Opinnäytetyö (AMK)

Tietotekniikka

Sulautettujen järjestelmien ohjelmistotekniikka

2010

Sami Elmroos

# CIPROJECT

## – PROJEKTINHALLINTAJÄRJESTELMÄ

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Sulautettujen järjestelmien ohjelmistotekniikka

Toukokuu 2010 | Sivumäärä: 28

TkL Jari-Pekka Paalassalo

Elmroos Sami

# CIProject -projektinhallintajärjestelmä

Työssä oli tavoitteena parantaa olemassa olevan projektinhallintajärjestelmän suorituskykyä ja käytettävyyttä. Tämä toteutettiin purkamalla nHibernate-tietokantakerros ja kirjoittamalla se uudelleen käyttämällä tietokantaproseduureja.

Näyttölogiikka on toteutettu ASP.NET-kielellä. Ohjelmalogiikka ja tietokantayhteydet C#.NET-kielellä. Kehitysympäristönä oli käytössä Microsoft Visual Studio 2008 SP1 ja tietokantana Microsoft SQL Server 2005.

Uudelleenkirjoitus toteutettiin noudattamalla kolmikerrosarkkitehtuurin mallia siten, että näyttölogiikka, ohjelmalogiikka sekä tietokantayhteydet olivat eroteltuina. Tietokantayhteydet toteutettiin luomalla jokaiselle tietokantataululle omat luokkansa ja luokille tietokantayhteydet tarvittavien metodien perusteella. Lähes jokaiselle luodulle luokalle ohjelmoitiin lataus-, tallennus- ja poistometodit. Tallennusmetodi toteutettiin siten, että se päättelee id-numeron perusteella onko kyseessä oleva tieto jo tietokantataulussa. Sovellus luo myös monia raportteja, joiden kohdalla vanhassa toteutuksessa suorituskyky oli suurin ongelma. Tätä parannettiin luomalla raporteille omat tietokantaproseduurit, joilla koko raportin sisältö voitiin hakea yhdellä tietokantakutsulla.

Käytettävyyden parantamiseksi käytiin käyttöliittymä läpi ja mietittiin sivujen elementtien sijoittelua ja erikoistilanteiden käsittelyä. Muutamassa laajassa raportissa kompensoitiin viivettä luomalla AJAX-tekniikalla siihen latausindikaattorit, jotta käyttäjälle olisi selvempää milloin ladataan ja ettei lataus ole keskeytynyt virheeseen. Samalla lisättiin muutama uusi ominaisuus, kuten tulevien vapaapäivien näyttö etusivulla. Toinen uusi ominaisuus oli tietokantaproseduuripohjainen työntekijän ja hänen alaisiensa työtunnit etusivulla näyttävä toiminto. Tämä toteutettiin kuitenkin siten, että työtunnit ovat sivulle siirryttäessä piilotettuna, jotta esimiestasoinen henkilö voi sovelluksen avata vaikka muita olisi samassa tilassa.

Sovelluksen uudistus onnistui erinomaisesti. Suorituskyky parani raporttien osalta noin 77 %, ja käyttöliittymän parannuksista on saatu työntekijöiltä positiivista palautetta.

ASIASANAT:

C#.NET, C#, SQL, Visual Studio, projektinhallinta, käytettävyys, suorituskyky

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Embedded Systems Software Technology

May 2010 | Total number of pages: 28

Paalassalo Jari-Pekka, Lic. Tech.. Principal Lecturer

Elmroos Sami

# CIProject – project management system

In this thesis the goal was to improve the performance and usability of an already existing project management system. This was implemented with deconstructing nHibernate database layer and writing it again by using procedures.

Presentation logic was implemented with ASP.NET language and business logic and database connections with C#.NET language. The development environment was Microsoft Visual Studio 2008 SP1 and the used database was Microsoft SQL Server 2005.

Rewriting was implemented by using three tier architecture, so that presentation logic, business logic and database access logic were separated. Database connections were implemented by creating each table their own classes and the needed database connections by needed methods. Almost every one of the classes were written with their own load, save and delete methods. The save method was implemented, so that it could decide by using the id number, if the information was on the table already. The system creates many reports and in these the performance was poor in the old system. This was improved by creating procedures to each report separately, so that the whole report could be loaded at one database query.

To improve usability, the user interface was gone through by designing every controller's best positioning and how they handle special cases. In some of the large reports delay was compensated with creating a load indicator with AJAX technologies, so that the user knows when loading is done and loading has not crashed to error. With these features, some new features were added, for example upcoming day offs to the front page. Another new feature was a procedure based widget that shows the cumulative working hours of an employee and those of his / her subordinates. However, this was implemented so that the working hours are hidden at first.

The system upgrade was very successful. The performance was improved approximately by 77 % and the user interface was praised by other employees.

KEYWORDS:

C#.NET, C#, SQL, Visual Studio, Project management, Usability, Performance

# SISÄLTÖ

**1 JOHDANTO** **1**
**2 TEKNIIKAT JA KEHITYSYMPÄRISTÖ** **2**
2.1 Kolmikerrosarkkitehtuuri 3
2.2 .NET-arkkitehtuuri 4
2.3 Visual Studio 2008 4
2.4 ASP.NET 5
2.5 C#.NET 6
2.6 nHibernate 7
2.7 ADO.NET 8
2.8 Microsoft SQL Server 2005 8
2.9 Tietokantaproseduurit 9
**3 KÄYTÄNNÖN TOTEUTUS** **10**
3.1 nHibernate-tietokantakerroksen purku ja uuden tietokantakerroksen kirjoitus 10
3.2 Raportit ja erikoisluokat 10
3.3 Näyttölogiikka ja muut parannukset 11
3.4 Esimerkki työssä käytetyistä ohjelmointityyleistä 12
**4 TULOKSET** **21**
4.1 Testaus 22
4.2 Virheiden hallinta ja niiden korjaus 22
4.3 Versionti 23
**5 PÄÄTELMÄT JA TULEVAISUUS** **24**
**LÄHTEET** **25**

# LYHENTEET

| | |
|---|---|
| ADO | ActiveX Data Objects, rajapinta tietokannoille |
| AJAX | Asynchronous JavaScript And XML, teknologia jolla voidaan tehdä verkkosivuista vuorovaikutteisempia |
| ASP | Active Server Pages, palvelinpuolen ohjelmointimenetelmä |
| CLR | Common Language Runtime, ohjelmakoodin suoritus ja hallinnointi |
| CRM | Customer Relationship Management, asiakkuudenhallintajärjestelmä |
| HTML | Hypertext Markup Language, merkintäkieli, jolla voidaan luoda www-sivuja |
| MSIL | Microsoft Intermediate Language, välikieli |
| ODBC | Open Database Connectivity, rajapinta tietokannoille |
| OLE | Object Linking and Embedding, teknologia objektien upottamiseen |
| ORM | Object-Relational Mapping, ohjelmointitekniikka jolla saadaan tietoa muunnettua tietokannan ja olio-ohjelmointikielen välillä |
| SQL | Structured Query Language, kyselykieli |
| T-SQL | Transact-SQL, Microsoftin laajennus SQL-kyselykieleen |
| XML | eXtensible Markup Language, merkintäkieli |

# 1 JOHDANTO

Toimeksiantona oli toteuttaa olemassa olevaan CIProject-nimiseen projektinhallintajärjestelmään tietokantayhteyksien uudelleenkirjoitus vastamaan yrityksen muita sovelluksia ja tällä tavoin parantaa myös suorituskykyä sekä käytettävyyttä. Työssä toimeksiantajana toimi tutkimusyhtiö Corporate Image Oy, ja kyseessä oli yrityksen sisäisesti käytössä oleva sovellus. Työ toteutettiin muiden töiden ohessa tutkimusyhtiö Corporate Image Oy:lle vuoden 2010 ensimmäisen neljänneksen aikana.

Tietokantayhteydet oli alkuperäisesti toteutettu käyttämällä nHibernate-tietokantakomponenttia. Komponentilla haettiin tiedot tietokannasta luokkiin, ja kaikki tietojenkäsittely suoritettiin ohjelmalogiikassa. Tästä syystä sovelluksen suorituskyky ei ollut tavoitteen mukainen, varsinkaan suurimmissa raporteissa.

Sovelluksen käyttöliittymä oli sijoittelultaan epäselkeä joillain sivuilla. Näitä sovelluksen käyttöliittymän osia pyrittiin parantamaan miettimällä käyttöliittymän elementtien sijoittelua sekä oletushakuehtoja uudelleen. Käyttöliittymän osalta kuultiin loppukäyttäjien kommentteja ja mielipiteitä siitä, miten käyttöliittymän elementtien sekä oletushakuehtojen pitäisi olla.

Sovelluksen suorituskykyä ja yhteneväisyyttä parannettiin kirjoittamalla uudelleen tietokantayhteydet käyttäen ADO.NET-kirjastoa sekä tietokantaproseduureja.

# 2 Tekniikat ja kehitysympäristö

Alkuperäinen sovellus oli toteutettu käyttäen C#.NET-ohjelmointikieltä sekä Visual Studio 2008 -kehitysympäristöä. Sovelluksen avulla yritys pitää kirjaa menneistä sekä meneillään olevista projekteista, laskutuksesta sekä työtunneista. Uuden projektin alkaessa luodaan sovellukseen uusi projekti. Tähän projektiin voivat siihen osallistuvat työntekijät kohdistaa tuntikirjauksensa, ja sovelluksella voidaan joko näiden tuntien pohjalta tai kiinteästi sovitulla summalla laskuttaa asiakasta.

Uutta projektia luotaessa projektille annetaan nimi ja asiakas -tiedot, asetetaan projektin kesto sekä valitaan minkä tutkimuskonseptin pohjalta projekti suoritetaan. Kun projekti luodaan, sovellus luo samalla verkkolevylle projektia varten työkansion, johon työntekijät voivat tallentaa projektiin liittyvät asiakirjat.

Työntekijöiden työtuntiseuranta suoritetaan sovelluksen avulla. Sovellus laskee joka viikko kuluvan työkauden kumulatiiviset työtunnit. Työtunteihin ei kuitenkaan lasketa kuluvaa viikkoa mukaan. Tällä tavoin työntekijöiden pitää viikoittain varmistaa työtuntiensa paikkansa pitävyys. Sovellusta on rajoitettu siten, että työntekijä pystyy kirjaamaan työtuntejaan vain 14 päivää taaksepäin. Mikäli työntekijän työtunnit eivät ole sallituissa rajoissa, lähetetään työntekijälle ja hänen esimiehilleen tästä muistutus.

Sovellus luo laskut automaattisesti projektin tietojen pohjalta muutamalla yksinkertaisella vaiheella. Laskua luotaessa valitaan tunnit, kulut tai sovitut summat laskuun lisättäväksi ja tämän jälkeen tarkastetaan, että asiakkaan osoitetiedot ovat oikein. Kun lasku on luotu, se voidaan tulostaa sovelluksesta yrityksen asiakirjapohjalle ja lähettää suoraan asiakkaalle.

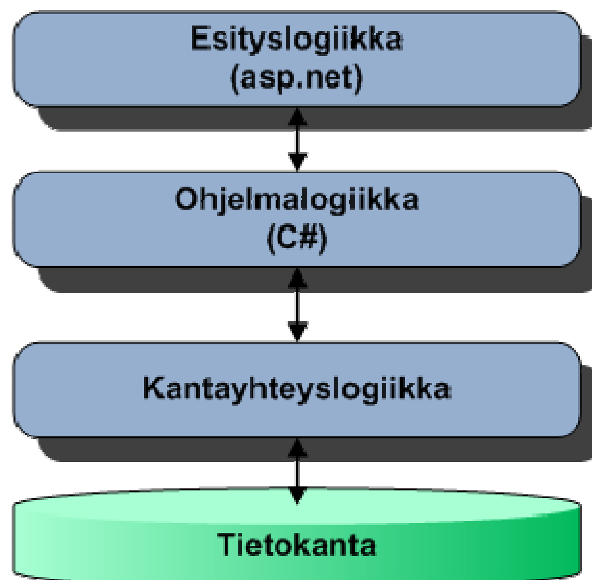
Sovellus on asiakkaiden yhteystietojen ja kontaktilistojen osalta yhdistetty yrityksessä olevaan CRM-järjestelmään. Sovellus synkronoituu joka yö CRM-järjestelmän kanssa, siten että se tuo CRM-järjestelmästä uudet tai muuttuneet tiedot omaan tietokantaansa.

Sovelluksesta pystyy myös lukemaan monia raportteja, kuten kannattavuus-, palautekysely-, laskutus- ja myyntiryhmäraportteja. Useimmiten käytetty laskutusraportti näyttää meneillään olevat projektit ja niihin kohdistuvat saatavat kuukausittain. Laskutusraportissa näytetään myös värikoodauksella, onko saatava jo laskutettu.

Uusi tietokantakerros toteutettiin käyttämällä Microsoftin ADO.NET-kirjastoa ja tietokantaproseduureja. Jokaisella tietokantataululla on oma luokkansa, jossa luokan ominaisuuksina ovat tietokantataulun kentät. ADO.NET-kirjaston avulla kutsutaan tietokantaproseduuria, jonka palauttama vastaus luetaan tietokantataulun luokan ominaisuuksiin. Kun luokalle on luettu arvot, voidaan sitä käyttää sovelluslogiikassa joko tietojenkäsittelyyn tai sen arvot voidaan viedä näyttölogiikan kautta näytölle.

## 2.1 Kolmikerrosarkkitehtuuri

Kolmikerrosarkkitehtuurista löytyy nimensä mukaisesti kolme kerrosta, esityslogiikka, ohjelmalogiikka, tietokantayhteyslogiikka (kuva 2.1).




Kuva 2.1 Vuokaavio kolmikerrosarkkitehtuurista [1].

Esityslogiikassa esitetään käyttöliittymä, joka näkyy käyttäjälle. Esityslogiikassa ei tyypillisesti ole lainkaan tietoa käsittelevää ohjelmakoodia, vaan kaikki käyttäjän antamat arvot ja tapahtumat välitetään ohjelmalogiikalle.

Kaikki toiminnot, arvojen laskemiset, tapahtumien käsittelyt sekä kutsut tietokantakerrokseen toteutetaan ohjelmalogiikassa. Ohjelmalogiikka välittää arvot esityslogiikalle ja sitä kautta käyttäjälle. Työssä ohjelmalogiikka on toteutettu ASP.NET-sivun taustalla olevaan ohjelmakoodiin C#.NET-ohjelmointikielellä sekä jokaiselle tietokantataululle

luotiin omat luokkansa. Tietokantataulujen luokille luotiin kokoelmat, jotta tietokannasta voitiin noutaa useampi kuin yksi rivi kerrallaan.

Tietokantayhteyslogiikassa suoritetaan kutsut tietokantaan ja palautetaan arvot ohjelmalogiikalle. Tyypillisesti tietokantayhteyslogiikka saa parametrina olion viittauksen ja täyttää olion ominaisuudet. [1]

Tässä työssä näyttölogiikkana toimii ASP.NET-ohjelmointikielellä toteutetut www-sivut. Tietokantayhteyslogiikka on toteutettu käyttäen ADO.NET-luokkia sekä tietokantaproseduureja.

## 2.2 .NET-arkkitehtuuri [2]

.NET-arkkitehtuuri on ajonaikainen ympäristö .NET-sovelluksille. Se sisältää peruskirjastoja monille toiminnoille sekä pystyy kommunikoimaan käyttöjärjestelmän kautta laitteiden kanssa. Esimerkiksi tiedoston käsittely, jossa arkkitehtuuri lukee tai kirjoittaa tiedostoa käyttöjärjestelmän kovalevyltä.

.NET-arkkitehtuuri tukee monia sille luotuja ohjelmointikieliä, joista yleisimpiä ovat Visual Basic.NET sekä työssä käytetty C#.NET.

Sovellusta kehitettäessä se käännetään Microsoft Intermediate Language (MSIL) -välikielelle. Ajonaikana suoritusta hallitsee Common Language Runtime (CLR), joka suorittaa MSIL-välikielen ja lataa käytettävät peruskomponentit. CLR hoitaa myös muistin hallinnan sekä liittymisen käyttöjärjestelmän palveluihin.

.NET-arkkitehtuurin ensimmäinen versio esitettiin vuonna 2000 ja keväällä 2010 uusin virallinen versio on 3.5 SP1.

## 2.3 Visual Studio 2008 [3]

Visual Studio -ohjelmisto on Microsoftin kehittämä integroitu ohjelmointiympäristö (IDE). Visual Studiolla voidaan luoda esityslogiikkaa graafisesti siirtämällä komponentteja ikkunaan halutussa järjestyksessä tai luodessa ASP.NET-kielellä www-sivuja voidaan ne luoda joko graafisella käyttöliittymällä tai ohjelmakoodia kirjoittamalla. Se sisältää myös ohjelmakoodieditorin. Ohjelmakoodieditorista löytyy syntaksin korostus, joka helpottaa ohjelmoijan työtä huomattavasti.

Erityistä Visual Studiossa on Intellisense, joka pyrkii avustamaan ohjelmoijaa ehdottamalla vaihtoehtoja siihen, mitä ohjelmoija on juuri kirjoittamassa. Intellisense myös korostaa ohjelmakoodissa olevat virheet sekä pyrkii pitämään automaattisesti kirjoitetun ohjelmakoodin jäsentelyn selkeälukuisena.

Ohjelmakoodin kirjoittamista helpottavat myös Visual Studiosta löytyvät valmiit rungot ehtolauseille, silmukoille sekä muille usein käytettäville ohjelmakoodin osille. Näitä runkoja pystyy myös itse helposti luomaan lisää XML-dokumenttien avulla.

Visual Studiosta löytyy myös debugger-toiminto, jolla ohjelmakoodia suorittaessa pystytään asettamaan pysähdyskohtia ja tarkastelemaan ajonaikana muuttujien arvoja ja näin etsimään mahdollisia virheitä ohjelmoinnista.

Ensimmäinen Visual Studio ohjelmisto oli Microsoft Visual Studio 97, joka esiteltiin vuonna 1997. Keväällä 2010 uusin virallinen versio on Microsoft Visual Studio 2008 SP1.

## 2.4 ASP.NET [4]

Microsoft esitteli vuonna 2000 ASP.NET 1.0:n, joka oli siihen aikaan suuri edistysaskel dynaamisissa verkkosovelluksissa. ASP.NET 1.0 oli hyvin lähellä HTML-ohjelmointikieltä ja siihen oli mahdollista ohjelmoida logiikkaa sivun elementtien toiminnallisuutta varten. Hyvin nopeasti kuitenkin ohjelmoijat huomasivat että toteuttaakseen kaiken tarvittavan toiminnallisuuden, he joutuivat sekoittamaan sivustoon ASP:NET-ohjelmointikielen lisäksi monia muita ohjelmointikieliä. Tällöin sivuston ylläpito ja jatkokehitys muuttui todella hankalaksi.

Versiossa ASP.NET 2.0:n Microsoft oli muuttanut koko ASP.NET pohjaratkaisun oliopohjaiseksi. Sivut määriteltiin edelleen HTML-ohjelmointikielen kaltaisella ASP.NET-ohjelmointikielellä ja siinä voitiin esitellä myös ASP.NET:in omia komponentteja. Suurin muutos oli sivujen yhteyteen luodut ohjelmakoodisivut, joissa jokaisella sivulla on luokka, jolla voi olla ominaisuuksia sekä muita luokille tyypillisiä osia. Ohjelmakoodisivuilla voidaan käsitellä lähes kaikki sivuston tapahtumat sekä tehdä tiedonkäsittelyä.

ASP.NET 3.5 -versioon Microsoft on laajentanut ja parantanut komponenttien kirjastoa sekä luonut lisää metodeja arkkitehtuuri-pohjaan. ASP.NET 3.5 on täysin yhteensopiva vanhemmilla arkkitehtuureilla luotujen sivujen kanssa ja se myös tukee 64-bittisyyttä.

## 2.5 C#.NET

C#.NET-ohjelmointikieli on yksinkertainen, täysin oliopohjainen ohjelmointikieli, joka on kehitetty C++:n ja Javan pohjalta. Se sisältää vain noin 80 varattua sanaa ja toistakymmentä tietotyyppiä. C#.NET-ohjelmointikielessä muistin hallintaa suorittaa CLR, mutta halutessaan ohjelmoija pystyy tekemään muistiviittauksia esimerkiksi olioihin, jolloin voidaan estää CLR:ä tuhoamasta olioita kunnes muistiviittaus on poistettu. Tällä tavoin esimerkiksi voidaan rakentaa toimivia sovelluksia jopa pienten resurssien ympäristöihin.

C#.NET-ohjelmointikieli on täysin oliopohjainen. Siinä jokainen ohjelmakoodisivu sisältää yhden tai useamman luokan. Luokkia voidaan periyttää kuten Javassakin sekä C#.NET-ohjelmointikieli tukee myös rajapintoja. C#.NET-ohjelmointikielessä käytetään kommentointiin normaalien *//*- ja */* */*-merkintöjen lisäksi myös XML-merkintäkieleen pohjautuvaa kommentointijärjestelmää metodeille. Microsoft Visual Studio osaa lukea nämä kommentit metodia kutsuttaessa ja näyttää ohjelmoijalle Intellisensen avulla kommentit ohjelmakoodia kirjoittaessa [5].

Luokista löytyy normaalisti ominaisuuksia, konstruktoreita, metodeja sekä harvemmin käytetty destruktori, joka suoritetaan kun olio poistetaan muistista.

Konstruktorit ovat luokan *alustajia*. Niitä kutsutaan kun luokasta luodaan esiintymä. C#.NET-ohjelmointikielessä, kuten Javassakin, ominaisuuksilla voi olla *ge*t- ja *set* -metodit tai vain toinen, jolloin ominaisuudesta tulee *vain luku* tai *vain kirjoitus* -ominaisuus.

C#.NET-ohjelmointikielestä löytyy erilaisia viittaustyyppejä, jotka kaikki periytyvät object-kantaluokasta. Näitä ovat esimerkiksi integer-kokonaisluvut, string-merkkijonot, double-liukuluvut, luokat, rajapinnat, taulukot, kokoelmat sekä delegaatit.

Viittaustyyppien näkyvyyksiä voidaan C#.NET-ohjelmointikielessä määrittää laajasti kuten Javassakin. C#.NET-ohjelmointikielen näkyvyystasot ovat private, protected, public, internal sekä protected internal. Näkyvyystasot toimivat nimensä mukaisesti, private-jäsenet näkyvät vain omassa luokassaan. Protected-jäsenet omassa ja perityissä luokissa, public-jäsenet ovat julkisia. Erikoistapauksia ovat internal-jäsenet, jotka näkyvät public-määreellä vain kyseessä olevassa nimiavaruudessa. Protected internal -jäsenet, jotka näkyvät protected-määreellä kyseessä olevassa nimiavaruudessa mutta private-määreellä, mikäli luokkaan referoidaan jostain toisesta nimiavaruudesta.

C#.NET-ohjelmointikielessä voidaan käyttää delegaatteja metodien osoittimina. Yhteen delegaattiin voidaan sijoittaa useampi metodi, jolloin delegaatin kutsuminen suorittaa kaikki siihen sijoitetut metodit. Delegaatteja käytetään usein asynkroniseen metodien suorittamiseen.

Operaattorien kuormitus on myös mahdollista C#:NET-ohjelmointikielessä. Esimerkiksi '+'-operaattori voidaan luokkakohtaisesti kuormittaa. Tätä ominaisuutta kuitenkin tarvitaan suhteellisen harvoin, yleensä vain matemaattisissa luokissa. Esimerkiksi mikäli arvoja kasvatetaan pareittain [6].

C#.NET-ohjelmointikielen ensimmäisen version 1.0 määritys esiteltiin joulukuussa 2001 ja keväällä 2010 uusin versio on 3.0, jonka määritys on julkaistu kesäkuussa 2005.

## 2.6 nHibernate [7]

nHibernate on .NET-arkkitehtuurille siirretty versio Hibernate ORM:stä. Hibernate ORM:n avulla ohjelmoija pystyy luomaan dynaamisesti kaikki tarvittavat luokat sekä SQL-kyselylauseet yksinkertaisen XML-merkintäkielellä luotavan tietokannan kuvauksen avulla.

nHibernate luo kuvauksen pohjalta jokaiselle tietokantataululle oman luokan, jossa on kaikki tietokantataulun tietokentät ominaisuuksina sekä tekee näille luokille SQL-kyselylauseet tiedon noutoa sekä tallentamista varten. nHibernaten luokkia voidaan täydentää kirjoittamalla lisää omia SQL-kyselylauseita nHibernaten metodien avulla. nHibernate tukee myös tietokantaproseduureja Microsoft SQL-palvelimen kanssa.

## 2.7 ADO.NET [8]

ADO.NET on .NET-arkkitehtuurin luokkakirjasto. Sen avulla pystytään muodostamaan yhteys useisiin tunnettuihin tietokantajärjestelmiin tiedonhallintaa varten käyttämällä yhteysjonoja. Yhteysjonot ovat tietokantakohtaisia merkkijonoja, joissa määritellään tietokannan osoite sekä käyttäjätiedot.

ADO.NET tarjoaa myös DataSet-luokan jota voidaan käyttää kuten nHibernate-kirjastoakin, joskin ohjelmoijan pitää DataSet-luokkaa kirjoittaessaan määrittää tietokantaproseduurit tai SQL-kyselylauseet tiedonkäsittelyä varten. Tämän jälkeen kuitenkin jokaisesta tietokantataulusta luodaan oma luokka, jolla on ominaisuuksina tietokantataulun kentät sekä perustoiminnot, tiedon lisäys, päivittäminen ja poistaminen.

Toinen vaihtoehto tiedonkäsittelylle on IDataReader-rajapinta. Sen avulla voidaan toteuttaa kannasta luku sekä kirjoitus. Rajapinnalla on neljä toteuttajaa .NET-arkkitehtuuriin sisäänrakennettuna, Oracle-, OleDb-, Odbc- sekä työssä käytetty SqlDatareader. Näillä toteuttajilla on erilaisten tietotyyppien lisäksi poikkeavat Get-metodit.

Työssä tietokantayhteys muodostetaan luomalla ensin esiintymä ensin yhteysluokasta, jolle kerrotaan yhteysjono. Tämän jälkeen luodaan esiintymä tietokannasta riippuen DataReader-toteuttajalle, jolle kerrotaan parametrit sekä SQL-kyselylause tai tietokantaproseduuri. Lopuksi suoritetaan Datareader-toteuttajan yhteysluokka, suoritetaan tiedon- ja paluuarvojen käsittely sekä suljetaan tietokantayhteys. Työssä tietokantayhteys ja DataReader-toteuttaja tehdään using-lohkossa, joten niitä ei tarvitse erikseen sulkea Close-metodilla.

## 2.8 Microsoft SQL Server 2005 [9]

Microsoft SQL Server 2005 on julkaistu lokakuussa 2005 koodinimellä ”*Yukon*”. SQL Server 2005 käyttää Transact SQL (T-SQL) – laajennusta SQL-kyselykielen kanssa. T-SQL on Microsoftin sekä Sybasen luoma laajennus perinteiseen SQL-kyselykieleen. Se mahdollistaa monien sisäänrakennettujen funktioiden käytön, kuten päivämäärien muokkauksen, ehtolauseet sekä väliaikaisten taulujen käytön.

Microsoft SQL-Server 2005:n etähallinta on mahdollista käyttämällä Microsoftin SQL Server Management Studiota. Microsoftin SQL Server Management Studiolla voidaan hallita tietokantatauluja, niiden suhteita ja indeksejä, suorittaa kyselyitä sekä kirjoittaa tietokantaproseduureja graafisessa käyttöliittymässä.

## 2.9 Tietokantaproseduurit [10]

Tietokantaproseduurit ovat ennalta kirjoitettuja SQL-kyselylauseita, joilla voidaan tehdä mitä tahansa tietokantatoimintoja SQL-kyselykielen rajoissa. Tietokantaproseduureille voidaan antaa parametreja, joilla voidaan esimerkiksi rajata tulosjoukkoa, lisätä tai päivittää tietoa tietokantatauluissa. Tietokantaproseduureilla saavutetaan myös parempi tietoturvallisuus, koska tietokantaproseduureissa voidaan luoda parametreille tarkistuksia. Esimerkiksi SQL-injektio on tietokantaproseduureilla vaikeaa.

Tietokantaproseduurit ovat erinomainen tapa hallita SQL-kyselylauseita verrattuna siihen että ne olisivat kirjoitettu ohjelmakoodin joukkoon. Mikäli esimerkiksi tavoitteena on hakea käyttäjät etunimen sijaan sukunimellä järjestettynä ja alkuperäinen lause on kirjoitettu ohjelmakoodiin, joudutaan koko ohjelmakoodi kääntämään ja lataamaan palvelimelle. Tietokantaproseduureilla tämän voisi muuttaa suoraan tietokantaproseduuriin, jolloin se olisi välittömästi käytettävissä ilman kääntämisiä.

Tietokantaproseduureissa voidaan T-SQL-laajennuksen avulla käyttää myös ehtolauseita joilla voidaan tulosjoukkoa rajata entistä tehokkaammin. Monimutkaisissa SQL-kyselylauseissa voidaan väliarvoja tulostaa esimerkiksi T-SQL-laajennuksen Print-funktiolla tai tarvittaessa muodostaa koko SQL-kyselylause esimerkiksi katenoimalla se parametrien sekä ehtolauseiden avulla.

T-SQL-laajennuksella voidaan myös käyttää kursoreita. Kursoreille määritellään SQL-kyselylause, jonka tulosjoukko luetaan ennalta määrättyihin muuttujiin. Tämän jälkeen kursori käy läpi jokaisen palautetun arvon silmukassa. Tämä on erinomainen esimerkiksi päivitettäessä tietoja tiettyjen ehtojen pohjalta.

# 3 Käytännön toteutus

Työ aloitettiin tutustumalla sovelluksen toimintaan jotta luokkia muutettaessa olisi paremmin tiedossa miten sovelluksen tulisi toimia. Tämän jälkeen purettiin nHibernate-tietokantakerros kirjoittaen samalla uutta tietokantakerrosta. Näiden yhteydessä, luokka kerrallaan, kirjoitettiin myös tarvittavat tietokantaproseduurit muutetuille luokille. Lopuksi käytiin läpi ohjelmalogiikka korvaamalla vanhat kutsut nHibernate-tietokantakerrokselle, kutsuilla uuteen tietokantakerrokseen. Testausten yhteydessä myös käyttöliittymää ja näyttölogiikkaa paranneltiin sijoittelemalla sivun elementtejä uudelleen, käytettävyyden parantamiseksi.

## 3.1 nHibernate-tietokantakerroksen purku ja uuden tietokantakerroksen kirjoitus

nHibernate-tietokantakerroksen purku toteutettiin tutustumalla vanhan kerroksen toimintoihin ja kirjoitettuihin erikoishakuihin. Samalla tarkasteltiin millaisia perushakulauseita tarvitaan riippuen tietokantataulusta. Pääsääntöisesti jokainen tietokantataulu tarvitsi ainakin SELECT-, INSERT-, UPDATE- ja DELETE-lauseet.

Tarvittavien toimintojen selvittämisen jälkeen, kirjoitettiin uusi tietokantakerros ja sille tarvittavat metodit. Samalla luotiin jokaiselle ohjelmalogiikan luokalle kutsut tietokantakerrokseen ja sitä kautta tietokantaproseduureille. Tietokantaan kirjoittaminen toteutettiin luomalla ohjelmalogiikan luokille Save()-metodit. Save()-metodit päättelevät kyseessä olevan olion id-ominaisuuden avulla onko kyseessä uusi tieto (INSERT) vai päivitys vanhaan (UPDATE).

## 3.2 Raportit ja erikoisluokat

Muutamia raportteja varten luotiin raporttikohtaiset luokat, jotka vastaavat ominaisuuksiltaan raportin rakennetta. Tällä tavoin voitiin kirjoittaa luokille omat tietokantaluokat ja palauttaa arvot tietokannasta suoraan erikoisluokkiin, jolloin saatiin vielä parannettua suorituskykyä ja vähennettyä muistin käyttöä. Normaalilla tavalla yhteen raporttiin olisi joissain tapauksissa pitänyt luoda esiintymä useasta luokasta, jotta kaikki tarvittavat tiedot olisivat olleet käytettävissä.

Uuden tietokantakerroksen yhteydessä kirjoitettiin myös tietokantatauluun kohdistuvat tietokantaproseduurit tiedonkäsittelyä varten. Joihinkin erikoistapauksiin, kuten raportteihin ja erikoisiin hakulauseisiin kirjoitettiin täysin omat tietokantaproseduurinsa suorituskyvyn parantamiseksi. Esimerkiksi raporteissa pitää usein laskea monia arvoja sekä noutaa tietoja useista tietokantatauluista, joten näiden suorittaminen ohjelmalogiikassa olisi aiheuttanut huomattavan monta pyyntöä tietokantapalvelimelle sekä ollut raskasta laskea ohjelmalogiikassa.

Useissa tapauksissa tietokannasta haettiin useampi kuin yksi rivi, joten jokaiselle datakerroksen luokalle luotiin myös oma kokoelma, jotta saadaan tietotyypiltään täsmällinen ja käyttöä helpottava säiliö, jossa kannasta haetulla tiedolla täytetyt oliot voidaan säilyttää. Kokoelmat toteutettiin käyttämällä ICollection-rajapintaa, joka esittelee valmiiksi muutamia hyödyllisiä metodeja, kuten lajittelu-, lisäys- sekä poisto-metodit.

## 3.3 Näyttölogiikka ja muut parannukset

Näyttölogiikkaa paranneltiin muuttamalla hieman asettelua joissain lomakkeissa, jotta saatiin paremmin toimiva asettelu. Joihinkin raportteihin lisättiin esimerkiksi laskutusta indikoivia tietoja sekä muita tarkempia tietoja.

Käyttöliittymään tehtiin useita parannuksia. Monissa lomakkeissa määriteltiin oletusnappi, jota voidaan kutsua tietoja täytettäessä painamalla rivinvaihto-näppäintä. Sivuston päävalikko muutettiin dynaamiseksi, jolloin valikoita ei tarvitse avata niitä klikkaamalla vaan pelkästään siirtämällä kursori valikon päälle.

Raporttien hakuehtojen valinta tehtiin reaaliaikaiseksi. Mikäli esimerkiksi tietoa haetaan käyttäjän ja päivämäärän avulla, päivämäärän valinta rajaa pois käyttäjät, joilla ei olisi tuloksia rajatulla aika-akselilla. Näin ollen se näyttää vain tarpeelliset käyttäjät. Kaikissa raporteissa, joissa on hakuehtona käyttäjä, pyritään tekemään oletusvalinnat kirjautuneen käyttäjän perusteella, jotta vältyttäisiin turhilta toistuvilta valintojen tekemisiltä.

Parantuneen suorituskyvyn ansioista voitiin useaan raporttiin lisätä esimerkiksi keskiarvo- ja summakenttiä, koska arvot voitiin palauttaa suoraan kannasta eikä niillä ollut olennaista vaikutusta suorituskykyyn.

Sovellukseen rakennettiin laskujen tyypitys, jotta saadaan tieto, mitkä laskut voidaan laskea työntekijöiden myynniksi. Tällä tavalla saadaan myös yrityksen asiakkaiden raportointijärjestelmän vuosimaksut eriteltyä. Maksetun vuoden lähestyessä loppuaan, lähetetään laskuttajalle muistutus uuden vuosilaskun lähettämisestä.

Etusivulle tuotiin myös tiedot työntekijän ja hänen alaistensa työtuntiliukuman tilanteesta kuitenkin niin, että ne ovat oletuksena piilotettuna. Tällä tavalla esimiestasoisen henkilön oli mahdollista avata sovellus ilman että tunnit olisivat näkyvillä. Samalla aloitussivulle lisättiin tieto tulevista vapaapäivistä, jotka noudetaan kannasta.

Jokaiseen sovelluksessa olevaan taulukkonäkymään kirjoitettiin lajittelutoiminto otsikkosarakkeeseen. Sarakkeen otsikkoa klikattaessa tulosjoukko lajitellaan otsikon mukaisesti joko numeeriseen järjestykseen tai aakkosjärjestykseen.

Sovelluksen liitoksia yrityksessä olevaan Super Office CRM – järjestelmään parannettiin, koska ominaisuus on aiemmin ollut todella hidas. Uudistus paransi suorituskykyä huomattavasti. Sillä voidaan noutaa erilaisia tapaamistietoja sekä niiden selitteitä. Toimintoa käytetään esimerkiksi myyntitapaamisten näyttämiseen henkilön perusteella tai jokaisen työntekijän myyntitapaamisten yhteenvetona. Toiminto toteutettiin siten, että sillä on mahdollista noutaa kaikentyyppisiä merkintöjä, mikäli tälle on tulevaisuudessa tarvetta.

Työtuntien kumulatiivinen laskeminen toteutettiin tietokantaproseduurilla, jolloin saatiin ohjelmalogiikassa tapahtuvaan laskemiseen minuuttien suorituskyvyllinen erotus.

Näiden muutosten lisäksi toteutettiin hitaimpiin raportteihin (yli 5 sekuntia) Microsoftin ASP.NET Ajax – komponentilla osittainen sivun päivitys sekä indikaattori, joka kertoo, milloin taulukkoa päivitetään. Tällä tavoin saatiin raportin lataaminen ulkoasullisesti siedettävämmäksi verrattuna selaimen omaan latausindikaattoriin.

## 3.4 Esimerkki työssä käytetyistä ohjelmointityyleistä

Esimerkissä toteutetaan yksinkertainen sovellus, jolla voidaan käsitellä autotietokantaa. Autoista kerätään tietokantaan rekisterinumero, malli sekä hinta. Näyttölogiikkaan ei esimerkissä ole luotu tallennustoimintoja.

Ohjelmakoodin esimerkki 1

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="ParkingLot._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Cars</title>
</head>
<body>
    <form id="CarsPage" runat="server">
    <div>
    <%--Luodaan Gridview-komponentti jolla voidaan autot näyttää--%>
        <asp:GridView ID="gvCars" runat="server" AutoGenerateColumns="False">
            <Columns>
                <asp:BoundField DataField="RegistryNumber"
                           HeaderText="RegistryNumber" />
                <asp:BoundField DataField="Model" HeaderText="Model" />
                <asp:BoundField DataField="Price" HeaderText="Price" />
            </Columns>
            <HeaderStyle BackColor="Blue" ForeColor="White" Font-Bold="True" />
            <RowStyle BackColor="Azure" />
            <AlternatingRowStyle BackColor="White" />
        </asp:GridView>
    </div>
    </form>
</body>
</html>
```

Ohjelmakoodin esimerkissä 1 esitetään yksinkertainen esimerkki näyttölogiikan ASP.NET www-sivusta. Ensimmäisellä rivillä määritetään sivun ominaisuuksia, kuten taustalle olevassa ohjelmakoodissa käytetty ohjelmointikieli ja sen tiedostonimi. Tämän jälkeen tyypitetään sivu, jotta selaimella voidaan mahdollisimman tarkasti piirtää sivun asettelu. ASP.NET-kielessä jokaisella sivulta pitää löytyä Form-elementti, jonka sisälle muut elementit sijoitetaan. Tässä esimerkissä on sivuille määritetty GridView-elementti, jolla voidaan näyttää tallennetut autot.

Ohjelmakoodin esimerkki 2

```
using System;

// Nimiavaruus jossa luokka sijaitsee
// Tällä voidaan asettaa luokkien näkyvyyttä muille luokille ja
// sitä voidaan käyttää viittauksissa

namespace ParkingLot
{
    public partial class _Default : System.Web.UI.Page
    {
        // Tapahtuma, joka suoritetaan aina kun sivu ladataan
        protected void Page_Load(object sender, EventArgs e)
        {
            // Varmistetaan että sivua ladataan ensimmäistä kertaa
            if (!IsPostBack)
            {
                // Haetaan tiedot Gridview-komponentille
                gvCars.DataSource = new ParkingLot().GetAllCars();
                // Kiinnitetään tulokset komponenttiin jolloin
                // ne tulostetaan ruudulle
                gvCars.DataBind();
            }
        }
    }
}
```

Ohjelmakoodin esimerkissä 2 esitetään ASP.NET www-sivun taustalla oleva ohjelmakoodi. Alussa määritellään .NET-arkkitehtuurin kirjastot, joihin ohjelmakoodissa viitataan. Seuraavaksi määritellään nimiavaruus, jossa ohjelmakoodi sijaitsee. Tällä voidaan määritellä ohjelmakoodin näkyvyyttä. `Page_Load()`-tapahtuma suoritetaan jokaisella kerralla, kun sivu päivitetään. ASP.NET www-sivut tekevät sivun päivityksen `PostBack`-tilassa, jokaisen sivun tapahtuman yhteydessä. Mikäli käyttäjä painaa esimerkiksi sivuilla olevaa nappia, tapahtuu sivun päivitys `PostBack`-tilassa. Tämän vuoksi voidaan vain kerran suoritettavaksi tarkoitettu ohjelmakoodi sijoittaa `if (!IsPostBack)` – ehtolauseen sisään. Tällä tavalla ohjelmakoodi suoritetaan vain sivuille siirryttäessä tai selaimen päivitys toiminolla. Esimerkissä ehtolauseeseen on sijoitettu GridView-elementin täyttö.

Ohjelmakoodin esimerkki 3

```
namespace ParkingLot
{
    public class Car
    {
        // Luodaan luokan ominaisuudet

        // Luodaan ensin luokalle yksityiset 'säiliöt' 
        ominaisuuksille
        private string _registryNumber;
        private string _model;
        private double _price;

        // Luodaan ominaisuuksille asettajat ja noutajat
        public string RegistryNumber
        {
            get { return this._registryNumber; }
            set { this._registryNumber = value; }
        }

        public string Model
        {
            get { return this._model; }
            set { this._model = value; }
        }

        public double Price
        {
            get { return this._price; }
            set { this._price = value; }
        }

        // Luodaan rakentajat luokalle.
        // Nämä suoritetaan kun luokasta
        // luodaan esiintymä
        public Car() { }

        public Car(string registryNumber)
        {
            this._registryNumber = registryNumber;
        }

        // Metodi jolla voi hakea auton tietokannasta
        public void Load()
        {
            DALCar dalCar = new DALCar(this);
            dalCar.Load();
        }
    }
}
```

Ohjelmakoodin esimerkissä 3 esitetään Car-luokka. Luokka on asetettu samaan nimiavaruuteen kuin ASP.NET www-sivun taustalla oleva ohjelmakoodi, jotta ne näkyvät toisilleen. Luokalle on ensin luotu private-määreellä ominaisuuksille muuttujat, joissa arvot säilytetään. Ominaisuudet noudetaan näistä muuttujista ja kirjoitetaan näihin muuttujiin. Tällä tavoin saadaan hallittua paremmin tietoturvaa ja näkyvyyttä. Ominaisuus voi koostua yhdestä tai useammasta muuttujasta, jolloin voidaan ominaisuutta noudettaessa esimerkiksi laskea kaksi arvoa yhteen. Luokalle on kirjoitettu kaksi konstruktoria. Ensimmäistä kutsutaan ilman parametreja ja se luo tyhjän esiintymän luokasta. Toinen saa parametrina auton rekisterinumeron ja esiintymää luodessa sijoittaa sen _registryNumber-muuttujaan. Luokalla on myös Load()-metodi. Metodi luo esiintymän DALCar-luokasta, jossa Car-luokan viittaus on ominaisuutena. Tämän jälkeen metodi kutsuu DALCar-luokan Load()-metodia.

Ohjelmakoodin esimerkki 4

```
// Tuodaan kokoelmien peruskirjastot
using System.Collections;

namespace ParkingLot
{
    public class ParkingLot : CollectionBase
    {
        public ParkingLot() : base() { }

        public void Remove(Car car)
        {
            List.Remove(car);
        }

        public int Add(Car car)
        {
            return List.Add(car);
        }
        // Tehdään metodi jolla voidaan hakea autoja indekseillä
        public Car this[int index]
        {
            get { return ((Car)List[index]); }
            set { List[index] = value; }
        }

        // Metodi jolla voidaan hakea kaikki autot tietokannasta
        public ParkingLot GetAllCars()
        {
            DALCar dalCar = new DALCar();
            return dalCar.GetAllCars();
        }
    }
}
```

Ohjelmakoodin esimerkissä 4, luodaan kokoelma Car-olioille. Kokoelma perii CollectionBase-luokan. Tällä tavoin sillä on käytössä kaikki kantaluokan ominaisuudet ja metodit. Kokoelmalle kirjoitettiin vielä Remove()-, Add()-metodit sekä metodi jolla olioihin voidaan viitata indeksillä. Remove()- sekä Add()-metodeilla voidaan poistaa tietty Car-olio ja lisätä Car-olioita kokoelmaan. Normaalisti olioita vertailtaessa olion koko sisällön pitää olla identtinen jotta vertailu onnistuu. Mikäli halutaan kuitenkin vertailla autoja vaikka pelkän rekisterinumeron perusteella pitää Car-luokan toteuttaa IComparable-rajapinta. Lopuksi kokoelmalle on luotu GetAllCars()-metodi, jolla voidaan Car-luokan Load()-metodin kanssa samalla logiikalla noutaa kannasta kaikki autot kerralla. Tässä on kuitenkin erona se että, DALCar-luokka ei saa viittausta kyseessä olevaan olioon.

Ohjelmakoodin esimerkki 5

```
// Tuodaan ADO.NET-kirjastot
using System.Data;
using System.Data.SqlClient;

namespace ParkingLot
{
    public class DALCar
    {
        public DALCar() { }

        public DALCar(Car car)
        {
            this.Car = car;
        }

        private Car _car;
        public Car Car
        {
            get { return _car; }
            set { _car = value; }
        }
```

Ohjelmakoodin esimerkissä 5 luodaan DALCar-luokka, sen konstruktorit sekä Car-tyyppinen ominaisuus. DALCar-luokkaan on tuotu System.Data.SqlClient-kirjasto, jotta .NET-arkkitehtuurin tietokanta-metodeja voidaan kutsua. System.Data.SqlClient-kirjasto tunnetaan paremmin ADO.NET-kirjastona.

Ohjelmakoodin esimerkki 6

```
// Ladataan auto rekisterinumeron perusteella
        public void Load()
        {
            using (SqlConnection oconn =
                new SqlConnection(System.Configuration.ConfigurationManager
                     .AppSettings["SqlConnect"]
                ))
            {
                using (SqlCommand ocmd = oconn.CreateCommand())
                {
                    // Asetetaan proseduurin nimi
                    ocmd.CommandText = "GetCarByRegistryNumber";
                    ocmd.CommandType = CommandType.StoredProcedure;
                    // Annetaan proseduurille tarvittavat parametrit
                    ocmd.Parameters.AddWithValue("@registryNumber",
                        this.Car.RegistryNumber);

                    // Muodostetaan yhteys
                    oconn.Open();

                    try
                    {
                        // Suoritetaan proseduuri
                        using (SqlDataReader oreader = ocmd.ExecuteReader())
                        {
                            if (oreader != null && oreader.HasRows)
                            {
                                oreader.Read();
                                // Luetaan tulokset
                                if (!oreader.IsDBNull(0))
                                    this.Car.Model = oreader.GetString(0);
                                if (!oreader.IsDBNull(1))
                                    this.Car.Price = oreader.GetDouble(1);

                                // Lopuksi suljetaan yhteys tietokantaan
                                oreader.Close();
                            }
                        }
                    }
                    // Tarkastetaan tuliko virheitä
                    catch (SqlException ex) { }
                }
            }
        }
```

Ohjelmakoodin esimerkissä 6 luodaan `DALCar`-luokalle `Load()`-metodi. Metodissa luodaan `using`-lohkossa uusi `oconn`-niminen `SqlConnection`-olio. `SqlConnecti-on`-olion konstruktori saa parametrina yhteysjonon, joka tässä tapauksessa noudetaan sovelluksen asetuksista `web.config`-tiedostosta. Seuraavaksi luodaan `SqlCommand`-olio `ocmd`, `using`-lohkossa. `SqlCommand`-olion ominaisuuksiin laitetaan tietokantaproseduuri jota kutsutaan sekä tietokantaproseduurin parametrit, tässä tapauksessa auton rekisterinumero. `oreader` nimisen `SqlDataReader`-olion avulla luetaan tietokannasta saatu vastaus `DALCar`-olion ominaisuutena olevaan `Car`-olioon. Suljetaan yhteys ja tarkastetaan `try-catch`-lauseella tapahtuiko lukemisen aikana virheitä. Metodin suo-

rittamisen jälkeen tietokannasta saadut arvot ovat `Car`-olion ominaisuuksia, ja ne ovat ohjelmalogiikan käytettävissä.

Ohjelmakoodin esimerkki 7

```
// Metodi jolla voidaan hakea kaikki autot tietokannasta
        public ParkingLot GetAllCars()
        {
            // Luodaan kokoelma johon vastausluokat sijoitetaan
            ParkingLot carCollection = new ParkingLot();
            using (SqlConnection oconn = new SqlConnection(
               System.Configuration.ConfigurationManager.AppSettings["SqlConnect"]))
            {
                using (SqlCommand ocmd = oconn.CreateCommand())
                {
                    ocmd.CommandText = "GetAllCars"; //Tähän Proseduurin nimi
                    ocmd.CommandType = CommandType.StoredProcedure;

                    oconn.Open();

                    try
                    {
                        using (SqlDataReader oreader = ocmd.ExecuteReader())
                        {
                            if (oreader != null && oreader.HasRows)
                            {
                                while (oreader.Read())
                                {
                                    Car car = new Car();

                                    if (!oreader.IsDBNull(0))
                                        car.RegistryNumber = oreader.GetString(0);
                                    if (!oreader.IsDBNull(1))
                                        car.Model = oreader.GetString(1);
                                    if (!oreader.IsDBNull(2))
                                        car.Price = oreader.GetDouble(2);
                                   // Sijoitetaan vastauksena saatu luokka
                                   kokoelmaan
                                    carCollection.Add(car);
                                }
                                oreader.Close();
                            }
                        }
                    }
                    catch (SqlException ex) { }
                }
            }
            // Palautetaan kokoelma
            return carCollection;
        }
    }
}
```

Ohjelmakoodin esimerkissä 7 luodaan `DALCar`-luokalle `GetAllCars()`-metodi, jolla voidaan noutaa kaikki autot tietokannasta. Metodin runko toimii lähes samalla tavalla kuin `Load()`-metodi, sillä erotuksella että tietokannasta saatua vastausjoukkoa ei lueta viittauksena saatuun olioon. Tässä metodissa tietokannasta saadun vastauksen lukeminen tehdään `while`-silmukassa. Silmukan alussa luodaan uusi esiintymä `Car`-luokasta parametrittoman konstruktorin kanssa. Tämän jälkeen tietokannasta saatu vastausrivi luetaan luodun olion ominaisuuksiin ja silmukan lopuksi olio lisätään meto-

din alussa alustettuun `ParkingLot`-kokoelmaan. Kokoelma palautetaan metodia kutsuneelle oliolle, kun kaikki vastaukset on luettu.

Ohjelmakoodin esimerkki 8

```
-- Luodaan proseduuri, jolla voidaan
-- hakea autoja rekisterinumeron
-- perusteella
CREATE PROCEDURE GetCarByRegistryNumber
    @registryNumber varchar(20)
AS
BEGIN
    SELECT
        Model,
        Price
    FROM
        Cars
    WHERE
        RegistryNumber = @registryNumber
END
```

Ohjelmakoodin esimerkissä 8 luodaan tietokantaproseduuri joka saa parametrina rekisterinumeron ja noutaa tietokannasta rekisterinumeroa vastaavan auton.

Ohjelmakoodin esimerkki 9

```
-- Luodaan proceduuri joka
-- hakee kaikki autot
CREATE PROCEDURE GetAllCars
AS
BEGIN
    SELECT
        RegistryNumber,
        Model,
        Price
    FROM
        Cars
END
```

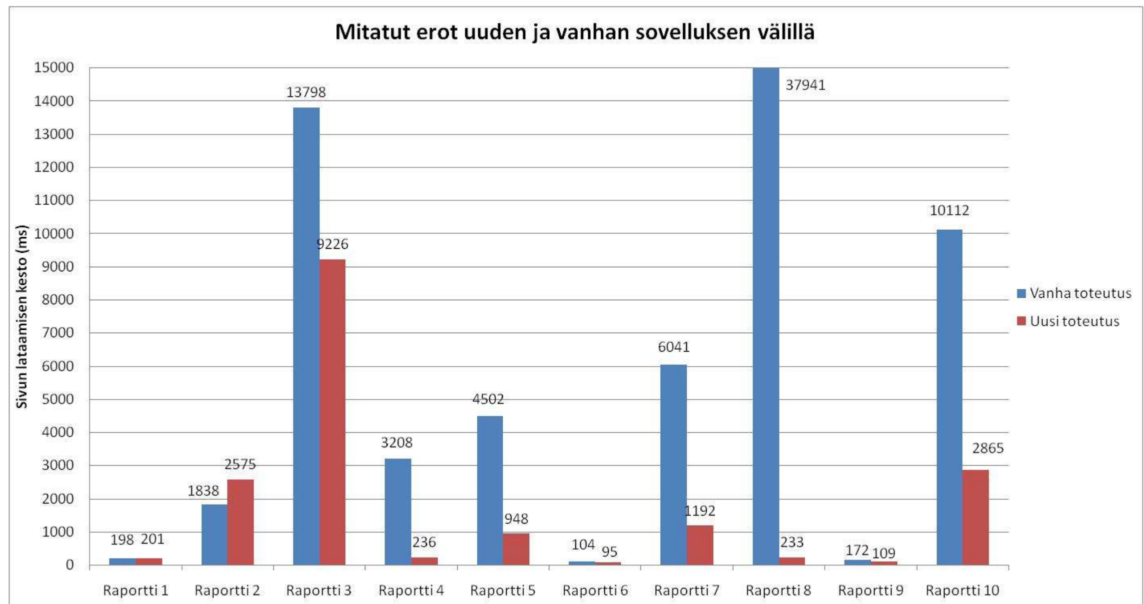
Ohjelmakoodin esimerkissä 9 luodaan tietokantaproseduuri joka noutaa tietokannasta kaikki autot.

# 4 Tulokset

Sovelluksen ensimmäisen julkaisun jälkeen suorituskykyä testattiin *Mozilla Firefox 3.6* -selaimella käyttäen Microsoft .NET-arkkitehtuurin 3.5 SP1 -versiota sekä *Extended Status Bar 1.5.4* -lisäosaa ajan mittaamiseen. Tämä lisäosa näyttää kuluneen ajan sivunvaihdon alkamisesta sivun latauksen loppuun. Molemmat sovellukset sijaitsivat rinnakkain samalla palvelimella. Tällä tavoin palvelimen asetukset eivät aiheuttaneet eroja mittaustuloksiin. Testit päätettiin keskittää raportteihin, jotka ovat aiemmin olleet sovelluksen hitaimpia toimintoja. Ne korostavat kuitenkin parhaiten vanhan ja uuden tietokantakäsittelyn eroja (taulukko 4.1) (Kuva 4.1). Yhteensä sovellus nopeutui raporttien osalta noin 77 %.

**Taulukko 4.1** Mitatut erot uuden ja vanhan sovelluksen välillä

| | *Vanha toteutus (ms)* | | | | *Uusi toteutus (ms)* | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ajo 1 | Ajo 2 | Ajo 3 | *Keskiarvo* | Ajo 1 | Ajo 2 | Ajo 3 | *Keskiarvo* | *Erotus* |
| *Raportti 1* | 213 | 208 | 173 | *198* | 208 | 199 | 195 | *201* | 1,52 % |
| *Raportti 2* | 1897 | 1827 | 1791 | *1838* | 2557 | 2571 | 2597 | *2575* | 40,10 % |
| *Raportti 3* | 13704 | 13827 | 13863 | *13798* | 9216 | 9305 | 9157 | *9226* | -33,14 % |
| *Raportti 4* | 3201 | 3179 | 3245 | *3208* | 224 | 231 | 254 | *236* | -92,64 % |
| *Raportti 5* | 4424 | 4414 | 4668 | *4502* | 1206 | 819 | 820 | *948* | -78,94 % |
| *Raportti 6* | 109 | 100 | 104 | *104* | 107 | 90 | 89 | *95* | -8,65 % |
| *Raportti 7* | 6036 | 6069 | 6019 | *6041* | 1207 | 1186 | 1184 | *1192* | -80,27 % |
| *Raportti 8* | 37455 | 37685 | 38684 | *37941* | 221 | 239 | 238 | *233* | -99,39 % |
| *Raportti 9* | 159 | 187 | 171 | *172* | 110 | 124 | 93 | *109* | -36,63 % |
| *Raportti 10* | 10086 | 10121 | 10130 | *10112* | 2861 | 2838 | 2895 | *2865* | -71,67 % |




**Kuva 4.1** Mitatut erot uuden ja vanhan sovelluksen välillä.

## 4.1 Testaus

Sovellusta testattiin monipuolisesti testausta varten luotua tietokantaa vasten ja havaitut virheet korjattiin. Sovelluksen ensimmäinen versio julkaistiin vanhan version rinnalle kuitenkin siten, että niillä on yhteinen tietokanta. Tällä tavoin käyttäjät voivat työn häiriintymättä käyttää uutta sovellusta ja ongelma tilanteessa suorittaa toimenpiteen käyttäen vanhaa versiota. Tämä ratkaisu antaa ohjelmoijalle liikkumavaraa virheiden korjaukseen ja korjausaikatauluun. Vanha versio poistetaan käytöstä kun kaikki ominaisuudet ja korjaukset ovat valmiit.

## 4.2 Virheiden hallinta ja niiden korjaus

Virheiden varalle sovellukseen ohjelmoitiin virheidenkäsittelijä, joka ottaa talteen virheet tekstitiedostoon palvelimelle. Tällöin käyttäjän ei tarvitse kirjoittaa virhetietoja muistiin. Virheen paikantaminen on näiden tietojen perusteella paljon helpompaa.

Useat esiintyneet virheet ovat johtuneet siitä, että kannasta tullut tulosjoukko ei ole vastannut määritettyä. Toinen yleinen virheen aiheuttaja on ollut alustamattomat oliot.

Kaikkien lomakkeiden hakuehtoihin on tehty varmistukset tietojen oikeellisuudesta. Tämä ominaisuus jäi vähälle huomiolle alkuperäistä versiota tehdessä, koska nHibernatessa on sisäänrakennettuna hallinta väärän tyyppisille hakuehdoille, kuten esimerkiksi tyhjille valintalistoille.

## 4.3 Versionti

Sovellus on versioitu ”*major.minor.revision.revision*”-periaatteella. Mikäli sovellukseen lisätään uusia ominaisuuksia, kasvatetaan ”*minor*”-lukua yhdellä, tai mikäli kyseessä on laajat uudistukset, kasvatetaan ”*major*”-lukua. Korjauksia julkaistaessa kasvatetaan niiden laajuudesta riippuen jompaakumpaa ”*revision*”-lukua.

Sovelluksen versio oli työtä aloitettaessa *2.0.0.1*. Ensimmäisessä uudessa julkaisussa päätettiin kasvattaa versiota *3.0.0.0*:aan.

# 5 Päätelmät ja tulevaisuus

Mielestäni suorituskyvylliseen tavoitteeseen päästiin erinomaisesti ja sovitussa aikataulussa. Sovellusta olisi pitänyt testata enemmän ennen ensimmäistä julkaisua, mutta rinnakkain julkaisemisella vältettiin sovelluksen virheistä johtuva työn vaikeutuminen. Kaikki pyydetyt raporttien lisäominaisuudet saatiin toteutettua, sekä käyttöliittymän parannuksista on saatu hyvää palautetta.

Mielestäni projekti oli kannattava, koska parantuneella suorituskyvyllä ja käyttöliittymällä säästetään jokaisen sovellusta käyttävän työntekijän aikaa. Kun tätä peilaa projektin kuluihin, jotka ovat käytännössä ohjelmoijan henkilöstökulut, projekti todennäköisesti maksaa itsensä takaisin vuoden sisällä säästyneen työajan ja laajennettavuuden takia.

Ohjelmointilogiikka on sovelluksessa projektin jälkeen samanlainen kuin yrityksen muissakin sovelluksissa. Tämä säästää kuluja muihin sovelluksiin perehdytyksestä.

Sovellusta jatkokehitetään tulevaisuudessa aktiivisesti. Sovellukseen tullaan lisäämään ainakin matkalaskutoiminto, jolla työntekijät voivat hallita matkalaskujaan. Käyttöliittymää parannetaan käytännössä havaittujen puutteiden ja tarpeiden mukaan.

# LÄHTEET


[1] Mains, B., *Introduction to 3-Tier Architecture*, [www-dokumentti], Saatavilla: http://dotnetslackers.com/articles/net/IntroductionTo3TierArchitecture.aspx (luettu: 20.3.2010)

[2] Beres, J., *Teach Yourself Visual Studio .NET 2003*. SAMS Publishing, s. 10 – 27, 2003

[3] Powers, L. ja Snell, M., *Microsoft ® Visual Studio 2008 UNLEASHED*. SAMS Publishing, 2008

[4] Andrew, Duthie G. ja MacDonald, M., *ASP.NET in a Nutshell, 2nd Edition*. O'Reilly, 2003

5] Stellman, A. ja Greene, J., *Head First C#*. O'Reilly Media, Inc., 2007

[6] MacDonald, M., *Beginning ASP.NET 3.5 in C# 2008: From Novice to Professional, Second Edition.* Apress, 2007

[7] [www-dokumentti]. Saatavilla: http://nhforge.org (luettu: 21.3.2010)

[8] Beres, J., *Teach Yourself Visual Studio .NET 2003*. SAMS Publishing, s.293 – 333, 2003

[9] [www-dokumentti] Saatavilla: http://www.microsoft.com/sqlserver/2005/ (luettu: 25.3.2010)

[10] Turley, P. ja Wood, D., Beginning T-SQL with Microsoft® SQL Server® 2005 and 2008. Wiley Publishing, Inc., 2009