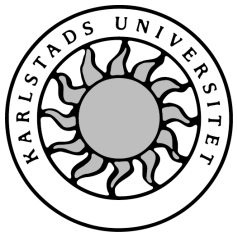


Datavetenskap

Karl Larsaeus och Pär Råstam

Utveckling av en grafisk databasapplikation i .NET

Examensarbete

2004:25

Utveckling av en grafisk databasapplikation i .NET

Karl Larsaeus och Pär Råstam

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 vårt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Karl Larsaeus

Pär Råstam

Godkänd, 2004-06-03

Handledare: Robin Staxhammar

Examinator: Donald Ross

Sammanfattning

Bakgrunden till denna rapport är att företaget Midroc Electro vill undersöka hur det är att utveckla ett grafiskt användargränssnitt i .NET för att administrera en databas. Vår uppgift har varit att enligt deras utvecklingsmodell skapa en prototypapplikation. Målet med denna prototyp var att utforma dess gränsnitt så att det utgör en förbättring av en befintlig applikation, då främst med avseende på användbarhet (usability). Då .NET-plattformen erbjuder nya tekniker för utveckling av gränssnitt och distribution av klienter önskade Midroc Electro en inledande studie för att undersöka möjligheterna dessa tekniker erbjuder och motivera användandet av en specifik teknik i vår prototyp. Delar av denna studie ingår i rapporten.

Resultatet av arbetet är en prototypapplikation med ett gränssnitt som vi anser är översiktligt och med potential till ett större antal databasinmatningar per minut än det befintliga gränssnittet.

Development of a graphical database application in .NET

The company, Midroc Electro, wants to examine the new possibilities for developing a graphical user interface in .NET. The user interface shall be used to administer a database. Our task has been to create an application prototype in accordance to the development model used by Midroc Electro.

The goal for this prototype was to design its user interface so that it would offer an improvement in usability to an existing application.

Because the .NET-platform offers new technologies for creating user interfaces and for client distribution, Midroc Electro wanted an initial study to be made, investigating the new possibilities. This study motivates the use of a specific technology in our prototype. Parts of this study are included in this report.

The result of our work is an application prototype with a user interface which we feel gives a good overview of the database, and which has the potential to increase the number of feeds of database entries per minute compared to the user interface of the existing application.

Innehåll

Figurer

1 Introduktion

1.1 Bakgrund

Midroc Electro i Karlstad arbetar med industriell IT, framförallt med datorbaserade lösningar för produktions- och labbmiljö, de utvecklar i huvudsak programvara för produktionsadministrativa system samt mät- och testsystem.
I en befintlig applikation som Midroc har utvecklat finns ett gränssnitt för att administrera en databas. Denna databas innehåller fyra objekttyper; ett huvudobjekt samt tre underordnade objekt. Applikationen används till att skapa kopplingar mellan huvudobjekten och de underordnade objekten, att lägga till nya objekt, samt att editera och ta bort befintliga objekt. Detta gränssnitt är för närvarande imlementerat som en webbapplikation och beskrivs närmare i kapitel två. Då nuvarande gränsnitt stundtals upplevs som tungarbetat och omständigt att utföra vissa arbetsmoment i vill Midroc undersöka vilka möjligheter till förbättring en applikation utvecklad i Microsofts .NET- plattform erbjuder. Främst är man intresserad av möjligheterna att skapa ett användarvänligare och mer lättarbetat gränssnitt men även möjligheterna med att använda flera olika programspråk i en och samma implementation.

1.2 Mål

Midroc Electro skall byta från Microsoft Visual Studio 6.0 som utvecklingsplattform till Microsoft Visual Studio .NET. Vår uppgift är att utforma en prototyp till ett nytt databasgränssnitt utvecklat i Visual Studio .NET och samtidigt undersöka vissa nyheter som .NET-plattformen erbjuder, som till exempel att kombinera flera programspråk i samma utvecklingsprojekt vilket behandlas i kapitel 5 och nya tekniker för distribuering av applikationer vilket tas upp i kapitel 3.

Vår färdiga prototyp skall uppfylla följande krav:

- Den skall innehålla ett lättarbetat och användarvänligt användargränssnitt som underlättar arbetet mot databasen och ökar antalet inmatningar per minut.
- Det skall finnas möjlighet att anpassa användargränssnittets språk till användarens nationalitet
- Då systemet har flera samtidiga användare måste stöd för transaktioner finnas.
- Alla förändringar i databasen måste signeras av användaren som utfört dessa.

Midroc har gjort klart att det är viktigt att vi följer företagets policy vad gäller utveckling, testning och granskning. Enligt Midrocs utvecklingsmall skall därför följande dokument utformas:

- En förstudie som är en del av en inledande analys för att identifiera vilka risker och möjligheter olika teknikval medför
- En funktionsspecifikation där man beskriver systemets tänkta funktionalitet för att fastställa att de krav som angetts i kravspecifikationen uppfylls
- En designspecifikation som i detalj beskriver hur systemet skall implementeras
- En användarmanual
- Installationsinstruktioner

De tester som skall ha utförts är Kodgranskning, Factory Acceptance Testing (FAT) och Site Acceptance Testing (SAT). Närmare beskrivning av testerna och deras resultat beskrivs i kapitel 6.

1.3 Anmärkning angående sekretess

Då innehållet i databasen är av känslig natur har vi skrivit på ett sekretessavtal med Midroc Electro där vi förbinder oss att inte lämna ut viss information. Detta innebär att

skärmbilder kommer få visst innehåll bortretuscherat och när vi talar om de olika objekten som databasen hanterar kommer dessa nämnas som ”objekttyp ett”, ”huvudobjektet” o.s.v.

1.4 Disposition

Rapportens åtta kapitel är uppdelade i tre stycken delar. Första delen som består av kapitel två och tre fungerar som en introduktion där kapitel två beskriver uppgiften vi skall lösa och kapitel tre ger en kort beskrivning av olika tekniker i .NET som läsaren kan ha nytta då hon[1] läser uppsatsen. Del två består av kapitel fyra och fem. I denna del beskriver vi den tänkta funktionaliteten hos vårat gränssnitt samt hur de olika komponenterna som bygger upp vår implementation fungerar.
I den sista delen av uppsatsen beskriver vi de resultat som uppnåtts och redogör de lärdomar och slutsatser vi gjort om hur det är att utveckla ett grafiskt användargränssnitt i .NET

[1]Där hon eller han uppträder i texten skall detta läsas som ”han / hon”

2 Den befintliga applikationen

Eftersom arbetet går ut på att implementera en prototyp som är tänkt att fungera som en förbättring av en befintlig applikation beskriver vi här hur den befintliga applikationen är uppbyggd. Vi väljer att fokusera på det vi upplever som brister i den befintliga applikationen, och presenterar därefter ett förslag på hur vi planerar att avhjälpa dessa brister i vår prototyp. Hur vi realiserar dessa förbättringar beskrivs i kapitel 4 och 5.

2.1 Befintlig applikation

Den befintliga applikationen är en webbapplikation där man har de fyra objekttyperna i en meny till vänster och ett huvudfönster där man får upp information om det aktuella objektet. Detta gör att man bara kan se en typ av objekt åt gången. För att komma åt olika objekt måste man navigera fram och tillbaka mellan de olika objektens vyer. Detta medför att det blir svårt att få en översikt av den data man arbetar med.
Varje gång en förändring görs i databasen, t.ex. en koppling mellan olika objekt eller en förändring av ett objekts egenskaper, måste en signering göras. Detta medför att mycket tid går åt till att signera förändringar.

2.2 Förslag på förbättringar

Vi vill använda oss av en trädstruktur där varje nod i trädet motsvarar ett objekt i databasen och trädets struktur visar relationerna mellan de olika objekten. Genom att välja en nod i trädet får man upp information om objektet som representeras av noden. Användaren skall på ett enkelt sätt kunna göra kopplingar mellan objekt genom att dubbelklicka på det objekt som skall bindas samman med den markerade noden ur en lista med tillgängliga objekt.
I vår prototyp kan användaren göra flera ändringar i en lokal representation av databasen, för att sedan göra en gemensam signering av dessa förändringar då de skrivs till databasen.

Detta ger möjlighet att öka antalet förändringar i databasen per minut. Genom att använda oss av ”No Touch Deployment” (se kap 3.3.2) så slipper vi de problem med distribution som en fristående klient medför men kan ändå utnyttja möjligheterna att skapa en ”rikare” (se kap. 3.2) applikation.

2.3 Kravsammanställning

För att implementera de förbättringar som nämnts ovan och kunna uppfylla de mål som satts upp i det inledande kapitlet skall vår prototyp uppfylla följande krav.

- Anpassning av systemets utseende och funktionalitet beroende på vilken användarbehörighet användaren som loggar in har. T.ex. så visas inte alla menyval för användare med lägre behörighet. Denna funktionalitet behövs för att uppfylla kravet om olika användarnivåer
- Automatiskt utloggning av en inaktiv användare efter ett av administratören fördefinierat tidsintervall. En användare betraktas som inaktiv då viss tid förflutit sedan hans senaste signering.
- När en användare utför ändringar i gränssnittets databasrepresentation så loggas dessa förändringar av systemet. När användaren väljer att skriva ändringarna till databasen får han upp en sammanställning över de gjorda förändringarna. Användaren kan där välja vilka av dessa som skall skrivas till databasen och ange anledning till detta. När dessa val är gjorda måste användaren signera förändringarna för att de skall skrivas till databasen.
- Systemet försöker skriva förändringar till databasen då en lyckad signering har genomförts. Systemet hämtar då först en aktuell bild av databasen och jämför den med en lokal bild för att kontrollera att ingen av de data som skall skrivas har blivit förändrad. De förändringar som tillåts skrivs till databasen och förändringar som misslyckats meddelas användaren.

3 Tekniker för gränssnitt i .NET

I detta kapitel undersöks vilka möjligheter som erbjuds i .NET för att utveckla en grafisk klient till en databas. Innehållet baseras på en förundersökning som gjordes i det inledande skedet av arbetet, för att Midroc skulle kunna fatta ett beslut om vilken lösning de ville ha. Denna lösning beskrivs mer ingående i kaptiel 4.
Ett av målen för arbetet är att utreda på vilka sätt en applikation kan distribueras över ett nätverk, även detta kommer behandlas i detta kapitel.
I .NET finns Windows Forms, ett klassbibliotek för utveckling av grafiska gränssnitt. För att distribuera klienter med gränssnitt baserade på Window Forms introducerar .Net två nya tekniker kallade IE Sourcing och No Touch Deployment [1].

3.1 Terminologi

Här presenterar vi begrepp och tekniker som man bör förstå innebörden av för att tillgodogöra sig den fortsatta diskussionen kring de nya möjligheterna som erbjuds i .NET.

3.1.1 Assemblies

Ett assembly är en samling datatyper och resurser betraktad som en logisk enhet. Assemblies utgör grundstenarna i .NET-ramverket och deras egenskaper och struktur erbjuder funktioner som versionshantering, återanvändning av programkod, enkel distribution av applikationer och säkerhet genom rättigheter. Ett assembly kan innehålla en eller flera kodmoduler och resurser, och därmed sträcka sig över en eller flera filer. De datatyper och resurser som ett assembly innehåller beskrivs av metadata, information som själv är en del av assemblyt. Ett assembly säges därför vara självbeskrivande. Ett assembly kan vara privat, alltså endast avsedd för en applikation, eller delat, där flera applikationer kan använda sig av assemblyts innehåll. Ett delat assembly måste ha något som kallas ”starkt” namn.

Ett assembly identifieras fullständigt av ett namn, ett versionsnummer, en publik nyckel och en kulturangivelse. Assemblyts namn är dess filnamn med filändelsen borttagen och versionsnumret är en siffersträng som innehåller information om assemblyts version och revision. Kulturangivelsen är en tvåteckens nationskod och anger om assemblyt har specifika nationella egenskaper, t.ex. ett visst språk. Den publika nyckeln är 128 bytes lång och används för att kontrollera giltigheten av en digital signering av assemblyt. Signering är en kryptografisk process som säkerställer dels identiteten hos upphovsmannen och dels integriteten hos assemblyts innehåll. Signering av ett assembly sker vid länkningen, då upphovsmannens privata nyckel används. Ett assembly med dessa fyra egenskaper angivna säges vara ”starkt” namngivet. För mer information om assemblies se [7].

3.1.2 Global Assembly Cache (GAC)

GAC är en i systemet central lagringsplats för assemblies som möjliggör global åtkomst. Delade assemblyn lagras här och de identifieras i GAC med hjälp av sitt starka namn. Assemblies installeras i GAC med verktyget gacutil som följer med .NET-ramverket. Se [6] och [12].

3.1.3 ”Dll hell”

En benämning på det tillstånd som med äldre Microsofttekniker (före .NET) kunde uppstå när flera applikationer var beroende av funktionaliteten hos ett dynamisk länkat bibliotek (DLL) vars gränssnitt plötsligt förändrades, kanske genom en uppdatering, och inte längre var kompatibel med det gamla. Detta fick följden att de beroende applikationerna började uppföra sig underligt eller slutade fungera. För mer information om ”Dll hell” se [8]

3.1.4 Säkerhetsmekanismer i .NET

Säkerhetspolicy i .NET är en uppsättning regler som anger vilka rättigheter som skall tilldelas vilka assemblies. Rättigheter representeras av permission-objekt, som kan grup-

peras i namngivna s.k. permission-sets. Säkerthetspolicyn uppdelas i fyra nivåer, som i sin tur indelas i en hierarki av kodgrupper, som utgör en logisk uppdelning med vissa angivna medlemskriterier. Den programkod som uppfyller kodgruppens medlemskriterier blir medlem. Varje kodgrupp har en associerad uppsättning permission-sets. Således bestäms ett assemblys rättigheter till systemet och dess resurser beroende på vilken kodgrupp det tillhör. För att undersöka vilka kodgruppers medlemskriterier ett assembly uppfyller samlas bevis (evidence) in om assemblyt och dess egenskaper. Det kan röra sig om assemblyts upphovsman, dess installationsbibliotek, den URL som det härrör ifrån, dess starka namn osv. För vidare information om säkerhetsmekanismer i .NET, se [5].

3.1.5 Zero Impact Deployment

Zero Impact Deployment är benämningen på det nya sättet att distribuera applikationer som .NET-ramverket möjliggör. Traditionellt innebär distribuering av applikationer att delar av den installerande applikationen införs i systemet på maskinen den installeras på, t.ex. registrering av komponenter och kopiering av filer till olika platser på filsystemet. Installationen lämnar systemet i ett annat tillstånd än det var innan. Nyheten som .NET-ramverket inför är möjligheten att installera applikationer på ett målsystem utan att förändra det på annat sätt än att skapa ett nytt bibliotek dit applikationens filer kopieras. Avinstallation av applikationen innebär endast att biblioteket och dess filer raderas.

3.1.6 Rapid Application Development (RAD)

RAD (Rapid Application Development) är en benämning på en utvecklingsmetod som tillåter utvecklaren att snabbt bygga ett fungerande program. Ett exempel på ett RAD-verktyg är Microsft Visual Studio .NETs Window Forms Designer.

3.1.7 Windows Forms

Windows Forms är Microsofts nya plattform för utveckling av grafiska applikationer i Microsoft Windows. I stora drag är det en samling klasser under .NET-plattformen som möjliggör skapandet av rika klienter. Rika klienter beskrivs mer ingående i nästföljande sektion.

3.2 Olika typer av klientapplikationer

En klient är ena parten i mjukvaruarkitekturen klient/server. Klientens roll är den att begära och få betjäning från servern. I följande stycke beskrivs två huvudsakliga egenskaper hos en klient; dels tjockleken, hur stor del av applikationslogiken som klienten innehåller och dels dess rikhetsgrad, som avser användargränssnittet utbud av kontroller och möjligheter att på olika sätt interagera med användaren. Dessa två egenskaper används sedan för att belysa skillnaderna mellan de två klienttyper som vi behandlar i den fortsatta diskussionen.

3.2.1 Tjocklek

Tunn klient En klient som endast innehåller ett användargränssnitt och där all affärslogik och övrigt funktionalitet finns på serversidan. Ett klassiskt exempel på en tunn klient är de textterminaler som används för att låta flera användare ansluta till och arbeta mot en stordator.

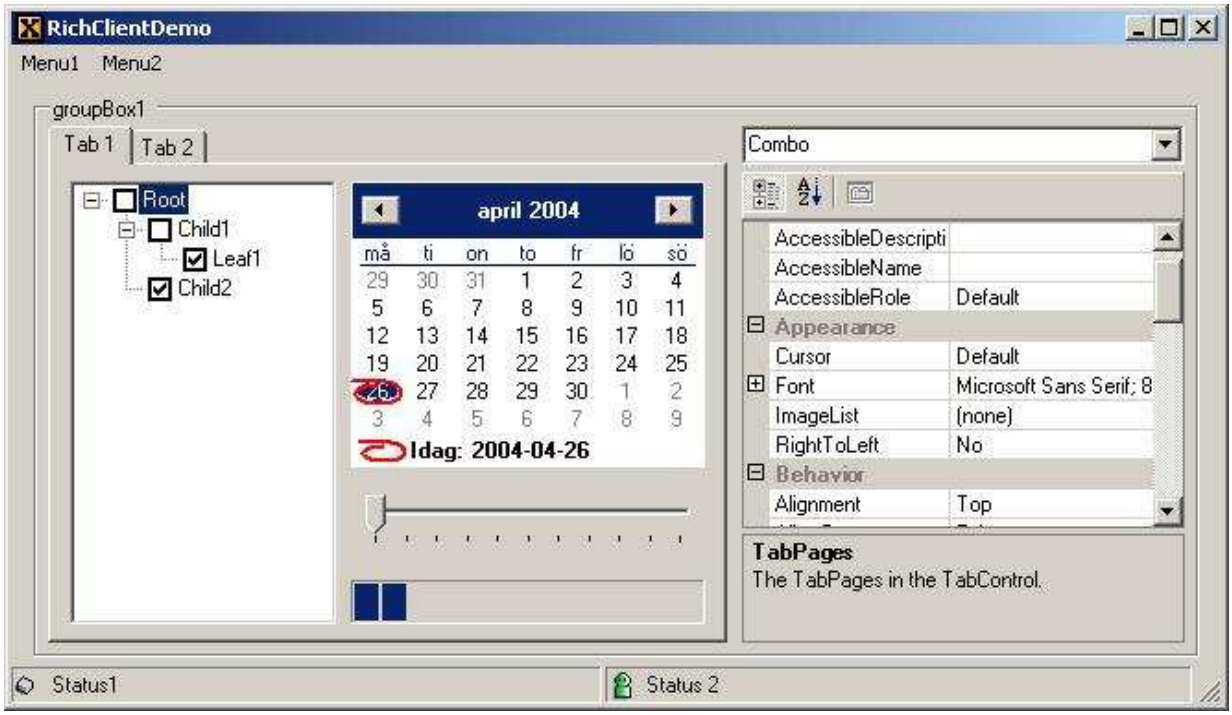
Fet klient En klient som till skillnad mot en tunn klient innehåller både gränssnitt och bakomliggande funktionalitet. Klienten utför den största delen av databearbetningen medan servern står för lagring av data.

3.2.2 Rikhetsgrad

Rik klient Ordet ”rik” i rik klient syftar på dess användargränssnitts utbud av grafiska kontroller och ska ej förväxlas med begreppet fet klient. En rik klient är en klientapplikation

vars användningssätt för användaren är välkänt från användargränssnitt som t.ex Microsoft Windows. En användare skall kunna utföra de operationer som han brukar, som t.ex. dra ett objekt mellan fönster, få upp den meny man är van vid högerklick, animationer som illustrerar en viss händelse o.s.v.

Figur 3.1 visar en påhittad applikation innehållande en mängd grafiska kontroller som kan finnas hos en rik klient.



Figur 3.1: Exempel på en klient med en rika grafiska kontroller

Torftig klient Att hävda att en klient är ”torftig” är inget vedertaget begrepp, men vi använder oss av detta i den här diskussionen för att underlätta jämförelsen mellan klienter. En torftig klient säger vi här vara motsatsen till en rik, d.v.s. den har en mycket begränsad uppsättning grafiska kontroller. Ett exempel på en sådan begränsad uppsättning grafiska kontroller är de standardkontroller för formulär som används i webbsidor.

3.2.3 Webbklient kontra fristående klient

Syftet med förstudien är att undersöka vad .NET erbjuder ifråga om tekniker som ger nya möjligheter att utveckla klientapplikationer och bidra med material till ett beslutsunderlag där Midroc kan besluta huruvida vår prototypapplikation skall implementeras som en webbklient eller en fristående klient. Härunder presenteras kortfattat vad som allmänt är karaktäristiskt för en webbklienten respektive en fristående klient.

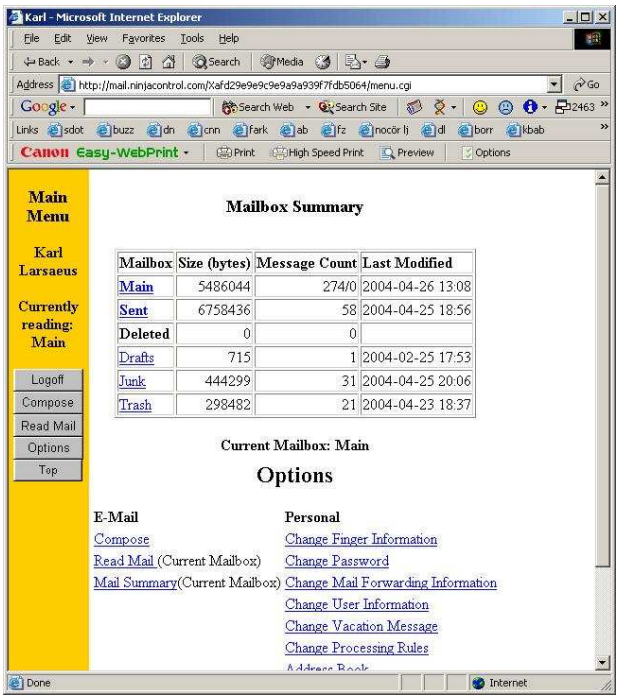
Webbklient En webbklient är klientparten i en klient-/serverlösning där gränssnittet mot användaren är en webbsida som visas i en webbläsare. Webbsidans utseende och innehåll är definierat i ett HTML-dokument som hämtas från webbservern av webbläsaren. Användaren kan ge indata via kontroller såsom knappar, textrutor, kryssrutor och rullgardinsmenyer. Inmatningar från användaren vidarebefordras av webbläsaren till servern där bearbetning av indatan sker och resultatet skickas tillbaks där det presenteras för användaren av webbläsaren. Eftersom HTML-formatet är standardiserat [10] kan en webbsida visas i vilken webbläsare som helst som implementerar standarden korrekt. Detta medför att en webbklient som är konstruerad i enlighet med HTML-standarden kan betraktas som plattformsoberoende.

I fallet med webbklienten är applikationslogiken placerad på serversidan, detta gäller även själva definitionen av gränssnittet. Ett HTML-dokument som representerar gränssnittet skickas av servern till webbläsaren på dennes begäran. I och med att dessa två delar av applikationen är belägna på servern så blir uppdateringar av dess funktionalitet och användargränssnitt mycket enkla att genomföra. Ändringar på serversidan kommer att vara synliga för användaren vid nästa tillfälle denna använder webbklienten.

De kontroller som HTML-standarden erbjuder är relativt begränsade, både till antal och funktionalitet. Vill man utöka funktionaliteten hos kontrollerna, som t.ex. validering av

inmatade värden eller hantera händelser som kontrollerna genererat krävs programmering i något skriptspråk som stöds av webbläsaren, t.ex. javascript.

Sammanfattar man webbklientens egenskaper kan man säga att den är tunn, då applikationslogiken och användargränssnittet ligger på servern, och att den i grundutförandet (utan utökad funktionalitet med scriptprogrammering) har ett relativt torftigt utbud av grafiska kontroller. Ett exempel på en typisk webbklient är webbmailtjänster som Microsofts Hotmail, eller den webbaserade epostklienten till Ipswitch IMail Server som visas i figur 3.2.



Figur 3.2: Webbklient

Fristående klient I en fristående klientapplikation ligger hela användargränssnittet och en stor del av applikationslogiken på klientsidan. Den är fristående i den mening att den inte, som webbklienten, är beroende av andra applikationer för att användas (webbklien-

ten visas i en webbläsare). Den fristående klienten har i allmänhet möjlighet att använda sig av alla de grafiska kontroller som finns tillgängliga i det underliggande operativsystemet i sitt användargränssnitt, exempelvis kontroller för trädvyer, typsnittsdialoger och kontroller anpassade för data i tabellform från databaser. Det finns också möjlighet att utöka de befintliga kontrollernas funktionalitet, t.ex. genom att skapa subklasser av de existerande.

Då en större del av applikationslogiken än i fallet med webbklienten är placerad inuti den fristående klienten är uppdateringar av applikationen mer omständiga; En förändring av servern kräver oftast en uppdatering av den fristående klienten, vilket kan innebära ett omfattande administrativt arbete om denna uppdatering rör många användare, där var och en av installationerna av den fristående klienten måste uppdateras innan hela systemet kan betraktas som uppgraderat.

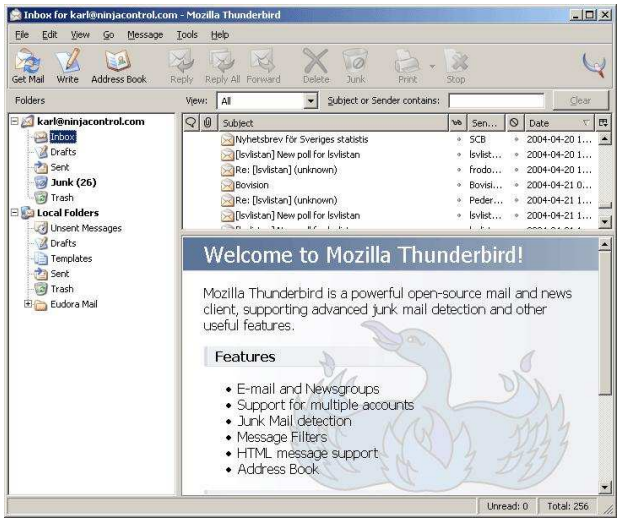
Den fristående klienten är rik, då den har tillgång till alla de grafiska kontroller som operativsystemet erbjuder och fet då den innehåller både applikationslogik och grafiskt gränssnitt. Ett exempel på en fristående klient är Mozillas epostklient Firebird som visas i figur 3.3.

3.3 Två klientlösningar i .NET

Informationen i detta delkapitel är i huvudsak tagen från [2] och [4].

3.3.1 Webbklient baserad på Windows Forms (IE Sourcing)

IE Sourcing är en benämning på en teknik där Windows Forms-kontroller, som vanligtvis används i fristående klienter, inbäddas i en webbsida som visas med Internet Explorer. Detta är tänkt som en efterträdare till Java-applets och activeX-kontroller. Tack vare att man kan använda Windows Forms kontroller så kan man få det mesta av den funktionalitet



Figur 3.3: Fristående klient

såsom drag and drop och animationer som man annars är van vid hos fristående klienter. Samtidigt behåller man de fördelar som en vanlig webbklient erbjuder.

I de fall när vanliga HTML-kontroller inte är tillräckliga används ofta activeX och Java applets. Problemet med activeX är att användaren tvingas ge activeX-kontrollen tillåtelse att exekvera och ifall man gör detta så ges kontrollen fulla rättigheter till systemet, en sorts ”allt eller inget”-policy, vilket utgör en potentiell säkerhetsrisk. Java-applets har inte detta problem men har istället begränsningar med vilka sorters applikationer som kan skapas. Den faktor som begränsar Java-applets är dess ”sandlådemodell”. Med sandlådemodell menas att appleten exekverar i en skyddad miljö, isolerad från systemet. IE Sourcing löser activeX säkerhetsproblem genom att applikationen exekveras som delvis betrodd. Detta innebär applikationen som körs inte har tillgång till filsystem, skrivare eller andra känsliga delar av klientdatorn. För tillförlitliga applikationer på finns möjligheten att höja rättigheterna för en viss assembly, från att exekevera som delvis betrodd upp till fullständigt betrodd.

3.3.2 Fristående klient baserad på Windows Forms distribuerad med No Touch Deployment (NTD)

NTD är en teknik för distribution av applikationer över nätverk. En applikation kan startas genom att användaren öppnar en URL i Internet Explorer och de delar av applikationen som behövs kommer att laddas ned till användarens maskin och exekveras. Nedladdade assemblies lagras på en speciell lagringsplats, assembly download cache, som är privat för varje användare av maskinen. I och med att en applikation startas över nätverket kommer en kontroll att göras av de lokala assemblies som finns i assembly download cache, och endast de nya eller uppgraderade laddas från webbservern.

En klientapplikation med ett gränssnitt baserat på Windows Forms och som distribueras med NTD ingår i Microsofts vision om den 'smarta' klienten. Kortfattat innebär en 'smart' klient en syntes mellan den tunna webbklienten och den rika klienten. Den tunna webbklientens fördelar med enkel distribution och underhåll erbjuds tillsammans med den rika klientens uppsättning av användargränssnittskontroller.

3.4 Utvärdering av de olika lösningarna

I och med den förenklade distributionen hos NTD och möjligheten att använda mer avancerade kontroller i IE Sourcing så har den klassiska webblösningen och den fristående klienten flutit ihop något, båda har lånat lösningar från varandra. Vi ska med avseende på utvecklingstid, användarvänlighet och underhållskostnad försöka avgöra vilken lösning som passar oss bäst.

Tack vare utbudet av utvecklingsverktyg anser vi att ur utvecklingssynpunkt är den fristående klienten att föredra. I Visual Studio .NET finns Windows Forms designer, ett RAD-verktyg för grafiska gränssnitt. Även om man kan använda RAD-verktyg vid delar av utvecklingen

av en webbklient så behövs även andra tekniker såsom scripting.
Ur användarsynpunkt har man tidigare ofta använt den fristående klienten med tanke på möjligheterna att utveckla ett användarvänligt grafiskt gränssnitt. Eftersom .NET nu har gjort det möjligt att använda mer avancerade kontroller i webbläsaren så är inte denna egenskap längre exklusiv för den fristående klienten. På grund av detta är det därför svårt att säga vilken av lösningarna som är att föredra på denna punkt. Webbklientens starkaste kort har tidigare varit enkelheten i distribution och underhåll. Därför har detta alternativ varit att föredra ur administratörsperspektiv. Detta har förändrats med .NET i och med introduktionen av assemblies och möjligheten att distribuera fristående klienter via en webbserver.
Då tyngdpunkten skall ligga på att utveckla ett användarvänligt grafiskt gränssnitt är vår slutsats att en fristående klient att föredra. Anledningen till detta är att även om det skulle gå att utveckla en webbaserad lösning med ett rikt användargränssnitt så kommer denna IE Sourcing-baserade lösning att omfatta fler tekniker, bl.a. scripting på klientsidan, html, och något av programspråken i .NET. Detta leder till att utvecklingstiden blir längre och arbetet mer omfattande än ett grafiskt gränssnitt med motsvarande rikhetsgrad utvecklat som fristående klient. Scripting och webbmiljön är ytterligare en källa till fel vilket vi gärna undviker då vi har så pass kort utvecklingstid.

4 Beskrivning av prototypens gränssnitt

I detta kapitel beskriver vi uppgiften vår prototyp skall lösa mer ingående. Vi förklarar även de olika kontroller vårt gränssnitt är uppbyggt av, hur vi kom fram till att använda just dessa kontroller och hur användaren interagerar med dessa för att manipulera databasen. När vi i kapitel 5 beskriver hur vi konstruerat vår prototyp är det lämpligt att först ha läst detta kapitel för att ha skapat sig en bild av applikationen.

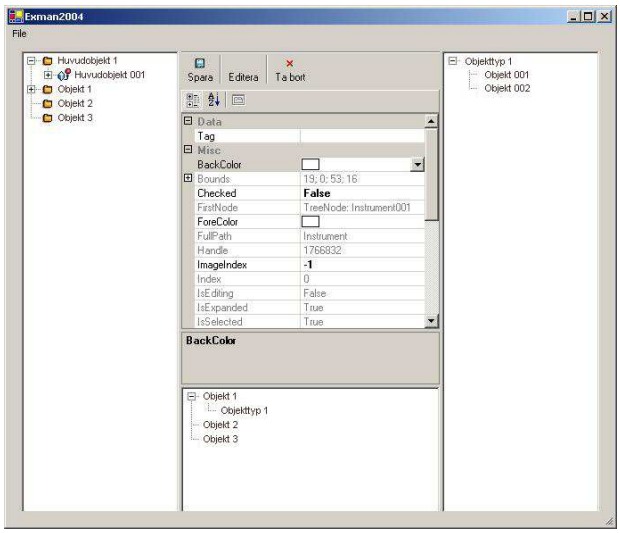
4.1 Problembeskriving

Vårt gränssnitt skall kunna lägga till tre olika sorters objekt i en databas samt koppla ihop dessa olika typer av objekt till en huvudobjektstyp. Man kan tänka sig huvudobjektet som en bottenplatta till en bil, dit man kan koppla olika varianter av karosser, motorer och hjul som symboliserar våra tre objektstyper. Objekttyperna har även en uppsättning attribut såsom färg, id, m.m. som skall kunna förändras. Dessa förändringar sker genom att användaren väljer det objekt som skall manipuleras ur en trädstruktur. Användaren kan sedan utföra en mängd operationer på objektet så som editera, skapa nytt eller ta bort ett objekt. För att skapa en miljö som är bekant för användaren skall utseendet och funktionaliteten i vårt gränssnitt påminna om ”Utforskaren” i Microsoft Windows.

4.2 Utformning av gränssnittet

Då vi började utformningen av gränssnittet hade vi två trädstrukturer istället för ett träd (se figur 4.1) och en lista som i den slutgiltiga versionen. Tanken var från början att vi skulle navigera i den ena trädstrukturen och skapa bindningar mellan objekt genom att dra noder från det ena trädet till det andra. Efter ha arbetat med denna modell kom vi fram till att det skulle gå snabbare att bara ha ett navigationsträd och istället för det andra trädet ha en lista med objekt som kunde bindas till den nod som var markerad i trädet. Resultatet av denna förändring visas i se figur 4.2. I och med det nya utseendet

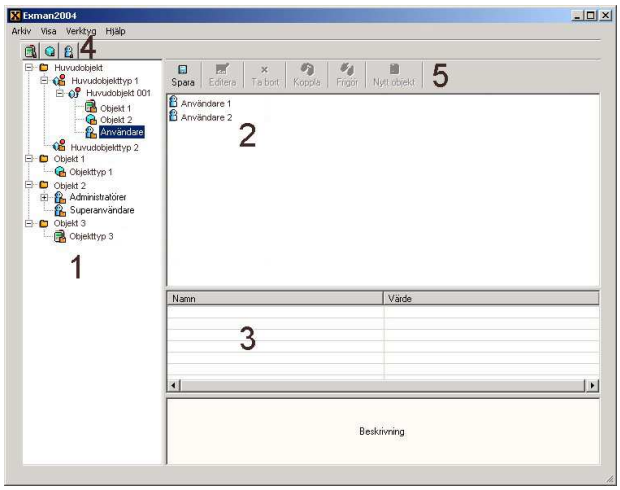
fick vi förutom ett mer snabbarbetat gränssnitt även ett gränssnitt som påminde mer om Microsoft Windows ”Utforskaren” vilket var ett önskemål ifrån Midroc.



Figur 4.1: Tidig version av gränssnittet

4.3 Beskrivning av prototypens funktionalitet

Vår applikation består av ett huvudfönster samt ett antal dialogfönster. Dialogfönstren hanterar händelser där användaren måste mata in mer omfattande data än enkel-, dubbel- eller högerklick som t.ex. vid för inloggning eller signering. Huvudfönstrets uppgift är att spegla det aktuella utseendet på databasen och därmed tillåta användaren att på ett enkelt sätt se vilka kopplingar och objekt som skall ändras. Huvudfönstret består av tre vyer samt två stycken verktygsfält. Delen märkt 1 i figur 4.2 är den trädvy som används för att navigera genom datan. Trädstrukturen representerar även strukturen på datan, det vill säga vilka objekt som tillhör vilken kategori samt vilka objekt som är bundna till varandra.
Den andra vyn märkt 2 i figur 4.2 är en objektslista som visar de objekt som den för tillfället markerade noden innehåller. Listan används för att skapa och ta bort bindningar mellan objekt och huvudobjekt. Redan bundna objekt visas med en särskild ikon i listan.



Figur 4.2: Huvudfönstret med egenskapsvyn aktiverad

Den sista vyn är egenskapsvyn märkt 3 i figur 4.2, den visar information om den valda noden i trädet eller raden i listan.

Verktygsfältet ovanför navigationsträdet (märkt 4 i figur 4.2) i har tre knappar för att lägga till nya objekt till databasen; en knapp för varje objekttyp som kan läggas till. Det sista verktygsfältet (märkt 5 i figur 4.2) har knappar för alla de olika manipulationer man kan göra i applikationen så som att skapa koppling, spara förändringar m.m. Dessa knappar aktiveras och avaktiveras beroende på de alternativ som finns för det objekt som är markerat. Vilka alternativ som finns tillgängliga kan till exempel bero på om det finns data att spara till databasen eller om objektet man har markerat har några kopplingar till andra objekt.

Huvudfönstret har även den huvudmeny man som användare är van att se i Windowsapplikationer. Huvudmenyn har alternativen ”Arkiv”, ”Visa”, ”Verktyg” och ”Hjälp”

Utöver de knappar och menyer vi beskrivit ovan kan användaren även få upp en kontextmeny om han högerklickar på en nod i navigationsträdet eller en rad i objektlistan. Kontextmenyn innehåller en rad snabbval som finns tillgängliga för den valda noden. Exempel på snabbval kan vara ”Ta bort objekt” eller ”Skapa nytt objekt”.

Användaren kan till viss del anpassa utseendet på huvudfönstret. Under ”Visa” i huvudmenyn kan användaren välja om han vill att egenskapsfönstret skall visas eller ej. Under ”Verktyg” kan man även ställa in vilka objekt som skall visas i objektslistan.

5 Implementation

I detta kapitel förklarar vi först några tekniker och nya komponenter i .NET vi använt oss av i vår prototyp. Vidare ger vi en översikt av hur vi byggt upp gränssnittet och det bakomliggande lagret för databashantering.

5.1 ADO.NET

ADO.NET är ett samlingsnamn på en uppsättning klasser för dataåtkomst i .NET. Även om ADO.NET främst förknippas med dataåtkomst från relationsdatabaser som SQL och Access så kan även andra datakällor användas. Man kan till exempel bygga upp en datastruktur med tabeller och fylla tabellerna med data programmatiskt och på så sätt simulera en databas.

ADO.NET är uppföljaren till Microsofts ADO vars första version kom 1996. ADO är en akronym för Activex Data Object och utvecklades för att man ville ha ett mer konsekvent sätt att skaffa sig åtkomst till datakällor oberoende av vilken sorts datakälla man arbetade mot och hur denna data var strukturerad. För en steg för steg anvisning om användandet av ADO.NET se [9].

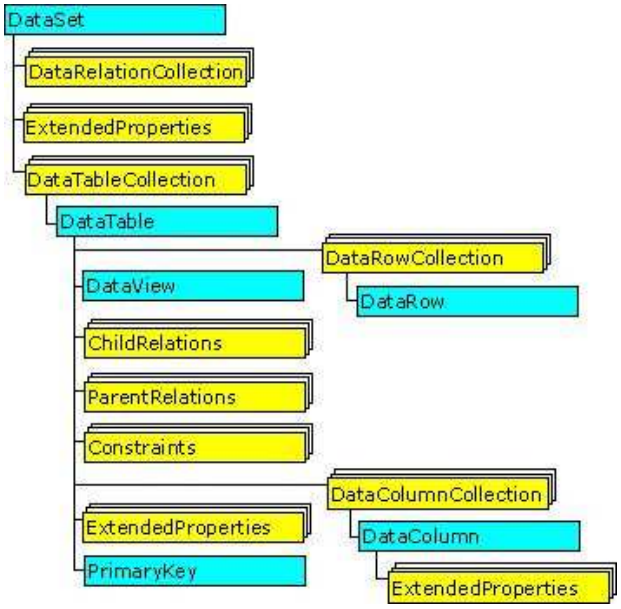
ADO.NET inför ett par stora fördelar jämfört med ADO. Den första är att ADO.NET använder sig av SOAP, som är ett XML-baserad protokoll, för all datatransport detta medför flera fördelar, bl.a. är XML [11] läsbart för människor vilket underlättar t.ex. felsökning. Då de flesta brandväggar är konfigurerade att släppa igenom HTML-data så passerar även XML-data dessa utan problem. Den annan stor fördel med att använda XML för datatransport är att det är enkelt för olika delar av ett system eller till och med olika system att samverka med varandra [14], det räcker att de andra delarna kan tolka XML för att kunna tillgodogöra sig datan. En annan fördel med ADO.NET är att det huvudsakligen är

tänkt att arbeta med frånkopplade DataSets (se mer om DataSets i nästa delkapitel), detta gör att man kommer ifrån den flaskhals som många samtidiga anslutningar till en databas utgör. Arbetet med DataSets istället för DataRecords som användes i ADO förenklar även utveckling och felsökning av en applikation.

Generellt sett finns det två sorters huvudobjekt i ADO.NET; konsumentobjekt (consumer objects) och tillhandahållningsobjekt (provider objects). Konsumentobjekten lagrar data medan tillhandahållningsobjeketen sköter anslutningen till datakällan. Tillhandahållningsobjeketet har bl.a. en anslutningssträng som innehåller adress till datakällan, lösenord och login m.m. och en kommandosträng som i vårt fall innehåller en SQL-fråga. Tillhandahållningsobjektet returnerar sedan en resultatmängd till datasettet. Olika datakällor använder olika klasser av tillhandahållningsobjekt medan data alltid lagras i ett och samma typ av DataSet. Tanken med detta är att om man vill byta datakälla så behöver man bara ändra på tillhandahållningsobjektet.

5.1.1 DataSet

En av nyheterna i ADO.NET är datastrukturen DataSet. Till skillnad från ett RecordSet som ADO använder sig av så kan ett DataSet innehålla flera tabeller. Ett DataSet kan även beskriva relationerna mellan data och begränsningar (constraints) för tabeller, strukturen hos ett DataSet visas i figur 5.1. Kort förklarat kan man säga att ett DataSet är en lokal minnesresident ögonblicksbild av en utvald del av databasen. DataSets har väl utvecklat stöd för arbete mot frånkopplade datakällor. Man kan till exempel editera den data som lagrats i datasetet genom att utföra operationer så som Insert, Update och Delete lokalt på datasetet och sedan skriva dessa förändringar till datakällan. Då det enbart är den data som blivit förändrad som skrivs tillbaka till datakällan bidrar detta arbetssätt till att minska belastningen på det nätverk som används mellan datakälla och applikationen.

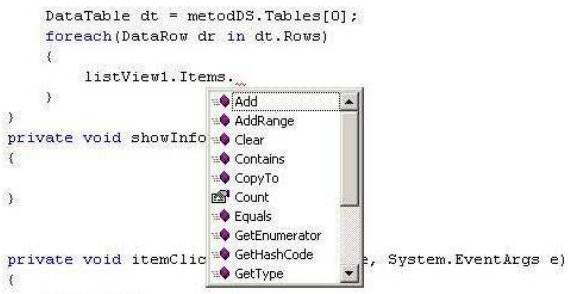


Figur 5.1: Strukturen hos ett DataSet

DataSets är per default otypade men kan även vara typade. Typade DatasSets får man genom att man skapar en klass som ärver av klassen DataSet. Typade DataSets har ett par stora fördelar; för det första så kan man använda Visual Studio .NETs mycket användbara IntelliSense. Intellisense sparar tid vid utvecklingsarbetet, dels för att man slipper kontrollera tabell och kolumnnamn i en eventuell databas och dels för att man skriver mindre slarvfel. En annan fördel med typade DataSets är att fel som skulle generera ett runtime fel som till exempel ett felstavat tabellnamn nu istället genererar ett kompileringsfel.

5.2 IntelliSense

IntelliSense är ett verktyg i Visual Studio.NET som försöker känna av vad användaren vill skriva och hjälper utvecklaren med så kallad ”Code Completion”. Code Completion innebär att Visual Studio ger utvecklaren en kontextkänslig lista med möjliga avslut på det han skriver för tillfället. Till exempel så ger IntelliSens utvecklaren alla metoder och attribut hos en klass då man använder punktnotation på en instans av klassen.



Figur 5.2: IntelliSense i Visual Studio.NET

5.3 Asynkron databasåtkomst

I ett fleranvändarsystem måste datakonsistens upprätthållas p.g.a. samtidighet(concurrency) för att man skall kunna försäkra sig om att datan i databasen är korrekt. Detta är särskilt viktigt i ett system där man arbetar frånkopplat från databasen då det kan ta lång tid mellan att man läser data till dess att den förändrade datan skrivs tillbaka till databasen. Samtidighetskontroll innebär att då flera användare arbetar med samma data så uppdateras datan i databasen på ett kontrollerat sätt. Det finns flera tillvägagångssätt till att lösa problem som uppstår vid samtidighet. Mer om detta finns att läsa i [3].

5.4 Lagrade procedurer

Lagrade procedurer är en namngiven sekvens av SQL-kommandon kompilerade till en enhet, som lagras och exekveras på serversidan. Vissa operationer, som t.ex. en specifik databasfråga, som ofta ställs av en eller flera applikationer implementeras lämpligen som en lagrad procedur. Appilkationen som använder sig av den lagrade proceduren (på Micorsoft SQL server 2000) kan bifoga inparametrar till proceduranropet, och läsa resultatet av anropet som utparametrar, returvärde (lagrad funktion) eller en samling datatupler. Fördelar med att använda lagrade procedurer är:

- **Modularitet** Genom att skapa databasfrågor som lagrade procedurer i databasservern kan man erhålla en mer modulär uppbyggnad av ett system; De lagrade procedurerna kan underhållas separat, och förändringar kan göras i de lagrade procedurerna utan att kompabilitetsproblem uppstår hos applikationer som anropar dem sålänge som gränssnittet, d.v.s. procedurens namn, parametrar och returvärde (för lagrade funktioner), inte förändras.

- **Säkerhet** Lagrade procedurer kan användas som en säkerhetsmekanism. Genom att endast låta en applikation anropa lagrade procedurer och ej tillåta direkt åtkomst av databasens tabeller kan applikationer tillåtas nyttja databasen under begränsade och kontrollerade former.

5.4.1 Lagrade procedurer i ADO.Net

SqlCommand - Enstaka fråga ADO.Net innehåller klassen SqlCommand, som implementerar funktionaliteten hos ett SQL-kommando. Nedan finns ett exempel på hur SqlCommand kan användas för att ställa en SQL-fråga till en databas:

```
...
SqlConnection conn = new SqlConnection(connection_string);
string select_statement = "SELECT * FROM DataTable";
SqlCommand
cmd = new SqlCommand(select_statement,conn);

conn.Open() //Öppna anslutning

SqlDataReader read = cmd.ExecuteReader();
...
```

Ovanstående kod skapar ett anslutningsobjekt till en Sql-server med anslutningsträngen connection_string. Strängen select_statement innehåller Sql-frågan som skall ställas till databasen. SqlCommand-objektet command knyter Sql-frågan till anslutningsobjektet, som öppnar en anslutning på nästföljande rad med metoden Open(). SqlDataReader-objektet read på sista raden läser den ström av datarader från servern som exekveringen av cmd resulterar i.

SqlCommand - Anrop till lagrad procedur Klassen SqlCommand kan förutom att användas för enstaka Sql-kommandon som i exemplet ovan även användas för att anropa lagrade procedurer på servern.

```
CREATE PROCEDURE sp_DataTableFilter (@Filter varchar(64))
    AS

    DECLARE @SQLSelect varchar(128)
    @SQL_Select = "SELECT * FROM DataTable " +
                "WHERE " + @Filter



...
SqlConnection conn = new SqlConnection(connection_string);
SqlCommand sp_cmd = new SqlCommand("sp_DataTableFilter",conn);
sp_cmd.CommandType = CommandType.StoredProcedure;
param.Value = "Flag = 1";
qlParameter param = p_cmd.Parameters.Add("@Filter",SqlDbType.VarChar,64);
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = sp_cmd;
DataSet ds = newDataSet();
```

```
da.fill(ds);
...
```

I ovanstående kodexempel används anslutningsobjektet conn med anslutningssträngen connection_string för att ansluta till en databasserver. SqlCommand-objektet sp_cmd använder anslutningsobjektet conn tillsamman med en sträng som anger vilken lagrad procedur som skall användas. Egenskapen CommandType i sp_cmd anger att det är en lagrad procedur som sp_cmd är kopplad till. Därefter skapas en parameter genom att methoden Add anropas i sp_cmds Parameters-samling (collection). Parametern ges namnet "@Filter", och sätts till typer VarChar, med längden 64. Parametern värde sätts att vara "Flag = 1", som i vårt exempel kommer att vara filtreringskriteriet. Därefter skapas ett SqlDataAdapter-objekt där sp_cmd sätts att vara det sql-kommando som skall användas när en fråga ställs mot databasen. Slutligen skapas ett dataset som fylls av dataadaptern med hjälp av den lagrade proceduren som sp_cmd anger.

5.4.2 Lagrade procedurer i vår prototyp

Midrocs befintliga applikation utför databasoperationer genom att anropa lagrade procedurer. Detta innebär att det finns en uppsättning färdiga lagrade procedurer för vår prototyp att använda. För varje typ av objekt som finns representerade i databasen finns lagrade procedurer för följande operationer

- lägga till
- ta bort
- uppdatera (anropas när objektegenskaper förändras)
- hämta en
- hämta alla

- skapa och ta bort koppling (för de objekt där upprättandet av kopplingar är möjliga)

Den lagrade proceduren innehåller förutom sql-kommandon för den givna operationen även operationer för att logga gjorda förändringar i en systemlogg.

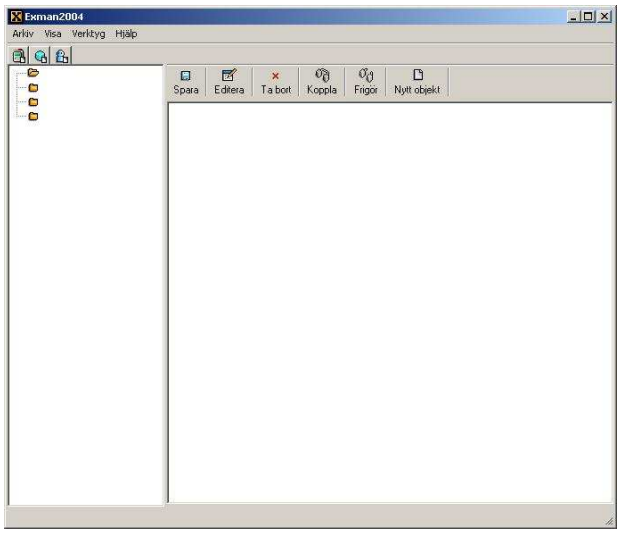
5.5 Prototypens frontend

Vi har valt att utveckla gränssnittet i VB.NET med hjälp av Microsoft Visual Studio .NETs RAD-verktyg Windows Forms designer. Under arbetets gång har vi testat ett flertal olika utseenden på vårt. Detta har inte varit speciellt tidskrävande just tack vare Windows Forms designer.

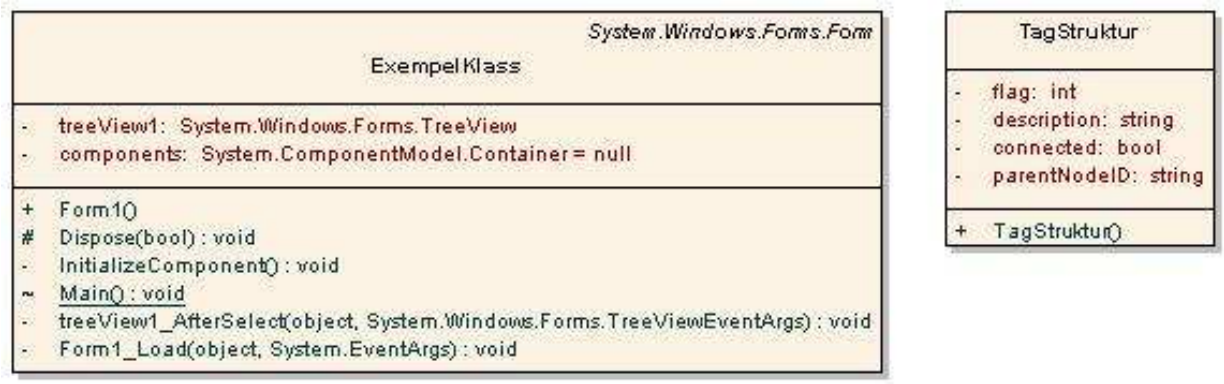
Varje fönster i gränssnittet är ett formulär. Dessa formulär ärver av klassen Form i namnrymden System.Windows.Forms. i .NETs klassbibliotek. Till dessa formulär lägger man sedan till de kontroller man vill ha, t.ex. list och trädvyer, och hanterare(handlers) till dessa kontroller. Hanterare används för att ta hand om de olika händelser(events) som kontrollerna i gränssnittet kan generera. Exempel på händelser kan vara enkelklick, dubbelklick och högerklick. Utöver hanterarna har vi skapat vissa hjälpfunktioner för att t.ex. bygga kontextmenyer eller för att visa / dölja vissa delar av gränssnittet. Figur 5.3 visar huvudfönstret då användaren valt att dölja egenskapsvyn.

När användaren interagerar med ett objekt i gränssnittet som representerar data i databasen fångar gränssnittet upp denna händelse och anropar applikationens backend. Gränssnittet skickar objektet användaren har manipulerat samt i vissa fall även en referens till den list- eller trädvy som det berörda objektet tillhör som inparametrar till den funktion i backendet som hanterar den aktuella händelsen. Varje objekt som representerar data i gränssnittet t.ex. en trädnod har en ”tagg” kopplad till sig, denna tagg är en datastruktur som innehåller den data backendet behöver för att kunna förändra databasen. Exempel på data i taggen kan vara id:t på det valda objektet, vilken typ av objekt som valts samt om det är bundet till något huvudobjekt. Backendet utnyttjar sedan denna information för att förändra databasen på ett sätt som motsvara den förändring användaren gjort i gränssnittet. En

Exempelklass med tillhörande tagg visas i 5.4.



Figur 5.3: Huvudfönstret med dold egenskapsvy



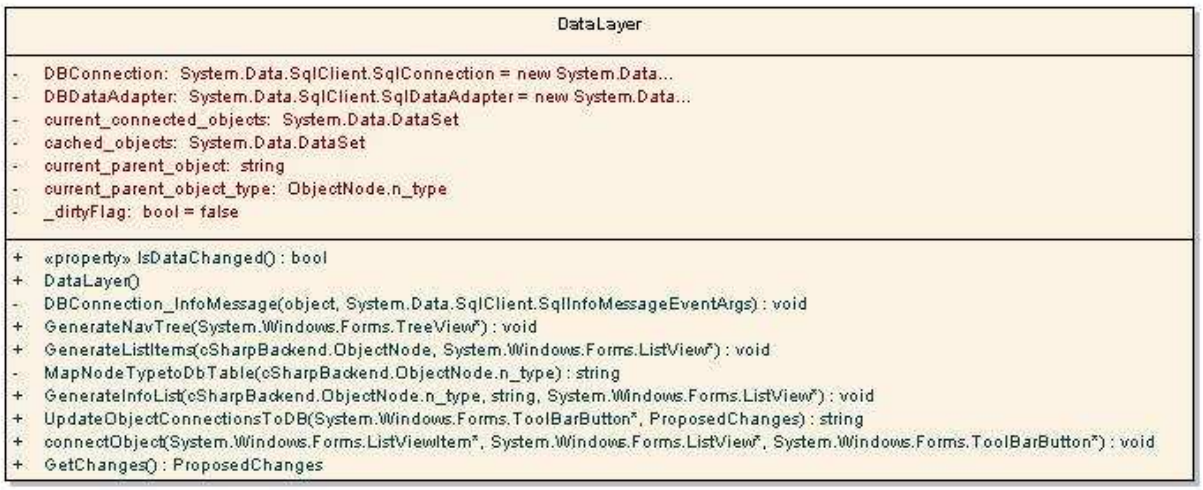
Figur 5.4: Ett formulär innehållande en trädkontroll och tillhörande tagg

5.6 Prototypens backend

Då ett av målen med arbetet var att vi skulle undersöka möjligheten att använda flera olika programspråk i en implementation valde vi att göra vår backend i C#. Det är möjligt att

använda flera olika programspråk i samma applikation i .NET, dock visade det sig att man inte kan ha både .vb-filer och .cs-filer i samma projekt i Visual Studio .NET. Lösningen till problemet är att man får skapa två separata projekt (ett projekt innehåller en eller flera programkodsfiler), inkludera dessa båda projekt i samma ”solution” (en solution innehåller ett eller flera projekt) och sedan lägga till en referens från det ena projektet till det andra. Se mer om ”projekt” och ”solutions” i [13].

Den klass som utgör största delen av vår backend har vi kallat för ”DataLayer” (se figur 5.5). Det är denna klass som sköter all databashantering. Man kan generalisera de uppgifter som DataLayer utför till läsning av data från databasen samt skrivning till databasen. Nedanstående stycke kommer gå igenom vilka steg som genomförs i dessa två fall.



Figur 5.5: Klassen DataLayer

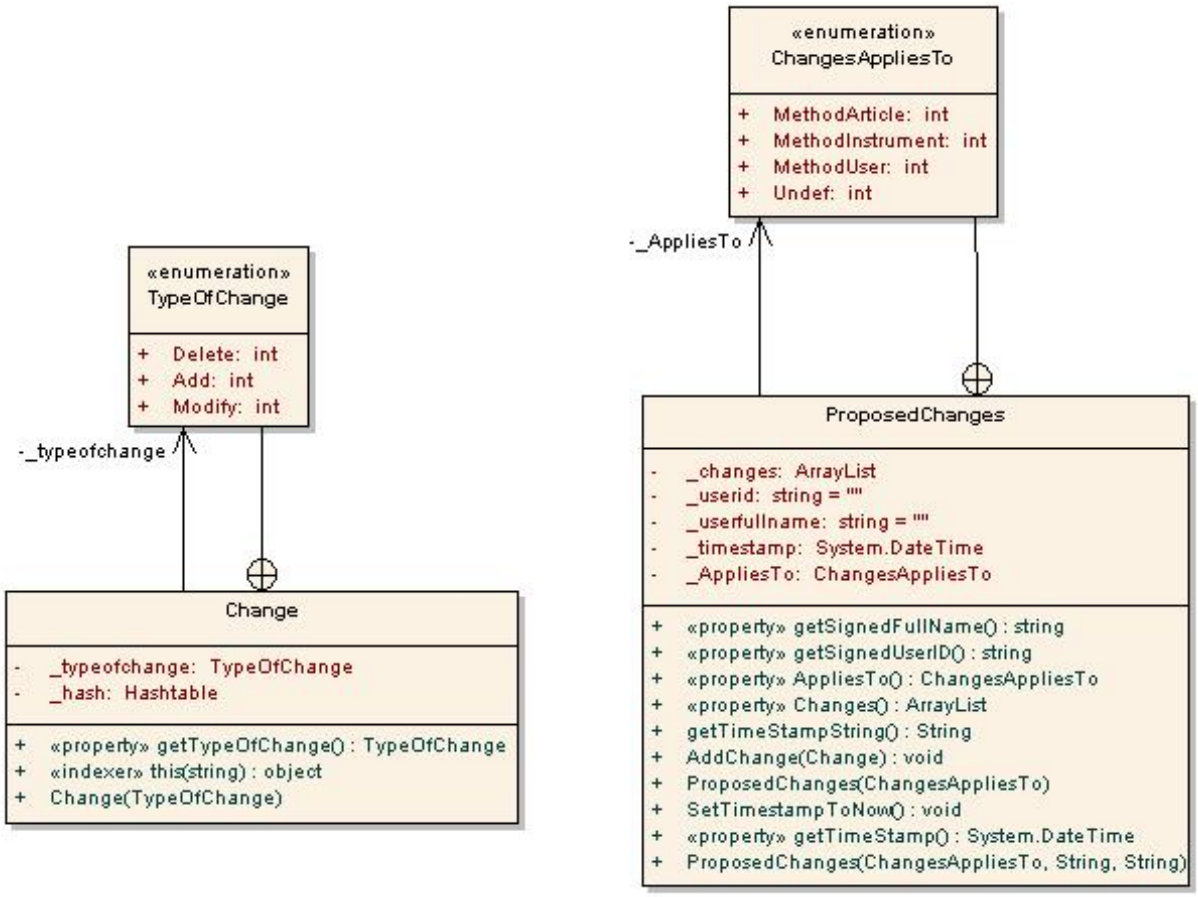
Läsning Ett exempel då data behöver läsas från databasen är då en trädnod markeras i det grafiska gränssnittet. Den markerade nodens barnnoder skall då visas i listvyn och behöver därför läsas in från databasen. Först anropar gränssnittet någon lämplig metod i DataLayerklassen, i detta fallet är det metoden GenerateListItems som anropas. GenerateListItems tar två argument, noden som klickats och en referens till listvyn som skall

fyllas med data. GenerateListItems använder information från den klickade nodens tagg (se ”Prototypens frontend”) för att ta reda på vilken data som skall hämtas från databasen. Denna data används sedan tillsammans med en lagrad procedur för att hämta rätt data från databasen. Datan sparas lokalt i två stycken DataSets, ett DataSet innehåller tabeller som visar vilka objekt som är kopplade till varandra i gränssnittet, det andra DataSettet innehåller detaljinformation om objekten. Utifrån denna data och med hjälp av referensen till listvyn lägger GenerateListItems sedan till de objekt som skall visas i listan.

Skriving Ett exempel då data skall skrivas tillbaka till databasen är då användaren har bundit ett eller flera objekt i till ett huvudobjekt(se kapitel 4). Ett objekt binds om det är obundet och det dubbelklickas i listvyn. När ett objekt dubbelklickas anropas metoden ConnectObject i DataLayer med det klickade objeketet samt en referens till listvyn som inparametrar. ConnectObject lägger till en rad i tabellen för kopplingar mellan den klickade objektstypen och huvudobjekt i DataSetet som hanterar kopplingar. Då användaren lämnar den aktuella trädnoden eller trycker på sparaknappen så startas en händelsekedja för att försöka skriva förändringarna som gjorts lokalt till databasen. Det första som sker är att en metod i DataLayer anropas, där alla rader som förändrats i DataSetet sedan det hämtades från databasen sammanställs. Ett objekt av klassen Change skapas för varje förändring som upptäcks. Detta objekt innehåller den information som krävs för att unikt identifiera de två databasobjekt (huvudobjekt och objekt) vars emellanliggande koppling förändrats och vilken typ av förändring det rör sig om: antingen upprättandet av en koppling eller en frånkoppling. Samtliga Change-objekt som skaptas vid denna sammanställning lagras innehållna i ett objekt av klassen ProposedChanges (se figur 5.6). Detta ProposedChanges-objekt skickas tillbaks till frontend.

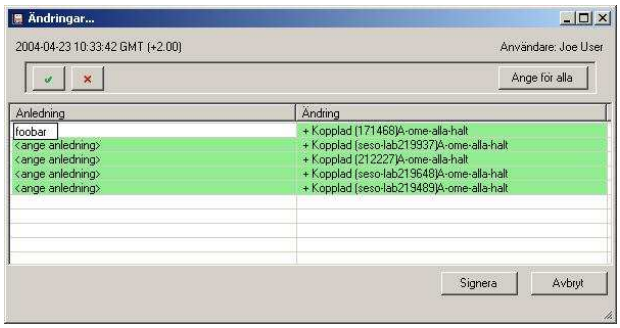
Åter i frontend presenteras en dialog för användaren där informationen om de gjorda ändringarna från PropsedChanges-objektet visas i listform (figur 5.7). Användaren har nu möjlighet att välja de förändringar som skall exkluderas från samlingen, och även ange en

text som beskriver anledningen till varför förändringen gjorts. Därefter, när användaren valt att signera förändringarna för skrivning till databasen visas en signeringsruta. Förutsatt att användaren anger korrekta användaruppgifter lagras information om den signerande användaren och klockslag för signeringen i ProposedChanges som skickas till DataLayer för skrivning.



Figur 5.6: Klasserna Change och ProposedChanges

I DataLayer gås varje förändring igenom en i taget, typen av förändring kontrolleras och beroende på denna anropas en lagrad procedur för antingen upprättandet av en koppling eller frånkoppling mellan två objekt i databasen. Information om vilka objekt det rör sig



Figur 5.7: Dialog där användaren kan välja vilka ändringar som skall sparas

om finns lagrade i det aktuella Change-objektet som representerar förändringen. I och med anropet till den lagrade proceduren skapas också en transaktion för att förhindra att två samtidiga användare utför kopplingsförändringar mellan samma objekt, samtidigt, vilket skulle kunna leda till fel i databasen (se kap. 5.3). Efter det att den lagrade proceduren anropats kontrolleras dess returvärde och eventuella fel som upstått under skrivningen meddelas användaren.

Att användaren tillåts samla på sig flera förändringar och sedan signera dessa med en enda signering har vi skapat för att uppfylla de krav på förbättringar som vi nämner i kap 2.2.

6 Tester

Enligt de ställda kraven skall acceptanstest (FAT och SAT) samt kodgranskning utföras. Se nedan för en kort beskrivning av dessa test. Avsnitt 6.3 redogör för hur vår prototyp klarade testerna.

6.1 Acceptanstest

Acceptanstest utförs för att verifiera att applikationen fungerar i enligthet med vad som angetts i dess funktionsspecifikation, eller m.a.o. att den uppfyller de krav som kunden ställer på den slutgiltiga produkten.
Factory Acceptanse Test(FAT) är ett acceptanstest som utförs hos utvecklaren, oftast i närvaro av kunden.
Site Acceptanse Test(SAT) är ett acceptanstest som utförs hos kunden på den installerade produkten.

6.2 Kodgranskning

Syftet med kodgranskningen är att säkerställa att programkoden håller en god kvalité och att den följer uppsatta riktlinjer, t.ex. standard för namngivning av programelement och programkodslayout. Vidare granskas läsbarheten och semantiken, d.v.s. huruvida programkoden är lätt att läsa och att innebörden är klar och tydlig.

6.3 Test av vår prototyp

Då denna rapport skrivs har vår prototyp endast genomgått en iteration av acceptanstestet (FAT). Då testades av uppdragsgivaren tre av prototypens funktioner: inloggning, navigering och upprättandet av kopplingar mellan objekt. Anledningen till att just dessa funktioner testades var främst tidsbrist; givet att mer tid funnits hade fler och mer ingående tester utförts. Testerna berörde den funktionalitet som vi koncentrerat oss mest på att

implementera. Tilläggas bör att vår prototyp vid detta tillfälle godkändes i de tester som rörde två av de tre nämnda funktionaliteterna. Funktioner för användarautentifiering fanns inte implementerade och därför underkändes prototypen i de tester som rörde inloggning. Bortsett från den funktionalitet som godkänts i denna inledande iteration och givet att användarautentifiering finns implementerad så är tester som kontrollerar hur prototypen uppför sig och fungerar i en fleranvändarmiljö något som vi anser bör testas i kommande testiterationer.

7 Resultat

När arbetet skulle planeras utgick vi ifrån en kravspecifikation vi fått från Midroc. Utifrån denna kravspecifikation satte vi upp de mål som vårt arbete skulle uppfylla, se avsnitt 1.2. I detta kapitel mäter vi målen, de flesta har uppnåtts men vi har tvingats ge avkall på några.

Huvudmålet med arbetet var att skapa en användarvänlig prototyp. Vi har inte utfört någon undersökning på hur olika användare upplever gränssnittet och de enda personer som använt gränssnittet är vi och vår handledare på Midroc. Vi som arbetat med projektet upplever det dock som att gränssnittet ger en bra översikt av datan i databasen och att systemet med att spara flera förändringar per signering fungerar bra.
Om det funnits mer tid till arbetet hade vi önskat att vi kunnat testa gränssnittet m.h.a. en grupp användare som inte redan var insatta i hur gränssnitt och databasen var utformad. Denna testgrupp hade kunnat ge en mer rättvis bild av hur användarvänligt gränssnittet egentligen är.
De frågeställningar Midroc hade om .NET såsom möjligheter att använda flera programspråk och alternativa metoder att distrubiera klienter, har vi undersökt och redovisat i vår förstudie. I enighet med Midroc Electros utvecklingsmodell har vi även sammanställt en funktionsspecifikation som ger en översiktlig beskrivning av systemet samt en mer detaljerad beskrivning av de olika användarfall vi identifierat. Vid arbetet med funktionsspecifikationen använde vi programmet Enterprise Architect(EA) som är en applikation för systemmodellering som används av Midroc Electro. EA kunde bland annat användas till att automatgenerera stora delar av dokumentationen för våra användarfall utifrån UML-diagram. Vi har tidigare använt programmet Rational Rose till liknande uppgifter men upplevde EA som mer intuitivt och lättförståligt.
De mål vi inte har uppfyllt är dynamisk språkhantering och transaktionshantering. Vi har implementerat transaktionshantering för vissa delar av våran prototyp och planerat för hur det skall införas i resterande delar av applikationen. ADO.NET har ett väl utvecklat

stöd för transaktioner och vi känner att det skulle gå relativt enkelt att uppfylla detta mål om vi hade något mer tid. Vi har även utelämnat delar av dokumentationen såsom användarmanual och installationsmanual då inläsning och implementation tog längre tid än förväntat.

8 Summering av projektet

De tio veckor vi hade till förfogande räckte inte till för att vi skulle bli färdiga med uppgiften. Fastän vi arbetat cirka fyrtio timmar per vecka skulle vi behövt ytterligare någon vecka för att uppfylla de mål vi missat. Det är svårt att säga ifall uppgiften var för omfattande för att utföras på tio veckor, men för någon med vana av att arbeta i .NET skulle tiden antagligen varit tillräcklig.
Microsofts .NET-plattform är mycket omfattande och om man som oss inte har arbetat med .NET tidigare bör man se till att man har extra tid till förfogande för att sätta sig in i plattformen ordentligt.
Då vi berört många olika delar av .NET har mycket av tiden gått åt till att ta reda på vilken teknik som är tänkt att användas för att lösa ett visst problem. Den stora fördelen med att arbeta i .NET som vi ser det, är att när man väl vet vilket tillvägagångssätt man skall ha för att ett problem så går det ofta väldigt fort och smärtfritt att implementera lösningen, mycket tack vare de omfattande klassbiblioteken. En annan anledning till att vi förlorade tid var att vi tidigt i implementationen skapade en del temporära funktioner och arbetade med ”dummy”-data för att få en känsla för hur gränssnittet skulle se ut när det var färdigt. Om vi istället börjat arbeta mot databasen direkt hade detta sparat oss en del tid.

Det mest intressanta och lärorika med arbetet har varit att komma ut i till ett företag som Midroc Electro och se hur det går till när de utvecklar mjukvara. Samtidigt är det detta nya sätt att arbeta som har varit det svåraste momentet. Mycket tid har gått åt till att sätta in i Midroc Electros arbetsmetoder, vänta på ansvarigas åsikter om arbetet samt att skapa en mer omfattande dokumentation än vad vi är vana vid.
En positiv del av arbetet har varit att arbeta med .NET och Visual studio. Detta har varit mycket lärorikt och då .NET är en av Microsofts nya storsatsningar tror vi att den kunskap vi skaffat oss under arbetets gång kommer vara av nytta för oss i framtiden. Får

vi möjlighet kommer vi välja att jobba med .NET i kommande programmeringsprojekt då vi upplevt både C# och utvecklingsmiljön Visual Studio .NET som lätt att arbeta i.

Referenser

[1] Jason Clark. Code access security and distribution features in .net client-side apps. http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/02/06/rich/toc.asp, 2004-03-09.

[2] Jason Clark. Return of the rich client. http://msdn.microsoft.com/msdnmag/issues/02/06/rich/, 2004-04-03.

[3] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems.* Addison-Wesley, 3st edition, 2000.

[4] Dino Esposito. Hosting .net applications in the browser. http://devcenter.infragistics.com/Articles/ArticleTemplate.Aspx?ArticleID=1264, 2004-03-17.

[5] D. Meier et al. .net security overview. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh06.asp, 2004-03-23.

[6] Andrew Ma. Adding assembly to gac. http://www.devhood.com/tutorials/ tutorial_details.aspx?tutorial_id=106, 2004-03-20.

[7] Ted Pattison. Naming and building assemblies in visual basic .net. http://msdn.microsoft.com/msdnmag/issues/03/08/basicinstincts/, 2004-03-23.

[8] Matt Pietrek. Avoiding dll hell. http://msdn.microsoft.com/msdnmag/issues/1000/metadata/default.aspx, 2004-03-23.

[9] Rebecca Riordan. *Microsoft ADO .NET Step by Step.* Microsoft Press, 1st edition, 2002.

[10] Unkown. W3c world wide web consortium. http://www.w3c.org, 2004-03-05.

[11] Unkown. O'reilly xml from the inside out. http://www.xml.com, 2004-03-15.

[12] Unkown. Global assembly cache tool. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptutorials/html/global_assembly_cache_utility__gacutil_exe_.asp, 2004-06-03.

[13] K. Watson, M. Bellinaso, O. Cornes, D.Espinosa, Z. Greenvoss, C. Nagel, J. Hammer, J. Reid, M. Skinner, and E. White. *Börja med C#.* Pagina.se, 1st edition, 2002.

[14] Roger Wolter. Simply soap. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml10152001.asp, 2004-03-09.