TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JANI HEININEN

# AUDIT TRAIL -TOIMINNALLISUUDEN TOTEUTTAMINEN .NET -TEKNIIKOILLA

Diplomityö

Tarkastaja: professori
Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston
kokouksessa 8. huhtikuuta 2015

# TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
**JANI HEININEN**: Audit Trail -toiminnallisuuden toteuttaminen .NET -tekniikoilla
Diplomityö, 49 sivua, 10 liitesivua
Joulukuu 2015
Pääaine: Pervasive Systems
Tarkastaja: professori Tommi Mikkonen
Avainsanat: Audit Trail, .NET, Entity Framework, herätin, tietokanta

Tietojärjestelmiin kohdistuu erilaisia luotettavuusvaatimuksia, joista yksi on tiedon jäljitettävyys. Jäljitettävyyden tarkoituksena on luoda tietoihin kohdistuneiden muutosten välille katkeamaton ketju. Ketjua seuraamalla pystytään selvittämään kuka on muokannut tietoja, milloin muutokset on tehty ja mitä muutoksia tietoihin on tehty. Jäljitettävyyden toteuttava mekanismi tunnetaan nimellä Audit Trail.

Työn taustalla on Turun PET-keskukselle toteutettu PET ERP -toiminnanohjausjärjestelmä, johon Audit Trail -mekanismi haluttiin toteuttaa. Positroniemissiotomografia (PET) on isotooppilääketieteen alaan kuuluva kuvantamismenetelmä.

Audit Trailista on olemassa erilaisia versioita. Toteutustapa riippuu siitä, mitä tietoa halutaan tallentaa ja millä tasolla. Tässä diplomityössä suunniteltiin ja toteutettiin kaksi erilaista tietokantaa käyttävää Audit Trail -ratkaisua. Toteutustapoja vertailtiin toteutettavuuden, suorituskyvyn, ylläpidettävyyden ja uudelleenkäytettävyyden näkökulmasta.

Ensimmäinen Audit Trail -mekanismi toimii sovellustasolla ja käyttää Audit Trail -tietojen tallentamisessa hyödyksi Entity Framework -ohjelmistokehystä. Toinen mekanismi toimii tietokannan tasolla ja perustuu tietokannan herättimiin. Tässä työssä käydään läpi kummankin mekanismin toteutuksen yksityiskohdat.

Tämän työn tuloksena saatiin selvitettyä kahden erilaisen Audit Trail -tallennusmekanismin hyvät ja huonot puolet. Sovellustason toteutuksen vahvuuksia ovat toteutettavuus, ylläpidettävyys ja uudelleenkäytettävyys. Tietokantatason toteutus hävisi hieman näissä kriteereissä, mutta se oli selvästi suorituskyvyltään tehokkaampi. Työn lopussa esitellään johtopäätökset vertailun tuloksista ja arvioidaan tämän työn tulosten perusteella mikä olisi järkevin tapa toteuttaa Audit Trail kokonaan uudessa projektissa.

# ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
**JANI HEININEN**: Implementation of Audit Trail with .NET technologies
Master of Science Thesis, 49 pages, 10 Appendix pages
December 2015
Major: Pervasive Systems
Examiner: Professor Tommi Mikkonen
Keywords: Audit Trail, .NET, Entity Framework, trigger, database

Information systems have a variety of reliability requirements, one of which is data traceability. Purpose of traceability is to create an unbroken chain between the changes in data. By following the chain it is possible to determine who has edited the information, when the changes have been done and what changes have been made to the information. Mechanism that implements traceability is known as Audit Trail.

This Master of Science thesis was part of a PET ERP project, where an ERP (enterprise resource planning) system was implemented for the Turku PET Centre. The Audit Trail mechanism was a part of PET ERP. Positron emission tomography (PET) is a nuclear medicine imaging technique.

There are different kinds of versions of Audit Trail mechanism. Implementation method depends on what information needs to be stored and at what level. In this master's thesis the focus is on design and implementation of two different Audit Trail recording mechanisms that use database as storage. Both solutions were evaluated from the viewpoint of feasibility, performance, maintainability and reusability.

The first Audit Trail mechanism operates at the application level and uses Microsoft Entity Framework to save the Audit Trail information. The second mechanism operates at database level and is based on database triggers. In this master's thesis the implementation details of both Audit Trail mechanisms are reviewed.

Pros and cons of two different Audit Trail mechanisms were clarified as the results of this master's thesis. The strengths of application-level implementation are feasibility, maintainability and reusability. The database-level implementation was slightly worse at these three criteria, but it was clearly more effective from a performance perspective. The conclusions of the comparison results are presented at the end of the work. Based on the results of this work an estimation is made, what would be the most sensible way to implement the Audit Trail on an entirely new project.

# ALKUSANAT

Haluan kiittää työni tarkastajaa Tommi Mikkosta ja ohjaajaa Miika Parviota hyvistä ja rakentavista kommenteista. Tämä työ on tehty Atostek Oy:ssä diplomi-insinöörityutkinnon opinnäytetyönä. Kiitän Atostek Oy:tä työn aiheen tarjoamisesta ja rahoituksesta. Kiitän Atostek Oy:n työntekijöitä ja johtoa erinomaisesta työilmapiiristä ja saadusta tuesta. Haluan kiittää myös tyttöystävääni Saanaa, vanhempiani ja ystäviäni tuesta ja motivoinnista opiskeluiden aikana ja tätä diplomityötä tehdessä.

Tampereella, 18.11.2015

Jani Heininen

heininej@kapsi.fi

# SISÄLLYSLUETTELO

# 1. JOHDANTO

Organisaation tiedot ovat usein pirstaloituneet useisiin eri tietojärjestelmiin. Toisistaan erillään olevat järjestelmät voivat olla organisaation liiketoiminnan kannalta tärkeitä, mutta yhdessä ne muodostavat hidasteen organisaation tuottavuudelle ja tehokkuudelle. Useiden erillisten tietojärjestelmien ylläpito aiheuttaa suuria kustannuksia. Yrityksen tietojärjestelmien pirstaloituminen johtaa siihen, että myös yrityksen liiketoiminta pirstaloituu.

Edellä mainittujen ongelmien ratkaisemiseksi yksi laajalti erilaisissa organisaatioissa käytössä olevista tietojärjestelmistä on toiminnanohjausjärjestelmä (ERP-järjestelmä, Enterprise Resource Planning System). Toiminnanohjausjärjestelmä on organisaation tietojärjestelmä, jonka tehtävänä on yhdistää ja tehostaa organisaation sisäisiä tietovirtoja ja toimintoja, kuten tuotantoa, jakelua, varastonhallintaa, henkilöstöhallintoa, toimitusketjuja, laskutusta ja kirjanpitoa. Nykyaikainen ERP-järjestelmä koostuu tyypillisesti erillisistä moduuleista, kuten palkanlaskenta, reskontra, kirjanpito, tuotannonohjaus, materiaalinhallinta ja materiaalien, projektien, huollon ja resurssien hallinta. Toiminnanohjausjärjestelmä käsittelee suurta tietomäärää, joka täytyy pystyä tallentamaan luotettavasti. Tätä varten toiminnanohjausjärjestelmä sisältää keskitetyn tietokannan, johon kaikki moduulit tallettavat tietonsa ja jonka välityksellä ne kommunikoivat keskenään. [1]

Tämän diplomityön taustalla on Turun PET-keskukselle toteutettu PET ERP -toiminnanohjausjärjestelmä, jonka toteuttavana osapuolena toimii Atostek Oy. Turku PET Centre on Turun yliopiston, Åbo Akademin ja Turun yliopistollisen keskussairaalan (Tyks) yhteinen, valtakunnallinen tutkimuskeskus. Keskuksen toimintaan kuuluvat potilastutkimukset, joissa PET-kuvausmenetelmällä (positroniemissiotomografia) selvitetään esimerkiksi syövän levinneisyyttä, tutkitaan aivoja ja sydäntä tai etsitään tulehduspesäkkeitä. Lisäksi keskuksessa tehdään korkeatasoista tieteellistä tutkimus- ja julkaisutoimintaa ja kehitetään ja valmistetaan kuvauksessa käytettäviä positronisäteileviä radionuklideja. Kuvausmenetelmä perustuu lyhytikäisien radionuklidien käyttöön. Radioaktiivinen merkkiaine koostuu positroneja lähettävistä radionuklideista. Merkkiaine injisoidaan kuvattavan potilaan elimistöön, jossa se kulkeutuu kuvattavaan kohteeseen. Positronien ja elimistössä olevien elektronien törmäyksen seurauksena vapautuu gammasäteilyä, joka voidaan havaita kehon ulkopuolelta erikoiskameralla (PET-skanneri). Gammasäteily on ionisoivaa säteilyä, joka suurina määrinä voi vaurioittaa solujen perimäainesta. Pahimmassa tapauksessa seurauksena voi olla sairastuminen syöpään tai muut terveyshaitat. Kuvauksissa käytetään pieniä säteilymääriä, mutta radioaktiivisen

aineen vaarallisuudesta johtuen radioisotooppien valmistaminen on tarkasti säännösteltyä ja luvanvaraista toimintaa. [2]

PET ERP on terveydenhuollon tietojärjestelmä. Tätä kautta siihen kohdistuu erilaisia vaatimuksia, joista yksi on tietojen jäljitettävyys. PET ERP:ssä oleviin tietoihin kohdistuvista muutoksista täytyy jäädä pysyvät merkinnät. Merkintöjen avulla voidaan myöhemmin tarkistella muun muassa kuka on tehnyt muutoksen, milloin muutos on tehty ja mitä tietoja on lisätty, muutettu tai poistettu. Jäljitettävyydelle pitää näin ollen toteuttaa oma mekanisminsa. Mekanismi tunnetaan nimellä Audit Trail. Audit Trail -toiminnallisuus voidaan toteuttaa useilla eri tavoilla.

Tässä diplomityössä suunniteltiin ja toteutettiin PET ERP -toiminnanohjausjärjestelmään liittyvä tietokantapohjainen Audit Trail -ratkaisu ja vertailtiin erilaisia toteutustapoja toteutettavuuden, suorituskyvyn, ylläpidettävyyden ja uudelleenkäytettävyyden näkökulmasta. Samalla työ rajautuu käsittelemään PET ERP:stä vain palvelinpään toteutusta, joka sisältää Audit Trail -mekanismin.

Audit Trail -mekanismin toteuttaminen voidaan jakaa kahteen eri osaan: tietojen tallennusmekanismin toteutustapa ja tietojen tallennukseen käytettävän tietokannan skeema. Tietojen tallennusmekanismin toteutusvaihtoehdoiksi valittiin kaksi erilaista tapaa. Ensimmäisessä ratkaisuvaihtoehdossa, joka otettiin käyttöön myös PET ERP:n tuotantoversiossa, käytetään Entity Framework -ohjelmistokehystä. Entity Framework toimii rajapintana sovelluksen ja tietokannan välissä abstrahoiden tietokantakohtaiset komennot ohjelmistokehittäjältä ja mahdollistaen tietokantaoperaatioiden suorittamisen käyttäen sovellustason olioita. Ensimmäinen ratkaisuvaihtoehto on näin ollen toteutettu sovellustasolla. Toinen ratkaisuvaihtoehto käyttää tietokannan herättimiä (trigger). Herättimet ovat käytännössä tietokannan tasolla olevaa ohjelmakoodia, jossa voidaan tarkastella tietokannan tilaa, siihen kohdistuvia operaatioita ja toteuttaa ehdollisia tietokantaoperaatioita. Molemmissa ratkaisuvaihtoehdoissa käytetään erillistä tietokantaa, jonka tauluihin Audit Trail -tiedot tallennetaan. Tietokannan skeema voidaan toteuttaa vaatimuksista riippuen monella eri tavalla. Tässä työssä käydään läpi muutama eri vaihtoehto skeeman toteuttamiseksi. Työn pääasiallinen tutkimuskohde on tietojen tallennusmekanismi. Kahden eri tallennusmekanismin vertailuosioon valittiin tietokannan skeemaksi yksi esitellyistä toteutustavoista.

Audit Trail -mekanismin taustat on kuvattu luvussa 2. Aluksi esitellään Audit Trail käsitteenä ja kerrotaan miksi sitä tarvitaan ja mitä hyötyjä sillä saavutetaan. Luvussa kerrotaan myös erilaisista lakien, viranomaisten ja asiakkaan asettamista vaatimuksista.

Luvussa 3 esitellään Audit Trail -mekanismin toteutuksissa käytetyt tekniikat ja työkalut. Työkalut ja tekniikat rajoittuvat tässä työssä erilaisiin Microsoftin tuotteisiin. Pääosassa ovat .NET-ohjelmistokehys, Visual Studio -kehitysympäristö,

C#-ohjelmointikieli, Microsoft SQL Server -tietokannan hallintajärjestelmä ja oliorelaatio-muunnoksesta huolehtiva Entity Framework -ohjelmistokehys.

Luvussa 4 esitellään erilaisia vaihtoehtoja Audit Trail -tietomallin toteuttamiselle ja tehdään rajaus tässä työssä käytettävästä ratkaisusta. Lisäksi luvussa esitellään vertailussa huomioon otettavat näkökulmat ja kerrotaan mitä ne sisältävät.

Luvussa 5 esitellään kaksi erilaista Audit Trailin tallennusmekanismia ja niiden toteutuksen periaatteet. Toteutustavat ovat Entity Frameworkia hyödyntävä sovellustason tallennuslogiikka ja tietokannan herättimiä hyödyntävä tietokantatason mekanismi.

Luku 6 sisältää luvussa 5 esiteltyjen tallennusmekanismien arvioinnin toteutettavuuden, suorituskyvyn, ylläpidettävyyden ja uudelleenkäytettävyyden näkökulmasta. Lisäksi lopussa vertaillaan mekanismien keskinäistä paremmuutta eri näkökulmien kannalta.

Luvussa 7 tehdään yhteenveto valituista ratkaisuista ja vertailun kohteena olleista tallennusmekanismeista.

# 2. AUDIT TRAIL

Tässä luvussa esitellään Audit Trail ja siihen liittyvät vaatimukset. Kohdassa 2.1 esitellään Audit Trailin käsite ja sen tarkoitus. Kohdassa 2.2 on kerrottu eritasoisista Audit Trail vaihtoehdoista. Kohdassa 2.3 esitellään Audit Trailin toteuttamiseen liittyviä haasteita. Kohdassa 2.4 kerrotaan asiakkaan, viranomaisten ja lakien asettamista vaatimuksista ja reunaehdoista. Lopussa muodostetaan lista tärkeimmistä vaatimuksista, jotka Audit Trail toteutuksen tulee täyttää. Luku perustuu lähteeseen [3], ellei tekstissä toisin mainita.

## 2.1 Mikä on Audit Trail

Suurissa järjestelmissä useilla käyttäjillä on mahdollisuus lukea ja muokata järjestelmän dataa. Jossain vaiheessa dataa saattaa hävitä tai se voi vääristyä joko käyttäjien tekemien toimien tai järjestelmässä olevan virheen seurauksena. Tällaisia tilanteita varten tarvitaan keino valvoa järjestelmän dataan kohdistuneita toimenpiteitä. Audit Trail ylläpitää tallenteita järjestelmä- ja sovellusprosessien tapahtumista ja käyttäjien tekemistä järjestelmä- ja sovellustoiminnoista.

Audit Trailia voidaan käyttää taustalla toimivana varmistuksena, säännöllisten järjestelmäoperaatioiden tukityökaluna tai molempina. Varmistuskäytössä Audit Trail -tietoa kerätään, mutta sitä käytetään vasta tarvittaessa. Tukityökaluna järjestelmäylläpitäjät voivat käyttää Audit Trailia varmistaakseen, että järjestelmä ei ole vahingoittunut ulkoisten hyökkääjien, järjestelmän sisäisten käyttäjien tai teknisten ongelmien vuoksi.

Audit Trailia voidaan tallentaa monella eri tasolla ja erilaisilla tekniikoilla. Audit Trailia voidaan tallentaa joko yksittäisen sovelluksen tasolla tai laajemmin esimerkiksi koko käyttöjärjestelmän tasolla. Tallennustapana Audit Trail -tiedolle voidaan käyttää yksinkertaisimmillaan tekstitiedostoa. Monimutkaisemmassa tilanteessa järjestelmässä on tietokanta, jonka sisältämästä tiedosta halutaan ylläpitää Audit Trailia.

Audit Trailiin täytyy tallentaa riittävän paljon Audit-tietoa, jotta sen pohjalta voitaisiin jäljittää virheitä. Vähimmäisvaatimukset ovat:

- Päivämäärä ja aika jolloin järjestelmän tapahtuma ilmeni.
- Käyttäjän ID tai nimi joka liittyy ilmenneeseen tapahtumaan.
- Ohjelma tai komento joka aiheutti tapahtuman.
- Tapahtumasta seurannut lopputulos tietoihin.

Audit Trailin avulla voidaan saavuttaa useita turvallisuuteen liittyviä tavoitteita ja hyötyjä. Näitä tavoitteita ovat yksilön saattaminen vastuuseen teoistaan, tapahtumien jälleenrakentaminen, tunkeutumisen havaitseminen ja ongelmien analysointi.

### 2.1.1 Yksilön vastuu

Jo pelkkä tieto käyttäjien toimia seuraavan Audit Trail -tiedon keräämisestä ohjaa käyttäjiä toimimaan oikein. Todennäköisyys järjestelmän turvallisuusominaisuuksien kiertämisyrityksille tai järjestelmän väärinkäytöksille on pienempi, mikäli käyttäjät tietävät heidän toimistaan kerättävästä tiedosta.

Audit Trailia voidaan käyttää yhdessä pääsymekanismien kanssa. Pääsymekanismi voi olla esimerkiksi käyttäjätunnus ja salasana. Tällöin pystytään tunnistamaan ja saamaan tietoa niistä käyttäjistä, joiden epäillään tehneen vääriä muokkauksia tietoihin. Audit Trail -mekanismi voi tallentaa tiedoista kaksi eri tilaa. Ensimmäinen tila sisältää tiedot siinä muodossa, missä ne ovat ennen niihin kohdistuneita muutoksia. Jälkimmäinen tila sisältää tietokantaoperaatioiden seurauksena muuttuneet tiedot. Näiden kahden version perusteella pystytään vertailemaan mitä muutoksia oikeasti tehtiin ja mitä olisi pitänyt tehdä. Tiedoista pystytään päättelemään johtuiko virhe käyttäjän tekemistä toimista, sovelluksessa olevasta virheestä vai jostakin muusta syystä.

Pääsymekanismit rajoittavat käyttöoikeutta järjestelmän resursseihin. Käyttäjille on myönnettävä riittävät oikeudet, jotta he voivat käyttää järjestelmää tehtäviensä suorittamiseen. Tämä ei kuitenkaan poista käyttäjän mahdollisuutta tahallaan väärinkäyttää saamiaan oikeuksia. Esimerkiksi lääkärillä on pääsy niiden potilaiden tietoihin, joista hän on vastuussa. Audit Trailiin tallentuneiden tietojen perusteella voidaan havaita, mikäli lääkäri yrittää hakea poikkeuksellisen suurta määrää potilastietoja. Tämä herättää epäilyksen potilastietojen väärinkäytöksestä.

### 2.1.2 Tapahtumien jälleenrakentaminen

Audit Trailia voidaan käyttää tapahtumien jälleenrakentamiseen ongelmatilanteiden yhteydessä. Audit Trailiin tallentuneen tiedon pohjalta pystytään tutkimaan miten, miksi ja milloin järjestelmän normaaliin toimintaan tuli häiriö. Tätä kautta voidaan päästä ongelman aiheuttaneen virheen jäljille. Analysoimalla Audit Trailiin tallentunutta tietoa pystytään usein päättelemään, johtuiko virhe järjestelmässä olevasta ohjelmointivirheestä vai käyttäjän tekemästä virheestä.

Jos järjestelmässä havaitaan toimintahäiriö ja esimerkiksi jokin osa sovelluksen tiedoista on korruptoitunut, voidaan tiedot yrittää palauttaa oikeaan tilaan Audit Trailin avulla. Jotta tämä olisi mahdollista, Audit Trailin pitää säilyttää koko muutoshistoria kaikesta sovelluksen sisältämästä tiedosta.

### 2.1.3 Tunkeutumisen havaitseminen

Pelkän ohjelman tietoihin tehtyjen lisäysten, muutosten ja poistojen kirjausketjun ylläpitämisen lisäksi Audit Trailin avulla voidaan tunnistaa tunkeutumisyritykset järjestelmään. Tällöin Audit Trailiin täytyy rakentaa mekanismi, joka tallentaa tietoa kaikista kirjautumisyrityksistä. Tunkeutumisyritykset voidaan tunnistaa Audit Trail -merkinnöistä joko reaaliajassa Audit Trail -merkintää tehdessä tai jälkeenpäin eräajona.

Reaaliaikaista tunkeutumisen havaitsemista käytetään yleensä ulkopuolisten tunkeutumisyritysten tunnistamiseen. Sitä voidaan käyttää myös esimerkiksi järjestelmän suorituskykymuutosten havainnointiin. Negatiiviset muutokset suorituskyvyssä voivat olla merkki viruksesta tai muusta haittaohjelmasta. Jälkeenpäin tehtävissä tarkistuksissa voidaan havaita tunkeutumisyritykset tai onnistuneet tunkeutumiset. Havaintojen pohjalta tehdään vahinkoarvio tai korjataan tunkeutumisen seurauksena aiheutuneet ongelmat.

## 2.2 Erilaiset Audit Trail -vaihtoehdot

Järjestelmä voi ylläpitää yhtäaikaisesti useampaa erilaista Audit Trailia. On olemassa yleisesti ottaen kaksi erilaista tallennusvaihtoehtoa: tapahtumapohjainen loki ja jokaisen näppäinpainalluksen tallentaminen. Tapahtumapohjainen loki sisältää yleensä järjestelmätapahtumien, sovellustapahtumien tai käyttäjätapahtumien lokitietoja.

Näppäinpainallusten valvomisessa tarkastetaan tai tallennetaan käyttäjän antamia syötteitä ja tietokoneen antamaa vastetta. Tämä on Audit Trailin erikoistapaus. Näppäinpainallusten valvontaa tehdään järjestelmän ja tietojen suojelemiseksi tunkeutujilta, joilla ei ole lupaa käyttää järjestelmää tai jotka ylittävät valtuutensa. Näppäinpainallusten valvonnasta esimerkkinä voidaan mainita käyttäjän syöttämien merkkien valvominen ja käyttäjien sähköpostin lukeminen.

Tapahtumapohjaisista lokeista järjestelmälokia käytetään yleensä järjestelmän suorituskyvyn valvontaan ja hienosäätämiseen. Sovelluslokeja voidaan käyttää sovellusvirheiden tai sovelluksessa tehtyjen tietoturvalinjausten rikkomisen havaitsemiseen. Käyttäjälokien avulla saadaan yksilöt vastuuseen teoistaan.

Järjestelmätason lokiin pitäisi tallentaa vähintään kaikki sisäänkirjautumisyritykset, kirjautumisen yksilöivä tunniste, sisäänkirjautumisten ja uloskirjautumisten päivämäärä ja aika, laitteet joita on käytetty ja sisäänkirjautumisen jälkeen suoritetut toiminnot, esimerkiksi sovellusten avaaminen.

Järjestelmätason loki ei välttämättä pysty tallentamaan kaikkea haluttua tietoa sovellusten sisällä. Tällöin joudutaan käyttämään sovellustason lokia, joka valvoo käyttäjien tekemiä toimia, avattuja ja suljettuja tiedostoja, tallenteiden tai kenttien lukemista, muokkaamista tai poistamista ja raporttien tulostamista. Joillekin sovelluksille saatetaan

asettaa korkeita vaatimuksia tiedon saatavuuden suhteen, jolloin toiminnan kohteena olevasta tiedosta pitää tallentaa Audit Trailiin versiot ennen ja jälkeen tietoon kohdistuvia operaatioita.

Audit Trailin tallentamisessa pitää ottaa huomioon yksityisyydensuoja. Käyttäjien tulee tietää sovellettavista yksityisyyttä koskevista laeista, säännöksistä ja menettelytavoista, jotka koskevat käyttäjien tekemien toimien tallentamista ja niiden myöhempää tarkistelua.

## 2.3 Toteutuksen haasteet

Audit Trail -tietoihin kohdistuu erilaisia vaatimuksia. Tietojen on oltava saatavilla silloin kun niitä tarvitaan ja tietojen on oltava eheitä, jotta ne olisivat hyödyllisiä. Audit Trailiin tallennetulle tiedolle täytyy suorittaa myös ajoittaisia katselmointeja. Katselmointi voidaan suorittaa tarpeen vaatiessa, automaattisesti reaaliajassa tai jollakin näiden muodostamalla yhdistelmällä. Audit Trail -tiedolle pitää määritellä aika, miten kauan tietoa säilytetään. Tähän voivat vaikuttaa eri lait ja säännökset.

Audit Trail -tieto vaatii suojaamista, ja pääsyä siihen pitää valvoa tarkasti. Tavallisille käyttäjille pääsyä ei myönnetä ollenkaan. Järjestelmävalvojille lupa sen sijaan myönnetään, jotta he voivat suorittaa tiedolle tarkistuksia. Tietoja pitää suojella tarkasti, sillä tunkeutujat voivat yrittää peitellä jälkiään muokkaamalla Audit Trailin tietoja. Audit Trailiin tallennettua tietoa saatetaan käyttää esimerkiksi oikeudenkäyntitapauksissa, jolloin tiedon yhtenäisyys ja eheys on ensiarvoisen tärkeää. Audit Trailiin tallentuneet tiedot voivat olla hyvin luottamuksellisia ja arkaluontoisia, jolloin ne eivät saa vuotaa julkisuuteen. Luottamuksellisuusvaatimus saadaan täytettyä tiedon salaamisella ja käyttämällä vahvoja pääsynvalvontamekanismeja.

## 2.4 Asiakkaan ja lakien asettamat vaatimukset

Turun PET-keskus on PET ERP -tietojärjestelmähankkeen asiakas. Hankkeen alussa asiakkaalta on saatu vaatimusmäärittely, jossa määritellään miten lopullisen ohjelmiston tulisi toimia ja millä keinoilla toiminnallisuus toteutetaan. PET ERP on terveydenhuollon tietojärjestelmä, joten siihen kohdistuu myös Suomen lain asettamia vaatimuksia.

PET ERP:n vaatimusmäärittelyssä mainitaan, että järjestelmässä käsitellään kuvantamisprosessissa potilastietoja. Järjestelmän tulee näiltä osin noudattaa potilastietojen käsittelyssä määrättyjä asetuksia ja GCP-standardia [4].

GCP (Good Clinical Practice) on kansainvälinen eettinen ja tieteellinen laatustandardi. GCP:n on luonut ICH (International Conference on Harmonisation of Technical Requirements for Registration of Pharmaceuticals for Human Use) organisaatio, joka koostuu Euroopan, Japanin ja Yhdysvaltojen säätelevistä viranomaisista ja lääketeolli-

suuden ammattilaisista. ICH määrittelee standardeja, joista eri maiden hallitukset voivat johtaa ohjesääntöjä. Standardin määrittelemiä eettisiä ja tieteellisiä laatuvaatimuksia pitää noudattaa kliinisen tutkimuksen kaikissa vaiheissa. Standardin tavoitteena on huolehtia ihmisoikeuksien ja yksilön hyvinvoinnin turvaamisesta ja niiden asettamisesta tieteen ja yhteiskunnan etujen edelle. Ohjeistusta pitää noudattaa luotaessa uutta kliinistä tutkimusdataa, joka lähetetään sääteleville viranomaisille. Ohjeistuksen periaatteita voidaan noudattaa myös muissa kliinisissä tutkimuksissa, jotka vaikuttavat ihmisten hyvinvointiin ja turvallisuuteen. [5]

GCP-standardi ottaa kantaa ihmisiin kohdistuvien tutkimusten suunnitteluun, suorittamiseen, taltiointiin ja raportointiin. Ohjeistuksen noudattaminen vaatii käytännössä kokonaisvaltaisen dokumentaation ylläpitoa koskien kliinisiä protokollia, tallenteiden säilyttämistä, kouluttamista ja laitteistoa mukaan lukien tietokoneet ja niissä toimivat sovellukset. Standardiin sisältyy tätä kautta vaatimus myös Audit Trailin tallentamisesta. GCP-ohjeistuksen mukaan Audit Trail on dokumentaatio, jonka avulla pystytään tarkastelemaan jälkeenpäin tapahtumien kulkua. Ohjeistus sisältää määritelmän CRF:lle (Case Report Form). Se on tulostettu, optinen tai elektroninen dokumentti, joka sisältää kaiken protokollan vaatiman informaation, joka raportoidaan kunkin tutkimuskohteen kohdalla kyseisen tutkimuksen rahoittajalle. Kaikki CRF:ään kohdistuvat muutokset ja korjaukset tulee varustaa päivämäärällä, tekijän nimikirjaimilla ja tehtyjen muutosten selityksellä. Alkuperäisiä merkintöjä ei saa hävittää tai yli kirjoittaa. Tämä asettaa vaatimuksen Audit Trailin ylläpitämiselle. [6]

Sosiaali- ja terveysministeriö on tehnyt asetuksen potilasasiakirjoista. Asetuksen mukaan potilasasiakirjoihin kuuluvat potilaskertomus ja siihen liittyvät potilastiedot tai asiakirjat, lääketieteelliseen kuolemansyyn selvittämiseen liittyvät tiedot tai asiakirjat ja muut potilaan hoidon järjestämisen ja toteuttamisen yhteydessä syntyneet tai muualta saadut tiedot ja asiakirjat. Tietojen eheys ja käytettävyys tulee turvata tietojen säilytysaikana laatimalla ja säilyttämällä potilasasiakirjat asiaankuuluvia välineitä ja menetelmiä käyttäen. Potilasasiakirjamerkintöjen korjaaminen tulee tehdä siten, että alkuperäinen ja korjattu merkintä ovat myöhemmin luettavissa. Korjauksen tekijän nimi, virka-asema, korjauksentekopäivä ja korjauksen peruste tulee merkitä potilasasiakirjoihin. Jos potilasasiakirjoista poistetaan potilaan hoidon kannalta tarpeeton tieto, potilasasiakirjoihin tulee tehdä merkintä siitä, sen tekijästä ja poistamisajankohdasta. Sähköisten potilastietojen käyttöön ja luovutukseen liittyvät lokitiedot tulee säilyttää eheinä ja muuttumattomina vähintään 12 vuotta niiden syntymisestä. [7]

Audit Trail -mekanismin toteutukseen liittyy useita erilaisia vaatimuksia.  jotka voidaan muodostaa edellä mainittujen asioiden pohjalta seuraavanlaiset vaatimukset:

- Audit Trail -merkintöjen pitää sisältää tietoa muokanneen henkilön nimi, virka-asema, korjauksentekopäivä ja korjauksen peruste.

- Tietojen pitää säilyä eheinä ja muuttumattomina vähintään 12 vuotta. Tämä asettaa luotettavuusvaatimuksia Audit Trailin sisältävälle tietokannalle ja tietokantaa ajavalle laitteistolle.
- Kuka tahansa käyttäjä ei saa päästä katselemaan ja muokkaamaan Audit Trailin sisältämiä tietoja.

# 3. RATKAISUVAIHTOEHDOT JA KÄYTETYT TEKNIIKAT

Tässä luvussa käydään läpi diplomityössä käytetyt ratkaisuvaihtoehdot ja esitellään keskeisimpien käytettyjen tekniikoiden taustat. Taulukossa 3.1 on listattu työssä käytetyt työkalut ja tekniikat sekä niiden versiot. Kohdassa 3.1 esitellään .NET-ohjelmistokehys, C#-ohjelmointikieli ja Visual Studio -kehitysympäristö. Kohdassa 3.2 käydään läpi Microsoft SQL tietokannan hallintajärjestelmä ja herättimet tämän työn kannalta oleellisiltaosin. Kohdassa 3.3 selitetään olio-relaatio-kuvauksen periaate ja käsitellään ADO.NET ja Entity Framework -ohjelmistokehykset, jotka toteuttavat olio-relaatio-kuvauksen.

*Taulukko 3.1 Työssä käytetyt tekniikat ja työkalut*

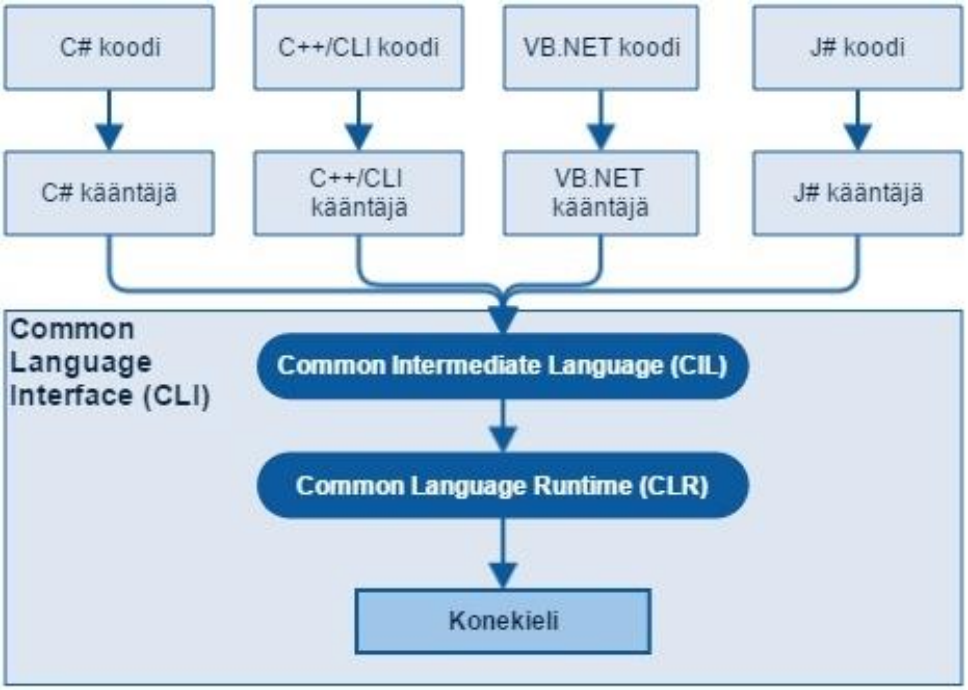
| **Komponentti** | **Versio** |
|---|---|
| Visual Studio | 2012 |
| .NET Framework | 4.0 |
| Entity Framework | 5.0.0 |
| Microsoft SQL Server | 2012 |

## 3.1 .NET Framework

.NET Framework on Microsoftin kehittämä kehittämä ohjelmistokomponenttikirjasto, jota käytetään sovelluskehitykseen eri Microsoftin alustoille. Kirjaston kehittäminen aloitettiin 1990-luvulla, ja sen nimi oli aluksi Next Generation Windows Services (NGWS). Kirjastosta on tähän asti julkaistu viralliset versiot 1.0, 1.1, 2.0, 3.0, 3.5, 4.0 ja 4.5. Tässä työssä käytetään .NET Frameworkin versiota 4. Kirjasto koostuu kahdesta osasta, ajonaikaisesta ympäristöstä (CLR, Common Language Runtime) ja luokkakirjastosta. .NET tukee lukuisia eri ohjelmointikieliä, kuten C#, C++/CLI, F#, J# ja Visual Basic .NET. [8]

.NET:n kieliriippumattomuus perustuu Common Language Infrastructure (CLI) -määrittelyyn. CLI määrittelee käännösympäristön, joka sallii useiden korkean tason ohjelmointikielten käyttämisen erilaisilla alustoilla ilman, että koodi pitäisi uudelleenkirjoittaa jokaiselle tietokonearkkitehtuurille. Kaikkien eri .NET-tuen sisältävien ohjelmointikielien koodi käännetään aluksi välikielelle (CIL, Common Intermediate Language) kielikohtaisilla kääntäjillä. Tämän jälkeen .NET:n ajonaikainen ympäristö (CLR, Common Language Runtime) tulkkaa CIL-välikielikoodin ajonaikaisesti prosessorilla

suoritettavaksi konekieliseksi koodiksi. Tulkattavuuden ansiosta kahdella eri .NET tuen sisältävällä kielellä kirjoitetut ohjelmat pystyvät kutsumaan toistensa metodeja. CLR on Microsoftin toteutus CLI:stä, mutta sille on olemassa myös vaihtoehtoisia toteutuksia, kuten Mono ja Portable.NET. .NET ohjelmistokehyksen kääntämisen vaiheet on esitetty kuvassa 3.1. [8]




*Kuva 3.1 NET ohjelmistokehyksen kääntämisprosessi*

CLR huolehtii muun muassa muistinhallinnasta, säikeiden suorituksesta, ohjelmakoodin suorituksesta, koodin turvallisuustarkistuksista, kääntämisestä, poikkeusten käsittelystä ja roskienkeruusta. Roskienkeruu huolehtii käyttämättömäksi jääneen varatun muistin vapauttamisesta. Se eliminoi yleisimmät ohjelmavirheet, muistivuodot ja virheelliset muistiviittaukset. CLR tukee koodin vakautta määrittelemällä yleisen tyyppijärjestelmän (CTS, Common Type System), joka kaikkien .NET-yhteensopivien kielten on toteutettava. CTS määrittelee kaikille kielille yhteiset CLR:n tukemat datatyypit ja rakenteet, joiden avulla eri kielillä toteutetut ohjelmat voivat kommunikoida helposti keskenään. [8]

.NET-luokkakirjasto tarjoaa laajan kokoelman sovelluskehityksessä käytettäviä luokkia, rajapintoja ja arvotyyppejä, jotka tarjoavat luotettavan korkeamman tason rajapinnan järjestelmätason toiminnallisuuksiin. Kirjasto on kaikkien CLI:tä noudattavien kielten käytettävissä. Luokkakirjasto on jaettu kahteen osaan: Framework Class Library (FCL) ja Base Class Library (BCL). FCL on osajoukko koko luokkakirjastosta ja sisältää CLR rajapinnan ydinluokat. BCL taas viittaa koko .NET ohjelmistokehyksen mukana tulevan luokkakirjastoon. Se sisältää toiminnot esimerkiksi tiedostojen käsittelyyn, XML-dokumenttien käsittelyyn, tietokantojen käsittelyyn, erilaisia kommunikointi- ja tietoliikennekomponentteja ja suuren määrän käyttöliittymäkomponentteja. [8]

### 3.1.1 C#

C# on Microsoftin .NET-ympäristöön kehittämä olio-ohjelmointikieli, jonka ensimmäinen versio julkaistiin vuonna 2000. Uusin vakaa versio on 5.0. C# on suunniteltu tukemaan Common Language Infrastructure -määrittelyä. C#:n syntaksi on samankaltainen verrattuna Javaan, ja toisinaan C#:a kutsutaankin virheellisesti Javan klooniksi.

C# on vahvasti tyypitetty kieli, joten jokaiselle muuttujalle, vakiolle, arvoksi evaluoituvalle lausekkeelle, funktion parametrille ja paluuarvolle pitää määritellä tyyppi. C#:ssa on olemassa viitetyyppejä ja arvotyyppejä. Viitetyypin mukaiset muuttujat varataan roskienkeruun valvomasta keosta, ja ne sisältävät vain viitteen objektin sijaintiin muistissa. Viitetyyppejä ovat luokat, taulukot ja rajapinnat. Viitetyypit tukevat periyttämistä. Arvotyypin mukaiset muuttujat varataan pinosta ja ne sisältävät suoraan niille annetun arvon. Arvotyypit eivät tue periyttämistä. Kaikki tyypit voivat sisältää seuraavat tiedot [9]:

- Tyypin mukaisen muuttujan viemä tila.
- Tyypin mahdolliset maksimi ja minimiarvot.
- Tyypin sisältämät jäsenet (metodit, kentät, tapahtumat).
- Kantatyyppi josta tyyppi periytyy.
- Muistipaikka joka muuttujille varataan ajoaikana.
- Sallitut operaatiot.

C# sisältää monia funktionaalisten kielien ominaisuuksia, kuten lambdalausekkeet ja tuntemattomat tyypit. Lambdalauseke on paikallinen funktio, jonka voi välittää funktioiden parametrina tai paluuarvona. Lambdalausekkeet ovat erityisen hyödyllisiä LINQ-kyselyjen (Language Integrated Query) kirjoittamiseen. LINQ on Microsoft .NET-sovelluskehyksen komponentti, jonka avulla voidaan tehdä monipuolisia tietokyselyjä .NET:n tukemissa ohjelmointikielissä. [9]

Yksityisten muuttujien käsittelyä helpottamaan C# tarjoaa ominaisuudet (properties). Ominaisuus on luokan jäsen, jonka kautta yksityisen muuttujan arvo voidaan lukea tai siihen voidaan sijoittaa uusi arvo. Sijoitustoiminnallisuus voidaan jättää toteuttamatta, jolloin ominaisuutta voidaan käyttää vain arvon lukemiseen. Ominaisuuden sisällä voidaan myös suorittaa laskentaa. Ominaisuudet näyttävät julkisilta muuttujilta, mutta tarkemmin ottaen ne ovat saannin tarjoavia jäseniä (accessor). Ominaisuuksien avulla luokan dataan päästään helposti käsiksi samalla säilyttäen metodien turvallisuus ja joustavuus. [9]

C# tukee reflektiota, joka on metaohjelmointitekniikka. Reflektion avulla ohjelma pystyy tarkastelemaan ja muokkaamaan rakennettaan ja toimintaansa (arvoja, metadataa, ominaisuuksia ja funktioita) ajonaikaisesti. Reflektiolla pystytään luomaan uusia tyyp-

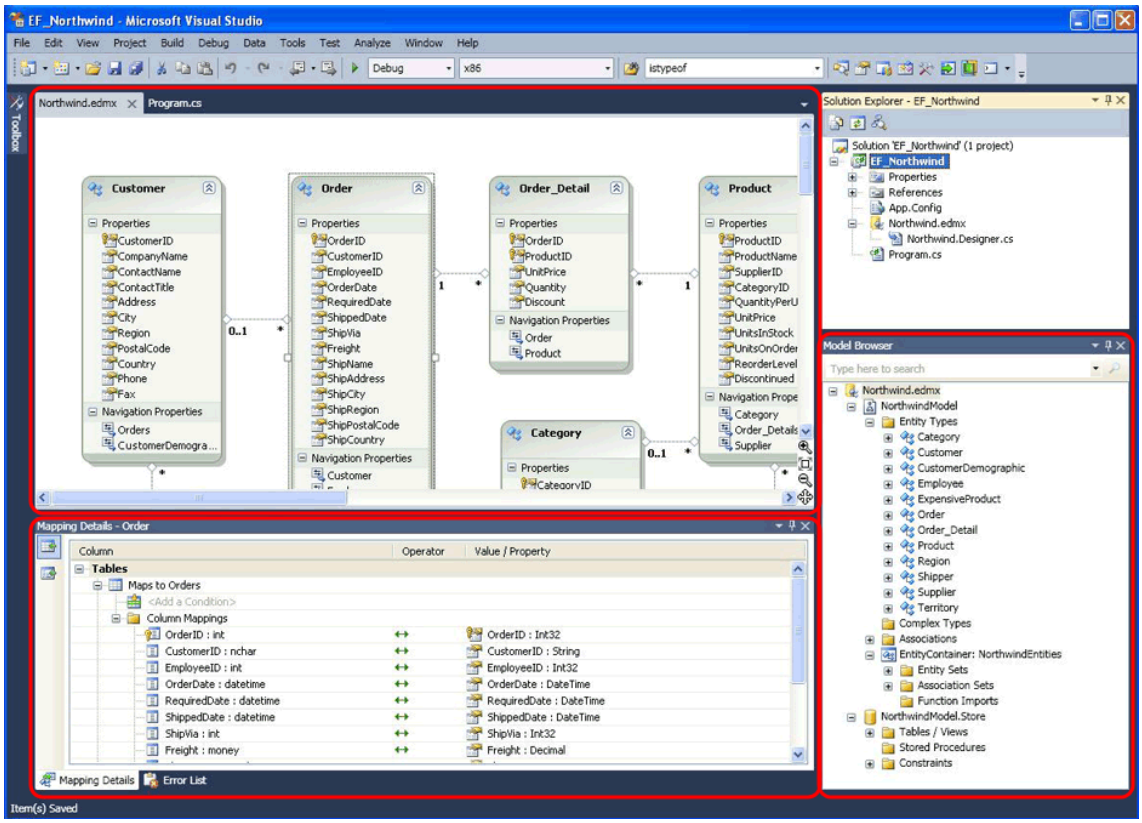
pejä dynaamisesti ajoaikana. Kyseistä ominaisuutta on käytetty tässä työssä Audit Trailin Entity Framework -toteutuksessa. [9]

### 3.1.2 Visual Studio -kehitysympäristö

Visual Studio on Microsoftin kehittämä kehitysympäristö. Se tukee useita ohjelmointikieliä, kuten Visual Basicia, C++:aa, C#:a ja F#:a. Visual Studiossa on panostettu erityisesti graafisten käyttöliittymien luomiseen, mikä tekee siitä houkuttelevan vaihtoehdon näyttäviä graafisia käyttöliittymiä sisältävien ohjelmien kehittämiseen. Visual Studiolla voidaan kehittää Windows ohjelmia, web-sivuja, web-sovelluksia ja serverillä ajettavia web-palveluja. Tarkempaa tietoa Visual Studion tarjoamista ominaisuuksista ja toiminnoista löytyy Visual Studion ohjesivustolta [10].

Visual Studioon on saatavilla paljon erilaisia laajennusosia. Niiden asentaminen ja versioiden hallinta useamman kehittäjän projektissa voi osoittautua haastavaksi, mikäli paketit asennetaan käsin. Helpotusta tilanteeseen tuo NuGet-paketinhallintakomponentti. NuGetin avulla lisäosia on helppoa etsiä ja asentaa. NuGet kopioi asennettavan lisäosan kirjastot projektiin ja päivittää automaattisesti muun muassa projektin asetustiedoston ja viittaukset asennettuun kirjastoon. NuGetin avulla kaikkien projektissa olevien kehittäjien kehitysympäristöt pysyvät helposti ajan tasalla. [10]

Visual Studioon integroituvalla ADO.NET Entity Data Model Designer -editointityökalulla pystytään luomaan ja muokkaamaan visuaalisesti Entity Frameworkin käyttämiä entiteettimalleja. Editorilla voidaan luoda ja muokata entiteettejä, entiteettien välisiä assosiaatioita, kuvauksia (mapping) ja periytymissuhteita. Entiteettimalleista on kerrottu tarkemmin alakohdassa 3.3.2. Editointityökalun päänäkymä on esitelty kuvassa 3.2. Työkalu koostuu kolmesta osasta: visuaalisesta tietomallin muokkaustyökalusta, kuvaustietojen (mapping details) muokkausikkunasta ja malliselain (model browser) -ikkunasta. Visuaalisella muokkaimella luodaan, muokataan ja poistetaan entiteettejä ja niiden välisiä liitoksia. Kuvaustietojen muokkausikkunassa voidaan määritellä miten entiteetit kuvautuvat tietokannan tauluiksi, sarakkeiksi ja proseduureiksi. Malliselain taas tarjoaa puunäkymän entiteettien konseptimallista ja tietokantatason varastomallista. Entity Designer käyttää pohjana edmx-tiedostoa, joka koostuu kolmesta metatietoa sisältävästä tiedostosta: conceptual schema definition language (CSDL), store schema definition language (SSDL) ja mapping specification language (MSL).




*Kuva 3.2 Entity Data Model Designer*

## 3.2 Microsoft SQL -tietokannan hallintajärjestelmä ja herättimet

Toteutuksessa käytettiin tietokantana Microsoft SQL Server -tietokantaa. Käyttöön olisi voitu valita minkä tahansa valmistajan tietokannan hallintajärjestelmä, sillä Entity Framework on suunniteltu tukemaan useita erilaisia tietokantoja. Microsoft SQL on kuitenkin luonnollinen valinta, koska toteutuksessa käytetään muiltakin osin Microsoftin tuotteita. Tärkein syy Microsoft SQL Serverin valintaan ovat asiakasvaatimukset.

Microsoft SQL Server on Microsoftin kehittämä relaatiotietokannan hallintajärjestelmä. SQL Serveristä on useita versioita, jotka on kohdennettu eri käyttäjäryhmille käyttötarpeiden mukaan. Tässä työssä käytetään SQL Serverin Enterprise-versiota, joka sisältää tietokannan ytimenä toimivan moottorin ja erilaisia lisäpalveluita. Tämän työn puitteissa oleellisin osa on tietokantamoottori. Tietokantamoottori on tietokannan ydinpalvelu, joka vastaa tiedon tallettamisesta, prosessoinnista ja turvallisuudesta. Relaatiotietokannan ja sen sisältämien taulujen, indeksien, näkymien ja tallennettujen proseduurien luominen kuuluu ydinpalveluihin. SQL Server 2012 tukee maksimissaan 524 272 teratavun kokoista tietokantaa. [11]

SQL Server sisältää tuen herättimille (trigger). Herättimellä tarkoitetaan ohjelmakoodia, jonka tietokannan hallintajärjestelmä suorittaa automaattisesti tiettyjen tapahtumien

kohdalla. Herättimien avulla voidaan esimerkiksi tarkistella tietokantaan lisättävän tiedon oikeellisuutta ja huolehtia tietojen yhteneväisyydestä.

SQL Server sisältää tuen kahdenlaisille herättimille. Data Definition Language -tapahtumiin reagoidaan DDL-herättimillä. Näitä tapahtumia ovat pääasiassa SQL-lauseet, jotka alkavat avainsanoilla *CREATE*, *ALTER*, *DROP*, *GRANT*, *DENY*, *REVOKE* tai *UPDATE STATISTICS*. DDL-herättimien avulla voidaan estää muutosten tekeminen tietokannan skeemaan, tehdä tietokantaan muutoksia skeeman muuttuessa ja tallettaa tietokannan skeemaan liittyviä muutoksia ja tapahtumia.

Data Manipulation Language -tapahtumien yhteydessä käytetään DML-herättimiä. Tällaisia tapahtumia ovat *INSERT*-, *UPDATE*- ja *DELETE*-avainsanoilla alkavat lausekkeet. DML-herättimiä voidaan käyttää bisnessääntöjen ja datan yhteneväisyyden varmistamiseen, kyselyjen tekemiseen toisiin tietokantatauluihin ja monimutkaisten Transact-SQL-lauseiden sisällyttämiseen. [11]

Tässä työssä on käytetty DML-herättimiä toisen vertailukohteena olevan Audit Trail -mekanismin toteuttamiseen. Herättimien avulla tietokantaan voidaan luoda jokaiselle taululle omat taustasäännöt, jotka huolehtivat Audit Trail -datan kirjoittamisesta aina kun päätietokannan taulujen tietoihin tehdään lisäys-, päivitys- tai poisto-operaatioita.

DML-herättimiä on kahta eri tyyppiä: tietokantaoperaation sijaan suoritettava herätin (*INSTEAD OF* trigger) ja tietokantaoperaation jälkeen suoritettava herätin (*AFTER* trigger). Tietokantaoperaation sijaan suoritettava herätin syrjäyttää tietokantakomennon joka laukaisi herättimen. Alkuperäinen komento voidaan suorittaa kirjoittamalla se herättimen logiikkaan. Tietokantaoperaation jälkeen suoritettava herätin taas suoritetaan herättimen laukaisevan tietokantakomennon jälkeen. *AFTER*-tyyppisen herättimen voi luoda vain tietokannan tauluun. *INSTEAD OF* -herättimiä voidaan luoda myös näkymiin.

Herättimet toimivat tietokantatransaktion sisällä ja pidentävät sen kestoa. Transaktion sisältämät tietokantamuutokset tulevat voimaan vasta herättimen onnistuneen suorittamisen jälkeen. Mikäli herättimen suoritus epäonnistuu, tietokannan tila palautetaan (rollback) siihen tilaan, missä se oli ennen transaktion alkua.

Herättimien toiminta perustuu virtuaalitauluihin, jotka sisältävät tietokantaoperaation kohteena olevasta datasta kaksi versiota. Ensimmäinen versio sisältää datan siinä muodossa missä se on ennen muutosta ja toinen muuttuneen version datasta. Virtuaalitaulujen nimet ovat lisätty (inserted) ja poistettu (deleted). Virtuaalitaulujen dataa pitäisi käsitellä aina kokonaisena joukkona. Herättimiä ei saa kirjoittaa sillä oletuksella, että tietokantaoperaatio kohdistuisi kerrallaan vain yhteen taulun riviin.

## 3.3 Olio-relaatio-kuvaus

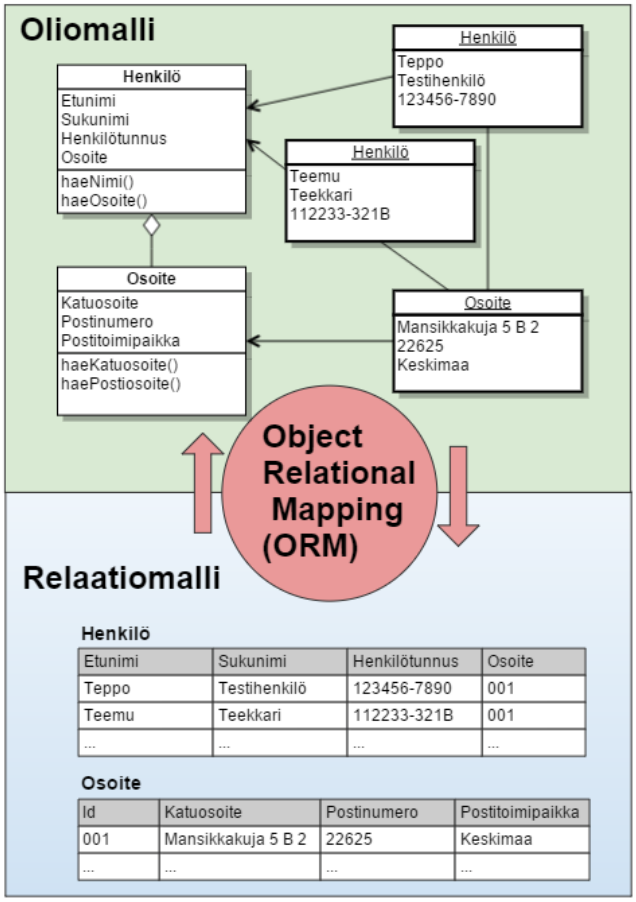
Sovelluksien tai palvelujen toteutuksessa on pitkään voimassa ollut käytäntö jakaa sovellus tai palvelu kolmeen osaan: domain-malliin, loogiseen malliin ja fyysiseen malliin. Domain-malli kuvaa mallinnettavan sovelluksen oliot ja niiden väliset suhteet. Relaatiotietokannan looginen malli normalisoi entiteetit ja suhteet vierasavainviittaukset sisältäväksi tietokannan tauluiksi. Fyysinen malli käsittelee käytettävän tietokantamoottorin kapasiteetin määrittelemällä tietovaraston tarkemmat tiedot, kuten partitioinnin ja indeksoinnin. Ohjelmistokehittäjät työskentelevät yleensä loogisen mallin tasolla kohdistamalla tietokantaan SQL kyselyjä ja kutsumalla tietokannan tallennettuja proseduureja. Tietokannan tietomallin ja domain-mallin lähentämiseksi on kehitetty olio-relaatio-kuvauksen toteuttavia olio-relaatio-muuntimia. Näiden kahden tietomallin välillä on merkittäviä eroavaisuuksia, jotka aiheuttavat omat haasteensa muunnokselle. [12]

Relaatiomallinnus kuvaa tiedon predikaattilogiikkana ja totuuslausekkeina. Relaatiotietokannalle annetaan looginen malli ja sille kerrotaan totuusväittämiä. Näiden perusteella relaatiotietokanta pystyy tallentamaan alkuperäisen tiedon ja johtamaan niistä uusia totuuksia. Jotta tietokannan kanssa pystyttäisiin kommunikoimaan, tarvitaan taulukon 3.2 mukainen hyvin määritelty termistö.

*Taulukko 3.2. Relaatiomallinnuksen termit ja niiden vastineet tietokannassa*

| **Relaatiomallin termi** | **Selite** | **Vastine tietokannassa** |
|---|---|---|
| Relaatio | Rakenteeltaan samanlaisten monikoiden joukko. | Taulu |
| Monikko | Järjestetty arvojen lista, jossa on yksi arvo kutakin relaatiokaaviossa nimettyä attribuuttia kohden. | Rivi |
| Ominaisuus | Otsake, joka ilmaisee arvon merkityksen relaation monikossa. | Sarake |
| Arvoalue | Äärellinen tai (teoriassa) ääretön arvojen joukko, johon ominaisuuden arvot kuuluvat. | Tietotyyppi |
| Ominaisuuden arvo | Tiettyyn monikkoon kuuluvan ominaisuuden arvo, jonka pitää kuulua arvoalueeseen. | Sarakkeen arvo tietyllä monikolla |

Oliomallinnuksessa järjestelmä kuvataan olioista rakentuneena kokonaisuutena. Olioilla on identiteetti, tila ja sisäistä toiminnallisuutta. Lisäksi oliomallinnukseen liittyy monia käsitteitä, kuten abstraktio, samanlaisuus (similarity), periytyminen ja modulaarisuus. Identiteetti erottaa olion kaikista muista olioista riippumatta olioiden tiloista. Olion tila kuvaa tietyn identiteetin omaavan olion sen hetkisen arvon. Sisäinen toiminnallisuus koostuu operaatioista, jotka voivat muokata olion tilaa ja palauttaa tietoa kutsujalle. Operaatiot muodostavat rajapinnan, jonka välityksellä olion ulkopuoliset komponentit voivat olla vuorovaikutuksessa olion kanssa. Yksi tapa toteuttaa olioita on luoda luokka, joka määrittelee toteutuksen kaikille siitä johdettaville olioille. [13]




*Kuva 3.3. Olio-relaatio kuvaus*

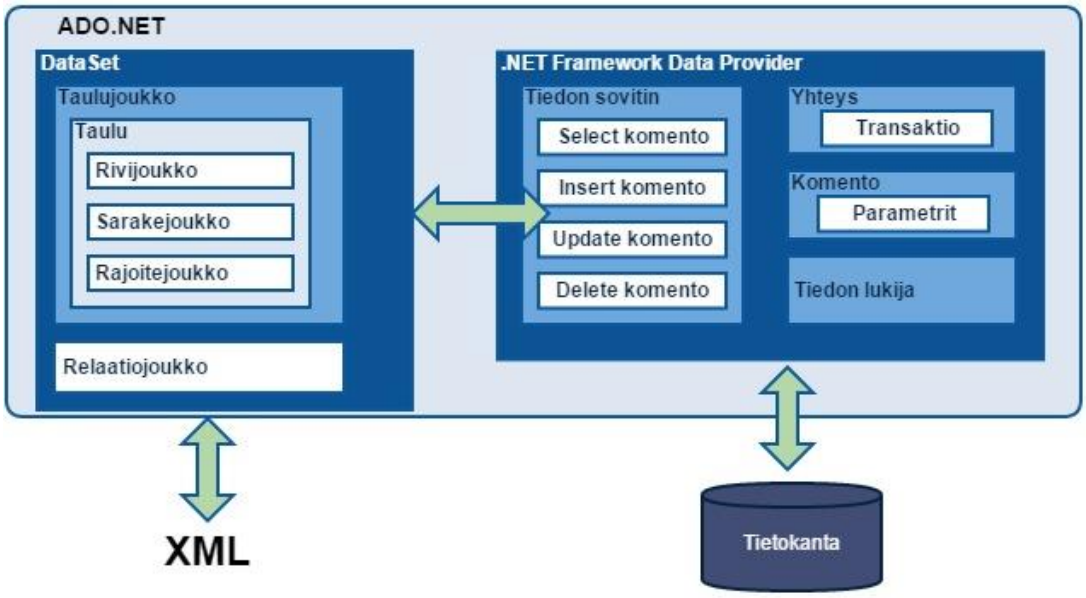
Olio-relaatio-kuvauksella (ORM, object-relational mapping) tarkoitetaan prosessia, jossa liikutaan oliomallinnuksen ja relaatiomallinnuksen välillä (kuva 3.3). Ihannetapauksessa yksi kokonaisvaltainen malli kuvaisi oliomallinnus- ja relaatiomallinnuslähestymistavat. Todellisuudessa oliomallin ja relaatiomallin toteuttavien järjestelmien toteutukset ovat puutteellisia tai epäjohdonmukaisia teoreettisiin lähestymistapoihin verrattuna. Relaatiotietokannat ovat vuosikymmenien ajan toteuttaneet relaatioteorian ydinkäsitteet puutteellisesti. Oliomallinnusta taas ei ole standardisoitu, jolloin jokainen ohjelmointiympäristö toteuttaa oman versionsa mallinnuksesta. Olio-relaatio-kuvauksen avulla oliot pystytään muuttamaan ja tallentamaan relaatiotietokannan taulujen riveiksi ja kyselyjen kohdalla palauttamaan takaisin olioiksi. [13]

### 3.3.1 ADO.NET

Microsoft ADO.NET on osa .NET Frameworkia oleva joukko ohjelmistokomponentteja, jotka helpottavat sovelluksen ja tietovaraston välistä kommunikointia. ADO.NET on ADO-tekniikan (ActiveX Data Objects) seuraaja. ADO:n heikkous on se, että sitä kutsutaan pääasiassa .NET:n hallitsemattoman koodin (unmanaged code) kautta. Hallitsemattomassa koodissa on monia huonoja puolia, kuten hitaampi suoritus verrattuna hallittuun koodiin (managed code), .NET:n tarjoaman turvallisuuden menetys ja roskienke-

ruun heikompi toiminta, mikä voi aiheuttaa muistivuotoja. Datakerros on usein sovelluksen kriittisimpiä osia tehokkuuden ja luotettavuuden osalta, joten muistivuodot tällä tasolla eivät ole hyväksyttäviä. ADO ei myöskään tue kunnolla XML:ää. ADO.NET mahdollistaa kommunikoinnin tietokannan kanssa täysin irrallisen data välimuistin (data cache) kautta, jolloin dataa voidaan käsitellä yhteydettömästi. [14]

ADO.NET:n arkkitehtuuri voidaan jakaa karkeasti kahteen osaan: yhdistettyihin (connected) ja irtikytkettyihin (disconnected) objekteihin. Yhdistetyn osan objektit vaativat jatkuvan avoimen yhteyden tietolähteeseen. Irtikytkettyjen objektien ansiosta on mahdollista rakentaa hajautettuja ohjelmistoja, jossa sovellus ottaa yhteyden tietolähteeseen mahdollisimman myöhään ja  katkaisee yhteyden mahdollisimman aikaisin. ADO.NET kierrättää varsinaisia fyysisiä tietokantayhteyksiä eri pyyntöjen välillä. Yhdistetyn osan osakomponentit yhdessä muodostavat tietoa tarjoavan pääkomponentin (Data Provider). Irtikytketyn osan tärkein osa on tietoa säilövä komponentti (DataSet), joka säilöö tietokannan sisältöä välimuistissa. ADO.NET:n arkkitehtuuri on esitetty kuvassa 3.4. [14]




*Kuva 3.4 ADO.NET -arkkitehtuuri [15]*

DataSet-komponentti voidaan ajatella kokonaan erillisenä keskusmuistissa sijaitsevana pienoistietokannan hallintajärjestelmänä. Se voidaan ajatella myös loogisena tietokannan tauluja ja relaatioita sisältävänä kokonaisuutena. DataSet ei ole suoraan yhteydessä tietokantaan, vaan se käyttää sovitinta (Data Adapter) tietojen hakemiseen. DataSetin avulla tietokannan tietoa voidaan lukea välimuistiin niin, että se on myös muokattavissa ja tallennettavissa takaisin tietokantaan. DataSetin sisältämä tieto voidaan jäsentää XML:ksi, mikä helpottaa sovellusten välistä kommunikointia. Koska DataSet ei kommunikoi suoraan tietokannan kanssa, se mahdollistaa sovelluksen tietokantariippumattoman toteutuksen. [14]

DataProvider -komponentti on suunniteltu kommunikoimaan tietokannan kanssa. Se huolehtii tiedon lukemisesta, päivittämisestä ja kirjoittamisesta. DataProviderin yhteysolio (connection) tarjoaa yhteyden tietokantaan. Komento-olio (command) mahdollistaa tiedon palauttamisen, muokkaamisen, tallennettujen proseduurien suorittamisen ja parametritietojen välityksen. Tiedon lukija (DataReader) tarjoaa nopean tiedonlukutoiminnallisuuden tietolähteestä. Tiedon sovitin (DataAdapter) toimii siltana DataSetin ja tietolähteen välillä. Erilaisia tietokantoja ja muita tietolähteitä on olemassa suuri joukko. Jotta ADO.NET voisi tukea suurta määrää erilaisia tietolähteitä, pitää jokaiselle eri tietolähteelle luodaan oma DataProvider -komponentti. [15]

ADO.NET tarjoaa monia hyötyjä verrattuna muihin tiedonhakukomponentteihin ja aikaisempiin versioihin ADO:sta:

- **Yhteentoimivuus**. XML tarjoaa joustavuutta ja se on laajasti käytetty formaatti.
- **Ylläpidettävyys**. ADO.NET DataSetit tarjoavat helpon tavan muokata sovelluksen arkkitehtuuria esimerkiksi jakamalla palvelinkerros useampaan tasoon.
- **Ohjelmoitavuus**. ADO.NETin komponentit kapseloivat tiedonhakutoiminnallisuutta monella tavalla, mikä helpottaa ohjelmointia.
- **Suorituskyky**. ADO.NET DataSetit ovat nopeampia kuin ADO:n RecordSetit, koska ylimääräisiä tietotyypin muunnoksia ei tarvitse tehdä.
- **Skaalautuvuus**. Kyky palvella kasvavaa käyttäjäjoukkoa heikentämättä tietolähteen suorituskykyä.

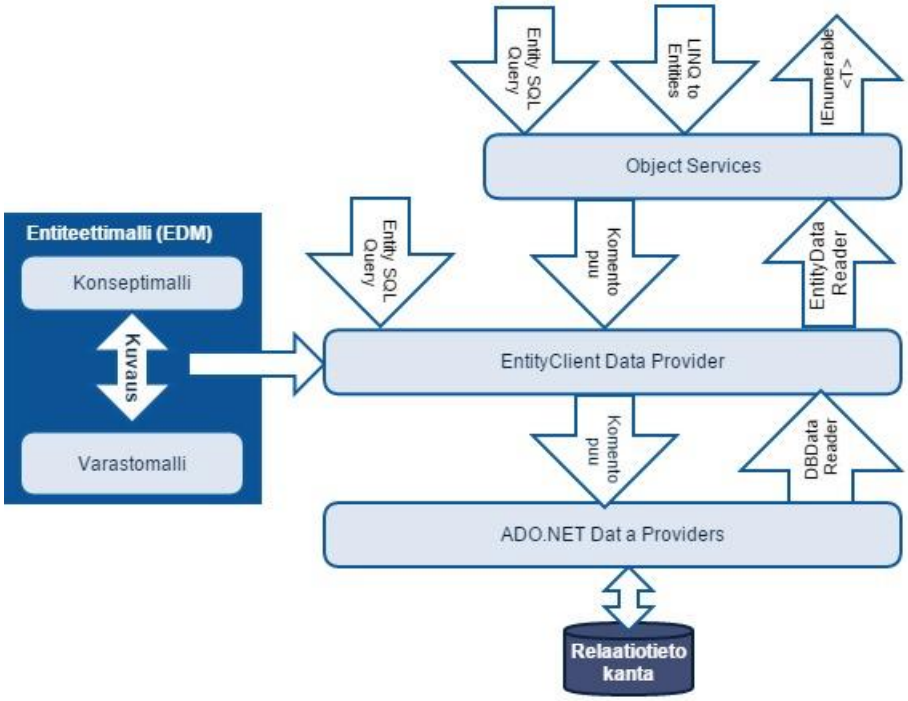
ADO.NET ei kuitenkaan ole paras mahdollinen vaihtoehto tietokantakerroksen abstrahointiin. Olio-relaatio-kuvauksen toteutus vaatii mukautetun ohjelmakoodin kirjoitusta. Tämä hankaloittaa ylläpidettävyyttä ja uudelleenkäytettävyyttä. Lisäksi ADO.NET:iä käyttävän koodin pitää olla myös CLR:n alla suoritettavaa.

### 3.3.2 Entity Framework

Microsoft ADO.NET Entity Framework on ADO.NET-mallin päälle rakennettu olio-relaatio-kuvauksen (ORM, Object Relational Mapping) toteuttava ohjelmistokehys. Entity Framework on osa .NET-ohjelmistokehystä. Näin ollen Entity Frameworkia voidaan käyttää missä tahansa sovelluksessa, jonka kehitysympäristö sisältää .NET ohjelmistokehyksen version 3.5 SP1 tai uudemman. Entity Frameworkin avulla relaatiomallin dataa ei tarvitse käsitellä tietokannan tasolla, vaan ohjelmoija voi käsitellä dataa domain-alueen olioina. Entity Framework suorittaa muunnoksen tietokantatauluista domain alueen olioiksi ja päinvastoin, jolloin päästään eroon yksittäiselle tietolähteelle kovakoodatuista tietokantaoperaatioista. Kyselyt suoritetaan LINQ -kyselyinä, jotka palauttavat datan vahvasti tyypitettyinä olioina. Entity Frameworkin arkkitehtuuri ja pääkomponentit on esitetty kuvassa 3.5. [16]

Entity Framework tarjoaa ohjelmistokehittäjälle useita hyödyllisiä ja helpottavia ominaisuuksia, mutta kaikkein tärkeimmät ominaisuudet ovat seuraavat [16]:

- Entity Framework luo automaattisesti käytettävät luokat tietomallin pohjalta ja päivittää luokat dynaamisesti aina kun tietomalliin kohdistuu muutoksia.
- Tietokantayhteyksien hallinta on Entity Frameworkin vastuulla, jolloin käyttäjän ei tarvitse aina tietokantaa käyttäessä miettiä yhteyksien luomista.
- Entity Framework tarjoaa yleisen syntaksin kyselyjen kohdistamiseksi tietomalliin ja muuntaa kyselyt tietokannan ymmärtämään muotoon.
- Entity Framework tarjoaa tietomallin olioille seurantamekanismin (Change Tracking), jonka avulla olioihin kohdistuneet muutokset saadaan jäljitettyä, ja joka huolehtii muutosten päivittämisestä myös tietokannan tasolle.




*Kuva 3.5 Entity Framework arkkitehtuuri ja pääkomponentit*

## ARKKITEHTUURI

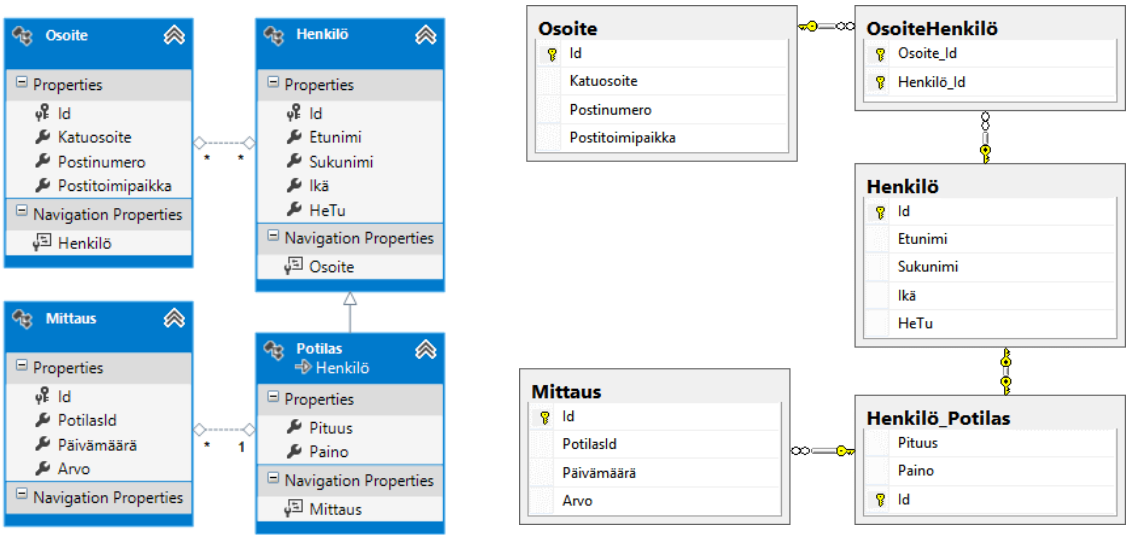
Entity Frameworkin rakentuu ADO.NET datantarjoajamallin päälle. ADO.NET tarjoaa kommunikointirajapinnan useisiin erilaisiin tietolähteisiin ja mahdollistaa näin tietokantariippumattomuuden. EntityClient Data Provider on ADO.NET-datantarjoajan päälle rakennettu laajennos, joka käsittelee dataa entiteettimallin määrittelemiä entiteettejä ja niiden välisiä suhteita vasten. Tietokantaan kohdistuvat kyselyt ja päivitykset tehdään käyttäen uutta entiteettipohjaista Entity SQL -kyselykieltä. Entity Client Data Providerin päätehtävä on kääntää LINQ to Entities tai Entity SQL kyselyt taustalla toimivan tietokannan ymmärtämään muotoon. [17]

Object Services -komponentti tarjoaa monipuolisen entiteetteihin pohjautuvan olioabstraktion, kyseisiin olioihin liittyvät monipuoliset palvelut ja mahdollistaa sovellusten tekemisen käyttäen tuttuja ohjelmointikielen rakenteita. Komponentti on vastuussa materialisoinnista eli se muuntaa EntityClient Data Providerilta tulevan datan entiteettiolioiden muotoon. Object Services tarjoaa olioiden tilan seurantaan käytetyt palvelut, kuten muutosten seurannan, tukee olioiden ja niiden välisten suhteiden lataamista ja navigointia, tukee LINQ to Entities- ja Entity SQL -kyselyjä ja mahdollistaa olioiden tilan päivittämisen ja säilyttämisen. Object Services tarjoaa EntityObject-luokan, josta kaikki entiteettioliot periytyvät. Tämän avulla Object Services pystyy hallitsemaan entiteettiolioita. Uutuutena Entity Framework tarjoaa .NET 4 versiosta lähtien myös POCO (Plain Old CLR Objects) -tuen. POCO-entiteettien ei tarvitse periytyä EntityObject-luokasta. [17]

LINQ to Entities on kyselykieli, jonka avulla kirjoitetaan kyselyitä oliomallia vasten. Entity Frameworkia vasten tehdyt kyselyt esitetään komentopuukyselyinä, jotka suoritetaan object contextia vasten. Komentopuu on oliomallitason esitys tietokantatason kyselyistä. LINQ to Entities muuntaa LINQ-kyselyt komentopuumuotoon, suorittaa kyselyt Entity Frameworkia vasten ja palauttaa entiteettimallissa määriteltyjä entiteettiolioita, joita sekä Entity Framework että LINQ pystyvät käyttämään. Entity SQL on vaihtoehto LINQ to Entities -kyselykielelle. Se on tietokantariippumaton ja muistuttaa syntaksiltaan perinteisiä SQL-kyselyjä. Entiteettidataan kohdistuvien kyselyjen tulokset voidaan palauttaa joko olio- tai taulukkomuodossa. Entity SQL:ää kannattaa käyttää, jos kyselyt pitää muodostaa ajonaikaisesti, halutaan määritellä kyselyt osaksi tietomallia, palautetaan dataa vain-luku -muodossa käyttäen EntityDataReaderia, tai perinteiset SQL-kyselyt ovat käyttäjälle entuudestaan tuttuja. Entity SQL on hieman vaikeampi käyttää kuin LINQ to Entities ja se palauttaa erityyppisen kyselyvastauksen.

### ENTITEETTIMALLI

Entity Frameworkin ydinosa on asiakasohjelman tietomallin sisältävä entiteettimalli (Entity Data Model, EDM). Entiteettimallia ei pidä sekoittaa tietokannan tietomalliin. Entiteettimalli kuvaa sovellustasolla käytettävien olioiden väliset rakenteet, kun taas tietokannan tietomalli sisältää tietokantatason normalisoidun skeeman. Nämä kaksi mallia voivat erota toisistaan melko paljon. Kuvasta 3.6 käy ilmi tietokantaskeeman ja entiteettimallin välinen ero. Periytymissuhde kuvataan tietokannassa *Henkilö_Potilas* taulun Id-kentällä, joka on vierasavainviittaus *Henkilö*-taulun *Id*-kenttään. *Osoite*- ja *Henkilö*-taulujen välinen monesta-moneen-suhde toteutuu tietokannan tasolla välitaululla, jonka *Osoite_Id-Henkilö_Id*-arvoparit kuvaavat kaikki *Osoite*- ja *Henkilö*-taulujen väliset suhteet.




*Kuva 3.6 Entiteettimallin ja tietokannan väliset eroavaisuudet*

Entiteettimalli koostuu kolmesta osasta: konseptimallista (conceptual model), varastomallista (storage model) ja näiden välisestä kuvauksesta (mapping). Konseptimalli edustaa varsinaista entiteettimallia. Se sisältää entiteettimallin malliluokat ja niiden väliset suhteet. Varastomalli edustaa varsinaisen taustalla olevan tietokannan skeemaa. Varastomallia nimitetään myös loogiseksi malliksi. Konseptimallin ja varastomallin välisessä kuvauksessa määritellään miten konseptimallin entiteetit ja niiden ominaisuudet kuvautuvat varastomallin tauluiksi ja sarakkeiksi. Kaikki kolme mallia on määritelty omassa erillisessä XML-pohjaisessa tiedostossa.

Entiteettimallin sisältävillä entiteeteillä on skalaarimuotoisia ominaisuuksia (property) ja suhteita eli assosiaatioita. Skalaariominaisuuksien arvot on talletettu suoraan entiteettiin. Esimerkkinä skalaariominaisuudesta voidaan mainita esimerkiksi kuvassa 3.6 *`Henkilö`*-entiteettiin kuuluvat kentät *`Etunimi`*, *`Sukunimi`* ja *`Ikä`*. Assosiaatiot määrittelevät entiteettien väliset suhteet ja sen miten monta entiteettiä kunkin suhteen eri päissä voi olla. Assosiaatiot voivat olla joko vierasavain-assosiaatioita (navigation property) tai vierasavain-ominaisuuksia (foreign key property). Vierasavain-assosiaatiot sisältävät viitteen toiseen kokonaiseen entiteettiin, jolloin esimerkiksi kuvan 3.6 tapauksessa päästään navigoimaan *`Henkilö`*-entiteetistä *`Osoite`*-entiteettiin. Vierasavain-ominaisuus taas sisältää toisen entiteetin pääavaimen. Se talletetaan suoraan yhdeksi viittaavan entiteetin kentäksi. Tällöin esimerkiksi kuvan 3.6 tapauksessa *`Henkilö`*-entiteettejä ei tarvitse hakea välttämättä ollenkaan, koska *`Mittaus`*-entiteetit sisältävät potilaan pääavaimen.

Tietokantojen rakenne ei suoraan tue oliomaailmassa laajasti käytettyä periyttämistä. Entity Framework sen sijaan mahdollistaa periyttämisen käytön entiteettimallissa ja huolehtii periytymisrakenteiden luomisesta tietokannan tasolla. Entiteettimallin tasolla periytyminen näkyy samalla tavalla kuin perinteisillä olioillakin eli kantaluokan ominaisuudet periytyvät kaikkiin lapsiluokkiin. Tietokannan tasolla periytyminen voidaan to-

teuttaa kolmella eri tavalla: yksi taulu periytymishierarkiaa kohden (TPH, table-per-hierarchy), yksi taulu tyyppiä kohden (TPT, table-per-type) tai taulu yhtä konkreettista luokkaa kohden (TPC, table-per-concrete class). [18]

Oletuksena Entity Framework käyttää ensimmäistä eli TPH-tapaa. Siinä kaikkien periytymishierarkian olioiden tiedot tallennetaan yhteen tietokannan tauluun. Periytymishierarkiassa olevat eri tyyppiset oliot erotellaan lisäämällä tauluun tyyppierottelusarake, joka sisältää kunkin olion tyypin. Yhden taulun käytön hyviä puolia ovat tehokkuus ja yksinkertaisuus. Heikkoutena yhden taulun mallissa on jokaisen periytymishierarkiaan kuuluvan olion kenttien tallentaminen jokaiselle tietokantataulun riville. Mitä suurempi periytymishierarkia on kyseessä, sitä enemmän yhdelle riville jää tyhjiä käyttämättömiä kenttiä. Taulun muokkaaminen on myös työlästä, mikäli periytymishierarkiaan tulee muutoksia. [18]

Toisessa eli TPT-tavassa tietokantaan luodaan oma taulunsa vastaamaan jokaista periytymishierarkian luokkaa ja periytymissuhteet esitetään relaatiotietokannan vierasavain-viittausten avulla. Etuna on tietokannan taulujen normalisoinnin säilyminen, periytymishierarkian muutosten toteuttaminen ja polymorfiset suhteet kantaluokan ja periytymishierarkian ulkopuolisten luokkien välillä. [18]

Kolmannessa eli TPC-tavassa periytymishierarkia toteutetaan niin, että jokaista ei-abstraktia luokkaa kohden luodaan tietokantaan taulu. Kyseiseen tauluun talletetaan myös kaikkien kantaluokkien kentät. [18]

Käsitteellinen malli voidaan luoda kolmella eri tavalla: Database First, Model First ja Code First. Jokaiselle vaihtoehdolle on oma käyttökohteensa. Database First -tavassa pohjana käytetään olemassa olevaa tietokantaa, jonka pohjalta Entity Framework luo entiteettimallin. Code First on päinvastainen lähestymistapa. Ohjelmassa käytettävät konseptimallin oliot kirjoitetaan käsin ja Entity Framework luo näiden olioiden pohjalta tietokannan taulut ja assosiaatiot. Model First sijoittuu kahden edellisen väliin. Siinä tietomalli suunnitellaan visuaalisen suunnittelutyökalun kanssa. Visuaalisesta mallista luodaan entiteettimalli ja SQL-komennot sisältävä tiedosto, joka sisältää komennot tietokannan skeeman luomiseen. Varsinainen tietokanta täytyy luoda ensin käsin, minkä jälkeen tiedoston sisältö voidaan suorittaa esimerkiksi käyttäen Microsoft SQL Server Management Studiota. Tämän diplomityön käsittelemissä Audit Trail -mekanismin toteutuksissa käytetään Model First -lähestymistapaa tietomallin suunnittelemiseen ja toteuttamiseen.

### PALVELUT

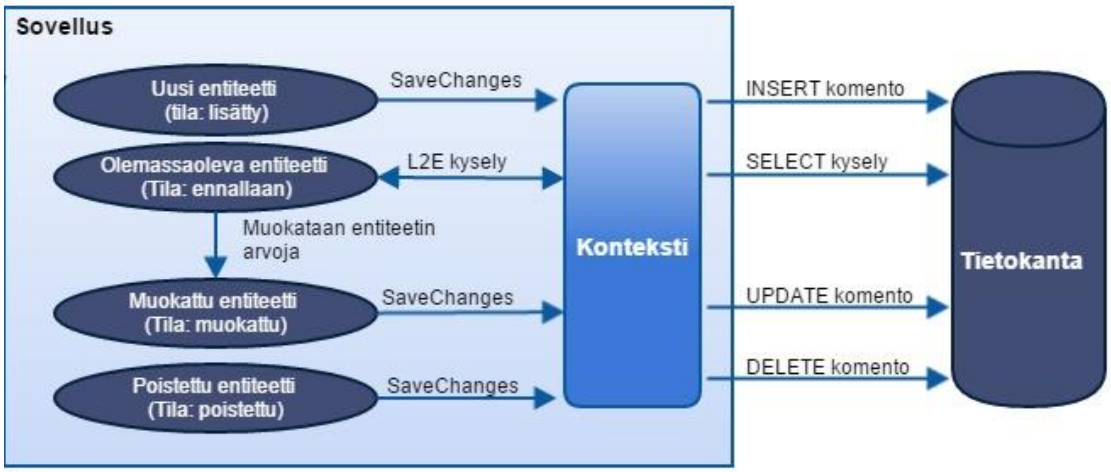
Entity Framework tarjoaa useita tiedon hakua ja muokkaamista helpottavia palveluja, kuten muutosten seuranta (Change Tracking), identiteetin selvitys (Identity Resolution), laiska lataus (Lazy Loading) ja kyselyn kääntäminen (Query Translation). Palveluiden

ansiosta ohjelmoija voi keskittyä sovelluksen bisnestason logiikkaan huolehtimatta tietokantatason rakenteista.

Yksi Entity Frameworkin keskeisistä käsitteistä on konteksti. Aiemmissa Entity Frameworkin versioissa kontekstin toteutti luokka `ObjectContext`. Uudemmissa versioissa toteuttava luokka on `DbContext`, joka käärii `ObjectContext` luokan sisälleen ja on `ObjectContextia` helpompi käyttää. Tässä työssä kontekstilla tarkoitetaan `DbContextia`. Konteksti toimii siltana tietokannan ja sovellustason entiteettien välillä. Kaikki dataan kohdistuvat operaatiot tehdään kontekstin kautta.

Muutosten seurantaan liittyy entiteetin tila, joka voi entiteetin elinkaaren aikana olla jokin seuraavista: lisätty, poistettu, muokattu, ennallaan tai irrotettu. Tila vaihtelee kontekstin kautta entiteettiin tehtyjen operaatioiden mukaan. Entiteetin tila on irrotettu ennen kuin entiteetti ladataan kontekstiin. Kuvassa 3.7 näkyvät entiteettien mahdolliset tilat ja eri tiloihin kohdistuvat operaatiot konteksti- ja tietokantatasolla. [16]

Muutosten seuranta pitää kirjaa niiden entiteettien tilasta ja arvoista, jotka on ladattu kontekstiin. Entity Framework ottaa entiteetin arvoista tilannekatsauksen kun entiteetti ladataan ensimmäistä kertaa mukaan kontekstiin. Tämä tapahtuu automaattisesti kun tietokantaan tehdään kysely ja kyselyn tuloksena palautuvat rivit muunnetaan olioiksi. Konteksti tallentaa entiteettien arvoista kaksi eri arvojoukkoa: alkuperäiset arvot (original values) ja nykyiset arvot (current values). Alkuperäiset arvot sisältävät entiteetin alkuperäiset arvot siinä tilassa, missä ne olivat haettaessa tietokannasta. Alkuperäiset arvot eivät muutu. Nykyiset arvot muuttuvat sen mukaan, mitä muutoksia olion tietoihin tehdään sovelluksessa. Näiden kahden arvojoukon perusteella Entity Framework osaa tallentaa olioihin tehdyt muutokset tietokantaan kutsuttaessa kontekstin metodia `SaveChanges`. [16]




*Kuva 3.7 Entiteetin tilat ja toiminnot eri tiloissa*

# 4. VERTAILUN TAUSTATIEDOT

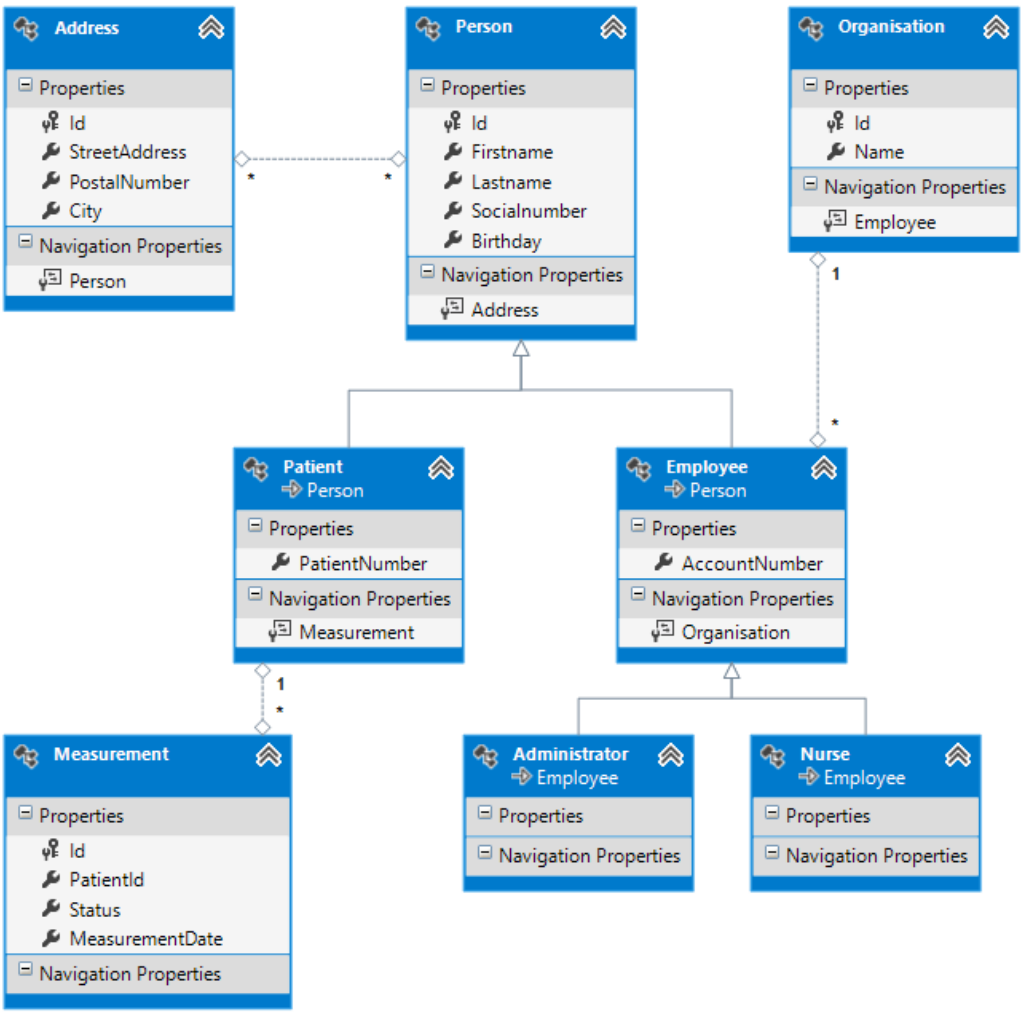
Audit Trailin toteutus jakautuu kahteen eri tasoon: tietokantaskeeman toteutukseen ja Audit Trail tietojen tallennusmekanismin toteutukseen. Tietokantaskeema voidaan toteuttaa usealla eri tavalla. Tämän diplomityön pääasiallisena vertailukohteena ovat kaksi eri toteutustapaa tietojen tallennusmekanismille. Toteutustapoina toimivat sovellustason .NET:n Entity Framework ja tietokantatason herättimet eli triggerit. Vertailussa käydään läpi Audit Trail -datan tallentamisessa ja käytössä vastaan tulevia yleisiä tilanteita ja arvioidaan toteutustapojen hyviä ja huonoja puolia. Vertailussa arvioidaan kummankin toteutustavan toteutettavuutta, suorituskykyä, ylläpidettävyyttä ja uudelleenkäytettävyyttä. Kohdassa 4.1 käydään läpi tarkemmin kaksi erilaista tietokantaskeeman toteutusta ja listataan muita mahdollisia vaihtoehtoja. Kohdassa 4.2 käydään läpi toteutettavuuden arviointikriteereitä. Kohdassa 4.3 tarkistellaan suorituskyvyn merkitystä Audit Trailin toteuttamisessa. Kohdassa 4.4 kerrotaan ylläpidettävyyden arviointiperusteista ja kohdassa 4.5 sovelluskomponenttien uudelleenkäytettävyydestä ja sen arvioimisesta.

## 4.1 Vertailussa käytettävät tietomallit

Tietomallilla on suuri vaikutus Audit Trailin käyttökelpoisuuteen ja tehokkuuteen. Näin ollen tietomalli on syytä suunnitella alusta alkaen niin, että se täyttää halutut vaatimukset ja tukee tietojen jäljitettävyyttä parhaalla mahdollisella tavalla. Tässä diplomityössä tarkastellaan tarkemmalla tasolla kahta erilaista tietomallisuunnitelmaa. Näiden lisäksi käydään pintapuolisemmin läpi muutamia muita mahdollisia suunnittelumalleja ja listataan niiden hyviä ja huonoja puolia.

Varsinaisen päätietokannan tietomallina käytetään kuvan 4.1 mukaista esimerkkitietomallia, joka esittää yksinkertaista potilastietojärjestelmää. Tietomalli sisältää *Henkilö*-luokan, josta periytyy *Potilas*- ja *Työntekijä*-luokat. Työntekijästä periytyy lisäksi työntekijän alaluokat *Ylläpitäjä* ja *Hoitaja*. Periytymistä on käytetty paljon, jotta Audit Trailin toimivuus myös näissä tapauksissa saadaan testattua ja todettua. Potilas- ja Mittaus-luokkien välillä on yhdestä-moneen-suhde, eli yksi potilas voi sisältää useita mittauksia ja mittaus voi kuulua vain yhdelle potilaalle. Samoin *Työntekijä*- ja *Organisaatio*-luokkien välillä on yhdestä-moneen-suhde, eli työntekijä voi kuulua yhteen organisaatioon ja organisaatioon voi kuulua useita työntekijöitä. Lisäksi *Henkilö*- ja *Osoite*-luokkien välissä on monesta-moneen-suhde. Tällöin tietokannan tasolla joudutaan luomaan erillinen välitaulu suhteiden tallentamiseksi. Luokat sisältävät erilaisia attribuutteja, jotka sisältävät eri luokkien kannalta oleellista tietoa. Luokkien instanssit

yksilöidään juoksevalla Id-numerolla, joka toimii myös jokaisen luokan kohdalla pääavaimena.



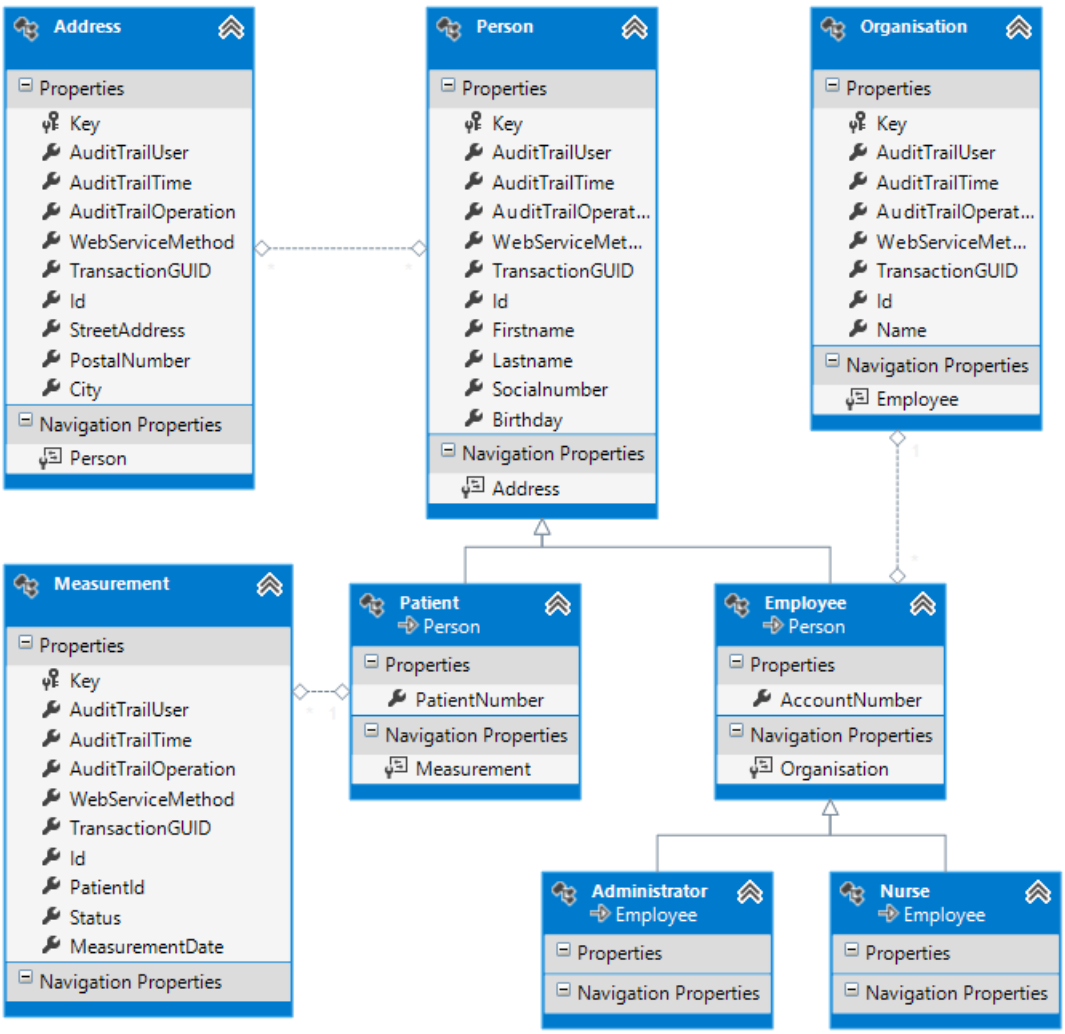

*Kuva 4.1 Työssä käytettävän testisovelluksen tietomalli*

Ensimmäinen vaihtoehto tietomallin toteuttamiseen on luoda Audit Trailille kokonaan erillinen tietokanta, johon luodaan kopiot jokaisesta päätietokannan taulusta. Lisäksi jokaiseen Audit Trail -tietokannan tauluun lisätään taulukossa 4.1 listatut kentät. Pääavaimeksi täytyy asettaa tapahtuman yksilöivä tunniste, koska Audit Trail -kantaan voi tulla useita rivejä yhtä päätietokannan taulun riviä kohti, ja näin ollen alkuperäistä pääavainta ei voida käyttää Audit Trail -taulun pääavaimena. Käyttäjänimi, aikaleima ja tietokantaoperaatio ovat vaadittuja kenttiä. Tietokantaoperaatiota kutsuvan metodin nimi on hyödyllinen, mikäli tietokannan tiedoissa havaitaan virheitä. Jokin taulun kenttä on saattanut jäädä esimerkiksi päivittymättä. Tällöin metodin nimen avulla voidaan jäljittää virheen aiheuttavaa kohtaa ohjelmakoodissa. Saman tietokantatransaktion sisällä saatetaan tehdä useampia tietokantaoperaatioita. Tällöin kaikki yhden transaktion sisällä muuttuneet tietokannan rivit pystytään etsimään transaktiotunnisteen avulla.

*Taulukko 4.1 Audit Trail tietokantakentät*

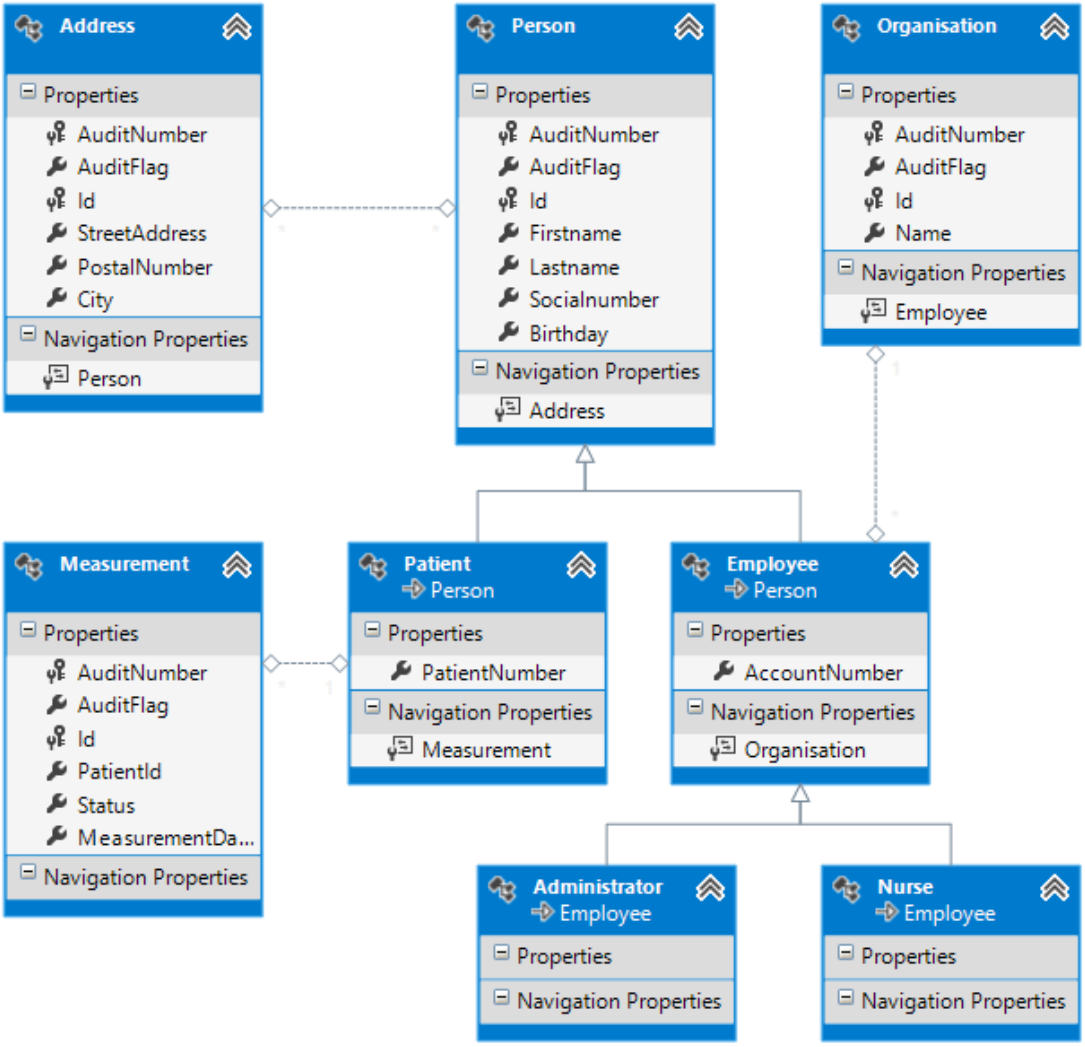
| Kentän nimi | Selitys |
|---|---|
| Key | Audit Trail taulun keinotekoinen pääavain |
| AuditTrailUser | Tietokantaoperaation tehneen käyttäjän nimi |
| AuditTrailTime | Tietokantaan kohdistuneen operaation aikaleima |
| AuditTrailOperation | Tietokantaoperaation tyyppi (INSERT/UPDATE/DELETE) |
| WebServiceMethod | Tietokantaoperaatiota kutsuvan palvelinpään metodin nimi |
| TransactionGUID | Tietokantatransaktion yksilöivä tunniste |

Kuvassa 4.2 esitellään Audit Trail tietomallin toteutus kopiotauluilla. Tietomallista on poistettu entiteettien väliset assosiaatiot, koska tässä diplomityössä Entity Frameworkilla toteutettu Audit Trail -toteutus ei osaa käsitellä assosiaatioita oikein. Lisäksi kaikki taulujen kentät pääavainta lukuun ottamatta on asetettu arvoon nullable, eli niiden arvo voidaan jättää asettamatta. Näin meneteltiin, jotta kaikkien Audit Trail -tietojen tallentaminen ei epäonnistuisi yhden puuttuvan tietokentän takia.




*Kuva 4.2 Audit Trail tietomallin toteutus kopiotauluilla*

Toisessa tietokantamallin toteutusvaihtoehdossa jäljitettävyys toteutetaan samaan tietokantaan ja samoihin tauluihin ajantasaisen datan kanssa. Kuvassa 4.3 on esitetty esimerkkitoteutus. Tässä ratkaisuvaihtoehdossa jokaiseen tauluun lisätään sarakkeet versionumerolle (*`AuditNumber`*) ja jäljitettävyyslipulle (*`AuditFlag`*). Aina kun johonkin riviin tehdään muutoksia, rivistä tehdään uusi kopio, jonka versionumeroa kasvatetaan. Vanhan rivin *`AuditFlag`*-kentän arvoksi asetetaan nolla ja uusimmalle riville arvoksi tulee yksi. Näin taulusta löydetään aina nopeasti uusin versio tietystä tietokannan rivistä.




*Kuva 4.3 Audit Trail tietomallin toteutus samoihin tauluihin varsinaisen datan kanssa*

## 4.2 Toteutettavuus

Sovelluskomponentin toteutettavuuteen vaikuttaa moni asia. Tässä työssä keskitytään tarkistelemaan tarvittavan ohjelmakoodin määrää eri toteutusvaihtoehdoissa ja arvioidaan toteutuksen vaikeusastetta. Toteutukseen kuluvaa aikaa ei suoraan pystytä arvioimaan, mutta koodin määrä ja toteutuksen vaikeusaste antavat jonkinlaisen suuntaa-antavan arvion toteutustyön kestosta.

Toteutuksen vaikeusastetta ei pystytä arvioimaan täysin objektiivisesti, sillä vaikeusasteeseen vaikuttavat ohjelmoijan aiempi kokemus, tiedot ja taidot. Näin ollen vaikeusasteen arviointi perustuu subjektiiviseen käsitykseen toteutettavuuden vaikeudesta. Käytettyjen tekniikoiden aiemman tuntemisen vaikutusta pyritään arvioinnissa välttämään ja lähestytään toteutustapoja sellaisen henkilön näkökulmasta, jolla ei ole kummastakaan toteutustavasta aiempaa kokemusta.

Ohjelmakoodissa käytettävä abstraktiotaso vaikuttaa paljolti toteutuksen vaatimaan koodin määrään. Toteutuksessa käytetyt kirjastot ja toteutustekniikat taas vaikuttavat abstraktiotasoon. Käytettävät kirjastot ja sovelluskehykset vähentävät huomattavasti tarvittavan koodin määrää, koska ne sisältävät jo osan vaadittavan toiminnallisuuden toteutuksesta. Kirjastojen ja sovelluskehysten dokumentaatio vaihtelee erinomaisen ja kokonaan puuttuvan välillä. Hyvin dokumentoitu komponenttikirjasto helpottaa toteutettavuutta huomattavasti. Huono tai kokonaan puuttuva dokumentaatio ja epäselvä komponenttikirjaston rajapinta voi tehdä toteutuksesta niin haastavaa, että on helpompaa hylätä kirjasto kokonaan ja tehdä toteutus alusta asti itse.

Ohjelmointikieli ja kielen syntaksi vaikuttavat myös koodin määrään. Koodin määrä saman toiminnallisuuden toteuttamiseen vaihtelee eri ohjelmointikielien välillä. Korkeamman tason ohjelmointikielet sisältävät valmiita kirjastoja ja tarjoavat näin ollen sovelluskehittäjälle korkeamman abstraktiotason ohjelmointirajapinnan, jolloin kehittäjän ei tarvitse itse toteuttaa kaikkein yksinkertaisimpia perusasioita alusta asti.

## 4.3 Suorituskyky

Suorituskyky on sitä tärkeämpi, mitä suurempi tietokanta on kyseessä, mitä enemmän kantaan on talletettu dataa ja mitä enemmän sillä on käyttäjiä. Suorituskykyyn vaikuttaa muun muassa laitteisto, jossa tietokantaa sijaitsee, käytettävä tietokannan hallintajärjestelmä ja mahdolliset tietokannan ja sovelluksen välissä toimivat kirjastot ja sovelluskehykset, kuten olio-relaatio-muuntimet.

Audit Trail -dataa tallennettaessa suorituskykyä on tärkeää tarkastella, sillä Audit Trail -tietokantaan lisätään aina uusia rivejä, kun päätietokannan johonkin tauluun lisätään, poistetaan tai päivitetään rivejä. Audit Trail kannan sisältämä tietomäärä saattaa kasvaa näin ollen hyvinkin nopeasti. Tietokannan on siis tuettava suuriakin datamääriä. Mikäli

tietokantaa käyttävällä sovelluksella on useita rinnakkaisia käyttäjiä, kohdistuu tietokantaan helposti monia samanaikaisia päivitysoperaatioita. Tietokantatransaktioiden on oltava riittävän nopeita, jotta sovelluksessa ei esiintyisi datan tallentamisen odottelusta johtuvaa hitautta.

Tietokannan suorituskyky voidaan jakaa kahteen osaan: tietokantaan tehtäviin kirjoitusoperaatioihin ja datan lukemiseen tietokannasta. Nämä kaksi toimintoa ovat tehokkuuden kannalta toisensa poissulkevia. Datan lukemista voidaan nopeuttaa luomalla tietokantaan indeksejä. Indeksi voidaan luoda yhdestä tai useammasta tietokantataulun sarakkeesta. Indeksien avulla hakuja pystytään tekemään nopeasti ilman että joka hakukerralla pitäisi käydä tietokantataulun kaikki rivit järjestyksessä läpi.

Tässä työssä suorituskykyä vertaillaan tekemällä suuri määrä tietokantatransaktioita peräkkäin ja mittaamalla komennon antamisen ja transaktion valmistumisen välistä aikaa.

## 4.4 Ylläpidettävyys

Ylläpidettävyys on yksi ohjelmistojen ylläpitoon liittyvistä laatutekijöistä. Ylläpidettävyydellä tarkoitetaan sitä, miten helppoa valmiiseen sovellukseen on tehdä muutoksia. Myös moni muu laatutekijä vaikuttaa sovelluksen ylläpidettävyyteen. Esimerkiksi mitä virheettömämpi, luotettavampi ja käytettävämpi sovellus on, sitä helpommin se on ylläpidettävissä. Ohjelman tehokkuus vastakohtaisesti usein vähentää ohjelman ylläpidettävyyttä.

Toisaalta ylläpidettävyydestä voi olla hyötyä jo sovelluksen kehitysvaiheessa etenkin suurikokoisten ohjelmistoprojektien kohdalla, mikäli muuttuneiden vaatimusten takia joudutaan tekemään muutoksia ohjelman rakenteeseen. Kesken kehitysvaiheen ilmenevien muutostarpeiden määrä riippuu projektissa käytetystä vaihejakomallista. Perinteisessä vesiputousmallissa vaatimusten määrittely ja suunnittelu tehdään huolellisesti ennen toteutusta ja toteutusvaiheessa sovelluksesta tuotetaan vain yksi lopullinen versio, joka sisältää kaikki halutut ominaisuudet ja toiminnallisuuden. Iteratiivisessa prosessissa sovelluksesta toteutetaan kehitysvaiheessa useita versioita. Mikäli kehitysvaiheessa huomataan muutostarpeita, voidaan ne toteuttaa seuraavan iteraation versioon. [19]

Ylläpito on ohjelmistotuotannon yksi vaihe, johon ylläpidettävyys olennaisesti liittyy. Ylläpitovaihe alkaa ohjelmiston käyttöönotosta ja päättyy, kun ohjelmisto poistetaan käytöstä. Vanhojen ohjelmistojen ylläpitoon on arvioitu kuluvan aikaa jopa 70 % ohjelmistoyritysten työajasta ja vain noin 30 % uusien ohjelmien kirjoittamiseen. Ylläpito ja ylläpidettävyys ovat tärkeässä asemassa tulevaisuudessakin, sillä vanhojen ohjelmien ja sovelluskomponenttien ikä ja määrä kasvavat jatkuvasti. [19]

Sovelluksiin voi kohdistua monenlaisia erilaisia muutostarpeita ja näiden myötä myös ohjelmien ylläpito voidaan jakaa seuraavan laisiin kohtiin [19]:

- **Korjaava ylläpito**. Käyttäjien päivittäisessä käytössä löytämien virheiden korjaaminen.
- **Mukauttava ylläpito**. Tarvitaan, kun ohjelmien täytyy mukautua uusiin laitteisiin, käyttöjärjestelmiin ja muihin uusiin ympäristöihin.
- **Täydellistävä ylläpito**. Parannellaan vanhoja ominaisuuksia tai toteutetaan uusia ominaisuuksia, joita ei osattu vielä ajatella sovelluksen määrittelyvaiheessa.
- **Ehkäisevä ylläpito**. Varaudutaan tulevaisuuteen muuttamalla ja parantamalla ohjelman rakenteita, jolloin tulevat ylläpito- ja muutostoimenpiteet helpottuvat.

Tässä työssä ei rajoituta pelkästään ylläpitovaiheessa tehtäviin muutoksiin. Huomioon otetaan myös toteutusvaiheessa ilmenevät muutostarpeet ja niihin reagoiminen.

## 4.5 Uudelleenkäytettävyys

Uudelleenkäytettävyys on tärkeä arviointikriteeri tässä diplomityössä käsiteltävälle Audit Trail -sovelluskomponentille. Kehitetyistä ohjelmistoista jopa 60-80% on väitetty olevan jo aiemmin toteutettua [20]. Työn tuottavuus kärsii, mikäli näin suuri osa työstä tehdään jokaisessa ohjelmistoprojektissa uudelleen. Uudelleenkäytettävyydellä saavutettaisiin teoriassa siis parempi tuottavuus ja säästöjä ajankäytössä ja näin ollen kustannuksissa. Uudelleenkäytettävyyttä voidaan soveltaa ohjelmistoprojektin eri vaihetuotteisiin. Esimerkiksi vanhan projektin määrittelydokumenttia voidaan käyttää pohjana uudessa projektissa. Yleisimmin uudelleenkäytettävyydellä kuitenkin tarkoitetaan juuri sovelluskomponenttien ja ohjelmakoodin uudelleenkäyttöä. Myös tässä työssä rajoitutaan sovelluskomponenttien yleiskäyttöisyyteen.

Sovelluskomponentit voidaan jaotella kolmeen eri luokkaan: yleiskäyttöisiin komponentteihin, sovellusaluekohtaisiin komponentteihin ja sovelluskohtaisiin komponentteihin. Yleiskäyttöisiä komponentteja ovat esimerkiksi matematiikkakirjastot, käyttöliittymäkirjastot ja tietorakennekirjastot. Sovellusaluekohtaiset komponentit on suunniteltu jollekin tietylle sovellusalueelle, esimerkiksi televerkkojen suunnitteluun. Sovelluskohtaiset komponentit liittyvät johonkin tuotteeseen tai tuoteperheeseen. Sovelluskohtainen komponentti voi olla vaikkapa käyttöliittymäkirjasto, joka huolehtii sovelluksen eri näkymien keskinäisestä yhtenäisyydestä. [20]

Uudelleenkäytön eduiksi lasketaan tuottavuuden ja laadun paraneminen. Tuottavuus kasvaa, koska koko ohjelmaa ei tarvitse toteuttaa joka kerta alusta alkaen uudelleen. Laajasti käytetyt ja hyväksi todetut komponentit ovat usein laadultaan virheettömiä tai mahdolliset ongelmat ovat ainakin laajalti tiedossa. Näin ollen aikaa säästyy virheiden korjaamiselta ja uudelleenkäyttö johtaa ohjelman laadun paranemiseen.

Uudelleenkäytöllä on myös ongelmakohtansa. Laadukkaiden komponenttikirjastojen luominen ja ylläpitäminen vaatii käytännössä suuren työmäärän ja osaamista. Kirjastojen etsimiseen joutuu usein näkemään vaivaa, dokumentoinnit ovat monesti puutteellisia eikä muiden tekemiin komponentteihin välttämättä luoteta. Uuden komponentin luominen saatetaan kokea helpommaksi kuin vanhaan komponenttiin perehtyminen.

Jotta sovelluskomponentti olisi uudelleenkäytettävä, se pitää räätälöidä jotenkin eri käyttökohteisiin sopivaksi. Erilaisia räätälöintitapoja on monia. Yksinkertaisin tapa on muokata komponentti käsin kuhunkin käyttötarkoitukseen sopivaksi. Tämä vaatii kuitenkin aina lisätyötä ja heikentää näin ollen tuottavuutta. Komponentti voidaan myös parametroida sopivalla tavalla. Esimerkkinä parametroinnista on C++-kielen template-rakenne. Myös C#-kieli sisältää mekanismeja geneerisyyden toteuttamiseen. Oliokielien periytymismekanismin hyödyntäminen on yksi varteenotettavimpia vaihtoehtoja komponentin muokkaamiseen. Periytymisessä lapsiluokka perii isäluokkansa attribuutit ja metodit. Lapsiluokassa voidaan määritellä kantaluokalta periytyneen metodin toiminta kokonaan uudelleen. Lisäksi lapsiluokassa voidaan määritellä kokonaan uusia attribuutteja ja metodeja. [20]

Tässä työssä arvioidaan kahden erilaisen Audit Trail -mekanismin toteutustavan uudelleenkäytettävyyttä siitä näkökulmasta, miten helppoa kumpikin toteutustapa on ottaa käyttöön kokonaan uudessa projektissa. Arvioinnissa keskitytään siihen, mikä osuus toteutuksesta on uudelleenkäytettävää, mitä joudutaan muuttamaan ja paljonko uuden projektin kohdalla joudutaan kirjoittamaan kokonaan uutta koodia.

# 5. TOTEUTUS ASIAKASJÄRJESTELMÄSSÄ

Audit Trail -tietojen tallennusmekanismi toteutettiin PET ERP -toiminnanohjausjärjestelmän yhteydessä. Toteutustapaa valittaessa tarkasteltiin aluksi muissa projekteissa käytettyjä toteutustapoja ja pohdittiin niiden käyttökelpoisuutta. Yksi näistä toteutustavoista on tietokantatason toteutus käyttäen tietokannan herättimiä. Kokonaan uudeksi lähestymistavaksi keksittiin tietojen tallentamismekanismin toteuttaminen korkeammalla tasolla käyttäen apuna Entity Frameworkia. Tässä luvussa kerrotaan tarkemmin kummankin tallennusmekanismin toteutuksesta. Kohdassa 5.1 kuvataan Entity Frameworkia hyödyntävän toteutuksen toimintaperiaate ja yksityiskohdat ja kohdassa 5.2 käydään läpi vastaavasti tietokantatason triggereitä käyttävän toteutuksen taustat.

## 5.1 Entity Framework -toteutus

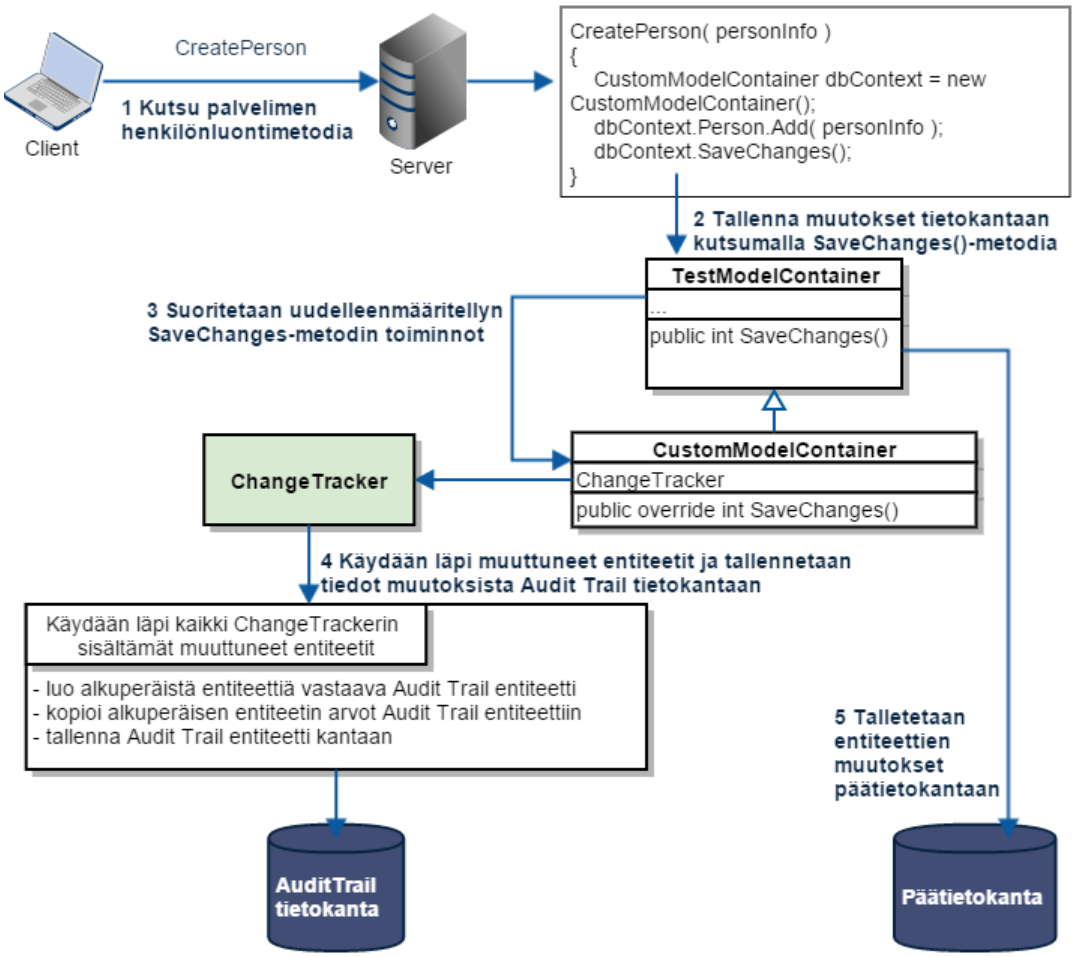
Audit Trailin ensimmäinen vaihtoehto toteutettiin käyttäen C#-ohjelmointikieltä ja Entity Framework -sovelluskehystä. Tietokannan skeema suunniteltiin Model Designer -työkalulla, joka osaa generoida mallista suoraan tietokannan luontiin käytetyt SQL-lauseet. Päätietokanta toteutettiin samoilla tekniikoilla. Päätietokantaa toteuttaessa Entity Framework luo *`DbContextista`* periytyvän *`ModelContainer`*-luokan, joka sisältää tietokannan tauluja vastaavat luokat. *`DbContext`* sisältää monia hyödyllisiä metodeja ja ominaisuuksia, joista tämän toteutuksen kannalta tärkeimmät ovat *`SaveChanges`*-metodi ja tietomallin tilaa ylläpitävä *`DbChangeTracker`*.

Kutsuttaessa kontekstin *`SaveChanges`*-metodia konteksti tallentaa kaikkiin sen seurannassa oleviin entiteetteihin kohdistuneet muutokset tietokantaan. *`SaveChanges`*-metodi voidaan ylimääritellä (override), jolloin tallennuksessa tehtävät operaatiot pystytään määrittelemään itse uudelleen. Tätä käytettiin hyödyksi Audit Trailin toteutuksen perustana. Päätietokannan *`ModelContainer`*-luokasta periytettiin uusi *`CustomModelContainer`*-luokka, joka määrittelee *`SaveChanges`*-metodin uudelleen. Sovelluksen tietokantakontekstina käytetään kyseistä *`CustomModelContaineria`*. Uudelleenmääritelty *`SaveChanges`*-metodi sisältää varsinaisen Audit Trailin toteutuslogiikan.

Kuvassa 5.1 esitetään Audit Trailin Entity Framework -toteutuksen päävaiheet. Kuvan taustalla on asiakas-palvelin-sovellus. Asiakassovelluksessa luodaan uusi henkilö ja kutsutaan palvelimen metodia, joka tallentaa henkilön tiedot. Palvelimella luodaan tietokantakonteksti, lisätään uusi henkilö kontekstiin ja aloitetaan muutoksien tallennus kutsumalla kontekstin *`SaveChanges`*-metodia. Aluksi suoritetaan ylimääritellyn *`Sa-`*

*veChanges*-metodin logiikka, joka huolehtii Audit Trail -tietojen tallentamisen. Tämän jälkeen suoritetaan alkuperäisen *SaveChanges*-metodin logiikka, joka tallentaa muuttuneet tiedot päätietokantaan.

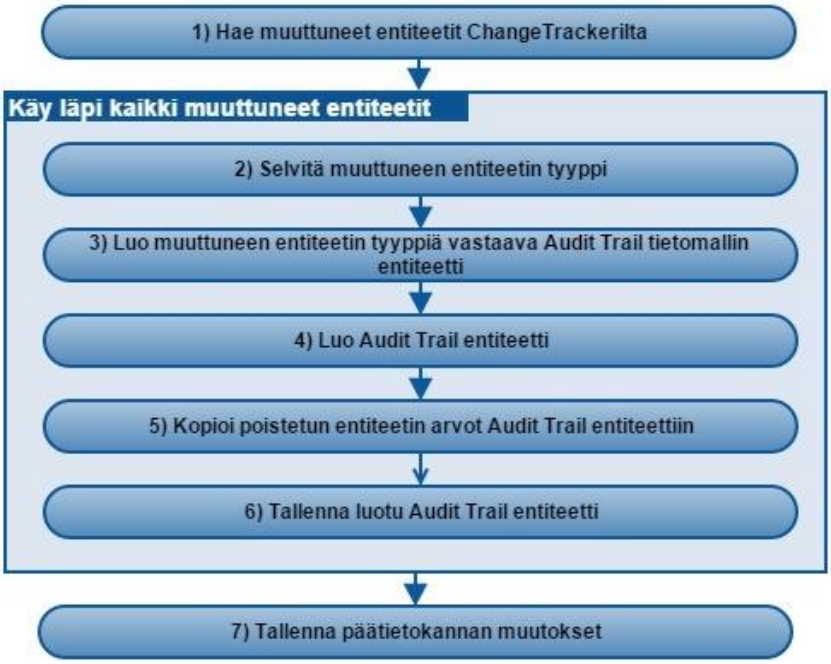
Toteutuksen esiehtovaatimuksena on, että Audit Trail -tietokanta sisältää täsmälleen samannimiset taulut kuin päätietokantakin. Audit Trail -kannan tauluissa on oltava kaikki ne päätietokannan taulujen kentät, jotka halutaan mukaan jäljityksen seurantaan ja kenttien on oltava samannimisiä kuin päätietokannassa. Lisäksi Audit Trail -kannan taulujen väliset suhteet on poistettu, koska tässä työssä toteutettu Entity Frameworkilla toteutettu mekanismi ei osaa käsitellä suhteita oikein.




*Kuva 5.1 Audit Trailin toteutusperiaate Entity Frameworkia hyödyntäen*

Toteutuslogiikka jakautuu muutamaan pääkohtaan, jotka on listattu kuvassa 5.2. Vaiheessa 1 *ChangeTrackerilta* haetaan kaikki kontekstin seurannassa olevat muuttuneet entiteetit, joiden tila on lisätty, poistettu tai muokattu. Tämän jälkeen kaikki muuttuneet entiteetit käydään läpi silmukkarakenteessa. Vaiheessa 2 selvitetään muuttuneen entiteetin tyyppi käyttäen C#:n *Object.GetType*-metodia. Vaiheessa 3 muuttuneen entiteetin tyypistä selvitetään käännösaikainen nimi (assembly-qualified name). Käännösaikainen nimi koostuu tyyppinimestä ja tyypin nimiavaruudesta. Esimerkiksi esi-

merkkitietomallin *Nurse* entiteetin käännösaikaisesi hyväksytty nimi on muotoa "*TestDataModel.Nurse, TestDataModel, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null*". Käännösaikainen nimi vaihdetaan osoittamaan Audit Trail -tietomallin vastaavaan entiteettiin korvaamalla nimiavaruus *TestDataModel* nimiavaruudella *AuditTrailDataModel*. Käännösaikaisesti hyväksytyn nimen perusteella voidaan ladata sitä vastaava tyyppi. Ominaisuutta hyödynnetään lataamalla Audit Trail -entiteettiä vastaavasta nimestä entiteetin tyyppi.




*Kuva 5.2 Audit Trailin korkean tason toimintaperiaate*

Seuraavaksi vaiheessa 4 luodaan dynaamisesti varsinainen Audit Trail -entiteetti, johon alkuperäisen muuttuneen entiteetin arvot kopioidaan. Dynaamiseen luomiseen käytetään *System.Activator*-luokan *CreateInstance*-metodia. Metodi ottaa parametrina luotavan olion tyypin, joka on tässä tapauksessa edellisessä vaiheessa selvitetty Audit Trail -entiteetin tyyppi.

Vaiheessa 5 kopioidaan muuttuneen entiteetin arvot Audit Trail -entiteettiin. *ChangeTracker* sisältää muuttuneesta entiteetistä alkuperäiset arvot ja nykyiset arvot. *ChangeTracker* ei sisällä muuttuneelle entiteetille alkuperäisiä arvoja, jos muuttuneeseen entiteettiin kohdistunut operaatio on lisäys. Vastaavasti jos entiteettiin kohdistunut operaatio on poisto, ei *ChangeTracker* sisällä kyseiselle entiteetille alkuperäisiä arvoja. Tästä johtuen poistettujen entiteettien kohdalla Audit Trailin arvoiksi kopioidaan muuttuneen entiteetin alkuperäiset arvot. Lisätyille ja päivitetyille entiteeteille taas käytetään entiteetin nykyisiä arvoja.

Arvojen kopiointiin käytetään entiteettien tyyppitietoja ja C#:n reflektiota. Aluksi selvitetään muuttuneen entiteetin ja tallennettavan Audit Trail -entiteetin tyypit *GetType*-metodilla. Tämän jälkeen muuttuneen entiteetin ominaisuudet saadaan haettua kutsu-

malla entiteetin tyypin sisältävälle muuttujalle *GetProperties*-metodia. Metodi palauttaa taulukollisen *PropertyInfo*-olioita, jotka ovat osa C#-kielen reflektio-ominaisuuksia. Kaikki ominaisuudet käydään läpi silmukkarakenteessa. Muuttuneen entiteetin ominaisuuden arvo haetaan kutsumalla ominaisuutta vastaavalle *PropertyInfo*-oliolle *GetValue*-metodia. Metodille annetaan parametrina se olio, jonka ominaisuuden arvo halutaan hakea. Tässä tapauksessa parametrina annetaan muuttunutta entiteettiä edustava olio. Seuraavaksi etsitään muuttuneen entiteetin ominaisuutta vastaava Audit Trail -entiteetin ominaisuus. Etsintä tehdään hakemalla Audit Trail -entiteetin kaikki ominaisuudet ja valitsemalla se ominaisuus, jonka nimi vastaa muuttuneen entiteetin käsittelyssä olevan ominaisuuden nimeä. Mikäli Audit Trail -entiteetistä löytyy vastaava ominaisuus, asetetaan ominaisuuden arvoksi muuttuneen entiteetin ominaisuuden arvo.

Muuttuneen entiteetin sisältämien tietojen kopioinnin jälkeen Audit Trail -entiteettiin asetetaan vielä Audit Trailille kohdassa 2.4 asetettujen vaatimusten mukaiset tiedot. Tässä työssä tietoja ovat muutoksen tehneen käyttäjän nimi, aikaleima, tietokantaoperaatio (*INSERT*/*UPDATE*/*DELETE*), tietokantaoperaation aloittaneen metodin nimi ja transaktiotunniste.

Kun alkuperäisen entiteetin arvojen kopiointi Audit Trail -entiteettiin on valmis, tallennetaan Audit Trail -entiteetti vaiheessa 6 Audit Trail tietokantaan. Jotta tallennus voitaisiin tehdä, pitää aluksi hakea Audit Trail -tietokannan joukko, johon uusi entiteetti lisätään. Haku tapahtuu kutsumalla Audit Trail -kontekstin *Set*-metodia, jonka parametriksi annetaan lisättävän entiteetin tyyppi. Tämän jälkeen entiteetti voidaan lisätä normaalisti kutsumalla haetun joukon *Add*-metodia ja antamalla parametrina lisättävä entiteetti.

Kun halutut tiedot on saatu tallennettua Audit Trailiin, täytyy tietokantaoperaatio suorittaa loppuun kutsumalla vielä alkuperäistä *SaveChanges*-metodin toteutusta komennolla *base.SaveChanges()*. Audit Trail -tallennuksen logiikka pitää rakentaa niin, että alkuperäistä *SaveChanges*-metodia kutsutaan aina, vaikka Audit Trailin tallennus epäonnistuisi ja heittäisi poikkeuksen. Tämä vaatii huolellista suunnittelua ja poikkeustilanteisiin varautumista.

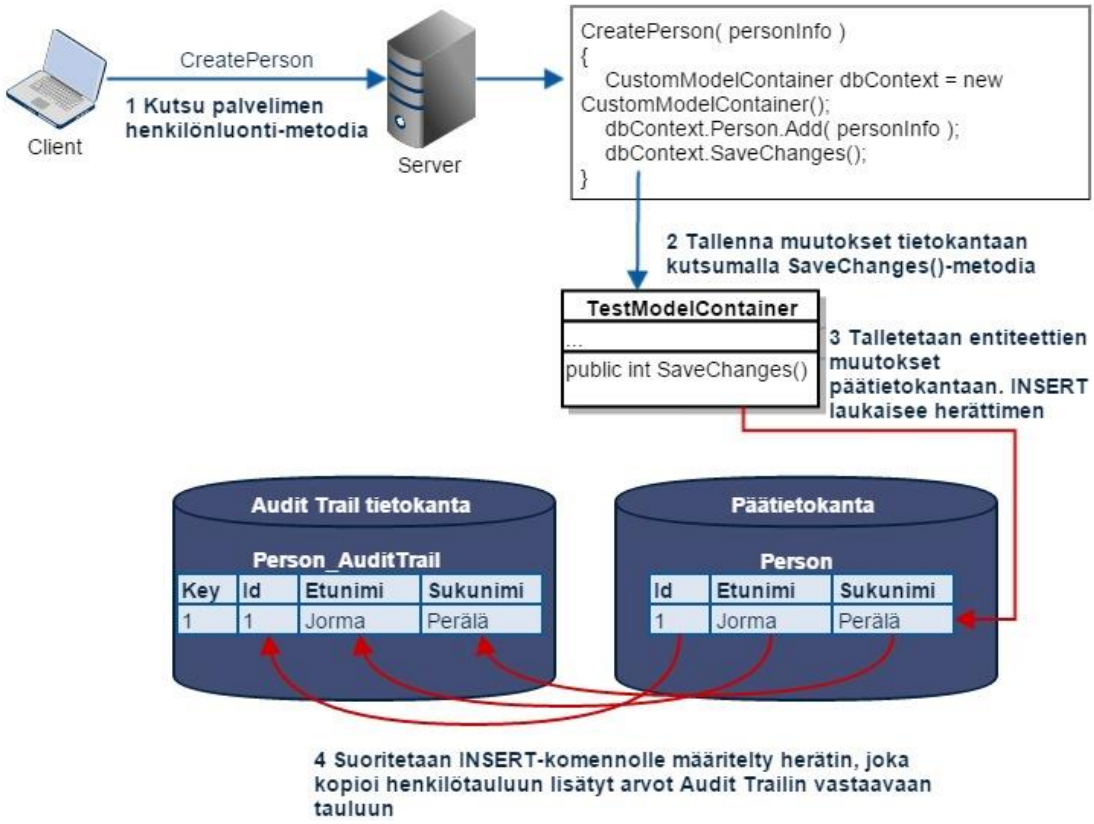
Audit Trailin toteuttaminen sovellustasolla tekee ratkaisusta joustavan. Tietojen tallennuslogiikkaan on mahdollista tehdä monipuolisesti erilaisia käyttäjän haluamia lisäyksiä ja muutoksia. Ongelmana tässä työssä toteutetulla versiolla on Audit Trail tietokannan taulujen välisten assosiaatioiden puuttuminen. Suuren kuormituksen alla sovellustason toteutuksen toiminta saattaa myös hidastua merkittävästi. Tehokkuusongelmien ratkaisemiseksi Audit Trail -mekanismi voidaan siirtää sovellustasolta tietokannan tasolle tietokannan herättimien avulla.

## 5.2 Tietokantaherättimillä toteutus

Toisessa Audit Trail -mekanismin toteutustavassa on käytetty Microsoft SQL Serverin herättimiä. Toteutus sijaitsee puhtaasti tietokannan hallintajärjestelmän tasolla. Päätie-

tokanta ja Audit Trail -tietokanta suunniteltiin ja toteutettiin samalla tavalla kuin kohdassa 5.1 käyttäen Entity Frameworkia ja Model Designeria. Audit Trail -kanta olisi voitu toteuttaa myös manuaalisesti kirjoittamalla taulujen luontilauseet alusta alkaen itse. Entity Frameworkia haluttiin kuitenkin käyttää, jotta tietojen hakeminen olisi yhtä helppoa kuin päätietokannasta.

Kuvassa 5.3 on esitetty herättimillä toteutetun Audit Trailin korkean tason toimintaperiaate. Vaiheessa 1 asiakassovelluksessa luodaan uusi henkilö ja kutsutaan vastaavaa palvelimen metodia henkilön luomiseksi. Palvelimella luodaan vaiheessa 2 päätietokannan konteksti, lisätään henkilö kontekstiin ja tallennetaan muutokset tietokantaan kutsumalla kontekstin `SaveChanges`-metodia. Vaiheessa 3 Entity Framework luo tarvittavat tietokantalausekkeet entiteetin lisäämiseksi ja aloittaa tietokantatransaktion. Päätietokannan `Person`-tauluun kohdistuva lisäys laukaisee tauluun `INSERT`-operaatiolle määritellyn herättimen. Herätin luo Audit Trail tietokannan `Person`-tauluun uuden rivin ja kopioi sinne päätietokannan tauluun lisätyt tiedot.




*Kuva 5.3 Audit Trailin toteutusperiaate triggereitä käyttäen*

Herättimet toteutettiin Microsoft SQL Server Management Studiolla. Herättimet tallennettiin päätietokannan tauluihin, jolloin ne aktivoituvat lisätessä, päivittäessä ja poistaessa tietoja päätietokannasta. Jokaiselle päätietokannan taululle luotiin oma herättimen-

sä komennoille *INSERT*, *UPDATE* ja *DELETE*. Herättimien määrä kasvaa nopeasti tietokannan taulujen määrän mukaan.

Kuvassa 5.4 on esitetty päätietokannan *Person*-tauluun määritelty *INSERT*-herätin. Päätietokannan nimi on *TestDataModelTriggers*. Audit Trail -tietokannan nimi on vastaavasti *AuditTrailDataModelTriggers*. Herättimen luontilause alkaa riviltä 9. Herättimen nimeksi annetaan *tr_Person_Insert*. Rivillä 10 määritellään tietokantataulu, johon herätin lisätään. Rivillä 11 määritellään herättimen tyyppi, joka on tässä tapauksessa *AFTER*. Herätin asetetaan aktivoitumaan *INSERT*-operaation yhteydessä. Päivityksen yhteydessä aktivoituva herätin aloitetaan komennolla *FOR UPDATE* ja poiston yhteydessä aktivoituva herätin komennolla *FOR DELETE*. Riveillä 13-19 määritellään tietokanta, taulu ja ne taulun kentät, joihin tietoja halutaan tallentaa. Tässä tapauksessa valitaan Audit Trail -tietokannan *Person*-taulu. Tauluun halutaan tallentaa kaikki alkuperäisen *Person*-taulun tiedot (*Id*, *Firstname*, *Lastname*, *Socialnumber* ja *Birthday*). Lisäksi talletetaan Audit Trail operaatio (*INSERT*/*UPDATE*/*DELETE*) ja muutoksen aikaleima. Rivillä 20 valitaan päätaulusta kopioitavat lähdetiedot. Listan lopussa määritellään lisäksi tietokantaoperaation nimi ja haetaan aikaleima *GETDATE*-metodilla. Lopuksi rivillä 21 valitaan virtuaalitaulu *inserted*, josta päätauluun talletetut tietomuutokset käydään kopioimassa.

```
1  USE [TestDataModelTriggers]
2  GO
3
4  SET ANSI_NULLS ON
5  GO
6  SET QUOTED_IDENTIFIER ON
7  GO
8
9  CREATE TRIGGER [dbo].[tr_Person_Insert]
10 ON [dbo].[Person]
11 FOR INSERT
12   AS
13   INSERT INTO [AuditTrailDataModelTriggers].[dbo].[Person](Id,
14                                                            Firstname,
15                                                            Lastname,
16                                                            Socialnumber,
17                                                            Birthday,
18                                                            AuditTrailOperation,
19                                                            AuditTrailTime)
20   SELECT Id,Firstname,Lastname,Socialnumber,Birthday,'INSERT',GETDATE()
21   FROM Inserted;
```

*Kuva 5.4 Person-taulun INSERT-operaatiolle määritelty herätin*

Tietokantatauluun kohdistuvien päivitys- ja poisto-operaatioiden aktivoimat herättimet toteutetaan vastaavalla kaavalla kuin edellä esitetty lisäyksen aktivoima herätin. Päivi-

tyksen sisältämät tiedot haetaan myös *inserted*-virtuaalitaulusta ja poiston tiedot *deleted*-virtuaalitaulusta.

# 6. AUDIT TRAIL TALLENNUSMEKANISMIEN ARVIOINTI JA VERTAILU

Edellä esitellyissä toteutustavoissa on omat vahvuutensa ja heikkoutensa. Tässä luvussa arvioidaan ja vertaillaan kummankin toteutustavan toimivuutta neljästä eri näkökulmasta. Kohdassa 6.1 arvioidaan kummankin mekanismin toteutettavuutta. Kohdassa 6.2 testataan toteutustapojen suorituskykyä suorittamalla suuri määrän tietokantaoperaatioita kummallakin mekanismilla ja mittaamalla suoritusaikoja. Kohdassa 6.3 arvioidaan ja vertaillaan toteutustapojen ylläpidettävyyttä ja kohdassa 6.4 käydään läpi molemmat toteutustavat uudelleenkäytettävyyden näkökulmasta.

## 6.1 Toteutettavuuden arviointi

Toteutettavuutta arvioidaan kohdassa 4.2 esiteltyjen asioiden perusteella. Pääasiallisena arvioinnin kohteena ovat toteutuksen pituus ja vaikeusaste. Lisäksi kummastakin toteutustavasta listataan mahdolliset ongelmat ja kehitystä vaativat kohdat, joihin törmättiin toteutusvaiheessa.

### 6.1.1 Entity Framework -toteutus

Entity Frameworkin avulla toteutetun Audit Trail -mekanismin vaatima koodin määrä on melko pieni. Perustoteutuksen pituus on noin 250 riviä koodia. Koodissa on käytetty paljon .NET-sovelluskehyksen tarjoamaa reflektiota ja dynaamisuutta, mikä lyhentää toteutuksen pituutta huomattavasti. Ilman dynaamisuutta koodissa olisi jouduttu luomaan jokaista entiteettityyppiä varten oma tallennuslogiikka. Toteutuksen pituus olisi tällöin venynyt tuhansiin riveihin ja toteutuksen vaatima aika ollut huomattavasti pidempi. Entity Framework helpottaa kommunikointia tietokannan suuntaan ja nopeuttaa tietokantaoperaatioiden kirjoittamista.

Audit Trailin tallentaminen Entity Framework -mekanismilla vaatii syvällistä perehtymistä Entity Frameworkin toimintaan ja .NET-sovelluskehyksen reflektio- ja tyyppiominaisuuksiin. Kun Entity Frameworkia hyödyntävää Audit Trail -mekanismia alettiin toteuttamaan, ei toteutuksen onnistumisesta ollut täyttä varmuutta. Mekanismin toteuttaminen ei ole täysin suoraviivainen tehtävä. Sen voidaan katsoa olevan tekniikoita aiemmin tuntemattoman kehittäjän kannalta haastava toteuttaa.

Toteutuksen logiikka on monimutkainen ja vaatii tarkkaa dokumentointia, mikäli ulkopuolisen henkilön pitää pystyä jatkokehittämään mekanismia. Monimutkaisuus johtaa helposti myös virhetilanteisiin, joihin ei osata varautua etukäteen.

Entiteettien välisiä assosiaatioita ei otettu mukaan Audit Trailin Entity Framework toteutukseen. Assosiaatiot jätettiin pois, koska tässä työssä toteutettu Audit Trail -tietojen tallennusmekanismi ei osaa käsitellä assosiaatioita oikein.

### 6.1.2 Tietokantaherättimillä toteutus

Tietokannan herättimillä toteutettu Audit Trail -mekanismi vaatii noin 30 riviä koodia yhtä tietokannan taulua kohti. Tässä työssä käytetyn esimerkkitietomallin kohdalla tietokannan tauluja on 9 kappaletta. Näin ollen vaadittujen koodirivien määrä on 270. Vaaditun koodin määrä kuitenkin kasvaa nopeasti sitä mukaa, kun tietokannassa on tauluja. Esimerkiksi sadan tietokantataulun tapauksessa koodia tarvittaisiin jo 3000 riviä. Eri taulujen herättimet ovat peruslogiikaltaan samanlaisia. Näin ollen riittää, kun kirjoitetaan yhden taulun herättimet, kopioidaan herättimet kaikkiin muihin tauluihin ja muokataan jokaisen taulun kohdalla herättimeen oikea taulun nimi ja kopioitavien sarakkeiden nimet.

Herättimien toimintaperiaate ja syntaksi Microsoft SQL Server -tietokannan hallintajärjestelmässä on helppo ymmärtää ja oppia. Tietojen kopioiminen päätietokannan taulusta Audit Trail -tietokannan tauluun on yksinkertainen operaatio. Tekniikkaa ja työkaluja aiemmin tuntematon kehittäjä pystyy toteuttamaan ensimmäiset herättimet nopeasti. Kun yhden tietokantataulun herättimet on toteutettu, on herättimien kirjoittaminen muihin tauluihin lähinnä mekaanista työtä. Samoja herättimiä voidaan käyttää pohjana ja muuttaa niihin vain taulun nimi ja halutut taulun sarakkeet.

Herättimien käytön etuna on sovellustason riippumattomuus. Toteutus toimii hiljaisesti taustalla tietokannan tasolla. Sovelluksen tasolla ei tarvitse tehdä muuta kuin päätietokantaan kohdistuvia tietokantaoperaatioita. Toteutuksen sijainnista tietokannan tasolla on hyötyä erityisesti, jos useampi sovellus käyttää samaa tietokantaa. Tällöin jokaisessa sovelluksessa ei tarvitse erikseen toteuttaa Audit Trailin tallennusmekanismia.

Tietokantaherättimillä toteutetussa tallennusmekanismissa havaittiin myös omat haasteensa. Audit Trail -tietokannan skeema haluttiin suunnitella ja toteuttaa Entity Frameworkia hyödyntäen, jotta sovelluskehystä voitaisiin käyttää tietojen hakemiseen. Tässä työssä käytetään olioiden periytymishierarkian kuvaamiseen tietokannan tasolla alakohdassa 3.3.2 kuvatuista periytymisvaihtoehdoista TPT-tapaa. Periytynyttä oliota kuvaava taulu linkitetään kantaluokan oliota kuvaavaan tauluun vierasavainviittauksella. Herättimet kopioivat yksitellen jokaisen päätietokannan taulun sisällön vastaavaan Audit Trail -tauluun. Periytymishierarkiaan kuuluvien taulujen kohdalla näin ei kuitenkaan voida tehdä. Vierasavainviittausten kohdalla lapsiluokan taulun pääavain on samalla

vierasavain, joka viittaa kantaluokan taulun pääavaimeen. Avain pitäisi hakea kantaluokan taulusta. Hakua ei kuitenkaan onnistuttu tekemään herättimissä. Näin ollen tässä työssä toteutetut herättimet eivät osaa kopioida periytymishierarkioita, vaan ainoastaan hierarkiassa ylimpänä sijaitsevan kantaluokan taulun tiedot.

Entiteettien väliset assosiaatiot jätettiin pois myös herättimillä toteutetussa Audit Trail versiossa. Ongelmaksi muodostui vierasavainviitteiden asettaminen kopioitaessa päätietokannan taulujen tietoja Audit Trail -tauluihin.

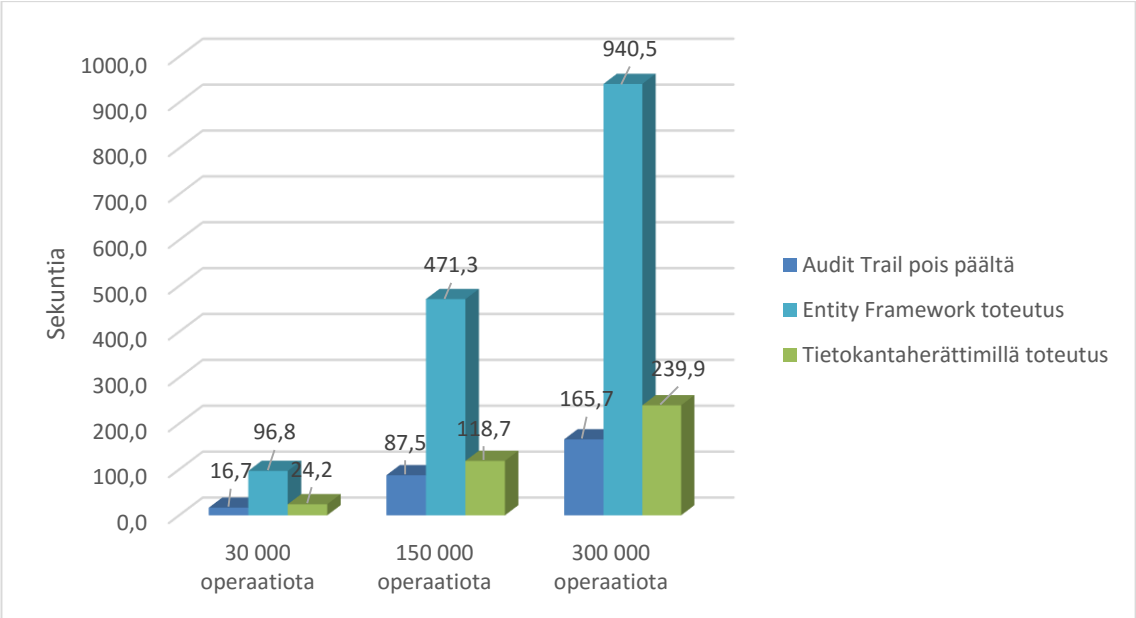
## 6.2 Suorituskyvyn arviointi

Toteutusten suorituskykyä arvioidaan edellä listattujen periaatteiden mukaisesti. Tässä diplomityössä pääpaino on asetettu Audit Trail -tiedon kirjoittamiseen, joten myös suorituskykytesteissä keskitytään mittaamaan tietokantaan kirjoittamisen tehokkuutta. Suorituskykyä testataan tekemällä tietokantaan suuri määrä peräkkäisiä lisäys-, muokkaus- ja poisto-operaatioita käyttäen kumpaakin Audit Trail -mekanismia.

Testi koostuu kolmesta erillisestä operaatiosta, joita suoritetaan silmukassa. Ensimmäisessä operaatiossa tietokantaan lisätään uusi `Person`-entiteetti. Toisessa operaatiossa lisätyn henkilön tietoja muutetaan ja muutokset tallennetaan tietokantaan. Kolmannessa operaatiossa henkilö poistetaan tietokannasta. Silmukkarakennetta suoritettiin eri määriä. Ensimmäisellä kerralla silmukkaa ajettiin 10 000 kertaa, jolloin tietokantaoperaatioita kertyi 30 000. Toisella kerralla silmukkaan asetettiin 50 000 kierrosta, jolloin tietokantaoperaatioita kertyi 150 000. Pisimmässä testissä kierroksia ajettiin 100 000, jolloin tietokantaoperaatioita kertyi 300 000. Testit tehtiin suorittamalla komennot ensin ilman Audit Trailia ja sen jälkeen Audit Trail -tallennus päällä. Testien välissä tietokanta asetettiin alkuperäiseen tilaan tyhjentämällä kaikki taulut.

Kuvassa 6.1 näkyy tehokkuusmittausten tulokset. Mittauksiin kulunut aika on merkitty sekunteina. Mittauksen tulokset on jaettu suoritettujen tietokantaoperaatioiden määrän mukaan kolmeen kategoriaan. Jokaisessa kategoriassa vasemmanpuoleisin pylväs esittää kulunutta aikaa Audit Trailin ollessa poissa päältä, keskimmäinen pylväs kulunutta aikaa Entity Framework Audit Trail -toteutuksen ollessa käytössä ja oikeanpuoleinen pylväs kulunutta aikaa herättimillä toteutetun Audit Trail -toteutuksen ollessa käytössä.

Entity Framework Audit Trail -toteutus osoittautui noin neljä kertaa hitaammaksi verrattuna herättimiä käyttävään toteutukseen. Etenkin suuria datamääriä tietokantaan kirjoittavan sovelluksen kohdalla tämä suorituskykyero on jo varsin merkittävä ja se tulisi ottaa huomioon valittaessa sopivaa ratkaisua.




*Kuva 6.1 Audit Trail tallennusmekanismien suorituskykytestit*

## 6.3 Ylläpidettävyyden arviointi

Ylläpidettävyyttä arvioidaan edellä esiteltyjen kohtien näkökulmasta. Ylläpitotoimia vaativia muutoksia ilmenee yleensä projektin ylläpitovaiheessa. Muutoksia joudutaan välillä tekemään myös toteutusvaiheessa etenkin ketterässä sovelluskehitysprosessissa. Ylläpidettävyyden osalta arvioidaan miten helppoa Audit Trail -mekanismiin on tehdä muutoksia toteutusvaiheessa ja ylläpitovaiheessa.

### 6.3.1 Entity Framework -toteutus

Kun päätietokannan tietomalliin kohdistuu muutoksia, täytyy muutokset tehdä myös Audit Trail -tietomalliin. Entity Frameworkilla toteutettu Audit Trail -tallennusmekanismi toimii dynaamisesti. Se osaa käsitellä kaikki tietokannan taulut, jotka löytyvät päätietokannasta ja Audit Trail -tietokannasta. Näin ollen riittää, kun halutut muutokset tehdään molempiin tietomalleihin.

Tallennusmekanismi toimii myös silloin, jos muutos on tehty vain päätietokantaan. Jos esimerkiksi päätietokannassa olevaan tauluun lisätään kenttä, mutta ei tehdä samaa muutosta Audit Trail -tietomalliin, osaa tallennusmekanismi jättää kyseisen tiedon väliin. Tällöin Audit Trail toimii muuten normaalisti, mutta uuden kentän sisältämä tieto ei tallennu Audit Trailiin.

Päätietokantaan tehdyt muutokset pitää päivittää käsin Audit Trail -tietomalliin. Päivittäminen tapahtuu yksinkertaisesti muokkaamalla entiteettimallia Model Designer -työkalulla. Audit Trail -tietomalli unohdetaan helposti päivittää, mikä muodostuu ongelmaksi etenkin usean sovelluskehittäjän projektissa.

Mikäli Audit Trailiin halutaan lisätä uusia Audit Trail -tietoja, pitää muutokset tehdä kahteen paikkaan. Ensin Audit Trail -tietomallin kaikkiin entiteetteihin täytyy lisätä halutut uudet kentät. Toiseksi lisättyjen kenttien täydentäminen täytyy toteuttaa Audit Trail -mekanismiin. Toteutus on helppoa tehdä lisäämällä halutut kohdat entiteettien arvot kopioivan metodin loppuun.

Kun tietomalliin tehdään muutoksia, täytyy muutokset päivittää myös tietokantaan. Päivittämiseen on PET ERP -projektissa käytetty kahta erilaista tapaa. Ensimmäinen vaihtoehto on pyyhkiä koko vanha tietokanta ja alustaa se uuden tietomallin mukaiseksi. Toinen vaihtoehto on kirjoittaa päivityksien tekemiseen migraatioskriptit. Migraatioskriptit päivittävät tietokannan skeeman ilman, että koko tietokanta joudutaan tyhjentämään. Ensimmäistä vaihtoehtoa voidaan käyttää sovelluksen kehitysvaiheessa, mikäli tietokanta sisältää vain poistettavissa olevaa testidataa.

### 6.3.2 Tietokantaherättimillä toteutus

Tietokantaherättimillä toteutetussa Audit Trailissa tietomalli on toteutettu samalla tavalla kuin Entity Framework toteutuksessa. Näin ollen tietomalliin kohdistuviin päivityksiin pätevät samat asiat kuin alakohdassa 6.3.1. Tietomallin muutokset pitää tehdä sekä päätietokantaan että Audit Trail -tietokantaan. Muutokset tietomalliin tehdään Model Designer -työkalulla.

Tietokannan herättimiin pitää tehdä päivityksiä, mikäli tietomalleihin tehdään muutoksia. Päivitykset pitää tehdä herättimeen, joka sijaitsee muutoksen kohteena olevassa päätietokannan taulussa. Muutokset pitää tehdä kaikkiin kolmeen herättimeen (`INSERT`/`UPDATE`/`DELETE`). Herättimiin voidaan tehdä päivityksiä helposti ilman että koko tietokantaa täytyy nollata.

Jos Audit Trailiin halutaan lisätä kokonaan uusia tai poistaa vanhoja Audit Trail -tietokenttiä, pitää muutokset tehdä kaikkiin päätietokannan taulujen herättimiin. Muutosten tekeminen on mekaanista työtä, mutta mitä enemmän tietokannassa on tauluja, sitä enemmän joudutaan tekemään käytännössä turhaa työtä.

Tietomalliin tehtyjen muutosten tallentaminen tietokantaan voidaan tehdä kahdella tavalla, kuten edellä esitettiin. Vaihtoehdot ovat samat, eli tietokannan nollaaminen ja uuden tietokantaskeeman syöttäminen tietokantaan tai migraatioskriptien kirjoittaminen.

## 6.4 Uudelleenkäytettävyyden arviointi

Uudelleenkäytettävyyden kohdalla arvioidaan miten helppoa Audit Trail -mekanismi on ottaa käyttöön täysin uudessa projektissa ja mitä toimenpiteitä se vaatii. Hyvä uudelleenkäytettävyys tuo lisäarvoa Audit Trail -toteutukselle, koska samaa toteutusta voidaan käyttää pienellä vaivalla muissakin projekteissa eikä kaikkea tarvitse toteuttaa alusta alkaen uudelleen.

### 6.4.1 Entity Framework toteutus

Entity Framework Audit Trail -toteutus ei ole riippuvainen tietokannasta, mikä tekee siitä siirrettävämmän ratkaisun. Mikäli tallennusmekanismin toteuttava koodi vain kopioidaan uuteen projektiin, pitää toteutuksesta muuttaa joitakin kohtia.

Aluksi pitää määritellä oikea konteksti, josta `CustomModelContainer` periytyy. Toiseksi toteutukseen pitää vaihtaa käytettyjen tietokantojen nimet, jotta Audit Trail -entiteettien dynaaminen luominen onnistuu. Kolmanneksi arvojen kopioinnin jälkeen suoritettava Audit Trail -kenttien arvojen asettamiseen käytetty logiikka pitää muokata sopiviksi. Esimerkkinä mainittakoon käyttäjänimi. PET ERP:ssä on käytetty Microsoftin Active Directory -tunnistautumista, joten sovellukseen kirjautuneen käyttäjän nimi haetaan sieltä. Mikäli Audit Trailiin halutaan tallentaa joitain muita Audit Trail -tietokenttiä, mitä tämän työn toteutuksessa on listattu, pitää kyseisten tietojen tallentamiseen lisätä omat kohtansa koodiin.

Toteutus sijaitsee sovellustason koodissa, joten tallennusmekanismin toteuttavasta koodista voitaisiin tehdä kokonaan oma sovelluskomponentti. Sovelluskomponenttia voidaan käyttää uudelleen muissa projekteissa, mikäli komponentista tehdään riittävän joustava esimerkiksi parametrisoimalla. Myös periytymistä ja metodien ylikirjoittamista voidaan hyödyntää tehokkaasti.

### 6.4.2 Tietokantaherättimillä toteutus

Herättimillä toteutettu Audit Trail -tallennusmekanismi sijaitsee kokonaan tietokannan tasolla. Toteutus on täysin tietokantaan sidottu, joten sitä ei voida suoraan sellaisenaan käyttää toisessa projektissa.

Uudessa projektissa herättimet täytyy luoda kokonaan uudelleen, mikäli esimerkkinä käytetään tässä diplomityössä käytettyä tapaa. Herättimet pitää räätälöidä uudessa projektissa käytettävän päätietokannan skeeman mukaiseksi. Alkuperäisiä herättimien luontiin käytettyjä skriptejä voidaan käyttää pohjana, mutta suurin osa tietokantakohtaisista kentistä pitää kirjoittaa uudelleen.

Herätintoteutus pystytään muuttamaan uudelleenkäytettävämmäksi kirjoittamalla tietokantaskripti, joka luo automaattisesti halutut herättimet. Tämäkään ei tee toteutuksesta täysin uudelleenkäytettävää, sillä herättimet luova skripti on tietokantariippuvainen. Microsoft SQL:ään luotu skripti ei toimi esimerkiksi MySQL-tietokannassa.

## 6.5 Vertailu ja johtopäätökset

Tehtyjen arviointien perusteella kummastakin ratkaisusta löytyi omat heikkoutensa ja vahvuutensa. Arvioinneissa tehtyjen havaintojen pohjalta voidaan verrata Audit Trail

-toteutustapojen keskinäistä paremmuutta. Taulukossa 6.1 on vertailtu kahta Audit Trail -ratkaisua kohdissa 6.1 - 6.4 läpi käydyistä näkökulmista. Kumpikin ratkaisu on arvioitu eri näkökulmien kannalta arviointiasteikolla: hyvä, normaali, huono.

*Taulukko 6.1 Audit Trail tallennusmekanismien vertailu*

| | Sovellustason toteutus | Tietokantatason toteutus |
|---|---|---|
| Toteutettavuus (koodin määrä) | Hyvä | Huono |
| Toteutettavuus (ymmärrettävyys) | Huono | Hyvä |
| Suorituskyky | Huono | Hyvä |
| Ylläpidettävyys | Hyvä | Normaali |
| Uudelleenkäytettävyys | Hyvä | Normaali |

Entity Framework ratkaisun toteutettavuus on koodin määrän kannalta hyvä. Koodin määrä on dynaamisuuden ja reflektion ansiosta pieni. Ymmärrettävyyden näkökulmasta Entity Framework -toteutus on hankala ja toteutuksen toimivuus joissakin erikoistilanteissa on epävarmaa. Herättimillä toteutetun Audit Trail -mekanismin toteutettavuus on koodin määrän kannalta huono. Herätintoteutus on melko suoraviivainen ja näin ollen helppo ymmärtää.

Suorituskykymittauksissa kävi ilmi herätintoteutuksen ylivoimaisuus. Tietokantaan kohdistuvat kirjoitusoperaatiot olivat herätintoteutuksessa neljä kertaa nopeampia kuin Entity Framework toteutuksella. Herättimillä toteutettu Audit Trail ei lisännyt tietokantatransaktioiden pituutta merkittävästi verrattuna tilanteeseen, jossa Audit Trailia ei käytetty ollenkaan.

Ylläpidettävyyden osalta kummassakin toteutustavassa Audit Trail -tietomallin ylläpitäminen vaatii saman työmäärän. Entity Framework -toteutus osaa hyödyntää suoraan päivitettyä tietomallia. Muutoksia joudutaan tekemään ainoastaan, mikäli halutaan muuttaa Audit Trail -tietokenttien sisältöä tai lisätä uusia Audit Trail -kenttiä. Herättimillä toteutetussa Audit Trailissa taas joudutaan päivittämään aina kaikki ne herättimet, joita koskeviin tauluihin on tehty muutoksia. Entity Framework -toteutuksen ylläpito on näin ollen helpompaa.

Uudelleenkäytettävyyden näkökulmasta Entity Framework -toteutus on selkeästi parempi. Entity Framework on tietokantariippumaton, joten sen avulla toteutettua Audit Trail -tallennusmekanismia on helppo käyttää muidenkin tietokantojen yhteydessä. Herättimillä toteutettu Audit Trail taas on tiukasti sidottu sovelluksen alla toimivaan tietokannan hallintajärjestelmään. Uudelleenkäytettävyyttä voidaan parantaa kirjoittamalla kaikki herättimet luova skripti.

# 7. YHTEENVETO

Tämän työn tutkimusongelmana oli PET ERP -toiminnanohjausjärjestelmälle asetetun tietojen jäljitettävyysvaatimuksen täyttäminen. Jäljitettävyyden toteuttavaa mekanismia kutsutaan Audit Trailiksi. Audit Trailin pitää tallentaa merkintä kaikista tietokannan tietoihin tehdyistä muutoksista. Merkinnöissä täytyy olla mukana vähintään aikaleima, tehty operaatio ja muutoksen tehneen käyttäjän identifioiva tunniste. Lisäksi voidaan tallentaa muuta hyödyllistä tietoa, kuten tietokantaoperaatiota kutsuneen metodin nimi.

Työssä tutkittiin kahden erilaisen tietokantaa käyttävän Audit Trail -tallennusmekanismin toteutusta. Ensimmäinen ratkaisu toteutettiin sovellustasolla hyödyntäen Entity Frameworkia. Audit Trail tietoja varten luotiin oma tietokanta, johon Audit Trail -mekanismi kopioi kaikki tiedot ja tietoihin kohdistuvat muutokset, jotka tallennetaan päätietokantaan. Toinen ratkaisu toimii tietokannan tasolla käyttäen tallennusmekanismina tietokannan herättimiä. Lisättäessä, muutettaessa tai poistaessa tietoja päätietokannan tauluihin tehdään muutoksista kopiomerkintä Audit Trail -tietokantaan.

Entity Framework -toteutus oli lyhyt, mutta sen toteutus oli melko vaikeasti ymmärrettävä ja hankala. Ylläpidettävyydeltään ja uudelleenkäytettävyydeltään ratkaisu oli hyvä, mutta tehokkuus oli huono. Tämä johtui sovellustasolla tehtävistä monimutkaisista reflektiota ja dynaamisuutta hyödyntävistä operaatioista.

Herättimiä käyttävä ratkaisu oli yksinkertainen, mutta vaati paljon koodia. Jokaiselle tietokantataululle piti kirjoittaa omat herättimet. Herättimien kirjoittaminen vaati paljon mekaanista työtä. Uudelleenkäytettävyys ja ylläpidettävyys ovat heikolla tasolla. Tietokantaskriptillä herättimien luonti pystytään automatisoimaan, mikä tekee ratkaisusta helpomman toteuttaa.

Tämän työn vertailun tuloksia voidaan käyttää hyödyksi, mikäli Audit Trail -mekanismi halutaan toteuttaa kokonaan uudessa projektissa. Yhtä täysin oikeaa ja ylivertaista ratkaisua ei voida löytää, koska toteutuksen reunaehdot riippuvat aina projektin vaatimuksista. Vertailun tuloksia voidaan kuitenkin käyttää apuna ratkaisunvalintaprosessissa.

Audit Trail kannattaa tallentaa omiin tietokantatauluihin, jotta päätietokanta ja Audit Trail saadaan erotettua toisistaan. Audit Trail tietokannassa entiteettien periytymissuhteiden kuvaamiseen kannattaa käyttää TPH-tapaa eli kaikkien periytymishierarkian entiteettien tietojen tallentamista yhteen tauluun. Varsinaiseen Audit Trail -tietojen tallentamiseen on suositeltavaa käyttää tietokannan herättimiä. Vaikka herättimet vaativatkin enemmän työtä, ovat ne luotettavampi ja nopeampi tapa tallentaa Audit Trailia.

# LÄHTEET


[1] Davenport, T.H. Putting the Enterprise into the Enterprise System. Harvard Business Review, 1998, 76(3). Sivut.121–131.)

[2] Varsinais-Suomen Sairaanhoitopiiri. PET-keskus [WWW]. [Viitattu 6.8.2015]. Saatavissa: http://www.vsshp.fi/fi/toimipaikat/tyks/osastot-ja-poliklinikat/Sivut/pet-keskus.aspx

[3] Guttman, B. Edward R. An introduction to computer security: the NIST handbook. DIANE Publishing, 1995. pp. 211-221.

[4] Turun PET-keskus. Tietojärjestelmän hankinta PET-keskuksen toiminnanohjaukseen, ERP-vaatimusmäärittely. Ei julkisesti saatavilla.

[5] Verma, K. 2013. Base of a Research: Good Clinical Practice in Clinical Trials. J Clin Trials 3: 128. doi: 10.4172/2167-0870.1000128.

[6] Guideline for Good Clinical Practice E6(R1). [WWW]. [Viitattu 10.11.2015]. Saatavissa: http://www.ich.org/fileadmin/Public_Web_Site/ICH_Products/Guidelines/Efficacy/E6/E6_R1_Guideline.pdf

[7] Sosiaali- ja terveysministeriön asetus potilasasiakirjoista. [WWW]. [Viitattu 23.8.2015]. Saatavissa: http://www.finlex.fi/fi/laki/alkup/2009/20090298

[8] .NET Development. [WWW]. [viitattu 24.4.2015]. Saatavissa: https://msdn.microsoft.com/en-us/library/ff361664(v=vs.110).aspx

[9] C# Programming Guide. [WWW]. [Viitattu 22.5.2015]. Saatavissa: https://msdn.microsoft.com/en-us/library/67ef8sbd(v=vs.110).aspx

[10] Visual Studio 2012. [WWW]. [Viitattu 24.4.2015]. Saatavissa: https://msdn.microsoft.com/en-us/library/dd831853(v=vs.110).aspx

[11] Books Online for SQL Server 2012. [WWW]. [Viitattu 21.8.2015]. Saatavissa: https://technet.microsoft.com/en-us/library/ms130214(v=sql.110).aspx

[12] ADO.NET Entity Framework. [WWW]. [Viitattu 21.8.2015]. Saatavissa: https://msdn.microsoft.com/en-us/library/vstudio/bb399572(v=vs.100).aspx

[13] Fussell, M. L. 1997. Foundations of Object Relational Mapping. White paper, ChiMu Corp.

[14] Mahil, S. 2005. Pro ADO.NET 2.0. Apress, ISBN 1-59059-512-2.

[15] ADO.NET Architecture. [WWW]. [viitattu 21.5.2015]. Saatavissa: https://msdn.microsoft.com/en-us/library/27y4ybxw(v=vs.100).aspx

[16] Lerman, J. 2010. Programming Entity Framework. O'Reilly, ISBN:978-0-596-80726-9.

[17] Adya, A., Blakeley, J. A., Melnik, S., Muralidhar, S. Anatomy of the ADO.NET Entity Framework [WWW]. Proceedings of the ACM SIGMOD International Conference on Management of Data. 2007, pp 877-888.

[18] Entity Framework Tutorial - Inheritance Strategy in Code First. [WWW]. [Viitattu 8.11.2015]. Saatavissa: http://www.entityframeworktutorial.net/code-first/inheritance-strategy-in-code-first.aspx

[19] Harsu, M. 2003. Ohjelmien ylläpito ja uudistaminen. Talentum Media Oy, ISBN 951-762-829-3.

[20] Haikala, I., Märijärvi, J. 2002. Ohjelmistotuotanto. Talentum Media Oy, ISBN 952-14-0486-8.

# LIITE A: ENTITY FRAMEWORK AUDIT TRAIL -MEKANISMIN KOODI

```
001  public class CustomModelContainer : TestModelContainer
002  {
003    #region Private members
004    private static readonly ILog logger = new FileLog();
005    private Guid _transactionGuid;
006
007    /// <summary>
008    /// All possible alternatives of operations that can be done to
       database
009    /// </summary>
010    private enum DatabaseOperations
011    {
012        ADD = EntityState.Added,
013        UPDATE = EntityState.Modified,
014        DELETE = EntityState.Deleted
015    };
016    #endregion
017
018    #region Constructors
019    /// <summary>
020    /// Initializes a new instance of the
       <see cref="CustomModelContainer"/> class.
021    /// </summary>
022    public CustomModelContainer() : base()
023    {
024        // Proxy creation and lazy loading must be disabled to make
           // the entities serialize over WCF.
025        this.Configuration.ProxyCreationEnabled = false;
026        this.Configuration.LazyLoadingEnabled = false;
027        this._transactionGuid = Guid.NewGuid();
028    }
029    #endregion
030
031    #region Public methods
032    public override int SaveChanges()
033    {
034      int baseResult = 0;
035      int auditLogMode;
036      Int32.TryParse(System.Configuration.ConfigurationManager.
                        AppSettings["auditLogging"], out auditLogMode);
037      if (auditLogMode == 1)
038      {
039        DateTime clock = DateTime.Now;
040        // NOTICE!
041        // OriginalValues can't be used on added (they don't
             exist).
042        // CurrentValues can't be used on deleted (they don't
             exist).
043        IEnumerable<DbEntityEntry> dbchanges =
           this.ChangeTracker.Entries();
044
045        List<DbEntityEntry> addedEntities =
           new List<DbEntityEntry>();
046        List<DbEntityEntry> modifiedEntities =
           new List<DbEntityEntry>();
047        List<DbEntityEntry> deletedEntities =
```

```
            new List<DbEntityEntry>();
048         foreach (var entry in this.ChangeTracker.Entries().Where
049         (x => (x.State == EntityState.Added) ||
050               (x.State == EntityState.Deleted) ||
051               (x.State == EntityState.Modified)))
052         {
053           if (entry.State == EntityState.Added)
054           {
055             addedEntities.Add(entry);
056           }
057           else if (entry.State == EntityState.Modified)
058           {
059             modifiedEntities.Add(entry);
060           }
061           else if (entry.State == EntityState.Deleted)
062           {
063             deletedEntities.Add(entry);
064           }
065         }
066
067         try
068         {
069           using (var auditTrailDB = new AuditTrailModelContainer())
070           {
071             foreach (DbEntityEntry deletedEntity in
                           deletedEntities)
072             {
073               Type deletedEntityType =
                        deletedEntity.Entity.GetType();
074               string deletedEntityTypeName =
                  deletedEntityType.AssemblyQualifiedName.ToString().
                  Replace("TestDataModel", "AuditTrailDataModel");
075               Type auditTrailEntityType =
                  Type.GetType(deletedEntityTypeName);
076               object auditTrailEntityToAdd =
                  Activator.CreateInstance(auditTrailEntityType);
077
078               object deletedEntityValues =
                  deletedEntity.OriginalValues.ToObject();
079               if (deletedEntityValues != null)
080               {
081                 CopyEntityValues(deletedEntityValues,
                    auditTrailEntityToAdd, EntityState.Deleted);
082                 // Save created audittrail entity to audittrail
                       database and save changes.
083                 System.Data.Entity.DbSet auditTrailTable =
                       auditTrailDB.Set(auditTrailEntityType);
084                 auditTrailTable.Add(auditTrailEntityToAdd);
085               }
086             }
087
088             try
089             {
090               baseResult = base.SaveChanges();
091             }
092             catch (Exception exp)
093             {
094               throw new BaseSaveChangesException(
                  "Exception occurred in base.SaveChanges() " +
                  Environment.NewLine, exp);
095             }
```

```
096
097             foreach (DbEntityEntry addedEntity in addedEntities)
098             {
099               Type addedEntityType = addedEntity.Entity.GetType();
100               string addedEntityTypeName =
                  addedEntityType.AssemblyQualifiedName.ToString().
                  Replace("TestDataModel", "AuditTrailDataModel");
101               Type auditTrailEntityType =
                  Type.GetType(addedEntityTypeName);
102               object auditTrailEntityToAdd =
                  Activator.CreateInstance(auditTrailEntityType);
103
104               object addedEntityValues =
                  addedEntity.CurrentValues.ToObject();
105               if (addedEntityValues != null)
106               {
107                 CopyEntityValues(addedEntityValues,
                    auditTrailEntityToAdd, EntityState.Added);
108                 // Save created audittrail entity to audittrail
                       database and save changes.
109                 System.Data.Entity.DbSet auditTrailTable =
                    auditTrailDB.Set(auditTrailEntityType);
110                 auditTrailTable.Add(auditTrailEntityToAdd);
111               }
112             }
113
114             foreach (DbEntityEntry modifiedEntity in
                        modifiedEntities)
115             {
116               Type modifiedEntityType =
                        modifiedEntity.Entity.GetType();
117               string modifiedEntityTypeName =
                  modifiedEntityType.AssemblyQualifiedName.ToString().
                  Replace("TestDataModel", "AuditTrailDataModel");
118               Type auditTrailEntityType =
                  Type.GetType(modifiedEntityTypeName);
119               object auditTrailEntityToAdd =
                  Activator.CreateInstance(auditTrailEntityType);
120
121               object modifiedEntityValues =
                  modifiedEntity.CurrentValues.ToObject();
122               if (modifiedEntityValues != null)
123               {
124                 CopyEntityValues(modifiedEntityValues,
                    auditTrailEntityToAdd, EntityState.Modified);
125                 // Save created audittrail entity to audittrail
                       database and save changes.
126                 System.Data.Entity.DbSet auditTrailTable =
                    auditTrailDB.Set(auditTrailEntityType);
127                 auditTrailTable.Add(auditTrailEntityToAdd);
128               }
129             }
130             auditTrailDB.SaveChanges();
131           }
132           //LOG THE TIME THAT WAS USED TO SAVE AUDITTRAIL
                INFORMATION
133           logger.RecordMessage("AuditTrail logging time: " +
              (DateTime.Now - clock).TotalSeconds);
134         }
135         catch (BaseSaveChangesException baseException)
136         {
```

```
137           logger.RecordMessage(baseException,
              EventLogEntryType.Error);
138         }
139         catch (DbEntityValidationException evex)
140         {
141           logger.RecordMessage(
              Helpers.GenerateEntityValidationErrorMessage(evex));
142         }
143         catch (Exception ex)
144         {
145           string message = String.Format("Saving AuditTrailLog
                failed. {0}\n{1}", ex.Message, ex.StackTrace);
146           logger.RecordMessage(message);
147         }
148       }
149       else
150       {
151         baseResult = base.SaveChanges();
152       }
153       return baseResult;
154     }
155     #endregion
156
157     #region Private methods
158     // Get type of the added entity.
159     // Loop through all properties of this added entity.
160     // Add propertyvalues of added entity to corresponding
        properties in audittrail database
161     // by looping through all properties of selected table in
        audittrail database and placing
162     // each added value to right corresponding audittrail field.
163     private void CopyEntityValues(object addedEntity,
                    object destinationEntity, EntityState state)
164     {
165       Type addedEntityType = addedEntity.GetType();
166       Type auditTrailEntityType = destinationEntity.GetType();
167
168       foreach (var addedEntityProperty in
                    addedEntityType.GetProperties())
169       {
170         var addedEntityPropertyValue =
            addedEntityProperty.GetValue(addedEntity);
171         var auditTrailEntityProperty =
            auditTrailEntityType.GetProperties().Where(
            it => it.Name == addedEntityProperty.Name).
            FirstOrDefault();
172         // auditTrailEntityProperty is null, if there isn't
              corresponding entity property in audittrail database.
173         if (auditTrailEntityProperty != null)
174         {
175           auditTrailEntityProperty.SetValue(destinationEntity,
                                          addedEntityPropertyValue);
176         }
177       }
178
179       //Add the actual audittrail information
180       //1. User who made changes <- NEEDS ACTUAL USER!
181       var auditTrailEntityPropertyUser =
          auditTrailEntityType.GetProperties().Where(
          it => it.Name == "AuditTrailUser").FirstOrDefault();
182       if (auditTrailEntityPropertyUser != null)
```

```
183       {
184         string auditTrailUser = "TestUser";
185         auditTrailEntityPropertyUser.SetValue(
            destinationEntity, auditTrailUser);
186       }
187
188       //2. Time when change was made
189       var auditTrailEntityPropertyTime =
          auditTrailEntityType.GetProperties().Where(
          it => it.Name == "AuditTrailTime").FirstOrDefault();
190       if (auditTrailEntityPropertyTime != null)
191       {
192         DateTime auditTrailDateTimeNow = DateTime.Now;
193         auditTrailEntityPropertyTime.SetValue(
            destinationEntity, auditTrailDateTimeNow);
194       }
195
196       //3. Operation (ADD | UPDATE | DELETE)
197       var auditTrailEntityPropertyOperation =
              auditTrailEntityType.GetProperties().Where(
              it => it.Name == "AuditTrailOperation").FirstOrDefault();
198       if (auditTrailEntityPropertyOperation != null)
199       {
200         string auditTrailOperation =
            Enum.GetName(typeof(DatabaseOperations), state);
201         auditTrailEntityPropertyOperation.
            SetValue(destinationEntity, auditTrailOperation);
202       }
203
204       //4. Add the webservice method name
205       var auditTrailEntityPropertyMethod =
          auditTrailEntityType.GetProperties().Where(
          it => it.Name == "WebServiceMethod").FirstOrDefault();
206       if (auditTrailEntityPropertyMethod != null)
207       {
208         string methodName = "";
209         try
210         {
211           StackFrame frame = new StackFrame(2);
212           methodName = frame.GetMethod().Name;
213         }
214         catch (Exception)
215         {
216           methodName = "Methodname unavailable";
217         }
218         auditTrailEntityPropertyMethod.
            SetValue(destinationEntity, methodName);
219       }
220
221       //5. Generate transaction GUID
222       var auditTrailEntityPropertyTransactionGUID =
          auditTrailEntityType.GetProperties().Where(
          it => it.Name == "TransactionGUID").FirstOrDefault();
223       if (auditTrailEntityPropertyTransactionGUID != null)
224       {
225         auditTrailEntityPropertyTransactionGUID.
            SetValue(destinationEntity, _transactionGuid);
226       }
227     }
228     #endregion
229   }
```

# LIITE B: TIETOKANTAHERÄTIN AUDIT TRAIL -MEKANISMIN KOODI

```
001  USE [TestDataModelTriggers]
002  GO
003
004  SET ANSI_NULLS ON
005  GO
006  SET QUOTED_IDENTIFIER ON
007  GO
008
009  -- PERSON TRIGGERS --
010  CREATE TRIGGER [dbo].[tr_Person_Delete]
011  ON [dbo].[Person]
012  FOR DELETE
013    AS
014    INSERT INTO [AuditTrailDataModelTriggers].
       [dbo].[Person](Id,Firstname,Lastname,Socialnumber,
       Birthday,AuditTrailOperation,AuditTrailTime)
015    SELECT Id,Firstname,Lastname,Socialnumber,
       Birthday,'DELETE',GETDATE()
016    FROM Deleted;
017
018  CREATE TRIGGER [dbo].[tr_Person_Update]
019  ON [dbo].[Person]
020  FOR UPDATE
021    AS
022    INSERT INTO [AuditTrailDataModelTriggers].
       [dbo].[Person](Id,Firstname,Lastname,Socialnumber,
       Birthday,AuditTrailOperation,AuditTrailTime)
023    SELECT Id,Firstname,Lastname,Socialnumber,
       Birthday,'UPDATE',GETDATE()
024    FROM Inserted;
025
026  CREATE TRIGGER [dbo].[tr_Person_Insert]
027  ON [dbo].[Person]
028  FOR INSERT
029    AS
030    INSERT INTO [AuditTrailDataModelTriggers].[dbo].
       [Person](Id,Firstname,Lastname,Socialnumber,
       Birthday,AuditTrailOperation,AuditTrailTime)
031    SELECT Id,Firstname,Lastname,Socialnumber,
       Birthday,'INSERT',GETDATE()
032    FROM Inserted;
033
034  -- PERSON PATIENT TRIGGERS --
035  CREATE TRIGGER [dbo].[tr_Person_Patient_Delete]
036  ON [dbo].[Person_Patient]
037  FOR DELETE
038    AS
039    INSERT INTO [AuditTrailDataModelTriggers].[dbo].
       [Person_Patient]([Key],PatientNumber)
040    SELECT Id,PatientNumber
041    FROM Deleted;
042
043  CREATE TRIGGER [dbo].[tr_Person_Patient_Update]
044  ON [dbo].[Person_Patient]
045  FOR UPDATE
```

```
046     AS
047     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Person_Patient]([Key],PatientNumber)
048     SELECT Id,PatientNumber
049     FROM Inserted;
050
051   CREATE TRIGGER [dbo].[tr_Person_Patient_Insert]
052   ON [dbo].[Person_Patient]
053   FOR INSERT
054     AS
055     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Person_Patient]([Key],PatientNumber)
056     SELECT Id,PatientNumber
057     FROM Inserted;
058
059   -- PERSON_NURSE TRIGGERS --
060   CREATE TRIGGER [dbo].[tr_Person_Nurse_Delete]
061   ON [dbo].[Person_Nurse]
062   FOR DELETE
063     AS
064     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Person_Nurse]([Key])
065     SELECT Id
066     FROM Deleted;
067
068   CREATE TRIGGER [dbo].[tr_Person_Nurse_Update]
069   ON [dbo].[Person_Nurse]
070   FOR UPDATE
071     AS
072     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Person_Nurse]([Key])
073     SELECT Id
074     FROM Inserted;
075
076   CREATE TRIGGER [dbo].[tr_Person_Nurse_Insert]
077   ON [dbo].[Person_Nurse]
078   FOR INSERT
079     AS
080     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Person_Nurse]([Key])
081     SELECT Id
082     FROM Inserted;
083
084   -- PERSON_EMPLOYEE TRIGGERS --
085   CREATE TRIGGER [dbo].[tr_Person_Employee_Delete]
086   ON [dbo].[Person_Employee]
087   FOR DELETE
088     AS
089     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Person_Employee]([Key])
090     SELECT Id
091     FROM Deleted;
092
093   CREATE TRIGGER [dbo].[tr_Person_Employee_Update]
094   ON [dbo].[Person_Employee]
095   FOR UPDATE
096     AS
097     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Person_Employee]([Key])
098     SELECT Id
099     FROM Inserted;
```

```
100
101  CREATE TRIGGER [dbo].[tr_Person_Employee_Insert]
102  ON [dbo].[Person_Employee]
103  FOR INSERT
104    AS
105    INSERT INTO [AuditTrailDataModelTriggers].[dbo].
       [Person_Employee]([Key])
106    SELECT Id
107    FROM Inserted;
108
109  -- PERSON_ADMINISTRATOR TRIGGERS --
110  CREATE TRIGGER [dbo].[tr_Person_Administrator_Delete]
111  ON [dbo].[Person_Administrator]
112  FOR DELETE
113    AS
114    INSERT INTO [AuditTrailDataModelTriggers].[dbo].
       [Person_Administrator]([Key])
115    SELECT Id
116    FROM Deleted;
117
118  CREATE TRIGGER [dbo].[tr_Person_Administrator_Update]
119  ON [dbo].[Person_Administrator]
120  FOR UPDATE
121    AS
122    INSERT INTO [AuditTrailDataModelTriggers].[dbo].
       [Person_Administrator]([Key])
123    SELECT Id
124    FROM Inserted;
125
126  CREATE TRIGGER [dbo].[tr_Person_Administrator_Insert]
127  ON [dbo].[Person_Administrator]
128  FOR INSERT
129    AS
130    INSERT INTO [AuditTrailDataModelTriggers].[dbo].
       [Person_Administrator]([Key])
131    SELECT Id
132    FROM Inserted;
133
134  -- ORGANISATION TRIGGERS --
135  CREATE TRIGGER [dbo].[tr_Organisation_Delete]
136  ON [dbo].[Organisation]
137  FOR DELETE
138    AS
139    INSERT INTO [AuditTrailDataModelTriggers].[dbo].
       [Organisation](Id,Name,AuditTrailOperation,AuditTrailTime)
140    SELECT Id,Name,'DELETE',GETDATE()
141    FROM Deleted;
142
143  CREATE TRIGGER [dbo].[tr_Organisation_Update]
144  ON [dbo].[Organisation]
145  FOR UPDATE
146    AS
147    INSERT INTO [AuditTrailDataModelTriggers].[dbo].
       [Organisation](Id,Name,AuditTrailOperation,AuditTrailTime)
148    SELECT Id,Name,'UPDATE',GETDATE()
149    FROM Inserted;
150
151  CREATE TRIGGER [dbo].[tr_Organisation_Insert]
152  ON [dbo].[Organisation]
153  FOR INSERT
154    AS
```

```
155     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Organisation](Id,Name,AuditTrailOperation,AuditTrailTime)
156     SELECT Id,Name,'INSERT',GETDATE()
157     FROM Inserted;
158
159   -- MEASUREMENT TRIGGERS --
160   CREATE TRIGGER [dbo].[tr_Measurement_Delete]
161   ON [dbo].[Measurement]
162   FOR DELETE
163     AS
164     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Measurement](Id,PatientId,Status,MeasurementDate,
        AuditTrailOperation,AuditTrailTime)
165     SELECT Id,PatientId,Status,MeasurementDate,'DELETE',GETDATE()
166     FROM Deleted;
167
168   CREATE TRIGGER [dbo].[tr_Measurement_Update]
169   ON [dbo].[Measurement]
170   FOR UPDATE
171     AS
172     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Measurement](Id,PatientId,Status,MeasurementDate,
        AuditTrailOperation,AuditTrailTime)
173     SELECT Id,PatientId,Status,MeasurementDate,'UPDATE',GETDATE()
174     FROM Inserted;
175
176   CREATE TRIGGER [dbo].[tr_Measurement_Insert]
177   ON [dbo].[Measurement]
178   FOR INSERT
179     AS
180     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Measurement](Id,PatientId,Status,MeasurementDate,
        AuditTrailOperation,AuditTrailTime)
181     SELECT Id,PatientId,Status,MeasurementDate,'INSERT',GETDATE()
182     FROM Inserted;
183
184   -- ADDRESS TRIGGERS --
185   CREATE TRIGGER [dbo].[tr_Address_Delete]
186   ON [dbo].[Address]
187   FOR DELETE
188     AS
189     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Address](Id,StreetAddress,PostalNumber,City,
        AuditTrailOperation,AuditTrailTime)
190     SELECT Id,StreetAddress,PostalNumber,City,'DELETE',GETDATE()
191     FROM Deleted;
192
193   CREATE TRIGGER [dbo].[tr_Address_Update]
194   ON [dbo].[Address]
195   FOR UPDATE
196     AS
197     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Address](Id,StreetAddress,PostalNumber,City,
        AuditTrailOperation,AuditTrailTime)
198     SELECT Id,StreetAddress,PostalNumber,City,'UPDATE',GETDATE()
199     FROM Inserted;
200
201   CREATE TRIGGER [dbo].[tr_Address_Insert]
202   ON [dbo].[Address]
203   FOR INSERT
204     AS
```

```
205     INSERT INTO [AuditTrailDataModelTriggers].[dbo].
        [Address](Id,StreetAddress,PostalNumber,City,
        AuditTrailOperation,AuditTrailTime)
206     SELECT Id,StreetAddress,PostalNumber,City,'INSERT',GETDATE()
207     FROM Inserted;
```