Teemu Ihalainen

TALOYHTIÖN VUOSIKERTOMUS

Työkalun lisääminen Tampuuriin

Opinnäytetyö
Tietotekniikan koulutusohjelma

Toukokuu 2013

MIKKELIN AMMATTIKORKEAKOULU
Mikkeli University of Applied Sciences

KUVAILULEHTI

MIKKELIN AMMATTIKORKEAKOULU Mikkeli University of Applied Sciences	**Opinnäytetyön päivämäärä** 30.5.2013
Tekijä(t) Teemu Ihalainen	**Koulutusohjelma ja suuntautuminen** Tietotekniikan koulutusohjelma
Nimeke Taloyhtiön vuosikertomus, työkalun lisääminen Tampuuriin	

Tiivistelmä

Opinnäytetyön tavoitteena oli etsiä toimiva ohjelmarakenne jolla voidaan työkaluun myöhemmin lisätä helposti tarvittavia laajennusosia ja ominaisuuksia. Opinnäytetyössä tutkittiin käytännössä toimivinta rakennetta. Yksinkertaisin ja toimivin rakenne valittiin ja sitä optimoitiin mahdollisuuksien mukaan niin että sitä voidaan käyttää tulevaisuudessa pohjana vastaaville työkaluille.

Työssä tehtiin Microsoftin .NET-ympäristöön Visual Studiossa uusi työkalu Agenteq Solutions oy:n Tampuuri-ohjelmistoon. Tavoitteena oli lisätä ohjelmistoon taloyhtiöitä varten vuosikertomus-työkalu, joka täyttää automaattisesti käyttäjän valmistelemaan Word-pohjaan Tampuurin sisältämät tiedot vuosikertomusta varten. Työn lopuksi tehtiin työkalusta demo testi-Tampuuriin, jossa työkalua testattiin käytännössä.

Työn aikana havaittiin .NET-ympäristössä käytetty tiedontuottaja/tiedonkuluttaja-malli toimivimmaksi. Työkalu toteutettiin käyttäen C#:ia ja tietokantatoiminnot siihen sisältyvällä Linq:lla. Tiedontoimittaja sijoitettiin kirjastoon johon myöhemmin lisätään tarvittavat luokat tietojen hakua varten. Käyttöliittymä tehtiin mahdollisimman yksikertaiseksi, ja suurin osa toiminnoista rakennettiin taustakoodiin luokkina.

Lopputulos vastasi odotuksia, vaikka useita muutoksia suunnitelmaan tulikin. Työkalun rakenne on yksinkertainen ja helposti päivitettävä, ja sitä laajennetaan tulevaisuudessa asiakkaiden tarpeiden mukaan. Suunnitelmissa on myös käyttää työkalua pohjana vastaaville projekteille.

Asiasanat (avainsanat)
Tietotekniikka, Ohjelmointi, C#, Tampuuri

Sivumäärä	**Kieli**	**URN**
47	Suomi	

Huomautus (huomautukset liitteistä)

Ohjaavan opettajan nimi	**Opinnäytetyön toimeksiantaja**
Jukka Selin	Agenteq Solutions Oy

DESCRIPTION

MIKKELIN AMMATTIKORKEAKOULU
Mikkeli University of Applied Sciences

Date of the bachelor's thesis

30 May 2013

Author(s)
Teemu Ihalainen

Degree programme and option
Information Technology

Name of the bachelor's thesis

Housing company's annual report: a new tool to the Tampuuri software

Abstract

The aim of the thesis was to find a functional programming model where users could later easily add the necessary expansions and properties. The thesis examined in practice what the most functional model for this case was. The best model was chosen and it was optimized and fine-tuned so that it could also be used as a template for similar tools later on.

The work was done by using Microsoft's .NET framework and with Visual Studio development platform. The aim was to develop a new tool for Agenteq Solutions Oy, and their Tampuuri program. The purpose of the tool was to help a housing company make their annual report by using the existing data in Tampuuri and a preloaded Word template. The final stage was to make a demo of the tool into the Tampuuri test platform where it was tested in practice.

The study indicated that Microsoft's .NET framework data provider/data consumer model was the most efficient and simple. The tool was implemented with C# language and the database functions with its Linq extension. The data providers were written to a library project where new providers could easily be written without changing the structure. The user interface was made with the simplest possible model, and most of the functionality was built into the codebehind.

There were many changes to the program during the development, but the end result was considered good. The program structure is simple and easily upgradeable, and the tool will be expanded in future to meet Tampuuri clients' wishes. It will also be used as a template for similar projects within Tampuuri.

Subject headings, (keywords)

Information Technology, IT, Programming, C#, Tampuuri

Pages	Language	URN
47	Finnish	

Remarks, notes on appendices

Tutor	Bachelor's thesis assigned by
Jukka Selin	Agenteq Solutions Oy

SISÄLTÖ

AVAINSANALISTA

Aspose.Words

Aspose Corporate:n kehittämä kirjasto .NET-ympäristöön jolla voidaan käsitellä Word-dokumentteja ohjelmakoodilla.

ADO.NET

Microsoftin .NET-ympäristön tarjoama rajapinta tiedonhaulle. Toteutettu yleisesti Linq:lla tai Entity FrameWork:llä.

CallBack-funktio (CallBack function)

CallBack-funktio annetaan jonkin muun ohjelman tai osan hallintaan. Esim. tapahtumanlaukaisin (event trigger) on sijoitettu käyttöliittymään (esim. web-selain) joka käynnistää funktion kun tapahtuman laukaisin aktivoituu.

Eager load

Tiedot tai toiminnot ladataan välittömästi käynnistyksen yhteydessä.

IDE (integroitu kehitysympäristö)

Integrated Developement Environment. Kehitystyökalu jolla voidaan kehittää ohjelmia usealle eri alustalle ja monella eri ohjelmointikielellä.

IntelliSense

Microsoftin Visual Studiossa oleva työkalu joka analysoi koodia ja auttaa ohjelmoijjaa. Se ilmoittaa esim. syntaksin virheistä ja tarjoaa ratkaisuja virheisiin. Se myös tarjoaa automaattisesti kaikki käyttökelpoiset luokat ja metodit.

Kyselymerkkijono (querystring)

Merkkijono joka sisältää komennot tiedonhakuun esim. SQL-tietokannasta.

Kirjasto (libarary)

.dll-muotoinen tiedosto jossa on yleisesti käytössä olevia luokkia.

Legacy-koodi

Vanhaa koodia joka on kehitetty vanhalla tyylillä ja tekniikalla.

Lazy load

Tiedot tai toiminnot ladataan vasta ensimmäistä kertaa tarvittaessa.

Lambda-lause

Antaa mahdollisuuden käyttää nk. anonyymejä funktioita. Esim. Linq:ssa voidaan käydä taulukon kaikki tiedot läpi yhdellä lambda-lauseella eikä että niitä tarvitse erikseen nimetä tai määritellä.

.NET

Microsoftin kehittämä ohjelmointiympäristö.

ORM-rajapinta

Object Relational Mapping. Ohjelman ja tietokannan väliin ohjelmoitava rajapinta joka jossa on tietokannan entiteetit valmiissa luokissa ohjelman käytettävänä.

Proseduuri (procedure)

Sql-palvelimeen ohjelmoitu skripti jolla toimintoja voidaan tehdä palvelimella. Ajamalla proseduurin voi yhdellä komennolla tehdä suuren määrän toimintoja kuten hakuja, suodatuksia, tyypitystä tai laskentaa.

Suunnittelumalli (design pattern)

Ratkaisumalli ongelmaan tai toimintoon. Mallit ovat yleensä ongelmakohtaisia nk. hyväksi todettuja toteutuksia.

TDD

Test Driven Developement. Ensiksi tehdään ohjelman toteuttavat testit, jonka jälkeen kirjoitetaan ohjelma niin että testit läpäistään.

Tuoteomistaja (productowner)

Scrum-ohjelmointimallin mukaan tuotteen kehityksestä ja laadusta vastaava henkilö on tuoteomistaja.

Toistin (repeater)

Repeater on .aspx:n elementti joka tulostaa järjestyksessä mallipohjaa (template) käyttäen kaikki tietokokoelman (esim. List, Array tai Icollection) tiedot sivulle.

Yhteysmerkkijono (connectionstring)

Palvelinyhteyden avaamiseen tarvittava tieto.

Web-projekti

Projektin osa johon .aspx-tyyppinen käyttöliittymä taustakoodeineen on tehty .NET-ympäristössä.

1 JOHDANTO

Kun valitsin opinnäytetyöni aihetta, halusin sen olevan jokin oikea toimeksianto joka tehtäisiin käytettäväksi eikä pelkästään esittelyä varten. Onnekkaasti pääsinkin marraskuussa 2012 töihin Agenteq Solutions Oy:lle, ja he antoivat minulle hyvän ja haasteellisen toimeksiannon opinnäytetyöhöni.

Toimeksianto liittyy Agenteqin kehittämään Tampuuri-ohjelmaan, jolla suurinta osaa suomen asuinkiinteistöistä hallinnoidaan. Taloyhtiöt velvoitetaan tekemään kerran vuodessa toimintakertomus edelliseltä vuodelta, ja tähän asti se on tehty käsin, vaikka Tampuurin järjestelmässä olisi kaikki kiinteistöä koskevat tiedot valmiina.

Sain tehtäväkseni kehittää Tampuuriin työkalun, jolla vuosikertomuksen voisi tehdä valmiiksi täytetyin tiedoin niin, että se myös tallentuu Tampuuriin ja on siellä muokattavissa. Työkaluun tehdään muutaman sivun yksinkertainen käyttöliittymä jolla Word-dokumentti voidaan ladata Tampuuriin, ja tämän jälkeen käyttöliittymässä nähdä siihen automaattisesti tulevat tiedot sekä muokata niitä, jos kaikkea tietoa ei ole tai jokin tieto ei ole ajan tasalla. Tämän jälkeen tiedot voidaan tallentaa Word-dokumenttiin käyttäen alkuperäistä pohjaa mallina. Word-pohjaan on merkattu tiedon paikoille ennalta määrättyjä kirjanmerkkejä joihin tietoa täytetään. Tunnistamattomiin kirjamerkkeihin kirjoitetaan tyhjää tai käyttäjän kirjoittamaa tietoa. Kaikki tieto ja dokumentit tallentuvat Tampuuriin ja tallennetut tiedot sekä dokumentit ovat helposti käytettävissä myöhemmin.

Tavoitteena oli kehittää toimiva rakenne jota olisi helppo myöhemmin laajentaa tai käyttää mallina vastaaville toiminnoille. Rakenteen tulisi olla yhteensopiva aiempien Tampuurin osien kanssa niin, että eri moduulien kehittäjät voivat vapaasti kirjoittaa omaa moduuliaan koskevat tiedontoimittajat, ilman että heidän tarvitsee muuttaa ohjelman rakennetta tai toimintoja. Tämä tarkoittaa myös sitä että tiedontoimittajien toimittamaa tietoa ei validoida tai tyypitetä muualla kuin tiedontoimittajan-luokassa, käyttöliittymän taustakoodi vain näyttää tiedon. Käyttöliittymästä katsottuna tiedontoimittajat sisältävälle luokalle toimitetaan vain kirjanmerkin nimi. Vastaukseksi saadaan oikea tieto jos tiedontoimittaja on olemassa, jos tiedon toimittajaa ei ole kirjoitet-

tu saadaan vastaukseksi tyhjätieto (empty). Tallennettava tieto validoidaan käyttöliittymässä ja sen taustakoodissa.

Työkalun kehitys alkaa tyhjältä pöydältä. Kehitys etenee vaiheittain ja aina seuraavan vaiheen valmistuttua sen tarkastaa ja arvioi työni ohjaaja Mika Karjunen, joka antaa myös lisä- ja parannusohjeita. Työtä jatketaan vaihe vaiheelta kunnes rakenne ja toiminnot ovat riittävän hyviä demon rakentamiseen. Tämän jälkeen teen demon joka integroidaan Tampuuriin ja arvioidaan.

Työssä joudun perehtymään syvällisesti .NET-ohjelmointiin ja Tampuuriin. Työ tehdään käyttäen asp.net-, C#- ja LinqToSql-tekniikoita. Koska olen aiemmin perehtynyt vain pinnallisesti ASP.NET:n ja Linq:hun, tulen oppimaan niistä paljon uutta. Lisäksi osaamiseni C#:lla ja moderneissa ohjelmistomalleissa (design patterns) on vanhentunutta, joten uudistan ja parannan osaamistani työllä merkittävästi. Tarvitsen projektissani myös osaamista Tampuurista jota tulen kehittämään työkseni projektin jälkeen, ja uskon oppivani paljon myös sen toiminnasta ja tekniikasta.

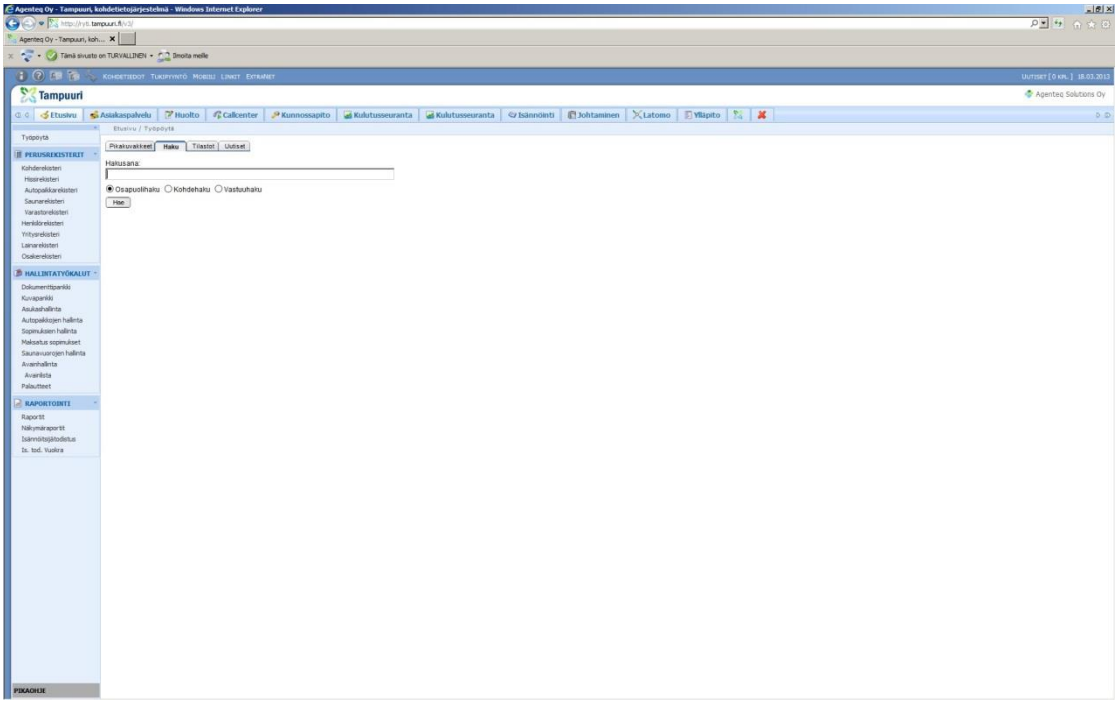
Saan tehdä työni alusta loppuun työaikana, mistä on suuri apu koska kaikki katkokset ja häiriöt hidastavat ja vaikeuttavat kehitystyötä. Saan tukea työni teossa aina tarvitessani, mutta tarkoitus on että teen työni itsenäisesti.

Uskon projektini parantavan ammattitaitoani merkittävästi ja opettavan minulle paljon uutta. Koen ennen kaikkea uudet mallit ja rakenteet haasteellisiksi, ja siksi tärkeäksi oppia. Oman ammattitaidon parantamisen lisäksi projektista on hyötyä myös työssäni, sillä vastaavat uudet osat voivat käyttää työkaluani mallina. Ja koska rakenne pidetään yksinkertaisena, on mallia helppo soveltaa muunkinlaisissa projekteissa.

2 TAMPUURI

Tässä luvussa kerron Tampuuri-ohjelmistosta joka on Agenteq Solutions oy:n päätuote ja projektini sekä nykyisen työni kohde. Kerron lyhyesti Tampuurin historiasta ja tekniikasta, sillä Tampuuri-kokonaisuus on itsessään vaativa ja monimutkainen ympäristö jota on kehitetty kauan ja monin eri tavoin. Tämän takia ohjelmistokehitys

Tampuuriin vaatii sekä tulevien tarpeiden ennakointia että vanhan tekniikan huomioon ottamista.



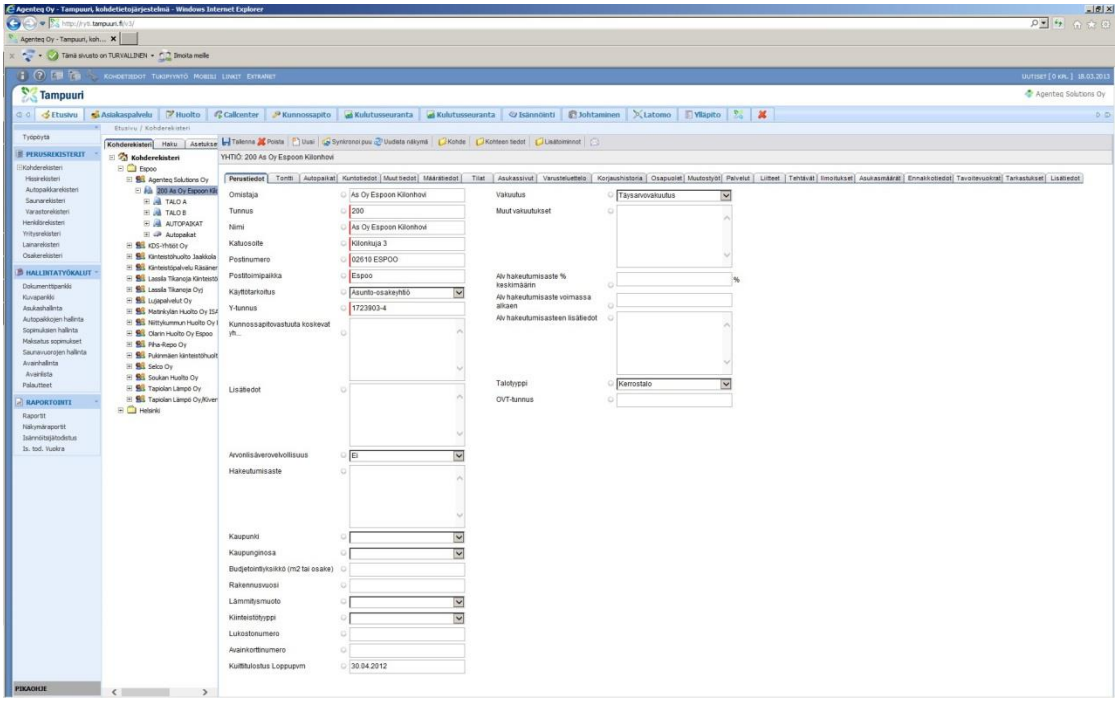
Kuva 1. Tampuurin etusivu

Tampuuri on kiinteistöjen hallintaan erikoistunut ohjelmisto (Kuva 1). Ensimmäinen Tampuurin versio tuli markkinoille 2000-luvun alussa minkä jälkeen on se monipuolisuudellaan ja dynaamisuudellaan ottanut merkittävän markkinaosuuden toimialalla. Vuosien varrella Tampuuri on laajentunut ja kehittynyt. Nykyisin se muodostuu 35 moduulista kattaen kaikki kiinteistöjen ja asukkaiden hallintaan vaadittavat osa-alueet. Tampuuria käyttävät kiinteistöjen omistajat, isännöitsijät, huoltoyhtiöt, hallintayhtiöt ja tietenkin Tampuurilla hallinnoitavien kiinteistöjen asukkaat. Tampuurin tietokannassa on jo yli 40 000 kiinteistöä ja kasvua tapahtuu joka vuosi. (Agenteq 2012.)

2.1 Tampuurin tekniikka

Tampuuri toimii selaimeen toteutetun käyttöliittymän kautta. Käyttöliittymä on toteutettu Microsoftin ASP.NET-pohjalle HTML:ää sekä CSS:ää käyttäen ja siihen on lisätty toimintoja Javascriptillä sekä sen Jquery-laajennuksella (Kuva 2). Lisäksi joissakin moduuleissa on käytetty Telerikin toimittamia valmiita käyttöliittymäkomponentteja. Tällä hetkellä Tampuuri tukee virallisesti ainoastaan Microsoftin Internet Explo-

rer -selainta joka asiakkaitten pääasiallisesti käyttämä selain. Tukemalla vain yhtä selainta virallisesti on päästy eroon suuremmista yhteensopivuusongelmista joita useiden selainten tukeminen aiheuttaa. Tampuuria testataan kuitenkin jatkuvasti myös muilla selaimilla jotta selain- ja versioriippuvuus ei kasva liian suureksi. (Agenteq 2012.)



Kuva 2. Tampuurin kohderekisteri

Tampuurin taustakoodi on toteutettu .NET-ympäristöön C#- ja VisualBasic-kielillä. Suurin osa Tampuurin yleisistä toiminnoista on Agenteqissä sisäisesti tehdyissä kirjastoissa tai Microsoftin .NET-ympäristön mukana tulevissa kirjastoissa. Joitakin ulkopuolisten toimittajien kirjastoja, kuten Aspose.Words ja Telerik, käytetään erikoistapauksissa muutamassa moduulissa. Moduuleissa on lisäksi omat kirjastot johon on sijoitettu vaativimmat ja käytetyimmät moduulin toiminnot.

Tietokantatoiminnot toteutuvat Tampuurin vanhoissa osissa Sql-proseduurien kautta, mutta uudemmat osat on toteutettu käyttäen ORM-rajapintaa, kuten NHibernate tai Entity Framework, tai käyttäen ADO.NET-rajapintaa.

Tietokanta on tehty Microsoftin SQL Server 2008 r2 -palvelimille jotka käyttävät Microsoftin SQL-tietokantaa.

2.2 Tampuurin rakenne

Tampuuri rakentuu 35 eri moduulista joista jokainen asiakas muodostaa haluamansa kokonaisuuden. Tampuuri on jakautunut osiin jotka ovat käytössä toimijan tarpeiden mukaan, ja vain muutamalla suurella toimijalla on kaikki osat käytössä. (Tampuuri 2012.)

Jos toiminnot haluaa jakaa selkeämpiin kokonaisuuksiin, niin Tampuurin osia perus toimintojen lisäksi ovat asukashallinta, kunnossapito, huolto, isännöinti, kulutusseuranta ja budjetointi.

Asukashallinta toimii kahdella tasolla. Asukashallinnasta isännöitsijä hallinnoi asiakkaiden tietoja ja asiakkaat ovat suoraan yhteydessä isännöitsijään tai huoltoyhtiöön esim. vikatilanteissa. Asiakkaat voivat myös hallinnoida mm. saunavuoroja.

Kunnossapito on yleisimmin kiinteistön isännöitsijän käyttämä työkalu josta mm. hankinnat ja remontit hallinnoidaan tarjouspyyntöineen ja kustannuslaskelmineen.

Huolto-osaa käyttävät kiinteistön huollosta vastaava toimija. Siitä hallinnoidaan huoltoja aina ilmoituksesta työn suorittamiseen ja raportointiin. Jos asukashallinta on käytössä, voivat asukkaat lähettää huoltoilmoitukset suoraan huoltoyhtiölle jonka Tampuurissa ne näkyvät kuvauksineen välittömästi. Tämän jälkeen työt määräytyvä huoltomiehille jotka kuittaavat omasta Tampuuri-profiilistaan työt ja suoritettuaan tehtävän tila päivittyy Tampuuriin ja ilmoituksen tehneelle asiakkaalle.

Isännöinti-osa käsittää lähinnä kaikki kiinteistöön liittyvät tiedot ja rekisterit. Täältä isännöitsijä hallinnoi tietoja ja saa tarpeelliset raportit automaattisesti. Koska isännöintiin liittyy paljon lakisääteisiä asioita, on osassa myös isännöintiin liittyvien lomakkeiden ja asiakirjojen tekemistoimintoja jolla toimintaa voidaan tehostaa.
Kulutusseuranta antaa käyttöön mm. sähkön ja vedenkulutuksen tiedot jos ne ovat kytkettävissä Tampuuriin. Nämä toiminnot ovat yleensä käytössä vain suurilla toimijoilla jotka haluavat tarkkaa ja ajantasaista tietoa.

Tampuuriin ei sisälly varsinaista taloushallinnan ohjelmistoa, vaan sitä on kehitetty yhteistyössä muiden yhtiöiden kanssa. Tarvittaessa Tampuurin saa kytkettyä taloushallinnon ohjelmistoihin. Budjetointiin on moduuli ja myös muita talouslaskennan työkaluja on saatavilla.

Tämä on kuitenkin vain hyvin karkea jako Tampuurin toiminnoista. Koska ohjelmistoa on kehitetty kauan, on siinä suuri määrä toimintoja ja monet niistä kuuluvat useampaan osaan. Myös monet moduulit ovat riippuvaisia keskenään, eli jotkin moduulit voi hankkia vain pakettina. Myös lisäämällä moduuleita saadaan lisätoimintoja, esimerkiksi ostamalla asiakaspalvelutoiminnot voivat asiakkaat tehdä ilmoituksia suoraan kunnossapitoon ja taloushallintaan omasta Tampuuristaan.

2.3 Tampuurin tietokanta

Tampuurin tietokanta on jaettu kahteen osaan. Master-kannassa ovat nk. yleiset Tampuurin vaatimat tiedot, kuten käyttäjät ja kirjautumistiedot. Toinen osa on varsinainen Tampuurin tietokanta jossa kaikki tuotantotieto on.

Tietokantaa on rakennettu ja laajennettu Tampuurin alkuajoista lähtien ja se sisältää paljon eri tavalla toteutettuja osia. Uusien Tampuurin versioiden yhteydessä sitä korjaillaan ja kehitetään eteenpäin, mutta tietokannan yhteensopivuuden takaaminen on vaikeaa koska jokaisen asiakkaan tietokanta on riippuvainen heidän käyttämästään versiosta, käytössä olevista moduuleista sekä heille räätälöidyistä lisäyksistä. Koska tietokantatoiminnot on aiemmin rakennettu proseduureja käyttäen, sisältävät pienetkin tietokannan muutokset suuren riskin.

Tampuuri on rakennettu dynaamiseksi ja se aiheuttaa lisähaasteita Tampuurin kehitystiimille. Dynaamisuus tarkoittaa sitä että asiakas saa melko pitkälle määritellä millaista tietoa tietokantaan tallennetaan. Tämän vuoksi asiakkaiden kannat sisältävät esim. asuntojen osalta erilaista tietoa. Tiedot ovat tietokannassa eri nimellä, eri muodossa tai joillakin asiakkailla on tietoa jota muilla ei ole. Dynaamisuus saadaan aikaan taululla joka antaa kaikille asiakkaan tallentamille tiedoille tunnuksen. Tähän tauluun merkitään mikä tieto on tallennettu päätauluun milläkin tunnuksella. Päätaulussa on vain tieto ja sen tunnus. Tällä tavalla voidaan luoda tietokantaan asiakkaan haluamia tietoja

Tampuuriin rakennettuja työkaluja käyttäen. Tampuuri vaatii toimiakseen samat perustiedot, mutta antaa asiakkaalle mahdollisuuden lisätä tietoja jotka he kokevat tärkeäksi.

3 .NET YMPÄRISTÖ

Microsoftin kehittämä .NET-ympäristö (Framework) jonka alkuperäisenä tarkoituksena ilmestyessään 2002 luoda voimakkaampi, joustavampi ja yksinkertaisempi ohjelmointimalli vanhan COM-ohjelmoinnin tilalle. (Troelsen 2012, 4.)

Suurina etuina .NET:llä on yhteensopivuus eri osien ja myös vanhan koodin kanssa. Tuki monelle ohjelmointikielelle kuten C#, F# ja Visual Basic. Vahva tiedon tyypitys jota kaikki .NET yhteensopivat kielet tukevat. Ohjelmointikielien integraatio, eli esimerkiksi C# kirjoitettu kirjasto toimii ja on virhekorjattavissa (debug) myös Visual Basicissa. Kaikki .NET-yhteensopivat ohjelmointikielet sisältävät erittäin kattavan pohjakirjaston josta saadaan valmiiksi toimivat mallit perustoimintojen ratkaisuiksi. Lisäksi .NET julkaisutoiminnot ovat yksinkertaistuneet ja antavat mahdollisuuden saman kirjaston eri versioiden käytölle yhtäaikaisesti. (Troelsen 2012, 4.)

.NET rakentuu kolmesta osasta CLR, CTS ja CLS. CLR eli Common Language Runtime pitää huolen mm. objektien paikannuksesta, latauksesta ja hallinnoinnista. CLR hoitaa myös alemman tason ohjelma toimintoja kuten muistin hallinnointi, sovellusten hostaus (application hosting), säikeiden (treads) koordinointi sekä ajoitettujen turvatarkastusten teko. (Troelsen 2012, 4.)

CTS eli Common Type System määrittelee kaikki käytössä olevat tietotyypit ja ohjelmointirakenteet. CTS määrittää myös, kuinka entiteetit voivat toimia keskenään ja kuinka ne merkitään metadataformaatissa. Huomioitavaa on että .NET-yhteensopivat ohjelmointikielet eivät välttämättä tue kaikkia näitä ominaisuuksia (Troelsen 2012, 4-5.)

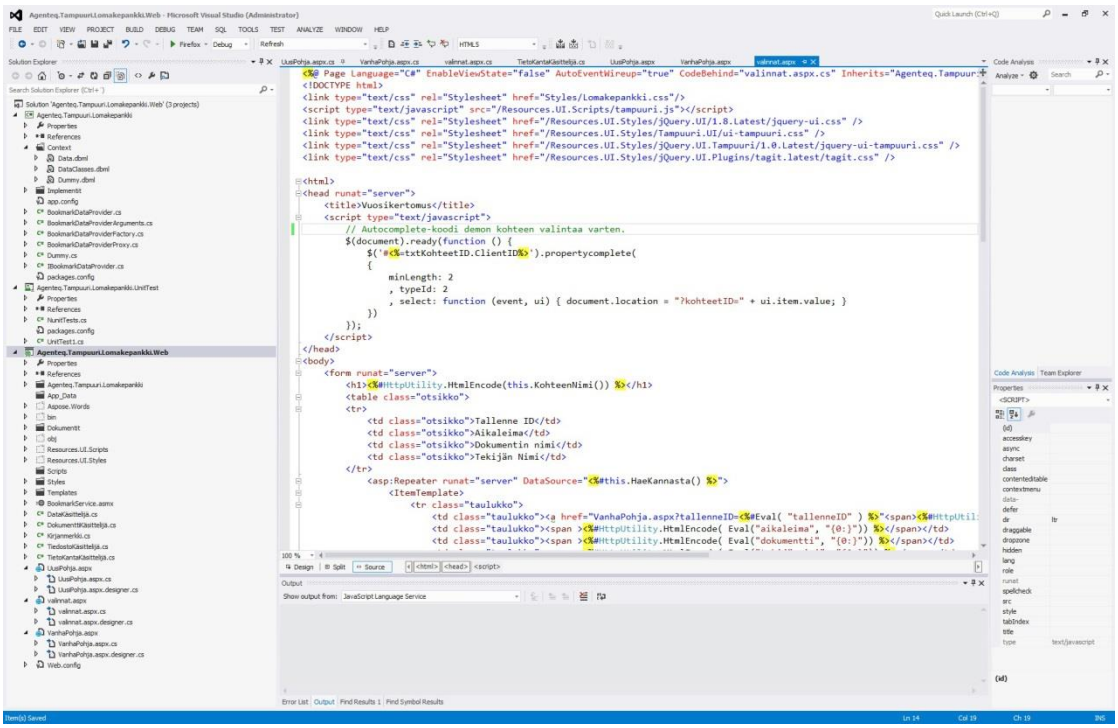
CLS eli Common Language Specification määrittelee kaikki ne tyypit ja rakenteet joita kaikki .NET-yhteensopivat ohjelmointikielet noudattavat. Jos käytetään tyyppejä tai rakenteita joita ei ole määritetty CLS:ssa, yhteensopivuutta jollain muulla .NET-

yhteensopivalla ohjelmointikielellä tehtyihin komponentteihin ei taata. (Troelsen 2012, 4-5.)

Lisäksi .NET pohjaluokkakirjasto (Base Class Library) joka tulee kaikkien .NET-yhteensopivien ohjelmointikielien mukana, tarjoaa huomattavan määrän valmiita luokkia joilla voidaan helposti ja luotettavasti hoitaa mm. ruudun renderoinnit (screen render), virtojen (stream) hallinta, I/O hallinta sekä ulkoisten laitteiden kuten tulostimien ja skannereiden kanssa kommunikoinnin. (Troelsen 2012, 5.)

3.1 Visual Studio

Visual studio on Microsoftin kehittämä IDE eli Integrated Developement Environment (Kuva 3). Ensimmäinen versio Visual Studiosta julkaistiin jo vuonna 1995, ja se luotiin helpottamaan ja vauhdittamaan uuden Windows 95 -käyttöjärjestelmän ohjelmien kehitystä. Vuosien varrella Visual Studiosta on kehittynyt markkinajohtaja ja uusin versio, Visual Studio 2012, tukee useita .NET-yhteensopivia alustoja perinteisten tietokoneiden lisäksi. Siinä on useita ohjelmointikieliä ja nykyään Visual Studioon on mahdollista integroida myös kolmannen osapuolen tekijöiden kehitystyökaluja. (Microsoft Inc 2013.)



Kuva 3. Visual Studio 2012

Visual Studiosta (Kuva 3) on kehittynyt monipuolinen web-kehitysalusta. Sen ASP.NET-kehitystyökalut mahdollistavat monipuolisten web-sivujen kehittämisen helposti. Visual Studio tarjoaa paljon valmiita rakennuspalikoita joita käyttämällä työn tuottavuutta saadaan parannettua helposti. Visual Studio tarjoaa myös mahdollisuuden luoda kaikki toiminnot ja työkalut itse, käyttämällä vaikka Javasriptiä tai sen Jquery-laajennetta. Visual Studio tukee myös kaikkia HTML- ja Css-standardeja ja sillä on helppo testata kehitettävää sivustoa vaikka usealla eri selaimella. (Microsoft Inc 2013.)

Voimallinen Visual Studion kehitystä auttava työkalu on IntelliSense. Se analysoi kirjoitettua koodia jatkuvasti ja pitää reaaliajassa huolen että ohjelmoija ei tee syntaksi-, tyyppi- tai kutsuvirheitä. IntelliSense tarjoaa automaattisesti kaikki käytössä olevat tyypit ja luokat ohjelmoijan käytettäväksi ja täydentää ne automaattisesti silloin kun se mahdollista. IntelliSense ei kuitenkaan voi korjata sisällön tai rakenteen virheitä, ne ovat täysin ohjelmoijan vastuulla. (Microsoft Inc 2013.)

3.2 ASP.NET

ASP.NET on Microsoftin kehittämä web-sovellusympäristö jolla sivustojen kehittäminen on nopeaa ja helppoa. Asp eli Active Server Pages, sisältää sekoituksen merkkaustapoja ja kieliä. Tietoa voi käsitellä palvelimella ennen kuin se lähetetään käyttöliittymään tai toisinpäin, näin saadaan helposti aikaan dynaamisia verkkosivuja. Palvelinkoodia voidaan kirjoittaa suoraan HTML-sivulle <% ja %> tagien sisälle. (Evjen ym. 2009, xxvii.)

ASP.NET:n kehitysalusta Visual Studio antaa käyttäjälle mahdollisuuden tehdä verkkosivut suunnittelumoodissa jossa kaikki objektit voidaan sijoittaa sivulle halutulla tavalla vetämällä ja pudottamalla (drag and drop) ne työtilaan. Visual Studio generoi automaattisesti tarpeelliset koodit HTML-sivuun. (Evjen ym. 2009, xxviii.)

ASP.NET-ympäristön etuja ovat mm. tehokas välimuistitus (caching), helppo HTML-sivun tilanhallinta, optimointi Microsoftin SQL-palvelimiin, asp-sovellusten kunnon ja tehokkuuden tarkkailu, tehokas ja monipuolinen kehitysalusta, dynaaminen data

käyttäen entiteetti-datamallia (entity data model), data-palvelut (data services) ja lokalisaation hallinta .resx-tiedostoasetuksilla. (Evjen ym. 2009, xli-xlix.)

3.3 C#

C# on Microsoftin kehittämä ohjelmointikieli joka pohjaa C-kieleen ja Javaan, mutta on kehitetty ennen kaikkea .NET-ympäristön ohjelmistokehitystä varten. C# on vahvasti tyypitetty ja ilmaisuvoimainen kieli johon on sisällytetty mahdollisuus käyttää mm. lambda-lauseita. Vahva tyypitys tarkoittaa sitä, että muuttujat ja tyypit pitää määrittää tarkasti, jolloin esim. duplikaattimuuttujia tai tyyppimuunnosvirheitä ei pääse tapahtumaan. (Microsoft Inc 2013.)

C# tukee luokka-ominaisuuksia eikä näin vaadi perinteisiä get- ja set-metodeja. C#:lla on myös mahdollista ylikuormittaa (overload) operaattoreita ja luoda rakenteita (structures), enumeraatioita (enumerations) ja CallBack-funktioita. C# sisältää automaattisen roskien keräilyn (garbage collector) eli lukuun ottamatta disposable-tyyppisiä luokkia ohjelmoijan ei tarvitse huolehtia luokkien tuhoamisista tai pyyhkimistä. (Troelsen 2012, 6.)

C#:ssa ei ole tarpeen käyttää osoittimia (pointers), mutta niiden käyttö on kuitenkin mahdollista, jos ohjelmoija niin välttämättä haluaa. Myös attribuuttipohjainen ohjelmointi onnistuu C#:ssa. Esimerkiksi merkkaamalla metodin tagilla ”[obsolete]”, saa käyttäjä ilmoituksen että hänen käyttämänsä osa on merkattu vanhentuneeksi (obsolete). (Troelsen 2012, 6.)

C# antaa mahdollisuuden luoda geneerisiä tyyppejä ja jäseniä (members). Tämä antaa mahdollisuuden luoda helposti tyyppiturvallista ja tehokasta koodia. On myös mahdollista luoda yksittäisiä tyyppejä useaan kooditiedostoon käyttämällä partial-avainsanaa. (Troelsen 2012, 6.)

Lisäksi hyödyllisiä ominaisuuksia ovat anonyymit tyypit, funktioiden laajentaminen laajennus-metodeilla (extension method), vahvasti tyypitetty tiedon haku Linq:lla, lambda-lauseet, uusi syntaksi joka mahdollistaa ominaisuuksien asettamisen objektin käynnistyksessä (intialize), metodien parametrit ja nimetyt argumentit, dynaamisen

jäsenten haun dynamic-avainsanalla ajon aikana ja helpompi työskentely geneerisillä tyypeillä. (Troelsen 2012, 6-7.)

Koska C#:lla voi kehittää vain .NET-ympäristöön, on sillä tehty koodi aina hallinnoitua. Kaikki osat ajetaan .NET runtime-moottorin kautta, joka pitää huolen käyttöjärjestelmän resursseista, vähentäen näin virhekoodin mahdollisuutta ja helpottaen ohjelmoijan työtä. (Troelsen 2012, 7.)

3.4 Linq

Linq eli Language-integrated query on Microsoftin .NET-ympäristöön kehittämä työkalu jolla saadaan tehtyä tehokkaasti hakuja suoraan koodista xml-tiedostoihin, taulukoihin, tietokantoihin tai muihin dataa sisältäviin tiedonlähteisiin. Haut voidaan tehdä käyttäen nk. Lambda-lauseita jotka tekevät tiedonhausta helpompaa ja yksinkertaisempaa. (Microsoft Inc 2013; Jennings 2009, 5-7.)

LinqToSQL on helppokäyttöinen ORM-työkalu. Linq:lla voidaan automaattisesti generoida tietokantaa vastaava malli joka kartoittaa (mapping) kaikki tietokannan taulut ja entiteetit. Tämä malli luo rajapinnan joka antaa kaikki tietokantamalliin määritellyt entiteetit suoraan luokkarakenteeksi ohjelmoijan käyttöön. Tämän tietokantamalliin voi myös lisätä ohjelmointilogiikkaa, riippuvuuksia tai rajoitteita (constraint). (Jennings 2009, 15-20.)

3.5 Aspose.Words -kirjasto

Aspose.Words on Aspose Corporationin .NET-ympäristöön kehittämä kirjasto jolla voidaan C#:ssa tai Visual Basic:ssä lukea ja kirjoittaa Word-dokumentteihin. Aspose.Words tukee myös kaikkia muita yleisiä dokumenttiformaatteja sekä myös kuvaformaatteja. Kirjasto mahdollistaa myös dokumenttien yhdistämiset, formaattien muutokset ja muotoilujen asettamiset. Aspose.Wordsissa on myös postitustyökaluja esim. massapostitusta varten. (Aspose corporation 2013.)

4 SQL-TIETOKANNAT

SQL eli Structured Query Language on relaatio-tietokannoissa käytettävä komentokieli-standardi. SQL-kieli käsittelee tietojoukkoja ryhmissä, eli relaatioilla ja esim. niiden eheysriippumattomuudella on suuri merkitys. Standardeja on olemassa kaksi, ANSI ja ISO, mutta yksikään valmistaja ei tue täysin standardeja johtuen niiden epämääräisyydestä ja tarpeesta tehdä optimointeja tietokantapalvelimiin. Jokainen palvelinohjelmisto tukee kuitenkin kaikkia perustoimintoja. (Stephens ym. 1999, 3-4; Vieira 2009, 1.)

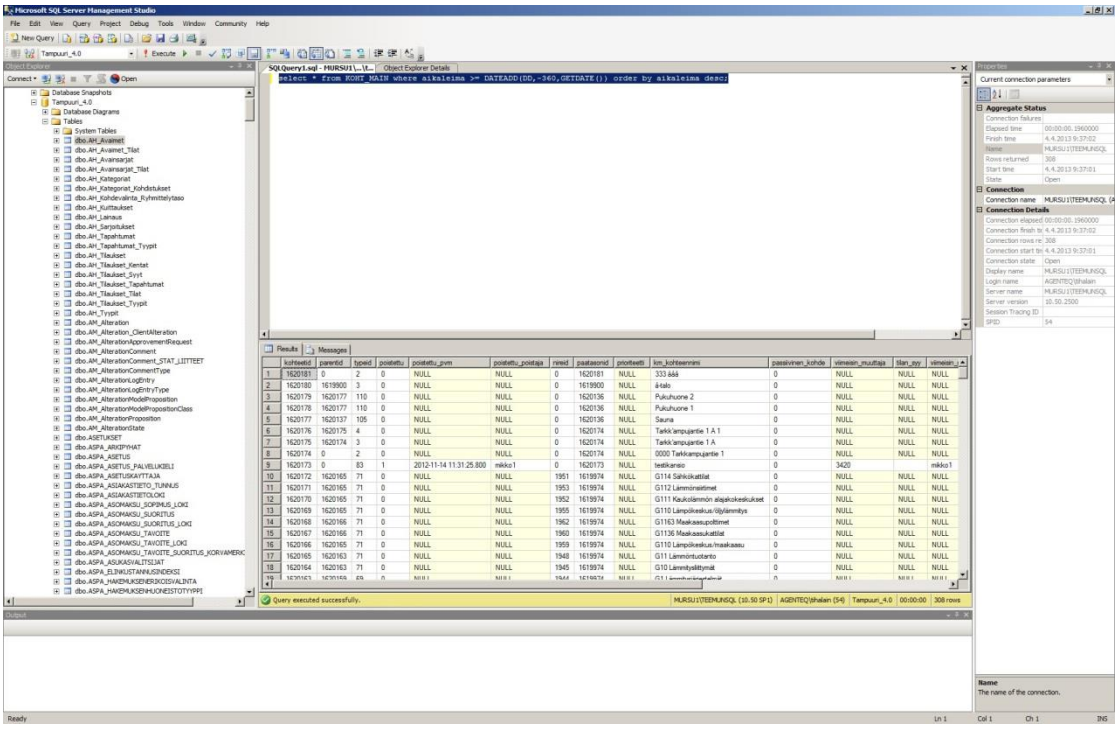
4.1 MS SQL SERVER 2008 R2

Microsoftin vuonna 2008 julkaisema SQL-palvelin ohjelmisto jota on päivitetty R2 versioon vuonna 2010. Koska uusin versio on tullut vasta vuonna 2012, on 2008 R2 kaikkein yleisin käytössä oleva MS SQL-palvelin.

MS SQL-server 2008 toi mukanaan muutamia erittäin hyödyllisiä ominaisuuksia kuten uudet tietotyypit päivämäärälle ja ajalle sekä tuen geospatiaaliselle tiedolle ja hierarkiselle tiedon esittämiselle. Uusi merge-komento jolla insert-, update- ja delete-komennot voitiin yhdistää yhteen komentoon. Lisäksi tiedon hallintaa ja optimointia on automatisoitu. Myös raportointiominaisuuksia on parannettu huomattavasti, ja R2 version myötä raportit saa tehtyä helposti myös sharepoint- tai excel-pohjiin. (Microsoft Inc 2013; Vieira 2009, xxxv-2.)

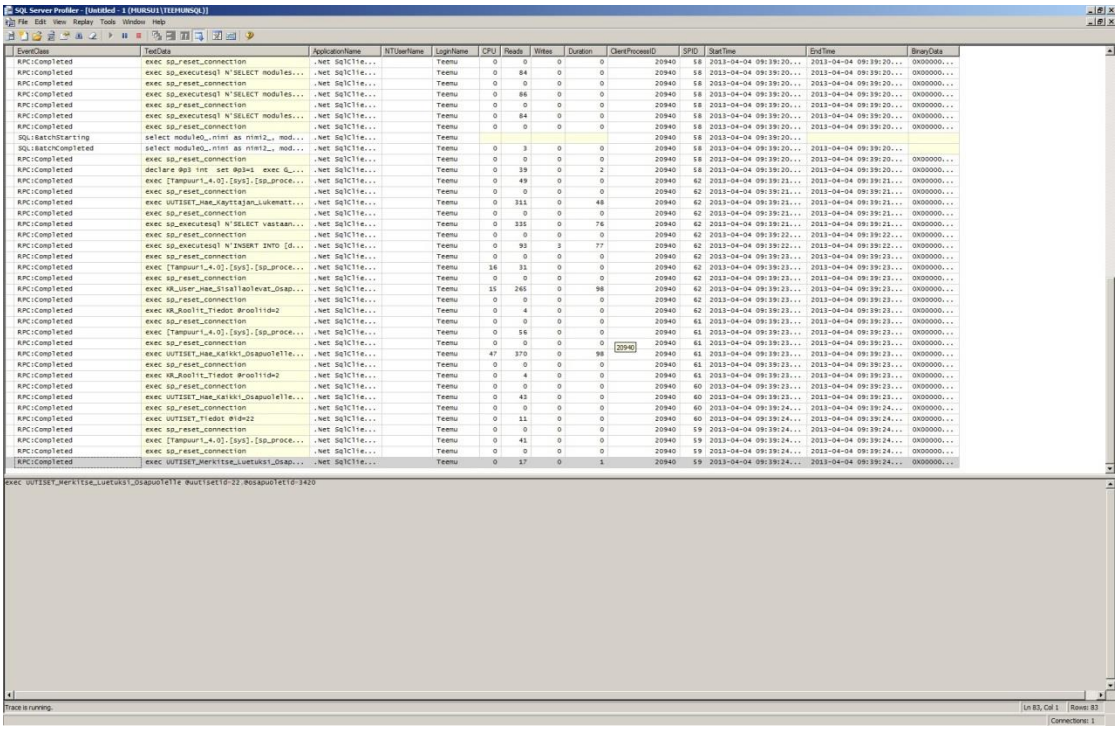
4.2 HYÖDYLLISET TYÖKALUT

MS SQL serveriä hallitaan Server Manager -ohjelmalla (Kuva 4). Ohjelmassa voidaan luoda skeemoja ja niihin taulut, proseduurit, laukaisimet (triggers) ja niin edelleen. Työkalu vaatii kuitenkin osaamista sillä sen käyttö tapahtuu pääosin Query Editorin kautta, ja se ottaa komennot vastaan vain SQL-muodossa.



Kuva 4. Server Manager jossa on avattuna Tampuurin tietokanta

Query Editor kuitenkin auttaa komentojen rakentamisessa, ja ilmoittaa heti jos jotain puuttuu, tai esim. taulua ei ole olemassa tai sen nimessä on virhe. Jos asetuksista on laitettu aputoiminnot päälle, tarjoaa Manager mm. proseduureja ja tauluja suoraan komentoa kirjoitettaessa. Tehokkain käyttötapa on kutenkin rakentaa toiminnoista scriptejä, joita ajamalla saadaan kaikki toiminnot suoritettua yhdellä ajolla, ja käytettäessä transaktiota (transaction) voidaan myös varmistaa ettei virheellinen scripti päivitä palvelinta. Kaikkia objekteja voidaan tarkkailla myös graafisesti, ja kaikkien objektien ominaisuuksia voidaan säätää valitsemalla ne oikealla napilla. (Vieira 2009, 26-32.)



Kuva 5. Server Profiler johon kaapattuna palvelin liikenne Tampuurin käynnistyessä

SQL Server Profiler (Kuva 5) on hyödyllinen työkalu jolla voidaan seurata palvelimella käytännössä tapahtuvia toimintoja. Seuranta kannattaa kuitenkin säätää niin että vain tarpeellinen tieto tulee Profileriin, koska sillä saa kaikki tiedot esim. kirjautumisista ja palvelimen ajastetuista toiminnoista, ja liika tieto tekee analysoinnista vaikeaa. Heikkoutena on myös tuotantopalvelimilla suuri käyttäjämäärä, jo muutamassa sekunnissa kirjautuu Profileriin tuhansia toimintoja ja niiden analysointi on miltei mahdotonta. Oikein säädetystä Profilerista saa kuitenkin helposti tiedot siitä mitä komentoja palvelimella ajetaan, ja mitä niistä saadaan vastaukseksi. Tällä saadaan selville virheet esim. proseduurien toiminnasta sillä niiden toiminnassa olevat virheet eivät välity ohjelmaan asti. Myös kyselyn vastaukseksi saatavan tiedon analysointi paljastaa mahdolliset virheet kyselyssä tai tietokannassa, tai niiden puute virheet ohjelmassa. (Vieira 2009, 34.)

5 TALOYHTIÖN VUOSIKERTOMUS -TYÖKALU

Tässä luvussa kerron projektin alussa työkalun tekemisessä huomioon otettavista asioista sekä siihen liittyvistä vaatimuksista, ominaisuuksista sekä asioista jotka Tampuuriin liittämisessä pitää ottaa huomioon.

5.1 Vuosikertomuksen tekemisestä

Taloyhtiöiltä vaaditaan vuosittainen toimintakertomus jossa kerrotaan viimeisen vuoden tapahtumat ja muutokset. Vuosikertomukseen kuuluvat myös tase- ja tuloslaskelma joiden tekeminen vuosittain on erittäin työlästä.

Suurin osa tiedoista on samaa vuodesta toiseen, joten kertomuksen tekeminen uudestaan joka vuosi on turhaa varsinkin kun kaikki ajankohtainen tieto olisi saatavilla suoraan Tampuurista. Aiemmin tiedot on luettu tai tulostettu Tampuurista ja kirjoitettu käsin Word-dokumentiksi useimmiten edellisvuoden dokumentin päälle.

Uuden työkalun tarkoituksena on automatisoida vuosikertomuksen tekeminen niiden tietojen osalta jotka ovat saatavilla Tampuurista. Tehdään vain pohja johon tietojen paikalle sijoitetaan kirjanmerkit ja ladataan se Tampuuriin. Tampuuri tulee tukemaan määrättyjä kirjanmerkkejä joiden tiedot haetaan ja täytetään automaattisesti järjestelmästä. Myös kirjanmerkit joissa tukea ei ole tallennetaan järjestelmään ja voidaan palauttaa automaattisesti seuraavaa vuosikertomusta tehdessä.

Tämä vähentää merkittävästi vuosikertomuksen tekemisen työtä, ja pitää huolen siitä että tiedot ovat ajan tasalla. Varsinainen työ onkin tietojen oikeellisuuden tarkastamisessa, sillä myös järjestelmässä voi olla inhimillisiä virheitä tai vanhentunutta tietoa.

5.2 Vaatimukset

Vuosikertomus-työkalun käyttö vaatii Tampuuriin kirjautumisen, jo pelkästään tietoturvasyistä, mutta myös siksi koska työkalu tarvitsee Tampuurin pohjapalveluja toimiakseen. Työkalu tarvitsee pohjapalveluilta käyttäjätiedot sekä yhteysmerkkijonon.

Jotta työkalu toimii pitää kohteen, eli tässä tapauksessa taloyhtiön, olla valittuna jotta tiedot kohdistuvat tietokantahauissa ja -tallennuksissa oikeisiin paikkoihin. Tampuuriin työkalu sijoitetaan niin että kohde on jo valmiiksi valittuna ja sen sivulta on linkki työkaluun joka saa kohdetunnuksen linkin mukana.

Seuraavaksi ladataan käytettävä dokumenttipohja. Tampuuriin pohja sijoitetaan dokumenttipankkiin josta se voidaan valita työkalun käyttöön. On myös mahdollista että suoralataus-toiminto otetaan käyttöön, jolloin pohja ei tallennu dokumenttipankkiin.

Kun pohja-dokumentti on valittu, työkalu lukee siitä kirjanmerkit. Jos kirjanmerkki on tuettu, työkalu pyytää tiedontoimittajalta kyseisen tiedon. Jos kirjanmerkkiä ei tueta työkalu antaa tyhjän kentän johon voi halutun tiedon kirjoittaa käsin ja myös se tallentuu järjestelmään.

Myös vanhoja dokumentteja voi ladata ja muokata tarvittaessa. Vanhoja dokumentteja voi myös käyttää pohjana uusille, jolloin vain tuettujen kirjanmerkkien tiedot tulee päivittää järjestelmästä. Käyttäjä voi muokata vanhan dokumentin tukemattomia kirjanmerkkejä ja lopuksi käyttäjä voi tallentaa sen uudeksi dokumentiksi järjestelmään.

Muokkaussivuille tulee myös esikatselutoiminto jolla valmista dokumenttia voidaan tarkastella käyttäjän koneen Word-ohjelmassa ilman että tiedot tallentuvat järjestelmään.

5.3 Valmiit osat

Tampuurissa on kirjasto Agenteq.Tampuuri.Core.Utility josta ladataan userhelper-työkalu jolla käyttäjätiedot ja yhteysmerkkijono saadaan järjestelmästä.

Tampuurissa on käytössä Aspose.Words joka on kolmannen osapuolen toimittama sovellus Word-dokumenttien lukemiseen ja muokkaamiseen.

Muut toiminnot tehdään käyttäen .NET-ympäristön tarjoamia työkaluja.

5.4 Laajennukset tulevaisuudessa

Projekti on prototyyppi jota aiotaan käyttää pohjana muille vastaaville palveluille tulevaisuudessa. Tällöin tiedontoimittaja-kirjastoon lisätään tarpeelliset tiedontoimittajat ja tehdään tarkoitusta varten oma käyttöliittymä joka hyödyntää projektissa luotua

mallia. Suurin osa dokumenteissä tarvittavasta tiedosta on samaa joten samat tiedontoimittajat voivat olla käytössä monessa eri dokumentteihin liittyvässä palvelussa.

Kun työkalu otetaan käyttöön jossain Tampuurin versiossa, on tarkoituksena tukea useaa kymmentä kirjanmerkkiä. Tämän jälkeen tukea lisätään asiakkaiden tarpeiden mukaan, tarkoituksena on lopulta tukea kaikkea tietoa jota Tampuurista voidaan haluta vuosikertomukseen. Tämä tarkoittaa uusien tiedontoimittajien kirjoittamista, siksi rakenteen tulee käyttää rajapintaa jotta lisäyksiä ja muutoksi voidaan tehdä ilman yhteensopivuusongelmia.

6 TYÖKALUN TOTEUTTAMINEN

Tämän luvun aluksi kerron yleisesti projektin etenemisestä vaiheittain. Kuvailen kehitystyön omassa osiossaan koska varsinainen kehitystyö tapahtui pienissä palasissa ja tein jatkuvasti muutoksia kaikkiin osiin. Ohjelmistokehitystyön tekeminen lineaarisesti on erittäin vaikeaa koska kaikki osat ja niiden muutokset vaikuttavat kokonaisuuteen.

Tämän jälkeen kerron toteutuksesta lopullisessa muodossa yksityiskohtaisesti sekä vastaan tulleista ongelmista ja tärkeistä huomioista joita tein työn edetessä.

6.1 Kehitystyö

Työkalun toteuttaminen alkoi tiedonhaun rakenteiden luomisella ja käyttöliittymän raakaversion tekemisellä. Ensimmäisessä toimintojen testaamisessa haettiin vain koodiin kovakoodattua tietoa valmiisiin kenttiin jolloin nähtiin että rakenne toimi ja tiedot sijoittuivat kenttiin oikein.

Tämän jälkeen työ eteni pienten parannusten ja lisäysten tekemisellä vaiheittain samalla testaten parhaimpia toimintatapoja ja varmistaen että toiminnot eivät rikkoonnu missään tilanteessa.

Koska suurin osa toiminnoista tarvitsee dokumenttien käsittelyä, tein seuraavaksi toiminnot joilla dokumenttia pystyy lukemaan sekä kirjoittamaan. Kun toiminnot olivat

valmiit, testasin niitä tekemälläni dokumenttipohjalla jonka kirjanmerkkeihin tallensin kovakoodattua tietoa sekä hain kirjanmerkkejä.

Kun dokumenttien kirjanmerkit olivat saatavilla, lisäsin tarvittavat taulut tietokantaan sekä testitietoja, jotta pystyisin testaamaan toimintoja. Tämän jälkeen tein tietokanta-toiminnot aloittaen hakutoiminnoista, joilla pohjatiedot sekä vanhojen tallennusten tiedot saa näkyviin. Seuraavaksi tein lisäys-, poisto- sekä päivitystoiminnot joilla kirjanmerkkejä sekä niiden sisältöä pystyi muokkaamaan tietokannassa.

Tässä vaiheessa ensimmäinen demoversio oli valmis ja siinä oli kaikki perustoiminnot karkeassa muodossa. Hioin ja testasin koodin toimintaa sekä parantelin rakennetta seuraavaa vaihetta varten.

Seuraavassa vaiheessa muokkasin projektin rakenteen lopulliseen muotoon. Hajautin toiminnot kolmelle eri aspx-sivulle ja siirsin yleiskäyttöiset toiminnot kuten dokumenttien käsittelyn sekä tietokantojen muokkaamiset, omiin luokkiinsa.

Eniten aikaa meni koodin siistimiseen ja parantamiseen. Kirjoitin koodia ainakin neljän ohjelman verran ennen kuin paras ja lisäksi helppolukuinen muoto löytyi. Toimintoja ei tullut lisää mutta miettiessäni käyttötapauksia testauksen yhteydessä muutin toimintoja useaan kertaan jotta mm. mahdolliset virhetilanteet ja käyttäjän tarpeet tuli otettua huomioon.

Tämän jälkeen Web-projekti oli toiminnoiltaan valmis, ja kehitys siirtyi Tampuuriin liittämiseen vaadittaviin toimintoihin. Tein uuden kirjasto-projektin johon kaikki tulevat tiedon toimittajat rakennetaan. Koska kirjastot ovat yleisessä käytössä ja niiden virheettömyys on paljon tärkeämpää kuin tavanomaisen koodin, joten tein kirjaston käyttäen Tdd:tä. Tein myös kirjastoon demoa varten valmiiksi tiedontoimittajia jotka hakevat tarvittavat perustiedot taloyhtiöstä. Eri moduulien kehitystiimit kirjoittavat myöhemmin kyseiset tiedontoimittajat kirjastoon.

Seuraavaksi muokkasin Web-projektia niin että se osaa kutsua kirjaston tiedontoimittajia. Lisäksi parantelin toimintoja niin että Tampuurista saatavat tiedot voi päivittää

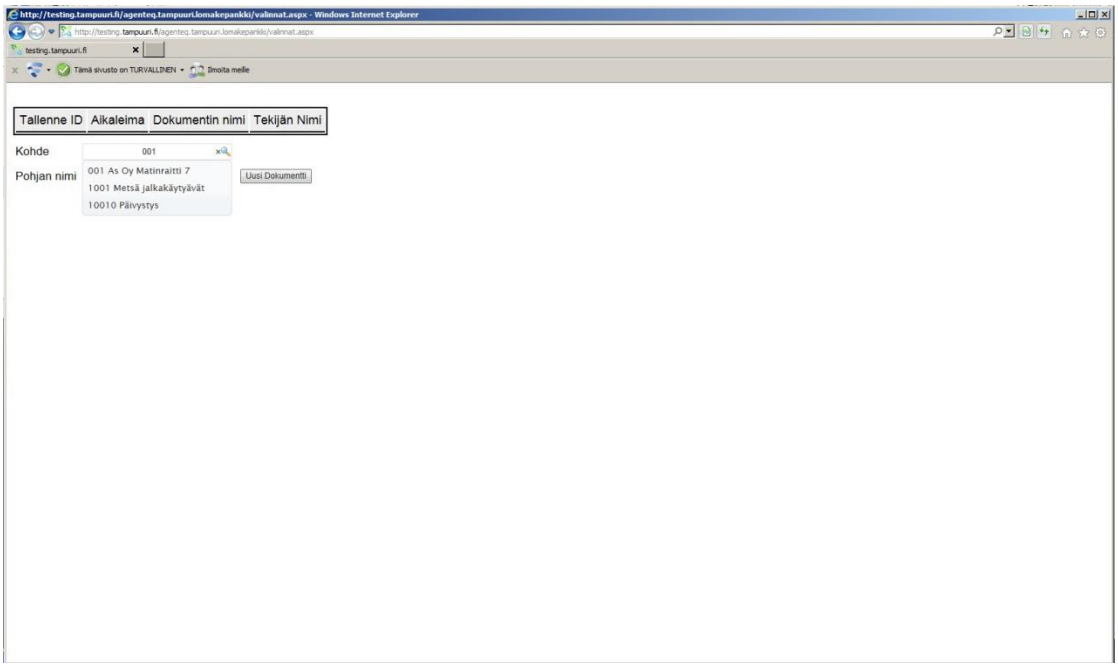
myös aiemmin tallennetusta dokumentista tehtävään uuteen dokumenttiin, jos kyseisen kirjanmerkin tietoa saadaan Tampuurin tiedontoimittajilta.

Tässä vaiheessa työkalu oli periaatteessa valmis. Oli kuitenkin vielä tarpeellista tehdä siitä demo jotta johtoryhmä tuotepäällikön kanssa näkee työkalun toiminnot ja voivat tehdä päätöksen työkalun lopullisesta integroinnista tulevaisuuden Tampuuri-versioon. Demoa varten lisäsin Tampuurin toiminnot joilla sain käyttäjätiedot, sekä käyttöliittymään kentän jolla voi hakea Tampuurin tietokannassa olevat kohteet automaattisesti täydentyvällä kentällä. Lisäksi muokkasin ja testasin koodia niin, että se toimii missä tahansa Tampuurin asennuksessa. Asensin projektin testipalvelimelle jolla Tampuurin uusia osia voidaan testata ja näyttää.

Tämän jälkeen projekti oli minun osaltani valmis. Kehitystyö siirtyi odottamaan päätöstä lopullisesta toteutuksesta.

6.2 Käyttöliittymä

Käyttöliittymästä tein yksinkertaisen, ja se toimii mallina lopulliselle toimivalle rakenteelle. Yksinkertainen käyttöliittymä on riittävä kaikkien toimintojen testaukseen, ja sen toimintojen integrointiin Tampuuriin. Graafikko suunnittelee myöhemmin käyttöliittymän, jolloin myös toiminnot tehdään lopulliseen muotoonsa ja laajuuteensa.

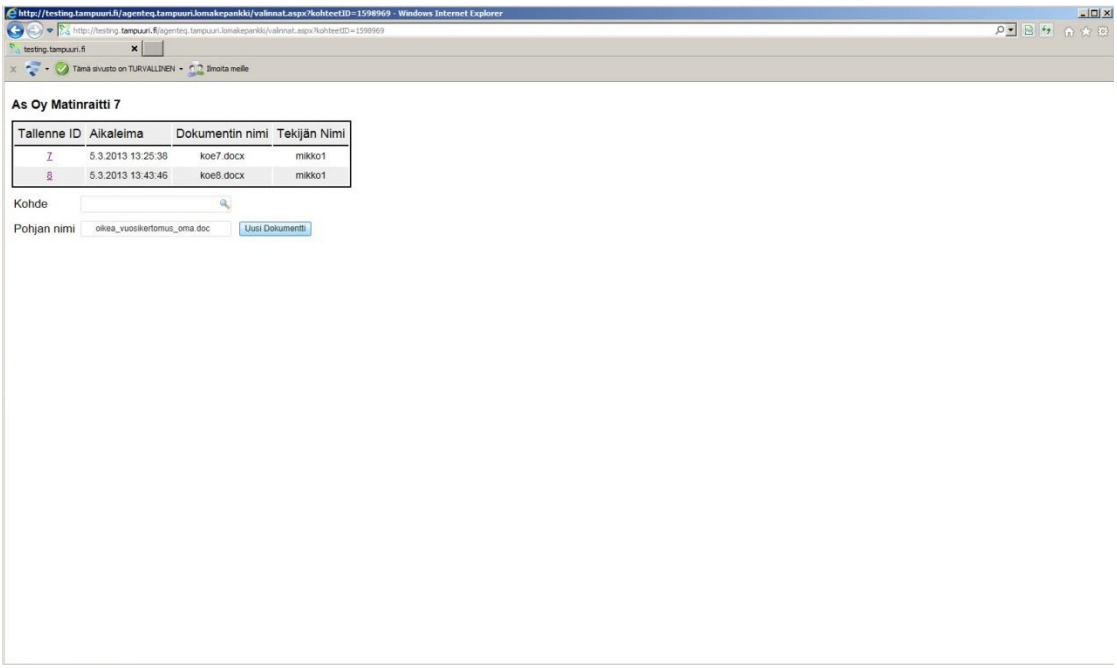


Kuva 6. Vuosikertomus-työkalun avaussivu Valinnat.aspx

Työkalun käyttö alkaa kun aloitussivu valinnat.aspx ladataan tampuuriin kirjautumisen jälkeen (Kuva 6). Ylimpänä on taulukko johon valitun kohteen tiedot latautuvat, jos aiempia tallennuksia on tietokannassa.

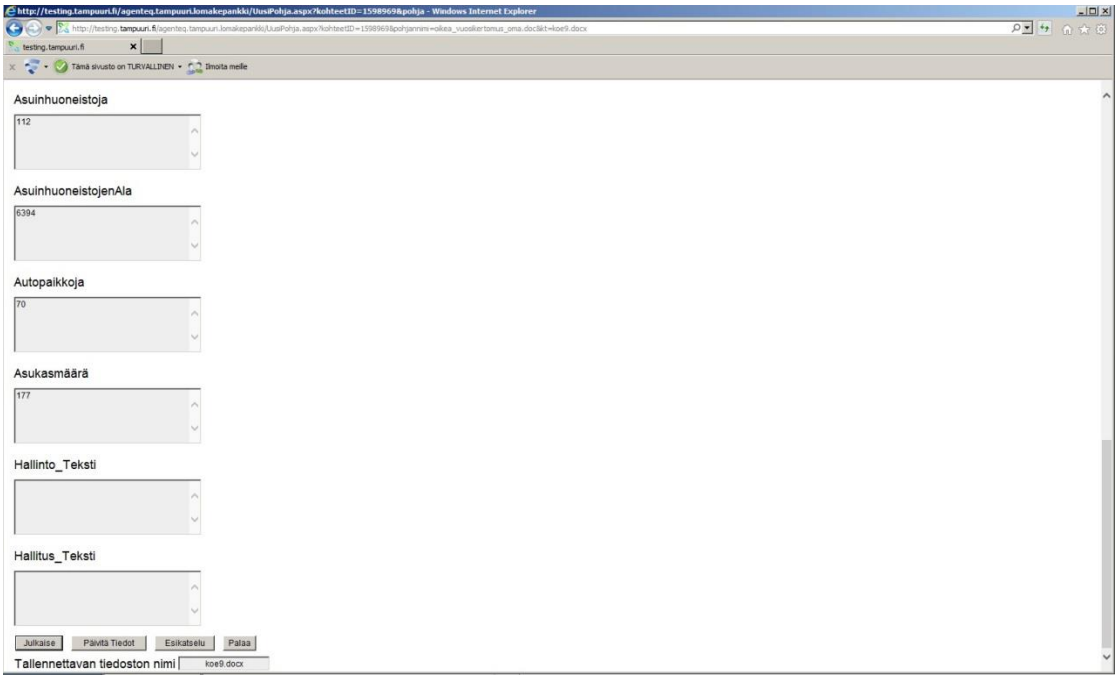
Tämän alle on sijoitettu Javascriptillä tehty automaattisesti täydentyvä kenttä johon kirjoitetaan kohteen nimi.

Alimpana on kenttä, johon kirjoitetaan demossa käytettävän pohjan nimi, jos dokumentti halutaan tehdä uutta pohjaa käyttäen. Tein testausta varten aidosta taloyhtiön vuosikertomuksesta version johon sijoitin kirjanmerkit oikeille paikoille. Kyseisen tiedoston nimi on demossa oletuksena tekstikentässä. Kenttään voi myös kirjoittaa uuden pohjan käsin, jos dokumentti on kopioitu oikeaan kansioon.



Kuva 7. Avaussivu jossa kohde valittuna

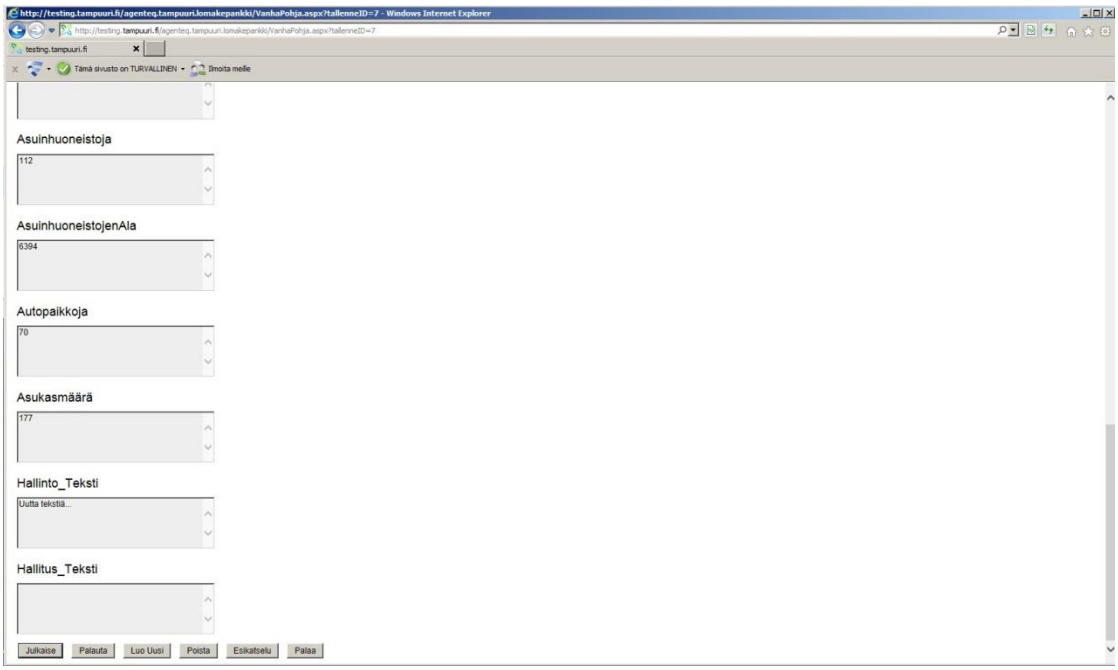
Kun kohde on valittu, näkyy ylimpänä taloyhtiön nimi ja taulukkoon sen alle ilmestyy aiemmin tallennetut dokumentit tallennusaikoineen, tiedoston nimineen sekä tallenteen tehneen käyttäjän käyttäjätunnus (Kuva7). Valitsemalla tallenteen tunnuksen voidaan tallenne avata vanhapohja.aspx -sivulle jossa sen tietoja voidaan muokata, luoda sitä pohjana käyttäen uusi dokumentti, tai poistaa se kokonaan. Valitsemalla uusi dokumentti -painike voidaan valittua pohjaa käyttäen luoda uusi dokumentti uusipohja.aspx -sivulla.



Kuva 8. Uusi pohja -sivu jonka tuettuihin kirjanmerkkeihin tieto on latautunut

Jos valitaan dokumentin tekeminen uutta pohjaa käyttäen, sivu latautuu valmiiksi Tampuurista tekstikenttiin haetuilla tiedoilla (Kuva 8). Jokaisen tekstikentän yllä on kyseessä olevan kirjanmerkin nimi. Sivun alalaidassa on tallennettavan tiedoston nimi, demoa varten tein iteroituvan oletusnimen.

Demossa ”Hallinto_Teksti” sekä ”Hallitus_Teksti” -kirjanmerkit eivät ole tuettuja kirjanmerkkejä ja siksi ne eivät saa valmiita tietoja Tampuurista. Esitäytettyjä tietoja voi tekstikentissä muokata ja tyhjiin tekstikenttiin tulee kirjoittaa tiedot jotka puuttuvat. Päivitä tiedot -painikkeella haetaan järjestelmän tiedot uudestaan kenttiin jos niitä on. Esikatselu-painikkeella lähetetään dokumentti käyttäjän koneelle katseltavaksi, mutta mitään ei tallenneta järjestelmään. Julkaise-painikkeella tallentaa järjestelmä tiedot, ja avaa dokumentin vanhapohja.aspx-sivulla. Palaa-painikkeella palataan aloitussivulle.



Kuva 9. Vanha pohja -sivu johon valitun tallenteen tiedot ladattu

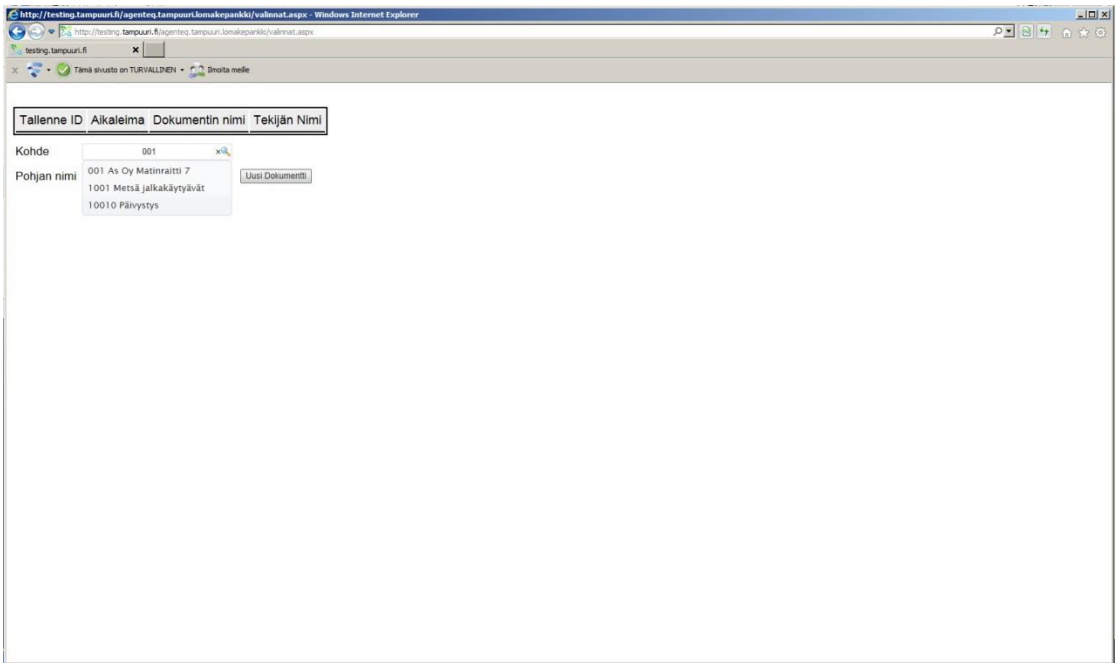
Kun valitaan aloitussivulla vanha dokumentti, latautuu vanhapohja.aspx-sivu johon tietokantaan tallennetut tiedot latautuvat automaattisesti, jos kirjanmerkkeihin on dokumentin tallennuksen yhteydessä tallennettu tietoa (Kuva 9).

Tietoja tekstikentissä voi muokata vapaasti, ja palauta-painikkeella voi järjestelmään tallennetut tiedot palauttaa tekstikenttiin. Luo uusi -painike avaa uusipohja.aspx-sivun johon tiedot latautuvat automaattisesti. Poista-painike poistaa tallennetut tiedot Tampuurista. Julkaise-painike tallentaa tiedot muutoksineen järjestelmään.

6.3 Sivujen toiminnot

Kaikilla sivuilla käytetään käyttöliittymässä vain HTML-elementtejä ja aspx-elementtejä, sekä valinnat.aspx-sivulla myös Jquery-kirjastoa, jotta automaattisesti täydentyvän kentän toiminnot saadaan selaimeen.

6.3.1 Valinnat-sivu



Kuva 10. Työkalun aloitussivu

Valinnat.aspx-sivun (Kuva 10) taulukko on toteutettu käyttäen aspx:n toisitin-elementtiä(repeater) johon tiedonlähteeksi sidottu tieto latautuu sivun latauksen yhteydessä. Sidonta toistimeen tehdään sivulta ja tarvittavat tiedot ladataan sapluunan-mukaiseen (template) rakenteeseen jossa kaikille tiedoille on oma taulukon solu.

```
<form runat="server">
    <h1><%#HttpUtility.HtmlEncode(this.KohteenNimi()) %></h1>
    <table class="otsikko">
    <tr>
        <td class="otsikko">Tallenne ID</td>
        <td class="otsikko">Aikaleima</td>
        <td class="otsikko">Dokumentin nimi</td>
        <td class="otsikko">Tekijän Nimi</td>
    </tr>
        <asp:Repeater runat="server" DataSource="<%#this.HaeKannasta() %>">
            <ItemTemplate>
                <tr class="taulukko">
                    <td class="taulukko"><a href="VanhaPohja.aspx?tallenneID=<%#Eval( "tallenneID" ) %>"<span><%#HttpUtility.HtmlEncode( Eval("tallenneID", "{0:}")) %></span></a></td>
                    <td class="taulukko"><span ><%#HttpUtility.HtmlEncode( Eval("aikaleima", "{0:}")) %></span></td>
                    <td class="taulukko"><span ><%#HttpUtility.HtmlEncode( Eval("dokumentti", "{0:}")) %></span></td>
                    <td class="taulukko"><span ><%#HttpUtility.HtmlEncode( Eval("tekijännimi", "{0:}")) %></span></td>
                </tr>
            </ItemTemplate>
            <AlternatingItemTemplate>
                <tr>
                    <td class="taulukko2"><a href="VanhaPohja.aspx?tallenneID=<%#Eval( "tallenneID" ) %>"<span><%#HttpUtility.HtmlEncode( Eval("tallenneID", "{0:}")) %></span></a></td>
                    <td class="taulukko2"><span ><%#HttpUtility.HtmlEncode( Eval("aikaleima", "{0:}")) %></span></td>
                    <td class="taulukko2"><span ><%#HttpUtility.HtmlEncode( Eval("dokumentti", "{0:}")) %></span></td>
                    <td class="taulukko2"><span ><%#HttpUtility.HtmlEncode( Eval("tekijännimi", "{0:}")) %></span></td>
                </tr>
            </AlternatingItemTemplate>
        </asp:Repeater>
    </table>
```

Kuva 11. Valinnat.aspx taulukko joka täytetään toistimella

Soluihin sijoitettava tieto löytyy Eval-metodilla ja tieto HTML-koodataan, jotta selain näyttää sen oikein tekstinä eikä tulkitse sitä HTML-tekstiksi (Kuva 11).

Käyttämällä vaihtuvaa sapluunaa (AlternatingItemTemplate) saadaan css-tiedostossa määritellyt muotoilut toimimaan niin että perättäiset rivit ovat erivärisiä ja erottuvat näin paremmin.

```
<table class="toiminnot">
<tr>
    <td>
        <asp:Label runat="server" Text="Kohde" AssociatedControlID="txtKohteetID"/>
    </td>
    <td class="toiminnot">
        <asp:TextBox ID="txtKohteetID" runat="server" ToolTip="Kohteen ID ***Testausta Varten***" Text="" CssClass="textbox" Width="200" />
    </td>
</tr>
<tr>
    <td>
        <asp:Label runat="server" Text="Pohjan nimi" AssociatedControlID="txtTemplate"/>
    </td>
    <td class="toiminnot">
        <asp:TextBox ID="txtTemplate" runat="server" Width="200" Text="oikea_vuosikertomus_oma.doc" ToolTip="Laita kenttään käytettävän pohjan nimi(esim. jokupohja.docx) ***Testausta Varten***" CssClass="textbox"/>
    </td>
    <td class="toiminnot">
        <asp:Button runat="server" Text="Uusi Dokumentti" OnClick="Uusi_Click" ToolTip="Tee uusi dokumentti valittua pohjaa käyttäen."/>
    </td>
</tr>
</table>
</form>
```

Kuva 12. Valinnat.aspx-sivun painikkeet ja tekstikentät

Loput elementit sijoitetaan taulukkoon käyttäen aspx:n nimike- (label) ja tekstikenttä-elementtejä (textbox).

```
#region "Properties"
protected UserHelper Käyttäjä
{
    get
    {
        if( null == this._currentUser ) { this._currentUser = new UserHelper( this.Session ); }
        return this._currentUser;
    }
}
protected UserHelper _currentUser { get; set;}
protected string _kt { get; set; }
protected ICollection<Context.juuri> _haku { get; set; }
protected string _kohteenNimi { get; set; }
protected int KohteetID
{
    get
    {
        string s = this.Request.QueryString["kohteetID"];
        //if (string.IsNullOrEmpty(s)) throw new ArgumentException("KohteetID tyhjä");
        if (string.IsNullOrEmpty(s)) return 0;//Demoa varten
        return int.Parse(s);
    }
}
#endregion
```

Kuva 13. Valinnat-sivun ominaisuuksia

Ominasuuksiin (properties) KohteetID ja Käyttäjä ladataan tieto nk. laiskasti(lazy loading) eli tietoa ei haeta ennen kuin sitä tarvitaan. Tämä vähentää latauksen yhteydessä olevaa kuormaa, kun kaikkea tieto ei aina haeta heti (Kuva 13).

KohteetID-ominaisuuteen tieto haetaan sivulle tuoneesta linkistä käyttäen Request-luokan querystring-metodia. Tämä kannattaa aina tehdä ominaisuuteen laiskalla latauksella jotta nk. koodin kronologisuudelta vältytään. Tämä tarkoittaa että kaiken pitää tapahtua oikeassa aikajärjestyksessä jotta koodi toimii.

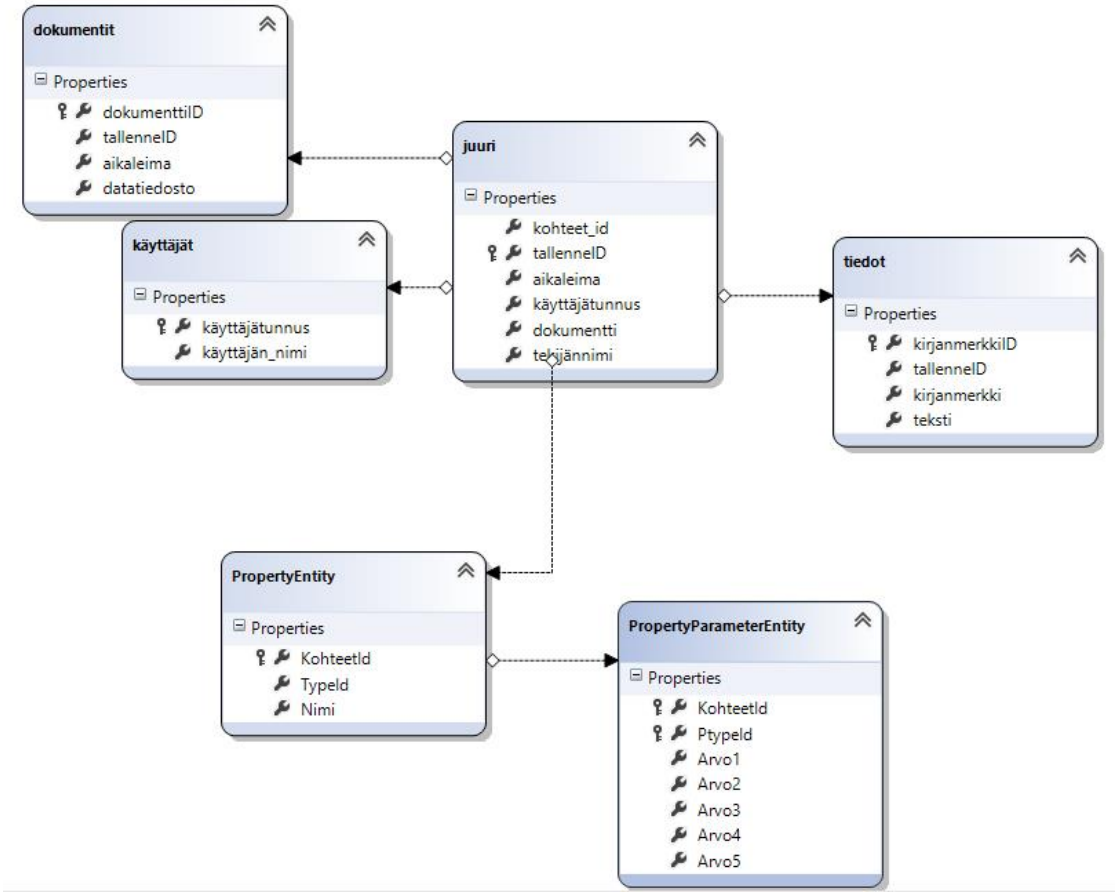
Käyttäjä-ominaisuuteen ladataan Tampuurin kirjastosta userhelper-luokka johon tiedot ladataan sessiosta. Session käyttäminen ei ole suotavaa, koska sen tiedon paikkansapitävyyttä tai eheyttä ei voi taata, mutta koska se on Tampuurissa käytössä ja toimiva, käytän sitä myös tässä projektissa.

```
#region "Methods"
protected string KohteenNimi()
{
    if (object.ReferenceEquals(null, this._kohteenNimi) && !this.KohteetID.Equals(0))
    {
        object arvo = new BookmarkDataProviderProxy("AsOyNimi").GetData(
        new BookmarkDataProviderArguments()
            {
                KohteetId = this.KohteetID,
                ConnectionString = this.Käyttäjä.ConnectionString,
            }
        );
        this._kohteenNimi = (arvo ?? string.Empty).ToString();
    }
    return this._kohteenNimi;
}

protected ICollection<Context.juuri> HaeKannasta()
{
    // Ladataan juuri-taulun tiedot kannasta,
    using (Context.DataClassesDataContext db = new Context.DataClassesDataContext(this.Käyttäjä.ConnectionString))
    {
        var haku =
            from juuri in db.juuris
            where juuri.kohteet_id == this.KohteetID
            select juuri;
        this._haku = haku.Where(j => j.kohteet_id == this.KohteetID).Where(j => j.käyttäjätunnus == this.Käyttäjä.LoginUserId).ToArray();

        //***Iteroidaan automaattisesti seuraavan dokumentin numero uuden dokumentin luonnin testausta varten
        var temp = this._haku.Select(j => j.tallenneID).LastOrDefault();
        temp ++;
        if (string.IsNullOrEmpty(this._kt)) this._kt = "koe"+temp.ToString()+".docx";
        // ***

        return this._haku;
    }
}
#endregion
```

Kuva 14. Valinnat-sivun metodeja

Valinnat sivulla on vain kaksi metodia, HaeKannasta ja KohteenNimi.
KohteenNimi-metodi käyttää kirjaston-tiedontoimittajaa jolla myöhemmin haetaan kyseinen tieto myös muokkaus-sivuille (Kuva 14). Tiedontoimittajista enemmän Kirjasto-kappaleissa.

HaeKannasta-metodi tekee LinqToSql-haun, jolla kaikkien valitun kohteen tallennettujen dokumenttien tiedot haetaan. Tämä rakenne vaati, että luodaan tietokantarajapinta, jota Linq käyttää tiedon hakemiseen. Tämä tehdään lisäämällä uusi projektiin uusi LinqToSql-luokka jonka tiedostopääte on .dbml.



Kuva 15. DataClasses.dbml

Kun luokka on luotu, otetaan sille yhteys tietokantaan, johon hakuja tehdään (Kuva 15). Työkaluun latautuvat kaikki tietokannan taulut, joista valitaan vain ne, joitten tietoja käytetään. Ladattuja tauluja voi muokata vapaasti, niitä voi nimetä, sidoksia voi muuttaa, taulujen elementtejä voi poistaa tai nimetä uudelleen, ja tauluihin sekä sidoksiin voi myös liittää koodia joka muokkaa tietoa jo rajapinnassa jos näin haluaa. Kun malli on valmis, LinqToSql generoi automaattisesti tarvittavan rajapintakoodin ja luo valmiin luokan, jolla tiedot ovat Linq:n käytettävissä. Hyvän käytännön mukaista on poistaa kaikki epäolennainen tieto tauluista ja olla lisäämättä tauluja joita ei tarvita. Tämä vähentää LinqToSql-koodin määrää eli parantaa tehokkuutta ja vähentää inhimillisiä virheitä. Olen jättänyt tauluihin kaikki tiedot, koska kyseessä on demo eikä ole vielä varmaa, mitä tietoa käsitellään lopullisessa versiossa. Rajapinnassa on vielä Tampuuri-integraatiota edeltäviä tauluja, kuten käyttäjät, PropertyEntity sekä PropertyParameterEntity, jotka olivat tarpeellisia ohjelmarakenteen testaamisessa.

Dbml-mallin luonnin jälkeen avataan Linq:lla using-lohkon sisällä tietokantayhteys käyttäen juuri generoitua luokkaa Context.DataClassesDatacontext. On tärkeää käyttää using-lohkoa sillä tietokantahaut ovat .NET-ympäristön roskienkeruutoiminnon ulkopuolella, eli yhteydet voivat jäädä auki ja kuormittaa järjestelmää, jos sen sulkeutumista ei varmisteta. Using-lohko hävittää (dispose) yhteyden riippumatta siitä onnistuiko haku vai ei.

```
protected ICollection<Context.juuri> HaeKannasta()
{
    // Ladataan juuri-taulun tiedot kannasta,
    using (Context.DataClassesDataContext db = new Context.DataClassesDataContext(this.Käyttäjä.ConnectionString))
    {
        var haku =
            from juuri in db.juuris
            where juuri.kohteet_id == this.KohteetID
            select juuri;
        this._haku = haku.Where(j => j.kohteet_id == this.KohteetID).Where(j => j.käyttäjätunnus == this.Käyttäjä.LoginUserId).ToArray();

        //***Iteroidaan automaattisesti seuraavan dokumentin numero uuden dokumentin luonnin testausta varten
        var temp = this._haku.Select(j => j.tallenneID).LastOrDefault();
        temp ++;
        if (string.IsNullOrEmpty(this._kt)) this._kt = "koe"+temp.ToString()+".docx";
        // ***

        return this._haku;
    }
}
```

Kuva 16. HaeKannasta-metodi Valinnat-sivulla

Varsinainen haku tehdään Juuri-tauluun käyttäen kohteen tunnusta (kohteetID) hakuavaimena (Kuva 16). Tämän jälkeen suodatetaan hakua vielä käyttäjän tunnuksella jotta näkyviin tulee vain käyttäjän tekemät dokumentit. Tämä tulee olemaan turha toiminto myöhemmin, kun Tampuuriin määritellyt käyttöoikeudet otetaan käyttöön.

```
#region "Events"
protected void Uusi_Click(object sender, EventArgs e)
{
    Response.Redirect(string.Format("UusiPohja.aspx?kohteetID={0}&pohjannimi={1}&kt={2}", this.KohteetID, txtTemplate.Text, this._kt));
}

protected void Päivitä_Click(object sender, EventArgs e)
{
    this.DataBind();
}
#endregion
```

Kuva 17. Valinnat-sivun painikkeiden toiminnot

Lopuksi on vielä luo uusi- ja päivitä-painikkeiden toiminnot (Kuva 17). Painikkeiden toiminnot tehdään aspx:n tapahtumiksi (event), jolloin se pitää huolen niiden laukaisusta (trigger). Sivukutsuihin liitetään querystring-parametreiksi pakolliset tiedot kuten kohteen tunnus (KohteetID) ja pohjan tiedostonimi (pohjannimi). Kt-parametri on vain testauksen helpottamista varten.

6.3.2 Uusi pohja -sivu

Uusi pohja -sivulla käytetään aspx:n toistin-elementtiä (repeater) samankaltaisesti kuin valinnat-sivulla (Kuva 18). Toistimeen ladataan kirjanmerkit sekä niiden tekstit. Samoin kuin valinnat-sivulla toistimen alle tulee taulukko, johon painikkeet sijoitetaan, sekä tekstikenttä johon tallennettavan dokumentin nimi sijoitetaan.

```
<html>
<head runat="server">
    <title>Vuosikertomus - Uusi</title>
</head>
<body>
    <form runat="server">
    <h1><%#HttpUtility.HtmlEncode(this.KohteenNimi()) %></h1>
    <asp:Repeater runat="server" DataSource="<%#this.HaeKirjanmerkit() %>">
        <ItemTemplate>
            <br/>
            <span class="otsikot"><%#HttpUtility.HtmlEncode( Eval("kmNimi", "{0:}")) %></span>
            <br/>
            <textarea class="tekstit" name="<%#HttpUtility.HtmlEncode( Eval("kmNimi", "{0:}")) %>" rows="5" cols="50"><%#HttpUtility.HtmlEncode( Eval("Teksti", "{0:}")) %></textarea>
            <br/>
        </ItemTemplate>
    </asp:Repeater>
    <table class="toiminnot">
        <tr>
            <td class="toiminnot">
                <asp:Button runat="server" text="Julkaise" OnClick="Julkaise_Click" ToolTip="Julkaise dokumentti."/>
            </td>
            <td class="toiminnot">
                <asp:Button runat="server" text="Päivitä Tiedot" OnClick="Päivitä_Click" ToolTip="Hakee tunnettuinin kirjanmerkkeihin uudet tiedot Tampuurista"/>
            </td>
            <td class="toiminnot">
                <asp:Button runat="server" text="Esikatselu" OnClick="Preview_Click" ToolTip="Esikatsele dokumenttiä."/>
            </td>
            <td class="toiminnot">
                <asp:Button runat="server" text="Palaa" onclick="btnAloitussivu_Click"/>
            </td>
        </tr>
    </table>
    <asp:Label CssClass="otsikot" runat="server" Text="Tallennettavan tiedoston nimi" AssociatedControlID="txtTiedostonimi" />
    <asp:TextBox CssClass="textbox" id="txtTiedostonimi" runat="server" ToolTip="Tallennettavan tiedoston nimi(esim. talonkirja.docx)." />
    </form>
</body>
</html>
```

Kuva 18. UusiPohja.aspx

Koska ei ole varmaa, mitä kautta dokumenttitoiminnot tullaan lopullisesti tekemään, tekstikenttään ei ole liitetty validointia. Samoin kuin esim. sähköposti tai päivämäärä-kentissä on tärkeää, että muoto tarkistetaan, ennen kuin toiminnot hyväksytään mahdollisten käyttäjävirheiden poistamiseksi. Tässä kohdassa kyse ei ole isosta vahingosta, sillä tiedosto tallentuu, jos ei käytetä erikoismerkkejä tai jätetä tyhjää merkkijonoa kenttään.

```
protected int KäyttäjäID
{
    get
    {
        string s = this.CurrentUser.LoginUserId.ToString();
        //if (string.IsNullOrEmpty(s)) throw new ArgumentException("KäyttäjäID tyhjä");
        if (string.IsNullOrEmpty(s)) return 0;//Testausta helpottamaan
        return int.Parse(s);
    }
}
protected int KohteetID
{
    get
    {
        string s = this.Request.QueryString["kohteetID"];
        //if (string.IsNullOrEmpty(s)) throw new ArgumentException("KohteetID tyhjä");
        if (string.IsNullOrEmpty(s)) return 0;//Testausta helpottamaan
        return int.Parse(s);
    }
}
protected int TallenneID
{
    get
    {
        string s = this.Request.QueryString["tallenneID"];
        //if (string.IsNullOrEmpty(s)) throw new ArgumentException("TallenneID tyhjä");
        if (string.IsNullOrEmpty(s)) return 0;//Testausta helpottamaan
        return int.Parse(s);
    }
}
protected string PohjanNimi
{
    get
    {
        string s = this.Request.QueryString["pohjannimi"];
        if (string.IsNullOrEmpty(s)) throw new ArgumentException("Pohjannimi tyhjä");
        return s;
    }
}
```

Kuva 19. Uusi pohja-sivun ominaisuuksia

Samoin kuin Valinnat-sivulla ominaisuuksiin ladataan laiskasti käyttäjätiedot, kohteen tunnus, tallenteen tunnus, jos käytetään vanhaa dokumenttia pohjana, sekä pohja-dokumentin nimi (Kuva 19).

```
#region "Methods"
protected ICollection<Kirjanmerkki> HaeKirjanmerkit()
{
    var kirjanmerkit = this.LataaKirjanmerkit().ToList();
    kirjanmerkit.ForEach(kirjanmerkki => this.Täytä(kirjanmerkki));
    return kirjanmerkit;
}
```

Kuva 20. Uusi pohja-sivun HaeKirjanmerkit-metodi

Uusipohja -sivun toistimeen sidottu HaeKirjanmerkit-metodi käynnistää tiedonhaku-toiminnot ajamalla LataaKirjanmerkit-metodin, jolla kirjanmerkit luetaan dokumentis-tä ominaisuuteen (Kuva 20).

```
protected IEnumerable<Kirjanmerkki> LataaKirjanmerkit()
{
    //Lataa kirjanmerkit dokumentistä

    if (this._dokumentti == null) this._dokumentti = new DokumenttiKäsittelijä(System.IO.Path.Combine(OletusKansio, this.PohjanNimi));

    var kirjanmerkit = this._dokumentti.KirjanmerkkiLukija();

    //Poistetaan aspose.wordsin luoma kirjamerkki hausta
    foreach (var k in kirjanmerkit)
    {
        if (!k.Name.Equals("_GoBack")) yield return new Kirjanmerkki() { kmNimi = k.Name };
    }

}
```

Kuva 21. Uusi pohja-sivun LataaKirjanmerkit-metodi

Kirjanmerkki-luokka on yksittäisiä kirjanmerkkejä varten tehty luokka, jolla on omi-naisuutena kmNimi sekä Teksti. Kirjanmerkit syötetään LataaKirjanmerkit-metodilla tämän luokan muodossa IEnumerable-rajapintaan (Kuva 21). IEnumerablea kuten ICollenctionia käytettäessä tulee muistaa, että rajapinta ei varsinaisesti tee mitään toi-mintoa, vaan se pitää asettaa tietynlaiseen muotoon, että toiminnot tapahtuvat. Esi-merkiksi käyttämällä ToList-metodia, joka tekee rajapintaa käyttäen listan.

Itse LataaKirjanmerkit-metodi kutsuu Dokumenttikäsittelijä-luokkaa, johon Aspo-se.Words:n kautta käsiteltävän dokumentin toiminnot on sijoitettu. Käsittelijä ladataan _dokumentti-ominaisuuteen laiskalla latauksella jotta käsittelijä ja dokumentti lada-taan vain kerran.

```
public IEnumerable<Aspose.Words.Bookmark> KirjanmerkkiLukija()
{
    var kirjanmerkit = this.dokumentti.Range.Bookmarks.Cast<Aspose.Words.Bookmark>().Where(k => !string.IsNullOrEmpty(k.Name)).ToList();
    return kirjanmerkit;
}
```

Kuva 22. Dokumenttikäsittelijä-luokan KirjanmerkkiLukija-metodi

Dokumenttikäsittelijällä on KirjanmerkkiLukija-metodi (Kuva 22), joka lataa doku-mentin kirjanmerkit ominaisuuteen Aspose.Words.Bookmark-luokan tyyppisenä IEn-umerable-joukkona. Metodi käyttää Aspose.Words-kirjaston metodeja lukeakseen kaikki ne kirjanmerkit, jotka eivät ole tyhjiä.

Koska Aspose.Words luo kirjanmerkeistä valikoima-ryhmiä (range), jotka erotetaan _GoBack nimisellä kirjanmerkillä vaikka valikoimia olisikin vain yksi, poistetaan kyseinen kirjanmerkki palautettavasta joukosta.

```
protected Kirjanmerkki Täytä(Kirjanmerkki kirjanmerkki)
{
    string value = this.Request.Params[kirjanmerkki.kmNimi];
    if (!this._päivitä) kirjanmerkki.Teksti = value ?? this.HaePohjasta(kirjanmerkki.kmNimi) ?? this.HaeKannasta(kirjanmerkki.kmNimi);
    else
    {
        if (string.IsNullOrEmpty(this.HaeKannasta(kirjanmerkki.kmNimi))) kirjanmerkki.Teksti = value;
        else kirjanmerkki.Teksti = this.HaeKannasta(kirjanmerkki.kmNimi);
    }
    return kirjanmerkki;
}
```

Kuva 23. Uusi pohja-sivun Täytä-metodi

Kun LataaKirjanmerkit-metodi on saanut kirjanmerkit, se kutsuu jokaiselle joukon kirjanmerkille Täytä-metodia (Kuva 23), joka hakee ensin sivulta kaikkien tekstikenttien sisällön. Tässä vaiheessa tällä ei ole väliä, koska kentät ovat tyhjiä, mutta sivun uudelleen latauksessa tulee kirjoitetun tiedon säilyä.

Metodiin on sijoitettu myös päivitystä varten toiminto joka _päivitä-ominaisuuden ollessa tosi (true), hakee tiedot uudestaan tietokannasta, ja jos tieto on saatavilla sijoittaa tiedon kirjanmerkkiin. Muutoin tekstikentässä oleva tieto säilyy kentässä.

Jos _päivitä-ominaisuus on epätosi (false), niin kirjanmerkin tekstiksi sijoitetaan tekstikentästä löytyvä arvo. Jos kentän arvo on tyhjä (null), kuten sen ensi kertaa ladattaessa tulee olla, käynnistetään tietokantahaku metodilla HaeKannasta.

```
protected string HaeKannasta(string kmNimi)
{
    object arvo = new BookmarkDataProviderProxy(kmNimi).GetData(
    new BookmarkDataProviderArguments()
    {
        KohteetId = this.KohteetID,
        ConnectionString = CurrentUser.ConnectionString,
    }
    );
    return (arvo ?? string.Empty).ToString();
}
```

Kuva 24. Uusi pohja-sivun HaeKannasta-metodi

HaeKannasta-metodi (Kuva 24) kutsuu kirjaston BookmarkDataProviderProxy-luokkaa ja sen GetData-metodia. Parametreiksi tulee kohteen tunnus sekä yhteysmerkkijono. Jos kirjanmerkki on tuettu, tiedontoimittaja suorittaa tiedonhaun ja antaa tiedon ohjelmalle. Jos kirjanmerkkiä ei ole vielä tuettu, tiedontoimittaja palauttaa tyhjän tiedon. BookmarkDataProvider-tiedontoimittajan toiminta on selitetty kirjastoluvun tiedontoimittajat-kappaleessa.

```
protected void Esikatselu()
{
    var kirjanmerkit = HaeKirjanmerkit();

    //Aspose.Words
    if (object.ReferenceEquals(null, this._dokumentti)) this._dokumentti = new DokumenttiKäsittelijä(System.IO.Path.Combine(OletusKansio, this.PohjanNimi));
    this._dokumentti.KirjanmerkkiTäyttäjä(kirjanmerkit);
    this._dokumentti.KirjanmerkkiTallentaja(System.IO.Path.Combine(OletusKansio, this.txtTiedostonimi.Text));
    this._dokumentti.Esikatselija(this.Response, this.txtTiedostonimi.Text);
}
```

Kuva 25. Esikatselu-metodi

Esikatselu-painikkeella käynnistetään Esikatselu-metodi (Kuva 25), joka hakee sivulta kirjanmerkit tietoineen, ja tämän jälkeen tallentaa ne Word-dokumenttiin käyttäen DokumenttiKäsittelijä-luokkaa ja sen KirjanmerkkiTäyttäjä- sekä KirjanmerkkiTallentaja-metodeja. Tämän jälkeen käynnistetään Esikatselija-metodi joka lähettää dokumentin käyttäjän selaimen kautta Wordin tarkasteltavaksi.

```
protected void Julkaise_Click(object sender, EventArgs e)
{
    // Tallennetaan tekstikenttien tiedot tietokantaan ja tämän jälkeen dokumenttiin. Ohjataan VanhaPohja sivulle jossa dokumenttiä voi editoida
    var kirjanmerkit = HaeKirjanmerkit();

    //Aspose.Words
    if (object.ReferenceEquals(null, this._dokumentti)) this._dokumentti = new DokumenttiKäsittelijä(System.IO.Path.Combine(OletusKansio, this.PohjanNimi));
    this._dokumentti.KirjanmerkkiTäyttäjä(kirjanmerkit);
    this._dokumentti.KirjanmerkkiTallentaja(System.IO.Path.Combine(OletusKansio, this.txtTiedostonimi.Text));

    var tallenneID = new TietoKantaKäsittelijä().TallennaUusi(this.KohteetID, this.CurrentUser.LoginUserId, this.CurrentUser.LoginName, this.txtTiedostonimi.Text, OletusKansio, kirjanmerkit, this.CurrentUser.ConnectionString);
    Response.Redirect(string.Format("VanhaPohja.aspx?tallenneID={0}", tallenneID));
}
```

Kuva 26. Uusi pohja-sivun julkaise-painikkeen toiminnot

Julkaise-painikkeella (Kuva 26) dokumentin tallennus tapahtuu kuten esikatselijassa, mutta esikatselun sijaan tietokantaan tallennetaan tiedot ja dokumentti sekä avataan dokumentti Vanhapohja-sivulle. Tietokantatallennus tapahtuu TietokantaKäsittelijä-luokalla ja sen TallennaUusi-metodilla.

```
public int TallennaUusi(int kohteetID, int käyttäjäID, string käyttäjänimi, string tiedostonimi, string oletuskansio, ICollection<Kirjanmerkki> kirjanmerkit, string connectionstring)
{
    var TiedostoBinary = new TiedostoKäsittelijä();
    //TiedostoBinary.LueTiedosto(tallennettavatiedosto);

    using (Context.DataClassesDataContext db = new Context.DataClassesDataContext(connectionstring))
    {
        var uusiJuuri = new Context.juuri()
        {
            kohteet_id = kohteetID,
            aikaleima = DateTime.Now,
            käyttäjätunnus = käyttäjäID,
            dokumentti = tiedostonimi,
            tekijännimi = käyttäjänimi
        };

        uusiJuuri.dokumentits.Add(new Lomakepankki.Context.dokumentit()
        {
            tallenneID = uusiJuuri.tallenneID,
            aikaleima = uusiJuuri.aikaleima,
            datatiedosto = TiedostoBinary.LueTiedosto(System.IO.Path.Combine(oletuskansio, tiedostonimi))
        });

        foreach (var k in kirjanmerkit)
        {
            uusiJuuri.tiedots.Add(new Lomakepankki.Context.tiedot()
            {
                tallenneID = uusiJuuri.tallenneID,
                kirjanmerkki = k.kmNimi,
                teksti = k.Teksti
            });
        }

        db.juuris.InsertOnSubmit(uusiJuuri);
        db.SubmitChanges();

        return uusiJuuri.tallenneID;
    }
}
```

Kuva 27. TietokantaKäsittelijä-luokan TallennaUusi-metodi

TallennaUusi-metodi (Kuva 27) käyttää Tiedostokäsittelijä-luokkaa, jonka LueTiedosto- ja TallennaTiedosto-metodeihin on tiedoston tallennus- ja lukemistoiminnot sijoitettu.

Kuten aiemminkin tietokantatoiminnallisuus tehdään käyttäen using-lohkoa. Tietokantatoiminnallisuus on tehty käyttäen LinqToSql:ää ja aiemmin käsiteltyä DataClasses.dbml-mallia.

Aluksi luodaan uusiJuuri-muuttujaan sijoitettava uusi Juuri-tyyppiä oleva taulu, johon Juuritauluun menevät tiedot sijoitetaan. Tämän jälkeen lisätään samaan muuttujaan uusi Dokumentit-tyyppiä oleva alitaulu, johon dokumentit-tauluun menevät tiedot sijoitetaan. Tämän jälkeen lisätään samalla tavalla jokainen kirjanmerkki tietoineen tiedot-alitauluun.

Tässä vaiheessa kaikki lisättävä tieto on vain uusiJuuri-muuttujassa. Koska tiedot ovat muuttujassa jo oikeassa muodossa ja rakenteessa, voidaan ne ajaa suoraan rajapintaa hyväksi käyttäen LinqToSql komennoilla tietokantaan.

LinqToSql on helppo ja monipuolinen tapa hoitaa tietokantatoiminnot, mutta koska se pohjaa automaattisesti luotuun koodiin, se ei ole vaativissa toiminnoissa kyllin tehokas. Jos projektissa käsiteltäisiin isompia tietomääriä tai toimintoja tapahtuisi usein, tietokannalle pitäisi kirjoittaa ORM- rajapinta (Object-relational mapping), joka antaa

samanlaisen ohjelmallisen helppokäyttöisyyden suuremmalla tehokkuudella ja luotettavuudella.

6.3.3 Vanha pohja -sivu

Vanhapohja -sivu on toiminnoiltaan hyvin lähellä Uusipohja-sivua, joten tässä luvussa esittelen vain kohdat jossa toiminnot eroavat toisistaan.

Käyttöliittymässä eroina ovat vain poista-, palauta-, luo uusi- sekä julkaise-painike joka tällä sivulla päivittää tietoja tietokantaan. Poista-painikkeella poistetaan tallenne dokumentteineen tietokannasta. Palauta-painikkeella tuodaan dokumentin tiedot uudestaan tietokannasta ja kirjoitetaan sivulla olevat tiedot yli. Luo uusi -painike tallentaa sivulla olevat tiedot tietokantaan sekä dokumenttiin ja avaa Uusipohja -sivun käyttäen tallennettua dokumenttia uuden pohjana.

```
protected void HaePohjaTiedot()
{
    // Haetaan tallenneID:n perusteella kannasta dokumentin tiedot ja tallennetaan viimeisin dokumentti käsittelyä varten palvelimen kansioon.
    if(object.ReferenceEquals(null, this._haku)) this._haku = HaeKannasta();

    this._tiedostonimi = this._haku.Single(j => j.tallenneID == this.TallenneID).dokumentti;
    int käyttäjä = this._haku.SingleOrDefault(j => j.tallenneID == this.TallenneID).käyttäjätunnus;
    int kohde = this._haku.SingleOrDefault(j => j.tallenneID == this.TallenneID).kohteet_id;

    if (käyttäjä.Equals(0) || kohde.Equals(0)) throw new NullReferenceException("Käyttäjätunnusta tai kohdetunnusta ei löytynyt");
    this._kohteetID = kohde;

    var temp = this._haku.Single(j => j.tallenneID == this.TallenneID).dokumentits.OrderByDescending(d => d.aikaleima).FirstOrDefault().datatiedosto;
    new TiedostoKäsittelijä().TallennaTiedosto(temp, System.IO.Path.Combine(OletusKansio, this._tiedostonimi));
}
```

Kuva 28. Vanha pohja-sivun HaePohjaTiedot-metodi

Vanhapohja-sivu tarvitsee tallenteen tunnuksen (tallenneID), jolla kaikki tieto saadaan haettua. Tallennetunnus saadaan sivun linkin kyselymerkkijonosta (querystring) laiskalla latauksella TallenneID-ominaisuuteen.

Muut tarvittavat tiedot ladataan HaePohjaTiedot-metodilla (Kuva 28) käyttäen tallennetunnusta avaimena. Tiedostonimi ja kohdetunnus haetaan ja asetetaan ominaisuuksiin, lisäksi kohdetunnus ja käyttäjätunnus tarkastetaan, jotta ne on asetettu tietokantaan. Jos jompikumpi on tietokannassa 0 tai tyhjä (null) nostaa ohjelma poikkeuksen. Tämä toiminto on täysin turha testausta lukuun ottamatta, sillä tiedon eheys hoidetaan tietokantatasolla eikä tyhjiä sallita kummassakaan tiedossa. Jätin kuitenkin testaustani varten tekemäni tarkistukset koodiin, koska on tärkeää asettaa vastaavia tarkistuksia rakennusvaiheessa varmistamaan toimintoja, ja ilmoittamaan jos jokin menee pieleen.

Tdd-kehityksessä tämä hoidetaan tekemällä kattavat testit ennen kehitystyötä, mutta sen toimintaa esitellään Kirjasto-luvun yksikkötestit-kappaleessa.
Viimeiseksi haetaan tietokantaan tallennettu viimeisin dokumentti palvelimen työhakemistoon jotta se voidaan avata sivulle. Tein tietokannan niin että kaikki dokumenttien tallennukset jäävät tietokantaan. Sivuilla näytetään kuitenkin aina vain viimeisin dokumentin tallennus ja sen tiedot. Tämä siksi, että dokumentit eivät tuhoudu vahingossa. En ole tehnyt järjestelmää vanhempien dokumenttien palauttamista varten, mutta sellainen tulee varmasti lopulliseen versioon, sillä Tampuurin dokumenttienhallinta-moduuli tukee palautusta.

Tämän jälkeen toiminnot ovat miltei identtiset UusiPohja-sivun kanssa. Ensin ladataan kirjanmerkit pohjasta, joka on työkansioon kopioitu viimeisin tälle tallennetunnukselle tehty dokumentti. Erona on tiedonhaku, koska nyt tietoja ei haeta tiedontoimittajilta vaan ne haetaan suoraan tietokannasta.

```
protected string HaeTeksti(string kmNimi)
{
    //Haetaan kirjanmerkkikohtaiset tekstit
    if (object.Equals(this._haku, null)) this._haku = HaeKannasta();

    var tiedot = this._haku.Single(j => j.tallenneID == this.TallenneID).tiedots.Single(t => t.kirjanmerkki.Equals(kmNimi));
    return tiedot.teksti ?? string.Empty;
}


protected ICollection<Context.juuri> HaeKannasta()
{
    //Haetaan tiedot tietokannasta
    using (Context.DataClassesDataContext db = new Context.DataClassesDataContext(this.Käyttäjä.ConnectionString))
    {
        var loadOptions = new System.Data.Linq.DataLoadOptions();
        loadOptions.LoadWith<Agenteq.Tampuuri.Lomakepankki.Context.juuri>(j => j.tiedots);
        loadOptions.LoadWith<Agenteq.Tampuuri.Lomakepankki.Context.juuri>(j => j.dokumentits);
        db.LoadOptions = loadOptions;

        var haku =
            from juuri in db.juuris
            select juuri;
        return haku.ToArray();
    }
}
```

Kuva 29. Vanha pohja -sivun metodeja

HaeTeksti-metodi (Kuva 29) hakee tietokannasta tallennetunnukselle tehdyn kirjanmerkin tiedon suodattamalla sen _haku-ominaisuuteen HaeKannasta-metodilla asetetusta tiedosta.

HaeKannasta-metodi (Kuva 29) toimii samalla tavalla kuin muilla sivuilla, mutta koska nyt tarvitaan tietoa useammasta taulusta, lisätään Juuritaulun latauskomentoon myös tiedot- ja dokumentitalitaulut.

Julkaise-painikkeella tallennetaan avattu dokumentti uudestaan käyttämällä TietokantaKäsittelijä-luokan PäivitäTietoja-metodia (Kuva 30).

```
public bool PäivitäTietoja(ICollection<Kirjanmerkki> kirjanmerkit, int tallenneID, int käyttäjäid, string käyttäjänimi, string tiedostonimi,string oletuskansio, string connectionstring)
{
    var TiedostoBinary = new TiedostoKäsittelijä();
    using (Context.DataClassesDataContext db = new Context.DataClassesDataContext(connectionstring))
    {
        //Päivitetään aikaleima juureen
        var päivitys_juuri = db.juuris.Single(j => j.tallenneID == tallenneID);
        päivitys_juuri.aikaleima = DateTime.Now;
        päivitys_juuri.tekijännimi = käyttäjänimi;
        päivitys_juuri.käyttäjätunnus = käyttäjäid;

        //Päivitetään kirjanmerkkien tiedot
        var päivitys_tiedot = db.tiedots.Where(t => t.tallenneID == tallenneID);
        foreach (var kirjanmerkki in kirjanmerkit)
        {
            var tmp = päivitys_tiedot.Single(t => t.kirjanmerkki == kirjanmerkki.kmNimi);
            tmp.teksti = kirjanmerkki.Teksti;
        }

        //Tallennetaan uusi dokumentti kantaan
        var tallennus = new Lomakepankki.Context.dokumentit()
        {
            tallenneID = tallenneID,
            aikaleima = päivitys_juuri.aikaleima,
            datatiedosto = TiedostoBinary.LueTiedosto(System.IO.Path.Combine(oletuskansio, tiedostonimi))
        };
        db.dokumentits.InsertOnSubmit(tallennus);
        db.SubmitChanges();
    }
    return true;
}
```

Kuva 30. TietokantaKäsittelijä-luokan PäivitäTietoja-metodi

Päivitys tapahtuu tiedon lisäyksen kaltaisesti, mutta nyt tiedot kannattaa päivittää yksitellen juuri- ja tiedot-taulujen riveille. Tauluista suodatetaan oikeat rivit Linq:lla käyttäen tallennetunnusta (tallenneID) ja uudet tiedot kirjoitetaan vanhojen päälle. Tämän jälkeen lisätään julkaise-painikkeen painamisen yhteydessä tallennettu dokumentti dokumentit-tauluun binääritietona. Lopuksi ajetaan komennot jotka lisäävät (InsertOnSubmit) ja päivittävät (SubmitChanges) tietokannan tiedot.

```
public bool PoistaTietoja(int tallenneID, string connectionstring)
{
    using (Context.DataClassesDataContext db = new Context.DataClassesDataContext(connectionstring))
    {
        var PoistaJuuresta = db.juuris.Single(j => j.tallenneID == tallenneID);
        db.juuris.DeleteOnSubmit(PoistaJuuresta);

        var PoistaTiedoista = db.tiedots.Where(t => t.tallenneID == tallenneID);
        foreach (var poista in PoistaTiedoista)
        {
            db.tiedots.DeleteOnSubmit(poista);
        }

        var PoistaDokumenteista = db.dokumentits.Where(d => d.tallenneID == tallenneID);
        foreach (var poista in PoistaDokumenteista)
        {
            db.dokumentits.DeleteOnSubmit(poista);
        }

        db.SubmitChanges();
    }
    return true;
}
```

Kuva 31. TietokantaKäsittelijä-luokan PoistaTietoja-metodi

Poista-painikkeella (Kuva 31) poistetaan sivulla auki oleva dokumentti tietoineen tietokannasta. Samoin kuin päivityksessä, suodatetaan Linq:lla poistettava tieto tauluista ja käytetään poistometodia (DeleteOnSubmit) jokaiselle taululle. Kun kaikki poistettavat tiedot on lisätty, ajetaan päivityskomento (SubmitChanges), jolla tietomuutokset tallennetaan tietokantaan.

6.4 Kirjasto

Projektiin tehdään myös kirjasto, johon kaikki tiedontarjoajat kirjoitetaan. Tarkoituksena on, että jokaisen moduulin vastuuryhmä kirjoittaa tiedontarjoajat oman moduulin tietoihin, joita päätetään tukea.

```
public class BookmarkDataProviderProxy : IBookmarkDataProvider
{
    private BookmarkDataProviderFactory _factory;
    private string _bookmark { get; set; }

    public BookmarkDataProviderProxy(string bookmark)
    {
        this._bookmark = bookmark;
        if (object.ReferenceEquals(null, this._factory)) this._factory = new BookmarkDataProviderFactory();
    }

    public object GetData(BookmarkDataProviderArguments args)
    {
        if (object.ReferenceEquals(null, args)) throw new ArgumentNullException("Argumentit on tyhjä");
        return this._factory.Create(this._bookmark).GetData(args);
    }

}
```

Kuva 32. Kirjaston BookmarkDataproviderProxy-luokka

Tiedontarjoaja-ajattelun mukaisesti kirjastoprojekti tehdään rajapintaa hyväksikäyttäen niin, että palvelun hakutiedon vastaanottava osa on välityspalvelu (BookmarkDataproviderProxy), joka antaa tiedon suoraan jos tieto on jo haettu. Jos tietoa ei ole vielä haettu luo välityspalvelu tehtaan (BookmarkDataproviderFactory), joka luo vuorostaan oikeantyyppisen tiedontoimittajan. Jos kirjanmerkkiä ei tueta, luodaan oletuksena tyhjä tiedontoimittaja.

```
public class BookmarkDataProviderFactory
{
    public BookmarkDataProviderFactory() : base() { ;}

    public IBookmarkDataProvider Create(string bookmark)
    {
        switch (bookmark)
        {
            case "AsOyNimi": return new AsOyNimi();
            case "Asuinhuoneistoja": return new AsuinHuoneistoja();
            case "AsuinhuoneistojenAla": return new AsuinHuoneistoAla();
            case "Asukasmäärä": return new AsukasMäärä();
            case "Autopaikkoja": return new Autopaikat();
            case "Kiinteistötunnus": return new Kiinteistötunnus();
            case "Osoite": return new Osoite();
            case "RakennustenMäärä": return new RakennustenMäärä();
            case "RakennustenTilavuus": return new RakennustenTilavuus();
            case "AsOyRekisteröity": return new AsOyRekisteröity();
            case "TonttiAla": return new TonttiAla();
            case "Hallintaperuste": return new Hallintaperuste();
            case "Valmistumisvuosi": return new Valmistumisvuosi();
            case "Ytunnus": return new Ytunnus();
            default: return EmptyImplement.Instance;
        }
    }
}
```

Kuva 33. BookmarkDataProviderFactory-luokka

Tehtaan (BookmarkDataProviderFactory) Create-metodi (Kuva 33) ottaa kirjanmerkin vastaan ja valitsee sekä luo tiedontoimittajan, joka palautetaan välityspalvelulle (BookmarkDataProviderProxy). Nämä tiedontoimittajat olen kirjoittanut itse demoa varten, lopullisessa versiossa tiedontoimittajia on enemmän.

Kun välityspalvelulla on luotuna oikeanlainen tiedontoimittaja-luokka, se käyttää tiedontoimittajan GetData-metodia joka tekee varsinaisen tiedonhaun ja palauttaa tuloksen välityspalvelulle, joka palauttaa tiedon tietoa pyytäneelle luokalle.

```
protected string HaeKannasta(string kmNimi)
{
    object arvo = new BookmarkDataProviderProxy(kmNimi).GetData(
    new BookmarkDataProviderArguments()
    {
        KohteetId = this.KohteetID,
        ConnectionString = CurrentUser.ConnectionString,
    }
    );
    return (arvo ?? string.Empty).ToString();
}
```

Kuva 34. Tiedontoimittajaa kutsuva metodi uusi pohja-sivulla

Kun tiedontoimittajaa kutsutaan ohjelmasta (Kuva 34), tulee mukaan liittää myös kohteen tunnus ja yhteysmerkkijono BookmarkDataProviderArguments-luokan ominaisuuksina. Tulee muistaa, että järjestelmään ei määritellä palautettavan tiedon tyyppiä, vaan se otetaan vastaan objektina, joka sitten muutetaan merkkijonoksi tekstikenttään lisäystä varten. Tämä mahdollistaa sen että myöhemmin tietoa voidaan ottaa vastaan minkä tyyppisenä tahansa, eikä ole sitouduttu vain yhdentyyppisen tiedon tarjoamiseen.

```
public interface IBookmarkDataProvider
{
    object GetData(BookmarkDataProviderArguments args);
}
```

Kuva 35. BookmarkDataProvider-rajapinta

Kaikki tiedontarjoajat toteuttavat rajapinnan (IBookmarkDataProvider) ja sen määrittelemän GetData-metodin (Kuva 35). Rajapinta takaa että tiedontarjoaja toimii ulospäin aina samalla tavalla.

```
public abstract class BookmarkDataProvider : IBookmarkDataProvider
{
    public BookmarkDataProvider() : base() { ;}

    #region IBookmarkDataProvider Members

    public abstract object GetData(BookmarkDataProviderArguments args);

    #endregion
}
```

Kuva 36. Rajanpinnan toteuttava BookmarkDataProvider-luokka

Rajapinnalle luodaan abstrakti luokka, joka toteuttaa kaikki rajapinnan metodit (Kuva 36). Tämä on tehty suojaamaan rajapintaa käyttäviä luokkia. Jos rajapintaan myöhemmin lisätään toimintoja, pitäisi kaikkiin luokkiin päivittää vastaavat metodit, mutta luomalla väliin abstraktin luokan, joka toteuttaa kaikki rajapinnan toiminnot, saadaan vanhemmat luokat säilyttämään yhteensopivuus rajapintaan. Nämä abstraktit-luokat toteuttavat toiminnot oletusarvoilla.

6.4.1 Tiedon toimittajat

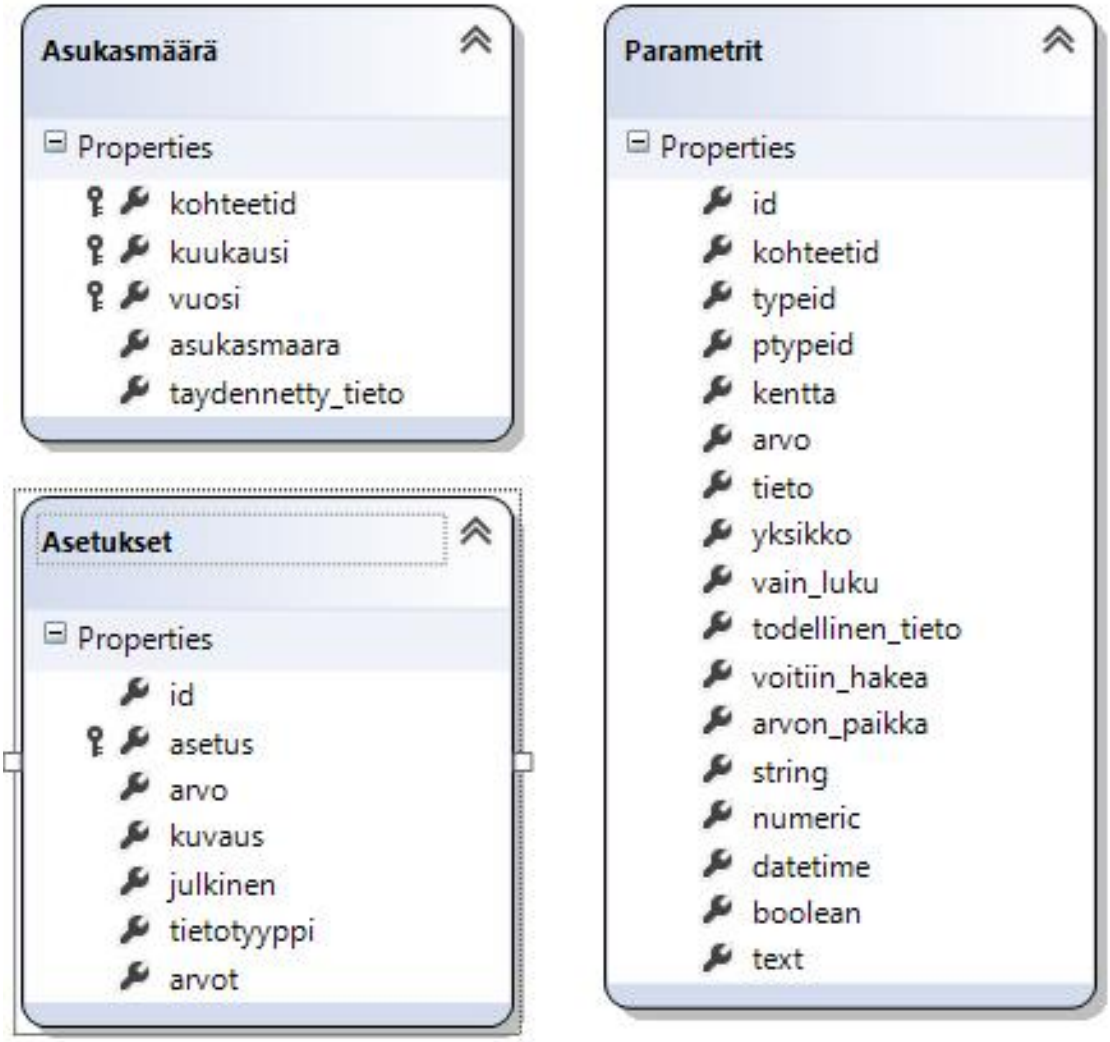
Jokainen tiedontoimittaja on oma luokkansa. Näin tiedontoimittajat toimivat omina osinaan helpottaen virheenkorjausta, ja vähentävät koodin monimutkaisuutta ja luettavuutta. Tämä mahdollistaa myös yksikkötestauksen jokaiselle osalle erikseen, yksikkötestauksesta enemmän Yksykkötestit-luvussa.

```
public class AsOyNimi : BookmarkDataProvider
{
    public AsOyNimi() : base() { ;}

    public override object GetData(BookmarkDataProviderArguments args)
    {
        if (object.ReferenceEquals(null, args)) throw new ArgumentNullException();
        using (Context.DataDataContext db = new Context.DataDataContext(args.ConnectionString))
        {
            var ptypeid = int.Parse(db.Asetuksets.Single(asetukset => asetukset.asetus.Equals("koht_main.yhtiö.nimi")).arvo);
            var data = db.Parametrits.Where(parametrit => parametrit.ptypeid == ptypeid).Single(parametrit => parametrit.kohteetid == args.KohteetId).arvo;
            return data;
        }
    }
}
```

Kuva 37. AsOyNimi-tiedontoimittajaluokka

Loin 14 valmista tiedontoimittajaa demoa varten, esimerkiksi kuvan 37 AsOyNimi-luokan. Tein tiedontoimittajien tiedonhaun käyttäen LinqToSql-toimintoja ja Data.dbml-mallia (Kuva 38), koska kyseessä oli tässä vaiheessa demo ja tiedonhaun te-

hokkuus ei ole välttämätöntä. Lopulliset tiedontoimittajat tehdään melko varmasti joko hyväksi käyttäen valmiita Sql-palvelimen proseduureja tai ORM-rajapintaa.



Kuva 38. Demon tiedontoimittajien malli Data.dbml

6.4.2 Yksikkö testit

Yksikkötestauksella voidaan testata jokainen kirjaston osa erikseen, kuin myös koko ketjun toiminnot. Olen tehnyt myös nk. tuotantotestauksen, jossa oletetaan että kaikissa tietokannoissa on aina samat testitiedot, jotta toimivuus voidaan testata myös tietokantatasolta asti (Kuva 39).

```
//Proxy
[TestMethod]
public void ProxyEmptyValue()
{
    var testi = new BookmarkDataProviderProxy(string.Empty).GetData(new BookmarkDataProviderArguments() { KohteetId = 1598970 });
    Assert.AreEqual("", testi);
}
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void ProxyGetDataNullValue()
{
    new BookmarkDataProviderProxy("AsOyNimi").GetData(null);
}
[TestMethod]
public void ProxyGetDataAsOyNimi()
{
    var testi = new BookmarkDataProviderProxy("AsOyNimi").GetData(new BookmarkDataProviderArguments() { KohteetId = 1598970, KäyttäjäID = 200, ConnectionString = @"Data Source=MURSU1\TEEMUNSQL;Initial Catalog=Tampuuri_4.0;Integrated Security=True" });
    Assert.AreEqual("As. Oy Matinraitti 14", testi);
}
[TestMethod]
public void ProxyGetDataAsuinHuoneistoAla()
{
    var testi = new BookmarkDataProviderProxy("AsuinhuoneistojenAla").GetData(new BookmarkDataProviderArguments() { KohteetId = 1598970, KäyttäjäID = 200, ConnectionString = @"Data Source=MURSU1\TEEMUNSQL;Initial Catalog=Tampuuri_4.0;Integrated Security=T
    Assert.AreEqual("4299", testi);
}
[TestMethod]
public void ProxyGetDataUnknownValue()
{
    var testi = new BookmarkDataProviderProxy("Tuntematon kirjanmerkki").GetData(new BookmarkDataProviderArguments() { KohteetId = 1598970 });
    Assert.AreEqual(string.Empty, testi);
}
```

Kuva 39. BookmarkDataProviderProxyn yksikkötestit

Tarkoituksena yksikkötesteillä on varmistaa että kirjasto toimii virheettömästi kaikissa odotettavissa olevissa tilanteissa, mukaan lukien tilanteet, joissa järjestelmän tulee antaa virheilmoitus (Kuva 40).

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void ProxyGetDataNullValue()
{
    new BookmarkDataProviderProxy("AsOyNimi").GetData(null);
}
```

Kuva 40. Virhetilanteen testaus BookmarkDataProviderProxy-luokalle

Jos kirjastossa ilmenee virhe, kirjoitetaan ensin testi joka toistaa virhetilanteen ja testi läpäistään virhetilanteessa. Sen jälkeen muokataan ohjelman virheellistä osaa niin, että virhe on korjattu. Seuraavaksi testiä muokataan niin, että se läpäistään, kun ohjelma toimii oikein (Kuva 41).

```
[TestMethod]
public void AsOyNimiEmpty()
{
    var testi = new AsOyNimi();
    Assert.IsNotNull(testi);
}
[TestMethod]
public void AsOyNimi()
{
    var testi = new AsOyNimi().GetData(new BookmarkDataProviderArguments() { KohteetId = 1598970, KäyttäjäID = 200, ConnectionString = @"Data Source=MURSU1\TEEMUNSQL;Initial Catalog=Tampuuri_4.0;Integrated Security=True" });
    Assert.AreEqual("As. Oy Matinraitti 14", testi);
}
```

Kuva 41. Yksikkötestaus AsOyNimi-tiedontoimittajalle

Yksikkötestauksen käytössä etuna on se, että jokaista virhettä vastaan tehty testi jää varmistamaan, että samaa virhettä ei tapahdu uudelleen. Lisäksi muut testit varmistavat, että ohjelman korjaaminen ei ole rikkonut muita ohjelman osioita. Yksikkötestit kannattaa ajaa joka kerta, kun ohjelma käännetään. Näin voidaan varmistua, että uu-

sissa kirjaston versioissa ei ole inhimillisiä virheitä. Rakenteelliset virheet eivät testeissä paljastu. Siksi olisikin parasta että Tdd-kehityksen mukaisesti eri henkilöt tekevät testit ja ohjelman. Mitä hankalammin toteutettavat yksikkötestit, sitä luotettavampi ohjelman koodista tulee.

6.5 Liittäminen Tampuuriin

Jotta työkalu saadaan liitettyä Tampuuriin, tulee ottaa huomioon useita eri asioita. Integrointia ei tässä vaiheessa viedä läpi lopullisesti, eli työkalu on Tampuurissa mutta sitä ei saa käyttöön esimerkiksi kohdepuun valintojen kautta. Itse työkalu ei tarvitse lähtötiedoikseen kuin kohteen tunnuksen, demoa varten lisätty UserHelper-luokka mahdollistaa kaiken muun tiedon, kunhan Tampuuriin on kirjauduttu.

Kun lopullista toteutusta tehdään, tulee työkaluun lisätä käyttöoikeuksien tarkistus. Ei ole suotavaa, että kaikki saavat muokata tai poistaa dokumentteja. Lisäksi Tampuurin asetuksiin lisätään näkyvyysasetus, sillä työkalu ei välttämättä tule kaikille asiakkaille käyttöön.

Kaikki pohjadokumentit ja tallennettavat dokumentit tulee käsitellä dokumenttipankin kautta. Työkalussa ei oteta kantaa siihen mistä käsiteltävä tiedosto tulee, ja on mahdollista, että sille voi olla useampia lähteitä. Työkaluun lisätään todennäköisesti mahdollisuus ladata pohjadokumentti suoraan käyttäjän koneelta dokumenttipankin toimintoja käyttäen tai lataamaan haluttu pohjadokumentti dokumenttipankista. Lähteestä riippumatta dokumenttia käsitellään aina samassa kansiossa, joten työkalun ei tarvitse ottaa kantaa dokumenttien lähteeseen, kunhan dokumentti on kansiossa.

Työkalun ulkoasu muokataan vastaamaan Tampuurin ulkoasua. Tämä ei vaikuta toimintoihin, mutta sivuille saatetaan lisätä Tampuuriin liittyviä painikkeita tai toimintoja käyttöä helpottamaan. Koska Tampuurin käyttöliittymä toimii kehyksessä (frame), sijoitetaan työkalu sisimpään kehykseen niin että yläpalkissa näkyvät Tampuurin yleistoiminnot, ja vasemmassa reunassa moduulin toiminnot ja kohdepuu josta valintoja voi tehdä.

Kaikki työkalun tallentamat tiedot menevät sitä varten tehtyihin tauluihin. Nämä taulut tulee lisätä jokaiseen Tampuurin tietokantaan johon työkalu tulee.

6.6 Demo

Jotta johtoryhmä pystyy tarkastelemaan uutta työkalua tuoteomistajan (product owner) johdolla, tehtiin projektista demo tarkoitusta varten tehdylle sisäiselle palvelimelle testing.tampuuri.fi. Itse asentaminen oli helppoa: kopioin vain käännetyt projektitiedostot oikeaan hakemistoon testipalvelimella. Sen jälkeen lisäsin tarpeelliset taulut testi-tietokantaan tekemälläni sql-skriptillä.

Ongelmia muodostui kuitenkin useita. Testauspalvelin käytti Tampuurin vanhempaa versiota, joten jouduin osin päivittämään sitä ja sen tietokantaa, jotta sain työkalun toimimaan. Jouduin tarkistamaan kaikki työkalun ominaisuudet ja muuttamaan mm. yhteysmerkkijonojen käyttöä, sillä tietokantahaut eivät toimineet.

Demoasennus oli kuitenkin hyvä testi, sillä siinä paljastui vielä muutama pieni virhe, joka olivat jääneet koodiin kehitystyön yhteydessä. Havaitsin että osa koodista toimi oikein vain paikallisesti asentamassani Tampuurissa ja olin unohtanut käyttää toimintoja, joilla käytössä olevasta asennuksesta saadaan oikeat asetukset. Nämä asetukset ovat Tampuuri-kohtaisia ja ne tulevat jokaisen Tampuurin asennuksen web.config-tiedostosta tai tietokannasta.

Sain lopulta demon toimimaan oikein usean tunnin virheenkorjauksen ja päivitysten jälkeen. Demoon oltiin tyytyväisiä ja työkalu toteutetaan vuoden 2013 lopulla julkaistavaan Tampuurin versioon.

7 PÄÄTÄNTÖ

Kun aloin työskennellä projektin parissa, se vaikutti melko yksinkertaiselta ja yksioikoiselta. Todellisuus oli kuitenkin toinen, koska törmäsin jatkuvasti ongelmiin, joihin ei löytynyt yksiselitteistä ratkaisua mistään kirjasta tai internetin palstoilta. Tiedonhaun vaikeutta pahensi internetlähteiden sekavuus ja eritasoisuus. Esimerkiksi Microsoftin msdn-sivuilla on usein hyvä mutta suppea kuvaus ilman kaikkia sovellusmah-

dollisia, ja ne kelpaavatkin lähinnä luokkien ja metodien ominaisuuksien ja syntaksin tarkistukseen niille, joille asia on jo tuttu. Toisaalta muilla sivuilla ongelmat oli hyvin yksityiskohtaisesti ratkaistu, mutta ratkaisut olivat ”häkkejä”(hack) eli epämääräisiä, puutteellisia tai huonosti tehtyjä.

Vaikeaa oli myös oppia koulumaailman esimerkeissä toimivien ”täydellisten” ratkaisujen puuttuminen, ja kompromissien tekeminen ohjelmistokehityksessä. Sain havaita, että asiat voi tehdä monella tavalla ja ohjelmaa kehitettäessä pitää usein valita siihen tilanteeseen sopivin ratkaisu, vaikka sen toteutus olisikin ”ruma”. Tämä on totta ennen kaikkea vanhan nk. legacy-koodin yhteydessä. Usein vaihtoehtona onkin kirjoittaa koko osa uudestaan tai tehdä lyhyt toimiva ratkaisu, joka korjaa ongelman, muttei varsinaisesti ratkaise sitä. Koska minun tehtäväni oli mallin ja toimintojen tekeminen, enkä joutunut tekemään esim. tilinpäätökseen tulevia tiedontoimittajia, en joutunut tekemään suuria kompromissejä, vaan sain toteuttaa luokat puhtaalta pöydältä. Törmäsin kuitenkin yhteensopivuuden haasteisiin jatkuvasti.

Onnistuin löytämään toimivia ratkaisuja suurimpaan osaan kohtaamistani teknisistä ongelmista, ja loppuihin sain apua kollegoilta. Saatuani toiminnot tehtyä ohjelmistoarkkitehtimme romutti käyttämäni mallit ja selitti, mitä heikkouksia koodissani oli, ja miten rakenteesta tehdään toimivampi kokonaisuus myös tulevia muutoksia ja taaksepäin yhteensopivuutta silmällä pitäen.

Voin sanoa että tässä vaiheessa käynnistyi todellinen oppiminen sillä siirryin ohjelmoinnissa tasolle, josta minulla ei ollut aiemmin mitään kokemusta. Uusien mallien sisäistäminen ja uusien ohjelmointitapojen oppiminen oli erittäin työlästä mutta loppujen lopuksi tärkeä prosessi, jonka jälkeen olen saanut paremman ymmärryksen suuren ohjelmiston kehittämisen ongelmista ja ratkaisuista. Jouduin pohtimaan ongelmia syvällisemmin, koska ohjelmiston toiminnassa on useita kerroksia ja useita toiminta- ja virhemahdollisuuksia.

Löysin lopulta yksinkertaisen ja toimivan rakenteen ohjelmalleni. Onnistuin yrityksen ja erehdyksen kautta poistamaan kaiken turhan koodistani, ja tekemään siitä varsin puhdasta ja selkeää. Graafikko muuttaa vielä käyttöliittymän ulkoasua, mutta rakenteeseen oltiin tyytyväisiä. Rakenne noudattaa melko puhtaasti Microsoftin suosimaa

tiedontoimittaja/tiedonkuluttaja-mallia (dataprovider/dataconsumer), joka on nykyisen .NET-ympäristön toimintojen malli.

Ohjelman lopullinen rakenne muodostui vasta loppuvaiheessa. Koska tein työtäni itsenäisesti ja vasta seuraavan kehitysvaiheen valmistuttua sain uusia ohjeita ja malleja käyttööni, oli projekti jatkuvassa muutostilassa. Hyvä puoli tässä oli se että jouduin kokeilemaan eri malleja ja toimintatapoja. Näin opin kantapään kautta eri sovellustapojen heikkouksia, ja missä tilanteessa mikäkin ohjelmointitapa soveltuu parhaiten käytettäväksi. Tein paljon virheitä, ja voikin sanoa, että kirjoitin ohjelman moneen kertaan, mutta virheistä olen oppinut eniten. Virheenkorjaus ja tunnistus ovat mielestäni yksi ohjelmistokehittäjän tärkeimmistä taidoista.

Yksi suurista ongelmista oli myös Tampuurin ymmärtäminen. Koska Tampuuri on suuri ja kauan kehitetty ohjelmisto, on siinä paljon sisältöä jota on kehitetty eri aikoina ja eri tavoin. Tämän vuoksi Tampuuriin kehittäminen vaatii hyvää ohjelmistokehityksen ymmärrystä, sillä sen lisäksi että otetaan huomioon tulevat muutokset ja laajennukset, pitää ottaa huomioon yhteensopivuus nk. legacy-koodin kanssa.

Koin vaikeudesta huolimatta projektin mielekkäänä. Projekti on opettanut minulle paljon ja siten myös helpottanut työtäni Tampuurin kehitystiimissä. Sain lopulta kaikki ongelmat ratkaistua, mistä suuri kiitos työtovereilleni, joilta sain apua tarvitessani sekä ohjausta projektin eri vaiheissa. Minua motivoi tehdä oikeaan käyttöön tuleva osa kehittämäämme ohjelmistoon, ja olenkin kiitollinen työnantajalleni, joka antoi minulle mahdollisuuden kehittää työkalua työssä ja yhtäjaksoisesti. Ilman saamaani tukea työkalu olisi ollut teknisesti aivan erilainen ja varmasti hajanaisempi, nyt olen erittäin tyytyväinen toteutukseen ja kaikkeen oppimaani projektin aikana.

LÄHTEET

Agenteq 2012. WWW-dokumentti. www.agenteq.fi. Ei päivitystietoa. Luettu 13.03.2013.

Aspose Corporation 2013. WWW-dokumentti. www.aspose.com/.net/word-component.aspx. Ei päivitystietoa. Luettu 13.03.2013.

Evjen, Bill, Hanselman, Scott, Rader, Devin 2009. Professional ASP.NET 3.5 SP1 edition: In C# and VB. Wiley publishing Inc: Indianapolis.

Jennings, Roger 2009. Professional ADO.NET 3.5 with Linq and Entity Framework. Wiley publishing Inc: Indianapolis.

Metsker, Steven John 2004. Design patterns in C#. Addison-Wesley: Boston.

MicroSoft Inc 2013. WWW-dokumentti. msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx. Ei päivitystietoa. Luettu 13.03.2013.

MicroSoft Inc 2013. WWW-dokumentti. msdn.microsoft.com/en-us/library/hcw1s69b(v=vs.71).aspx. Ei päivitystietoa. Luettu 13.03.2013.

MicroSoft Inc 2013. WWW-dokumentti. msdn.microsoft.com/en-us/vstudio/hh341490.aspx. Ei päivitystietoa. Luettu 13.03.2013.

MicroSoft Inc 2013. WWW-dokumentti. www.microsoft.com/visualstudio/eng/downloads. Ei päivitystietoa. Luettu 13.03.2013.

Stephens, Ryan K., Plew, Ronald R., Morgan, Bryan & Perkins, Jeff 1999. SQL Tietokantaohjelmointi. IT Press: Helsinki

Tampuuri 2012. WWW-dokumentti. www.tampuuri.fi. Ei päivitystietoa. Luettu 13.03.2013.

Troelsen, Andrew 2012. Pro C# 5.0 and the .NET 4.5 Framework. Apress: New York.

Vieira, Robert 2009. Microsoft SQL-server 2008 programming. Wiley publishing Inc: Indianapolis.