Opinnäytetyö (AMK)

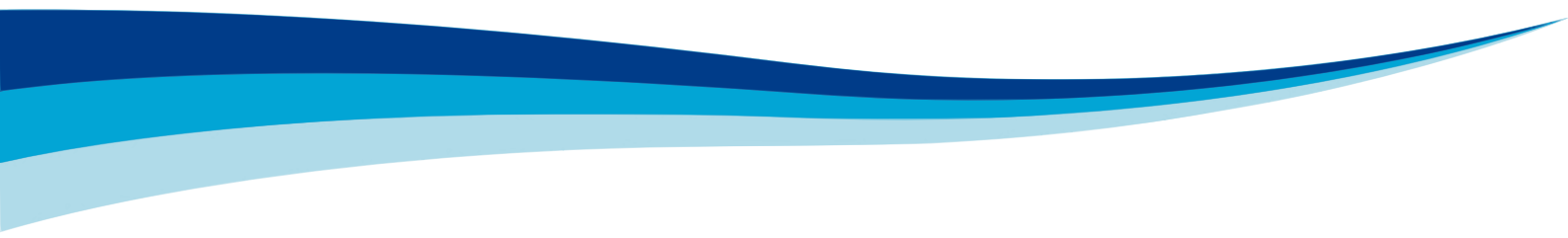
Tietotekniikka

Sulautetut ohjelmistot

2014

Tatu Soukka

LISENSSIAVAINHALLINTA-SOVELLUKSEN SUUNNITTELU JA TOTEUTUS



TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Sulautetut ohjelmistot

2014 | 44 sivua

Tiina Ferm

Tatu Soukka

LISENSSIAVAINHALLINTASOVELLUKSEN SUUNNITTELU JA TOTEUTUS

Opinnäytetyössä suunniteltiin ja toteutettiin keskitetty lisenssiavainhallintasovellus, joka korvaa nykyiset hajautetut hallintaratkaisut. Tavoitteena on käyttöohjeistuksettakin helposti lähestyttävä sovellus.

Teoriaosuus keskittyi ajoympäristön muistinhallintaan, käyttöliittymän arkkitehtoniseen malliin, oliosuhdekartoitukseen ja yksikkötestaukseen.

Sovellus toteutettiin C#-ohjelmointikielellä ja se käyttää Microsoft SQL Server -relaatiotietokantajärjestelmää tiedon säilytykseen. NHibernate-kirjastoa käytetään oliosuhdekartoitukseen. Käyttöliittymä toteutetaan hyödyntäen MVVM-mallia, joka on WPF-alijärjestelmälle suunniteltu arkkitehtoninen malli. Toteutusvaiheen avustukseen ja virheiden paikallistamiseen käytetään yksikkötestejä. Ohjelmakoodin ja projektitehtävien hallintaan käytetään TFS-projektinhallintajärjestelmää. Sovellus pyörii sisäisessä verkkoalueessa sijaitsevassa palvelimessa, johon tarvitsee ottaa RDP-etätyöpöytäprotokollayhteys. Käyttäjä tarvitsee pääsyoikeudet palvelimen lisäksi erikseen myös tietokantaan. Varmuuskopiot tietokannoista otetaan päivittäin ja niihin on luku- ja kirjoitusoikeudet ainoastaan järjestelmänvalvojalla.

Suurin haaste sovelluksen toteutuksessa oli ominaisuuksien rajaus ja intuitiivisen käyttöliittymän suunnittelu. Liiallinen abstrahointi hidasti toteutusta, mutta helpotti ohjelmakirjastojen vaihtoa. Tulevaisuudessa sovelluksesta voisi kehittää www-pohjaisen, mitä kautta asiakkaat luovat lisenssiavaimen tarvittaessa. Tämä vähentäisi lisenssihallinnan tuottamaa kuormaa. Isoksi haasteeksi muutoksessa tulisi tietoturva ja sen varmentaminen.

ASIASANAT:

C#, MVVM, WPF, SQL, muistinhallinta, roskankeruu, oliosuhdekartoitus, yksikkötestaus, TDD

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information technology | Embedded software

2014 | 44 pages

Tiina Ferm

Tatu Soukka

LICENSE KEY MANAGER DESIGN AND IMPLEMENTATION

The purpose of this thesis is to design and implement a centralized license key management application that replaces the currently used management solutions. The application should be easy to use even without user manual.

Theory focuses on memory management of the runtime environment, architectural model of user interface, object-relational mapping, and unit testing.

C# programming language is used to implement the application and it uses Microsoft SQL Server relational database management system for data storage. Object-relational mapping is handled by NHibernate class library. MVVM architectural model, which is designed for the WPF graphical subsystem, is used for implementing the user interface. Unit tests are used to support the implementation phase and to detect programming errors. TFS project management system is used for managing the source code and project tasks. The final application is run from a local domain server via a remote desktop protocol connection. User requires separate permissions to the server and the database. Daily backups are taken from the database and only system administrator has read and write access to the backups.

Biggest challenge during development was feature creep and designing intuitive user interface. Excessive abstraction slowed down the implementation phase, but in the end made it easy to replace libraries when necessary. In the future, the load created by license creation could be reduced by introducing web-based interface that customers could use to create licenses whenever needed. Data security would be the greatest challenge in this change.

KEYWORDS:

C#, MVVM, WPF, SQL, memory management, GC, object-relational mapping, unit testing, TDD

SISÄLTÖ

KUVAT

KUVIOT

KOODIT

SANASTO

BAML	XAML käännetään BAML-muotoon. (Binary Application Markup Language)
CIL	.NET-alustan virtuaaliympäristön käyttämä tavukoodi. (Common Intermediate Language)
CLI	.NET-alustan tekniset määrittelyt. (Common Language Infrastructure)
CLR	Microsoftin toteutus CLI:n VES-virtuaalikoneesta. (Common Language Runtime)
CRUD	Pysyvän varastoinnin neljä perustoimintoa: luoda, lukea, päivittää ja poistaa. (Create, Read, Update, Delete)
CSC	.NET-alustan C#-kääntäjä. (C-Sharp Compiler)
CTS	.NET-alustan arvotyyppijärjestelmä. (Common Type System)
DB	Tietokanta (engl. database).
DSV	Tekstimuodossa erotinmerkillä eroteltu taulukon sisältö. (Delimiter Separated Values)
GC	Automaattisen muistinhallinnan muistin vapautusrutiini, jota kutsutaan roskankeruuksi. (Garbage Collector)
GUI	Graafinen käyttöliittymä. (Graphical User Interface)
JIT	Just-In-Time tarkoittaa juuri ennen ohjelman käynnistystä tapahtuvaa natii-vikoodikäännöstä.
JVM	Java-alustan virtuaaliympäristö. (Java Virtual Machine)
LINQ	.NET-alustan datakyselykomponentti. (Language Integrated Query)
LOH	Yli 85 kt:n objekteille tarkoitettu keko C#-ohjelmaprosessissa. (Large Object Heap)
MVVM	WPF:lle suunniteltu arkkitehtoninen malli. (Model-View-View Model)
RDP	Microsoftin kehittämä etätyöpöytäprotokolla Windows-alustalle. (Remote Desktop Protocol)
SQL	Relaatiotietokantakyselykieli. (Structured Query Language)
TDD	Testivetoinen ohjelmistokehitys. (Test Driven Development)
TFS	Microsoftin sovelluksen elämänkaaren hallintajärjestelmä. (Team Foundation Server)
UI	Käyttöliittymä. (User Interface)
VES	CLI:n virtuaalikone. (Virtual Execution System)
WPF	.NET-alustan vektorigrafiikkaan pohjautuva graafinen alijärjestelmä. (Windows Presentation Foundation)
XAML	XML-pohjainen graafisten rakenteiden kuvauskieli. (eXtensible Application Markup Language)
XML	Tekstimuotoinen kuvauskieli. (eXtensible Markup Language)

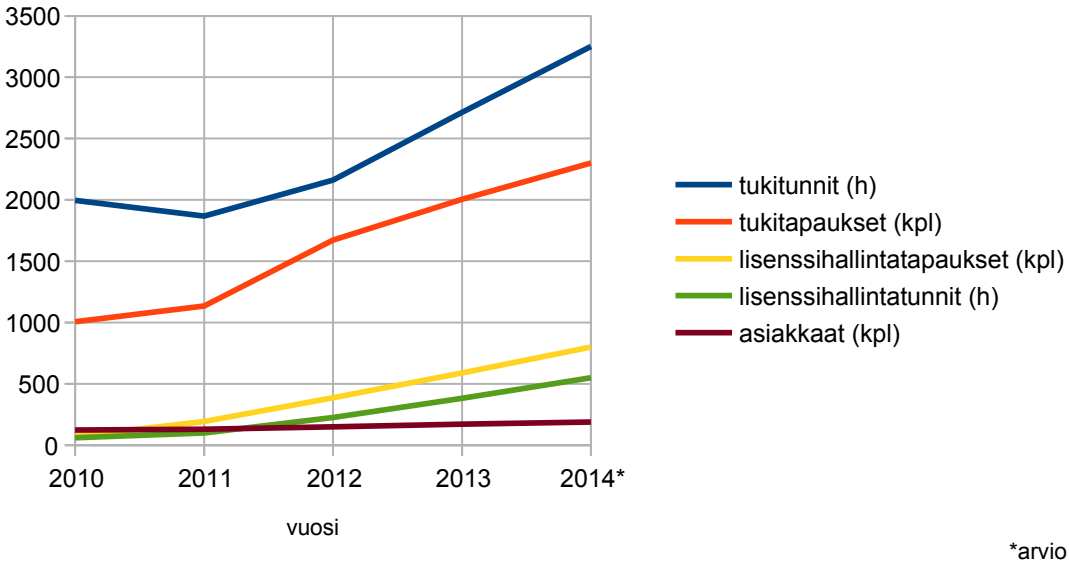
1 JOHDANTO

Työn toimeksiantaja on Wallac Oy:n Health Informatics Software Services -ohjelmistotukiryhmä (lyh. HISS). HISSin vastuualueeseen kuuluu geneettisten sairauksien seulontaa avustavien ohjelmistojärjestelmien hallinta. Wallac Oy on osa PerkinElmer konsernia ja työllistää noin 500 työntekijää PerkinElmerin maailmanlaajuisesta noin 7 500 työntekijästä [1][2].

Opinnäytetyön päätarkoituksena on helpottaa ja yksinkertaistaa nykyistä lisenssiavainten hallintaa. Tietokanta-osiossa mainittu aktivointiavain on synonyymi lisenssiavaimelle, jonka tarkoitus on selventää ohjelmistolisenssisopimuksen ja ohjelmiston aktivointiin tarkoitetun lisenssiavaimen ero asiakkaille. Lisenssiavaimella tässä tapauksessa tarkoitetaan merkkijonoa tai tiedostoa, jolla poistetaan koeajettavan tuotteen aika- tai ominaisuusrajoitteet.

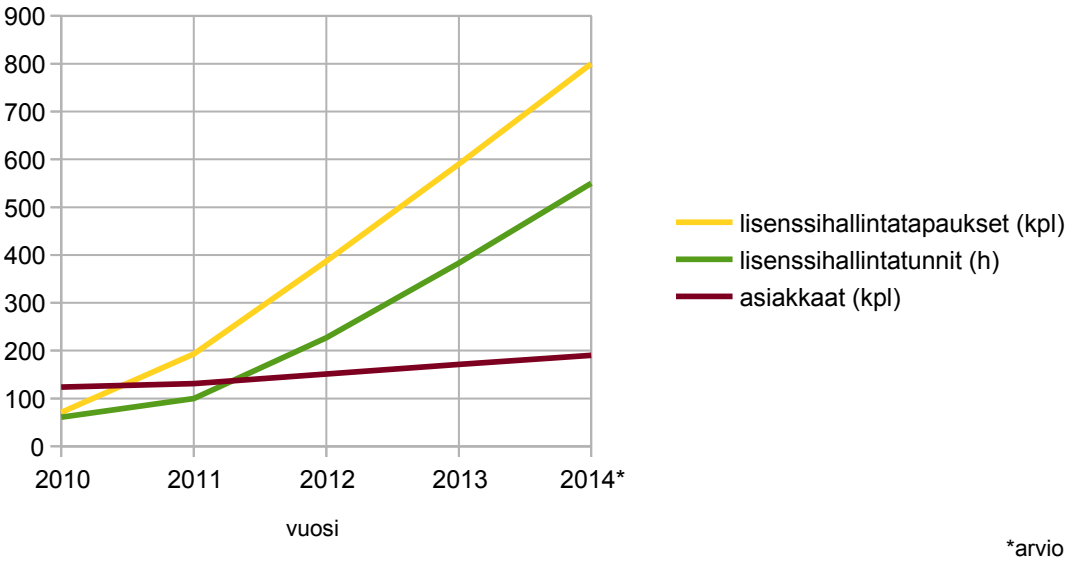
Lisenssiavain luodaan ohjelmistoasennuksen yhteydessä asiakkaan tilaamalle koneelle. Jos asennus tehdään kentällä asiakkaan luona, kenttäinsinööri kyselee lisenssiä HISSiltä. Jälkeenpäin lisenssiavainta uusittaessa asiakas lähettää sähköpostitse arkistoitavan lisenssiavainhakemuslomakkeen, jossa kerrotaan asiakkaan yhteystietojen lisäksi lisenssiavaimen luontiin tarvittavat tiedot. Asiakkaan tiedot tarkistetaan, minkä jälkeen uusi lisenssiavain luodaan ja lähetetään asiakkaalle sähköpostitse. Hallintasovellus tulee työvuossa kuvaan asiakkaan tietoja tarkistettaessa ja lisenssiavaimia luotaessa ja uusittaessa. Lisenssihallinnan osuus tukitehtävistä kasvaa vuosittain tasaisesti asiakasympäristöjen laajentuessa. Tästäpä syystä lisenssihallinnan nopeuttaminen on tärkeää ottaa huomioon sovellusta kehitettäessä.

Kuvio 1:ssä nähdään tukitapausten kehitys vuosien 2010 ja 2014 välillä. Uusien asiakasympäristöjen suunnittelut ja toteutukset tehdään erillisinä projekteina, joten niihin käytetyt työtunnit eivät sisälly tukitapaustilastoihin.



Kuvio 1. HISSin tukitapaukset v. 2010–2014.

Lähempää lisenssihallintatapauksia tarkasteltaessa (Kuvio 2) huomataan, kuinka asiakasympäristöjen laajeneminen vaikuttaa lisenssihallinnan vaatimaan aikaan asiakasmäärään nähden.



Kuvio 2. Lisenssihallintatapaukset v. 2010–2014.

Tällä hetkellä tuotteet käyttävät laidasta laitaan erilaisia lisenssiavainhallintaratkaisuja. Joidenkin tuotteiden lisenssihallintaan käytetään Excel-laskentataulukkoja ja toisissa tietokantapohjaisia asiakassovelluksia. Lisenssihallintaratkaisujen väliset eroavaisuudet hankaloittavat lisenssiavainhallintaa ja uusien työntekijöiden perehdyttämistä. Osa ratkaisuista ei myöskään skaalaudu hyvin lisenssiavainmäärien kasvaessa, joten nämä tarvitsee kuitenkin korvata ennemmin tai myöhemmin.

Tarkoituksena on luoda yhtenäinen hallintasovellus kaikille nykyisille ja tuleville lisenssiavaimille. Uusien tuotteiden kohdalla lisenssiavainhallinta pysyisi samankaltaisena ja lisäohjeistuksen tarve vähenisi. Lisenssiavaintiedot löytyvät myös vaivattomammin, kun ne on kaikki koottu yhteen paikkaan.

Alun perin olisi ollut mahdollista kehittää jostain vanhasta hallintaratkaisusta uudistettu sovellus, mutta parhaimmat sovellusvaihtoehdot tähän käyttävät vanhentuneita alustoja ja tulevaisuutta ajatellen vaikutti paremmalta vaihtoehdolta aloittaa puhtaalta pöydältä, vaikka se pidentääkin moninkertaisesti kehitysvaihetta. Vanhojen sovellusten hyödyllisiä piirteitä kuitenkin siirretään uuteen sovellukseen, jotta sen sisäistämiseen ja käyttäjien ohjeistamiseen menisi vähemmän aikaa.

Sovellus toteutetaan C#-ohjelmointikielellä ja se tulee käyttämään Microsoft SQL Server -relaatiotietokantajärjestelmää tiedon tallentamiseen. Molempia teknologioita käytetään entuudestaan työpaikalla, joten tarvittavat työkalut ja tietotaito löytyvät sieltä.

Sovelluksen tulisi olla helppokäyttöinen, jotta käyttöohjeistus pysyisi yksinkertaisena ja sovelluksen käyttö vähäisin opastuksin olisi mahdollista. Käyttöohjeistusta ei kuitenkaan tulisi laiminlyödä, mutta jos yksinkertaisimmankin toiminnon ymmärtämiseen tarvitaan perusteellinen opastus, on käyttöliittymässä todennäköisesti parantamisen varaa. Ohjeistuksella ei kannata korvata huonoa käytettävyyttä, vaan pohjimmainen ongelma tulisi korjata.

Oikeanlainen ajankäyttö kehitysvaiheessa on tärkeää ohjelmiston käytettävyyden kannalta. Tarpeelliset ohjelman ominaisuudet saattavat jäädä puutteellisiksi, jos liikaa aikaa panostetaan toissijaisiin ominaisuuksiin. Puutteelliset ominaisuudet vaikuttavat suoraan negatiivisesti ohjelman käytettävyyteen. Yksi tähän liittyvä haaste on valita oikeat ohjelmakomponentit, joiden uudelleenkäytettävyyteen kannattaa panostaa. Useimmiten on helpompaa tehdä komponentista ensin rajoittuneempi versio ja parannella sitä myöhemmässä vaiheessa. Silloin hahmottuu paremmin, mitkä komponentin osat tarvitsee muuttaa, jotta komponenttia pystyisi käyttämään muualla tai olisiko järkevämpi loppujen lopuksi tehdä erillinen komponentti.

Teoriaosuus keskittyy C#-ohjelmointikielen muistinhallintaan, MVVM-malliin, oliorelaatiokartoitukseen ja yksikkötestaukseen.

Ajoympäristön muistinhallinnan tunteminen on tärkeää, vaikkei sitä jokapäiväisessä ohjelmoinnissa tarvitsisikaan. Ennemmin tai myöhemmin tulee tilanteita, jolloin muistinhallinnan tuntemisella pystyy ennaltaehkäisemään suorituskyvyllisiä ongelmia.

Käyttöliittymien arkkitehtoniset mallit useasti käytännössä monimutkaistavat muuten yksinkertaisten ominaisuuksien toteutusta. Niiden tunteminen kuitenkin antaa osviittaa modulaaristen käyttöliittymäkomponenttien toteuttamiseen. MVVM-mallilla on omat hyötynsä käyttöliittymän erottamisesta liiketoiminnallisesta kerroksesta, mutta sitäkään ei tulisi sokeasti noudattaa, vaan soveltaa loogisissa tilanteissa.

Relaatiotietokantajärjestelmä ja olio-ohjelmointi eivät suoraan toimi keskenään, vaan niiden välille tarvitsee rakentaa jonkinlainen sovitin. Nykyisin ORM on yleistynyt relaatiotietokantatiedon sovittamiseen olioympäristöön suoran tietojoukkojen käsittelyn sijaan.

Kriittisten komponenttien toiminta on testattava läpikotaisin. Yksikkötestien avulla pystytään vähentämään manuaalisen testauksen tarvetta komponenttien osalta ja varmistamaan, ettei refaktorointi aiheuta muutoksia lopputulokseen.

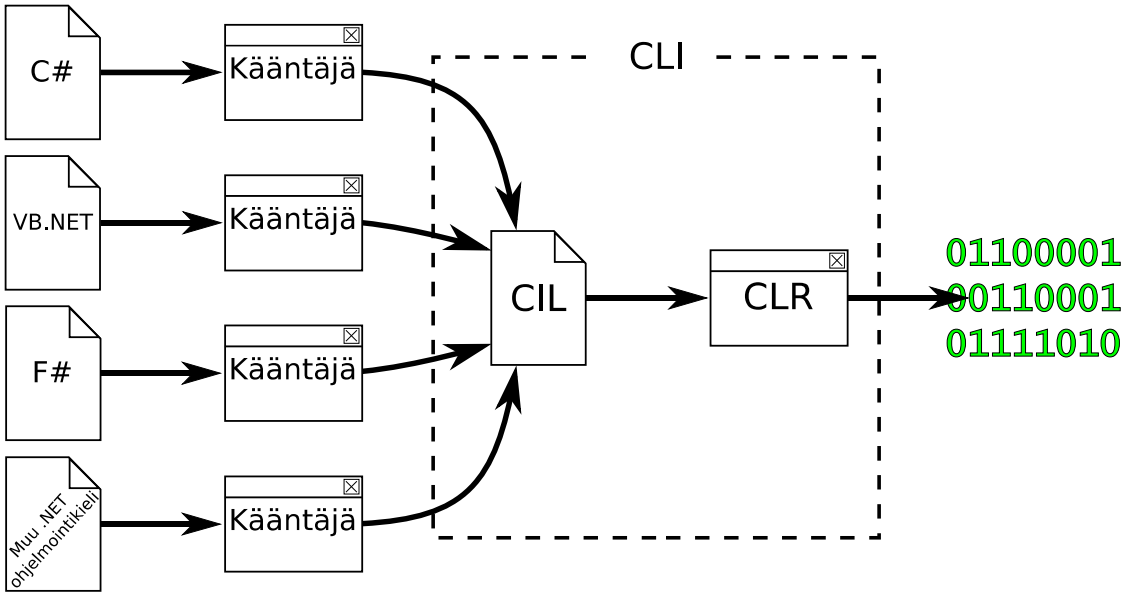
2 RELAATIOTIETOKANTAPOHJAISEN OLIO-OHJELMAN KOMPONENTTEJA

Sovelluksen toteuttamiseen sisältyy ohjelmointikieleen perehtymisen lisäksi ajo-ympäristön tunteminen sekä siinä käytetyt ohjelmakirjastot ja erinäiset suunnittelu- ja toteutusmallit.

2.1 C#-ohjelmointikielen muistinhallinta

C# (engl. laus. see sharp) on Microsoftin kehittämä ja ylläpitämä staattisesti tyypitetty olio-ohjelmointikieli. Kieli on osa Windowsille suunniteltua .NET-alustaa, joka pohjautuu CLI:hin. [3] CLI:n määrittelyihin kuuluu muun muassa yhdistetty tyyppijärjestelmä, tavukoodirakenne ja virtuaaliajojärjestelmä, jotka mahdollistavat ohjelmistokirjastojen yhteentoimivuuden eri CLI-ohjelmointikielien välillä. [4]

Ohjelmakoodi käännetään kielispesifisen kääntäjän avulla CIL-tavukoodiksi. Ohjelman käynnistyksen yhteydessä CLR-virtuaaliajojärjestelmä muuttaa tavukoodin prosessoriarkkitehtuurille sopivaksi natiivikoodiksi. Kuva 1 auttanee hahmottamaan prosessia. [4]



Kuva 1. Pelkistetty visualisointi ohjelmakoodin työnkulusta natiivikoodiksi.

CLR on periaatteessa verrattavissa JVM:ään, mutta se ei tee profilointiavusteista ajonaikaista dynaamista optimointia JVM:n tapaan, vaan lopullinen optimointi tehdään JITin yhteydessä ennen ohjelmalogiikan suoritusta. [5][6]

Ohjelman muistialueet jaetaan pääsääntöisesti kahteen ryhmään, pinoon (engl. stack) ja kekoon (engl. heap). Pino sisältää pääosin ohjelmasuorituksen kulkuun tarvittavaa tietoa ja keko taasen sisältää itse viitatun datan. Jokaiselle ohjelmasäikeelle on oma pino ja säikeet jakavat keon keskenään. [7][8]

Viittaustyypeille muisti varataan aina keosta ja arvotyypeille sieltä missä ne on määritelty. Pelkistettynä muisti varataan useimmiten pinosta, jos arvotyyppi on metodin paikallinen muuttuja, ja muussa tapauksessa keosta. [7]

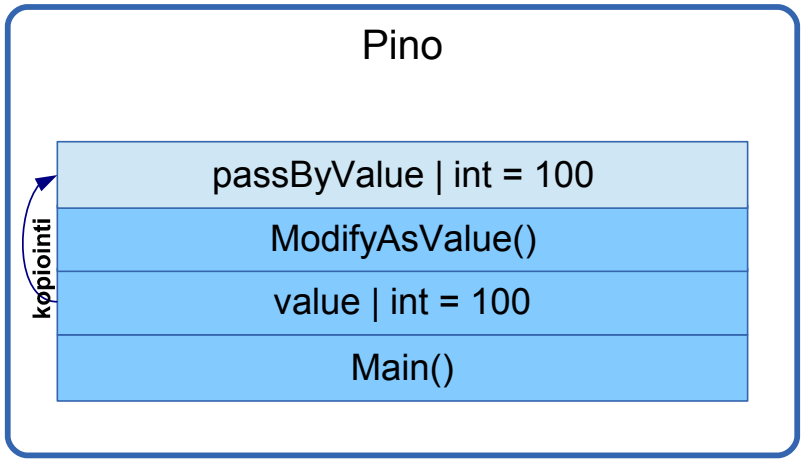
2.1.1 Muistinvaraus

Arvotyyppien kopioitumista voi kiertää määrittämällä metodiparametrin eteen ref-avainsanan [7]. Koodi 1:ssä nähdään, miten ref-avainsana vaikuttaa arvon käsittelyyn.

Koodi 1. Esimerkki arvotyyppisten metodiparametrien käyttäytymisestä.

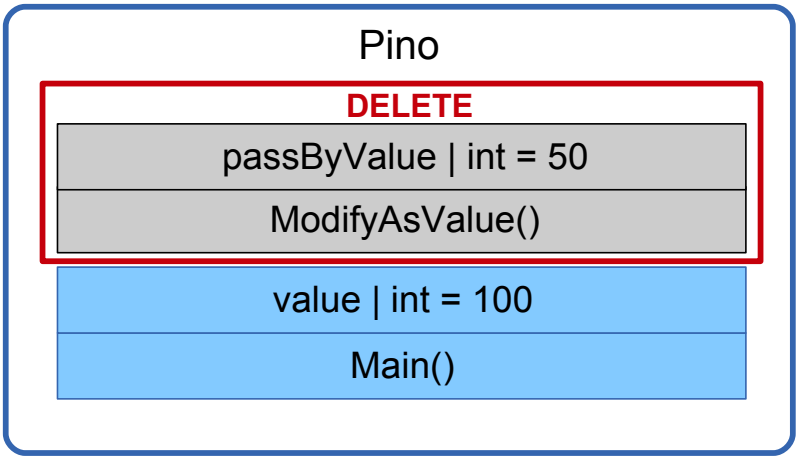
```
static void Main(string[] args) {
   int value = 100;
   Console.WriteLine(value); // 100

   ModifyAsValue(value);
   Console.WriteLine(value); // 100

   ModifyAsRef(ref value);
   Console.WriteLine(value); // 50

   Console.ReadKey();
}

private static void ModifyAsValue(int passByValue) {
   passByValue = 50;
}

private static void ModifyAsRef(ref int passByRef) {
   passByRef = 50;
}
```

Kuvasarja (Kuva 2 - Kuva 4) esittää Koodi 1:n pinon rakennetta ohjelmasuorituksen eri vaiheilla.



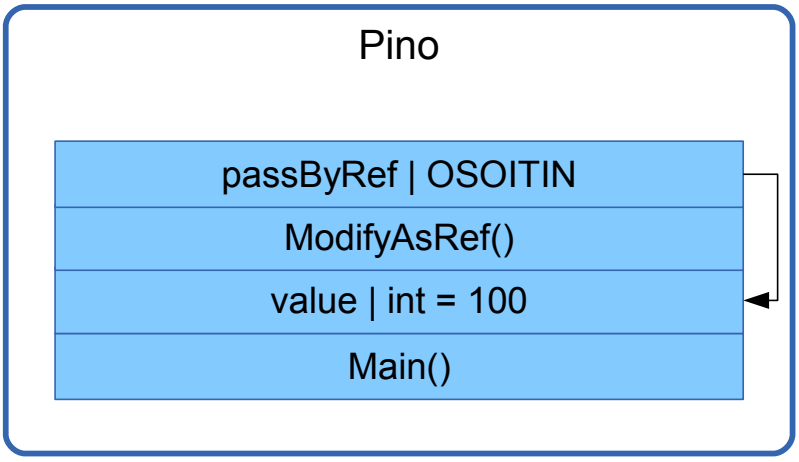
Kuva 2. Muuttujan arvo kopioidaan metodin parametriksi.

`ModifyAsValue`-metodiin syötetyn muuttujan arvo kopioidaan ja kopiota muutetaan. Alkuperäinen muuttuja säilyy entisellään, koska se viittaa eri muistipaikkaan.



Kuva 3. Metodista poistuttaessa sen tiedot poistetaan pinosta.

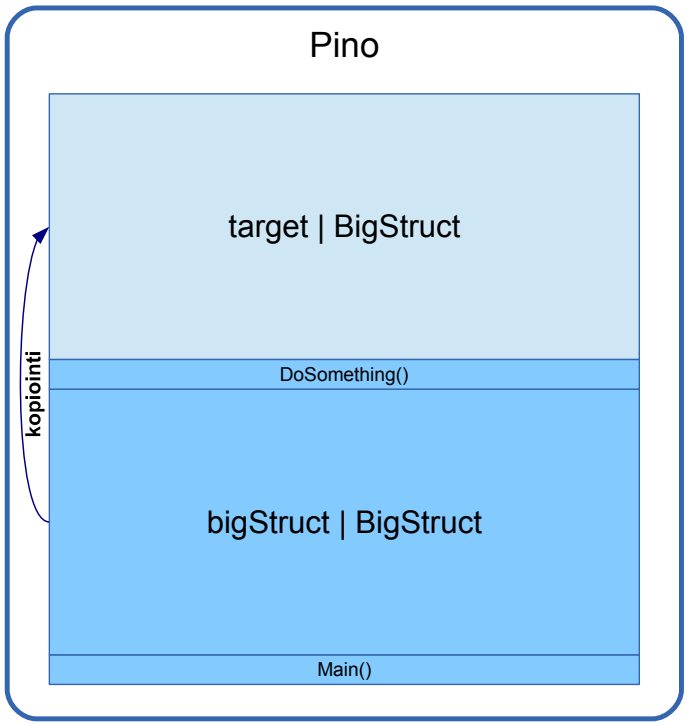
Metodista poistuttaessa pinosta poistetaan kaikki metodikutsuun liittyvä tieto.



Kuva 4. Ref-parametrit viittaavat alkuperäiseen muuttujaan.

Muuttujan arvoa ei kopioida, jos ref-avainsanaa käytetään, vaan muuttujaan tehdään suora viittaus, minkä vuoksi alkuperäinen arvo muuttuu.

Kuva 5 korostaa arvotyyppien kopioitumisen merkitystä suurikokoisten tietueiden käsittelyssä.




```
public struct BigStruct {
   public long a, b, c,
        d, e, f, g, h
        /* ... y, z */;
}
```

Kuva 5. Tietueet kopioituvat metodiin syötettäessä.

Metodeihin syötetyt tietueet kopioituvat muiden arvotyyppimuuttujien tapaan. Tästäpä syystä isojen tietueiden syöttämistä ilman ref-avainsanaa syvään metodikutsuhierarkiaan tulisi välttää.

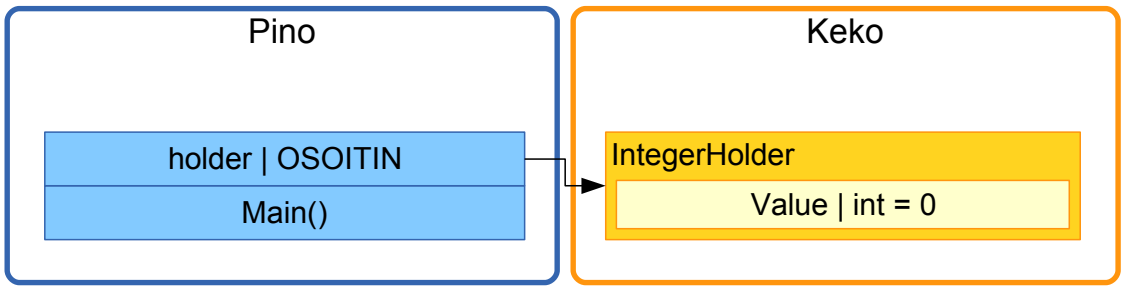
Pinolle varattu muisti on oletuksena 1 Mt:n kokoinen [9]. Pinon oletuskokoa ei tavallisesti kannata kasvattaa, koska se hidastaa säikeiden luontia ja yleisin syy pinon muistin loppumiseen on ohjelmointivirheistä tapahtuvat ikuiset silmukat tai rekursiot eikä niinkään varatun muistin puutteellisuus. [10]

Koodi 2 esittää viittaustyyppien eron arvotyypeistä. Viittaustyyppien muisti varataan aina keosta, joten pinossa on vain viittaus keossa säilytettävään objektiin.

Koodi 2. Esimerkki muistinvarauksesta keossa.

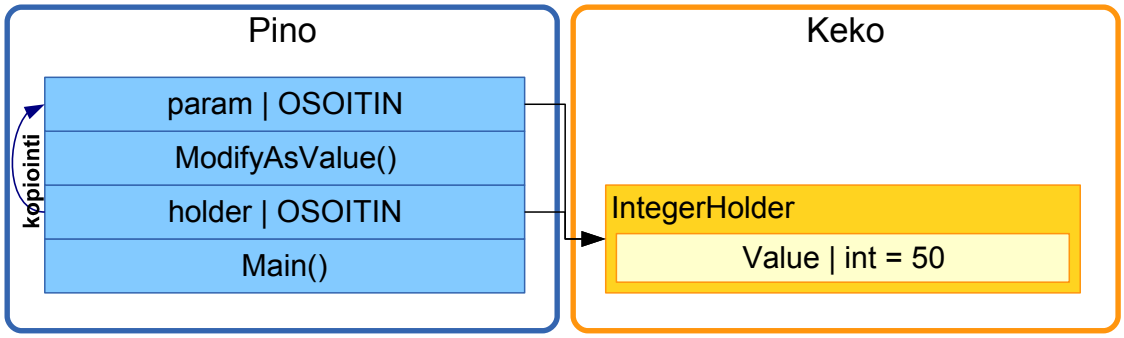
```
static void Main(string[] args) {
   IntegerHolder holder = new IntegerHolder();

   ModifyAsValue(holder);
   Console.WriteLine(holder.Value); // 50

   ModifyAsRef(ref holder);
   Console.WriteLine(holder.Value); // 100

   AssignNewAsValue(holder);
   Console.WriteLine(holder.Value); // 100

   AssignNewAsRef(ref holder);
   Console.WriteLine(holder.Value); // 0
}

private static void ModifyAsValue(IntegerHolder param) {
   param.Value = 50;
}

private static void ModifyAsRef(ref IntegerHolder param) {
   param.Value = 100;
}

private static void AssignNewAsValue(IntegerHolder param) {
   param = new IntegerHolder();
}

private static void AssignNewAsRef(ref IntegerHolder param) {
   param = new IntegerHolder();
}

public class IntegerHolder {
   public int Value { get; set; }

   public IntegerHolder(int value = 0) {
      Value = value;
   }
}
```

Kuvasarja (Kuva 6 - Kuva 10) esittää Koodi 2:n objektiviittauksen käyttäytymistä pinon ja keon välillä ohjelmasuorituksen eri vaiheissa.



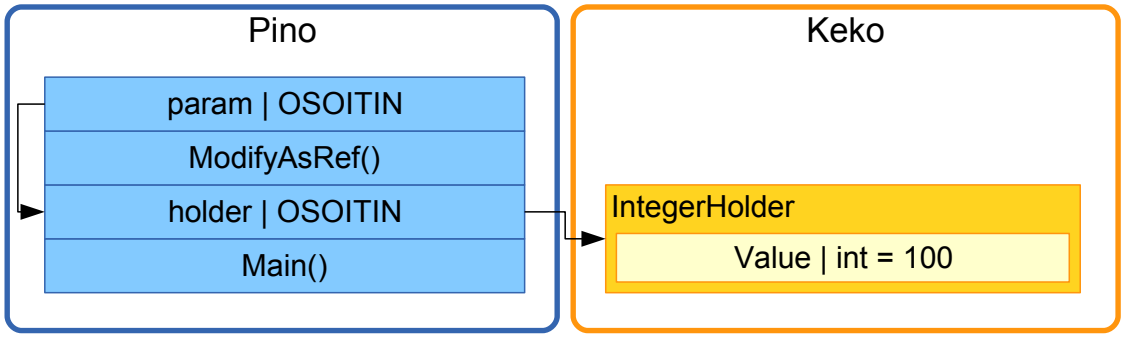
Kuva 6. Viittaustyypit säilytetään keossa.

Päämetodissa luodaan uusi `IntegerHolder`-luokan instanssi, joka säilytetään keossa ja pinossa on vain osoitin siihen.



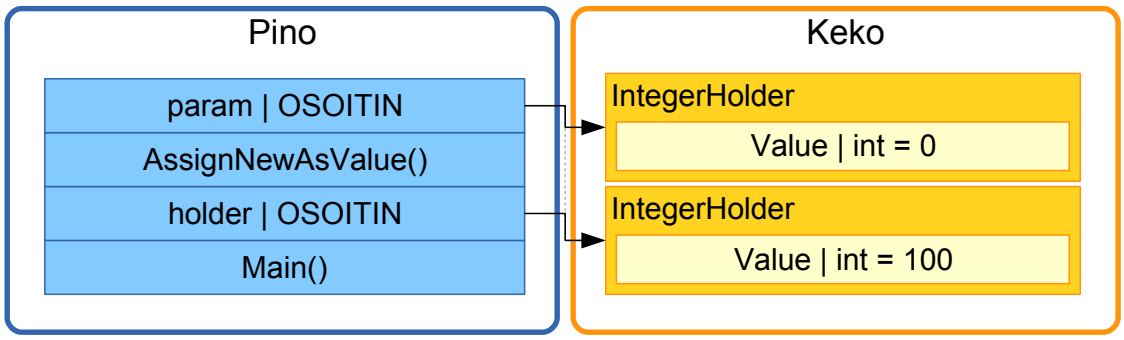
Kuva 7. Kopioitu osoitin viittaa samaan objektiin.

Osoitin kopioidaan `ModifyAsValue`-metodin parametriin. Alkuperäisen objektin arvo muuttuu, koska molemmat osoittimet viittaavat siihen.



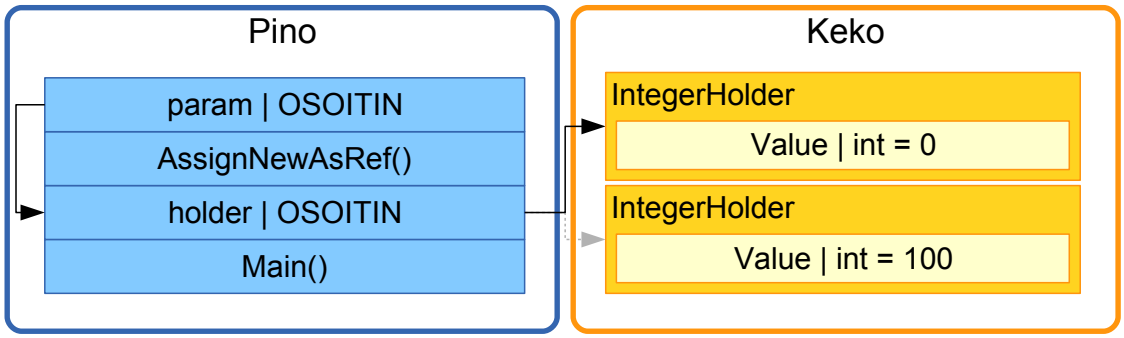
Kuva 8. Parametrin osoitin viittaa alkuperäiseen osoittimeen.

`ModifyAsRef`-metodin sisällä viitataan alkuperäiseen osoittimeen, mikä ei tässä tilanteessa vaikuta toiminnallisuuteen mitenkään.



Kuva 9. Alkuperäisen osoittimen viittaus ei muutu.

`AssignNewAsValue`-metodin sisällä luodaan uusi objekti, mutta koska parametrin osoitin on kopio, ei alkuperäinen metodin ulkopuolella esitelty viittaus muutu.



Kuva 10. Alkuperäinen objektiviittaus vaihtuu.

`AssignNewAsRef`-metodin osoitin on suora viittaus alkuperäiseen osoittimeen, minkä vuoksi alkuperäinen viittaus vaihtuu uuteen metodin sisällä luotuun objektiin.

2.1.2 Roskankeruu

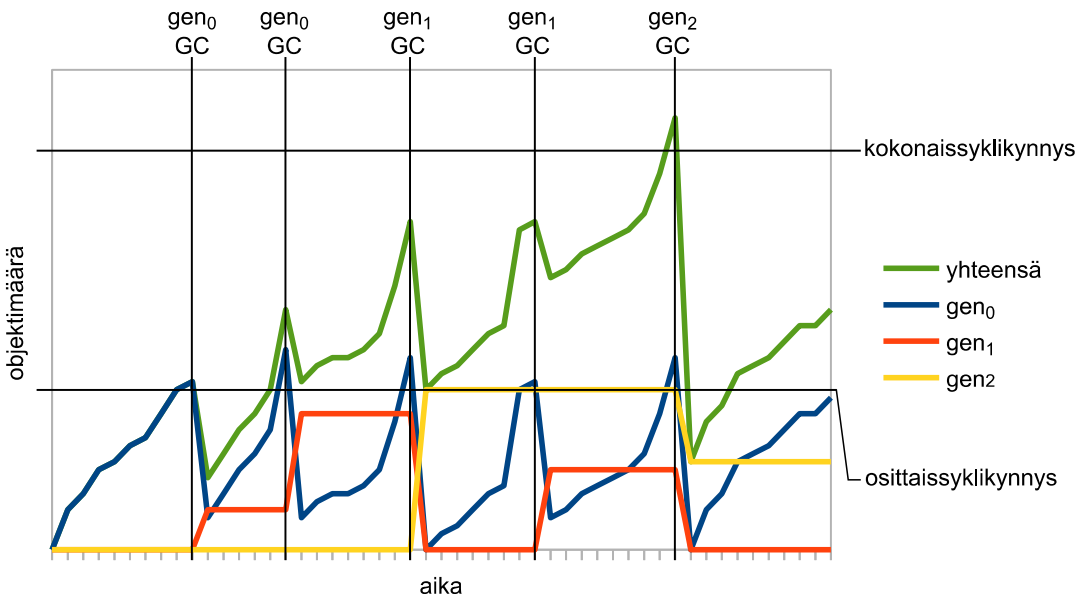
GC pitää kirjaa objekteista ja vapauttaa käyttämättömien objektien varaaman muistin. Roskankeruun nopeuttamiseksi on objektit jaoteltu kolmeen sukupolveen, gen_0–gen_2. Ensimmäiseen luodaan uudet objektit ja kolmas sukupolvi on varattu pitkäaikaisille objekteille. Poikkeuksena yli 85 kt:n objektit säilytetään LOHissa. [8]

Suurin osa roskankeruusta kohdistuu ensimmäiseen sukupolveen, koska yleisesti suurin osa objekteista on lyhytaikaisia ja niiden siivous vapauttaa useimmiten tarpeeksi muistia ohjelmasuorituksen jatkamiseen. Roskankeruu kohdistetaan seuraaviin sukupolviin ainoastaan, jos aikaisemman siivoaminen ei vapauta riittävästi muistia. Kolmannen sukupolven siivous on huomattavasti hitaampaa kuin ensimmäisen tai toisen, koska tällöin kaikki objektit kartoitetaan. Kolmannen sukupolven siivousta kutsutaan kokonaissykliksi ja vastaavasti ensimmäisen ja toisen sukupolven siivousta kutsutaan osittaissykliksi [8]. Osittaissyklissä roskankeruu pystyy kirjoitusesteiden (engl. write barrier) avulla jättää väliin muuttumattomien objektien käsittelyn. [11]

Siivouksesta selvinneet objektit korotetaan seuraavaan sukupolveen, millä mahdollisesti vähennetään seuraavassa roskankeruusyklissä läpikäytävien objektien määrää. Roskankeruu suoritetaan, kun järjestelmämuisti on vähissä, objektien varaama muisti keossa ylittää tietyn kynnyksen tai GC.Collect-metodia kutsutaan. Ajoympäristö säätää muistinvarauskynnystä automaattisesti ohjelmaprosessin olosuhteiden mukaan. [8]

Kuvio 3:ssa nähdään objektien jakauma teoreettisessa ohjelmasuoritus skenaariossa eri roskankeruukierrosten jälkeen.

Oletuksena GC.Collect-metodikutsu suorittaa kokonaissyklin, mikä pahimmillaan ennenaikaisesti korottaa objektit pitkäaikaisiksi, mikä mahdollisesti hidastaa seuraavaa kokonaissykliä, jos korotettuja objekteja on kertynyt tarpeeksi. [12]



Kuvio 3. Yksinkertaistettu teoreettinen GC-skenaario.

Selvittääkseen mitkä objektit ovat käyttämättömiä, roskankeruurutiini pyytää virtuaaliajoympäristöltä juurissa viitatut objektit ja näiden perusteella rakentaa graafin käytetyistä objekteista rekursiivisesti läpikäyden objektiviittaukset. Tämän jälkeen käyttämättömät objektit vapautetaan ja jäljelle jäävät objektit tiivistetään keon muistialueen pirstoutumisen vähentämiseksi. Poikkeuksena LOHia ei tiivistetä, koska isojen objektien siirtäminen on suorituskyvyllisesti kallis operaatio. Lopuksi roskankeruu korjaa muuttuneet objektiviittaukset oikeisiin muistipaikkoihin. Kaikki ohjelman säikeet pysäytetään roskankeruun ajaksi, mikä useasti esiintyy katkoksina, jos roskankeruussa kestää tavanomaista kauemmin. [8]

Tavanomaisia muistivuotoja ei roskankeruun ansiosta tapahdu, mutta on olemassa tilanteita joissa pidempiaikainen objekti pitää kirjaa lyhyempiaikaisista objekteista. Muistivuoto esiintyy helposti tapahtumaseurantalistoille huomiotta jääneistä objekteista. [13]

2.2 WPF ja MVVM

WPF on .NET-alustan vektorigrafiikkaan pohjautuva alijärjestelmä, joka on vaihtoehtoinen valinta vanhemmalle rasterigrafiikkaan perustuvalle WinForms-alijärjestelmälle. Käyttöliittymän rakenteet määritellään XAML-kuvauskielellä elementtipuuhierarkiassa [14]. Käännettäessä XAML muutetaan BAML:ksi ja sulautetaan ohjelman resurssiksi [15]. WPF tukee laitteistokiihdytystä, jolla pystytään osa piirron vaatimasta laskennasta siirtää prosessorilta näytönohjaimelle. [14]

MVVM on WPF:lle suunniteltu käyttöliittymän arkkitehtoninen malli. MVVM-mallin tarkoituksena on erotella logiikka käyttöliittymästä jakamalla komponentit kolmeen osaan: model (malli), view model (näkymämalli) ja view (näkymä). [16]

Kuva 11. MVVM-mallin välisten komponenttien yhteydet.

Malli on liiketoiminnallinen kerros ja näkymämalli toimii mallin ja näkymän välisenä sovittimena. Näkymän rakenne määritellään XAML-kuvauskielen muodossa. Tavanomainen näkymän taustakoodi irrotetaan näkymästä ja kirjoitetaan näkymämalliin. Erottelu näkymästä helpottaa sovelluksen testausta, koska yksikkötesteillä pystytään testaamaan näkymämalli. [16]

Näkymämalli toteuttaa `INotifyPropertyChanged`-rajapinnan (Koodi 3), jonka kautta näkymälle ilmoitetaan ominaisuuksien muutoksista.[16]

Koodi 3. Näkymämallin toteuttama rajapinta.

```
public interface INotifyPropertyChanged {
    event PropertyChangedEventHandler PropertyChanged;
}
```

Useasti näkymämalleille tehdään abstrakti perustaluokka, joka sisältää rajapintatoteutuksen lisäksi apumetodeja tapahtumien laukaisuun. Kolmannen osapuolen MVVM-runkokirjastojen mukana yleensä tulee valmiina Koodi 4:ssä kuvatun tapainen perustaluokka. [16]

Koodi 4. Esimerkki tyyppiturvallisesta näkymämallin perustaluokasta.

```
public abstract class ViewModelBase : INotifyPropertyChanged {
   public event PropertyChangedEventHandler PropertyChanged;

   protected void OnPropertyChanged<T>(Expression<Func<T>>
                                          propertyExpr) {
      string propertyName = GetMemberName<T>(propertyExpr);
      OnPropertyChanged(propertyName);
   }

   protected void OnPropertyChanged(string propertyName) {
      var handler = PropertyChanged;
      if (handler != null) {
         var args = new PropertyChangedEventArgs(propertyName);
         handler(this, args);
      }
   }

   protected bool SetValue<T>(ref T property, T value,
                              Expression<Func<T>> propertyExpr) {
    if (Equals(property, value)) return false;
      property = value;
      OnPropertyChanged(propertyExpr);
      return true;
   }

   private static string GetMemberName<T>(Expression<Func<T>>
                                             property) {
      var lambda = (LambdaExpression)property;
      var unaryExpr = lambda.Body as UnaryExpression;
      var memberExpr = (MemberExpression)(unaryExpr == null
         ? lambda.Body
         : unaryExpr.Operand);
      return memberExpr.Member.Name;
   }
}
```

Perustaluokan avulla selkeytetään lopullisten näkymämallien koodia ja vähennetään ominaisuuksien duplikointia, kuten Koodi 5:ssä näkyy. [16]

Koodi 5. Perustaluokka yksinkertaistaa näkymämallien toteutusta.

```
public class CustomerListViewModel : ViewModelBase {
   private Customer _selectedCustomer;
   private readonly ObservableCollection<Customer> _customers;

   public Customer SelectedCustomer {
      get { return _selectedCustomer; }
      set {
         SetValue(ref _selectedCustomer, value,
                  () => SelectedCustomer);
      }
   }

   public ObservableCollection<Customer> Customers {
      get { return _customers; }
   }

   public CustomerListViewModel(IEnumerable<Customer> customers) {
      _customers = new ObservableCollection<Customer>(customers);
   }

   public ICommand EditCustomerCommand { get; private set; }
   public ICommand RefreshListCommand { get; private set; }
}
```

Ilman näkymämallin perustaluokkaa tarvitsisi duplikoida samoja toiminnallisuuksia jokaiseen toteutettavaan näkymämalliin [16]. Näkymän XAML:ssä viitataan näkymän ominaisuuksiin ja komentoihin, kuten Koodi 6:ssa tehdään.

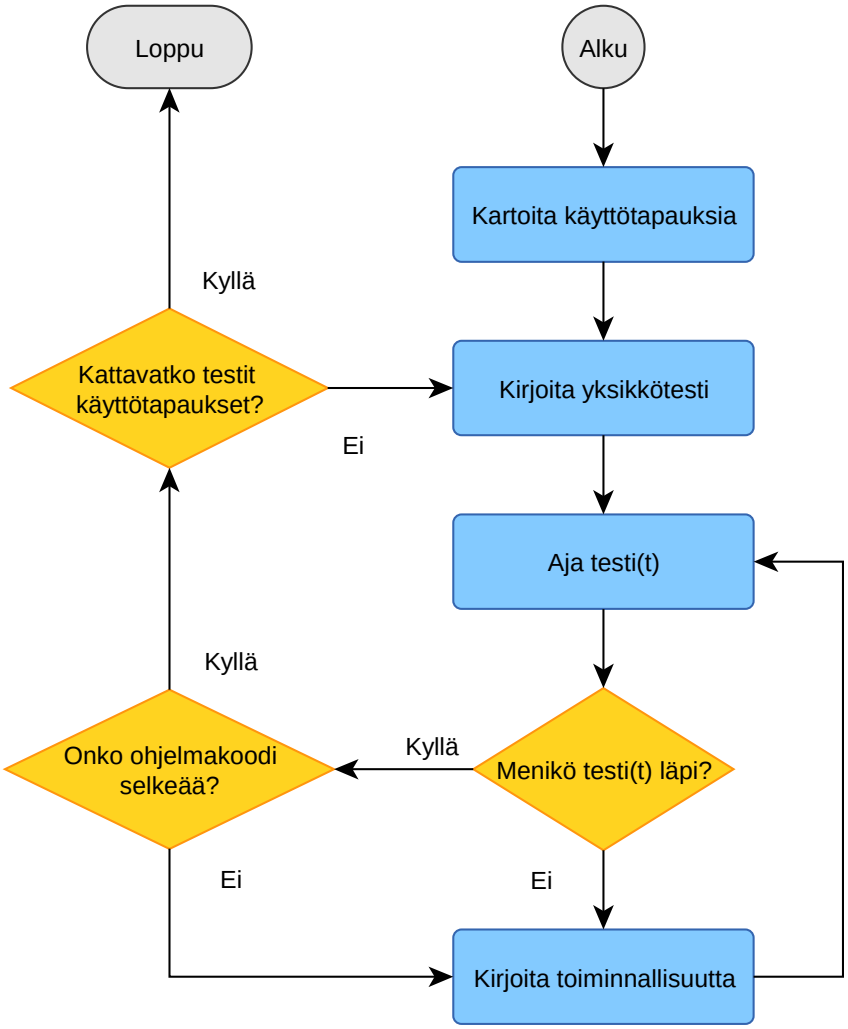
Koodi 6. MVVM-malli toimii WPF:n datasidonnan kanssa.

```
<ListBox ItemsSource="{Binding Customers, Mode=OneWay}"
         SelectedItem="{Binding SelectedCustomer}" />
<Button Content="Edit" Command="{Binding EditCustomerCommand}" />
<Button Content="Refresh" Command="{Binding RefreshListCommand}" />
```

Näkymän datakontekstiksi määritelty näkymämalli jäsennetään heijastuksen avulla ajonaikana. Datasidonnassa näkymämallin ominaisuudet sidotaan näkymän kontrolleihin, jotka päivittyvät ominaisuuksien muutostapahtumien perusteella. [16]

2.3 TDD ja yksikkötestaus

Testivetoinen ohjelmistokehitys on yksikkötestauskeskeinen ohjelmistokehittämismalli, jossa tarkoituksena on kirjoittaa yksikkötestit ennen toiminnallisuutta. Kuva 12 kuvastaa testivetoisen kehityksen työnkulkua. [17]



Kuva 12. Testivetoisen ohjelmistokehityksen työvuo.

Koko TDD perustuu käyttötapauksia kartoittavien yksikkötestien kirjoittamiseen ennen ohjelmatoiminnallisuutta. Tällä varmistetaan ohjelmakoodille hyvä testikattavuus heti kehitysprojektin alusta alkaen ja tekee ohjelman toimintojen kehittämisestä helpommin lähestyttävää, koska kattavien yksikkötestien avulla ohjelmointivirheet havaitaan heti. Tämä ei kuitenkaan tarkoita sitä, etteikö ohjelmointivirheitä olisi, vaan niiden laajuus supistuu. [17]

Yksikkötestien tarkoituksena on vähentää ohjelmointivirheitä, pienentää mahdollisten jäljelle jäävien virheiden laajuutta, kartoittaa ohjelman toiminnallisuutta ja parantaa ohjelmakoodin laatua automatisoimalla alirutiinien validointia. [18]

Tarkoituksena ei ole siis korvata järjestelmä- tai integraatiotestausta, koska yksikkötestien tarkoitus ei ole testata toimivuutta kokonaisuutena. Yksikkötestejä tosin voi soveltaa integraatiotestauksessa, sillä laajempien yksikkötestien avulla voidaan testata komponenttien välistä toiminnallisuutta. [18]

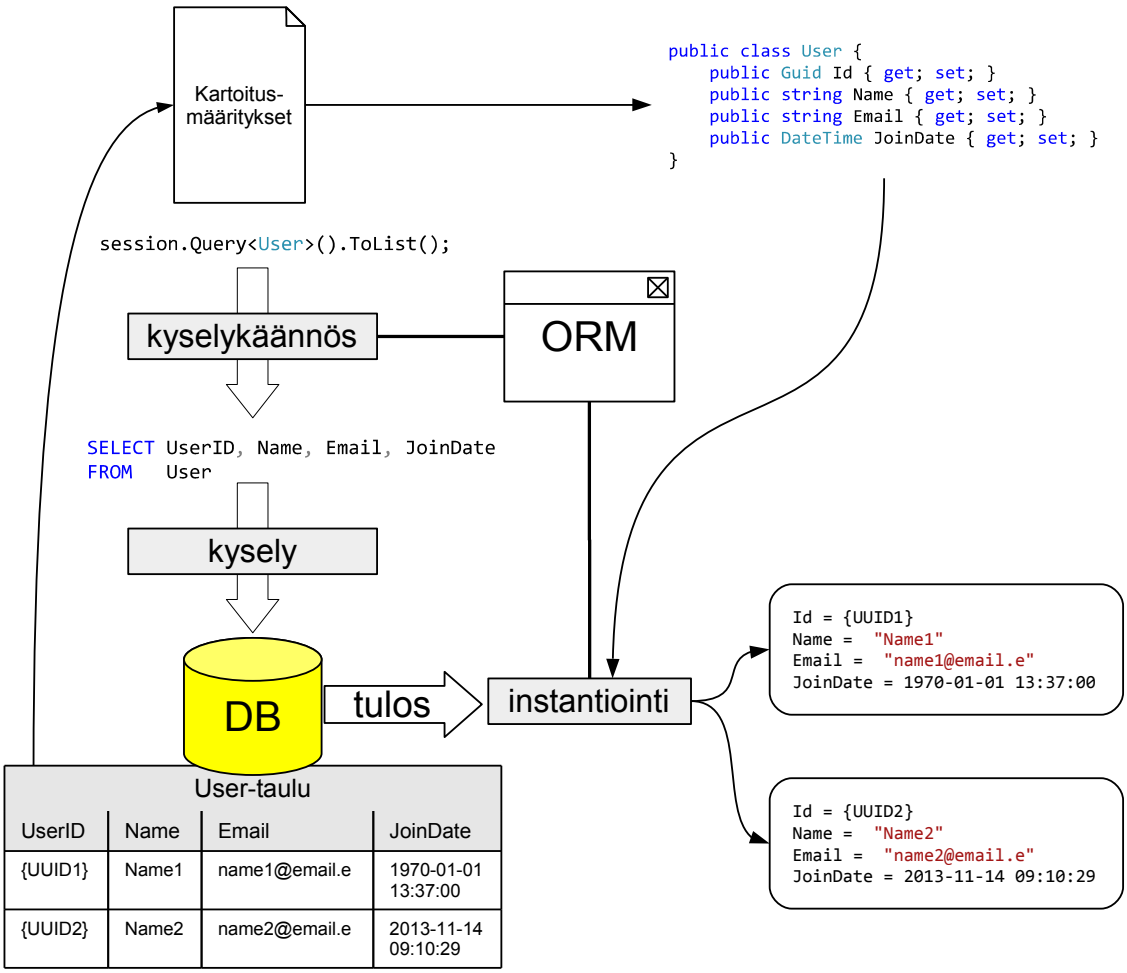
Yksikkötestit helpottavat virheiden paikallistamista, jos aliohjelma toimii virheettömästi, mutta sitä käyttävä ylempi rutiini ei toimi. Tällöin voidaan ensisijaiset virhepaikannukset keskittää ylempään rutiiniin eikä suinkaan aloittaa alemmasta tasosta. [18]

Yksikkötestit epäsuorasti parantavat aliohjelmien laatua, koska niiden tarvitsee olla käyttöympäristöriippumattomampia, jotta niitä on helpompi soveltaa yksikkötestauksen ja loppuympäristön välillä. Aliohjelman refaktorointi on suoraviivaisempaa, koska kattavat yksikkötestit kertovat heti, jos perustoiminnallisuus muuttuu, kun aliohjelman rakennetta muokataan. [18]

Yksikkötestit myös auttavat hahmottamaan aliohjelman kokonaiskuvaa, koska aliohjelma ei ole yksikkötestissä sidonnaisena pääohjelman rutiineihin. Ohjelmakoodin laadun lisäksi dokumentaatio paranee, koska kattavat yksikkötestit kertovat suoraan mihin aliohjelma kykenee, mitä mahdollisia rajatapauksia tarvitsee ottaa huomioon ja mitä virhetilanteita saattaa ilmetä. Käyttöliittymän visuaalisten elementtien testausta ei kuitenkaan pysty helposti automatisoimaan yksikkötestauksella. Ohjauslogiikan pystyy kuitenkin irtauttamaan käyttöliittymästä ja tätä kautta yksikkötestaamaan nuo logiikkaosat, mutta visuaalisen näkymän tarkistus jää edelleen suurimmaksi osin manuaalitestauksen harteille. [18]

2.4 Oliosuhdekartoitus

Oliosuhdekartoituksen (engl. object-relational mapping, ORM) tarkoituksena on avustaa tietokantajärjestelmien relaatiotiedon mallinnusta objektiympäristössä. [19]



Kuva 13. Pelkistetty kuvaus objektirelaatiokartoituksesta.

Luokka kartoitetaan tauluun ja muuttujat taulun sarakkeisiin, jolloin objektit vastaavat taulun rivejä [19]. Kartoitukset määritellään yleensä ulkoisissa asetustiedostoissa, mutta ne ovat useasti liian monisanaisia. Kartoitusmäärittelyt on myös mahdollista määrittää suoraan ohjelmalähdekoodissa, jolloin kääntäjä ilmoittaa useimmista virheellisistä määrittelyistä. Haittapuolena koodimäärittelyis-

sä on se, että ohjelma tarvitsee kääntää uudelleen, jos tietokantarakenteeseen tehdään muutoksia. [20]

Ylimääräisten tietokantalukujen vähentämiseksi on kehitetty laiska lataus (engl. lazy loading), jolloin vasta viiteobjektiin viitattaessa sen tiedot haetaan tietokannasta. Laiska lataus on oletuksena päällä, mutta sen voi kiertää määrittelemällä kyselyyn tai kartoitusmäärittelyyn aikaisen latauksen (engl. eager loading), joka on tehokkaampaa, jos viitetietoja tarvitaan heti latauksen jälkeen. Transaktion sisällä aikaisemmin ladattuja instansseja pystytään hyödyntämään uudelleen sisäisten kätköjen (engl. cache) avulla. [20][21]

SQL-kyselyiden sijasta ORM-kirjasto käyttää omia kyselysyntakseja. Tietokantajärjestelmäkohtainen kyselykääntäjä muuttaa ORM-kyselyn lopuksi sopivaksi SQL-kyselyksi. Suorat SQL-kyselytkin ovat mahdollisia, mutta ne kirjoitetaan ohjelmakoodiin merkkijonoina. [20][21][22]

2.5 Ohjelmistokirjastoja

Sovelluksen kehityksessä on käytetty erinäisiä kolmannen osapuolen ohjelmistokirjastoa.

2.5.1 MVVM-runkokirjasto

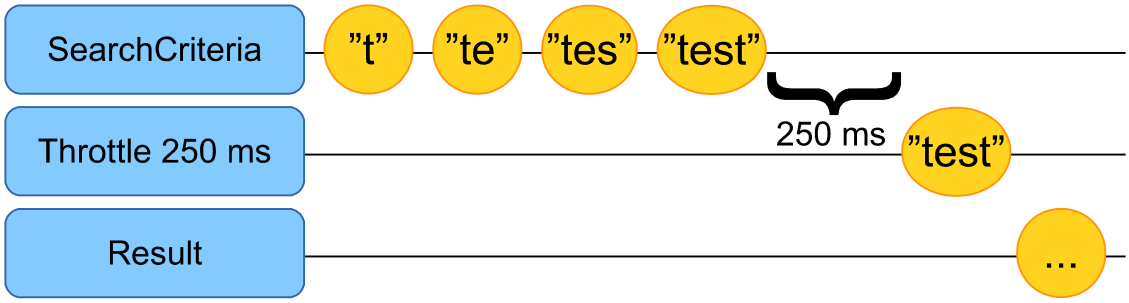
MVVM-runkokirjastona käytetään ReactiveUI-kirjastoa, jossa reagoivaa olio-ohjelmointimallia käyttäen pystyy rakentamaan asynkronisia näkymiä. [23]

Koodi 7 sisältää esimerkin hakutoiminnallisuudesta, joka laukaistaan automaattisesti 250 ms:n kuluttua hakukriteerin viimeisimmän muutoksen jälkeen. Rajoitusmekanismin avulla pystytään vähentämään keskeneräisten hakukriteerien laukaisemia hakuja. Kuva 14 visualisoi tapahtumien virtaa Koodi 7:n tapauksessa.

Koodi 7. Esimerkki hakutulosten käsittelystä reagoivassa olio-ohjelmoinnissa.

```
this.ObservableForProperty(x => x.SearchCriteria)
   .Throttle(TimeSpan.FromMilliseconds(250))
   .Value()
   .Select(searchCriteria => FetchNewSearchResult(searchCriteria))
   // Päivitä tulokset UI säikeessä.
   .ObserveOnDispatcher()
   .Subscribe(result => {
      SearchResult.Clear();
      foreach (var item in result) {
         SearchResult.Add(item);
      }
   });
```

Reagoivassa olio-ohjelmoinnissa rakennetaan toiminnallisuutta yhdistelemällä asynkronisia tapahtumavirtoja. [24]



Kuva 14. Tapahtumien kulku tapahtumavirroissa.

Funktionaalisen ohjelmoinnin tapaan ketjutettavien metodien avulla kootaan yksinkertaisemmista yleisistä palasista monimutkaisempi toiminnallisuus. [24]

2.5.2 ORM-kirjasto

Opinnäytetyössä käytetty ORM-kirjasto on NHibernate, joka on alun perin Java-alustalle kehitetty Hibernate niminen ORM-kirjasto, mikä on sittemmin käännetty myös C#-ohjelmointikielelle. NHibernate tukee useita eri kyselysyntakseja, joista jokaisella on omat hyvät ja huonot puolensa.

Vanhin näistä kyselysyntakseista on HQL (Hibernate Query Language), joka muistuttaa eniten SQL-kyselysyntaksia. HQL-kysely (Koodi 8) määritetään merkkijonona, joten se on altis kirjoitusvirheille, mutta on tämän vuoksi myös joustava. [20]

Koodi 8. Esimerkki HQL-kyselysyntaksista.

```
IEnumerable result = session
    .CreateQuery(@"from User u
                   where u.Name like :namelike
                   order by u.JoinDate desc")
    .SetString("namelike", "Name%")
    .Enumerable();
```

Toinen vaihtoehto on käyttää Criteria-rajapintaa (Koodi 9), joka on HQL:ää vähemmän kirjoitusvirhealtis, mutta edelleen luokkien osiin viitataan merkkijonoina. [20]

Koodi 9. Esimerkki Criteria-kyselysyntaksista.

```
IEnumerable result = session
    .CreateCriteria<User>()
    .Add(Expression.Like("Name", "Name%"))
    .AddOrder(Order.Desc("JoinDate"))
    .List<User>();
```

Kolmantena on QueryOver-kyselysyntaksi, joka on käytännössä vahva tyypitetty versio Criteria-rajapinnasta. QueryOver (Koodi 10) mahdollistaa käännönaikaiset syntaksitarkistukset geneeristen rajapintojen ja lambdojen avulla. [20]

Koodi 10. Esimerkki QueryOver-kyselysyntaksista.

```
IEnumerable result = session
    .QueryOver<User>()
    .WhereRestrictionOn(u => u.Name).IsLike("Name%")
    .OrderBy(u => u.JoinDate).Desc
    .List<User>();
```

Näiden kolmen lisäksi NHibernatessa on sisäänrakennettu tuki LINQille, mikä mahdollistaa yleisimpien LINQ-kyselyiden (Koodi 11) käännön SQL-kyselyiksi. [20]

Koodi 11. Esimerkki LINQ-kyselysyntaksista.

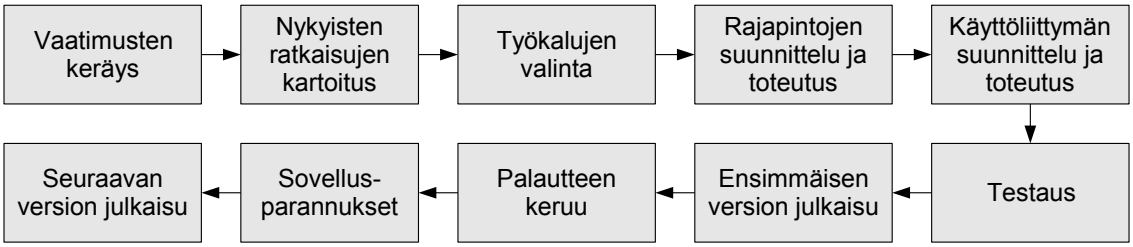
```
IEnumerable result = session
    .Query<User>()
    .Where(u => u.Name.StartsWith("Name"))
    .OrderBy(u => u.JoinDate)
    .ToList();
```

Ulkoisten XML-pohjaisten relaatiokartoitusmäärittelytiedostojen sijasta käytetään Fluent NHibernate -laajennuskirjastoa, joka parantaa NHibernaten tukea

ohjelmakoodimäärittelyille. Kartoitusmallimäärittelyt voi serialisoida binaaritiedostoon, jolloin niitä ei tarvitse seuraavan käynnistyksen yhteydessä luoda uudestaan. [21]

3 SOVELLUKSEN SUUNNITTELU JA TOTEUTUS

Suunnitteluun on monta eriasteista lähestymistapaa, suoraviivaisesta arkkitehtuurin täysimittaisesta laatimisesta aina sovellusrungon hahmotteluun. Tämän sovelluksen kehityksessä on sovellettu enimmäkseen jälkimmäistä tapaa.



Kuva 15. Sovelluksen toteutusvaiheen kulku.

Sovelluksen käyttäjävaatimuksiin kuuluvat:

- asiakkaiden ja näiden kontaktien lisäys, muokkaus ja selaus
- tuki nykyisten tuotteiden lisenssiavaimille
- lisenssiavaimien lisäys, muokkaus, selaus ja haku
- lisenssiavaimissa käytettyjen moduulien helppo jälkikäteen selvitys.

Käyttäjätoiveisiin kuuluu lisenssiavainten asiakaskohtainen massapäivitys.

Käyttäjävaatimusten avulla oli helpompi hahmottaa sovelluksen kokonaiskuvaa ja jakaa sovellus eri toiminta-alueisiin. Lisenssiavainhallintaratkaisuista kerättiin generointialgoritmien lisäksi keskeiset erot ja yhtäläisyydet, joista tietokantarakenne osittain laadittiin.

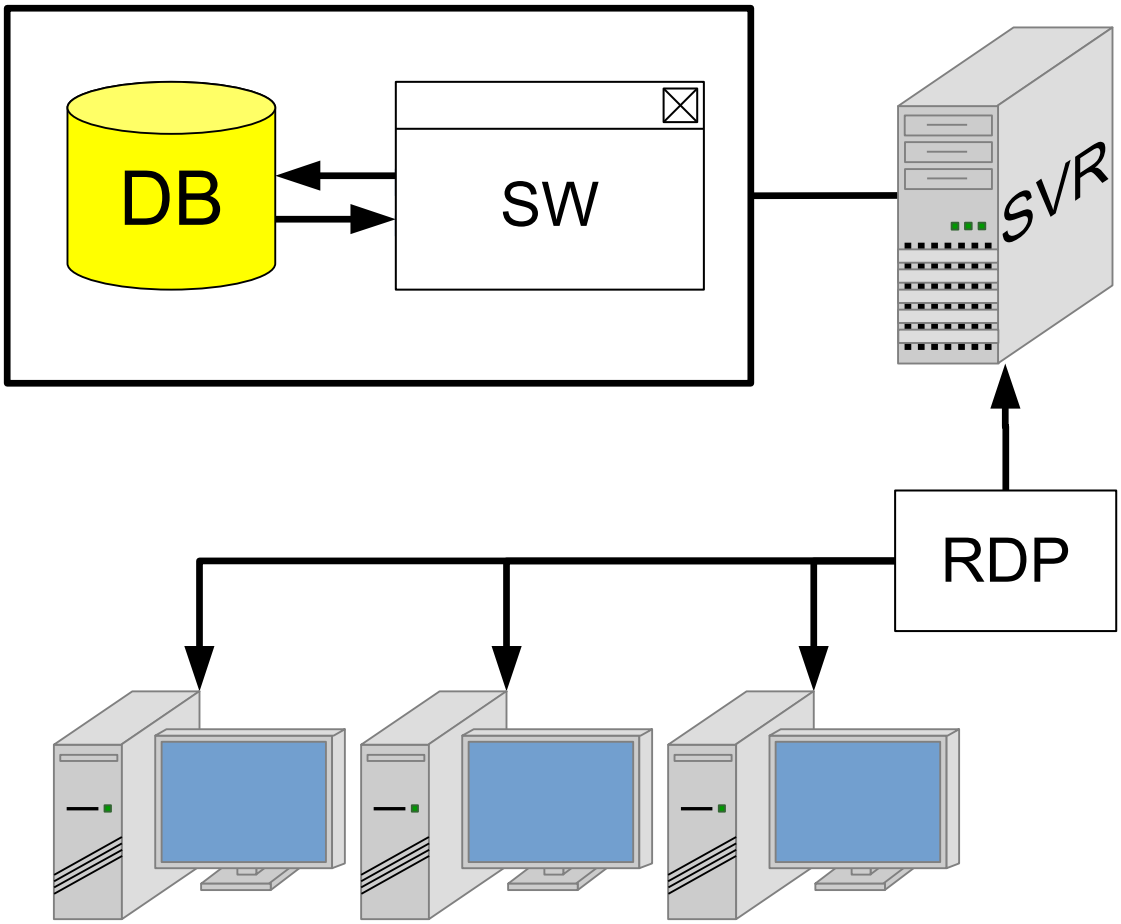
Vanhoista hallintaratkaisuista ei löytynyt sopivaa ehdokasta projektin pohjaksi, joko vanhentuneen alustan takia tai muuten rajoittuneesta rakenteesta johtuen. Sovelluksen alustaksi valittiin C#-ohjelmointikieli, johon osa generointialgoritmeista käännettiin, jollei alkuperäisiä rajapintoja pystynyt kutsumaan suoraan. Yksikkötesteillä varmistettiin lisenssiavaingeneraattoreiden palauttamien lisenssiavainten oikeellisuus.

Käyttöliittymä suunniteltiin tietokantarakenteita ja logiikkarajapintoja silmällä pitäen. Näkymien prototyypitystä joutui tekemään useassa vaiheessa, koska ennestään ei ollut paljon kokemusta käyttöliittymien kehittämisestä. Liiallisen rajapintojen abstrahoinnin ja MVVM-mallin käytön takia suurin osa kehitysajasta meni käyttöliittymän toteutukseen. Abstrahoinnin avulla kuitenkin pystyi vaihtamaan käytetyn ORM-kirjaston ilman suurempaa käyttöliittymätason muokkausta. Osa rajapinnoista kävi läpi refaktorointivaiheen, mikä jarrutti kehitystä, mutta selkeytti ohjelmakoodin rakennetta ja teki lisäominaisuuksien lisäyksestä helpompaa ja vähemmän virhealtista.

Yksikkötestien ja lisättyjen ominaisuuksien testauksen lisäksi lopullinen kattavampi testaus tehtiin ennen ensimmäisen version julkaisua. Sovelluksen ensimmäisen version toiminnan oikeellisuuden varmistamiseksi ajettiin se aluksi rinnakkain vanhojen lisenssiavainhallintasovellusten kanssa. Mahdollisen kriittisen ongelman sattuessa lisenssiavainten luonti ei estyisi täysin ja käyttäjät pystyivät paremmin vertaamaan sovellusten välisiä eroja.

Käyttäjäpalautteiden perusteella seuraavaan versioon tehtiin parannuksia ja korjauksia, joiden lisäksi seuraavaan versioon tuli myös radikaaleja käyttöliittymämuutoksia. Ikkunoiden määrän vähentämiseksi suurin osa näkymistä ilmestyy pääikkunaan välilehtinä erillisten ikkunoiden sijaan. Uudelleenkäytettävien käyttöliittymäkomponenttien ansiosta muutos ikkunoista välilehtiin ei vienyt paria päivää kauempaa. Ikkunat pystyi suoraan korvaamaan välilehdillä, koska näkymät on eroteltu itse ikkunakomponenteista. Lisenssiavainten massapäivitys on jäänyt toistaiseksi puutteelliseksi, mitä on osittain koitettu lieventää yllä mainitulla välilehtien käytöllä.

Hallintasovellus on CRUD-asiakasohjelma, joka käsittelee suoraan tietokantaa. Tämä ei tässä tapauksessa tuo sen enempää riskejä kuin hallitumpi asiakas-palvelin-malli tai muu vastaava monimutkaisempi arkkitehtuuri, koska kyseessä on talon sisäisessä käytössä oleva sovellus. Kuva 16 hahmottaa sovelluksen käyttöympäristön eri osia.



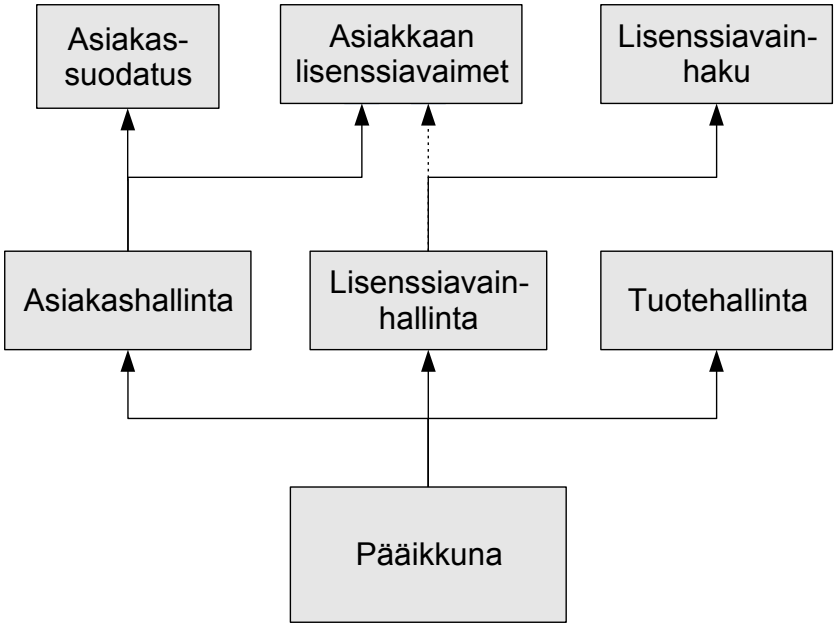
Kuva 16. Sovelluksen käyttöympäristö.

Sovellus sijaitsee palvelinkoneella tietokantoineen, mutta ne voisivat myös sijaita eri paikoissa, koska tietokannan sijainnin voi määrittää asetustiedostosta, jota luetaan sovelluksen käynnistyksessä. Varmuuskopiot tietokannasta otetaan päivittäin ja vain järjestelmänvalvojalla on niihin luku- ja kirjoitusoikeus.

Käyttääkseen sovellusta tarvitsee palvelimeen ottaa yhteys suojatun RDP-etätyöpöytäprotokollan kautta ja käynnistää sovellus palvelimelta. Palvelin on sisäverkon toimialueessa, johon tarvitaan sisäiset tunnukset ja tietokantaan tarvitaan erikseen käyttäjälle oikeudet. Tietoturvariski ei ole paljon sen suurempi kuin muunlaisessa monimutkaisemmassa arkkitehtuurissa. Varmuuskopioilla varmistetaan peukaloitujen tietojen palauttamismahdollisuuden. Tietokantajärjestelmän ja sovelluksen välinen yhteys voidaan salata, jolloin ei ole väliä käyte-

täänkö sovellusta etätyöpöytäprotokollan kautta suoraan palvelimelta vai sijaitseeko sovellus käyttäjän koneella. Sovellusta käytetään toistaiseksi etätyöpöydän kautta päivittämisen helpottamiseksi.

Kuva 17 kuvaa käyttöliittymän toimintojen jakautumista. Käyttöliittymä on jaettu eri toiminta-alueisiin. Asiakas-, tuote- ja lisenssiavainhallinta on eritelty toisistaan.



Kuva 17. Käyttöliittymäjakautuma.

Työtehokkuuden parantamista ajatellen ja toimintojen duplikoinnin vähentämiseksi joidenkin näkymien toiminnat risteävät toisiin toiminta-alueisiin. Esimerkiksi asiakashallinnasta saa aukaistua näkymän valitun asiakkaan lisenssiavainlistaukseen. Lisenssiavainlistaukseen ei tällöin tarvitse erikseen tehdä asiakassuodatusta, koska asiakkaan voi jo valita asiakashallinnan puolelta.

Tuotehallinnan päätarkoituksena on uuden julkaistun tuoteversion määrittely. Tuotehallinnasta on mahdollista myös määrittää kokonaan uusi tuote, mutta sillä ei ole käytännön hyötyä tällä hetkellä, koska kaikki tuotekohtaiset näkymät ja niihin liittyvät käsittelylogiikat määritellään ohjelmakoodissa eikä asetustiedostoissa. Tässä vaiheessa ei tuntunut vaivan arvoiselta tehdä tuotenäkymistä

dynaamisesti määriteltäviä, koska uusia tuotteita tulee harvoin. Uusia tuoteversioita tulee kohtuullisin määrin tuotteesta riippuen, minkä tämän hetkinen toiminnallisuus kattaa hyvin, ellei uusi tuoteversio aiheuta lisenssin käsittelylogiikan muutosta, jolloin ohjelmakoodiin tarvitsee tehdä muutoksia.

Lisenssiavainhallinnan toimintoihin kuuluu lisenssiavainten listaus ja käsittely. Haku monipuolisemmin kriteerein on myös mahdollista ja hakutulokset pystyy viemään Exceliin tai DSV-tekstitiedostoon auditointia varten. Lisenssiä luodessa näkymä muuttuu kuvastamaan valitun tuotteen lisenssiparametreja, mikä kertoo mitkä kentät ovat pakollisia, jolloin lisenssiavaimen tekijän on helpompi syöttää välttämättömät tiedot.

Sovellus ei suoranaisesti tue monen käyttäjän rinnakkaiskäyttöä, mutta päällekkäiset tallennukset on estetty tietokantataulutasolla riviversioinnilla. Tauluilla on erityinen riviversiosarake, johon tallennetaan juokseva kokonaisluku. Ennen tallennusta tarkistetaan, ettei muokkauksen aikana ole alkuperäinen riviversio muuttunut. Tietokantajärjestelmärajapinta heittää poikkeusvirheen, jos riviversio on muuttunut muokkauksen aikana. Virheen voi käsitellä käyttäjäystävällisesti tai sovelluksen voisi myös pahimmassa tapauksessa antaa kaatua, koska se on pienempi pahe kuin ylikirjoittava huomaamaton päällekkäinen tallennus.

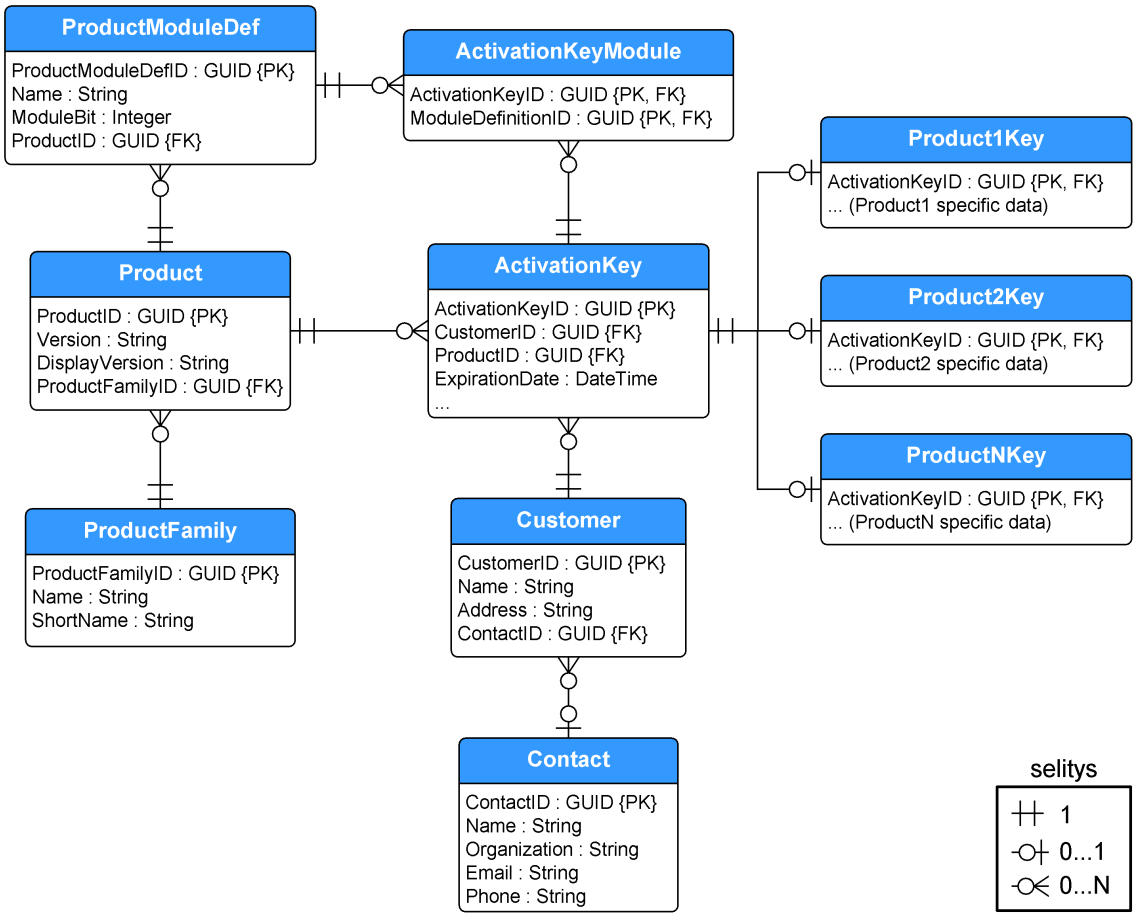
3.1 Tietokanta

Relaatiotietokantajärjestelmä valittiin tietojen säilytykseen, koska tallennettava tieto on helposti jaoteltavissa relaatioihin. Valittu relaatiotietokantajärjestelmä on Microsoft SQL Server, koska se on entuudestaan käytössä talon sisällä, minkä pitäisi vähentää mahdollisessa lisäkehitysvaiheessa vaadittavaa tietokantarakenteen sisäistämisaikaa.

Tietokantarakenne suunniteltiin olemassa olevien tuotteiden lisenssiavaingeneraattoreiden ja lisenssiavainhallintaratkaisujen pohjalta. Suunnittelussa on otettu huomioon vanhojen hallintaratkaisujen heikkouksia. Esimerkiksi aikaisemmin haku tuotemoduulien mukaan ei ollut suoraviivaista, koska niille ei oltu määri-

telty tauluja, eikä relaatiota lisenssiavaimiin. Tuotemoduulit piti useasti jäsentää lisenssiavaimesta.

Kuva 18 sisältää pelkistetyn visualisoinnin tietokantarakenteesta. Osa sarakkeista on jätetty pois tai yhdistetty toisiinsa liittyvien sarakkeiden kanssa tietoturvan ja tilansäästön vuoksi.



Kuva 18. Pelkistetty tietokantarakenne.

Tietokanta sisältää asiakaslisenssitietojen lisäksi perustiedot asiakkaasta, kuten nimi ja sijainti sekä tuotekohtaisia tietoja, kuten versio ja tuotemoduulit.

Asiakkaalle on mahdollista määrittää kontaktihenkilö tai -organisaatio, jolle ollaan yhteydessä asiakkaaseen liittyvissä asioissa. Kontaktina pääsääntöisesti on kenttäinsinööri, joku asiakkaan vastuuhenkilö tai väliorganisaatio.

Jokainen aktivointiavain vaatii asiakkaan ja tuotteen. Valittu tuote määrittelee mihin tuotespesifiseen avaintauluun lisätään tietoja. Lisenssitiedot eroavat tuotekohtaisesti, joten jokaiselle tuotteelle on oma tuotekohtaiset lisenssitiedot sisältävä taulu. Tuotekohtaisessa taulussa viitataan aktivointiavaintauluun, jossa säilytetään kaikille lisensseille yhteiset tiedot. Alun perin tuotekohtaisten avaintaulujen tilalla oli aktivointiavaintaulussa XML-sarake, joka sisälsi tuotekohtaiset avaintiedot, mutta tämä toteutus korvattiin erillisillä tauluilla, jotta tietokantakyselyiden toteuttaminen olisi yksinkertaisempaa.

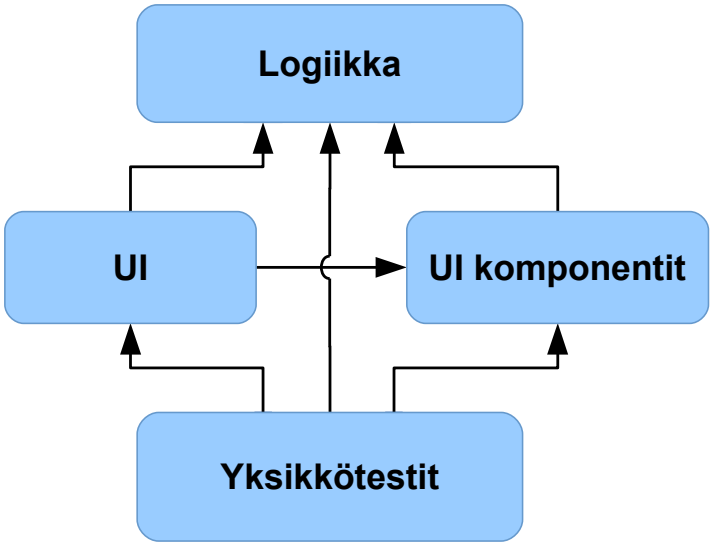
Nykyinen tietokantarakenne ei estä laittamasta yhdelle aktivointiavaimelle moneen tuotespesifiseen avaintauluun yhtäaikaisesti tietoa, mutta ohjelmalogiikka huolehtii siitä ettei näin käy. Tarkistuslogiikka on melko yksinkertainen, joten tämän tietorakennepuutteen korjaus ei ole ollut välttämätöntä.

Pääavaimien lisäksi taulujen viiteavaimet ovat indeksoidut, joita ei oletuksena indeksoida MS SQL Serverissä. Viiteavainten indeksointi parantaa kyselyiden taulujen yhdistämisen suorituskykyä. [25]

3.2 Sovellusarkkitehtuuri

Ohjelmakoodi on pääohjelman lisäksi jaoteltu kolmeen erilliseen ohjelmakirjastoon. Eriteltyjen ohjelmakirjastojen tarkoituksena on edistää ohjelmakoodin modulaarisuutta.

Kuva 19 havainnollistaa ohjelmakirjastojen väliset riippuvaisuudet. Yksikkötestikirjasto käyttää kaikkia muita ohjelmakirjastoja, koska se sisältää kaikkiin ohjelman toimintoihin kohdistuvia yksikkötestejä.



Kuva 19: Ohjelmakirjastoriippuvaisuudet.

Logiikkakirjasto sisältää lisenssiavaingeneraattoreiden lisäksi muun muassa tietokanta-, lokikirjaus- ja asetustiedostorajapinnat. Käyttöliittymäkomponenttikirjastossa on syötevalidointimoduulin lisäksi muun muassa erinäisiä uudelleenkäytettäviä näkymiä. Validointimoduuli ei ole logiikkakirjastossa, koska moduuli soveltaa .NET-alustan käyttöliittymärajapintoja. Ei pitäisi kuitenkaan olla suhteellisen iso työ siirtää tarvittaessa käyttöliittymärajapintariippuvaiset osat erilleen itse validoinnin suorittavasta logiikasta, mutta sille ei ole toistaiseksi ollut tarvetta. Sovelluksen käyttöliittymäkirjasto sisältää kaikki suoraan hallinta-alueisiin liittyvät näkymät ja niiden logiikat ja näiden lisäksi määrittelee myös sovelluksen aloituspisteen.

3.3 Tiedon validointi

Lisenssiavainten oikeellisuus on välttämätöntä asiakastyytyväisyyden ja työtehokkuuden ylläpitämisen kannalta. Toimimaton lisenssiavain saattaa koetella asiakkaan kärsivällisyyttä ja vie aikaa tukihenkilöstöltä. Lisenssiavainten luonti ei kuitenkaan ole aikakriittinen operaatio, koska lisenssiavaimia käyttävät tuotteet ilmoittavat yleensä muutaman viikon tai kuukauden ennakkoon vanhenevasta lisenssistä.

Projektitasolla yksikkötestauksien avulla varmistetaan, että lisenssiavaingenerointi tuottaa oikeanlaisen tuloksen annetuille syötteille. Yksikkötestit eivät kuitenkaan varmista lopullisen käyttöliittymän näyttämän tuloksen oikeellisuutta, joten se testataan erikseen.

Lisenssiavaimia käsiteltäessä sovellus tarkkailee muutettuja parametreja ja huomauttaa tarvittaessa virheistä tai puutteellisista syötteistä. Tallennus estetään kunnes virheet on korjattu, jottei virheellisiä lisenssiavaimia tallenneta järjestelmään. Samat menettelyt pätevät lisenssiavainten lisäksi tuotteita ja asiakkaita käsiteltäessä. Kentän sisällön muuttuessa laukaistaan ohjelmatapahtuma, joka ilmoittaa validointimoduulille muuttuneesta kentän sisällöstä. Validointimoduuli tarkistaa, onko kenttään liitetty mitään validointisääntöä ja suorittaa kentän validoinnin. Mahdolliset poikkeamat kentän sisällössä tulostetaan ruudulle virheiden tai varoitusten muodossa. Virheistä poiketen varoitukset eivät estä tallennusta, mutta tallennettaessa kuitenkin muistutetaan käymään läpi varoitukset.

4 YHTEENVETO

Kehitysprojektin alussa ei käytetty versionhallintajärjestelmää, mutta näin jälkikäteen ajateltuna se on välttämätön työkalu projektin alusta alkaen. Muutoksista jää pysyvä jälki ja versionhallintajärjestelmän vertailutyökalut auttavat koodimuutosten tarkistuksessa ennen niiden tallennusta. Myöhemmin versionhallintaan käytetyn TFS-projektihallintajärjestelmän sisäänrakennettu tehtävienhallinta auttoi aikataulutuksen arvioinnissa ja tarvittavien sovelluksen ominaisuuksien kirjanpidossa.

Tietokantatauluissa tuoteperhe ja tuote nimet ovat osittain virheelliset, koska suurin osa tuotteista kuuluu samaan tuoteperheeseen ja tuotetaulun määrittelyt ovat pikemminkin tuoteversioita eivätkä tuotteita. Tuote ja tuoteversio olisivat olleet lähempänä totuutta taulujen niminä verrattuna nykyisiin nimivaihtoehtoihin, mutta sillä ei ole käytännön merkitystä ohjelmiston toimivuudelle.

TDD:tä ei pääsääntöisesti ole käytetty opinnäytetyön kehittämisessä, mutta joidenkin monimutkaisempien rajapintojen suunnittelua se on helpottanut huomattavasti. Yksikkötestien avulla on varmistettu eteenkin kriittisten aliohjelmien oikeanlainen toiminta. TDD:n laajempi hyödyntäminen olisi varmasti yksinkertaistanut joidenkin isompien komponenttien toteuttamista.

Alun perin ORM-kirjastona oli .NET-alustan mukana tuleva LINQ2SQL, mutta se tukee ainoastaan yhden-suhde-yhteen-relaatioita, jolloin yhden-suhde-moneen- ja monen-suhde-moneen-relaatioiden käsittely turhaan monimutkaisti ohjelmakoodia. Sittemmin .NET v3.5 SP1 julkaisussa esiteltiin LINQ2SQL:n seuraaja, Entity Framework, joka korjasi tilanteen, mutta sitä ei kuitenkaan valittu korvaajaksi, koska valinnan aikaan NHibernate tarjosi joustavamman kehitysalustan.

Kaikkia kirjastorajapintoja ei kannata abstrahoida palvelurajapintojen taakse. Alun perin ORM-kirjasto oli abstrahoitu niin paljon, ettei kaikkia sen transaktioita optimoivia ominaisuuksia pystynyt hyödyntämään. Tämä laski tietokantaoperaa-

tioiden suorituskykyä sen verran, että loppujen lopuksi kannatti ottaa askel taaksepäin ja kutsua suoraan ORM-kirjaston rajapintoja isoimmissa kyselyissä. ORM itsessään on jo voimakkaasti abstrahoitu rajapinta relaatiotietokantajärjestelmän ja sovelluksen välillä, ettei sitä kannata paljoakaan koittaa peittää omien rajapintojen taakse.

Keskeneräisen lisenssiavainten massapäivityksen lisäksi auditointihistorian lisäys olisi mahdollinen seuraava lisättävä ominaisuus. Tällä hetkellä auditointia korvaa vanhojen hallintaratkaisujen tapainen muistiinpanokenttä lisenssiavaimille, johon kirjataan lisenssin uusinta-ajankohdat ja perusteet.

Tulevaisuudessa yksi tapa vähentää lisenssiavainhallinnan viemää aikaa muilta asiakastukitehtäviltä, olisi kehittää www-sovellus jota kautta asiakkaat voisivat luoda tunnuksia vastaan lisenssiavaimia ottamatta yhteyttä tukeen. Asiakkaan sisäänkirjautumistunnuksille asetettaisiin tukisopimuksessa määritetyt sallitut lisenssiavainmäärät, lisenssin maksimivoimassaolopituus ja muunlaiset lisenssiavaimiin liittyvät parametrit. Tämä toiminnallinen muutos ei tosin olisi mikään pieni työ, eikä yksinkertainen tietoturvan ja ympäristön hallinnan kannalta, sillä käyttöliittymä pitäisi kirjoittaa kokonaan uusiksi ja käsittelylogiikat pitäisi eristää standardoitujen salausten, transaktiokäsittelyiden ja autentikointien taakse käyttäjien ulottuvilta. Tulevaisuutta ajatellen uusien asiakkaiden myötä lisääntyvän lisenssiavainhallinnan aiheuttaman kuorman vähentäminen saattaa olla hyvinkin välttämätöntä. Vapautuneet resurssit voitaisiin hyödyntää muihin aikakriittisempiin tehtäviin. Ohjeistuksen tarve lisenssiavainhallinnan osalta saattaisi lisääntyä, koska aikaisemman tukihenkilöstön perehdyttämisen lisäksi tarvitsisi myös asiakkaita ohjeistaa, mikä tuo lisähaasteita asiakkaiden vaihtelevista tietotekniikkataidoista johtuen.

LÄHTEET

[1] Fonecta, "Wallac Oy - taloustiedot," [www-dokumentti]. Saatavilla: http://www.finder.fi/Terveydenhuollon%20laitteita%20ja%20tarvikkeita/Wallac%20Oy/TURKU/taloustiedot/185226 (Luettu: 16.1.2014)

[2] PerkinElmer, "About Us," [www-dokumentti]. Saatavilla: http://www.perkinelmer.com/fi/ourcompany/aboutus/default.xhtml (Luettu: 16.1.2014)

[3] *ECMA-334, C# Language Specification 4th edition*. ECMA, 2006.

[4] *ECMA-335, CLI Partitions I to VI*. ECMA, 2012.

[5] Goetz, B., "Java theory and practice: Dynamic compilation and performance measurement," [www-dokumentti]. Saatavilla: http://www.ibm.com/developerworks/java/library/j-jtp12214/index.html (Luettu: 2.10.2013)

[6] Todorov, T., "Understanding .NET Just-In-Time Compilation," [www-dokumentti]. Saatavilla: http://blogs.telerik.com/justteam/posts/13-05-28/understanding-net-just-in-time-compilation (Luettu: 2.10.2013)

[7] Cochran, M., "C# Heap(ing) Vs Stack(ing) in .NET: Part I," [www-dokumentti]. Saatavilla: http://www.c-sharpcorner.com/UploadFile/rmcochran/csharp_memory01122006130034PM/csharp_memory.aspx?ArticleID=9adb0e3c-b3f6-40b5-98b5-413b6d348b91 (Luettu: 14.11.2013)

[8] Microsoft, "Garbage Collection," [www-dokumentti]. Saatavilla: http://msdn.microsoft.com/en-us/library/0xy59wtx%28v=vs.100%29.aspx (Luettu: 14.11.2013)

[9] Microsoft, "Thread Stack Size," [www-dokumentti]. Saatavilla: http://msdn.microsoft.com/en-us/library/windows/desktop/ms686774%28v=vs.85%29.aspx (Luettu: 12.11.2013)

[10] Minerich, R., "Increasing the Size of your Stack," [www-dokumentti]. Saatavilla: https://www.atalasoft.com/cs/blogs/rickm/archive/2008/04/22/increasing-the-size-of-your-stack-net-memory-management-part-3.aspx (Luettu: 15.11.2013)

[11] Mariani, R., "Garbage Collector Basics and Performance Hints," [www-dokumentti]. Saatavilla: http://msdn.microsoft.com/en-us/library/ms973837.aspx (Luettu: 28.10.2013)

[12] Mariani, R., "When to call GC.Collect()," [www-dokumentti]. Saatavilla: http://blogs.msdn.com/b/ricom/archive/2004/11/29/271829.aspx (Luettu: 28.10.2013)

[13] Friedman, A., ".Net Memory Leaks. It IS possible!," [www-dokumentti]. Saatavilla: http://crazorsharp.blogspot.fi/2009/03/net-memory-leaks-it-is-possible.html (Luettu: 15.11.2013)

[14] Microsoft, "Introduction to WPF," [www-dokumentti]. Saatavilla: http://msdn.microsoft.com/en-us/library/aa970268%28v=vs.100%29.aspx (Luettu: 18.11.2013)

[15] Shamam, T., "Compiled XAML = BAML not IL," [www-dokumentti]. Saatavilla: http://blogs.microsoft.co.il/tomershamam/2007/05/25/compiled-xaml-baml-not-il (Luettu: 18.11.2013)

[16] Smith, J., "WPF Apps With The Model-View-ViewModel Design Pattern," [www-dokumentti]. Saatavilla: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx (Luettu: 18.11.2013)

[17] Shore, J., "The Art of Agile Development: Test-Driven Development," [www-dokumentti]. Saatavilla: http://www.jamesshore.com/Agile-Book/test_driven_development.html (Luettu: 14.1.2014)

[18] Seshadri, S., "The Advantages of Unit Testing Early ," [www-dokumentti]. Saatavilla: http://googletesting.blogspot.fi/2009/07/by-shyam-seshadri-nowadays-when-i-talk.html (Luettu: 13.1.2014)

[19] Ambler, S., "Mapping Objects to Relational Databases: O/R Mapping In Detail," [www-dokumentti]. Saatavilla: http://www.agiledata.org/essays/mappingObjects.html (Luettu: 23.1.2014)

[20] NHibernate, "NHibernate Reference Documentation," [www-dokumentti]. Saatavilla: http://nhforge.org/doc/nh/en/ (Luettu: 17.1.2014)

[21] Dentler, J., *NHibernate 3.0 Cookbook*, Packt Publishing, 2010.

[22] Smith, D., "Solving Performance Problems with nHibernate (or any ORM)," [www-dokumentti]. Saatavilla: http://geekswithblogs.net/Optikal/archive/2013/03/10/152371.aspx (Luettu: 23.1.2014)

[23] Betts, P., "Zen of ReactiveUI," [PDF-tiedosto]. Saatavilla: http://reactiveui.net/welcome/pdf (Luettu: 18.11.2013)

[24] Campbell, L., "Introduction to Rx," [www-dokumentti]. Saatavilla: http://www.introtorx.com/ (Luettu: 18.11.2013)

[25] Stellato, E., "The Benefits of Indexing Foreign Keys," [www-dokumentti]. Saatavilla: http://www.sqlperformance.com/2012/11/t-sql-queries/benefits-indexing-foreign-keys (Luettu: 24.10.2013)