Peter Polviander

REVISIONSHANTERING OCH GENERISK FUNKTIONALITET – ETT .NET-PROJEKT

FÖRETAGSEKONOMI OCH TURISM

2010

VASA YRKESHÖGSKOLA

Utbildningsprogrammet för företagsekonomi

ABSTRAKT

Författare	Peter Polviander
Lärdomsprovets titel	Revisionshantering och generisk funktionalitet – Ett .NET projekt
År	2010
Språk	svenska
Sidantal	55
Handledare	Kenneth Norrgård

Detta arbete är en beställning gjord av en stor tillverkare av diesel-maskiner och dess behov att ett nytt system för lagring och hantering av data för maskiners konfigurationer med deras prestandadata.

Projektet börjar med specificering av uppdragsgivarens behov varefter olika ramverk och lösningar diskuteras samt en design på hög nivå presenteras. Nya ramverk har tagits i beaktande och valet föll på en, då arbetet gjordes, ny teknik i Microsofts arsenal det vill säga ASP.NET MVC.

Arbetet var rätt omfattande och en fokusering har här gjorts på hur generisk funktionalitet har uppnåtts på olika sätt. Databasen som stod för grund för arbetet innehöll många tabeller och fält och för att åstadkomma en applikation lätt att underhålla och för att minska på utvecklingstiden var det ett viktigt mål. Alla tabeller filtreras och sorteras till exempel med samma metoder.

Systemet som utvecklades lever inte isolerat och i arbetet visas även en viss problematik som uppstod för att uppfylla omgivningens krav och hur detta löstes. Kort presenteras även hur datasäkerheten löstes med olika användarrättigheter.

Slutligen i kapitlet framåtblick presenteras hur programmet ytterligare skulle kunna utvecklas på basen av ny teknik som kommit till under senare tid.

Ämnesord programutveckling, ASP.NET, relationsdatabaser, IT-arkitektur

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Utbildningsprogrammet för företagsekonomi

ABSTRACT

Author	Peter Polviander
Title	Revisioning and Generic Functionality – A .NET Project
Year	2010
Language	Swedish
Pages	55
Supervisor	Kenneth Norrgård

This work is an order made by a large manufacturer of diesel-engines for its need of a new system, which stores and handles data concerning different engine-configurations and their performance-data.

The project starts with a specification of the needs and follows with a discussion about different frameworks and solutions to presenting a design on a high level. New frameworks have been taken into consideration and the choice, ASP.NET MVC, was a relatively new technology when the work was made.

The work is rather large and this presentation focuses on how a generic functionality has been achieved in different ways. The previous database, which was a base for this work contained many tables and fields. In order to make an application easy to maintain and to diminish the coding work this was an important goal. For example all tables are sorted and filtered with the same methods.

The system which was developed does not live in isolation and in this work is also presented some problems that arose in order to meet the surrounding demands and how it was resolved. How the data-security is solved with different user-rights is also briefly presented.

In the end a look is made into the future, how the program could be improved further by adopting new technology which has been added to the used framework lately.

Keywords Programming, ASP.NET, Relational Database, Software Architecture

INNEHÅLL

ABSTRAKT....2
ABSTRACT....3

INNEHÅLL....4
1 BAKGRUND, PROBLEMANALYS OCH UPPLÄGG....6
1.1 Det gamla systemet....6
1.2 Krav på nya systemet och analys av problem....7
1.2.1 Revisionshantering och historiska data....7
1.2.2 Läsbarhet och underhållbarhet....8
1.2.3 Flexibilitet....8
1.3 Upplägg....9
2 VAL AV RAMVERK OCH DESIGN PÅ HÖG NIVÅ....10
2.1 Val av ramverk....10
2.1.1 Silverlight....10
2.1.2 ASP.NET....11
2.2 Databas och layout....12
2.3 Generisk funktionalitet....13
2.3.1 .NET Generics....13
2.3.2 .NET Reflection....14
3 UTFÖRANDE....18
3.1 Arkitektur....18

3.1.1 Datamigration 18
3.1.2 N-tier och MVC 18
3.2 Revisionshantering 20
3.2.1 Nya revisioner 21
3.2.2 Revision-Cascading 24
3.2.3 Kontroll av historiska data 28
3.3 Fälttillägg dynamiskt i tabeller 33
3.4 Filtrering och sortering generiskt 33
3.5 Val av fält för visning på ListVy-sidan 41
4 EXTERNA SYSTEM 48
5 DATASÄKERHET 49
6 FRAMÅTBLICK 51
6.1 Datatypen Dynamic 51
6.2 Namngivna och valfria parametrar 53
7 SAMMANFATTNING 55
KÄLLFÖRTECKNING 56

1 BAKGRUND, PROBLEMANALYS OCH UPPLÄGG

Ett större företag inom den tunga industrin som tillverkar stora dieselmotorer hade behov av en uppdatering av en databas och nytt användargränssnitt för underhåll av data. Databasen ifråga lagrar olika mätvärden som tas vid provkörning av nya motormodeller och konfigurationer.

Företaget hade från tidigare en Microsoft-Access databas som hittills direkt uppdaterades i tabellerna av nyckelanvändarna. Behovet var att övergå till en SQL-databas och skapa ett användargränssnitt.

En ny viktig egenskap för den nya databasen var också att man skulle kunna få ut ändringshistorik d.v.s. ändringarna skulle göras så att man skapar nya revisioner istället för direkt ändrar data.

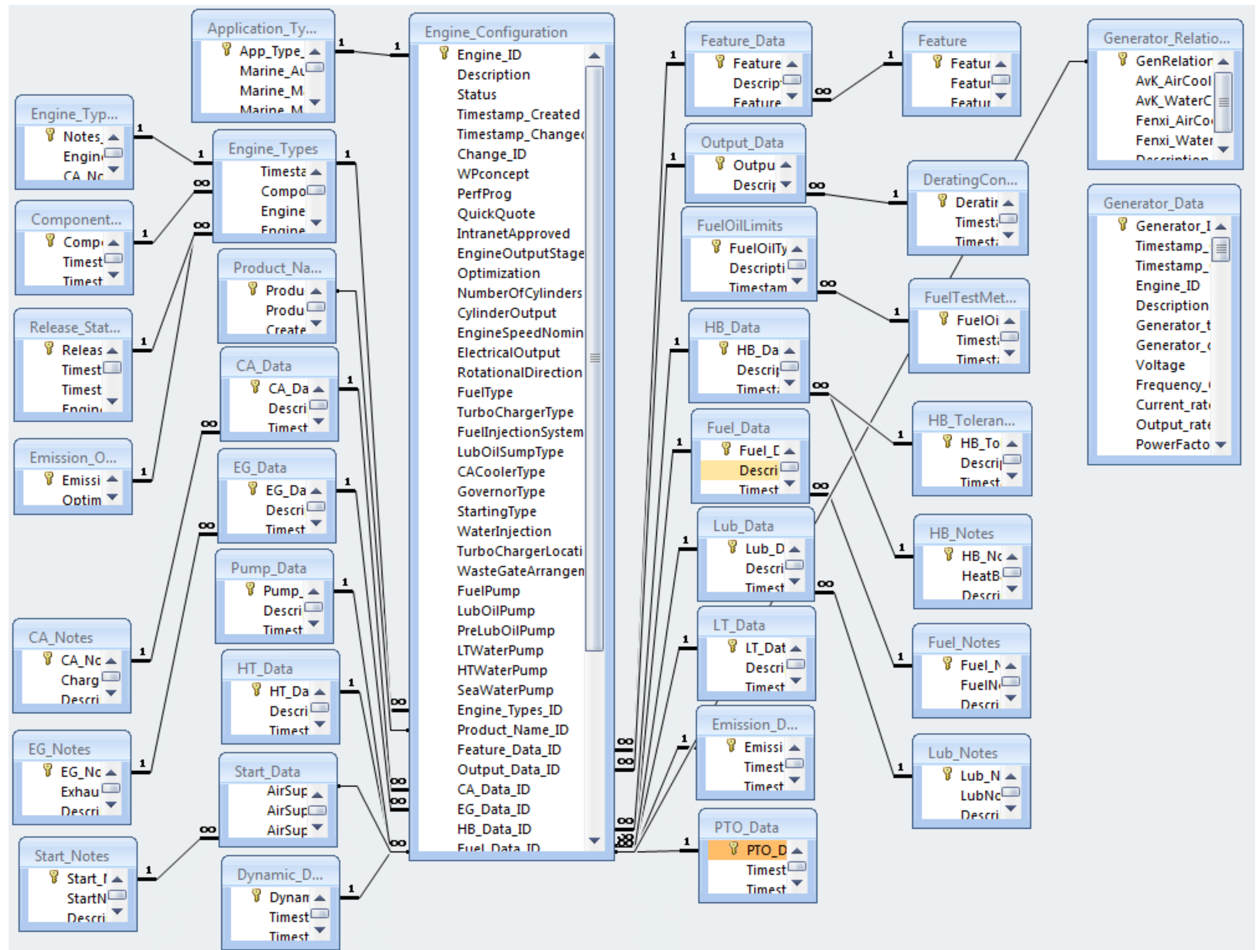
Vidare uppkom under arbetets gång önskemål och krav, många p.g.a. att man ville ha liknande möjligheter och flexibilitet som att arbeta direkt med en Access-databas såsom att enkelt lägga till kolumner, filtrera data etc. En närmare genomgång av det gamla befintliga systemet de nya önskade egenskaperna presenteras här nedan.

1.1 Det gamla systemet

Den ursprungliga gamla databasen består av knappa 50 tabeller varav en är den så kallade huvudtabellen, Engine_Configuration. Till denna är så kallade sub-konfigurationer kopplade så att en sub-konfiguration kan användas av flere huvudkonfigurationer men inte vice versa. Sub-konfigurationerna kan i sin tur även använda sig av sub-tabeller, schemat illustreras i Figur 1.

Databasen är enkel och logisk med en primärnyckel bestående av ett (integer) fält. Som tidigare nämnts hade databasen inget användargränssnitt utan nyckelanvändarna redigerade direkt i databasen.

Det fanns ingen behörighetskontroll förutom att databasen var lösenordsskyddad, hade man tillgång till den hade man fulla rättigheter att göra ändringar, radera data etc.



Figur 1 Schema över den gamla databasen

1.2 Krav på nya systemet och analys av problem

1.2.1 Revisionshantering och historiska data

Den viktigaste nya egenskapen var att data inte skrivs över utan uppdateringar skall ske genom att skapa nya revisioner. Man skall kunna se ändringshistoriken och kunna ta fram en viss revision. Lösningen för detta var att göra primärnyckeln sammansatt

så att den består av ID samt Revisionsnummer. I och med att en konfiguration är sammansatt av fler tabeller uppstår vissa frågor när nya revisioner skall skapas och för vilka tabeller.

I det följande diskuteras när nya revisioner skapas. En huvudkonfiguration (Engine_Configuration) består av flere tabeller, d.v.s. alla under konfigurationer som kan vara länkade till flere huvudkonfigurationer. Dessa är alltså beståndsdelar i en huvudkonfiguration. När en beståndsdel ändras, ändras också helheten. Sålunda när en ny revision skapas för en underkonfiguration, måste också nya revisioner skapas för alla objekt som använder denna som beståndsdel. Det kan här bli fråga om en kedja där en uppdatering orsakar en kaskad av följduppdateringar. Det nya systemet måste hantera sådana funktioner automatiskt.

1.2.2 Läsbarhet och underhållbarhet

Det gamla systemet hade få personer som skötte uppdateringar och dessa hade god kännedom om miljön och verktyget (MS Access). En viktig målsättning med det nya systemet var att skapa ett intuitivt användargränssnitt där man har god överblick över datastrukturen och kan få ut sammanfattande information utan att direkt behöva gå in i databasen.

Man skall alltså med kortare inskolning kunna upprätthålla data på olika orter och ändringar skall kunna spåras. Man skall kunna se vem som har gjort en ändring och när samt lätt kunna få en överblick över ändringshistoriken.

1.2.3 Flexibilitet

Önskemål om ett flexibelt system framfördes och nedan presenteras de viktigaste punkterna beträffande detta.

I det gamla systemet, där man direkt modifierade MS Access-objekt kunde man lätt lägga till kolumner till tabeller. När man övergår till en SQL-databas med en separat användarapplikation är det inte lika enkelt längre. Önskemålet var från huvudanvändarna att denna egenskap skulle finnas tillgänglig i den nya

applikationen. Detta kan innebära en avsevärd extra utveckling eftersom filtrering måste ta detta i beaktande samt vyer tillgängliga för externa system.

En viktig funktion för användarapplikationen är naturligtvis en mångsidig filtreringsfunktionalitet. Databasen innehåller nästan 50 tabeller och vissa tabeller närmare 100 fält. En generisk filtreringsfunktionalitet skulle vara önskvärd för att reducera utvecklingsarbetet och få det mer anpassningsbart då nya fält och tabeller kommer till i framtiden.

1.3 Upplägg

I kapitel 1 presenteras bakgrund, problem och krav på systemet.

I kapitel 2 analyseras olika tekniker, hur tillämpliga de är för detta projekt samt en design på hög nivå presenteras för det fortsatta arbetet.

I kapitel 3 prestenteras med exempel hur nyckelfunktioner har realiserats. I detta verk fokuseras på hur generisk funktionalitet har åstadkommits för att göra programmet lättare att underhålla och för att minska tiden för utvecklingsarbetet. Framför allt två tekniker inom .NET-ramverket har använts nämligen Reflection och Generics. Hur nya revisioner skapas, hur nya fält tas i bruk dynamiskt och hur filtering och sortering utförs med generiska metoder presenteras.

I kapitel 4 presenteras kort hur vissa problem för att presentera data för externa system har gjorts. Programmet genererar skript för uppdatering av vyer i databasen för att kunna presentera nya i bruktagna fält med deras nya namn.

Datasäkerheten med användarbehörigheter på olika nivåer presenteras kort i kapitlet 5. Sist visas i kapitel 6 Framåtblick, hur vissa metoder kunde rationaliseras med ny teknik som tillkommit nyligen.

2 VAL AV RAMVERK OCH DESIGN PÅ HÖG NIVÅ

Olika alternativ dryftades, den första iden var att göra en Windows-applikation eller alternativt en modernare variant d.v.s. WPF (Windows Presentation Foundation) med klientprogram installerade i användarnas datorer. Motivet var att applikationen bara används inom företaget. Vidare kan man åstadkomma ett mer avancerat och responsivt användargränssnitt med denna teknik. Beställaren gjorde dock i ett tidigt skede klart att de vill ha en Web-applikation eftersom erfarenheterna visat att det kräver betydligt mindre ansträngningar att underhålla, konfigurera och uppdatera.

2.1 Val av ramverk

Beställaren har omfattande Microsoft-licenser och infrastruktur och kunnande så det stod också klart att systemet skall bygga på Microsoft-teknik d.v.s. .NET. Man kan programmera .NET applikationer i olika språk, tillbuds finns Visual Basic.NET, C# m.fl. Det dominerande språket är C# och det är också valet för denna applikation p.g.a. god dokumentation och det är det språk som är bäst bekant för utvecklaren (undertecknad). Syntaxen hos C# baseras på C-språken och påminner kanske mest om Java. Inom .NET finns sedan olika ramverk (frameworks) tillgängliga för olika typer av applikatoner. Tillbudsstående alternativ för web-applikationer var det mest etablerade ASP.NET WEBFORMS, Silverlight och det senaste teknologin ASP.NET MVC. Nedan ges en kort beskrivning av dessa ramverk.

2.1.1 Silverlight

Silverlight är namnet på en Microsoft .NET-baserad teknologiplattform som möjliggör nästa generationens mediarika, interaktiva RIAs (Business Rich Internet Applications). Det är en teknik som kan användas på olika plattformar och webbläsare på ett kostnadseffektivt sätt.

Alla versioner av Silverlight baseras på WPF (Windows Presentation Foundation), vilken är ett ramverk byggt för att integrera audio, video, bilder och data. Kärnan i Silverlight är det XML-baserade deklarativa språket XAML (Extensible Application

Markup Language). XAML möliggör en separering av användargränssitt och egentlig kod och är en naturlig utvidgning av teknologier som redan finns. (Goda, 2010, s. 15)

Silverlight presenterades för beställaren som en intressant teknologi att basera användargränssnittet på, men beställaren ville inte satsa på en så pass ny teknik, vilken eventuellt inte nått mognadsskedet ännu och sålunda kan innebära oförutsedda komplikationer.

2.1.2 ASP.NET

Den första versionen av .NET presenterades redan för ca. 10 år sedan och var starten på en radikalt ny rikting inom mjukvarudesign. ASP.NET var tekniken för att bygga websites inom .NET och idag är ASP.NET populärare än någonsin. Även om basfunktionerna praktiskt taget är de samma som för 10 år sedan har Microsoft lagt till nya egenskaper och kodabstraktioner på högre nivå. De har också introducerat en ny riktning som konkurrerar med traditionell ASP.NET (ASP.NET WEBFORMS) som kallas ASP.NET MVC (MacDonald, Freeman, & Szpuszta, 2010, s. 20).

På basen av erfarenheter from traditionell ASP.NET och studier av ASP.NET MVC beslöts att göra utvecklingen med detta ramverk. Det finns många orsaker och här redogörs kort för detta val. Framför allt är det ett ramverk där man har kontroll över det mesta, framför allt genererar det en ren html och upplägget är sådant att alla delar går att testa. Det fungerar även bra tillsammans med klientskript (Galloway;Haack;Hanselman;Guthrie;& Conery, 2010).

Under ASP.NET MVC presenteras här kort JQuery fastän JQuery är ett fritt stående separat opensource ramverk. Det hör till ovanligheterna att sådana delar inkluderas som standard till en Microsoft-modell, vilket det gör. JQuery är en samling funktioner baserade på JavaScript och använder CSS (Cascading Style Sheets) som identifierare. Det har ibland beskrivits som den största gåvan någonsin till Web-utvecklarna. JQuery fungerar väl ihop med ASP.NET MVC, vilket också javascript direkt och andra klientskript gör och bl.a. det gör ASP.NET MVC till ett intressant ramverk. ASP.NET MVC genererar nämligen tämligen ren html-kod utan genererade kontroll-

id:n etc. I denna applikation används JQuery i någon utsträckning, bl.a. datum-kontroller, design av sidor och vissa visuella egenskaper. Detta arbete har emellertid andra fokusområden, så JQuery presenteras endast kort här som komplement till ASP.NET MVC presentationen eftersom det är ett viktigt argument för valet av denna teknologi.

2.2 Databas och layout

Det stod klart från början att databasen skulle vara en SQL-server databas. Det är det naturliga valet för en .NET applikation eftersom den är designad speciellt för .NET. Det är dock inte det enda möjliga alternativet, även Oracle och andra kan väl användas med .NET. Emellertid har beställaren redan befintliga SQL-server instanser till förfogande och sålunda var det inte någon diskussion om detta val.

Layouten skulle i stort vara den samma som den tidigare Access-versionen med den största skillnaden att data inte raderas utan ändringar görs med att skapa nya versioner eller revisioner.

En intressant funktion som finns tillgänglig i SQL-SERVER 2008 som kallas Change Data Capture (CDC) undersöktes som en tänkbar teknik att använda för att upprätthålla ändringshistorik utan att man själv behöver skapa triggers eller queries. CDC har minimal inverkan på snabbheten, den fungerar asynkront. Ändringarna (UPDATES, INSERTS, DELETES) sparas i separata tabeller (Sack, 2008). En ingående presentation av CDC görs inte här, jag valde att inte använda tekniken för som tidigare nämnts betyder en ändring, eller ny revision rättare sagt, att nya revisioner även måste skapas i andra tabeller. Ibland kan det bli fråga om en insättning av många poster, i fler tabeller som är hierarkiskt kopplade. En huvudkonfiguration (Engine_Configuration) består av fler underkonfigurationer, om en del ändras, ändras även helheten. Ett medvetet val gjordes alltså att avstå från denna teknik och i stället bygga denna funktion med .NET-kod istället för i databasen.

2.3 Generisk funktionalitet

Eftersom det finns många tabeller och funktionerna är tämligen lika som behövs beträffande alla tabeller skulle det vara önskvärt att en stor del av funktionerna kan vara generiska, d.v.s. de går att tillämpas på olika objekt. Som tidigare nämnts finns det nästan 50 tabeller och i vissa tabeller närmare 100 fält. Att göra en filtreringsfunktion där varje fält tas separat i beaktande skulle vara väldigt arbetsdrygt. Dessutom kan man räkna med att miljön förändras kontinuerligt, nya tabeller och fält kommer till.

Två tekniker som speciellt är av intresse för att åstadkomma detta presenteras kort nedan d.v.s. .NET-generics och .NET-Reflection.

2.3.1 .NET Generics

Generisk programmering är en typ av programmering där algoritmerna är skrivna så att typerna de hanterar kan bestämmas senare som parametrar när de skapas. Generisk programmering kan ha olika betydelse beroende på vilket programmeringsspråk som är i fråga, men här presenteras kort vad det innebär inom .NET med exempel i C#.

När man skapar behållare för objekt är det möjligt att skriva specifika implementeringar för varje datatyp som den kommer att innehålla, även om koden är praktiskt taget identisk förutom datatyperna. Detta duplicering kan undvikas genom användande av Generics, se Figur 2.

```
using System;

class Sample {
    public static void Main() {
        int[] array = { 0, 1, 2, 3 };
        MakeAtLeast<int>(array, 2); // Change array to { 2, 2, 2, 3 }
        foreach (int i in array)
            Console.WriteLine(i); // Print results.
        Console.ReadKey(true);
    }

    public static void MakeAtLeast<T>(T[] list, T lowest) where T : IComparable<T> {
        for (int i = 0; i < list.Length; i++)
            if (list[i].CompareTo(lowest) < 0)
                list[i] = lowest;
    }
}
```

Figur 2 Expempel på en generisk funktion

Ovan T är en platshållare för vilkensomhelst typ som implementerar interfacet IComparable och typen bestäms vid där objektet skapas eller i detta fall när funktionen anropas.

2.3.2 .NET Reflection

Inom .NET betyder Reflection processen där man får fram information om en typ (Type) under programexekvering. Man kan t.ex. få fram programmatiskt alla typer som finns i en programenhet som kallas Assembly, samt typens metoder, fält, egenskaper, händelser (events), metoder. Man kan även dynamiskt få fram interface som typen implementerar, metodernas parametrar och andra relaterade detaljer såsom basklasser, namnrymdinformation (namespace), manifest data etc. (MacDonald, Freeman, & Szpuszta, 2010).

Tabell 1 Typer i Reflection

Type	Betydelse
Assembly	Denna abstrakta klass innehåller statiska metoder som tillåter att ladda (load), undersöka och manipulera en Assembly.

AssemblyName	Med denna klass kan man undersöka detaljer bakom en Assemblys itentitet, kulturinformation etc.
EventInfo	Denna abstrakta klass innehåller information om en viss händelse (event)
FieldInfo	Denna abstrakta klass håller information för ett visst fält
MemberInfo	Denna abstrakta klass innehåller information om en viss metod
Module	Denna abtrakta klass ger access till en viss modul i en multi-fil assembly
ParameterInfo	Denna klass innehåller information om en viss parameter
PropertyInfo	Denna abstrakta klass innehåller information om en viss egenskap (property).

I Tabell 1 listas några typer inom Reflection med en beskrivning. I utvecklingen av denna applikation är det främst klassen PropertyInfo som är av intresse och kommer att användas. Vi behöver klasser som under programexekvering kan få fram egenskaper eller Properties för ett okänt objekt samt få fram värdet för egenskapen och även uppdatera värdet.

På basen av tidigare beskrivning av Generics skall vi här med ett exempel visa hur vi sätter värdet på en egenskap i en generisk metod.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Reflection;
6
7  namespace LinqToSql.Models
8  {
9      public class GenericsAndReflection<T> where T : class
10     {
11         List<string> GetPropertyNames(T item)
12         {
13             PropertyInfo[] pis = typeof(T).GetProperties();
14             List<string> nameList = new List<string>();
15             foreach (PropertyInfo pi in pis)
16             {
17                 nameList.Add(pi.Name);
18             }
19             return nameList;
20         }
21
22         void SetProperty(T item, string propertyName, object value)
23         {
24             PropertyInfo pi;
25             pi = typeof(T).GetProperty(propertyName);
26             pi.SetValue(item, value, null);
27         }
28     }
29 }
```

Figur 3 Exempel på Generics och Reflection

I Figur 3 Exempel på Generics och Reflection ett enkelt exempel på en klass som använder sig av Generics och Reflection. Klassen innehåller två metoder varav den första GetPropertyNames returnerar en lista på namnen på egenskaper som gällande klass besitter (T). Som tidigare nämnts, bestäms klassen vid instansiering av GenericsAndReflection t.ex. för string:

```
GenericsAndReflection<string> myGenRef = new GenericsAndReflection<string>();
```

På rad 13 skapar vi en array av PropertyInfo via reflection-metoden GetProperties() på T:s typ, vilken vi får genom metoden typeof(T). Därefter görs en foreach-loop (rad 15-18) där namnet från varje PropertyInfo läggs till listan på teckensträngar (List<string>). Efter loopen returneras denna lista.

Den andra metoden i Figur 3 Exempel på Generics och Reflection SetProperty visar hur värdet på en egenskap sätts. Metoden har som argument objektet vars egenskap skall sättar (typen på objektet okänt), egenskapens namn samt det värde som egenskapen skall få. Först får vi en referens till en PropertyInfo genom metoden GetProperty (rad 25), vilken har som argument egenskapens namn och sedan sätts egenskapen med metoden SetValue (rad 26), vilken tar som argument objektet vars egenskap skall sättas samt det nya värdet. Metoden har också som argument ett index, men det är inte relevant i detta exempel och ges värdet null.

3 UTFÖRANDE

3.1 Arkitektur

3.1.1 Datamigration

Det befintliga gamla systemet är i bruk kontinuerligt och övergången till det nya måste ske på en gång. Man måste utgå från att datamigrationen måste göras fler gånger eftersom man inte kan räkna med att allt fungerar som det skall genast. Det finns färdiga lösningar för överföring av data från Access till Sql-server. Bl.a. SQL-servers SSIS (Sql Server Integration Services) är ett omfattande program för transferering av data och mycket mer. Det är emellertid ett tidskrävande projekt att sätta sig in i dess funktioner. Eftersom den nya databasen är modifierad från den gamla, sammansatt primärnyckel (Id och Revisionsnummer) samt vissa tabeller sammanslagna där förhållandet var 1-1 valdes att göra ett separat .NET-projekt för migrationen. Prestanda är inte av primär betydelse, det viktigaste är att det kan göras enkelt, processen kan göras på nytt vid behov och det inte kräver vidlyftig insats i arbetstid.

3.1.2 N-tier och MVC

Målet är att skapa en sk. Multi-tier (N-Tier) applikation med vilket menas att programmet delas upp i olika ansvarsområden eller skikt. Det vanligaste är en indelning i tre skikt DAL (Data Access Layer) eller dataåtkomst, BLL (Business Logic Layer) eller själva logiken för applikationen samt UI (User Interface) d.v.s. användargränssnittet. I denna applikation görs likaså en indelning i olika skikt, anpassat efter möjligheterna. På grund av använda tekniker, följer inte skikten direkt projektindelningen men detta är en följd av ”recommended best practices” för ramverken. Nedan behandlas kort denna indelning och vilken teknik som används. Eftersom detta är en MVC (Model, View, Controller) applikation gör vi en skiktindelning enligt denna terminologi.

Det är lättast att börja med att klargöra vad Vyn (View) innebär. Det är själva sidorna som användaren ser. De skall innehålla så lite funktionalitet som möjligt. I ASP.NET MVC är det rekommenderat att ha kod i sidorna men den skall vara till för själva presentationen av data t.ex. for-each loopar, val av färger etc. View-delen i MVC är entydigt en del av UI (User Interface) eller användargränssnittet.

Controller är den del som kontrollerar hur och vad som visas i vyn. Denna del, om man implementerat MVC enligt rekommendationerna, är ävenså en del av UI.

Modellen (Model i MVC) är då allt annat det vill säga det innefattar BLL (Business Logic Layer) och DAL (Data Access Layer) och eventuella andra skikt. Model View Controller-modellen tar inte ställning till hur detta implementeras men här har valts att dela upp den i en DAL och BLL. Detta presenteras närmare nedan.

Object-Relational Mapping (ORM, O/RM eller O/R Mapping) inom dataprogrammering innebär en teknik för att konvertera inkompatibla datatyper i objekt-orienterade programmeringsspråk. Det här skapar i själva verket en ”virtuell databas” som direkt kan användas från programmeringsspråket. Det finns både gratis versioner och kommersiella paket tillgängliga för ändamålet. Att använda en ORM teknik betyder också att mängden kod kan reduceras och följaktligen arbetsmängden. För .NET finns olika ORM-tekniker men för denna applikation valdes Linq-To-Sql, vilken redan är en etablerad teknik och torde lämpas sig bra för denna applikation. Linq-To-Sql presenteras kort nedan.

Först klargör vi vad LINQ står för. LINQ (Language Integrated Query) är ett språk som är integrerat i .NET för att göra queries (frågor) på olika data. Tidigare för att göra queries, måste man skriva det som en teckensträng och kunde således inte få hjälp av programmeringsverktyget för att kontrollera syntax, intellisense etc.

LinqToSql är således Linq som tillämpas på SQL-data. Man skapar ett skilt lager (layer) vari man genererar object som motsvarar Tabellerna i databasen. Dessa objekt kan sedan användas som andra .NET-objekt och även funktioner för att hämta, ändra,

sätta till och radera data generas av ramverket. Mycket utvecklingstid kan sparas och fel undviks eftersom felstavningar etc. upptäcks redan under kompilationen. I denna applikation valdes att använda Linq-To-Sql som ORM-verktyg. Detta skikt innehåller all funktionalitet för hämtande av data samt uppdatering etc. men dessa funktioner kallas inte direkt från BLL (Business Logic Layer) utan vi skapar ett separat projekt som "mellanhand" så att man vid behov kan ändra datalagring utan att BLL (och följaktligen även UI) inte har någon vetskap om hur detta sker. Detta projekt använder sig långtgående av Generics och Reflection för att kunna skapa generisk funktionalitet som beskrivits ovan. Allt detta är till för att åstadkomma så lite beroenden som möjligt. (Mehta, 2008)

I och med att LinqToSql skapar själva huvudobjekten, det vill säga objekt som motsvarar tabellerna med deras relationer som genererats från databasen är detta skikt inte ett rent DAL utan dessa genererade objekt hör i själva verket till BLL (Business Logic Layer). Det är inte ett problem, men man bör vara uppmärksam på detta faktum.

Om man undantar själva dataobjekten är all businesslogik i projektet WebUI. Som namnet säger är detta UI, User Interface eller användargränssnittet men det innehåller också businesslogik och dessa delar finns i olika mappar indelade enligt MVC (Model-View-Controller) principen. Modellen är de genererade dataobjekten och finns i LinqToSql-projektet. I detta projekt är huvudsyftet att enbart spara data med ändringshistorik. Själva objekten har inte något beteende, eller närmare bestämt är beteendet hur revisioner skapas i en kaskad beroende på deras relationer.

3.2 Revisionshantering

Den viktigaste nya egenskapen för denna applikation var att sätta till ett revisionshanteringssystem. Poster ändras eller raderas i regel inte utan istället skapas en ny version eller revision av en post. Som tidigare nämns finns det en del problematik i detta eftersom det inte är fråga om isolerade tabeller utan en helhet kan bestå av fler tabeller. För det första får alla tabeller en sammansatt primärnyckel som

består av Id samt RevisionsNr. Om en befintligt objekt uppdateras skapas en ny revision. Vi vill ännu ha en indelning i mindre (minor) och större (major) revision. Lösningen för denna indelning i minor och major är enkel, för varje ny Minor-Revision ökas revisionsnumret med 1 och för varje ny Major-Revision ges revisionsnumret nästa hela hundra tal. Om en revisionsnummer således är 311 kan man splitta upp det i Major-Revision 2 och Minor-Revision 11 genom en enkel modulus-funktion.

3.2.1 Nya revisioner

I Figur 4 visas hur en normal read-only sida ser ut för ett valt objekt. Längst uppe är en rad med knappar som visar olika alternativ för revidering, nya revisioner etc.

Edit last revision | New minor revision from this revision | New major revision from this revision | Create new (new id) based on this | Create new emtpy item

Back to list

Revision

ID:	2
RevisionNr	Major: 0 Minor: 1
Revision history:	1 Get revision Change history
Created:	26.6.2000 0:00:00
Last updated by:	Initial data transfer
Last updated time:	1.4.2003 0:00:00
Revision comment manual:	
Revision comment generated:	Initial data transfer

Air Mass Flow

Description:	W4L20 B CS 750	
Air Mass Flow 100:	0,99	Combustion air mass flow at 100% nominal output, kg/s
Air Mass Flow 90:		Combustion air mass flow at 90% output, kg/s
Air Mass Flow 85:	0,86	Combustion air mass flow at 85% output, kg/s
Air Mass Flow 75:	0,79	Combustion air mass flow at 75% output, kg/s

Figur 4 Read sidan för ett objekt

Den första ”Edit Last Revision” innebär att man inte skapar en ny revision utan editerar den senaste. Det är ju tvärtemot specifikationen, men idén är att man skall kunna göra revideringar under en tidsperiod från att en revision eller ett nytt id skapats. Längden på perioden som man kan editera utan att skapa en ny revision kan bestämmas i konfigurationen och är initialt satt till två timmar eller 120 minuter. Nedan visas ett utdrag från web.config-filen.

```
   <!--Within the timespan TimeInMinutesToEdit configs can be edited without creating new revisions-->
   <setting name="TimeInMinutesToEdit" serializeAs="String">
    <value>120</value>
   </setting>
```

Det andra och tredje alternativet ”New minor revision from this revision” och ”New major revision from this revision” innebär naturligtvis att man skapar en ny minor eller major revision”. Dessa alternativ fokuseras på nedan. Det nästsista alternativet ”Create new (new id) based on this” betyder att man inte gör en ny revision utan ett nytt objekt med det aktuella objektets värden som default. Det sista alternativet ”Create new empty item” skapar ett nytt objekt med tomma värden initialt.

I nästa steg följer vi med händelseförloppet när användaren klickar på ”New revision based on this revision”.

```
299         [AcceptVerbs(HttpVerbs.Get)]
300         public ActionResult CA_DataEdit(int id, int page, CreateEntityOption createEntityOption, string revisionAsBaseForNew)
301         {
302             if (!Authorization.Instance.AllowCommonEdit()) return RedirectToAction("NoRights", "Home");
303             RenewRepository();
304             CA_Data_TBL newItem = null;
305             newItem = RepositoryUtil.GetEntityByOption<CA_Data_TBL>(id, revisionAsBaseForNew, createEntityOption, repository.DContext, Authorization
306 
307             string message = null;
308             if (createEntityOption == CreateEntityOption.EditNotNewRevision && !common.AllowedToEditByTime(newItem.LastUpdatedTime, ref message))
309             {
310                 TempData["message"] = message;
311                 return RedirectToAction("CA_DataRead", new { id = id, revision = revisionAsBaseForNew, page = page });
312             }
313 
314             SetEditViewData(newItem, id.ToString(), newItem.CA_Data_ID.ToString(), revisionAsBaseForNew, page.ToString());
315 
316             return View(newItem);
317         }
```

Figur 5 Edit sidans controller och http.get-funktionen

På rad 305 i koden i Figur 5 visas hur ett nytt objekt (newItem) skapas. Beroende på parametern ”createEntityOption” skapas en ny minor-revision eller major-revision etc. Vi följer denna kedja och analyserar vidare den statiska GetEntityByOption-metoden i klassen RepositoryUtil (Figur 6).

```
324        public static T GetEntityByOption<T>(int id, string revisionAsBaseForNew, CreateEntityOption createEntityOption,
325            DataClassesDataContext dcontext, string userName) where T : class
326        {
327            GenericRepository<T> repository = new GenericRepository<T>(dcontext, userName);
328            T newItem = null;
329
330            if (createEntityOption == CreateEntityOption.NewId)
331            {
332                T basetype = repository.GetSpecificRevision(id, int.Parse(revisionAsBaseForNew));
333                newItem = repository.GetNewEntityFromBase(basetype);
334            }
335            if (createEntityOption == CreateEntityOption.NewIdEmpty)
336            {
337                newItem = repository.GetNewEmtpyItemWithIdAndRevisionSet();
338            }
339            else if (createEntityOption == CreateEntityOption.EditNotNewRevision)
340            {
341                newItem = repository.GetEntityLastRevision(id);
342            }
343            else if (createEntityOption == CreateEntityOption.NewMinorRevision)
344            {
345                newItem = repository.GetNewRevisionOfEntity(id, int.Parse(revisionAsBaseForNew), false);
346            }
347            else if (createEntityOption == CreateEntityOption.NewMajorRevision)
348            {
349                newItem = repository.GetNewRevisionOfEntity(id, int.Parse(revisionAsBaseForNew), true);
350            }
351
352            return newItem;
353        }
```

Figur 6 Metoden GetEntityByOption

Denna metod riktar processen vidare till andra metoder beroende på argumentet ”createEntityOption”. I vårt fall är det på raden 343 som villkorssatsen stämmer överens med vårt val, d.v.s. vi går vidare till metoden repository.GetNewRevisionOfEntity. Argumentet är objektets id samt en boolean som bestämmer om det är frågan om en major-revision (=true). I vårt fall är det en minor alltså är argumentet false (se rad. 345).

Som följade analyserar vi GetNewRevisionOfEntity-metoden (Figur 7).

```
133        public T GetNewRevisionOfEntity(int id, int revisionBase, bool major)
134        {
135            object objectToBeCopied = GetSpecificRevision(id, revisionBase);
136            T clone = (T)idAndVersionUtil.GetNewRevisionRecord(objectToBeCopied, major, true);
137            return clone;
138        }
```

Figur 7 GetNewRevisionOfEntity

Denna metod gör egentligen ingenting förutom att fungera som mellanhand. Den dirigerar commandot till en annan klass och metod: idAndVersionUtil.GetNewRevisionRecord. Det är därför att IdAndVersionUtil finns i en annan assembly, dvs vårt DAL (Data Access Layer). Mellanhandfunktionen

hjälper alltså till att hålla våra skikt (layers) separerade så att beroendena är så få som möjligt. Vi går följaktligen som nästa till idAndVersionUtil.GetNewRevisionRecord.

```
52        public object GetNewRevisionRecord(object baseRevisionObj, bool majorRevision, bool updateContainingObjects)
53        {
54            object newRevisionObj;
55            newRevisionObj = GetCopyOfEntity(baseRevisionObj);
56            object newRevisionValue;
57
58            int maxRevisionValue = GetMaxRevisionForItem(baseRevisionObj);
59            if (majorRevision) newRevisionValue = GetNextMajorRevisionNr(maxRevisionValue);
60            else newRevisionValue = GetNextMinorRevisionNr(maxRevisionValue);
61
62            PropertyInfo pi = baseRevisionObj.GetType().GetProperty("RevisionNr");
63            pi.SetValue(newRevisionObj, newRevisionValue, null);
64
65            if (updateContainingObjects)
66            {
67                CreateLinkedRevision(baseRevisionObj, newRevisionObj);
68            }
69            SetUpdatedByAndTime(newRevisionObj, _currentUser, false);
70
71            return newRevisionObj;
72        }
```

Figur 8 GetNewRevisionRecord i IdAndRevisionUtil

Denna funktion (Figur 8) är ganska enkel, på rad 55 anropas en funktion GetCopyOfEntity som skapar en kopia av argumentet ”baseRevisionObj”. Kopia är egentligen fel uttryck, det är ett helt nytt objekt med olika referens, men dess värden är de samma. Vidare ges denna ”kopia” ett nytt revisionsnummer på basen av argumentet ”majorRevision” som är true om det är major och false om det är minor. Tidpunkt för uppdateringen samt vem som utför den uppdateras även. Separata metoder sköter detta men vi går inte närmare in hur de fungerar här. Det som här är intressant är raderna 65 till 68. Om argumentet ”updateContainingObjects” är true anropas metoden ”CreateLinkedRevision”. Det är här som en kedja av nya revisioner skapas beroende på relationerna i databasen, vilket behandlas i nästa avsnitt.

3.2.2 Revision-Cascading

Det första försöket att skapa en s.k. ”Revision-Cascading” var en helt automatisk funktion som med hjälp av reflection skulle skapa nya revisioner av länkade objekt så att principen ”om en del ändras, ändras även helheten”. Databasen består av en ”main Engine-configuration” och ”Sub-configurations”. En ”sub-configuration” kan vara beståndsdel till flere ”main-configurations”. Det är alltid fråga om 1-n kardinalitet. Försöket att skapa denna automatiska funktion lyckades såtillvida att det fungerade,

men misslyckades p.g.a. att det tog alltför lång tid. Designen ändrades så att en visst mått av kodning per objekt gjordes men huvuddelen är ändå generisk. Det är heller inget problem att programmet är hårdkodat per tabell, inga anspråk på dynamiska tabelltillägg har gjorts.

```
private void CreateLinkedRevision(object baseObj, object newObj)
{
    Engine_Configuration_TBL

    Release Status
    Emission_Optimisation_TBL
    Engine Types

    Product_Name_TBL

    CA_Data_TBL
    CA_Notes_TBL

    EG_Data_TBL
    EG_Notes_TBL

    Pump_Data_TBL

    HT_Data_TBL

    Start_Data_TBL
    Start_Notes_TBL

    Dynamic_Data_TBL

    Feature_Data_TBL
    Feature_TBL

    Output_Data_TBL
```

class System.Object
Supports all classes in the .NET Framew
the root of the type hierarchy.

Figur 9 CreateLinkedRevision-metoden (början)

I Figur 9 visas början på metoden CreateListedRevision. I själva verket är koden komprimerad i ”regions”, men med figuren visas att alla objekt som har revisionsfunktionalitet har en egen region. De är alla lika förutom att de behandlar olika klasser. Vi öppnar en region, t.ex. CA_Data_TBL.

```
166   Product_Name_TBL
178
179       #region CA_Data_TBL
180       //----------------------------------------------------------------------------------------------------
181       if (baseObj.GetType() == typeof(CA_Data_TBL))
182       {
183           CA_Data_TBL item = (CA_Data_TBL)baseObj;
184           CA_Data_TBL itemNew = (CA_Data_TBL)newObj;
185           List<Engine_Configuration_TBL> refList = item.Engine_Configuration_TBLs.ToList();
186
187           CreateLinkedRevision<CA_Data_TBL, Engine_Configuration_TBL>(item, itemNew, refList, "Engine_ID", "CA_Data_Revi
188       }
189       //.....................................................................................................
190       #endregion
191   CA_Notes_TBL
203
204   EG_Data_TBL
```

Figur 10 CA_Data -regionen i CreateListedRevision

Vad vi ser i CA_Data_TBL sektionen (Figur 10) är att ifall objektet är av typen CA_Data_TBL , så gör vi en cast till denna typ från den allmänna objekt-typen (rad 181-184). Sålunda kan vi lätt och snabbt få en referens till relaterade objekt (rad 185).

Vi ser vidare i dess signatur att metoden är ”overloaded” inte rekursiv som man först skulle tro, det vill säga den anropar inte sig själv (som tidigare nämnts gjordes första versionen av ”Revision Cascading” helt automatisk genom Reflection, till stor del genom att analysera .NET attribut som LinqToSql skapar. Denna version fungerade, men prestandan var för dålig). Det är som synes (rad 187) en generisk funktion där typerna för relationen ingår i argumenten. De sista teckensträngargumenten är namnen på id och den främmande nyckeln för revisionsnumret. Denna funktion (Figur 9 och Figur 10) specificerar alltså typerna och med dem hörande argument för att skickas till den generiska funktionen vilken visas nedan med förklaringar och kommentarer i textrutor inne i koden. Kommentarerna behandlar koden ovanför textrutan.

```
        private void CreateLinkedRevision<T, TL>(T baseObj, T newObj, IEnumerable<TL>
refList
            , string TLidName, string foreignRevisionKeyName, string revComment)
```

Medodens argument: inom <T, TL> kommer typerna, T = typen för den nya revisionen som skapats, TL typen för den relaterade revisionen som skall skapas. Inom parententeserna: baseObj = basen för den nya revisionen, newObj = den nya revisionen, refList = ett obj. Innehållande en lista på relaterade objekt, TLidName = namnet på id-fältet på det relaterade objektet, foreigneRevisionKeyName = fältnamet på det relaterade objektets främmande nycket, revComment = en kommentar som kommer för den relaterade revisionen.

```
        {
            PropertyInfo TrevProp = typeof(T).GetProperty("RevisionNr");
            PropertyInfo TLrevProp = typeof(TL).GetProperty("RevisionNr");
            PropertyInfo TLidProp = typeof(TL).GetProperty(TLidName);
            PropertyInfo TLforeignRevisionKeyProp = typeof(TL)
```

```
                .GetProperty(foreignRevisionKeyName);
            PropertyInfo TLrevComment = 
typeof(TL).GetProperty("RevisionCommentGenerated");
```

Här skapas PropertyInfo objekt för de fält som behövs.

```
               //remove isArchived=true from refList if main-config
            if (typeof(TL) == typeof(Engine_Configuration_TBL))
            {
                PropertyInfo isArchivedProp = typeof(Engine_Configuration_TBL)
                    .GetProperty("IsArchived");
                refList = refList.Where(a => (bool?)isArchivedProp.GetValue(a, null) 
!= true);
            }
```

Här undersöks om fältet isArchived är lika med true. Om så är fallet skall inte en ny revision skapas. IsArchived är i normala fall true om maskintypen har status ”Discontinued”.

```
            IEnumerable<IGrouping<int, TL>> idGroupSet = refList
                .GroupBy(a => ((int)TLidProp.GetValue(a, null)));
            List<TL> lastRevisionRefList = new List<TL>();
            foreach (IGrouping<int, TL> idGroup in idGroupSet)
            {
                if (idGroup.Count() > 0)
                {
                    int maxRev = idGroup.Max(a => (int)TLrevProp.GetValue(a, null));
                    lastRevisionRefList.Add(idGroup
                        .Where(a => (int)TLrevProp.GetValue(a, null) == 
maxRev).Single());
                }
            }
```

I detta avsnitt filtreras refList så att endast de sista revisionsnumren tas med för varje id. Här används Linqs group by funktion för att skapa en grouping och för denna grouping tas max-värdet av revisionsnumret med hjälp av Reflection.

```
            IEnumerable<TL> allList = GetAllItemsFromOne<TL>(typeof(TL));
```

I denna rad skapas ett IEnumerable-objekt innehållande raderna för aktuella tabell. En annan funktion anropas för detta och det görs genom att använda datacontexten. Detta kräver en mer ingående presentation av LinqToSql vilket inte behandlas i denna uppsats.

```
            foreach (TL ec in lastRevisionRefList)
            {
                int nextRevisionForId = ((int)allList
                    .Where(a => (int)TLidProp
                        .GetValue(ec, null) == (int)TLidProp
                        .GetValue(a, null))
                        .Max(a => TLrevProp.GetValue(a, null))) + 1;
                TL newEc = (TL)GetCopyOfEntity(ec);
                TLforeignRevisionKeyProp.SetValue(newEc, TrevProp.GetValue(newObj, 
null), null);
                TLrevProp.SetValue(newEc, nextRevisionForId, null);
                TLrevComment.SetValue(newEc, revComment, null);
                SetUpdatedByAndTime(newEc, _currentUser, false);
                DoInsertOnSubmit(newEc);
```

Här itereras den filtrerade lista som innehåller de sista revisionerna av den relaterade tabellen. En nytt objekt med samma värden skapas förutom den främmande nyckeln för det relaterade objektets revisionsnummer, det egna revisionsnumret samt en kommentar om den nya revisionen.

```
                CreateLinkedRevision(ec, newEc);
```

Till slut anropas den första CreateLinkedRevision igen för att göra om proceduren för eventuella vidare relationer.

}
}

Ovan har händelseförloppet för hur nya revisioner skapas med ”Revision-Cascading”.

3.2.3 Kontroll av historiska data

En del av specifikationen var att det skulle finnas möjlighet att få fram ändringshistoriken på ett åskådligt sätt. Det kan man göra på två sätt, det första är i ett objekts read-only sida, det finns en dropdown som innehåller alla revisionsnummer (Figur 11).

Revision

ID:	**2**
RevisionNr	Major: **2** Minor: **1**
Revision history:	201 Get revision Change history
Created:	**26.6.2000 0:00:00**
Last updated by:	**Peter-Asus\Peter**
Last updated time:	**21.10.2010 11:08:04**
Revision comment manual:	**test, ny minor-revision**
Revision comment generated:	

Figur 11 Revisionsdata för ett valt objekt

På rad 2 RevisionNr ser vi att numret som finns i dropdown just direkt nedanför, 201, är uppdelat i Major och Minor. För klarhetens skull påpekas här att revisionsnumret endast är en siffra d.v.s. 201. Som default visas alltid den sista revisionen, det vill säga den som gäller för tillfället. Om man väljer en annan revision i dropdown visas denna på samma sätt som nuvarande.

För att enklare kunna följa med ändringshistoriken kan man klicka på linken ”Change history”. Då kommer man till en helt annan vy som visas i Figur 12.

Revision change history of CA_Data id: 2

Show only modified fields   back

CA_Data_ID	RevisionNr	Description	Timestamp_Created	Timestamp_Changed	AirMassFlow_100	AirMassFlow_90	AirMassFlow_85	AirMassFlow_75	AirMassFlo
Rev: 1 2	Rev: 1 1	Rev: 1 W4L20 B CS 750	Rev: 1 26.6.2000 0:00:00	Rev: 1 1.4.2003 0:00:00	Rev: 1 0,99	Rev: 1 -	Rev: 1 0,86	Rev: 1 0,79	Rev: 1 0,57
CA_Data_ID Rev: 100 2	RevisionNr Rev: 100 100	Description Rev: 100 W4L20 B CS 750	Timestamp_Created Rev: 100 26.6.2000 0:00:00	Timestamp_Changed Rev: 100 21.10.2010 11:07:14	AirMassFlow_100 Rev: 100 0,99	AirMassFlow_90 Rev: 100 -	AirMassFlow_85 Rev: 100 0,86	AirMassFlow_75 Rev: 100 0,79	AirMassFlo Rev: 100 0,57
CA_Data_ID Rev: 200 2	RevisionNr Rev: 200 200	Description Rev: 200 W4L20 B CS 750	Timestamp_Created Rev: 200 26.6.2000 0:00:00	Timestamp_Changed Rev: 200 21.10.2010 11:07:34	AirMassFlow_100 Rev: 200 0,99	AirMassFlow_90 Rev: 200 -	AirMassFlow_85 Rev: 200 0,86	AirMassFlow_75 Rev: 200 0,79	AirMassFlo Rev: 200 0,57
CA_Data_ID Rev: 201 2	RevisionNr Rev: 201 201	Description Rev: 201 W4L20 B CS 750	Timestamp_Created Rev: 201 26.6.2000 0:00:00	Timestamp_Changed Rev: 201 21.10.2010 11:08:04	AirMassFlow_100 Rev: 201 0,99	AirMassFlow_90 Rev: 201 -	AirMassFlow_85 Rev: 201 0,86	AirMassFlow_75 Rev: 201 0,79	AirMassFlo Rev: 201 0,57

Figur 12 Revision change history-sidan för ett valt objekt

I varje ruta visas på första raden fältets namn, på den andra raden revisionsnumret och på den sista själva värdet för kolumnen. För detta exempel har endast kommentaren ändrats för detta objekt. Kommentarfältet finns längre till höger och om vi rullar till höger ser vi de ändrade kolumnerna (Figur 13).

e_ID	CA_Note_RevisionNr Rev: 1 1	CA_inactive Rev: 1 False	LastUpdatedBy Rev: 1 Initial data transfer	LastUpdatedTime Rev: 1 1.4.2003 0:00:00	RevisionCommentManual Rev: 1 -	RevisionCommentGenerated Rev: 1 Initial data transfer	s1 Re 1 -
e_ID 00	CA_Note_RevisionNr Rev: 100 -	CA_inactive Rev: 100 False	LastUpdatedBy Rev: 100 Peter-Asus\Peter	LastUpdatedTime Rev: 100 21.10.2010 11:07:00	RevisionCommentManual Rev: 100 test, ny minor-revision	RevisionCommentGenerated Rev: 100 -	s1 Re 10 -
e_ID 00	CA_Note_RevisionNr Rev: 200 -	CA_inactive Rev: 200 False	LastUpdatedBy Rev: 200 Peter-Asus\Peter	LastUpdatedTime Rev: 200 21.10.2010 11:08:00	RevisionCommentManual Rev: 200 test, ny major-revision	RevisionCommentGenerated Rev: 200 -	s1 Re 20 -
e_ID 01	CA_Note_RevisionNr Rev: 201 -	CA_inactive Rev: 201 False	LastUpdatedBy Rev: 201 Peter-Asus\Peter	LastUpdatedTime Rev: 201 21.10.2010 11:08:00	RevisionCommentManual Rev: 201 test, ny minor-revision	RevisionCommentGenerated Rev: 201 -	s1 Re 20 -

Figur 13 Revisionhistory med ändrade fält

Här ser vi att de kolumner som ändrats har orange bakgrund. Uppe på denna sida kan man även välja linken ”Show only modified fields” (Figur 12) och då visas endast de ändrade fältena (Figur 14).

Revision change history of CA_Data id: 2

Show all fields   back

CA_Note_RevisionNr Rev: 100 -	LastUpdatedBy Rev: 100 Peter-Asus\Peter	RevisionCommentManual Rev: 100 test, ny minor-revision	RevisionCommentGenerated Rev: 100 -
RevisionCommentManual Rev: 200 test, ny major-revision			
RevisionCommentManual Rev: 201 test, ny minor-revision			

Figur 14 Change History - Visa endast modifierade fält

Med denna vy får användaren presenterat endast vad som ändrat vid olika revisioner. Denna funktion är även så helt generisk och nedan presenteras kort hur detta gjorts. Reflection är igen nyckelkomponenten för denna funktion. Nedan visar List funktionen i RevisionHistoryController. Kommentarer och förklaringar har satts i textboxar vilka behandlar koden ovanför boxen.

```
        public ActionResult List(string typeName, int id, bool?
showOnlyChangedColumns)
        {
            string typeNameShort = Repository.RepositoryUtil
                .RemoveNamespaceFromObjectName(typeName).Replace("_TBL", "");
            ViewData["heading"] = "Revision change history of " + typeNameShort + "
id: " + id.ToString();

            ViewData["typeName"] = typeName;
            ViewData["id"] = id;
            ViewData["showOnlyChangedColumns"] = showOnlyChangedColumns ?? false;

            ModelData modelData = RepositoryUtil.ModelList
                .Where(a => a.ModelNameNoSpaces == typeNameShort).SingleOrDefault();

            ViewData["sourceControllerName"] = modelData != null ?
modelData.ControllerName : "";
            ViewData["sourceActionName"] = modelData != null ?
modelData.ReadItemActionName : "";
```

Ovan skapas olika ViewData-objekt bl.a. rubrik för sidan. Det som är av störst intresse är ModelData objektet strax nedan om mitten av avsnittet. Alla objekt har en motsvarande ModelData och det är ett custom-objekt som innehåller viktig information per objekt. Bl.a. primärnyckelns namn, Controllerns namn och namnet på vissa av dess metoder. Etc. Med hjälp av detta objekt kan många generiska funktioner åstadkommas.

```
            extraColumnList = extraColumnRep.table.Where(a => a.TableName ==
typeName).ToList();
            Type type = IdAndVersionUtil.GetTypeFromName(typeName);
            PropertyInfo piId, piRev;
            IdAndVersionUtil.GetPrimaryKeyColumnProperties(type, out piRev, out piId);
            List<object> list = (new IdAndVersionUtil(new DataClassesDataContext(),
Authorization.CurrentUserName)
                .GetListOfRevisionsForObject(type, id));

            List<RevisionHistoryRow> rows = new List<RevisionHistoryRow>();
            PropertyInfo[] props = type.GetProperties();
            List<PropertyInfo> propList = GetListOfColumnProperties(props);
```

Här får extra-kolumn map listan, typen, primärnyckeln, en lista på revisionerna och varje posts columner sparas i en lista på PropertyInfos. Olika funktioner på olika ställen används för att få dessa värden.

```
            object previousRow = null;

            foreach (object item in list)
            {
                RevisionHistoryRow r = new RevisionHistoryRow();
                r.Revision = (int)piRev.GetValue(item, null);
                foreach (PropertyInfo pi in propList)
                {
                    RevisionHistoryColumn col = new RevisionHistoryColumn();
                    col.ColumnName = GetColumnNameCheckExtraColums(pi.Name,
typeName);//pi.Name;

                    object val = pi.GetValue(item, null);
                    string valStr = ReturnToStringOrMinusIfNullOrEmpty(val);

                    object prevVal = null;
                    if (previousRow != null) prevVal = pi.GetValue(previousRow, null);
                    string prevValStr = ReturnToStringOrMinusIfNullOrEmpty(prevVal);

                    col.Value = valStr;
                    col.IsChanged = false;

                    if (previousRow != null &&
!ColumnExcludedFromChangeStatus(col.ColumnName))
                        col.IsChanged = valStr != prevValStr;

                    if (showOnlyChangedColumns == true)
                    {
                        if (col.IsChanged) r.Columns.Add(col);
                    }
                    else
                    {
                        r.Columns.Add(col);
                    }
                }
                previousRow = item;
                rows.Add(r);
```

Här sker en foreach-loop för varje post och för varje kolumn per post. Den föregående radens post sparas i ett separat variabel och kolumnnvärdena jämförs sedan. Om värdet är olika ges värdet isChanged = true.

```
            }
            return View(rows);
        }
```

3.3 Fälttillägg dynamiskt i tabeller

En önskan från användarna var möjligheten att kunna sätta till fält dynamiskt utan att behöva skriva om kod, kompilera etc. Man skulle kunna tänka sig att generera skript som modifierar databasen för detta ändamål. Det skulle fungera bra och används också i denna applikation, men inte för att modifiera tabeller utan vyer (views) i databasen. Problemet med att modifiera tabeller är att det inte finns något (lätt) sätt att automatiskt skapa om de genererade LinqToSql-klasserna d.v.s. .NET-objektena som skapats på basen av databasobjekten, i vårt fall tabellerna.

Lösningen var att lura användaren att tro att nya fält skapas. För varje tabell skapades 40 fält med samma namn, 20 st. av typen varchar(255) som döptes till s1, s2 .. s20 och 20 st. av typen real (flyttal). När man sedan så att säga skapar ett nytt fält, tar man i användning ett fält av dessa 40 och det den ett annat namn. I databasen är namnet fortfarande det samma men namnet mappas i en annan tabell. På detta sätt har man en reserv på 40 fält varefter man måste ändra i kod för att få fler.

Det här visade sig bli mycket mer arbetsdrygt eftersom det skapade mycket extra arbete

3.4 Filtrering och sortering generiskt

Den viktigaste funktionen för listvyn är att hjälpa användaren välja de poster (konfigurationer) han är intresserad av. Det uppnås med filtrering och sortering. Den generiska filter- och sorteringsfunktionen fungerar lika för alla tabeller och dess listvyer. Listan kan filtreras och sorteras på alla fält och kan sorteras på fler fält samtidigt, inga restriktioner finns för filtering på tabellens egna fält. Ifall det behövs filtrering för fält i kopplade tabeller, kan man skapa en separat anpassat (custom) filter. I detta avsnitt behandlas enbart den generiska filtrerings- och sorteringsfunktionen. I Figur 15 visas en tabells listvy med dess filtreringsruta.

Show-Hide Column Selection Show-Hide Filter-Pane

Filtering and Sorting

Column: CA_Data_ID Compare Option: EqualTo Filter value:

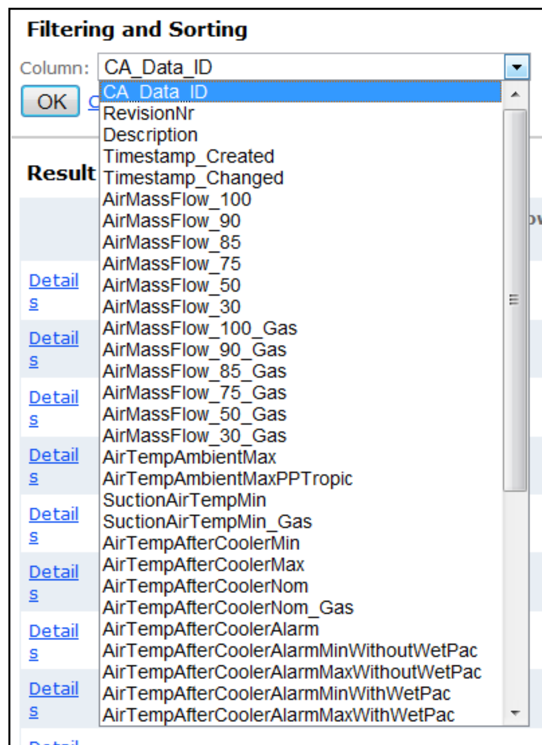
OK Clear Filter *You can filter a text on several values, separated by space*

Result

	AirMassFlow_100	AirMassFlow_50	AirMassFlow_75	AirMassFlow_85	AirMassFlow_90	CA_Data_ID	Description
Details	0,94	0,53	0,74	0,81		1	W4L20 B CS 720
Details	0,99	0,57	0,79	0,86		2	W4L20 B CS 750
Details	1,24	0,69	0,98	1,09		3	W4L20 B CS 900
Details	1,31	0,77	1,04	1,15		4	W4L20 B CS 1000
Details	1,35	0,75	1,05	1,18		5	W4L20 C CS 900
Details	1,42	0,82	1,12	1,24		6	W4L20 C CS 1000
Details	1,5	0,91	1,21	1,33		7	W5L20 B CS 1000
Details	1,36	0,77	1,06	1,17		8	W6L20 B CS 720
Detail	1,43	0,81	1,12	1,25		9	W6L20 B CS 750

Figur 15 Listvy med standardfilter

Standardfiltret fungerar så att man först väljer det fält man vill filtrera eller sortera på. I dropdown kontrollen listas alla fält i tabellen se Figur 16.



Figur 16 Lista på fält för filtrering och sortering

Därefter väljs det jämförelsealternativ man är intresserad av se Figur 16.

Hur värdena fås genom Reflection visas Figur 17. Funktionen returnerar en lista innehållande egenskapsinformation (List<PropertyInfo>). Denna funktion finns i en generisk klass som kallas FilterUtil.

```
public class FilterUtil<T> where T : class
```

Det innebär som tidigare behandlats kort att den instansieras för en viss klass som här bär platshållaren T (Type) med Typ som argument. I funktionen GetColumnPropertyInfos() fås en array av Egenskaper för denna typ med funktionen GetProperties(). Därefter väljs de egenskaper ut från denna lista som har attrubutet ”ColumnAttribute”.

Funktionen returnerar en generisk lista på Egenskaps-Info (PropertyInfo)

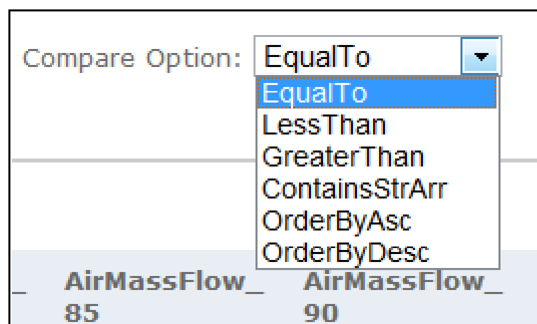
En ny EgenskapsInfo -lista instansieras.

En Array på Egenskaps-Info fås genom Reflection. På typvariabeln (T) genom funktionen GetProperties(). Därefter itereras denna lista med en foreach-loop på nästa rad.

```
#region GetColumnPropertyInfos
//---------------------------------------------------------------------------------------
public List<PropertyInfo> GetColumnPropertyInfos
{
    List<PropertyInfo> piList = new List<PropertyInfo>();
    PropertyInfo[] pis = typeof(T).GetProperties();
    foreach (PropertyInfo pi in pis)
    {
        object ca = pi.GetCustomAttributes(typeof(ColumnAttribute), false).FirstOrDefault();
        if (ca != null) piList.Add(pi);
    }
    return piList;
}
//.......................................................................................
#endregion
```

De PropertyInfo som har attributet ColumnAttribute tas med i resultatet. Det är ett attribut som genereras av LinqToSql ramverket.

Figur 17 Funktionen som returnerar kolumndata för en tabell



Figur 18 Jämförelsealternativ vid filtrering / sortering

Slutligen skriver man in värdet för filtreringen i textrutan längst till höger se Figur 15. När man gjort en filtrering eller sortering läggs automatiskt till en ny rad i för att kunna utvidga filtret, se Figur 19.

Filtering and Sorting

Column: CA_Data_ID Compare Option: EqualTo Filter value: 2

Column: CA_Data_ID Compare Option: EqualTo Filter value:

OK Clear Filter You can filter a text on several

Figur 19 En ny filter/sorteringsrad skapas

På detta vis kan filtret byggas på obegränsat. I det följande beskrivs denna funktion steg för steg. I detta arbete finns inte utrymme för en ingående presentation av alla tekniker som används ss. MVC i ASP.NET men när sådan kod visas så ges en ytlig förklaring till det som är relevant för det behandlade ämnet.

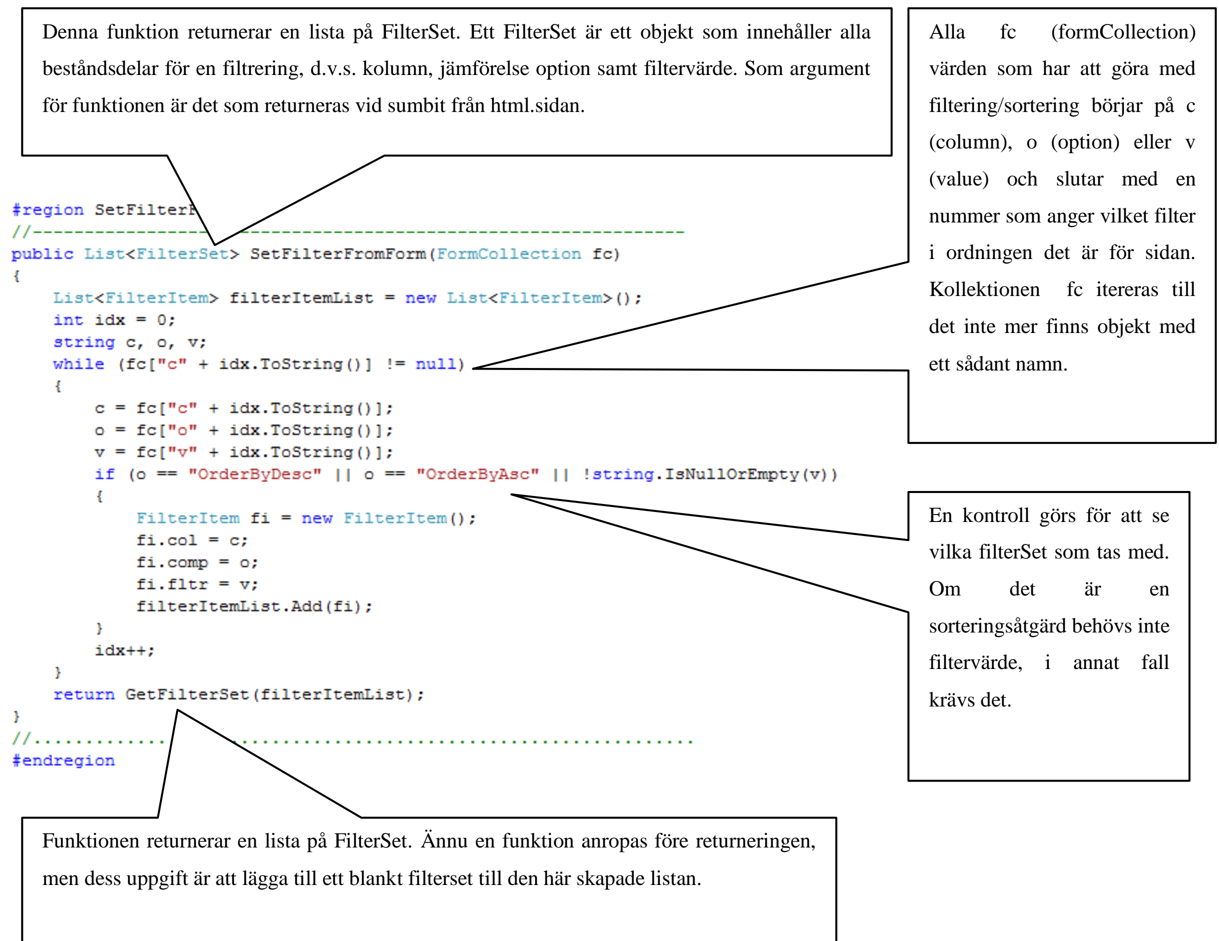
```
#region CA_DataList Post from filter
//----------------------------------------------------------------------------------
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult CA_DataList(FormCollection fc, bool? setColumns)
{
    if (setColumns == true)
    {
        listColSel.SetColumnSelectionFromForm(Session["ListColumnSelection"] as List<ListColNameMap>, fc, Auth
    }
    else
    {
        Session["FilterSet" + extraColumnSuffix] = filterUtil.SetFilterFromForm(fc);
    }

    return RedirectToAction("CA_DataList");
}
//.............................................................................
#endregion
```

Figur 20 Filtreringsfunktion, Sidans Controller tar emot filtervärdena

I Figur 20 visas funktionen i aktuella sidans kontroller (Controller) som anropas när ett filtervärde sänds med sidans postback. Som första argument för funktionen finns ”fc” av typen FormCollection (det andra argumentet ”setColumns” av typen bool? Bortser vi från i detta skede, det är ”true” om man vill sätta vilka kolumner som skall visas i listvyn). ”fc”-argumentet skickas vidare till en annan funktion, filterUtil.SetFilterFromForm(fc). Resultatet från denna funktion sparas i en sessionsvariabel, Session[”FilterSet” + extraColumnSuffix]. Sessionsvariabeln har ett standardnamn plus ett suffix specifikt för denna klass för att skilja den från FilterSet skapade för andra tabeller. Vid sidan kan här nämnas att denna typ av index-syntax har den fördelen att variabelnamnet kan skapas genom manipulation av

teckensträngar, vilket man inte kan göra med ”normala” variabler. I Figur 21 visas hur FilterUtil skapar detta ”FilterSet”.



Figur 21 FilterSet skapas från html-form input

Dessa FilterSet skapas direkt när resultatet från html-formuläret returneras. De används sedan till att filtrera raderna för tabellen som visas på sidan. Hur det sker behandlas som följande.

När listvy-sidan skapas anropas filterUtil för att returnera den filtrerade och sorterade listan. Nedan ett utdrag källkoden.

```
list = filterUtil.GetFilteredList(Session["FilterSet" + extraColumnSuffix] as List<FilterSet>, out filterError);
```

Här ser vi att samma sessionsvariabel är

argument för denna funktion. Metoden har även ett out-argument, ifall fel uppstår visas beskrivningen i denna sträng. Ifall inga fel uppstod returneras en tom sträng. Nedan visas hur filterutil utför filtreringen och sorteringen d.v.s funktionen ”GetFilteredList”. Själva koden med egna kommentarer visas som monotype-text och förklaringarna här tillsatta i textrutor. Textrutorna ger förklaringar till koden som finns ovanför.

```
        public List<T> GetFilteredList(List<FilterSet> filterSetList, out string
errMsg)
        {
            List<T> list = repository.ListLastRevisionOnly();
            errMsg = string.Empty;
            List<ColumnNameMap> colNameMap = GetColumnNameMapList();
            PropertyInfo pi = null;
            filterSetList.Reverse();//gets the orderby order right
```

ListLastRevisionOnly returnerar en lista med de senaste revisionerna.

errMsg är platshållaren för felmeddelande, ifall inget fel uppstår returneras en tom sträng.

colNameMap är en lista på kolumnnamn med extra kolumner.

Listan reverseras för att orderBy, d.v.s. sorteringen, om den görs på fler fält, går i rätt ordning (filterSetList.Reverse).

```
            foreach (FilterSet fset in filterSetList)
            {
                try
                {
```

Alla filterSet itereras i en foreach-loop.

Varje iterering innesluts i ett try – catch block. Ifall en misslyckas läggs felmeddelandet till errMsg.

```
                    if (fset.col == null) continue;

                        string dbColName = colNameMap.Where(a => a.ColNameMapped ==
fset.col).Select(a => a.DbColName).Single();
                        pi = typeof(T).GetProperty(dbColName);

                        if (fset.comp == "OrderByAsc")
                        {
                            list = list.OrderBy(a => pi.GetValue(a, null)).ToList();
                        }
                        else if (fset.comp == "OrderByDesc")
                        {
                            list = list.OrderByDescending(a => pi.GetValue(a,
null)).ToList();
```

```
                }
```

On den aktuella kolumn är null gås direkt till nästa objekt (FilterSet)

i dbColName sparas det egentliga kolumnnamnet, inte det mappade.

i PropertyInfo pi sparas egenskapen (property) som fås genom funktionen type.GetProperty(string propertyName).

Därefter kontrolleras om det är fråga om sortering för detta FilterSet. Om så är fallet sorteras listan stigande (ascending) eller fallande beroende på vad som är valt. (CompareOption).

```
                else
                {
                    if (string.IsNullOrEmpty(fset.FilterValue)) continue;
                    DateTime date;
                    decimal number;
                    bool yesno;
                    string text = fset.FilterValue;

                    bool parseOk = DateTime.TryParse(text, out date);
                    if (parseOk && fset.comp != "ContainsStrArr")
                    {
                        if (fset.comp == "EqualTo") list = list.Where(a =>
pi.GetValue(a, null) != null && (DateTime)pi.GetValue(a, null) == date).ToList();
                        else if (fset.comp == "LessThan") list = list.Where(a
=> pi.GetValue(a, null) != null && (DateTime)pi.GetValue(a, null) < date).ToList();
                        else if (fset.comp == "GreaterThan") list =
list.Where(a => pi.GetValue(a, null) != null && (DateTime)pi.GetValue(a, null) >
date).ToList();
                        continue;
                    }
```

Först kontrolleras att filtervärdet inte är tomt eller null. Om så är fallet gås till nästa objekt i itereringen.

Funktionen klargör vilken datatyp som filtervärdet är genom (datum, tal, boolean eller text). Det mest specifika, datum, kontrolleras först genom DateTime.Parse- funktionen. Om så är fallet sker filteringen enligt angiven CompareOption (EqualTo, LessThan, GreaterThan). Om CompareOption är ”ContainsStrArr” sker filtreringen senare i funktionen.

Filtreringen sker med ett Linq-Where-uttryck som använder Reflection (pi.GetValue(a, null)).

```
                    parseOk = decimal.TryParse(text, out number);
                    if (parseOk && fset.comp != "ContainsStrArr")
                    {
                        if (fset.comp == "EqualTo") list = list.Where(a =>
pi.GetValue(a, null) != null && decimal.Parse(pi.GetValue(a, null).ToString()) ==
number).ToList();
                        else if (fset.comp == "LessThan") list = list.Where(a
=> pi.GetValue(a, null) != null && decimal.Parse(pi.GetValue(a, null).ToString()) <
number).ToList();
                        else if (fset.comp == "GreaterThan") list =
list.Where(a => pi.GetValue(a, null) != null && decimal.Parse(pi.GetValue(a,
null).ToString()) > number).ToList();
                        continue;
                    }

                    parseOk = bool.TryParse(text, out yesno);
                    if (parseOk && fset.comp != "ContainsStrArr")
                    {
                        if (fset.comp == "EqualTo") list = list.Where(a =>
pi.GetValue(a, null) != null && bool.Parse(pi.GetValue(a, null).ToString()) ==
yesno).ToList();
                        continue;
                    }
```

På samma sätt som i tidigare förklaringsruta kontrolleras datatypen genom att sedan testa om värdet är av typen decimal och därefter boolean varefter filtreringen görs.

```
                            list = FilterUtil<T>.FilterListByStringArray(list, 
dbColName, text);
```

Det sista alternativet: filtering på en teckensträng sker med en separat funtion, FilterListByStringArray. Med hjälp av denna funktion kan man filtrera på fler strängar genom att filtervärdet splittas upp vid varje mellanslag. Denna funkton gås inte närmare igenom i detta arbete.

```
                    }
                }
                catch
                {
                    errMsg += "Error in filter on column:" + fset.col + " and filter 
value " + fset.FilterValue + ".\n";
                    continue;
                }
            }
            filterSetList.Reverse(); //reverse back
            return list;
        }
```

Till sist svängs filterSet kollectionen om (reverseras) med funktonen Reverse igen och listan returneras. Före det ses catch-delen av try-catch-blocket, i vilken eventuella felmeddelanden läggs till errMsg som returneras i ett out-argument.

Med dessa funktioner kan alltså listan filtreras och sorteras utan begränsning på alla kolumner och dessa funktioner kan tillämpas på alla objekt eller tabeller i databasen efter att de instansierats för gällande typ. Generics och Reflection är huvudkomponenterna i denna generiska funktion.

3.5 Val av fält för visning på ListVy-sidan

Förutom valmöjligheter för filtrering finns även möjlighet att välja vilka kolumner som skall visas i listvyn för varje tabell. Man får fram en lista på alla kolumner med en checkbox och textfält för i vilken ordning kolumnen skall visas genom att klicka på ”Show-Hide Column Selection” uppt till vänster på sidan. Denna del visas och döljs med en JQuery funktion, men detta är utanför denna uppsats fokus. Detta val sparas per användare individuellt i databasen och har man gjort ett val, visas således samma urval tills man gör en ändring (om man loggar in som samma användare). Om man inte har gjort något personligt val, visas ett standardurval som måste göras individuellt för varje tabell. Det kan en person med admin-behörighet göra om han i konfigurationen ändrar en inställning temporärt.

Användargränssnitter är intuititv för denna funktion, man väljer de kryssrutor som man vill skall visas och om man vill sätter man även i vilken ordning. Därefter klickar man på OK-knappen och inställningen är sparad. På samma sätt som för sortering och filtrering används här även Generics och Reflection för att skapa en algoritm som kan användas för alla typer eller tabeller. I Figur 22 visas en listvy-sida med column-select öppnat.

Show-Hide Column Selection   Show-Hide Filter-Pane

Choose Listed Columns

OK	Column	Order
☑	AirMassFlow_100	0
☐	AirMassFlow_100_Gas	0
☐	AirMassFlow_30	0
☐	AirMassFlow_30_Gas	0
☑	AirMassFlow_50	0
☐	AirMassFlow_50_Gas	0
☑	AirMassFlow_75	0
☐	AirMassFlow_75_Gas	0
☑	AirMassFlow_85	0
☐	AirMassFlow_85_Gas	0

Result

	AirMassFlow_100	AirMassFlow_50	AirMassFlow_75	AirMassFlow_85	AirMassFlow_90	CA_Data_ID	Description
Details	0,94	0,53	0,74	0,81		1	W4L20 B CS 720
Details	0,99	0,57	0,79	0,86		2	W4L20 B CS 750
Details	1,24	0,69	0,98	1,09		3	W4L20 B CS 900
Details	1,31	0,77	1,04	1,15		4	W4L20 B CS 1000
Details	1,35	0,75	1,05	1,18		5	W4L20 C CS 900
Details	1,42	0,82	1,12	1,24		6	W4L20 C CS 1000
Details	1,5	0,91	1,21	1,33		7	W5L20 B CS 1000
Det	1,36	0,77	1,06	1,17		8	W6L20 B CS 720

Figur 22 Val av kolumner för visning i ListVyn

Nedan visas hur denna funktion har åstadkommits. I funktionen nedan (Figur 23) finns resultatet från sumit-form för urval av kolumner till visning i argumentet ”FormCollection fc”.

```
210  #region SetColumnSelectionFromForm
211  //-------------------------------------------------------------------------------------
212  public void SetColumnSelectionFromForm(List<ListColNameMap> origColMapList, FormCollection fc, string userName)
213  {
214      List<ListColNameMap> newColMapList = new List<ListColNameMap>();
215      int cidx = 0;
216      while (fc["ColSel" + cidx.ToString()] != null)
217      {
218          int sortInt = 0;
219          string value = fc["ColSel" + cidx.ToString()] as string;
220          string sortStr = fc["ColSelSort" + cidx.ToString()] as string;
221          int.TryParse(sortStr, out sortInt);
222          if (value.Contains("true"))
223          {
224              ListColNameMap colNameMap = origColMapList[cidx];
225              colNameMap.Selected = true;
226              colNameMap.Sort = sortInt;
227              newColMapList.Add(colNameMap);
228          }
229          cidx++;
230      }
231      SetNewColSelection(newColMapList, userName);
232  }
233  //.....................................................................................
234  #endregion
```

Figur 23 Funktionen SetColumnSelectionFromForm

Denna metod fungerar på ungefär samma sätt som motsvarande för filtrering och sortering. Varje formulärvärde för en kolumn som valts har ett namn i formen ”ColSel” + index. Om t.ex. två kolumner valts har kontrollen för deras värde namnet ”ColSel0” och ”ColSel1”. På samma sätt kontrolleras ”ColSelSort”, d.v.s. den eventuella sorteringsordningen för kolumnen. En while-loop (rad 216) ökar indexvärdet för varje iterering och om ”fc” innehåller objekt med detta index fortsätter itereringen. En lista med kolumner skapas för varje vald kolumn och skickas till nästa funktion SetNewColSelection (rad 231).

Denna metod returnerar void, men resultatet newColMapList av typen List<ListColNameMap> (ListColNameMap är en klass designat för att förvara nödvändig information) skickas till en annan metod SetNewColSelection (rad 231 i Figur 23).

Metoden SetNewColSelection (Figur 24) har som uppgift att spara den aktuella användarens val i databasen. Olika användare kan vara intresserad av olika kolumner och kan alltså på detta sätt välja vilka kolumner som visas i listvyn. I det följande förklaras kort metoden. På raden 183 raderas användarens befintliga preferenser, om sådana finns genom att anropa en annan metod, ClearCurrentColSel. Därefter itereras objekten i metodens första argument som är en lista på colList-objekt. Under

iterationen skapas nya column preferens objekt (ListColumnSelect) och ges värden från det aktuella objetet i iterationen samt användarnamnet som är det andra argumentet för metoden.

```
181        public void SetNewColSelection(List<ListColNameMap> colList, string userName)
182        {
183            ClearCurrentColSel(userName);
184
185            foreach (ListColNameMap item in colList)
186            {
187                ListColumnSelect lcs = new ListColumnSelect();
188                lcs.UserName = userName;
189                lcs.TableName = typeof(T).Name;
190                lcs.Sort = item.Sort;
191                lcs.ColumnName = item.ColNameDb;
192                rep.table.InsertOnSubmit(lcs);
193            }
194
195            rep.Save();
196        }
```

Figur 24 Metoden SetNewColSelection sparar den inloggade användarens preferenser

När det aktuella objektets (tabellens) controller skall preparera data för visning i vyn anropas GetListRows (Figur 25 GetListRows anropas från sidans Controller) vars uppgift är att skapa ett objekt som för varje rad i listobjekt-argumentet innehåller endast de kolumner som definierats tidigare.

```
146        public List<ListRow> GetListRows(object listObj)
147        {
148            IEnumerable list = (IEnumerable)listObj;
149            List<ListRow> rows = new List<ListRow>();
150
151            //get right order of dbColNames
152            List<string> dbColNames = colSelList
153                .Join(listColNameMapList, a => a.ColumnName, b => b.ColNameDb, (a, b) => new { a = a, b = b })
154                .OrderByDescending(c => c.b.Sort)
155                .ThenBy(c => c.b.ColNameMapped)
156                .Select(c => c.b.ColNameDb)
157                .ToList();
158
159            foreach (object item in list)
160            {
161                ListRow ls = new ListRow();
162                ls.Id = (int)piId.GetValue(item, null);
163                List<string> colVals = new List<string>();
164                foreach (string c in dbColNames)
165                {
166                    PropertyInfo pi = item.GetType().GetProperty(c);
167                    object value = pi.GetValue(item, null);
168                    string valStr = value == null ? string.Empty : value.ToString();
169                    colVals.Add(valStr);
170                }
171                ls.ColVals = colVals;
172                rows.Add(ls);
173            }
174            return rows;
175        }
```

Figur 25 GetListRows anropas från sidans Controller

Metodens uppgift är alltså att välja ut de kolumner som är valda att visas för den aktuella användaren och om denne inte gjort några personliga val, så default-kolumnerna.Metoden får som argument den färdigt sorterade och filtrerade listan som skall visas i listvyn. På rad 152 (Figur 25) finns ett Linq-uttryck där kolumnnamnen sorteras enligt de av användaren angivna preferenserna. Dessa värden finns sparade i colSelList-objektet som definieras när den generiska klassen ListColumnSelection instansieras för gällande Typ (Tabell).

I iterationen på rad 159 skapas ett ListRow-objekt, som är en custom-klass vilken innehåller radens id samt en List<string> med de valda kolumnerna värden. Kolumnerna i källtabellen kan vara av olika datatyp men de ändras här till en teckensträng. Här används igen Reflection som med hjälp av datatypen och fältets namn får fram fältets värde. Denna metod returnerar alltså ett kolumnurval av den sorterade och filtrerade listan vilken den har som argument.

Som följande visas hur själva vyn presenterar denna lista. Nedan visas hela den s.k. partiella vyn (Partial View) som listar elementen i listvyn. Denna partiella vy används

för alla tabeller och är kort och koncis. En kort förklaring av symbolerna ges här först. Det som finns innanför <% %>-taggar är kod och uttryck som servern behandlar innan det sänds till klientens webläsare. Det som inte är innanför sådana taggar är html. I traditionell ASP.NET (WEB FORMS) avråds från att skriva kod på själva sidorna men i MVC är det normalt och skall så göras för kod som inte har med business-logiken att göra utan med själva presentationen. Ett annat objekt som inte finns i traditionell ASP.NET är ViewData[”somename”]. Det fungerar på samma sätt som en sessionsvariabel (Session[”somename”]) förutom att det försvinner efter två requests i motsats till sessionsvariabeln som, som namnet säger, bevaras under hela sessionen.

I koden finns textboxar med förklaringar till vad som finns ovanför boxen.

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
```

Detta är ett så kallat direktiv som finns uppe på alla ASP.NET sidor eller UserControls. Detta är alltså en MVC-userControl och språket som används på sidan i kod är C#.

```
    <%
        int selectedId = 0;
        if (ViewData["id"] != null) selectedId = (int)ViewData["id"];
    %>
```

Här deklareras in int variabel som ges värdet på det valda id:t om sådant finns. Sådant finns om en post har varit vald tidigare och vi kommer tillbaks till sidan från ViewItem eller EditItem. Observera att kod är innesluten i <% %> taggar.

```
<table class="zebra word-break-all">
<tr>
<th></th>
```

Här deklareras en html-tabell med css klasser och därefter början på en ny rad och en tom rubrikkolumn. Observera att här inte finns kod eller uttryck och följaktligen inga <% %> taggar.

```
    <% foreach (string colHeader in ViewData["ColHeaders"] as List<string>)
       { %>
            <th><%= Html.Encode(colHeader) %></th>
     <%   } %>
```

En foreach loop som skriver html-rubriker för varje kolumnnamn som finns i ViewData[”ColHeaders”]. Dessa viewdata objekt definieras i sidans controller.

```
</tr>
    <% foreach (Repository.ListRow row in ViewData["rows"] as
List<Repository.ListRow>)
       { %>
        <tr class="<%= selectedId == row.Id ? "selectedRow" : "" %>">
            <td>
                <%= Html.ActionLink("Details", ViewData["ReadActionName"] as string,
       new { id =                                              row.Id, page =
       ViewData["CurrentPage"] })%>
```

```
            </td>
```

Foreach loopen går igenom raderna i ViewData[”rows”] d.v.s. alla rader som skall visas på sidan. På den tredje raden ges tabellraden css-klassen ”selectedRow” ifall dess id är samma som selectedId. Sedan skapas en ActionLink, d.v.s. en link med vilken man kan navigera till en annan vy där man ser alla detaljer för denna rad.

```
          <%  foreach (string val in row.ColVals as List<string>)
             {%>
                  <td><%= Html.Encode(val) %></td>
           <% } %>
          </tr>
      <%  } %>

</table>
```

Till sist itereras ColVals d.v.s. en lista med värden i de kolumner som skall visas på sidan.
Noteras kan att värdet ges inom en <%= Html.Encode( )%>, fast det skulle kunna ges direkt i <td> </td>. Html.encode gör att t.ex. tecknet < kodas till html och blir således <

4 EXTERNA SYSTEM

Denna databas lever inte isolerat utan dess data används för olika ändamål i olika sammanhang. Externa system har vyer (views) definierade och deras namn och vad de presenterar är i stort oförändrat från den tidigare versionen. Vyer har kommit till p.g.a. revisionshanteringen. Problem uppstod i.o.m. önskemålet att dynamiskt kunna lägga till kolumner i tabeller. Som tidigare behandlats ser det så ut ur användarens synvinkel, men i själva verket skapas inte nya kolumner utan de finns färdigt i databasen med namn som s1, s2.. etc. När man tar i bruk en ny kolumn syns den för användaren med det namn som mappats för den. När en ny kolumn tagits i bruk skall den också synas i vyerna tillgängliga för de externa systemen med det namn som getts för den.

Hittills har alla funktioner beträffande mappningen skett i klientapplikationen och databasen har lämnats oberörd beträffande modifikationer i dess schema. För att vyerna som konsumeras av externa system skall få uppdaterade kolumnnamn måste de genereras på nytt, vilket här sker automatiskt genom att klientapplikationen genererar ett sql-skript som skickas till databasen för exekvering genom en lagrad procedur (stored procedure).

5 DATASÄKERHET

Applikationen fungerar inom beställarens intranet, men det är en stor firma, data är inte tillägnat att komma i fel händer och i synnerhet måste det finnas kontroll på vilka som har behörighet att modifiera data. I detta avsnitt behandlas till slut hur behörigheterna kontrolleras i denna applikation.

Det finns en användargrupp som har rätt att läsa data och en som har behörighet att ändra, lägga till och radera data. Man behöver inte logga in med användarnamn och lösenord utan eftersom det fungerar i företagets intranet och användarna hör till företagets domain, har de tillgång beroende på vilken eller vilka Windows Active Directory grupper de hör till.

För att kunna editera data skall man höra till admin-gruppen. I applikationens web.config finns en setting som heter AdminRoleName.

```
   <!--logged in users who belong to the adminRole can edit values--
>
   <setting name="AdminRoleName" serializeAs="String">
    <value>xxxxx</value>
   </setting>
```

När en användare försöker editera en post anropas först en klass som heter Authorization. Där kontrolleras om användaren ifråga hör till den gruppen som har namnet som finns sparat i ”AdminRoleName” i settings.

```
        public static bool CurrentUserIsInRole(string roleName)
        {
            //exception occurs when there is no such rolename,
      trying to check     other domain which are untrusted etc.
            try
            {
                return CurrentUser.IsInRole(roleName);
            }
            catch
            {
                return false;
            }
        }
```

Ifall så inte är fallet, dirigeras användaren till en sida som meddelar att denne inte har behörighet för denna operation.

För maskintyper finns ett finmaskigare system. Produktionen av olika maskintyper sker på olika ställen och man vill begränsa rätten att modifiera dessa data till dem som arbetar på stället ifråga. Detta har lösts på följande sätt.

För varje maskintyp bestäms en eller fler Active Directory grupper som har behörighet att editera denna typs data. Då en person försöker öppna en edit-sida kontrolleras på samma sätt ifall personen hör till denna eller dessa grupper. Här används ett objet eller tabell ”AccessPoints” som innehåller produktionsorten eller landet, namnet på ad-gruppen samt ett id. Man måste alltså hör till åtminstone två grupper för att editera maskindata, gruppen vars namn bestäms i web.config samt gruppen som finns definierad i maskinens AccessPoint, vilken sparas i databasen.

6 FRAMÅTBLICK

De flesta applikationer kräver underhåll och uppdateringar beroende på olika faktorer. Man upptäcker fel, antingen programfel eller i hur businessmodellen implementeras. Sedan kommer ny information, nya behov och man kommer på sätt hur man skulle kunna effektivera arbetet. Bra är även att försöka förbättra kvaliteten med refaktorisering.

Omstrukturering av kod eller refaktorisering (eng: refactoring) är en teknik för att stegvis förbättra kvaliteten på programkod. Syftet är att möblera om koden för att få den enklare och därmed mer lättläst, lättare att underhålla och lättare att vidareutveckla.

En viktig detalj med omstruktureringen är att funktionaliteten utåt sett skall vara identisk efteråt, det vill säga beteendet på koden skall inte ha ändrats. Därför bör omstrukturering göras stegvis även vid enkla ändringar. Den nya .NET versionen har nyheter som skulle lämpa sig bra för refaktorisering av detta projekt.

Under år 2010 kom Microsoft ut med version 4 av .NET samt Visual Studio 2010. Här finns några intressanta nyheter som skulle ha varit av intresse för denna applikation. Det är framförallt de nya dynamiska funktionerna som är av intresse samt den nya datatypen ”dynamic”. Därefter behandlas kort även namngivna och valfria parametrar. .NET 4 har även andra nyheter med dessa är av huvudsakligt intresse för detta program.

6.1 Datatypen Dynamic

.NET 4.0 introducerar ett nytt nyckelord till C#-språket nämligen *dynamic*. Ordet möjliggör att införa skriptliknande beteende i ramverket. Man kan med de dynamiska funktionerna förenkla många komplexa uppgifter och det möjliggör även att samverka med ett antal dynamiska språk såsom IronRuby eller IronPython.

Här ges ingen genomgående presentation utan med ett exempel visas hur många kodavsnitt med Reflection kan skrivas kortare. I kodsnuttarna nedan kan med de dynamiska egenskaperna skrivas på följande sätt.¨

```
static void CreateUsingLateBinding(Assembly asm)
{
  try
  {
    // Get metadata for the Minivan type.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // Create the Minivan on the fly.
    object obj = Activator.CreateInstance(miniVan);

    // Get info for TurboBoost.
    MethodInfo mi = miniVan.GetMethod("TurboBoost");

    // Invoke method ('null' for no parameters).
    mi.Invoke(obj, null);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Här har reflection utan nyckelordet ’dynamic’ använts. Samma sak utför kodsnutten nedan med mindra mängd kod.

```
static void InvokeMethodWithDynamicKeyword(Assembly asm)
{
  try
  {
    // Get metadata for the Minivan type.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // Create the Minivan on the fly and call method!
```

```
    dynamic obj = Activator.CreateInstance(miniVan);
    obj.TurboBoost();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Som vi ser kan metoden TurboBoost kallas direkt på det dynamiska objektet obj. Med dessa nya dynamiska egenskaper kunde detta projekt förenklas på många ställen (Troelsen, 2010).

6.2 Namngivna och valfria parametrar

Namngivna och valfria parametrar kan användas i många andra språk och är heller ingen nyhet, men en välkommen nyhet är det för C#. Koden blir mer informativ, vid ett funktionsanrop ser man namnen på parametrarna direkt och de kan skrivas i vilken ordning som helst (om de är namngivna). Det största nyttan är att man kan minska rader kod avsevärt genom att istället för att överladda (overload) en metod kan den skrivas med namngivna och valfria parametrar, se exempel nedan.

```
public void Process(string data)
{
    Process(data, false);
}

public void Process(string data, bool ignoreWS)
{
    Process(data, ignoreWS, null);
}

public void Process(string data, bool ignoreWS, ArrayList moreData)
{
    // Actual work done here
}
```

Ovan har metoden Process 3 överladdningar, d.v.s. den kan anropas med en till tre argument. Samma sak kan skrivas som en metod med de nya möjligheterna nedan.

```
public void Process(string data, bool ignoreWS = false, ArrayList
moreData = null)
 {
     // Actual work done here
 }
```

Som visats har tre olika versionen av metoden komprimerats till en.

7 SAMMANFATTNING

En maskintillverkare har ett befintligt system baserat på Microsoft Access för lagring av maskinkonfigurationer och deras olika prestandavärden. Deras önskemål är att flytta databasen till SQL-server och som nya egenskaper lägga till ett revisionshanteringssystem samt ett web-användargränssitt. Systemet skall vara lätt att använda, ge en god helhetsbild av data vid en blick och fungera i samverkan med externa system.

Databasen består av många olika tabeller och huvudfunktionen, förutom lagring av prestandadata, är att skapa och hantera revisioner. För att uppnå generisk funktionalitet d.v.s. genensamma funktioner för de olika objekten har det använts vissa delar av .NET ramverket, framförallt Generics och Reflection. Denna funktionalitet har varit fokus för detta arbete.

Under arbetets gång uppstod vissa oförutsedda moment, bl.a. med avseende på externa system och prestanda. En mix av hårdkodning, allmängiltiga funktioner och generering av objekt i databasen måste göras för att dessa aspekter kunde tillfredställas. I det stora hela visade sig funktionerna fungera väl och programmet har visat sig stabilt.

.NET utvecklas hela tiden och nya egenskaper kommer till, nyligen (2010) har .NET 4.0 presenterats. Här finns ny funktionalitet som ytterligare skulle kunna förenkla metoder i detta arbete, framförallt de den dynamiska funktionaliteten är intressant i detta avseende.

KÄLLFÖRTECKNING

Galloway, J., Haack, P., Hanselman, S., Guthrie, S., & Conery, R. (2010). *Professional ASP.NET MVC 2.* Hoboken: Wrox Press, Tillgänglig i elektronisk form via <URL:http://www.books24x7.com>, by SkillSoft.

Goda, A. (2010). *Introducing Silverlight 4.* New York, USA: Apress, Tillgänglig i elektronisk form via <URL:http://www.books24x7.com>, by SkillSoft.

MacDonald, M., Freeman, A., & Szpuszta, M. (2010). *Pro ASP.NET 4 in C# 2010, Fourth Edition.* New York: Apress, Tillgänglig i elektronisk form via <URL:http://www.books24x7.com>, by SkillSoft.

Mehta, V. P. (2008). *Pro LINQ Object Relational Mapping with C# 2008.* New York: Apress, Tillgänglig i elektronisk form via <URL:http://www.books24x7.com>, by SkillSoft.

Sack, J. (2008). *SQL Server 2008 Transact-SQL Recipes.* New York: Apress, Tillgänglig i elektronisk form via <URL:http://www.books24x7.com>, by SkillSoft.

Troelsen, A. (2010). *Pro C# 2010 and the .NET 4 Platform, Fourth Edition.* New York: Apress, Tillgänglig i elektronisk form via <URL:http://www.books24x7.com>, by SkillSoft.