation* skapas endast en ny version av katalogen genom händelsen "Rename".

9.1.5.4 Skillnader vid in- och utcheckning av filer

I *Visual SourceSafe* måste man checka ut en fil explicit, men man checkar bara in filen om några ändringar har gjorts. I *Team Foundation* däremot finns ett krav att man alltid både checkar ut och checkar in en fil explicit.

I *Visual SourceSafe* utförs en "Get"-operation när en fil checkas ut. Så fungerar det inte i *Team Foundation*, där flera användare kan checka ut och ändra i ett och samma objekt samtidigt. I *SourceSafe* har den användare som checkar ut ett objekt som default ensamrätt till objektet. I *Team Foundation* kan man dock låsa en fil, så att övriga inte kan checka ut den, eller checka in ändringar.

9.1.5.5 Förgrening och sammanslagning

Visual SourceSafe har ett väldigt enkelt stöd för förgrening (eng. branch) och sammanslagning (eng. merge), eftersom ingen historia lagras om sammanslagningar mellan två förgrenade filer eller kataloger. *Team Foundation* å sin sida, erbjuder stöd för lagring av historik för sammanslagningar. Utan denna sammanslagningshistorik sker i *Visual SourceSafe* grundlösa sammanslagningar.

Funktionalitet i *Team Foundation* som saknas i *SourceSafe*:

- Workspaces⁷
- Changesets
- Shelvesets
- Team Foundation Work Items
- Check-in Policies
- Check-in Notes
- E-mail notifications by Setting Alerts

Funktionalitet i *SourceSafe* som saknas i *Team Foundation*:

- Share
- Pin

⁷ *Visual SourceSafe* har arbetskataloger

- Archive and Restore
- Destroy
- Keyword expansion
- Rollback⁸

Referens:

[http://msdn2.microsoft.com/en-us/library/ms181369\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms181369(VS.80).aspx) (2007-04-11)

⁸ Verktöget *Power Toy*, som finns för *Team Foundation* innehåller en "roll-back"-funktion