Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Pekka Salo

XML-tiedon vastaanotto ja tallennus tietokantaan

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

19.11.2019

Tekijä Otsikko Sivumäärä Aika	Pekka Salo XML-tiedon vastaanotto ja tallennus tietokantaan 31 sivua 19.11.2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	tietotekniikka
Ammatillinen pääaine	ohjelmistotuotanto
Ohjaajat	Software Architect Tuomas Viskari Yliopettaja Auvo Häkkinen

Opinnäytetyön tarkoituksena oli luoda järjestelmä, joka lukee sisään käyttäjän lähettämää XML-muotoista tietoa, validoida se, ja jos tieto on oikeanmukaista, tallentaa se tietokantaan. Opinnäytetyössä kerrotaan, miten järjestelmä toteutettiin, miksi nämä toteutustavat valittiin ja mitä ongelmia kehityksessä oli.

Työ tehtiin Digian eräälle asiakkaalle. Koska työn tarkat yksityiskohdat ovat luottamuksellisia, on työssä kuvattu tehtyjä ratkaisuja yleisemmällä tasolla. Tavoitteena työssä oli luoda edellä kuvattu järjestelmä ja päivittää se asiakkaan vanhan tiedon sisäänlukujärjestelmän tilalle.

Työ tehtiin pääasiassa kesällä 2019, mutta jatkokehitys jatkuu vielä tulevaisuuteen.

Työn lopputuloksena oli järjestelmä, joka vastaa asiakkaan vaatimuksia. Järjestelmä tulee käyttöön, kun sen muiden osien päivitys on valmis ja asiakas on saanut testattua järjestelmän kokonaisuudessaan.

Avainsanat	XML, tietokanta, tiedonsiirto

Author Title Number of Pages Date	Pekka Salo Receiving and saving XML-data to a database 31 pages 19.11.2019
Degree	Bachelor of Engineering
Degree Programme	Degree Programme in Information Technology
Professional Major	Software engineering
Instructors	Tuomas Viskari, Software Architect Auvo Häkkinen, Principal lecturer

The purpose of the Thesis is to create a program that reads in XML data sent to it by users, validate it, and, if the data is valid, save in in a database. The thesis explores how the system was implemented, why it was implemented that way and what problems there where during the development.

The work was made for one of Digia's customers. Because some of the details of the system are confidential, this thesis explores the system in a more general level. The goal of the project was to create the above system and use it to replace the customers old information input system.

The work was done mainly in the summer of 2019, but some development is still ongoing.

The result of the project was a system that fulfilled the customers' requirements. The system will be deployed to production once the other parts of the project are completed and the customer has tested and approved the whole.

Keywords	XML, database, data transfer

Sisällys

Lyhenteet

DTD	Document Type Definition. Yksi merkintäkielistä, jolla kuvataan XML-dokumentin rakennetta.
HTML	Hypertext Markup Language. Merkintäkieli, jolla kuvataan tiedon rakennetta ja joka on yleisimmin käytetty kieli verkkosivujen rakenteen kuvaamiseen.
LINQ	Language-Integrated Query. .NET-komponentti, jolla voi tehdä datakyselyjä .NET-ympäristössä.
SGML	Standard Generalized Markup Language. Merkintäkieli, jolla kuvataan tiedon rakennetta.
SQL	Structured Query Language. Kyselykieli, jolla voi hakea ja manipuloida tietokannoista.
W3C	World Wide Web Consortium. Kansainvälinen standardisointiorganisaatio, joka luo ja ylläpitää internetin standardeja.
XHTML	eXtensible HyperText Markup Language. Merkintäkieli, jolla kuvataan tiedon rakennetta. Käytetään verkkosivujen rakenteen kuvaamiseen.
XML	Extensible Markup Language. Standardi merkintäkieli, jolla kuvataan tiedon rakennetta.
XSD	XML Schema Definition. Yksi merkintäkielistä, jolla kuvataan XML-dokumentin rakennetta.

1 Johdanto

Opinnäytetyön tavoitteena on rakentaa ohjelma, jonka avulla voi lukea XML-muotoista dataa ja tallentaa se tietokantaan. Työn on tilannut Digia Oyj eräälle heidän asiakkaallensa.

Digia Oyj on ohjelmisto- ja palveluyritys, joka työllistää yli 1200 ihmistä Suomessa ja muissa Pohjoismaissa. Digia Oyj toimittaa palveluja monille eri toimialoille, joista tärkeimmät ovat finanssi- ja julkinen sektori. Finanssialalla Digia Oyj on tunnettu DIFS-tuotteestaan (Digia Financial Services), joka on yksi Pohjoismaiden laajimmista finanssialan järjestelmäkokonaisuuksista. Julkisen sektorin puolella Digia Oyj toteutti verohallinnon tulorekisterin sovelluskokonaisuuden [1].

Digia Oyj muodostui kun 1990 perustettu SysOpenin ja 1997 perustettu Digia Oy yhdistyivät vuonna 2005. Tässä vaiheessa yrityksen nimi oli SysOpen Digia Oyj, joka kuitenkin lyhentyi vuonna 2008 vain Digia Oyjksi. Tämän jälkeen Digia Oyj on ostanut monia yrityksiä ja integroinut heidän osaamisensa, ja laajentanut heidän ammattiosaamistaan. Vuonna 2015 Digia Oyj päätti jakauttaa Qt-liiketoiminnan omaksi yrityksekseen ja tämä rekisteröitiin omaksi yritykseksi vuonna 2016. Tämän työn tavoitteena on päivittää vanha vastaava dataa sisäänlukeva ominaisuus uuteen version. Tässä aikaisemmin käytetty teknologia on vanhentunutta, epätarkkaa ja vaikeasti päivitettävää, joten siitä on päätetty tehdä uusi versio.

Työn alussa käydään lyhyesti läpi pääteknologiat, joita työssä on käytetty. Nämä oli määritelty jo ennen kuin opinnäytetyön tekeminen aloitettiin, joten niihin ei opinnäytetyössä voitu vaikuttaa.

Opinnäytetyö pyrkii pääasiassa selvittämään ainoastaan yhden tavan ratkaista ongelma. Tosin joissain kohdin esitetään muitakin vaihtoehtoja. Tavoitteena ei ole esittää yhtä parasta tapaa tehdä tämänkaltainen ohjelma, vaan esittää tapa, jolla minä ratkaisin ongelmat, jotka tulivat vastaan työn ja ajan puitteissa.

Opinnäytetyö sisältää kuvaukset käytetyistä teknologioista sekä järjestelmästä. Lisäksi siinä kerrotaan, miksi työssä tehtiin mitäkin ratkaisuja sekä mitä ongelmia työssä tuli vastaan.

2 Käytettävät teknologiat

Ohjelman toteutuksessa käytetään useita teknologioita. Näistä XML (Extensible Markup Language) ja XSD (XML Schema Definition) liittyvät sisään lukuun, ja SQL (Structured Query Language) liittyy tiedon tallennukseen. Itse ohjelma on toteutettu käyttämällä C#-kieltä ja .NET Frameworkin versiota 4.6.2. Näitä teknologioita käytetään työssä, koska ne ovat joko asiakkaan määrittämiä, kuten XML ja XSD, tai jo olemassa olevassa järjestelmässä käytettyjä teknologioita, joita käytetään järjestelmän yhteensopivuuden takia, kuten SQL Serveriä, .NET Framework 4.6.2:ta.

2.1 Tiedon siirto ohjelmalle

Tiedonsiirrossa ohjelmalle käytetään XML-tiedostoja. Ohjelma lukee sisään XML-muotoista dataa, tarkastaa sen sisällön ja tekee sille joitain muutoksia ennen kuin se tallennetaan tietokantaan.

XML on W3C:n (World Wide Web Consortium) kehittämä tiedon metakieli, joka perustuu 1986 vuonna julkaistuun ISO 8879-standardin SGML-kieleen (Standard Generalized Markup Language). XML on SGML:n osajoukko, ja se on paljon tarkemmin rajattu kieli kuin SMGL. Tämä rajaus tekee XML-dokumenttien lukemisesta ja parsimisesta helpompaa. XML-kieli lisää useita tarkennuksia SGML-kielen luoman pohjan päälle, kuten esimerkiksi sen, että myös elementin lopputagin tulee sisältää elementin nimi.

XML-kieli ei itse kuvaa tiedon sisältöä, vaan se määrittelee sen sisältävän tiedon rakenteen ja muodon. Tämä tekee XML:stä erinomaisen kielen tiedon välitykseen. Niin kauan kuin XML-tiedosto on luotu oikein ja vastaanottaja tietää tiedoston rakenteen, kuka tai mikä tahansa voi lukea dokumentin sisällön tiedon oikeassa muodossa. Koska XML on hierarkkinen kieli (esimerkki 1), se myös mahdollistaa yksi-yhteen ja yksi-moneen yhteyksien teon tietojen välille.

```
<?XML version="1.0" encoding="utf-8"?>
<Document XMLns:xs="http://www.w3.org/2001/XMLSchema" XMLns="urn:esimerkki">
  <Tyonantaja TyonantajanNimi="Esimerkki Oy" Maa="FI" Kieli="sv" YTun-
nus="1231234-1">
     <Tyontekijat>
       <Tyontekija Henkilotunnus="151299-111B" Sukunimi="Esimerkki" Etuni-
met="Mikko Matti"/>
       <Tyontekija Henkilotunnus="151299-111A" Sukunimi="Mallikas" Etuni-
met="Maija Malla"/>
     </Tyontekijat>
  </Tyonantaja>
</Document>
```

Esimerkkikoodi 1. XML-tiedosto, jossa on työnantajan ja työntekijän tiedot.

XML-dokumentti alkaa määrittämällä käytettävä XML-versio ja käytettävä merkkikoodaus. Jos dokumentti halutaan validoida jotain validointidokumenttia vastaan, tämä tulee myös määrittää dokumentin alussa. Mahdollisia validointikieliä ovat esimerkiksi DTD (Document Type Definition) ja XMD-skeema. Tässä työssä on käytetty XSD-skeemaa, joten siitä on lisää seuraavassa luvussa.

XML-dokumentin tieto on jaettu hierarkkisiin elementteihin. Tämä hierarkia on puurakenteinen. Tämä tarkoittaa, että kaikki elementit, paitsi ylin juuri-elementti, ovat suoraan vain yhden toisen elementin alla. Yhden elementin sisällä voi olla monta elementtiä, joita kutsutaan elementin lapsielementeiksi. XML ei salli ympyrämäisiä suhteita elementtien välillä, eli elementti ei voi olla hierarkiassa itseään alempana olevan elementin alla.

XML kehitettiin tiedon siirtoon internetin yli eri järjestelmien välillä. W3C:n tarkoituksena oli luoda yksinkertainen, mutta monipuolinen merkintäkieli, joka on sekä ihmisten että koneiden ymmärrettävissä. Tämä yksinkertaisuus ja ymmärrettävyys myös mahdollistaa sekä helpottaa ohjelmien kirjoittamista, jotka prosessoivat XML-kieleen perustuvia dokumentteja. Tämä on myös auttanut tekemään XML-kielestä yhden suosituimmista tiedonvälityskielistä.

Esimerkki XML-kieleen perustuvasta teknologiasta on XHTML (eXtensible HyperText Markup Language). XHTML, kuten pelkkä HTML (Hypertext Markup Language), on verkkosivujen rakennetta kuvaava kieli. Erona näissä on se, että XHTML on XML:n tapaan tarkempi formatoinnista. Esimerkkejä tästä tarkemmasta formatoinnista XML:stä ja siihen perustuvissa kielistä ovat, että kaikki elementit pitää sulkea, ja attribuuteille pitää antaa arvot.

2.2 Tiedon rakenteen määrittely

XSD:tä käytetään työssä määrittämään XML-datan rakenne. Jotta dataa voi käsitellä, sen pitää olla jossain ennalta sovitussa muodossa. Datan rakenteen tunteminen mahdollistaa sen tietojen lukemisen muuttujiin ja hierarkkisiin yhteyksiin. Jos rakennetta ei ole, määritelty tiedoston sisältö on vain tekstiä, josta tietoa ei saa otettua ulos.

XSD on W3C:n vuonna 2001 kehittämä standardi, jolla voidaan määrittää XML-dokumentin rakenne. Sillä voidaan varmistaa, noudattaako dokumentti XSD-tiedostossa määriteltyjä sääntöjä ja onko dokumentti validi XSL-skeemaa vastaan.

XSD-dokumentti voi määrittää XML-dokumentissa käytettävien elementtien ja attribuuttien nimet, järjestyksen, muodon sekä tyypin (esimerkkikoodi 2). Toisin kuin muut skeemakielet, XSD suunniteltiin siten, että validointi tuottaisi tietoa dokumentin oikeellisuudesta, jota sovelluksen kehittäjä voisi käyttää hyväkseen luodessaan ohjelmia.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="urn:esimerkki" xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:esimerkki" elementFormDefault="qualified">
    <xs:simpleType name="Maatunnus">
        <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z0-9]{2,2}"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="Kielikoodi">
        <xs:restriction base="xs:string">
            <xs:enumeration value="FI"/>
            <xs:enumeration value="SV"/>
            <xs:enumeration value="EN"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="Henkilotunnus">
        <xs:restriction base="xs:string">
            <xs:pattern value=" [0-9]6}[A+-][0-9]{3}[0-9A-FHJ-NPR-Y]"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:element name="Document">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Tyonantajat"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="Tyonantajat">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Tyonantaja" minOccurs="0" maxOccurs="un-
bounded"/>
            </xs:sequence>
        </xs:complexType>
```

```
    </xs:element>
    <xs:element name="Tyonantaja">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Tyontekijat" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="Nimi" type="xs:string" use="required"/>
            <xs:attribute name="YTunnus" type="xs:string" use="required"/>
            <xs:attribute name="Maa" type="Maatunnus" use="required"/>
            <xs:attribute name="Kieli" type="Kielikoodi" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="Tyontekijat">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Tyontekija" minOccurs="0" maxOccurs="un-
bounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="Tyontekija">
        <xs:complexType>
            <xs:attribute name="Etunimet" type="xs:string" use="required"/>
            <xs:attribute name="Sukunimi" type="xs:string" use="required"/>
            <xs:attribute name="Henkilotunnus" type="Maatunnus" use="re-
quired"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Esimerkkikoodi 2. Esimerkki XSD-tiedostosta, joka sisältää peruselementit, joita työssä käytetään

Esimerkkikoodi 2 sisältää monia XSD-dokumentissa yleisimmin käytettäviä määrityksiä. XSD-skeematiedostot käyttävät XML-syntaksia, joten sitä käyttääkseen ei tarvitse opetella täysin uutta kieltä, vaan voi käyttää hyväksi jo olemassa olevaa tietoa. Tämä myös mahdollistaa muiden XML-kieltä hyväksi käyttävien ominaisuuksien käytön, kuten esimerkiksi XSLT-muutosten tekemisen.

Juurielementissä julistetaan nimiavaruus, joka tavanomaisesti merkitään ‘xsd:’-nimiseksi. Tämä merkitään tavanmukaisesti ‘xsd:’-nimiseksi, mutta se ei ole pakollista, vaan käyttäjä voi nimetä sen, miksi haluaa. Tämä julistaa dokumentin olevan XSD-dokumentti ja antaa siinä olevat ominaisuudet käyttöön, Samaa ‘xsd:’ -merkintää käytetään sekä elementtien että XSD-skeeman sisäänrakennettujen tyyppien edessä. Tämä merkitsee ne osaksi XSD-spesifikaatiota, eikä osaksi käyttäjän omia määrittelyjä

XSD sisältää joitakin sisäänrakennettuja tietotyyppejä, kuten ‘string’, ‘long’ ja ‘decimal’, jotka edustavat merkkijonoa, kokonaislukua ja desimaalilukua. Nämä yksinkertaiset tietotyyppielementit eivät voi sisältää tietoa suoraan XSD-elementin sisällä, eikä niissä voi

olla attribuutteja. Nämä sisäänrakennetut yksinkertaiset tietotyypit luovat pohjan XSD-validoinnille, mutta niillä ei voi tehdä kovin monimutkaisia validointeja. Jos tarvitaan yksinkertaisia tietotyyppejä monimutkaisempia validointeja joillekin kentille, niin pitää käyttää monimutkaisempia tietotyyppejä.

Tietotyyppimääritykset voivat sisältää tietoja kuten esimerkiksi merkkijonon pituuden, merkkijonon mahdolliset arvot, tai säännöllisen lausekkeen (regular expression, regex), jonka mukainen annetun merkkijonon pitää olla.

Tietotyypit voivat myös olla monimutkaisempia (complex types) ja pitää sisällään elementtejä sekä attribuutteja. Näille attribuuteille määritetään joku tietotyyppi, joko jokin sisäänrakennetuista tietotyypeistä tai itse määritelty tietotyyppi. Attribuutit voi myös määrittää pakollisiksi käyttämällä ‘required’-attribuuttia. Kuten esimerkkikoodissa 2, elementteihin voi määrittää, kuinka monta sellaista pitäisi olla nykyisen alla käyttämällä minOccur-s ja maxOccurs-attribuutteja. ‘MinOccurs’ määrittää, kuinka monta kertaa elementin pitää vähintään sisältyä sen emon sisälle, ‘maxOccurs’ taas kertoo suurimman mahdollisen määrän näitä määriteltyjä elementtejä. Jos jompaa kumpaa tai molempia näistä ei ole määritelty, niin niillä**,** joita ei ole määritelty, on oletusarvona 1**.** Elementit ovat samoja ‘complexType’-elementtejä, joten ne voivat myös sisältää muitakin elementtejä sekä attribuutteja. Näin määrittämällä ‘complexType’-elementtejä, joissa on sisällä muita elementtejä, voi määrittää haluamansa XML-dokumentin rakenteen.

2.3 Tiedon siirto tietokantaan

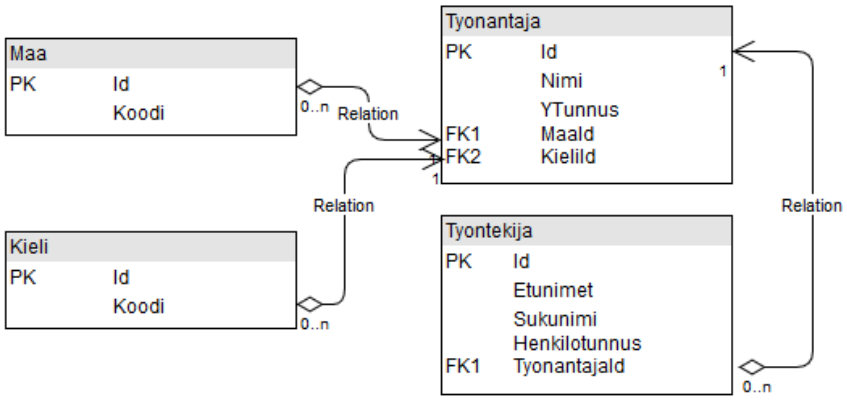
SQL:ää käytetään työssä tallentamaan XML:llä välitetty data myöhempää tarkastelua ja käsittelyä varten.

SQL on yleinen relaatiokantojen kyselykieli, joka tarkoittaa sitä, että sillä voi lähettää erilaisia kyselyjä tietokantaan, joilla voi joko hakea, muuttaa tai poistaa tietoa, tai muokata tietokannan rakennetta. Relaatiokannassa olevissa tauluissa voi olla relaatioita (eli linkkejä) muihin tauluihin.

Jokaisessa taulussa pitää olla pääavain. Tämän arvon pitää olla uniikki taulussa, eli jos taulussa pääavainkentän arvoksi on määritelty 9876, niin ei taulun minkään muun rivin

pääavainkentässä voi olla arvona 9876. Eri taulut voivat kuitenkin sisältää samanarvoisen pääavaimen. Sekä tauluissa A ja B olevissa riveissä voi pääavaimen arvo olla 9875. Pääavaimeksi voi valita minkä tahansa taulun kentän, kunhan se on oikeasti ainutlaatuinen. Hyvä tapa on tehdä tauluun Id-kenttä, johon laitetaan numeroita peräkkäin, eli ensimmäisessä rivissä on arvo 1, seuraavassa 2 ja niin edelleen. Yleisesti katsottuna kaikissa relaatiotietokannoissa on mahdollista määrittää kenttä, johon tietokanta laittaa kasvavan numeron automaattisesti tietoa tauluun lisättäessä. Etu tämän käytössä on se, että käyttäjän ei tarvitse huolehtia pääavaimen arvosta vaan tietokanta hoitaa sen hänen puolestaan. Toinen etu tässä tavassa määritellä pääavain on, että tällä arvolla ei ole merkitystä, joten sellaista tilannetta, jossa arvo pitää muuttaa, ei tule vastaan. Jos pääavaimeksi määrittää esimerkiksi käyttäjänimen tai sähköpostiosoitteen, niin mitä tehdään, jos se halutaan vaihtaa? Se on mahdollista vaihtaa, mutta se saattaa vaatia monimutkaisia tietokantaoperaatioita. Merkityksettömän numeron käyttö taas on helpompaa, koska sen arvolla ei määrityksen mukaan ole mitään väliä. Tällaisten peräkkäisten avainten etsiminen on myös tietokannasta nopeaa ja helppoa.

Taulun kentän voi myös määrittää vierasavaimeksi. Tämä tarkoittaa sitä, että siinä oleva arvo osoittaa jonkun tietokantaulun kenttään. Siihen mihin kenttään taulu osoittaa on tietokannassa merkintä. Vierasavaimeksi voi valita minkä tahansa kentän, kunhan se on aidosti ainutlaatuinen koko taulun kentässä [4]. Yleensä vierasavaimeksi valitaan toisen taulun Id-pääavain -kenttä (kuva 1), jos tämä on automaattisesti kasvava kokonaisluku, niin tiedon etsiminen on myös nopeaa ja helppoa.



Kuva 1. Kuvassa näkyy yksinkertainen esimerkkikaavio, jossa työntekijällä on yhteys työnantajaan, ja työnantajalla on yhteys sekä maahan että kieleen.

Vierasavainta ei tarvitse välttämättä käyttää, vaan jokainen taulu voi olla täysin erillään toisistaan. Jos näin haluaa tehdä, kannattaa valita jokin NoSQL-tietokanta, sillä relaatiokannat on rakennettu käyttämään relaatioita. Tämä tarkoittaa sitä, että tietokannassa olevalla tiedolla voi olla yhteyksiä muuhun tietokannan tietoon. Esimerkiksi tietokannassa olevalla työntekijätaululla voi olla linkki työnantajatauluun. Relaatioita käyttämällä on mahdollista normalisoida tietokanta.

Tietokannan rakennetta voi yksinkertaistaa normalisoimalla tietokannan taulut. Tämä tarkoittaa, että tieto jaetaan pienimpiin loogisiin kokonaisuuksiin, joissa taulussa olevat tiedot liittyvät toisiinsa. Esimerkiksi henkilöön liittyvät tiedot ovat kaikki yhdessä taulussa. Näitä kokonaisuuksia yhdistetään suuremmiksi kokonaisuuksiksi käyttäen vierasavaimia. Kuvassa 2 työntekijä ja työnantaja ovat erillisiä tauluja, jotka on kytketty toisiinsa työntekijätaulun TyonantajaId-kentän avulla (esimerkkikoodi 3).

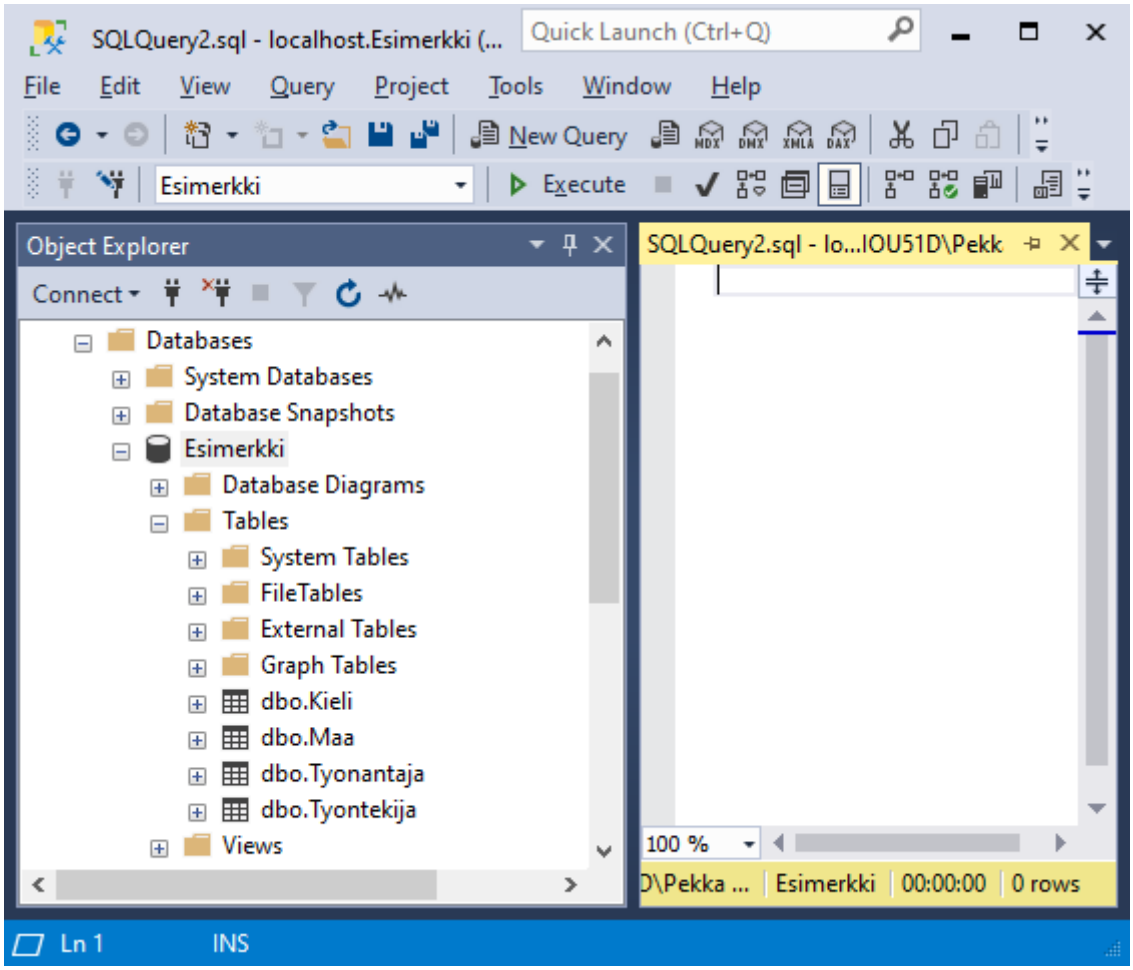
```
CREATE TABLE Tyontekija (Id INT PRIMARY KEY,
    Etunimet VARCHAR(50),
    Sukunimi VARCHAR(50),
    Henkilotunnus CHAR(11),
    TyonantajaId INT FOREIGN KEY REFERENCES Tyonantaja(Id)
);
```

Esimerkkikoodi 3. Työntekijätaulun luontilause, huomaa, että siihen on merkitty vierasavaimeksi jo olemassa oleva Työnantajataulun Id-kenttä.

Olisi mahdollista aina toistaa työnantajatiedot jokaisen työntekijän kohdalla, mutta tämä veisi turhaa tilaa ja tekisi työnantajan tietojen päivittämisestä hankalaa. Työnantajan tiedot pitäisi päivittää jokaisessa siihen kuuluvassa työntekijäkentässä. Jos taas ne ovat eri tauluissa ja ne on yhdistetty vierasavaimella, työnantajan tiedot pitää päivittää vain yhdessä paikassa, ja ne päivittyvät automaatiisesti kaikille työntekijöille.

Joissain tapauksissa tietojen yhdessä pitäminen on kuitenkin parempi vaihtoehto. Tilanteissa**,** joissa tietokannasta haetaan usein suuria määriä tietoa, saattaa olla parempi toistaa nämä tiedot. Tämä säästää tietokannalta aikaa, sillä sen ei tarvitse tehdä monimutkaisia hakuja ja yhdistelyjä tiedoille. Tässä työssä kuitenkin käytettiin jo olemassa olevaa normaalia relaatiokantaa, joten tällaista tietokantaratkaisua ei tutkittu enempää.

SQL-tietokantoja on olemassa monia erilaisia. Tässä työssä kuitenkin käytettiin jo olemassa olevaa SQL Server-tietokantaa. SQL Server on Microsoftin tekemä SQL-tietokantahallintaratkaisu. Se on monipuolinen ja kattava ratkaisu, joka sopii hyvin työssä oleviin tarpeisiin. Se myös integroituu hyvin muihin työssä käytettyihin työkaluihin. SQL-palvelimessa on monta versiota, siitä on ilmainen versio, kehitysversio sekä kaupallinen versio. Työssä on käytetty Developer Edition -kehitysversiota (kuva 2), sillä tähän opinnäytetyöhön ei vaadittu mitään laajempaa ratkaisua.



Kuva 2. SQL Management Studio-ikkuna, jossa on auki esimerkkitietokanta.

Työssä SQL Management Studiota käytettiin tietokannassa olevan tiedon tarkasteluun. Tietoja katsottiin kehityksessä, jotta nähtiin, olivatko tiedot siirtyneet tietokantaan onnistuneesti ja mitä tietoa tietokantaan oli siirtynyt. Työssä tuli monta kertaa vastaan tilanteita, joissa tiedon siirrossa tietokantaan oli ongelmia. Joissain tapauksissa jokin tiedoista ei tallentunut ollenkaan ja joskus se taas tallentui väärin.

2.4 C# ja .NET

Työ toteutettiin korvaamaan osa jo olemassa olevaa .NET Framework -ohjelmaa. Vaikka se olisi ollut teoriassa mahdollista toteuttaa käyttäen jotain muuta kieltä, niin koko ohjelmiston ylläpidon kannalta päätettiin käyttää samaa teknologiaa kuin muissakin ohjelmiston osissa.

C# on Microsoftin kehittämä ohjelmointikieli. Vuonna 1988 Microsoft kehitti omaa versiota C-kielestä. Tätä projektia ei kuitenkaan koskaan viety loppuun asti. On kuitenkin selvää, että C-kieli vaikutti nykyisen C#-kielen version määrittelyssä. Toinen kieli, jolla oli vaikutusta C#-kielen kehityksessä, oli vuonna 1994 julkaistu Java-kieli. Kun C# -kieli julkaistiin ensimmäisen kerran vuonna 2000, se ja Java olivat erittäin samanlaisia. Kielien kehityssuunnat ovat kuitenkin alkaneet erota toisistaan enemmän vuodesta 2005, jolloin C#-kielestä julkaistiin versio 2. Esimerkkikoodissa 4 on esimerkki yksinkertaisesta C#-ohjelmasta.

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

Esimerkkikoodi 4. Ohjelma, joka tulostaa konsoliin tekstin “Hello, world!”

C# on vahvasti oliopohjainen ohjelmointikieli. Kaikki C#-metodit kuuluvat aina johonkin luokkaan. Luokat voivat olla normaaleja luokkia, joista voi luoda uuden olion, tai staattisia luokkia, joiden kaikki metodit ovat myös staattisia. Staattinen metodi on metodi, jota kutsutaan luokan nimen kautta, ei jonkin sen ilmentymän kautta. Staattisia metodeja ei ole kuitenkaan sidottu vain staattisiin luokkiin vaan normaalit luokat voivat myös sisältää niitä. Luokan merkitseminen staattiseksi ei siis tee staattisten metodien luomisesta mahdollista, vaan se estää ei staattisten metodien luomisen.

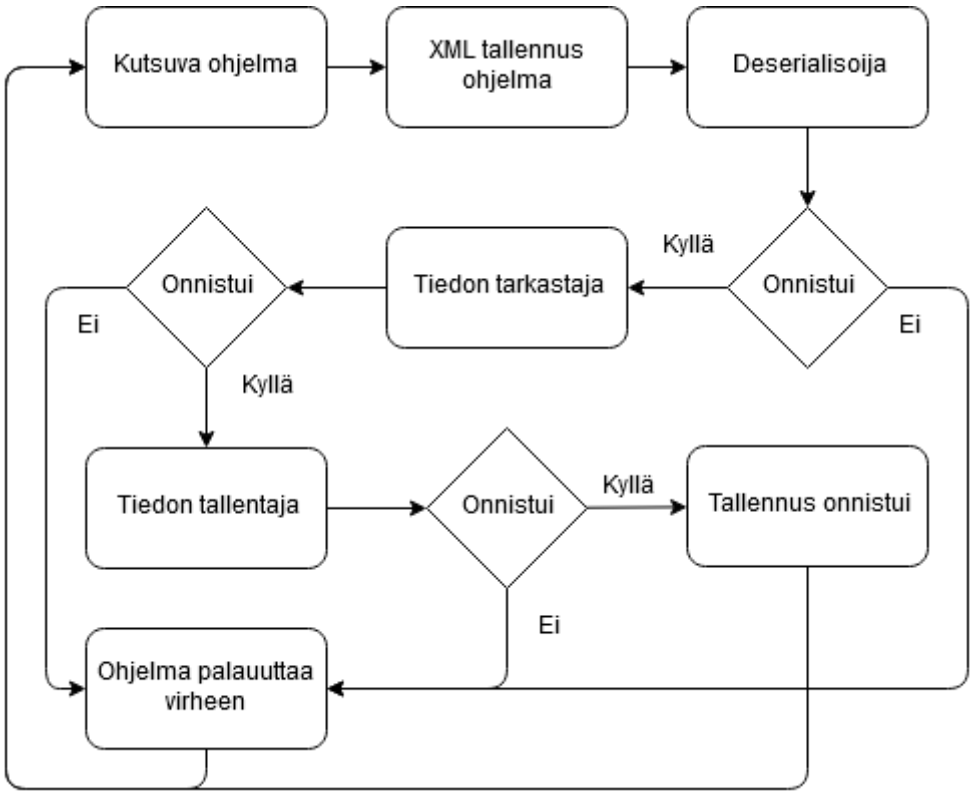
.NET on Microsoftin kehittämä ohjelmointialusta, joka on suunniteltu mahdollistamaan monien eri tyyppisten ohjelmien tekemisen. .NET-ohjelmia käytetään pääasiassa Windows-ympäristössä, mutta sitä voi käyttää myös muissakin ympäristöissä, vaikka se ei niihin välttämättä aina parhaiten sovellu.

Työssä on käytetty .NET-alustaa luomaan verkkosovellus. .NET-alustoja on monia eri versioita, kuten .NET Framework ja .NET Core. Nykyään, jos aloittaa uuden verkkosovelluksen luonnin, niin suositellaan käyttämään .NET Corea. Se on uudempi versio, jonka Microsoft on kehittänyt korvaamaan .NET Frameworkin. Vaikka Microsoft on muokannut monia osia ja kirjastoja niin, että ne sopivat yhteen .NET Coren kanssa, on vielä olemassa paljon kirjastoja, jotka toimivat vain .NET Frameworkillä. Monet vanhemmat Microsoftin ulkopuoliset kirjastot myös toimivat vain .NET Frameworkillä. Nykyään uusien kirjastojen kirjoittamiseen käytetään .NET Standardia, jos mahdollista. Tällä kirjoitettuja kirjastoja voi käyttää molemman tyyppisissä projekteissa, kunhan projektin versio on tarpeeksi uusi.

.NET Framework toimii Microsoft Windows -alustalla. Se julkaistiin alun perin vuonna 2002. Tämän jälkeen se on saanut useita päivityksiä, ja uusin versio työtä kirjoitettaessa oli 4.8. Tässä työssä käytetty versio on 4.6.2, koska se oli versio, jolla muut osat verkkosovelluksesta oli tehty.

3 Kuvaus järjestelmästä

Ohjelman suoritus on jaettu eri vaiheisiin (kuva 3). Vaiheistus auttaa pitämään ohjelman rakenteen selvänä, ja se jakaa eri asioiden teon omiin luokkiin. Tämä jako oli hyvä idea, koska työssä piti palata takaisin olemassa olevaan koodiin useamman kerran. Koodiin piti palata joko muuttamaan tai korjaamaan joko olemassa olevaa koodia, tai lisäämään siihen jotain tarvittavaa. Koska ohjelma oli suunniteltu tekemään asiat eri vaiheissa, oli tällaisissa tilanteissa helppo löytää paikka, jonne muutokset piti tehdä.



Kuva 3. Ohjelman eri osien yksinkertaistettu suoritusjärjestys.

Kuvassa 3 on yleistasolla kuvattu ohjelman toiminta, vaiheet ja niiden järjestys. Ohjelman suoritus alkaa ajamalla tarkoitukseen tehty konsoliohjelma, joka kutsuu tallennusohjelmaa sille annetuilla parametreilla

Tallennusohjelman suoritus alkaa, kun luokasta luodaan olio, jolle annetaan tarvittavat tiedot kuten XSD-skeema. Kun olio on luotu, kutsutaan olion deserialisointimetodia, ja sille annetaan parametreina tiedosto ja sen nimi.

Heti ohjelman suorituksen alussa tallennetaan kopio lähetetystä tiedostosta väliaikaiseen hakemistoon, josta se myöhemmin siirretään joko onnistuneille tai epäonnistuneille ajoille tarkoitetun hakemistoon. Se mihin kansioon tiedosto siirretään, riippuu, tallennettiinko tiedot onnistuneesti tietokantaan vai ei. Tiedosto tallennetaan ohjelman ajon yhteydessä sille annetulla nimellä, ja sen perään lisätään ajon päivämäärä ja aika muodossa ’yyyyMMddTHHmmssZ’. Eli esimerkiksi tallennetun tiedoston nimi voisi olla ‘esimerkki-20190102T121314.xml’, jossa 2019 on vuosi 01 on kuukausi 02 on päivä, 12 on tunnit, 13 on minuutit ja 14 on sekunnit.

Aika lisätään tiedoston perään, koska se mahdollistaa tarvittaessa saman tiedoston lähettämisen useaan kertaan. Yksi syy, jonka takia saman niminen tiedosto voidaan haluta lähettää useamman kerran on, jos tiedoston tallennuksessa tuli virheitä, ja lähettäjä on korjannut ne. Erillisten tiedostojen olemassaolo helpottaa tulevaisuudessa ohjelmassa mahdollisesti ilmenneiden ongelmien tutkimisessa.

Tiedostosta tallennetaan kopio ajon yhteydessä, koska aiemmin asiakkaalla tuli vastaan ongelma tiedoston tallennuksessa, ja lähettäjä oli jo poistanut oman kopion tiedostosta. Ongelman ratkaisu oli hankalaa ja työlästä, joten he päättivät, että uudessa versiossa sisäänlukuohjelmasta tallennetaan aina varmuuden vuoksi aluksi kopio tiedostosta.

Seuraavaksi tiedosto lähetetään deserialisoijalle joka luo XML-tiedostosta C#-luokat XSD-skeeman perusteella. Tässä vaiheessa voi tiedostossa tulla vastaan virheitä, jotka deseriailisoija ottaa kiinni ja lähettää takasin sisäänlukuohjelmalle.

Seuraavassa vaiheessa sisäänluetulle tiedolle tehdään vielä joitain tarkastuksia. Tässä vaiheessa tehdyt tarkastukset ovat sellaisia, joita ei skeemassa voitu määrittää, ja deserialisoijassa tehdä. Esimerkiksi tarkastukset, kuten onko työnantaja, jonka tietoja muutetaan jo olemassa tietokannassa vai ei, tai onko työntekijä merkitty olemaan töissä sille työnantajalle, jonka alla se on lähetetyssä XML-tiedostossa. Tässä käydään kaikki tiedot läpi, vaikka vastaan tulisi virheitä. Jos virheitä on, niin ne lähetetään kaikki sisäänlukuohjelmalle.

Viimeisessä vaiheessa tiedot siirretään tietokantaan. Tässä vaiheessa tiedot siirretään XSD-skeemaa vastaavista luokista tietokantatauluja vastaaviin luokkiin ja joko päivitetään jo olemassa olevat tiedot, tai tarvittaessa luodaan uudet. Kun tiedot on siirretty tietokantaluokkiin, ne yritetään tallentaa tietokantaan. Jos tallennuksessa tulee vastaan ongelmia, niin tallennus perutaan ja sisäänlukuohjelmalle ilmoitetaan virheestä.

Tilanteessa, jos missään vaiheista ei tullut virheitä, on tiedot luettu XML-tiedostosta, ne on tarkastettu ja tallennettu tietokantaan. Seuraavaksi alussa välisikaiseen kansioon tallennettu tiedosto siirretään kansioon, joka on tarkoitettu onnistuneesti tallennetuille tiedostoille. Kun nämä toiminnot on tehty, niin ohjelman suoritus päättyy onnistuneesti.

Jos taas jossain tiedoston käsittelyvaiheista tuli vastaan ongelma, niin alussa tallennettu tiedosto siirretään virhehakemistoon. Seuraavaksi ohjelman suorituksessa tulleet virheet kootaan yhteen ja tarvittaessa muotoillaan ihmisten luettavaan muotoon. Kun virheet on koottu, niin luokka heittää poikkeuksen, joka sisältää 'Dictionary':n joka sisältää virheet sitä kutsuvalle ohjelmalle.

Konsoliohjelmassa virheet ottaa vastaan konsoliohjelma ja se tulostaa ne konsoli-ikkunaan käyttäjän tarkasteltaviksi. Lopullisessa järjestelmässä tiedostot tulevat verkkokäyttöliittymän kautta. Jos tiedoston tallennuksessa on virheitä, niin käyttäjälle näytetään virheilmoitus, ja sen alla tulleet virheviestit lueteltuna.

4 Tiedostojen luonti

4.1 XSD-skeeman luonti

XSD-skeemaa luotaessa pitää ensin määrittää, millaista XML-dataa haluaa lukea tietokantaan. Tässä määritetään sisäänluettavan tiedon rakenne, nimet ja tietotyypit. XSD-skeemaa tarvitaan sekä saadun XML-datan oikeanmukaisuuden tarkistamiseen että skeemaa vastaavien luokkien luontiin.

Skeeman luokkia luodessa kannattaa miettiä tarkasti, missä muodossa tietoja haluaa pyytää. XSD mahdollistaa monessa tapauksessa jo suoraan oikean tietotyypin vaatimisen, kuten esimerkiksi double, long tai date. Näissä tapauksissa tietotyypit sisältävät validoinnin jo sisäänluvun yhteydessä, ja niille osataan määritellä koodissa oikea muoto. Joissain tapauksissa se tekee tiettyjen validointien tekemisestä vaikeaa. Esimerkiksi, jos halutaan ottaa jokin reaaliluku kahden desimaalin tarkkuudella, ei voi varmistaa, onko annettu numero oikean muotoinen. Tässä tapauksessa arvo on otettava vastaan muodossa string ja validointi on tehtävä joko skeemassa säännöllisten lausekkeiden avulla tai erikseen koodissa.

Skeemassa voi määrittää rajoituksia annetuille arvoille. Nämä rajoitukset voivat olla long-arvon rajoittaminen joidenkin arvojen välille, string-arvon rajoittaminen vain joihinkin vaihtoehtoihin, tai string-arvon rajoittaminen säännöllisen lausekkeen perusteella.

Säännölliset lausekkeet mahdollistavat hieman monimutkaisempien validointien tekemisen, esimerkiksi aiemmin mainitun kahden desimaalin tarkkuudella olevan numeron tarkastuksen luomisen. Huono puoli tässä esimerkkitapauksessa on, että arvo on string-muotoinen eikä numero. Tämä tarkoittaa, että se pitää muuttaa jossain vaiheessa oikeaan muotoon ennen kuin sen voi tallentaa tietokantaan. Jos tieto on oikeassa muodossa, jonka voi ratkaista säännöllisellä lausekkeella, on tämä muutos kuitenkin helppo tehdä, joten se ei tuota suurta ongelmaa.

Yksi ongelma, joka työssä tuli vastaan tiedoston sisäänluvussa, oli, että .NET:in skeeman validointi ei luotettavasti tehnyt säännöllisiin lausekkeisiin perustuvia validointeja, vaan nämä piti tehdä myöhemmässä vaiheessa manuaalisesti. Tämä ei kuitenkaan tullut työssä suureksi ongelmaksi. Projekti vaati joka tapauksessa validointeja, jotka oli mahdotonta tehdä XSD-skeemassa. Näiden validointien joukkoon pystyi samalla sisältämään tarvittavat säännollisillä lauseilla tehtävät validoinnit.

Oikea ongelma, joka tässä saattaa tulla jossain vaiheessa vastaan on**,** että jos säännöllisillä lausekkeilla toteutettuja validointeja pitää joskus päivittää, se pitää muistaa tehdä molemmissa paikoissa. Työssä jouduttiin näitä muutoksia tekemään muutaman kerran, joka aiheutti virheitä, joita oli välillä vaikea löytää.

XSD-tiedosto pitää sisällyttää jotenkin projektiin, jos sitä vastaan haluaa validoida tarkemmin sisäänluettavia tiedostoja. Työssä XSD-tiedosto päätettiin pitää projektin ulkopuolisessa kansiossa ja lukea se sieltä tarvittaessa. Jos tiedosto olisi sisällytetty projektin tiedostoihin, niin työssä olisi voitu olla varmoja, että siitä olisi ollut oikea versio käytettävissä.

4.2 Skeemaa vastaavat luokat

Ennen kuin XML-tiedostossa oleva tieto saadaan vietyä tietokantaan, se luetaan kahteen erilaiseen C#-luokkaan. Ensimmäiset näistä ovat skeemaluokkia, joka vastaavat XSD-skeemassa olevia tietoja. Toiset ovat tietokannassa olevia tauluja vastaavat luokat. Jos tietoa haluaa siirtää moneen eri tauluun, tieto pitää siirtää kaikkiin niitä vastaaviin luokkiin Vaihtoehtoisesti tiedon voi myös lukea tietokantaan tekemällä tietokantaan proseduurit jotka tekevät tallennuksen, ja kutsumalla näitä koodissa.

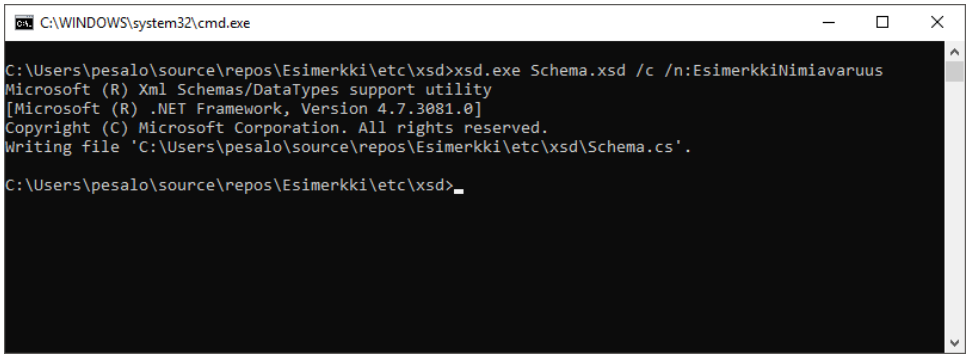
Molemmat näistä luokkatyypeistä luodaan käyttämällä Microsoftin tarjoamia työkaluja. Jos nämä luokat tehtäisiin käsin, niin se olisi monimutkaista ja aikaa vievää. Lisäksi jos skeemaan tai tietokantaan tulee muutoksia, joutuisi ne tekemään luokkiin käsin, Jos taas luokat luo käyttäen näitä työkaluja, ne voidaan aina luoda uudestaan, kun skeemaan tai tietokantaan tulee muutoksia. Luokkien luominen työkaluilla on myös erittäin nopeaa. Molemmat ohjelmat loivat tarvittavat luokat muutamassa sekunnissa.

Ensimmäisenä pitää luoda skeemaa vastaavat luokat, joihin XML-muotoinen tieto luetaan myöhempää käsittelyä varten. Luokat luodaan XSD-skeeman perusteella, sillä sen tarkoituksena on kuvata tiedon rakenteen malli. Tämä tarkoittaa, että siinä on varmasti kaikki tieto luokista, joita ohjelma tarvitsee.

Helpoin tapa luoda XSD-skeema vastaavat luokat on ajaa luotu skeema Xsd.exe-työkalun läpi. Ohjelma varmistaa, että luokat ovat yhteensopivia .NET-työkalujen kanssa ja niiden käyttö on helppoa.

Xsd.exe on Visual Studion mukana tuleva työkalu, jolla olemassa olevista XSD-tiedostoista voidaan luoda niitä vastaavat .cs-luokat [6]. Ohjelmalla luodut luokat vastaavat XSD-skeeman määrittelyä rakenteeltaan, nimiltään ja tietotyypeiltään. Luokkien nimiksi tulee XSD:n elementin nimet, ja sen muuttujiksi elementin attribuutit. Myös elementin sisällä olevat elementit luetaan muuttujiksi, jotka ovat joko yksittäisiä ohjelman generoimia luokkamuuttujia tai taulukko näitä luokkia. Nämä muuttujat mahdollistavat skeemassa olevien yhteyksien säilytyksen ja niiden välisen navigoinnin, kun dataa siirrytään käsittelemään koodissa.

Ohjelma ajetaan avaamalla Visual Studio Developer Command Prompt -sovellus ja navigoimalla kansioon, jossa XSD-tiedosto on ja ajamalla xsd.exe -ohjelma komennolla: ‘xsd [polku *.xsd tiedostolle] /c /n:[nimiavaruus]’ kuvassa neljä olevalla tavalla. Tämä luo *.cs-tiedoston samaan kansioon, jossa XSD-tiedosto on. Tiedosto kannattaa viedä johonkin projektin kansioon, jotta siihen on helppo pääsy projektista ja sen voi sisällyttää projektiin luokkatiedostona.



Kuva 4. Kuva xsd.exe-ohjelman ajamisesta, komennolla luodaan Schema.cs-niminen luokka, jossa on skeemaa vastaavat luokat ‘EsimerkkiNimiavaruus’ nimiavaruuden sisällä.

Huomaa komennossa oleva /n:[nimiavaruus]. Tämä tekee niin, että xsd.exen luomat luokat tulevat automaattisesti komennossa määritellyn nimiavaruuden sisälle. Jos aiotaan nimetä XML-elementit samaan tapaan kuin tietokannassa olevat taulut, niin tämä kannattaa määrittää. Nimiavaruus auttaa välttämään nimiristiriitoja käytettyjen luokkien välillä, joka helpottaa niiden käyttöä koodissa. Seuraavassa osassa käytetty työkalu generoi luokkien nimet tietokantataulujen nimistä. Jos ei ole varovainen syntyy helposti luokkia, joiden nimillä on ristiriitoja jo olemassa olevien luokkien kanssa. Vaihtoehtoisesti voidaan myös varmistaa, että XSD-elementtien nimet ovat eri kuin taulujen nimet. Nimiavaruus kannattaa kuitenkin määritellä tulevaisuutta ajatellen. On mahdollista, että tulevaisuudessa XSD-skeemaan ja tietokantaan tulee saman nimisiä elementtejä ja tauluja, jotka generoisivat saman nimisiä luokkia. Projektin organisointi on myös helpompaa, jos skeema ja tietokantaluokat ovat eri nimiavaruuksissa.

Kun XSD-tiedostoon tekee muutoksia, pitää muistaa aina generoida nämä luokat uudestaan. Jos yrittää lukea uutta skeemaa vastaavaa dataa vanhan skeeman vastaaviin luokkiin, tässä voi tulla virheitä. Jos sisään luettaessa käytetään XSD-skeeman validointia, nämä virheet tulevat helposti esille.

Tämä ongelma tuli vastaan usean kerran työtä tehdessä, koska skeemaa jouduttiin muuttamaan muutaman kerran. Vanhat skeemaversiot aiheuttivat ongelmia myös versionhallinnassa. Jos jollain haaralla tekee töitä eikä siellä ollut käytettyä skeemaversiota vastaavia luokkia, niin tiedoston sisäänluvussa tuli virheitä.

4.3 Tietokantaluokat

Tietokantaluokkia käytetään tiedon tallentamiseen tietokantaan. Tiedon voisi tallentaa tietokantaan myös ilman näitä luokkia, mutta niiden olemassaolo helpottaa prosessia huomattavasti. Ilman tietokantaluokkia tietojen tarkastelu ja tallennus pitäisi tehdä manuaalisesti SQL-kutsuilla, kun taas tietokantaluokilla .NET huolehtii näiden kutsujen rakentamisesta ohjelmoijan puolesta.

Tietokantaluokat ovat luokkia, jotka vastaavat rakenteeltaan tietokannassa olevia tauluja. Luokat helpottavat tietokannassa olevan tiedon hakua ja käsittelyä tekemällä loogisen luokkarakenteen, jonka avulla yhteyksiä on helpompi hahmottaa. Näitä luokkia käyttäen on helppo lisätä uusia tietoja tietokantaan sekä hakea, päivittää ja poistaa jo olemassa olevia tietoja. Tiedon käsittelyyn käytetään .NET:n LINQ (Language-Integrated Query) to SQL -toimintoa. Tällä SQL-kutsut, kuten tiedon haku tai päivitys, voidaan kirjoittaa C#-kielellä, jonka .NET muuttaa SQL-kutsuiksi, kun koodi suoritetaan [7].

Luokat ovat SqlMetal-nimisen työkalun luomia. Työkalun tarkoituksena on generoida C#, tai jonkin muun tuetun ohjelmointikielen luokat [8]. Se tekee tämän lukemalla ensin tietokannassa olevien taulujen rakenteet sekä yhteydet ja rakentamalla niitä vastaavat luokat. SqlMetal.exe voi tietokantataulujen lisäksi myös generoida sekä esivalmisteltuja proseduureja että näkymiä vastaavat luokat. Generoiduille luokille voi myös määritellä nimiavaruuden, jonka sisään luodut luokat laitetaan. Luokat voi myös asettaa toteuttamaan yhteisen rajapinnan. Tässä kannattaa olla varoivainen, sillä jos rajapintaan määrittää jonkin ominaisuusfunktion, niin tietokannan kaikissa tauluissa pitää olla sen niminen kenttä. Ainoa ominaisuusfunktio joka tietokannan rajapintaan kannattaa määrittää on Id-kentän getteri.

Generoidut luokat on merkitty partialeiksi. Tämä tarkoittaa, että luokan määrittelyn ja metodit voi kirjoittaa moneen eri tiedostoon. Täten jos XSD-skeemaa luokkia luodessa ei määritelty nimiavaruutta, on mahdollista, että SqlMetal-ohjelman luomat luokat sekoittuvat skeemaluokkiin. Jos näin tapahtuu, pitää luokat erottaa toisistaan, joka on helpointa tehdä laittamalla ne eri nimiavaruuksiin.

Luokkien generoinnissa voi tulla vastaan ongelmia. Ainakin tilanteissa, joissa tarvittavia kenttiä ei ole merkitty vierasavaimeksi. Tällaisia ongelmia on mahdollista ratkaista luomalla tai muokkaamalla luokkia käsin. Ongelma luokkien muokkaamisessa on, että kun tietokannan rakennetta muuttaa, niin uudet muutokset joudutaan myös lisäämään luokkiin käsin. Jos taas virheet korjaa tietokantaan, voi luokat generoida uudelleen niin, että niitä ei tarvitse muokata käsin. Tässä on se hyvä puoli, että ei vahingossa menetä tekemiään muokkauksia joka kerta, kun luokat pitää luoda uudestaan.

SqlMetal.exe merkitsee generoidut luokat osittaisiksi. Tämä tarkoittaa, että luokan määrityksiä voi olla monissa eri tiedostoissa. Tällaisissa tapauksissa kaikki luokat, joilla on sama nimi ja jotka on merkitty osittaisiksi, ovat kääntäjän mielestä osa samaa luokkaa. Tämä mahdollistaa uusien metodien lisäämiset, jotka esimerkiksi hakevat, formatoivat tai muokkaavat tietoa ennalta määritetyillä tavoilla. Tämä on erittäin hyödyllinen ominaisuus varsinkin tällaisessa tapauksessa, jossa luokat saatetaan generoida moneen kertaan. Jos lisäykset tehtäisiin suoraan generoituihin luokkiin, niin SqlMetal.exe ylikirjoittaisi ne joka kerta, kun se ajetaan. Jos taas muutokset tekee tiedoston ulkopuoliseen osittaiseksi merkittyyn luokkaan, silloin ne säilyvät ennallaan (esimerkkikoodi 5).

```
public partial class Tyontekija {
      public string Kokonimi {
            get {
                  return this.Etunimet + " " + this.Sukunimet;
            }
      }
}
```

Esimerkkikoodi 5. Lisätään työntekijää esittävään olioon getteri, joka palauttaa henkilön koko nimen hänen etu- ja sukunimen perusteella.

Getterit ovat joissain tapauksissa käytännöllisiä, sillä niitä voi käyttää kuten normaaleja muuttujia. Niissä on myös se etu, että niihin voi metodien tapaan määrittää rajoituksia, kuka niitä voi kutsua. Getterin voi esimerkiksi määrittää yksityiseksi tai suojatuksi, jos sitä ei haluta muuttaa luokan tai siitä perivien luokkien ulkopuolella. Näissä on myös se etu normaaleihin parametreihin verrattuna, että sen arvo voi muuttua dynaamisesti riippuen olion tilasta. Esimerkki tästä on esimerkkikoodissa 5, jossa ‘Kokonimi’-arvo riippuu ‘Etunimet’- ja ‘Sukunimet’-arvoista.

5 Tiedon käsittely

5.1 Tiedon sisäänluku

Kun kaikki tarvittava luokat on luotu, niin on aika lukea data niihin sisään. Tiedon sisäänlukemiseen pitää tehdä ohjelma, joka ottaa vastaan XML-tiedostot (Esimerkkikoodi 6) ja haluttaessa myös XSD-tiedostot. Tämä ohjelma voi olla mikä tahansa manuaalisesti ajettavasta konsoliohjelmasta verkkosivuun, jolla tiedosto lähetetään. Pääasia on, että tiedosto saadaan luettua sisään.

```
<?xml version="1.0" encoding="utf-8"?>
<Document xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="urn:esimerkki">
    <Tyonantajat>
        <Tyonantaja Nimi="Esimerkki työnantaja" YTunnus="1234567-8" Maa="fi"
Kieli="FI">
            <Tyontekijat>
                <Tyontekija Etunimet="Erkki" Sukunimi="Esimerkki" Henkilotun-
nus="010101-0101" />
            </Tyontekijat>
        </Tyonantaja>
    </Tyonantajat>
</Document>
```

Esimerkkikoodi 6. Skeeman mukaan validi XML-tiedosto, johon on laitettu kaikki elementit ja niille attribuutit. Skeema mahdollistaa myös Työnantajalla olevan monta Työntekijää, sekä dokumentissa olevan useita Työnantajia.

Kun tiedosto on luettu sisään, se pitää muuttaa muotoon, jossa .NET voi validoida sen skeemaa vastaan ja lukea sen skeemaluokkiin. .NET XML-sisäänluku haluaa ‘Stream’-tyyppisen muuttujan, jossa on tiedoston sisältö. Joissain sisäänlukutavoissa, kuten tiedon lukeminen verkkosivuilta, tieto saattaa olla jo ‘Stream’-tyyppisessä muuttujassa, mutta joskus se ei ole näin helppoa. Paras vaihtoehto näissä tapauksissa on muuttaa tiedoston sisältö ensin bittitauluksi, jonka voi myöhemmin lukea sisään ‘MemoryStream’-tyyppiseen olioon.

.NET sisältää tarvittavat luokat XML-tiedoston validonitiin ja deserialisointiin, joten niitä kannattaa käyttää hyväksi. Deserialisointi tarkoittaa tässä tiedon purkamista tekstimuotoisesta tiedostosta skeemaa vastaaviin luokkiin, jotta niitä voidaan käsitellä koodissa. Validoinnissa on myös mahdollista lisätä oma skeematiedosto oletustarkastusten lisäksi. Tätä kannattaa hyödyntää, jos tiedossa on joitain yksinkertaisia testejä, joita halutaan tehdä.

On monta tapaa deserialisoida data skeemaa vastaaviin luokkiin, mutta jos halutaan käyttää myös XSD-skeemaa validoinnissa, vaihtoehdot vähenevät. Työssä valittiin käytettäväksi ‘XMLReader’-luokka, joka ottaa vastaan ‘XMLReaderSettings’-olion. Tähän olioon voi lisätä XSD-skeeman.

Kannattaa ottaa huomioon, ettei .NET-skeeman validointi ole täydellistä, joten siihen ei kannata täysin luottaa. Työssä yritettiin käyttää kahta eri .NETin sisäänrakennettua tapaa validoida XML-tiedostoa skeemaa vastaan, ja molemmissa oli ongelmia. Ensimmäinen tapa, joka lopulta valittiin, ei tehnyt säännöllisten lauseiden mukaisia validointeja eikä toinen suostunut suorittamaan validointeja XSD-skeemoja vastaan, joissa oli ‘targetNamespace’-attribuutti. Työssä valittiin käytettäväksi ensimmäinen tapa, koska siinä oli pakko tehdä joka tapauksessa lisävalidointeja. Säännöllisten lausekkeiden validointien toteuttaminen muiden validointien ohella ei tuottanut paljoa ylimääräistä työtä. Tässä pitää kuitenkin muistaa, että jos skeemassa validoinnit muuttuvat, ne pitää korjata myös koodissa. Tämä monessa paikassa validointien teko saattaa tulevaisuudessa johtaa ongelmiin.

Kun ‘XMLReader’-luokkaan on lisätty skeema ja XML-tiedosto, sen voi yrittää deserialisoida skeemaluokkiin**.**

Jos deserialisoinnissa tulee virheitä, ‘XMLSerializer’-luokka heittää poikkeuksen. Sen sisäinen viesti sisältää tietoa siitä, mikä virhe tuli vastaan ja missä kohdassa tiedostoa se oli. Nämä virheet ovat englannin kielellä**,** ja ne joudutaan joko näyttämään ne englanniksi**,** tai parsimaan niistä tarvittavat tiedot ja kirjoittamaan ne uudelleen tarvittavalle kielelle. Virheen rivi ja sen alkukohta rivillä on kuitenkin helposti saatavissa poikkeuksesta, joten geneerisen viestin, joka kertoo virheen kohdan, on helppo toteuttaa. Tällaista tilannetta ei todellisuudessa pitäisi kovin usein tulla käytännössä vastaan.

XML-dokumentit todennäköisesti tuottaa joku ohjelma asiakkaan päässä heidän tietokannassaan olevasta tiedosta. Jos ongelma tulee kerran vastaan, niin asiakkaan tulee korjata se itse. Asiakkaille on myös lähetetty XSD-skeematiedosto, joten heillä on kaikki valmiudet tehdä oma skeeman validointi tiedostolle ennen sen lähettämistä.

5.2 Tiedon validointi

Kun data on luettu skeemaluokkiin, on aika tehdä mahdolliset tarkemmat validoinnit ja lukea tieto tietokantaluokkiin.

Vaikka tiedosto olisi mennyt läpi skeeman validoinnista, niin se ei vielä tarkoita, että kaikki tiedostossa oleva data on oikein. On hyvin mahdollista, että lähetty data on muodollisesti oikeaa, mutta sen sisältö ei ole odotetunlaista. Esimerkiksi on mahdollista, että henkilötunnus vaikuttaa olevan oikean muotoinen (esimerkkikoodi 7), mutta se ole validi suomalainen henkilötunnus [9].

```
^[0-9]6}[A+-][0-9]{3}[0-9A-FHJ-NPR-Y]$
```

Esimerkkikoodi 7. Säännöllinen lauseke, joka validoi suomalaisen henkilötunnuksen oikeanmuotoisuuden, mutta ei esimerkiksi pysty laskemaan, onko viimeinen tarkastusmerkki oikein.

Tässä vaiheessa tiedoston sisältö on luettuna skeemaluokkiin. Nämä ovat normaaleja C#-luokkia, joten niille on tässä vaiheessa helppo tehdä muut mahdollisesti tarvittavat tarkastukset. Esimerkki mahdollisesta tarkastuksesta, jonka voi tehdä vasta tässä vaiheessa, on jos tietokannan tietoja pitää päivittää, niin päivitettävä olio löytyy tietokannasta.

Tässä vaiheessa data on luettuna skeemaluokkiin, jotka ovat normaaleja C#-luokkia, joten on hyvä aika tehdä mahdolliset validoinnit, joita skeemassa ei saatu tehtyä. Esimerkki tällaisesta tarkastuksesta on tilanne, jossa tiedon oikeellisuus pitää tarkastaa tietokannasta. Jos mahdollinen annettava tieto voi muuttua usein, voi olla helpompaa antaa käyttäjän laittaa kentään mitä tahansa ja validoida arvo tietokantaa vastaan, kun määritellä sallitut arvot ja päivittää ne aina skeemaan.

XSD-skeemassa voi tehdä monimutkaisiakin validointeja, mutta kaikki yksinkertaisia säännöllisiä lauseita monimutkaisemmat validoinnit kannattaa hoitaa koodissa. Tässä on se hyvä puoli, että jos validoinneissa on jokin virhe, sen voi korjata vain päivittämällä koodi. Jos ulkopuolisille tahoille on jo lähetetty esimerkkitiedostot ja skeemat, niin niitä ei tällöin tarvitse päivittää.

Koska skeemaa vastaavat luokat ovat XML-tiedoston mukaisessa puumaisessa järjestyksessä, on joitain eri tapoja tehdä tarkastukset.

Ensin työssä päätettiin tehdä validoinnit samanlaisella puumaisella rakenteella kuin XML-tiedostossa. Jokaiselle validoitavalle luokalle tehtiin oma metodi, joka tekee luokalle kaikki sen vaatimat validoinnit. Kun luokalle on tehty validoinnit ja luokalla on muuttujina muita luokkia, joille pitää kutsua tarkastusmetodeja, kutsutaan näitä luokan sisällä ennen virheiden palauttamista. Tällä tavalla kaikki tiedoston tiedot käydään läpi samassa järjestyksessä, missä ne on laitettu XML-tiedostossa. Tämä mahdollistaa mahdollisten virheiden palauttamisen käyttäjälle samassa järjestyksessä, jossa ne ovat lähetetyssä tiedostossa.

```
public void Deserialisoi(Document dokumentti)
{
    List<int> virheet = new List<int>();
    /* Virheiden tarkastus */
    foreach (Tyonantaja tyonantaja in dokumentti.Tyonantajat)
        virheet.AddRange(TarkastaTyonantaja(tyonantaja));
    if (virheet.Count > 0)
        throw new TarkastajaException(virheet);
}

        private List<int> TarkastaTyonantaja(Tyonantaja tyonantaja)
{
    List<int> virheet = new List<int>();
    /* Virheiden tarkastus */
    foreach (Tyontekija tyontekija in tyonantaja.Tyontekijat)
        virheet.AddRange(TarkastaTyontekija(tyontekija));
    return virheet;
}

private List<int> TarkastaTyontekija(Tyontekija tyontekija)
{
    List<int> virheet = new List<int>();
    /* Virheiden tarkastus */
    return virheet;
}
```

Esimerkkikoodi 8. Esimerkki tarkastusrakenteesta, jossa oliot kutsuvat niille kuuluvien olioiden tarkastusmetodeja.

Myöhemmin tarkastusmetodeja muutettiin niin, etteivät ne enää kutsu luokan skeemaa vastaavilla muuttujilla niille tarkoitettuja metodeja. Luokat tekevät vain omat tarkastuksensa ja palauttavat mahdolliset virheet kutsujalle. Tarkastusmetodin kutsuja kutsuu luokan skeemassa alla olevia oliota vastaavia tarkastusmetodeja. Tämä jatkuu, kunnes kaikki tiedot on validoitu.

```
public List<int> TarkastaDokumentti(Document dokumentti)
{
    List<int> virheet = new List<int>();
    foreach (Tyonantaja tyonantaja in dokumentti.Tyonantajat)
    {
        virheet.AddRange(TarkastaTyonantaja(tyonantaja));
        foreach (Tyontekija tyontekija in tyonantaja.Tyontekijat)
            virheet.AddRange(TarkastaTyontekija(tyontekija));
    }
    return virheet;
}

private List<int> TarkastaTyonantaja(Tyonantaja tyonantaja)
{
    List<int> virheet = new List<int>();
    /* Virheiden tarkastus */
    return virheet;
}

private List<int> TarkastaTyontekija(Tyontekija tyontekija)
{
    List<int> virheet = new List<int>();
    /* Virheiden tarkastus */
    return virheet;
}
```

Esimerkkikoodi 9. Esimerkkitarkastus rakenteesta, jossa tarkastusmetodeja kutsutaan keskitetysti.

Etu tässä tavassa tehdä tarkastukset on, että kaikki tarkastusmetodikutsut ovat samassa paikassa. Tällöin jos jotain validoinnissa pitää muuttaa, ei tarvitse etsiä, missä niitä kutsutaan. Koska metodeja kuitenkin kutsutaan samassa järjestyksessä kuin aiemmassa menetelmässä, virheet tulevat vieläkin vastaan oikeassa järjestyksessä.

Jos validoinnissa tulee virheitä, ne lisätään tarkoitusta varten luotuun listaan, jonka metodi palauttaa. Virhelista ei sisällä itse virheviestejä, vaan int-muotoisia muuttujia, joilla myöhemmässä vaiheessa haetaan tarvittavat virheviestit. Tämä tehdään tällä tavalla, koska se mahdollistaa virheviestien näyttämisen eri kielillä helpommaksi. Tässä tapauksessa viestit haetaan tietokannasta niitä yksilöivällä numero- ja kielivalinnalla. Palautetut virhelistat yhdistetään, ja lopussa heitetään poikkeuksena kutsuvalle ohjelmalle näyttää ne käyttäjälle.

Jos tiedoston tiedoissa ei löydy tässä vaiheessa virheitä, voidaan olla melko varmoja, että tietojen tallennus onnistuu.

5.3 Tiedon siirto tietokantaan

Datan siirto skeemaluokista tietokantaluokkiin on yksinkertaista, koska tässä vaiheessa datan pitäisi olla validoitu niin, että se on oikeanlaista ja se on oikeassa muodossa. Jos tässä vaiheessa tulee vastaan ongelmia, kannattaa harkita joko XSD-skeematiedoston ja validoinnin muuttamista. Koska data on jo oikeassa muodossa, suurin jäljellä oleva työ on siirtää muuttujien arvoja paikasta toiseen, ja tarvittaessa muuttaa niiden muotoja. On esimerkiksi mahdollista, että jokin numerotyyppinen muuttuja luettiin string-tyyppisenä, koska sille haluttiin tehdä joitain ylimääräisiä tarkistuksia. Tällaisten muuttujien arvot pitää muuttaa tietokannan haluamaan muotoon.

Tässä vaiheessa kannattaa harkita halutaanko tietokantaan aina lisätä uutta tietoa, kun uusi tiedosto tulee, vai halutaanko mahdollisesti siellä jo olemassa olevaa tietoa päivittää. Jos olemassa olevaa dataa halutaan päivittää, pitää tietokannassa tarkistaa, onko siellä jo samalla avaimella olemassa olevaa tietoa. Ongelmana on, että kaikella tiedolla ei välttämättä ole käyttäjän tietämää ja lähettämää avainta. Jos tällainen avain on olemassa, niin tiedon päivittäminen on helppoa. Pitää vain hakea avaimella tietokannasta vanhat tiedot ja päivittää ne. Jos taas tällaista avainta ei ole, niin tietokannasta on vaikeaa hakea oikean rivin tietoja päivitettäväksi.

Yksi mahdollisuus tilanteessa, jossa tiedolle ei ole käyttäjän tietämää avainta, on tallentaa tiedosta aina uudet versiot, toinen taas on poistaa aikaisemmat tiedot ja luoda niiden tilalle uudet ja kolmas on lisätä kaikille elementeille yksilöivä tieto. Kaikissa näistä on omat hyvät ja huonot puolensa.

Ensimmäisessä tapauksessa ongelmana on, jos käyttäjä yrittää tallentaa samaa tietoa useamman kerran, samasta tiedosta tallentuu tietokantaan useita kopioita. Tämän ongelman voi estää tarkastamalla, onko tietokannan taulussa rivi, jossa on täysin samat tiedot kuin mitä yritetään tallentaa. Tämä ratkaisu kuitenkin luo ongelmatilanteen siinä tapauksessa, jos saman tiedon lisääminen on validi operaatio. Tämän tapauksen toteuttaminen kuitenkin tarkoittaa, että tällaisen tiedon päivittäminen myöhemmin ei ole mahdollista.

Toisen vaihtoehdon pitäisi aina päivittää tiedot onnistuneesti, kun ohjelma ajetaan, ja se ei jätä tietokantaan ylimääräisiä rivejä. Rivien poistamisessa saattaa tulla vastaan ongelmia, jos niiden tiedoissa on yhteyksiä muihin tietokannan tauluihin. Jo olemassa olevaa tietoa ei myöskään voi päivittää, voidaan vain tuhota kaikki vanha tieto ja luoda ne uudestaan.

Kolmas vaihtoehto on vaatia skeemassa, että kaikilla tiedoilla on ne yksilöivä tieto. Ongelma on, että tällaiseksi tiedoksi ei aina ole luonnollista vaihtoehtoa. Joissain tapauksissa joutuisi lisäämään keinotekoisen kentän. Jos on vaatimuksena päivittää tietokannassa oleva tieto, eikä luoda uutta sen tilalle, niin tämä saattaa olla paras tai jopa ainoa vaihtoehto.

Mikään näistä vaihtoehdoista ei ole täydellinen, ja ne sopivat eri tilanteisiin eri tavalla. Tässä projektissa päätettiin käyttää hybridiä ensimmäisestä ja toisesta tavasta. Ohjelmassa pidetään vanha data tietokannassa ja merkitään se poistetuiksi, tämän jälkeen uusi data lisätään tietokantaan vanhan lisäksi. Rajoitteena kuitenkin on, että jo tietokannassa olevaa tietoa ei poisteta tai ylikirjoiteta. Tämä toteutettiin niin, että kun tieto tallennetaan tietokantaan, niin siellä on taulu, jossa on lähetetyn tiedoston nimi, ja muilla tiedoilla on linkki tähän tauluun. Tällä tavalla kaikki tiedostossa lähetetyt tiedot ovat helposti löydettävissä tietokannasta, ja ne voi merkitä poistetuiksi.

Tieto siirretään tietokantaluokkiin yksinkertaisesti kopioimalla jo oikeassa muodossa olevat tiedot suoraan skeemasta tietokantaluokissa oleviin kenttiin. Joissain kohdissa saattaa olla, että joudutaan hakemaan referenssejä muihin tauluihin, mikä vaatii tietokantayhteyden avaamista ja tiedon hakua. Monissa kohdissa on mahdollista, että data ei ole oikeassa muodossa vietäväksi tietokantaan. Jos esimerkiksi haluttiin tarkistaa reaaliluvussa desimaalien määrä, tämä on jouduttu lukemaan string-muodossa. Nyt tämä pitää muuttaa double-muotoiseksi tietokantaan viemistä varten. Toinen ongelma on luetellut-arvot (enum-arvot). Jos XSD-skeemassa näihin annetuissa arvoissa on välilyönti, niin sitä ei tule näkymään luetelman nimessä, vaan se pitää hakea ‘XMLEnumAttribute’ attribuutista (Esimerkkikoodi 10).

```
public static class EnumExtension
{
    /// <exception cref="ArgumentException"></exception>
    public static string GetXMLEnumAttributeOrName(this Enum value)
    {
```

```
        return (value.GetType().GetMember(value.ToString())?
                .FirstOrDefault()?
                .GetCustomAttributes(typeof(System.XML.Serializa-
tion.XMLEnumAttribute), false)?
                .FirstOrDefault() as System.XML.Serialization.XMLEnumAttrib-
ute)?.Name
                ?? Enum.GetName(value.GetType(), value);
    }
}
```

Esimerkkikoodi 10. Koodissa tehty extension-metodi, joka hakee luetelman nimen

Tässä koodissa käytetään reflektiofunktiota tarkastamaan, onko luetelmalle määritelty 'XMLEnumAttribute'-attribuutti, ja jos on, niin palauttaa sen arvon. Siinä tapauksessa, että attribuuttia ei ole määritelty, koodi palauttaa luetelman arvon nimen. Tällä koodilla saadaan varmasti oikea arvo skeemaluokan luetelmasta olettaen, että metodia käytetään extension-metodina.

Reflektio on .NETissä samannimisessä nimiavaruudessa olevia funktiota, joilla voidaan tarkastella ladattuja ohjelmia ja sen osia. Se mahdollistaa muuten mahdottoman tiedon haun ja käsittelyn.

Extension metodit ovat staattisten luokkien staattisia metodeja, jotka ottavat ensimmäisenä, this-sanalla merkittynä parametrina jonkin tyypin. Tämä tyyppi voi olla perustyyppi, kuten int tai float, tai jokin luokka. Extension-metodeja voi kutsua joko normaalin staattisen metodin kutsun mukaan, tai this parametrina välitetyn luokan kautta. Yllä olevaa metodia voi kutsua myös jonkin lajitellun arvon kautta. Jos on esimerkiksi laitettu arvo muuttuja 'Maa', niin metodia voi kutsua Maa.GetXMLEnumAttributeOrName(). Metodin kutsutapa on siis sama, kuin jos se olisi osa käytettyä luokkaa. Sillä ei ole pääsyä luokan yksityisiin metodeihin tai muuttujiin, mutta kutsumistapa tekee metodin käyttämisestä käytännöllisempää.

Tässä vaiheessa data on jo tietokantaluokissa, joten niiden vieminen tietokantaan on suurimmalta osalta LINQ to SQL:n vastuulla. Jokainen tietokantaluokka pitää viedä EntityFrameworkin tietokantajonoon joko 'InsertOnSubmit', 'DeleteOnSubmit' metodeilla tai muuttamalla tietokannasta haetun olion tietoja. Kun ne on viety jonoon, voi ne viedä tietokantaan käyttäen 'SubmitChanges'-metodia.

Tässä luokassa tietojen tallennukseen käytetiin samanlaista rakennetta kuin tietojen tarkastukseen. Tämä tehtiin, koska rakenne oli jo todettu hyväksi, ja todettiin että, mahdollisten tulevien muutosten takia olisi hyvä, että luokkien rakenteet olisivat samanlaisia.

Jos tietokantaan tietoa viedessä tulee virheitä, ne löytävät kontekstin muuttujasta 'ChangeConflicts'. Virheet otetaan kiinni ja lähetetään takaisin ohjelmalle heittämällä poikkeus, jossa on tiedot tulleesta virheestä.

6 Konsoliohjelman luonti

XML-sisäänluvun testaamiseksi luotiin konsoliohjelma, jolle annetaan parametrina sisään luettava tiedosto ja XSD-skeema, jota vastaan XML-tiedosto validoidaan. Myöhemmässä vaiheessa sisäänluku kytketään osaksi suurempaa kokonaisuutta, mutta tämä ei ole osa opinnäytetyötä. Konsoliohjelman tarkoituksena on auttaa tiedon sisäänluvun testausta ottamalla väliaikaisesti jo olemassa olevan sisäänlukuohjelmiston aseman. Koska konsoliohjelma on muuta ohjelmistoa huomattavasti yksinkertaisempi, on helpompi olla varma, kun sitä käytetään datan sisäänluvussa. Vastaantulevat ongelmat johtuvat uudesta ohjelman osasta, eikä jo järjestelmässä olemassa olevista ongelmista.

Tiedoston ja skeeman nimien sisäänluku parametreina on helppoa. Parametrien tarkoituksen kommunikoiminen käyttäjälle ja niiden tarkastaminen on hieman monimutkaisempaa. NuGet-pakkaustenhallintajärjestelmässä on monia jo olemassa olevia hyviä ratkaisuja. Valitettavasti projektin .NET Framework- ja MSBuild-versioiden takia tähän ei löytynyt toimivaa ratkaisua. Tarvittavat tarkastukset ja validoinnit piti luoda manuaalisesti.

Validoinnissa tarkastetaan annettujen parametrien määrä ja se, onko tiedostoja olemassa parametrien osoittamissa paikoissa. Jos parametrien lukumäärä on väärä, tai parametreissa olevia tiedostoja ei ole olemassa, niin ohjelma tulostaa konsoliin virheviestin ja lopettaa heti ohjelman ajon. Jos taas tiedostot ovat olemassa, ohjelma lukee tiedoston sisällön 'MemoryStream'-olioon ja ottaa niistä bittitaulukon. Konsoliohjelma kutsuu työssä luotua XML-sisäänlukijaa annetuilla tiedostoilla, ja yrittää ajaa sen.

Ohjelman ajo tehdään try catch -lohkon sisällä. Jos ajossa tapahtuu virheitä, catch lohko ottaa ne kiinni ja tulostaa konsoli ikkunaan saamansa virheviestit. Viestien tulostus helpottaa ongelmatilantilanteiden selvittämistä. Jos taas ohjelma ajetaan läpi onnistuneesti, ohjelma tulostaa konsoliin onnistumisviestin.

Työn myöhäisemmässä vaiheessa ohjelmaa ei enää ajettu konsoliohjelman kautta. Se yhdistettiin verkkosivulla olevaan lomakkeeseen, jota oli käytetty edellisessä versiossa sisäänlukuominaisuudesta. Ohjelmaa ei heti yhdistetty tähän lomakkeeseen, koska se vaati muutoksia sisäänlukuun, joiden toteutusta ei ollut vielä suunniteltu.

7 Yhteenveto

Työssä toteutettiin järjestelmä, joka lukee sisälle XML-muotoista dataa, tarkastaa sen, ja lopuksi tallentaa sen tietokantaan.

Ohjelmiston toteutuksessa oli joitain ongelmia, lähinnä liittyen .NETissä olevaan XML-deserialisoijaan ja XSD-skeemamäärittelyjen puutteellisuuteen ja tästä johtuneisiin muutoksiin sekä skeemaan että ohjelmaan. Skeeman ongelmia piti iteroida monta kertaa asiakkaan kanssa, mutta lopulta päädyttiin versioon, joka oli sekä heidän haluamansa mukainen että meidän toteutettavissa. XML-deserialisointiongelmille taas ei saatu hyvää ratkaisua, vaan ongelma piti kiertää.

Työssä saatiin toteutettua järjestelmä, joka vastasi suurimmilta osin asiakkaan vaatimuksia. Ainoa osa, johon asiakas ei välttämättä ollut täysin tyytyväinen oli XML-tiedoston sisäänluvussa tapahtuvien tiedoston rakenteellisten ongelmien raportoinnissa. Tähän mennessä tällaisia mahdollisia rakenteellisia ongelmia on testeissä tullut vastaan kahdenlaisia. Ensimmäinen mahdollinen rakenteellinen ongelma liittyy lueteltuihin attribuutteihin. Jos näihin annetaan jokin muu arvo kuin joku määrätyistä, niin ohjelma lopettaa suorituksen siihen ja ilmoittaa asiakkaalle virheestä. Ongelma tässä on se, että asiakas halusi, että kaikki tiedostossa olevat virheet näytettävän kerralla. Koska ohjelman suoritus loppuu, niin muita mahdollisia virheitä ei voi tiedostossa näyttää.

Tästä on kuitenkin annettu asiakkaalle suositus, että nykyiset rajoitukset tässä hyväksyttäisiin, koska niiden kiertäminen olisi vaikeaa. Näiden ongelmien ratkaisu olisi teoriassa

mahdollista, mutta se vaatisi ratkaisuja, jotka ovat vaikeasti ylläpidettäviä. Ongelmat myös liittyvät vain tiedoston rakenteellisiin ongelmiin, joten jos asiakas korjaa omassa päässään tiedostot luovan ohjelman, niin ongelman ei pitäisi enää toistua. Näiden lisäksi rakenteelliset ongelmat ovat osa XML- ja XSD-määrityksiä. Näiden kiertäminen ei olisi valittujen teknologioiden käyttömallien mukaista.

Opinnäytetyössä tehty ohjelman osa on osa isompaa asiakkaan ohjelmistolle tehtävää päivitystä. Ohjelmisto menee tuotantokäyttöön, kun loput päivityksestä saadaan valmiiksi ja asiakas on sen hyväksynyt.

Lähteet

1 Verohallinto valitsi Digia Finland Oy:n kansallisen tulorekisterin ohjelmistotoimittajaksi. <https://www.vero.fi/tietoa-verohallinnosta/uutishuone/lehdist%C3%B6tiedotteet/2016/verohallinto_valitsi_digia_finland_oyn_/> Luettu 10.11.2019.

2 Extensible Markup Language (XML) 1.0 (Fifth Edition) <https://www.w3.org/TR/2008/REC-XML-20081126/> Luettu 10.11.2019.

3 W3C XML Schema Definition Language (XSD). <https://www.w3.org/TR/XMLschema11-1/> Luettu 10.11.2019.

4 FOREIGN KEY Constraints. <https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms175464(v=sql.105)> Luettu 10.11.2019.

5 C# 6.0 draft specification. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction> Luettu 21.7.2019.

6 XML Schema Definition Tool (Xsd.exe) <https://docs.microsoft.com/en-us/dotnet/standard/serialization/xml-schema-definition-tool-xsd-exe> Luettu 10.11.2019.

7 LINQ to SQL. <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/> Luettu 10.11.2019.

8 SqlMetal.exe (Code Generation Tool). <https://docs.microsoft.com/en-us/dotnet/framework/tools/sqlmetal-exe-code-generation-tool> Luettu 10.11.2019.

9 Henkilötunnus <https://vrk.fi/henkilotunnus> Luettu 10.11.2019.