Jesse Honkanen

Mobiilisovellusten kehitys .NET-teknologioilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

27.11.2013

Metropolia

Tekijä Otsikko Sivumäärä Aika	Jesse Honkanen Mobiilisovellusten kehitys .NET-teknologioilla 52 sivua + 2 liitettä 27.11.2012
Tutkinto	insinööri (AMK)
Koulutusohjelma	mediatekniikka
Suuntautumisvaihtoehto	digitaalinen media
Ohjaajat	vanhempi konsultti Tarja Hakala lehtori Olli Alm

Insinöörityössä tutkittiin sovelluskehitystä .NET-teknologioilla mobiilikäyttöjärjestelmille. Käyttöjärjestelmien eroavaisuuksien ja käytössä olevien eri sovelluskehysten myötä kehittäjät joutuvat kuitenkin toteuttamaan sovelluksen erikseen jokaiselle alustalle. Kehittäjien työn helpottamiseksi eri .NET-teknologioissa on samankaltaiset sovelluskehykset, ohjelmointimallit ja työkalut sovellusten kehittämiseen.

Työssä tutkittiin myös hyvän sovellusarkkitehtuurin saavuttamista käyttäen Separation of concerns -periaatetta ja lisäksi tutkittiin kuinka sovellus kannattaa tehdä modulaariseksi. Näitä periaatteita ja tekniikoita käyttämällä esitetään, kuinka Portable Class Librarya ja Model–View–ViewModel-arkkitehtuurimallia voidaan hyödyntää alustariippumattoman sovelluksen toteutuksessa.

Insinöörityössä toteutettiin Portable Class Librarya ja Model–View–ViewModel-arkkitehtuurimallia käyttäen kahdelle mobiilikäyttöjärjestelmälle käyttöliittymäprototyyppi, joka pystyy käyttämään kahden eri asiakkuudenhallintajärjestelmän web-palveluita. Insinöörityössä toteutettua prototyyppiä ja sen lähdekoodia on jo hyödynnetty sovelluksessa, jonka tarkoituksena on osoittaa räätälöityjen mobiilisovelluksen hyötyä yritysten liiketoiminnassa ja sisäisissä prosesseissa.

Portable Class Libraryn avulla pystytään säästämään sekä aikaa että vaivaa, kun kehitetään .NET-sovelluksia usealle eri alustalle. Portable Class Library kuitenkin edellyttää tiettyjen .NET-sovelluskehikkojen ja ohjelmointimallien käyttämistä. Nämä rajoitukset voidaan myös käsittää hyödyllisenä piirteenä, koska ne johtavat modulaarisen sovellusarkkitehtuurin toteuttamiseen.

Avainsanat	Model–View–ViewModel, XAML, Portable Class Library, asiakkuudenhallintajärjestelmä, .NET

Author	Jesse Honkanen
Title	Mobile application development using .NET technologies
Number of Pages	52 pages + 2 appendices
Date	27th of November 2013
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Tarja Hakala, Senior Consultant Olli Alm, Lecturer

The goal of the thesis was to study application development for mobile platforms using .NET technologies. Due to differences in the platforms and software development kits, developers have to implement applications separately for each platform. To facilitate developers work, separate .NET technologies offer similar software development kits, programming models and tools.

The thesis also studies how to apply Separation of concerns principle and a modular application structure, to achieve good application architecture. These theories will build the basis on how to use the Portable Class Library and the Model–View–ViewModel pattern in cross platform development.

Above mentioned Portable Class Library and Model–View–ViewModel pattern were used to create an application prototype that can use two different customer relationship management systems as its back-end and works on two different mobile platforms. The application prototype and its source code is already utilized in a new prototype. The new prototype aims to demonstrate the benefit of tailor-made mobile applications in the business and internal processes of corporate clients.

Portable Class Library can save both time and effort when developing cross platform .NET applications. However, Portable Class Library requires certain programming principles, patterns and .NET technologies to work. These limitations can also be thought as a benefit, because they lead to the use of a modular application architecture.

Keywords	Model–View–ViewModel, XAML, Portable Class Library, CRM, .NET

Sisällys

Lyhenteet ja määritelmät

CLR	Common Language Runtime on .NET-alustoissa toimiva virtuaalikone, joka suorittaa ja ylläpitää .NET-sovelluksia.
JIT	Just-in-time-kääntäminen tarkoittaa koodin kääntämistä juuri ennen suoritusta natiiviksi koodiksi. .NET-sovelluksen JIT-kääntäminen tehdään CLR-virtuaaliympäristössä.
JSON	JavaScript Object Notation on datan kuvauskieli, jota käytetään tiedonvälitykseen palvelimen ja asiakkaan välillä.
LINQ	Language integrated query on datakyselyihin liittyvä komponentti .NET-sovelluskehyksessä. Se mahdollistaa kyselyn kirjoittamisen esimerkiksi C#-kielellä, joka tulkitaan suorituksen aikana kyselykieleksi relaatiotietokantaan tai muihin datan lähteisiin.
Mahdollisuus	Mahdollisuus (business opportunity) on tunnistettu tilaisuus saada myytyä yrityksen tuote tai palvelu asiakkaalle.
Managed code	Managed code eli hallittu koodi on Microsoftin termi koodille, jota suoritetaan ja käännetään natiiviksi koodiksi CLR-virtuaaliympäristössä.
MVVM	Model–View–ViewModel-arkkitehtuurimalli on tarkoitettu käyttöliittymäsovellusten kehittämiseen. Tavoitteena on erottaa käyttöliittymä sovelluksen logiikasta ja tietomallista.
Natiivi koodi	Natiivi koodi käännetään suoraan jonkin tietyn prosessoriarkkitehtuurin konekieleksi.
.NET	.NET on yhteinen nimi Microsoftin kehitysalustoille ja ohjelmistokirjastoille, joiden avulla kehitetään muun muassa työpöytä-, mobiili- ja web-sovelluksia.
PCL	Portable Class Library on työkalu, jonka avulla voi kirjoittaa luokkakirjastoja, jotka toimivat useammalla .NET-sovelluskehitysalustalla.

	Muissa .NET-luokkakirjastotyypeissä voi asettaa vain yhden tuetun ympäristön.
SDK	Software development kit eli ohjelmistokehityspaketti on kokoelma työkaluja ja luokkakirjastoja, joilla kehitetään sovelluksia tietylle alustalle, kuten Windows Phone 8 -käyttöjärjestelmälle.
SoC	Separation of concerns on suunnitteluperiaate ohjelmistosuunnittelussa. Tavoitteena on välttää päällekkäisiä toimintoja jakamalla sovelluksen eri vastuualueet moduuleihin.
SOQL	Salesforce Object Query Language on kyselykieli, jota käytetään Salesforce-asiakkuudenhallintajärjestelmässä.
WCF	Windows Communication Foundation on web-palveluiden toteuttamiseen tarkoitettu .NET-sovelluskehyksen versio.
WPF	Windows Presentation Foundation on kokoelma luokkakirjastoja ja ohjelmointimalli, jota käytetään .NET-pohjaisten työpöytäsovellusten kehittämiseen Windows-käyttöjärjestelmille.
XAML	Extensible Application Markup Language on XML-pohjainen kieli, jota käytetään useilla eri .NET-alustoilla käyttöliittymien kuvauskielenä.
XML	Extensible Markup Language on merkintäkieli dokumenteille ja datalle. Sen tarkoituksena on olla yhtä aikaa ihmisten ja koneiden luettavissa.

1 Johdanto

Windows 8-, Windows RT- ja Windows Phone 8 -käyttöjärjestelmien myötä Microsoftin tavoitteena on tuoda yhtenäinen käyttökokemus tabletille, matkapuhelimeen ja työpöydälle. Myös kehittäjät voivat hyötyä tästä toteuttamalla yhtenäisen sovelluksen näille käyttöjärjestelmille ja saada mahdollisesti laajemman käyttäjäkunnan sovellukselle. Käyttöjärjestelmien eroavaisuuksien ja käytössä olevien eri sovelluskehysten myötä kehittäjät joutuvat kuitenkin toteuttamaan sovelluksen erikseen jokaiselle alustalle. Helpottaakseen kehittäjien työtä Microsoft tarjoaa yhtenäiset sovelluskehykset, ohjelmointimallit ja työkalut sovellusten kehittämiseen.

Insinöörityössä tutustutaan sovelluskehitykseen Windows-ympäristöissä ja Windows-sovelluskehityksen kannalta olennaiseen Model–View–ViewModel-arkkitehtuurimalliin ja Portable Class Libraryyn. Model–View–ViewModel-arkkitehtuurimalli on tarkoitettu käyttöliittymäsovellusten kehittämiseen, mukaan lukien Windows RT- ja Windows Phone 8 -sovellusten kehittämiseen. Sen tavoitteena on erottaa käyttöliittymä sovelluksen logiikasta ja tietomallista. Portable Class Library on apuväline, joka tukee .NET-sovelluskehitystä usealle alustalle yhtä aikaa. Lisäksi insinöörityössä käsitellään Windows RT- ja Windows Phone 8 -sovellusten kannalta hyviä sovelluskehityksen periaatteita.

Yhdessä Portable Class Library ja Model–View–ViewModel-arkkitehtuurimalli antavat perustan alustariippumattomaan kehitykseen Windows-ympäristöissä sekä työpöydälle että selaimeen. Tavoitteena on, että sovelluksille toteutetaan yhteiskäyttöistä logiikkaa ja jaettuja toiminnollisuuksia sekä jokaista kohdealustaa varten oma käyttöliittymäsovellus. Portable Class Libraryn rajoitettu ohjelmointikielten ja luokkakirjastojen tuki saattaa vaikeuttaa kehittäjien työtä. Samalla se kuitenkin pakottaa noudattamaan tiettyjä periaatteita, jotka ohjaavat sovellusarkkitehtuurin yleiskäyttöiseksi ja alustariippumattomaksi.

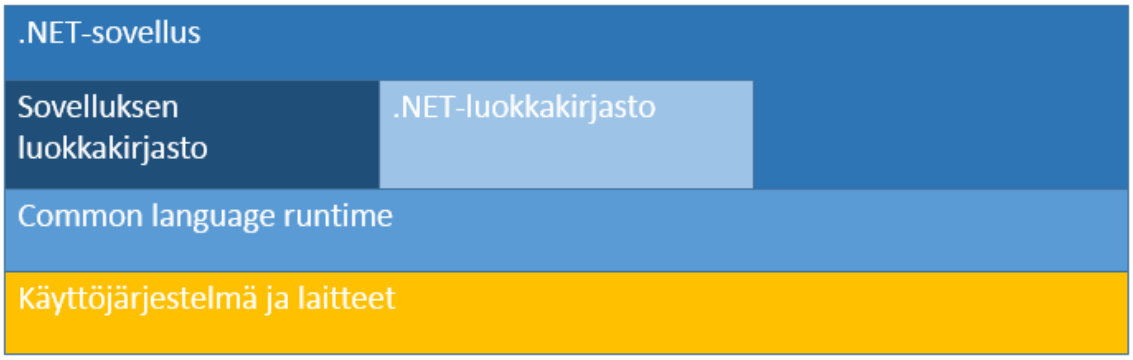
Insinöörityön tavoitteena on suunnitella ja toteuttaa alustariippumaton tablettisovellus Windows RT -käyttöjärjestelmälle käyttämällä Portable Class Librarya ja Model–View–ViewModel-arkkitehtuurimallia. Sovellusarkkitehtuurissa pyritään ottamaan huomioon sekä riippuvuuksien hallinta että alustakohtaiset erovaisuudet, joita tarkastellaan luvuissa 3.1 ja 3.3. Näin pyritään mahdollisimman alustariippumattomaan toteutukseen.

Samalla työssä tutkitaan Portable Class Libraryn kehittäjäystävällisyyttä ja käytettävyyttä alustariippumattoman sovelluksen pohjana toteuttamalla sovellusprototyyppi myös Windows Phone 8 -käyttöjärjestelmälle.

2 Sovelluskehitys .NET-teknologioilla

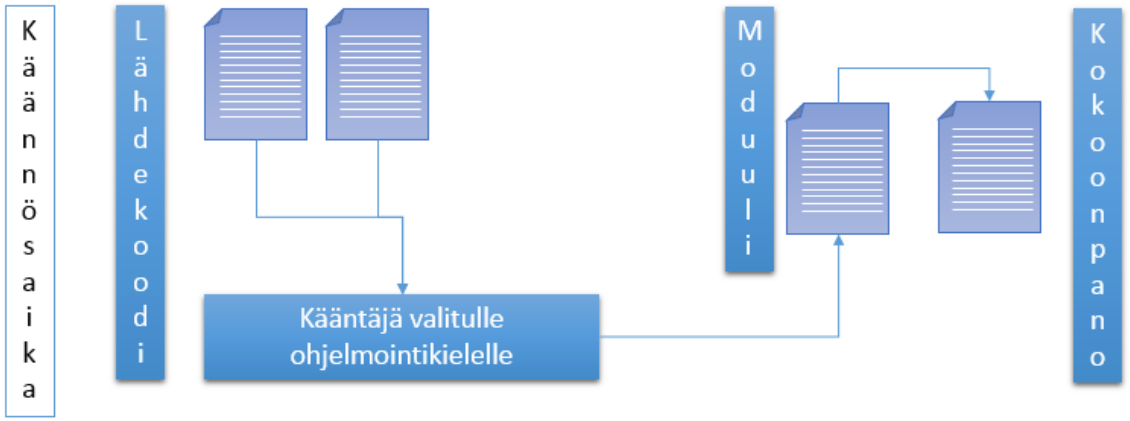
2.1 .NET-sovelluskehys

.NET-sovelluskehys koostuu kahdesta osasta, Common Language Runtimestä (CLR) eli virtuaalikoneesta, joka suorittaa ja ylläpitää .NET-sovelluksia, ja .NET-luokkakirjastosta. Kuviossa 1 on kuvattu kerrosmalli, joka muodostuu sovelluksesta ja sen lähdekoodista sekä niiden suhteesta .NET-luokkakirjastoon ja CLR-ympäristöön. .NET-sovellus käyttää sekä omaa luokkakirjastoa että .NET-luokkakirjastoa, ja sitä suoritetaan CLR-virtuaalikoneessa. [38.]



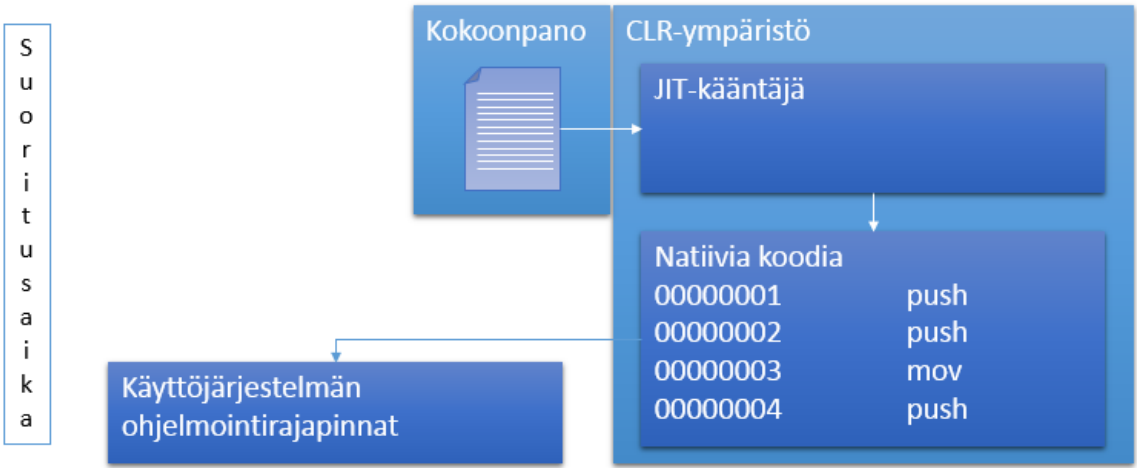
Kuvio 1. Sovelluksen suhde .NET-sovelluskehykseen [mukaillen lähteestä 41].

.NET-sovellukset kirjoitetaan niin sanotulla hallitulla koodilla (managed code) ja CLR-virtuaalikone mahdollistaa näiden sovellusten suorittamisen eri prosessorikantojen päällä. Se kääntää ja suorittaa hallitun koodin, ja lisäksi se ylläpitää muun muassa muistin käyttöä, säikeistystä ja turvallisuutta. [41.] Koodin kääntämiseen käytetään Just-in-time-kääntäjää, joka kääntää koodin natiiviksi konekieleksi vaadittaessa sovelluksen suorituksen aikana. Kuviossa 2 kuvataan sovelluksen lähdekoodin käännösaikainen prosessi. Lähdekoodi on kirjoitettu managed code -kielellä, ja se käännetään moduuleiksi. Moduuleista kootaan välikielinen kokoonpano (assembly), joka on kirjoitettu Microsoftin omalla välikielellä. [8; 45, s. 3.]



Kuvio 2. Sovelluksen lähdekoodin kääntäminen kokoonpanoksi [mukaillen lähteestä 45, s. 3].

Kuviossa 3 kuvataan sovelluksen suoritusaikainen prosessi. CLR-virtuaalikone lataa välikielelle käännetyn kokoonpanon ja suorittaa sovelluksen. Just-in-time-kääntäjä suorittaa käännöksen välikielestä konekielelle sovelluksen suorituksen aikana.



Kuvio 3. Sovelluksen lähdekoodin kääntäminen natiiviksi koodiksi [mukaillen lähteestä 45, s. 3].

.NET-sovelluskehyksen keskeisiä piirteitä ovat esimerkiksi asynkronisen ohjelmoinnin async/await-malli ja Language Integrated Query -komponentti (LINQ). Async/await-malli yksinkertaistaa ja helpottaa kehittäjän työtä siirtämällä asynkronisen operaation säikeiden hallinnan järjestelmälle ja palauttaa suorituksen automaattisesti await-avainsanan kohdalle operaation valmistuttua. LINQ yksinkertaistaa kyselyiden kirjoittamiseen tarvittavan koodin tietokantoja ja paikallisia kokoelmia tai objekteja vasten. [43; 38.] .NET-sovelluskehyksen päälle on myös toteutettu valmiita sovellusarkkitehtuureja, kuten Windows Runtime ja Windows Phone, joita käsitellään luvuissa 2.2 ja 2.3.

.NET-sovelluskehys tukee useita eri ohjelmointikieliä, kuten oliopohjaisia kieliä C#, Visual Basic ja C++. Lisäksi .NET-sovelluskehys tukee funktionaalisia kieliä, kuten F#:ia, ja skriptikieliä, kuten JavaScriptiä. Myös Python ja Ruby ovat tuettuja omien .NET-versioiden kautta (IronPython ja IronRuby). Edellä mainituilla kielillä voidaan toteuttaa niin sanotun yleiskielen spesifikaation (Common Language Specification) mukaisia luokkia, jolloin niitä pystyy käyttämään millä tahansa muulla .NET-sovelluskehyksen tukemalla kielellä. Spesifikaatio on kokoelma sääntöjä, jotka määrittävät esimerkiksi luokan ja sen jäsenten nimeämiskäytäntöjä sekä sen, millä merkinnöillä spesifikaation noudattamista ilmaistaan. [26.]

Visual Studio on yleisin ohjelmointiympäristö .NET-sovelluksille. Se sisältää työkalut muun muassa koodin kirjoittamiseen, kääntämiseen, tietokantojen suunnitteluun. Visual Studio tarjoaa myös valmiit testiympäristöt ja emulaattorit useille eri alustoille, kuten Windows Phonelle ja web-palvelimille. Versiosta riippuen se sisältää myös kattavat testaustyökalut. [48.] Vaikka Visual Studiossa on mukana suunnittelunäkymä käyttöliittymien toteutukseen, sen mukana saa myös Blend for Visual Studio -sovelluksen. Sen avulla voi suunnitella ja toteuttaa XAML- ja HTML-pohjaisia käyttöliittymiä. [16.] Blend-sovellus helpottaa käyttöliittymien ohjelmoijaa komponenttien sapluunoiden tekemisessä ja tyylien määrittelyssä. Useimmiten Visual Studion suunnittelunäkymä on kuitenkin riittävä työkalu käyttöliittymien toteuttamisessa.

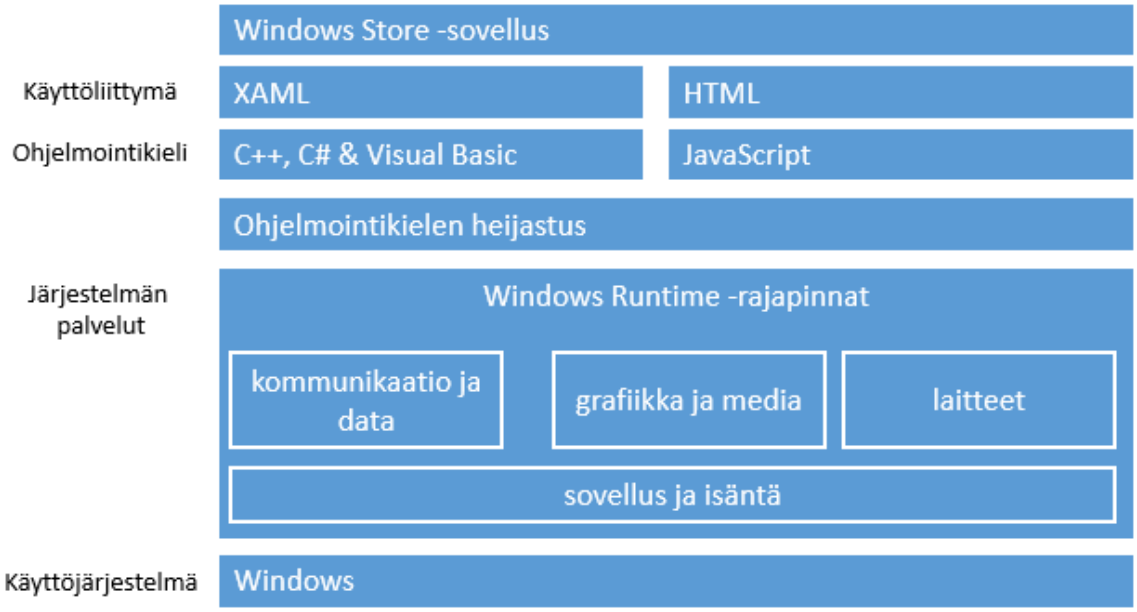
Visual Studio ja .NET-sovelluskehys sisältävät pääasialliset luokkakirjastot sovellusten kehittämiseen. Muiden kirjastojen hallintaan voi käyttää esimerkiksi Nuget-paketinhallintalaajennusta Visual Studiolle. Se integroituu Visual Studioon, ja sen kautta voi hakea, asentaa ja päivittää sekä Microsoftin että kolmansien osapuolien luokkakirjastoja. Lisäksi yritykset voivat jakaa omia sisäiseen käyttöön tarkoitettuja luokkakirjastoja esimerkiksi verkkolevyn tai oman Nuget-palvelimen kautta. [18.]

Alustojen eroavaisuuksien vuoksi Microsoft on luonut eri alustoille omat teknologiat ja ohjelmointimallit. Ne ovat usein riisuttuja versioita täydestä .NET-sovelluskehikosta, mutta saattavat sisältää joitakin alustalle ominaisia toteutuksia tai kehittämistä helpottavia tekijöitä. Esimerkiksi Windows Presentation Foundationia käytetään .NET-pohjaisten työpöytäsovellusten kehittämiseen Windows-käyttöjärjestelmille. ASP.NET kattaa teknologiat ja työkalut www-sovellusten ja -sivujen toteuttamiseen. Web-palveluiden toteuttamiseen voi käyttää ASP.NET- tai Windows Communication Foundation -alustoja (WCF). [37; 40.]

2.2 Windows Runtime -sovelluskehitysalusta

Sovelluskehitys Windows RT -käyttöjärjestelmälle tehdään uuden Windows Runtime -sovelluskehitysalustan päälle. Windows RT on kosketukselle optimoitu Windows 8 -käyttöjärjestelmän karsittu versio, ja se on suunniteltu vähemmän tehokkaille prosessoreille, joita käytetään tableteissa. Windows RT -käyttöjärjestelmälle kehitettyjä sovelluksia kutsutaan Windows Store -sovelluksiksi, koska niitä jaetaan Microsoftin Windows Storen -sovelluskaupan kautta. Windows Store -sovelluksia voi käyttää Windows RT- ja Windows 8 -käyttöjärjestelmissä. [28; 58.]

Windows Runtime antaa kehittäjille ohjelmointirajapinnat käyttöjärjestelmään, laitteisiin ja käyttöliittymään. Windows Store -sovelluksen voi kehittää C#-, Visual Basic-, JavaScript- ja C++-ohjelmointikielillä, ja sen käyttöliittymän voi toteuttaa HTML- tai XAML-pohjaisena. Jokaisella sovelluksella on niin sanottu sovellusinstanssi, jonka käyttöjärjestelmä luo automaattisesti kun sovellus käynnistetään. Sen kautta voi käyttää esimerkiksi sovelluksen asetuksia ja käyttöjärjestelmän toimintoja. Sovellusinstanssia suoritetaan sovellusisännän alaisuudessa, joka huolehtii järjestelmän resursseista ja sulkee muita sovelluksia vapauttaakseen resursseja tarpeen mukaan. [30; 12, s. 5–7.] Windows Runtime -rajapintojen lisäksi sovelluksissa voi käyttää osaa tavallisesta .NET-sovelluskehikosta. .NET-luokkakirjastosta on poistettu ne luokat, jotka eivät sovellu Windows Store -sovelluksiin. [39.] Kuvio 4 havainnollistaa Windows Runtime -arkkitehtuuria ja edellä mainittuja asioita.

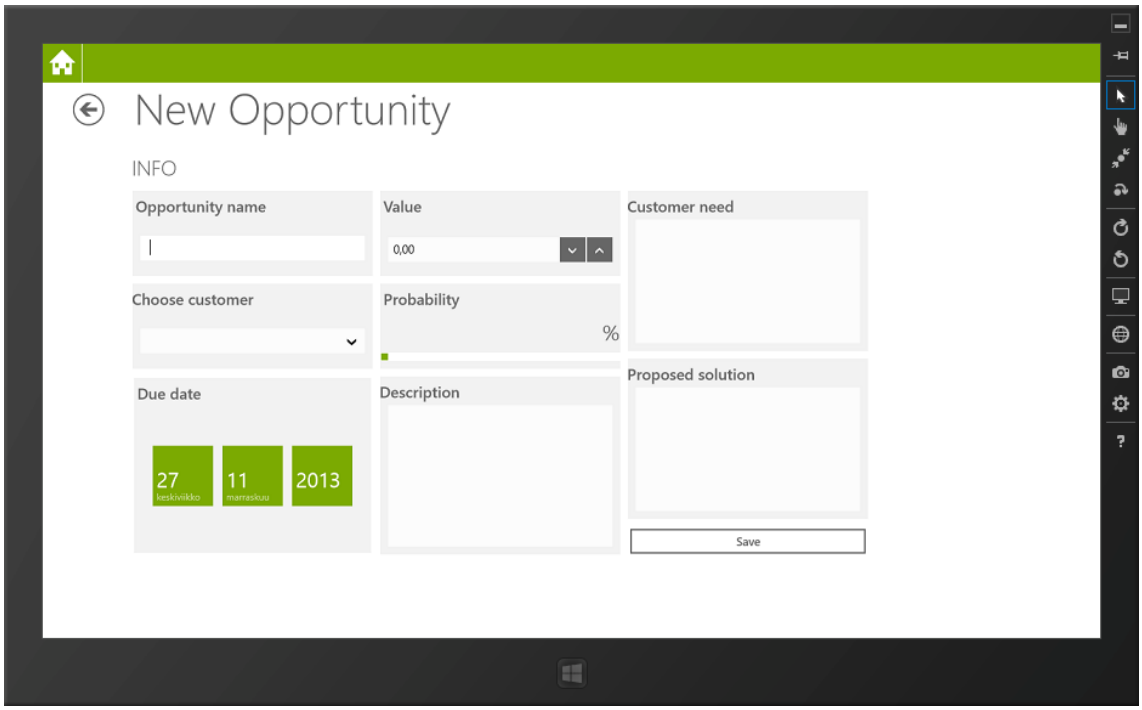


Kuvio 4. Windows Runtime -arkkitehtuuri [mukaillen lähteestä 30].

Tuki useille ohjelmointikielille on toteutettu niin sanotulla kielen heijastuksella (Language Projection). Kaikista luokista on lähdekoodin lisäksi metadataa, josta luodaan kääritty versio muille tuetuille kielille. Koska Windows Runtime on toteutettu C++-kielellä, muita kieliä käyttävät kehittäjät käyttävät käärittyjä objekteja. Heijastus mahdollistaa myös kehittäjille tavan toteuttaa jaettavia komponentteja eri kielille. [30; 12, s. 5–7; 29.]

Sovellusten kehittäminen

Windows Store -sovellusten kehittäminen vaatii ohjelmistokehityspaketin asentamisen. Joissakin Visual Studion versioissa se on jo valmiiksi asennettuna. Windows Store -sovellusten kehittäminen edellyttää Windows 8 -käyttöjärjestelmän käyttöä. [59.] Uuden sovelluksen voi aloittaa tyhjällä projektilla tai käyttää Visual Studion valmiita projektimallipohjia, jotka sisältävät esimerkiksi aloitusnäkymän ja yksittäisen objektin tarkempia tietoja näyttävän näkymän. Sovellusta voi testata kehityksen aikana tietokoneella, tabletilla tai emulaattorilla, joka tulee ohjelmistokehityspaketin mukana (kuva 1). Emulaattorilla voi esimerkiksi testata sovellusta eri resoluutioilla sekä pysty- että vaakatilassa. [5, s. 34–43.]



Kuva 1. Windows RT -emulaattori.

Sovelluksen kehittämisessä tulee ottaa huomioon tabletteihin ja Windows Runtimeen liittyviä ominaisuuksia ja rajoitteita. Windows Runtime ei tue esimerkiksi varsinaista moniajoa Windows Store -sovelluksille. Ainoastaan käyttäjälle näkyvillä olevat sovellukset ovat toiminnassa ja muut aikaisemmin avoinna olleet sovellukset ovat joko keskeytetyssä tilassa tai suljettu sovellusisännän toimesta, kun järjestelmä tarvitsee lisää resursseja. Kun sovellus tuodaan takaisin keskeytetystä tilasta, sen tulisi olla samassa tilassa, jossa käyttäjä sen viimeksi näki. Kehittäjän on siis tallennettava sovelluksen tila käyttäjän siirtyessä toiseen sovellukseen. [7.] Windows Store -sovelluksissa ja muissa XAML-pohjaisissa sovelluksissa käyttöliittymä toimii yhden säikeen alaisena. Jos siinä suorittaa pitkiä operaatioita, se käytännössä tukkii käyttöliittymän ja käyttäjän toimet jäävät huomiotta tai tapahtuvat vasta operaation päätyttyä. Sovelluksessa pitää siis toteuttaa kaikki raskaat ja pitkät operaatiot, kuten tiedostojen käsittelyt, asynkronisena. Tähän on hyvä käyttää aikaisemmin mainittua async/await-mallia, koska se siirtää suorituksen pois käyttöliittymän säikeeltä, jolloin se pystyy vastaanottamaan käyttäjän toimia. Operaation päätyttyä järjestelmä palauttaa suorituksen automaattisesti takaisin käyttöliittymän säikeelle. [30; 2.]

Sovelluksen käyttöliittymän ja käytettävyyden suunnittelu

Microsoft on luonut ohjeet erityyppisten sovellusten käyttöliittymien ja käytettävyyden suunnittelua ja toteutusta varten. Ohjeet ovat hyviä etenkin aloitteleville Windows Store -sovellusten kehittäjille. Ohjeista selviävät esimerkiksi yleisimmät navigointimallit, käyttöliittymäkomponenttien sijoittelut ja tuotemerkin rakentaminen [50]. Kaikkia näitä ohjaavat Microsoftin esittelemät viisi periaatetta:

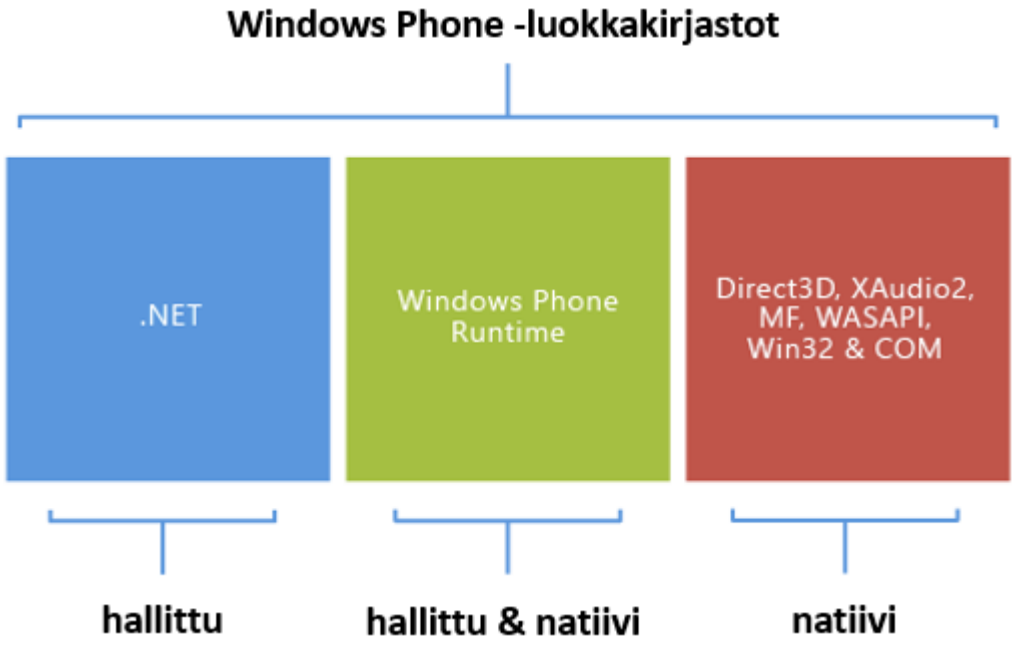
1. *pride in craftsmanship*: ylpeys käsityötaidosta
2. *be fast and fluid*: nopeus ja sulavuus
3. *authentically digital*: digitaalisesti autenttinen
4. *do more with less*: tee enemmän vähemmällä
5. *win as one*: voita yhtenä.

Ensimmäinen periaate ohjaa sovelluksen kehittäjiä tekemään täydellistä työtä ja välittämään pienistäkin yksityiskohdista. Toinen periaate ohjaa kehittäjiä toteuttamaan sovellukset sitä varten, mihin Windows RT on luotu: kosketukselle. Kolmas periaate ohjaa välttämään oikean maailman imitointia sovelluksissa. Neljäs periaate korostaa sisällön tärkeyttä asettamalla sisältö ulkoasun edelle. Käytännössä se tarkoittaa sitä, että käyttöliittymän tulisi keskittyä sisältöön eikä koristeisiin tai toimintoihin. Viides periaate ohjaa ottamaan muut sovellukset käyttöön omassa sovelluksessa ja korostaa yhtenäisyyttä koko Windows Runtime -sovellusympäristössä. [34.] Viides periaate tarkoittaa esimerkiksi sitä, että jos oma sovellus keskittyy ruoanlaittoon ja tarjoaa reseptejä ynnä muuta, reseptien jakaminen sosiaaliseen mediaan tehdään jollain muulla sovelluksella, joka on sitä varten tehty. Olen kuitenkin itse pitänyt sitä hyvänä ohjeena, jos toteuttaa samaa sovellusta useille eri alustoille, sillä sovelluksen on oltava yhtenäinen laitteesta riippumatta. Näistä periaatteista pidän sitä tärkeimpänä, koska harva haluaa tehdä hidasta sovellusta, josta ei ole ylpeä.

2.3 Windows Phone -sovelluskehitysalusta

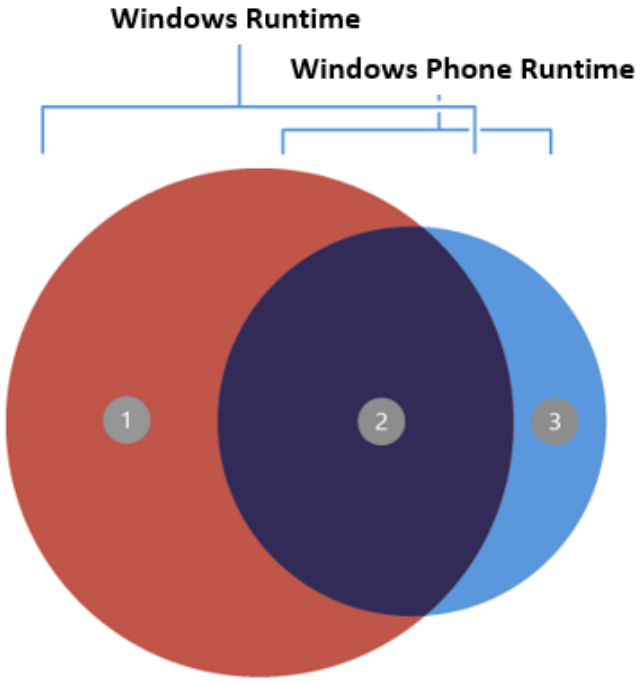
Windows Phone on Microsoftin käyttöjärjestelmä puhelimille. Sen uusin versio on Windows Phone 8, ja sovelluskehitys sille tehdään Windows Phone SDK 8.0 -ohjelmistokehityspaketilla [35]. Paketti on jaettu kolmeen osaan, joihin kuuluvat .NET-

rajapinta, Windows Phone Runtime -rajapinta ja kokoelma natiivin koodin rajapintoja (kuvio 5). [52.] SDK 8.0 antaa kehittäjille paljon uusia ominaisuuksia hyödynnettäviksi, muun muassa lähiluvun (near-field communication) ja internetpuhelut (voice over IP). SDK tuo myös parannuksia sovelluskehitystyökaluihin ja sovellusten suorituskykyyn [51; 55].



Kuvio 5. Windows Phone -luokkakirjastot jakautuvat kolmeen osaan [52].

Ensimmäinen osio, eli .NET-luokkakirjasto, on kokoelma luokkia, joista osa on samoja kuin muissa täysissä .NET-luokkakirjastoissa, ja osa on täysin Windows Phonelle tarkoitettuja luokkia [54]. Toinen osio on Windows Phone Runtime -luokkakirjasto, jota on havainnollistettu kuviossa 6. Se sisältää osan Windows Runtimen -kirjastosta (2), johon on lisätty Windows Phonelle olennaisia luokkia puhelimessa olevien toimintojen käyttöön (3). [56.] Kolmas osio, eli natiivit rajapinnat, sisältävät muun muassa grafiikan piirtämiseen, kameraan ja käyttöjärjestelmään liittyvät rajapinnat, joita voi käyttää esimerkiksi pelien, kuvankäsittely- tai mediasovelluksen kehittämisessä [60].



Jaetut rajapinnat (2.)
verkko
laitteiden väliset yhteydet
ostaminen sovelluksen sisällä
sensorit
sijainti ja GPS
tiedostojärjestelmä
säikeistys
sovelluksen ydin
käyttöliittymän toteutus

Kuvio 6. Windows Phone Runtime- ja Windows Runtime -kirjastot jakavat osan rajapinnoista [mukaillen lähteestä 6].

Windows Phone 8 -sovelluksia voi kehittää C#-, VB.NET- ja C++-ohjelmointikielillä, ja käyttöliittymän toteuttamiseen käytetään XAML-merkintäkieltä. HTML-pohjaiset sovellukset eivät ole tuettuja, koska JavaScript ja HTML eivät ole tuettuja kieliä Windows Phone 8 -sovelluskehityksessä. Rajoituksen voi kuitenkin kiertää tekemällä sovelluksen, johon upotetaan selainkomponentti. Omia ja SDK:n luokkia pystyy näin käyttämään selaimen InvokeScript-metodin ja ScriptNotify-tapahtuman kautta. [51; 52.]

Windows Phone 8 on toinen versio Windows Phone -käyttöjärjestelmästä, ja sille kehitetyt sovellukset eivät toimi edellisen sukupolven laitteissa. Windows Phone 8 on kuitenkin osittain taaksepäin yhteensopiva, ja Windows Phone 7:lle kehitetyt sovellukset toimivat yleensä uudella versiolla. Luokkakirjastoissa ja käyttöjärjestelmissä on kuitenkin huomattava määrä eroja, joten edellisen sukupolven sovelluksien toimivuus pitää tarkistaa Windows Phone 8 -laitteella. [55.]

Sovellusten kehittäminen

Ohjelmistokehityspaketin asentaminen antaa tarvittavat luokkakirjastot ja työkalut, mukaan lukien emulaattorin, sovellusten kehittämiseen ja testaamiseen [17]. Muita tarvit-

tavia kirjastoja voi lisätä Nugetilla sovelluksen tarpeiden mukaan. Esimerkiksi Windows Phone -ohjelmistokehityspaketin kehittäjät julkaisevat vapaasti käytettäviä käyttöliittymäkomponentteja, jotka eivät ehtineet mukaan ohjelmistokehityspakettiin [57]. Sovellusta voi testata pelkästään emulaattorin avulla, mutta sitä olisi hyvä testata myös laitteella. Puhelin on kuitenkin rekisteröitävä, jotta siinä voi käynnistää sovelluksen Visual Studion kautta. Rekisteröinnin jälkeen puhelimessa toimivat samat testaus- ja virheenetsintämahdollisuudet kuin emulaattorissa. [19; 21.]

Windows Phone 8 -sovelluksien kehittämisessä on syytä huomioida seuraavat piirteet, jotka ovat osittain samoja asioita kuin Windows Store -sovelluksien kehittämisessä. Näitä asioita ovat esimerkiksi moniajon rajoitteet, sovellusprosessin keskeyttäminen, tilan tallentaminen ja uudelleen käynnistäminen [27]. Uuden ohjelmistokehityspaketin myötä Windows Phone -sovelluksissa voi myös käyttää async/await-mallia. [2].

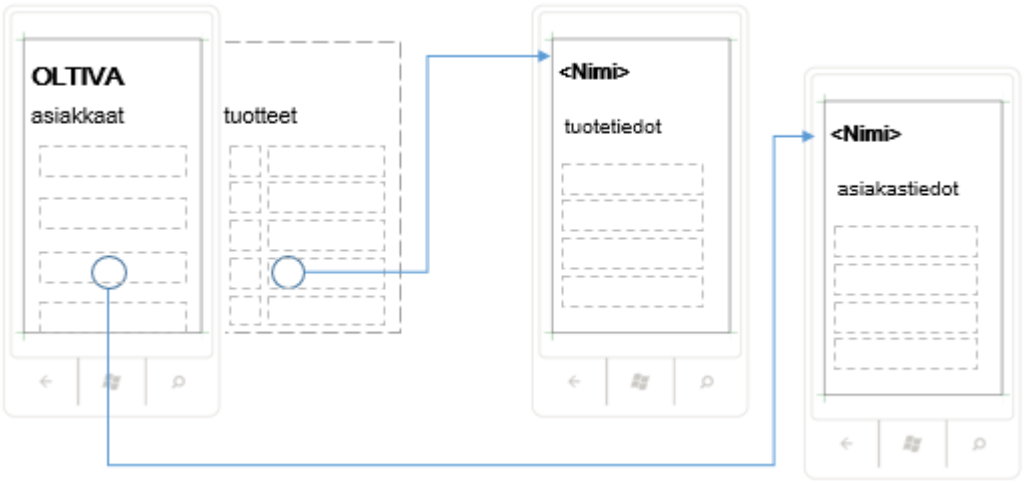
Sovelluksen käyttöliittymän ja käytettävyyden suunnittelu

Windows Phone -sovellusten käyttöliittymän ja käytettävyyden suunnittelussa ja toteutuksessa tulee ottaa huomioon samat periaatteet kuin Windows Store -sovellusten toteuttamisessa. Kun sovellusta tehdään usealle laitealustalle, mielestäni tärkein periaate on ”Win as One” -periaate. Sovelluksen käyttöliittymä on toteutettava puhelimelle ominaisena, mutta yhtenäisenä sovelluksen muiden versioiden kanssa, kuten kuvassa 2 on tehty. [53.]

Kuva 2. Microsoftin designesimerkeissä sovellukset ovat yhtenäisen näköisiä eri alustoilla, mutta ominaisia omalle alustalleen [53].

Sovelluksen käyttöliittymää suunniteltaessa ja toteutettaessa on myös hyvä huomioida, toimiiko sovellus pelkästään vaaka- tai pystytilassa vai molemmissa tiloissa. Toinen

tärkeä osa-alue sovelluksessa on navigointimalli. Yhdessä ne vaikuttavat yhdessä käytettäviin käyttöliittymäkomponentteihin. [11.] Kuviossa 7 havainnollistetaan sovelluksen navigointimallia. Esimerkkisovelluksella on päänäkymä ja kaksi muuta näkymää, joissa näytetään valitun tietueen tarkemmat tiedot. Päänäkymään upotetaan panoraamakäyttöliittymäkomponentti, jonka sisälle on upotettu eri dataa näyttäviä listakomponentteja. Tietoihin navigoidaan painamalla haluttua tietuetta. Tietoja esittävät näkymät voi myös toteuttaa panoraamana tai vaihtoehtoisesti jotain muuta komponenttia käyttämällä.



Kuvio 7. Esimerkki navigointimallista.

3 Alustariippumattoman sovelluskehityksen periaatteita Runtime-ympäristöissä

Monimutkaisuus ja satunnaisuus lisäävät sovelluksen huonoja piirteitä. Kun huonoja piirteitä on liikaa, sovelluksen lähdekoodia voidaan kutsua jo mädäntyneeksi (code rot). Huono suunnittelu, kehno koodi ja väärät valinnat tulisi korjata mahdollisimman aikaisin. [22, s. 4.] Martinin [31] mukaan sovellus on huonosti suunniteltu, vaikka se toteuttaisi kaikki määritellyt ominaisuudet, mutta olisi yksi tai useampia seuraavista kolmesta ominaisuudesta:

- Sovellus on liian jäykkä (rigid).
- Sovellus on liian hauras (fragile).
- Sovellus on liikkumaton (immobile).

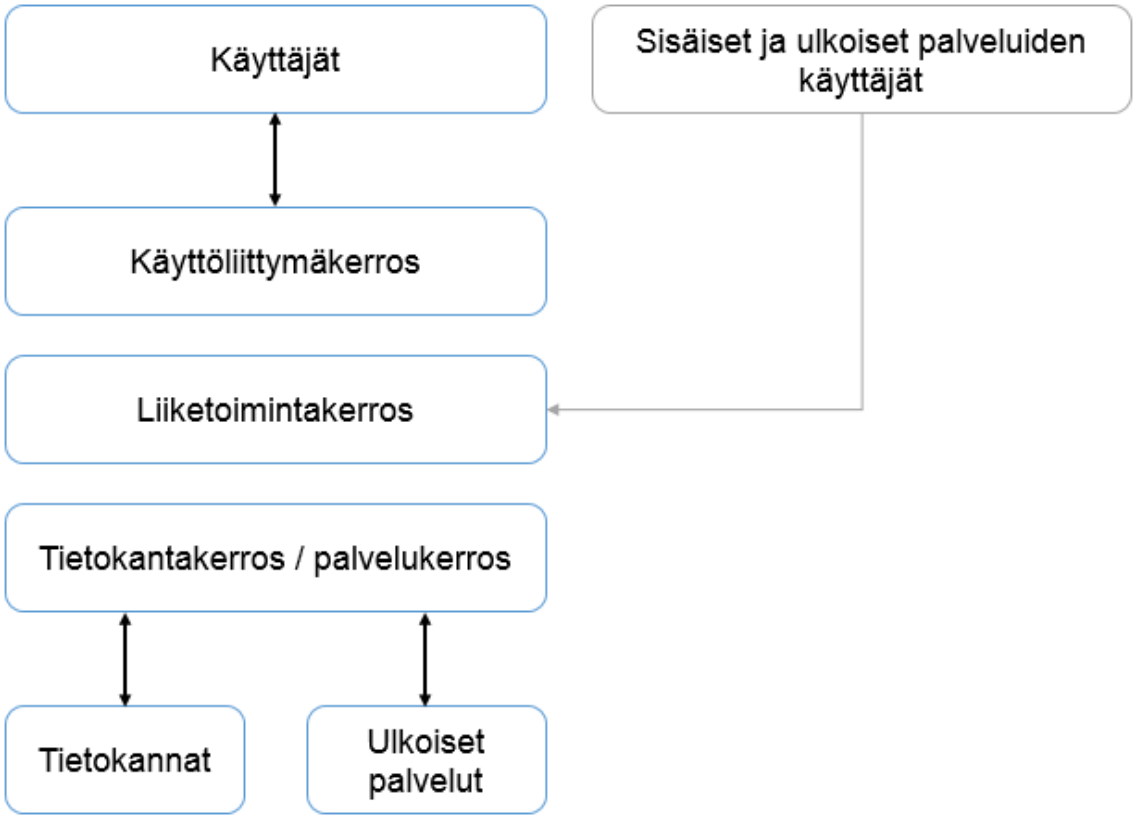
Sovellus on liian jäykkä, kun muutoksia on vaikea tehdä, koska vaikutukset ovat liian laajoja. Kun muutos vaikuttaa liian laajalti sovelluksessa, se johtuu usein liian riippuvaisista moduuleista, joissa yksi muutos valuu muihin moduuleihin. Kun muutoksia tehdään ja sovellus menee rikki arvaamattomissa paikoissa, se on liian hauras. Rikkoutuminen vähentää sovelluksen uskottavuutta ja muutoskykyä. Liikkumaton sovellus on huonosti uudelleenkäytettävissä, koska järjestelmän riippuvaisuudet ovat sekaisin ja ristiin kytkettyjä. Tässä tapauksessa sovellusta ei voida siirtää helposti uudelle alustalle, koska työn määrä on liian suuri. Näiden kolmen ominaisuuden vastakohtina ovat joustavuus, kestävyys ja uudelleen käytettävyys, jotka muodostavat hyvin suunnitellun sovelluksen pohjan. [31.] Luvussa 3.1 esitetään, kuinka käyttämällä separation of concerns -periaatetta ja sovelluksen modulaarista rakennetta pyritään välttämään sovelluksen huonoja ominaisuuksia ja toteuttamaan hyvien piirteiden mukainen järjestelmä.

3.1 Kerrostettu arkkitehtuuri ja modulaarisuus

Separation of concerns (SoC) on sovellusarkkitehtuurin suunnitteluperiaate, jonka tavoitteena on jakaa sovellus ja sen arkkitehtuuri eri osioihin tai moduuleihin niiden vastuualueiden mukaisesti ja välttäen päällekkäisiä toimintoja [44, s. 17; 36, s. 5]. Termi on esitetty tietojenkäsittelyyn liittyen jo 1970-luvulla professori Edsger W. Dijkstran esseessä tieteellisestä ajattelusta [13]. Noin 15 vuotta myöhemmin sitä on alettu käyttää sovelluskehitykseen liittyvässä kirjallisuudessa [46, s. 3; 36, s. 5]. Nykyään se on yleinen Microsoft .NET-teknologioiden yhteydessä esiintyvä periaate. Termi esiintyy sekä sovellusarkkitehtuuriin että käyttöliittymäsovellusten suunnitteluun ja toteutukseen liittyvässä kirjallisuudessa [33; 44, s. 17–23]. Meier ym. [33, s. 11; 33, s. 26] kutsuvat SoC-periaatetta yhdeksi pääsuunnitteluperiaatteeksi, koska se edistää joustavuutta ja ylläpidettävyyttä.

Kerrostettu arkkitehtuuri

Meier ym. [33, s. 11] kuvaavat kerrostetun arkkitehtuurin yhtenä SoC-periaatteen toteutustavoista, ja hyvin suunnitellun arkkitehtuurin avulla voidaan vähentää sovelluksen monimutkaisuutta. Hyvä arkkitehtuuri voidaan saavuttaa jakamalla arkkitehtuurisuunnitelma loogisesti eri osa-alueisiin, esimerkiksi käyttöliittymä-, sovelluksen logiikka- ja tietokantakerrokseen (kuvio 8), joita pidetään myös kolmena pääalueena [15, s. 17].

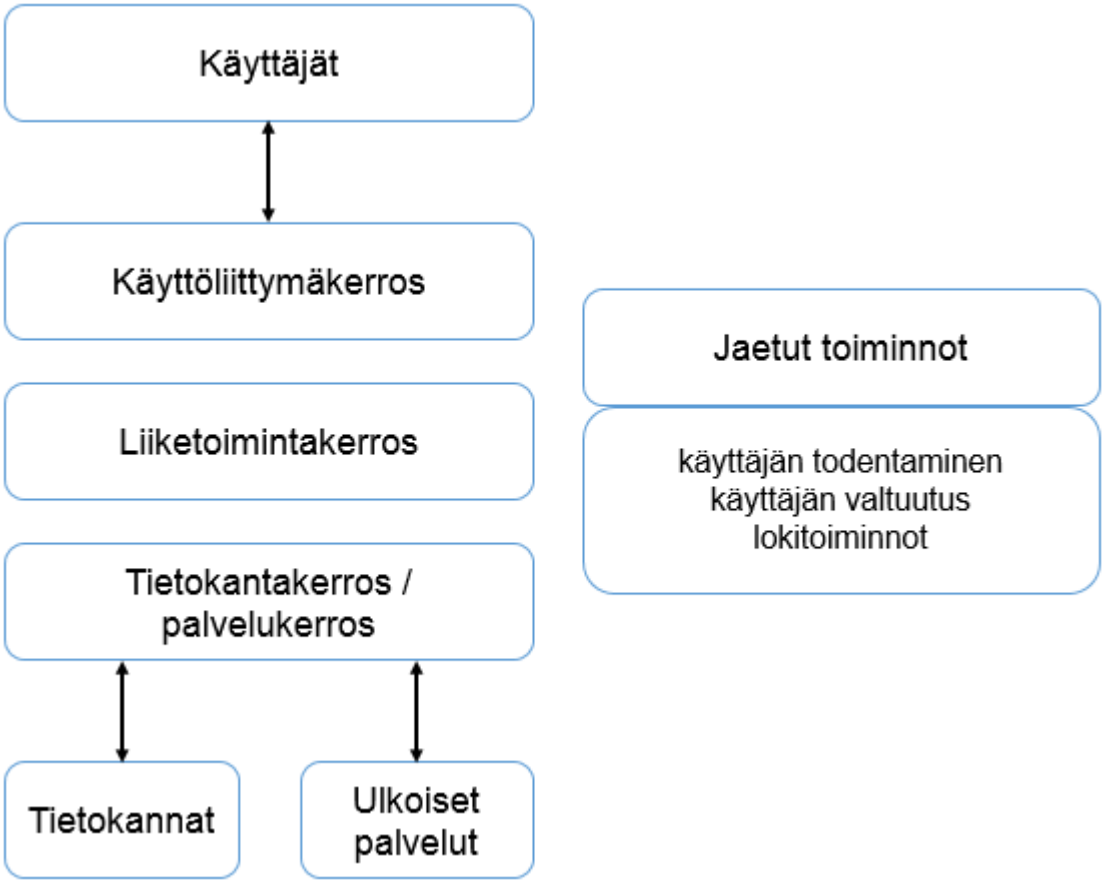


Kuvio 8. Kerrosarkkitehtuuri on jaettu loogisesti eri osa-alueisiin [33, s. 57].

Eri kerroksiin suunnitelluilla komponenteilla on tietyt roolit ja ominaisuudet. Komponentit eivät saa olla päällekkäisiä muiden kerrosten komponenttien kanssa ominaisuuksiltaan. Kerrostetun arkkitehtuurin esimerkkejä ovat muun muassa edellä mainitun tapainen kolmijako sovelluksen eri tasoille. Tämä sopii hyvin mobiili- tai käyttöliittymäsovellukselle. Kerrosarkkitehtuuri sopii myös web-sovelluksille, joille on ominaista, että käyttöliittymä ja liiketoiminta- sekä tietokantakerrokset sijaitsevat eri koneilla. [33, s. 21.] Yhden kerroksen komponentit voivat olla vuorovaikutuksessa sisäisesti ja seuraavan alla olevan kerroksen komponenttien kanssa ohjelmointirajapintojen kautta. Käyttöliittymäkerros käyttää liiketoimintakerroksen komponentteja, mutta ei ole lainkaan tietoinen tietokantakerroksen komponenteista. Näin saadaan aikaan niin kutsuttu löyhä kytkentä (loose coupling), jolloin muutokset eivät aina vaikuta muihin kerroksiin. [33, s. 26; 15, s. 17.] Hunt ja Thomas [22, s. 34–47] kuvaavat tällaista järjestelmää joustavaksi ja muunneltavaksi.

Joitakin yleisiä toimintoja ei pysty lokeroimaan tiettyyn kerrokseen. Nämä jaetut toiminnot tulee määrittää ja toteuttaa erillisenä muista kerroksista eikä niiden sisällä, koska niitä käytetään kaikista muista kerroksista. Näitä toimintoja ovat esimerkiksi käyttäjien

todentaminen, käyttäjän valtuutus ja lokitoiminnot, jotka on kuvattu kuviossa 9. [33, s. 17.] Jos jaetut toiminnot toteutettaisiin erikseen jokaiseen kerrokseen, myös muutokset pitäisi toteuttaa jokaiseen kerrokseen. Tämä vaatii huomattavan määrän ylimääräistä työtä ja rikkoo hyviä käytäntöjä koodin toistamisen suhteen. [33, s. 205; 22, s. 27.]



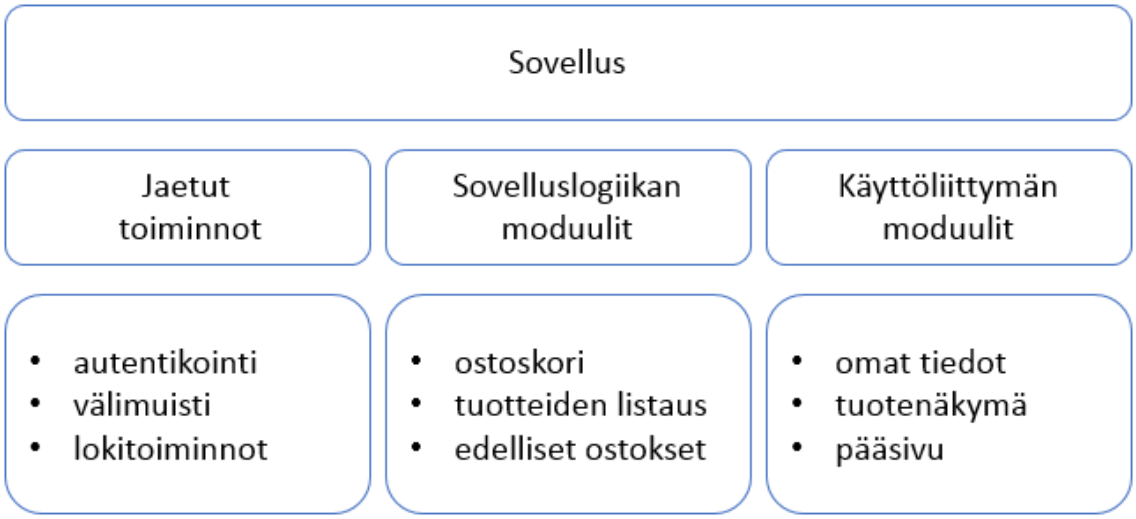
Kuvio 9. Jaetut toiminnot kerrosarkkitehtuurissa [mukaillen lähteestä 33, s. 59].

Meier ym. [33, s. 11] tarkentavat, että kerrosarkkitehtuurin toteutuksessa tulisi noudattaa muun muassa seuraavia periaatteita: abstraktio, kapselointi, koheesio, uudelleen käytettävyys ja löyhä kytkentä. Abstraktiolla pyritään käsittelemään asioita ilman niiden yksityiskohtia ja kapseloinnilla pyritään piilottamaan yksityiskohdat. Ne kulkevat usein käsi kädessä. Esimerkiksi sovellus voidaan käynnistää yhdellä metodilla, mutta sen takana saattaa tapahtua useita metodikutsuja, joista kehittäjällä ei tarvitse olla tietoa. Koheesio eli yhteenkuuluvuus tarkoittaa sitä, kuinka hyvin luokka tai sovelluksen moduuli keskittyy yhden toiminnon toteuttamiseen. Vahva koheesio merkitsee usein hyvää abstraktiota. [32, s. 89–90; 32, s. 138–139.] Edellä mainittuja periaatteita noudattamalla voi saavuttaa Huntin ja Thomasin [22, s. 34–47] kuvaaman järjestelmän, joka on kehitettävissä tuotettavammin ja kehityksen aikana esiintyvien riskien määrä vähenee.

Fowlerin [15, s. 18] mukaan kerrosarkkitehtuurissa ilmeneviin riskeihin sisältyy mahdolliset suorituskykyongelmat ja kaikkiin kerroksiin vaikuttavat muutokset, esimerkiksi uuden luokan lisäys tietomalliin, joka on lisättävä käyttöliittymään ja tietokantaan.

Sovelluksen modulaarinen rakenne

Jakamalla sovellus moduuleihin pyritään välttämään tiiviisti kytkettyjä sovelluksia (tightly coupled), jotka voi käsittää yhtenä yksikkönä. Modulaarinen sovellus koostuu siis tiettyä toimintoa suorittavista yksiköistä. Ne toimivat joko yksin tai löyhästi kytkettyinä toisiin moduuleihin. [3.] Modulaarisen sovelluksen hyötyihin voi lukea muun muassa paremman testattavuuden, laajennettavuuden ja ylläpidon. Sovellusta on myös helpompi kehittää. [3; 25.] Kuviossa 10 täydennetään kuviossa 9 esitettyjä arkkitehtuurikerroksia niihin liittyvillä moduuleilla.



Kuvio 10. Esimerkki kerrosarkkitehtuuria noudattavan sovelluksen moduuleista [mukaillen lähteestä 3].

Eri moduulien välillä tulisi välttää riippuvuuksia. Kuvan 3 mukainen esimerkkikoodi Ostoskori-luokasta on sidottu kiinni yhteen lokitoiminnon toteutukseen. Näin toteutuksen vaihto lokitoiminnoissa vaikuttaa suoraan Ostoskori-luokkaan ja muihin sitä käyttäviin luokkiin. Jos riippuvuus on pakollinen, sen tulisi olla löyhästi kytketty. Tämä saadaan aikaan ohjelmoimalla rajapintoja vasten. [3.] Kun ohjelmoidaan rajapintoja vasten, tavoitteena on saada aikaan Huntin ja Thomasin [22, s. 34] kuvaama riippumattomuus, jolloin käyttöliittymän muuttaminen ei vaikuta tietokantaan ja toisinpäin.

```
public class Loki
{
    void KirjaaLokiin(string msg)
    {
     ...
    }
}

class Ostoskori
{
    private Loki _loki;
    public void Ostoskori(Loki loki)
    {
        _loki = Loki loki;
    }

    private void LisaaTuote(Tuote tuote)
    {
        ...
        // Lisätään käyttäjän toiminto lokiin
        _loki.KirjaaLokiin
        (string.Format("Tuote {0} lisätty ostoskoriin.", tuote));
    }
}
```

Kuva 3. Esimerkkikoodi tiettyyn toteutukseen sidotusta luokasta.

Rajapinnan voi ajatella yksinkertaisesti luokan suunnitelmana. Rajapinta määrittelee mitä luokassa pitää toteuttaa, mutta ei miten toteutus tehdään. Kuvan 4 esimerkkikoodissa Loki-luokka toteuttaa lokitoiminnon määrittävän ILoki-rajapinnan. Kuvan 3 Ostoskori-luokkaan verrattuna, kuvassa 4 Ostoskorin riippuvuus Loki-luokkaan on vaihtunut lokitoiminnon määrittävään rajapintaan. Näin toteutus voidaan vaihtaa huonoksi todetusta toteutuksesta, johonkin toteutukseen jonka toimivuudesta voi olla varmempi. Kuvan 4 esimerkkikoodissa toteutettu rajapinnan käyttö toteuttaa riippuvuuden käänteisyyden periaatteen (dependency inversion), jossa moduulin riippuvuus toisesta moduulista toteutetaan abstraktiolla. Näin korkean tason moduulit, joita halutaan käyttää uudelleen, ovat riippumattomia alemman tason moduulien toteutuksesta. Kun riippuvuus syötetään komponentin konstruktorille, kuten Ostoskori-luokassa, toteutetaan riippuvuuden injektio -suunnittelumallia (dependency injection). [25; 31.]

```
public interface ILoki
{
    void KirjaaLokiin(string msg);
}

public class Loki : ILoki
{
    void KirjaaLokiin(string msg)
    {
     ...
    }
}

class Ostoskori
{
    private ILoki _loki;
    public void Ostoskori(ILoki loki)
    {
        _loki = loki;
    }

    private void LisaaTuote(Tuote tuote)
    {
        ...
        // Lisätään käyttäjän toiminto lokiin
        _loki.KirjaaLokiin(
        string.Format("Tuote {0} lisätty ostoskoriin.", tuote));
    }
}
```

Kuva 4. Käyttämällä rajapintoja konkreettisten luokkien sijaan riippuvuutta vähennetään.

Rajapintoja vasten ohjelmoitaessa hyötyy esimerkiksi yksikkötestauksessa ja käyttöliittymäohjelmoinnissa, koska kehittäjä voi vaihtaa toteuttavan luokan. Sen sijaan että yksikkötesteissä käytettäisiin oikeaa tietokantaa tai palvelua, kehittäjä voi toteuttaa rajapinnan erityiseen yksikkötestiä varten tehtyyn luokkaan, joka palauttaa tai käyttää pelkästään testidataa. Sama onnistuu myös tietyillä käyttöliittymäteknologioilla ja toteutustavoilla, kuten Model–View–ViewModel-arkkitehtuurimallilla. [25.]

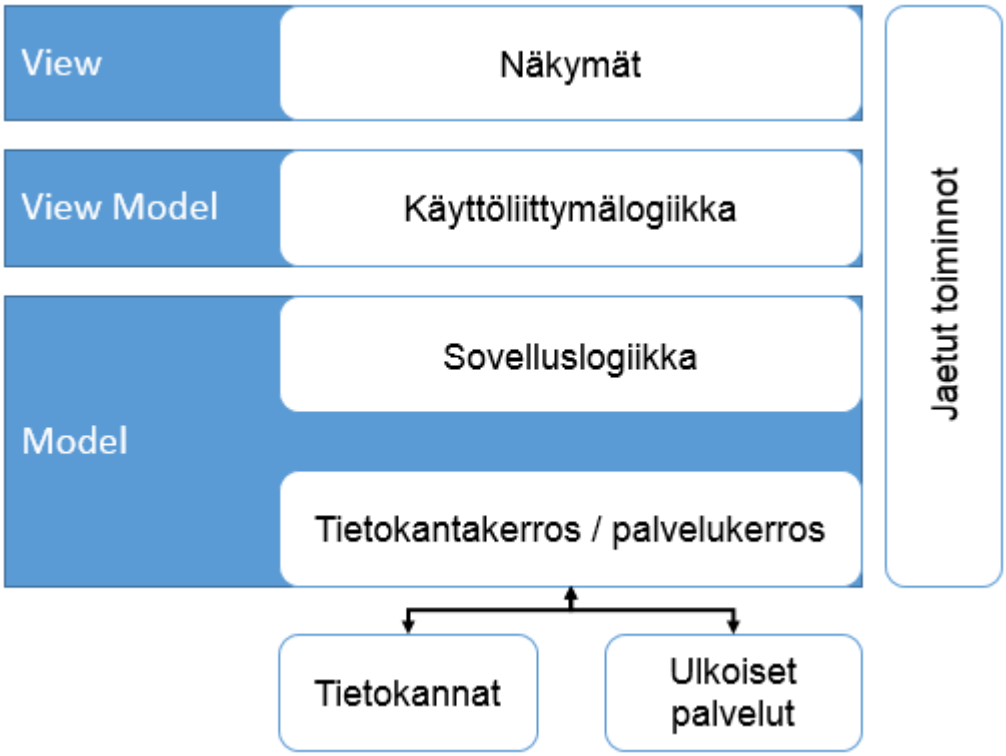
3.2 Model–View–ViewModel-arkkitehtuurimalli

Model–View–ViewModel-arkkitehtuurimalli (MVVM) on tarkoitettu käyttöliittymäsovellusten kehittämiseen. Se on kehitetty yhdessä Windows Presentation Foundationin kehittämisen yhteydessä ja tuotu julkisuuteen vuonna 2005. Julkaisun yhteydessä John

Gossman kuvaa MVVM-mallia sopivaksi moderneihin käyttöliittymäteknologioihin, ja sen tavoitteena on käyttöliittymän erottaminen tietomallista ja sovelluslogiikasta. [61.] Se on siis erityisesti XAML-pohjaisten käyttöliittymäsovellusten kehittämiseen tehty arkkitehtuurimalli, koska siinä käytetään hyväksi XAML-pohjaisten sovellusten ominaisuuksia [47]. Nykyisin MVVM-arkkitehtuurimallia käytetään Silverlight-, Windows Presentation Foundation-, Windows Phone- tai Windows Store -sovelluskehityksessä. [49; 5, s. 127.]

Toimintaperiaate

Microsoftin käyttöliittymäteknologiat pohjautuvat niin sanottuun tapahtumapohjaiseen ohjelmointiin (event-based programming). Kun käyttäjä tekee jotain, sovelluksessa käytettävät kontrollit nostavat erityyppisiä tapahtumia, joita kehittäjät voivat rekisteröidä ja määrittää niille käsittelijöitä. Ennen MVVM-arkkitehtuurimallia sovelluslogiikka kytkettiin tapahtumien käsittelijöihin, jotka tekivät näkymistä riippuvaisia sovelluslogiikasta ja tietomallista. MVVM-arkkitehtuurimallin periaate on erottaa käyttöliittymät sovelluslogiikasta ja tietomallista. Se on SoC-periaatteen mukaisesti jaettu kolmeen kerrokseen, jossa jokainen kerros on riippuvainen vain alemmasta kerroksesta (kuvio 11). Jaettujen toimintojen käyttö on MVVM-arkkitehtuurimallissa yleistä ja hyvä tapa estää kerrosten väliset riippuvuudet. [49.]

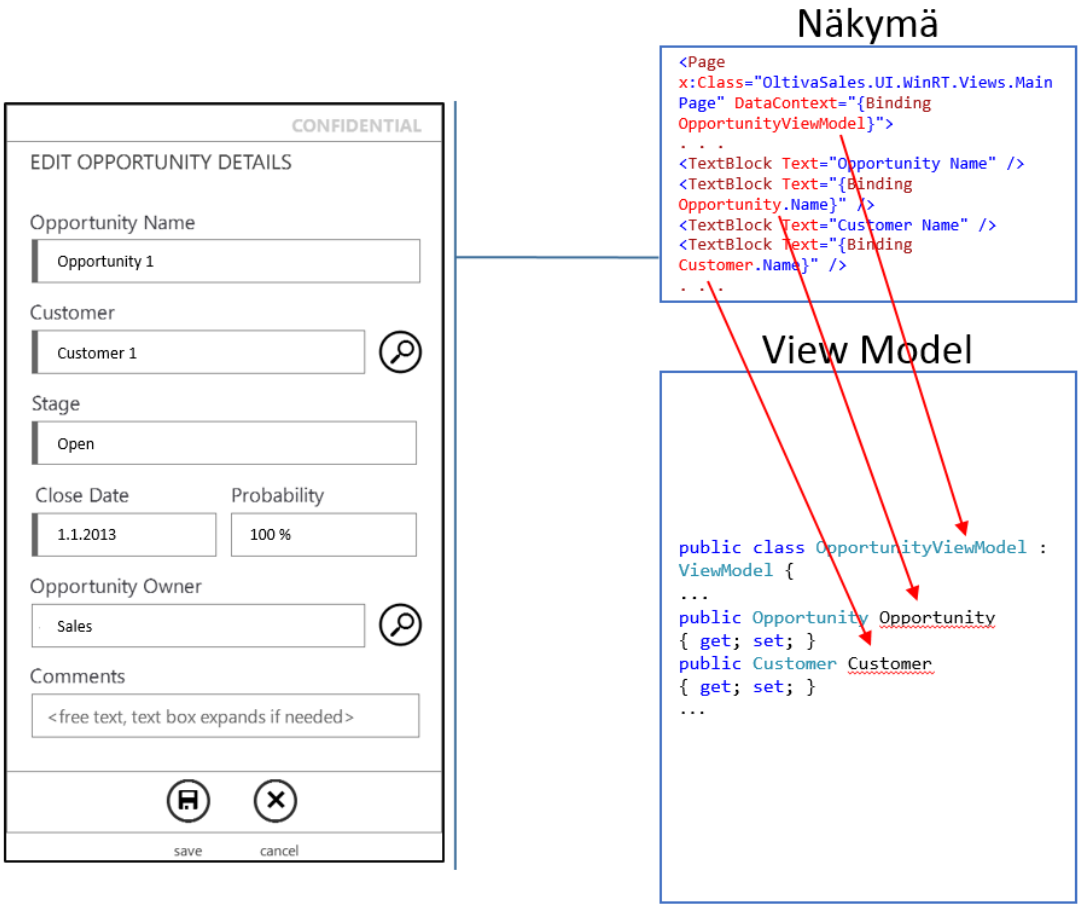


Kuvio 11. MVVM-mallin kerrokset.

MVVM-mallissa Model-kerrokseen sijoitetaan sovelluslogiikka, tietomalli ja palvelurajapinnat sekä tietokannan käyttö. Tämä kerros ei ole tietoinen kahdesta muusta kerroksesta. ViewModel-kerrosta kutsutaan näkymien tietomalliksi ja logiikkakerrokseksi. Se sitoo kaksi muuta kerrosta yhteen, mutta sen objektien ei tarvitse olla tietoisia näkymien (View-kerros) olemassaolosta. ViewModel-objekteihin kerätään usein tarvittavat tietomallin objektit ja sovelluslogiikka. Lisäksi näissä luokissa on logiikkaa, jolla käsitellään esimerkiksi näkymistä nostettuja tapahtumia, tarkkaillaan lomakkeiden oikeanlaista täyttöä tai haetaan näkymään tarvittava data Model-kerroksen luokkia käyttäen. View-kerrokseen tulee sijoittaa vain näkymät, eli niiden kuvaus, ja näkymien toimintaan liittyvä koodi, mutta ei sovelluksen logiikkaa. [49; 47.] Esimerkiksi osa tapahtumien (events) käsittelystä tehdään näkymissä, ja niistä hyviä esimerkkejä on vierittää sivu tiettyyn sijaintiin, jossa käyttäjä oli aikaisemmin, tai ponnahdusikkunan tuominen esiin. Niiden suorittaminen ei kuulu ViewModel-kerrokselle, koska se vaatii usein tietyn käyttöliittymäkomponentit metodia ja ViewModel-kerros ei ole tietoinen niiden olemassaolosta, eikä se vaadi sovelluslogiikan suorittamista.

Näkymälle annetaan instanssi halutusta ViewModel-objektista, jonka kautta käyttöliittymän komponentit sidotaan (data binding) ViewModel-objektin jäseniin. ViewModel-

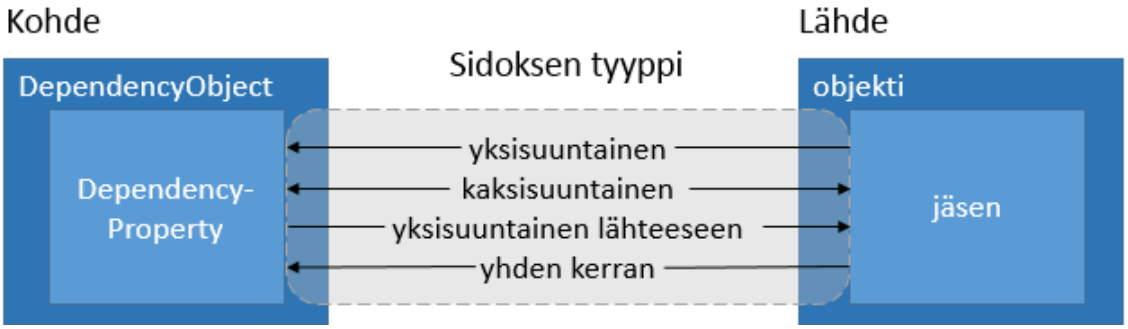
objektista tehdään näkymän konteksti (data context), ja sen data siirretään näkymiin automaattisesti (kuvio 12). Datan sidonnan käyttöön ja toimivuuteen on tiettyjä vaatimuksia. Se pitää aina luoda eksplisiittisesti. Kohteena olevan objektin pitää periä DependencyObject-luokka. Kuviossa 12 käytetyt sivu-, tekstilaatikko- ja painikekomponentit perivät kaikki DependencyObject-luokan. Jäsenen, johon sitominen tehdään, täytyy olla DependencyProperty-luokan objekti. Kuviossa 12 tekstilaatikoiden teksti (Text) on sidottu ViewModel-objektin Opportunity- ja Customer-jäsenten nimeen (Binding Opportunity.Name). [10; 49; 47.]



Kuvio 12. Näkymä on sidottu ViewModel-luokkaan.

Kuviossa 13 havainnollistetaan datan sidonnan eri tyyppejä, eli mihin suuntaan dataa päivitetään. Kun näkymä alustetaan ja ladataan, se käy läpi kaikki XAML:ssa ilmoitetut sidokset ja esittää saatavilla olevan datan. Sidoksen tyyppi on silloin yhden kerran (OneTime). Jos data kuitenkin haetaan esimerkiksi tietokannasta eikä se näin ole saatavilla alustuksen aikana, näkymä ei automaattisesti hae sitä uudestaan. Datan sidos -järjestelmä kuitenkin kuuntelee tiettyä PropertyChanged-tapahtumaa, jonka nostamalla

(raise event) voi ilmoittaa tapahtuneista muutoksista. Kun tapahtuma nostetaan, sille annetaan parametriksi jäsenen nimi, johon muutos kohdistui. Tällöin vain siihen sidotut käyttöliittymäkomponentit päivittyvät. Tämä sidoksen tyyppi on yksisuuntainen (OneWay). Jos dataa halutaan päivittää käyttöliittymästä lähteeseen, voidaan käyttää kaksisuuntaista (TwoWay) tai yksisuuntaista sidostyyppiä (OneWayToSource). Kaksisuuntainen sidos päivittää sekä kohteesta lähteeseen että lähteestä kohteeseen ja jälkimmäinen vain lähdettä [10.] Päivittämällä vain tarvittavat käyttöliittymäkomponentit säästetään resursseja, koska sovelluksen ei tarvitse ladata koko näkymää uusiksi. Lisäksi se tekee käyttökokemuksesta sujuvamman.



Kuvio 13. Sidosjärjestelmän objektit ja datan liikkuvuus niiden välillä [10].

Huomioita MVVM-mallin käytöstä

Muutoksien ilmoituksiin toteutetaan yleensä INotifyPropertyChanged-rajapinta [20]. Sen voi toteuttaa ViewModel-luokissa, mutta se kannattaa toteuttaa myös tietomallin luokissa. Tällä tavalla pystytään toteuttamaan kattavasti automaattinen tiedon päivittyminen näkymiin. Kun data on kokoelmatyyppistä, on hyvä käyttää valmista ObservableCollection-kokoelmaluokkaa, joka ilmoittaa kokoelman muutoksista automaattisesti. Jos kokoelmaluokan toteuttaa itse ja haluaa ilmoittaa kokoelman muutoksista käyttöliittymälle, kehittäjä voi toteuttaa INotifyCollectionChanged-rajapinnan. [10.] En kuitenkaan ole joutunut toteuttamaan omaa kokoelmaluokkaa, joka ilmoittaa muutoksista, koska ObservableCollection-luokka on riittänyt kaikkiin tarpeisiin.

MVVM-mallin käytössä tiedon validointiin on kaksi tapaa. Käyttöliittymästä tulevan syötteen validointiin käytetään ValidationRule-luokan perivää objektia, joka annetaan sidosobjektille tiedoksi. Useimmat käyttöliittymäkomponentit, esimerkiksi tekstilaatikko, näyttävät automaattisesti tätä kautta nostetut virheet. Tiedon validoinnin voi myös toteuttaa tietomallin tai ViewModel-kerroksen luokissa. Tällöin kannattaa toteuttaa

INotifyDataErrorInfo-rajapinta, jolloin validoinnissa löydetyt virheet saadaan käyttöliittymään automaattisesti datan sidonnan avulla. [10; 23.] Erottelen nämä tavat niin, että ValidationRule-luokilla validoidaan käyttöliittymästä tulevan syötteen muotoa tai oikeellisuutta. Esimerkiksi sovelluksen validointisäännöt määrittävät, että käyttäjän nimen pitää olla järjestelmässä yksilöllinen eikä se saa sisältää erikoimerkkejä. Ensimmäinen sääntö vaatii käyttäjänimen tarkastamista tietokannasta, joten se kuuluu toteuttaa Model-kerrokseen. Jos käyttäjänimi ei ole yksilöllinen, virheen voi nostaa Model- tai ViewModel-kerroksesta. Toinen sääntö vaatii vain syötteen muodon tarkastamista, joten sen voi toteuttaa ValidationRule-luokalla View-kerroksessa.

Useimmat käyttöliittymäkomponentit käsittelevät tiedon tekstinä, mutta tietomallissa sen tyyppi voi olla esimerkiksi numero, päivämäärä tai jokin muu komponentille sopimaton luokka, jolloin datan sidonta ei toimi. Toisaalta tekstin syöttäminen käyttöliittymästä lähteen jäseniin aiheuttaa poikkeuksia, jotka voivat kaataa sovelluksen. Tällöin pitää toteuttaa muuntajaluokka, jonka metodeissa muunnetaan lähteen data kohdetta varten, ja tarpeen mukaan toisinpäin takaisin lähteeseen. [10.] Tämä on yleistä, kun käyttöliittymässä pitää piilottaa komponentteja datan perusteella tai lomakkeesta tuleva data ei ole yhteensopivaa tietomallin kanssa, jolloin se täytyy muuttaa.

Jotkin käyttöliittymäkomponentit, kuten linkki ja nappi, sisältävät komentojäsenen (command). Kun käyttäjä painaa niitä, ne nostavat tapahtuman, jonka kehittäjä voi käsitellä. ViewModel-kerroksessa ei kuitenkaan voi rekisteröidä näitä tapahtumia, koska ne vaativat käyttöliittymäobjektin instanssin. Järjestelmä kuitenkin kierrättää tiettyjä tapahtumia, kuten napin painalluksen, komennon kautta. Komennon käsittely suoritetaan näkymien sijaan ViewModel-kerroksessa datan sidoksen avulla. [10.] Näin esimerkiksi lomakkeen tallentaminen tai tiedon haku voidaan suorittaa ViewModel-luokasta.

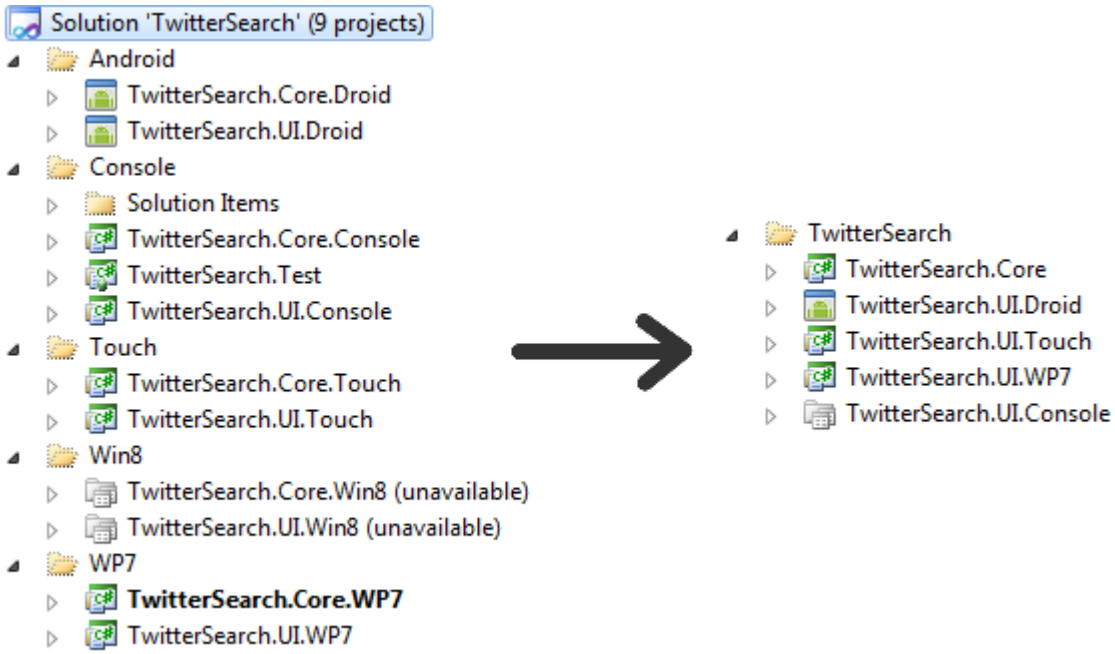
Joskus tietoa on pakko siirtää objektien välillä, mutta niillä ei ole suoraa yhteyttä tai instanssia, jonka kautta se onnistuisi. Tämä johtuu usein huonosta arkkitehtuurisuunnittelusta. Tällöin voi käyttää esimerkiksi viestinvälitystä ja lähettää viestejä objektien välillä. Viestien lähettäminen suoritetaan yhden luokan kautta, johon rekisteröidään tietyntyyppisen viestin vastaanottajat. Tätä toteutusta kutsutaan MVVM-mallin yhteydessä usein Messengeriksi [4]. Tällöin koodista tulee kuitenkin vaikeaselkoista ja vikojen etsinnästä hankalaa, kun pitää selvittää mistä viestejä lähetetään ja missä niitä vastaanotetaan. Toinen vastaava tapa on käyttää delegaatteja ja suorittaa tarvittava metodi

delegaatin kautta. Delegaattien käyttö on yhtä lailla vaikeaselkoista, ja niiden toinen huono puoli on, että ne aiheuttavat muistivuotoja, jos kehittäjä ei muista poistaa delegaattia sen suorittavasta objektista. Oman kokemukseni perusteella näiden tapojen käyttö johtaa siihen, että niitä käytetään liialti kaikkien ongelmien ratkaisemiseen sen sijaan, että korjattaisiin ongelmien lähde.

3.3 Portable Class Library -työkalu

Portable Class Library (PCL) on työkalu Visual Studiossa, joka tukee .NET-sovelluskehitystä monelle alustalle yhtä aikaa. Visual Studio hoitaa automaattisesti käytössä olevien .NET-luokkakirjastojen käyttöönoton, tarkistaa manuaalisesti lisättyjen luokkakirjastojen yhteensopivuuden ja kääntää luokkakirjaston alustalle sopivaksi. Tuettuja ympäristöjä ovat Silverlight, Xbox 360, Windows- ja Windows Phone -käyttöjärjestelmät. Windows-käyttöjärjestelmille voi kehittää sovelluksia käyttämällä esimerkiksi Windows Presentation Foundationia tai Windows Runtimea. Käytännössä PCL on Visual Studion projekityyppi luokkakirjaston toteuttamiseen. Se on uusi työkalu, joka julkaistiin Visual Studio 2012:n kanssa. Perinteisessä Visual Studion luokkakirjastoprojektissa voi valita vain yhden tuetun alustan kerrallaan. Esimerkiksi Windows Presentation Foundationille kehitettyä luokkakirjastoa ei voi hyödyntää Windows Runtimessä. Jos sovellus pitää toteuttaa myös Windows Phonelle, tarvittaisiin jo kolme luokkakirjastoprojektia. [9.]

PCL:n avulla voi siis kehittää esimerkiksi sovelluksen logiikan, tietomallin ja datan käytön yhden kerran ja jakaa ne luokkakirjastona eri käyttöliittymäsovelluksille. Se on huomattava etu verrattuna siihen, että kehittäjä ylläpitää useita projekteja ja valitsee asetukset sekä käytettävät luokkakirjastot jokaiselle alustalle erikseen. Kuvassa 5 havainnollistetaan erään sovelluksen projektirakennetta (vasen puoli), jossa eri alustoille on omat luokkakirjastoprojektit. Käyttämällä PCL-luokkakirjastoprojekteja (oikea puoli) tarvitaan vain yksi luokkakirjastoprojekti ja jokaiselle alustalle oma käyttöliittymän toteutus.



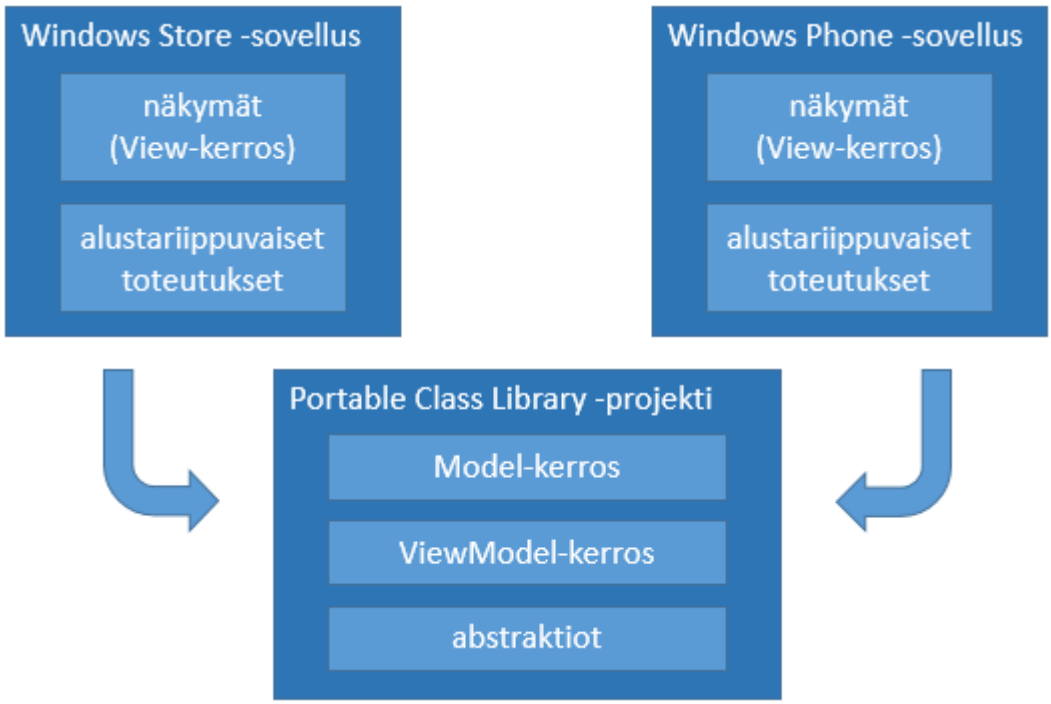
Kuva 5. Erään sovelluksen projektijako ennen PCL:n käyttöä, ja PCL:n käyttöönoton jälkeen [42].

Kun kehittäjä luo PCL-projektin, hän joutuu valitsemaan alustat, joille sovellus kehitetään. Tämä vaikuttaa käytössä oleviin .NET-luokkakirjaston ominaisuuksiin, sillä kaikki eivät ole tuettuja eri ympäristöissä. Portable Class Library -projektiin ei voi lisätä luokkakirjastoja, joita ei ole toteutettu PCL-yhteensopivana. Läheskään kaikkia täyden .NET-luokkakirjaston ominaisuuksia ei ole toteutettu yhteensopivana Portable Class Libraryn kanssa, koska niitä ei ole suunniteltu monialustaiseen käyttöön. Niiden muokkaaminen yhteensopivaksi vaatii paljon aikaa ja työtä. Tällä hetkellä vain käytetyimmät on tehty PCL-yhteensopivaksi. Lisäksi . NET-luokkakirjastossa on joitakin vanhentuneita ja vanhentuvia toteutuksia, joita ei ole syytä muuttaa PCL-yhteensopivaksi. Kaikkia ominaisuuksia ei ole järkevää toteuttaa PCL-yhteensopivana, koska ne on jo toteutettu erikseen esimerkiksi Windows Phone-, Windows Runtime- ja täydessä .NET-luokkakirjastossa, mutta toteutukset ovat hyvin erilaisia. Alustalle ominaisia toteutuksia ei tehdä PCL-yhteensopivana, koska esimerkiksi puhelimen toimintoihin liittyvillä luokilla ei ole käyttötapauksia tietokoneella. [42; 9.]

MVVM-mallin tuki ja hyödyntäminen Portable Class Libraryssä

MVVM-mallin tukea varten tarvittavat luokat ja rajapinnat, joiden avulla käytetään datan sidosta, on toteutettu PCL-yhteensopivina. Kun sovellus kehitetään usealle alustalle

Portable Class Libraryn avulla, Model- ja ViewModel-kerrokset toteutetaan PCL-projektissa ja näkymät (View-kerros) toteutetaan jokaiselle alustalle erikseen (kuvio 14). Kuten jo aikaisemmin mainittiin, Portable Class Library -projektiin ei voi lisätä luokkakirjastoja, jotka eivät ole PCL-yhteensopivia. Windows Store- ja Windows Phone -sovelluksissa niitä ovat esimerkiksi navigointiin, tallennukseen ja sovelluksen elinkaareen liittyvät luokat. Kuviossa 14 on havainnollistettu, että nämä alustariippuvaiset toiminnot abstrahoidaan PCL-projektiin, eli niitä varten määritellään rajapinnat. Rajapintojen toteutukset tehdään jokaiselle käyttöliittymäsovellukselle erikseen. [42; 9.].



Kuvio 14. MVVM-mallin kerrosten jako PCL-projektissa [mukaillen lähteestä 42].

Luvussa 3.1 esiteltiin dependency inversion -periaatteen käyttöä rajapintoja vasten ohjelmoitaessa. Lisäksi esitettiin rajapinnan toteutuksen syöttöä konstruktorille käyttämällä dependency injection -suunnittelumallia. Kuvan 6 esimerkkikoodissa käytetään inversion of control -ohjelmointitekniikkaa (IoC) toteuttamaan dependency injection -suunnittelumallia. Inversion of control -tekniikassa käytetään niin sanottua inversion of control containeria, jolle määritetään rajapinnat tai luokat, jotka alustetaan sen kautta. Kuvassa 6 container-objektille määritetään välimuistin käyttöä määrittelevä rajapinta ja rajapinnan toteutus sekä yksi ViewModel-luokka (1.). Koska rajapinta ja ViewModel-luokka on molemmat määritelty container-objektille, rajapinnan toteutus syötetään automaattisesti ViewModel-luokan konstruktorille (2.), jos luokka alustetaan container-

objektin kautta (3.). IoC-tekniikan toteutuksena käytetään tässä insinöörityössä Microsoftin luokkakirjastoa, koska niiden toteuttaminen on haastavaa. Kuvassa 6 on myös esitelty, kuinka ViewModel-luokan instanssi voidaan sitoa näkymään datan sidoksen toteuttamiseksi (4.).

```
public static class Container
{
 // Create the singleton container
 private static readonly IUnityContainer _current = new UnityContainer();
 public static IUnityContainer Current { get { return _current; } }

 public static void ConfigureUnityContainer()
 {
   Current.RegisterType<ICacheService, FolderCacheService>(
        new ContainerControlledLifetimeManager());                    1.
   Current.RegisterType<MainPageViewModel>(
        new ContainerControlledLifetimeManager());
 }
}
...
public MainPageViewModel(ICacheService cache)   2.
...
public class ViewModelLocator
{
    public static MainPageViewModel MainPageViewModel     3.
    { get { return Container.Current.Resolve<MainPageViewModel>(); } }
}
...
<Page x:Class="OltivaSales.UI.WinRT.Views.MainPage"
DataContext="{Binding MainPageViewModel, Source={StaticResource      4.
ViewModelLocator}}">
```

Kuva 6. Dependency injection -suunnittelumallin hyödyntäminen MVVM-arkkitehtuurimallissa.

Toinen vaihtoehto on antaa kääntäjälle ohjeita ja määrittää tarvittavat toteutukset käännöksen aikana. Tämä on nopein ja mahdollisesti helpoin tapa, mutta se ei kuitenkaan toteuta hyvän sovellusarkkitehtuurin periaatteita. Kolmas vaihtoehto on ohjelmoida rajapintoja vasten ja käyttää samalla kääntäjäohjeita, jos toteutustapojen erot ovat pieniä. [14.] Kuvan 7 esimerkkikoodissa Windows Store- ja Windows Phone -sovelluksen navigointi on toteutettu kääntäjäohjeilla. Ensimmäinen lohko (#if) tarkistaa suorituksen aikana, onko sovelluksen tyyppi Windows Store -sovellus. Toinen lohko (#elif) tarkistaa saman Windows Phone -sovellukselle. Ympäristöjen erot navigoinnin suhteen ovat kohtalaisen pienet. Windows Phone -ympäristössä navigoinnin parametrina voi käyttää vain tekstiä, kun Windows Store -sovelluksessa parametriksi voi antaa

objektin. Lisäksi näkymä, jolle navigoidaan, annetaan erityyppisenä luokkana, mutta se haetaan molemmissa nimen perusteella.

```
public bool Navigate(string pageName, object parameter)
{
 #if (NETFX_CORE) // Windows 8
  var type = Type.GetType(pageName);
  return _frame.Navigate(type, parameter);
 #elif (WINDOWS_PHONE)
  if (parameter == null)
  {
    return _frame.Navigate(
        new Uri(string.Format("/Views/{0}.xaml", pageName))
    );
  }

  return _frame.Navigate(new Uri(
    string.Format("/Views/{0}.xaml?parameter={1}", pageName, parameter))
  );
 #else
  throw new Exception("Operating system not supported!");
 #endif
}
```

Kuva 7. Kääntäjäohjeiden käyttö navigoinnin toteutuksessa.

Kolmesta esitetystä tavasta suosin pelkästään IoC-tekniikkaa. Vaikka kuvassa 7 esitetyn navigoinnin erot ovat pieniä ja sen voisi toteuttaa yhteen tiedostoon kääntäjäohjeita käyttämällä, mielestäni rajapintojen toteutus jokaiselle alustalle erikseen on parempi tapa. Toistetun koodin määrä on kuitenkin pieni haitta verrattuna siihen, että kääntäjäohjeiden käyttäminen tekee koodista vaikeaselkoista.

4 Asiakkuudenhallintasovelluksen toteuttaminen

Insinöörityössä toteutettiin sovellus, joka integroitiin kahteen eri asiakkuudenhallintajärjestelmään. Sovelluksen kohdeympäristöt olivat Windows Runtime- ja Windows Phone 8 -sovelluskehitysalustat. Portable Class Libraryä hyödynnettiin sovelluksen logiikan, tietomallin ja web-palvelurajapintojen toteutuksessa. Projekti toteutettiin elo-marraskuussa, ja sen aikajana on kuvattu liitteessä 1. Tässä luvussa esitellään projektisuunnitelman lisäksi sovelluksen suunnittelu ja toteutus sekä tulokset. Koska insinöörityössä toteutettu sovellus on riippuvainen asiakkuudenhallintajärjestelmästä, seuraa-

vassa luvussa esitellään asiakkuudenhallinta ja asiakkuudenhallintajärjestelmien tarkoitus.

Asiakkuudenhallinta

Asiakkuudenhallinnalla tarkoitetaan yrityksen asiakassuhteiden ylläpitoa, ja siihen kuuluvat myynti, markkinointi ja asiakaspalvelu. Asiakkuudenhallintajärjestelmä on asiakkuudenhallinnan tekninen osa. Siihen kerätään tietoa muun muassa yritysten asiakkaista, ja se tukee päätöksenteossa. Tähän insinöörityöhön liittyy kahden eri toimittajan tuotteet, jotka ovat Microsoftin Dynamics CRM ja Salesforce.com-nimisen yrityksen asiakkuudenhallintajärjestelmät. [1; 24.]

Insinöörityössä keskitytään tiedon keräämiseen ja tarkasteluun asiakkaiden ja mahdollisuuksien (business opportunity) osalta. Asiakkaat ovat yrityksen ulkopuolisen sidosryhmän osa, joihin yrityksen liiketoiminta kohdistetaan. Nykyisten asiakassuhteiden ylläpito ja uusien asiakkaiden hankinta ovat tärkeässä asemassa liiketoiminnan ylläpitämisessä. Mahdollisuudet ovat tunnistettuja tilaisuuksia saada myytyä yrityksen tuote tai palvelu asiakkaalle. Esimerkiksi asiakas ilmoittaa haluavansa päivittää tietojärjestelmiään ja aloittaa keskustelun yrityksen edustajan kanssa sen toteutumisesta. Yrityksen edustaja syöttää uuden mahdollisuuden asiakkuudenhallintajärjestelmään. Myöhemmin ilmenee, että päätöksenteko viivästyy, koska budjettia ei ole lyöty vielä lukkoon. Yrityksen edustaja päivittää mahdollisuuden tietoja. Lopulta ilmenee, että tietojärjestelmän uudistus ei mahtunut budjettiin, minkä seurauksena mahdollisuus suljetaan. Mahdollisuus saattaa siis pysyä avoimena hyvin pitkään, ennen kuin se suljetaan. Asiakkuudenhallintajärjestelmässä voi ylläpitää ja seurata mahdollisuuksia niiden elinkaaren ajan.

4.1 Projektisuunnitelma

Asiakkuudenhallintajärjestelmään kerätään tietoja asiakkaista ja mahdollisuuksista. Dynamics CRM- ja Salesforce-järjestelmät tarjoavat molemmat web-käyttöliittymät tietojen tarkasteluun ja keräämiseen. Insinöörityössä toteutettavan sovelluksen ei ole tarkoitus syrjäyttää toimivia järjestelmiä vaan parantaa käyttökokemusta mobiililaitteilla. Sovelluksen käyttäjät ovat henkilöitä, jotka ovat usein muualla kuin toimistossa. He voivat olla tilanteessa, jossa on mahdotonta käyttää kannettavaa tietokonetta, tai he

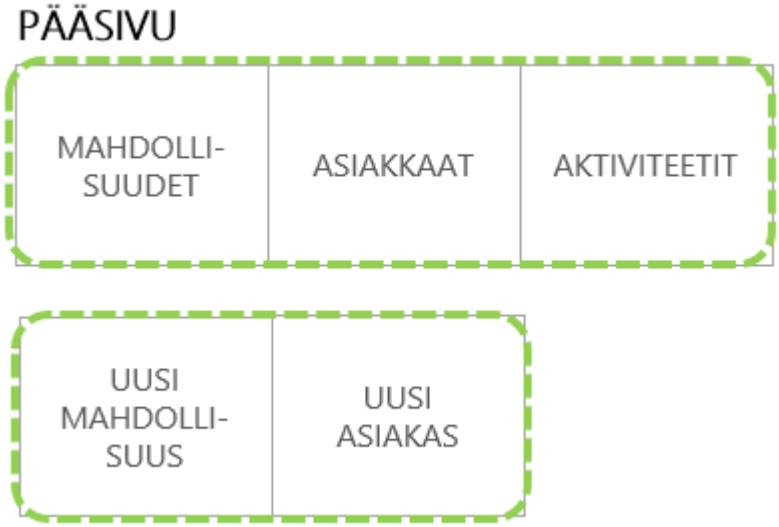
haluavat toimia nopeasti ja suosivat matkapuhelinta tai tablettia. Sovelluksen tarkoituksena on olla aina käytettävissä oleva, nopea ja sulava käyttöliittymä käyttäjille, jotka haluavat tarkastella tai syöttää uutta tietoa tien päällä.

Insinöörityön tilaajalle sovellusprototyypin ensisijainen käyttötarkoitus on toimia pohjana sisäiseen käyttöön tulevalle mobiilille asiakkuudenhallintasovellukselle. Sovelluksen loppukäyttäjät ovat insinöörityön tilaajana toimivan yrityksen johtoryhmä ja myynnin parissa toimiva henkilöstö. Toinen prototyypin käyttötarkoitus on toimia demosovelluksena, jolla myyntihenkilöstö voi osoittaa asiakkaille mobiilisovellusten edun ja käytettävyyden yrityssovelluksissa. Kolmas käyttötarkoitus, joka nousi esiin kesken insinöörityön toteuttamisen, on yhdistää asiakkuudenhallintajärjestelmä insinöörityön tilaajan ja asiakasyritysten liiketoimintaprosesseihin. Kolmannen käyttötarkoituksen myötä sovellusprototyyppi voisi toimia pohjana asiakkaille räätälöitävissä sovelluksissa.

Vaatimukset, laajuus ja riskit

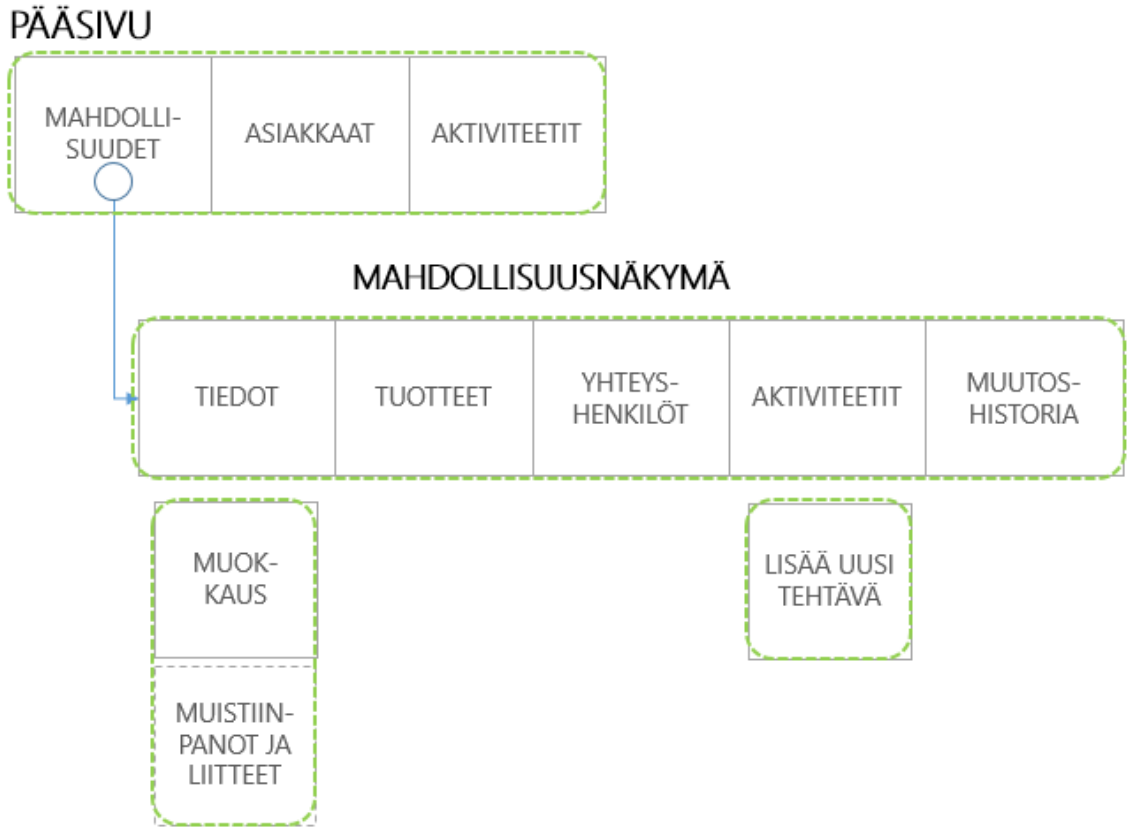
Sovelluksen tulee toimia Windows RT -käyttöjärjestelmässä, mutta koodin tulisi olla uudelleen käytettävää, koska sama sovellus halutaan toteuttaa Windows Phone 8 -käyttöjärjestelmälle mahdollisimman pienellä vaivalla. Sovelluksen tulee toimia samalla tapaa riippumatta siitä, käytetäänkö Dynamics CRM- vai Salesforce-asiakkuudenhallintajärjestelmää. Koska kyseessä on prototyyppi ja sen on toivottu valmistuvan mahdollisimman nopeasti, projektiin ei ole sisällytetty tavanomaista laadunhallintaa ja yksikkötestaus on jätetty pois. Projektin idea ja toiminnalliset vaatimukset pohjautuvat asiakkaalle toteutettuun projektisuunnitelmaan, jossa alustana on pelkästään Windows Phone 8.

Sovelluksen vaatimukset pohjautuvat asiakasprojektissa toteutettavaan vastaavaan sovellukseen. Liitteessä 2 on kuvattu kaikki tälle sovellukselle toivotut ominaisuudet, näkymät ja toiminnalliset vaatimukset. Niistä on insinöörityön laajuuteen valittu vain osa, ja ne on merkitty vihreällä katkoviivalla. Insinöörityössä toteutettavan sovelluksen näkymiin kuuluvat siis pääsivu, jossa näytetään käyttäjän mahdollisuudet, asiakkaat ja aktiviteetit (kuvio 15). Aktiviteetti voi olla esimerkiksi tehtävä, joka pitää suorittaa mahdollisuuden edistämiseksi. Pääsivulta voi myös lisätä uusia mahdollisuuksia ja asiakkaita.



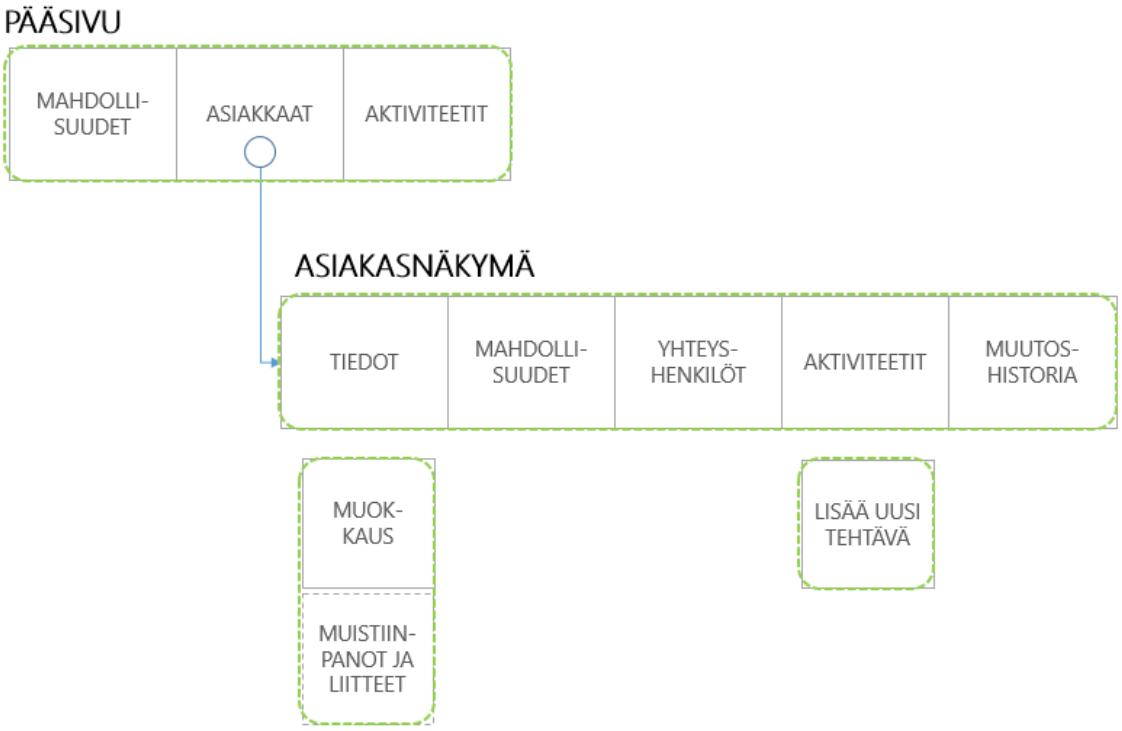
Kuvio 15. Pääsivulla näytettävät tiedot ja toiminnot.

Pääsivulta voi siirtyä mahdollisuusnäkymään, jossa esitetään valittuun mahdollisuuteen liittyviä tietoja, kuten mahdollisuuden arvo, jos se toteutuu. Lisäksi näkymässä näytetään mahdollisuuteen liittyvät tuotteet, yhteyshenkilöt, aktiviteetit, muutoshistoria ja muistiinpanot. Mahdollisuusnäkymässä pystyy myös muokkaamaan mahdollisuuden tietoja ja lisäämään tehtäviä (kuvio 16).



Kuvio 16. Mahdollisuusnäkymän tiedot ja toiminnot.

Pääsivulta voi siirtyä myös asiakasnäkymään, jossa esitetään valittuun asiakkaaseen liittyviä tietoja, kuten yhteystiedot. Lisäksi näkymässä näytetään asiakkaaseen liittyvät mahdollisuudet, yhteyshenkilöt, aktiviteetit, muutoshistoria ja muistiinpanot. Asiakasnäkymässä pystyy myös muokkaamaan asiakkaan tietoja ja lisäämään tehtäviä (kuvio 17).



Kuvio 17. Asiakasnäkymän tiedot ja toiminnot.

Projektin suurimmat riskit ovat vain yhden pääaikaisen kehittäjän käyttäminen ja se, että projektia toteutetaan asiakasprojektien välissä. Jos kehittäjä siirtyy uuteen asiakasprojektiin, ennen kuin sovellus on valmis, sen kehittäminen saattaa jäädä kesken.

Windows Store -sovellus ja Dynamics CRM -asiakkuudenhallintajärjestelmä asetettiin ensisijaisiksi, joten niihin liittyvä toteutus aloitetaan ensimmäisenä. Windows Phone -sovelluksen ja Salesforce-järjestelmän palvelurajapintojen kehittäminen tehdään, jos siihen jää aikaa. Tämä järjestys pohjautuu edellä mainittuihin riskeihin. Jos kehitys päättyy, kun kehittäjä siirtyy uuteen projektiin, sovelluksen tulisi toimia ainakin yhdellä käyttöjärjestelmällä ja asiakkuudenhallintajärjestelmällä.

Projekti- ja sidosryhmät

Projektiryhmään kuuluu kolme pysyvää jäsentä. Insinöörityön tekijä on vastuussa sovelluksen arkkitehtuurista ja kehittämisestä. Niin sanottu tuoteomistaja vastaa sovelluksen toiminnallisista vaatimuksista ja käyttöliittymäsuunnittelusta. Kolmas jäsen on projektipäällikkö, joka toimii lisäksi insinöörityön ohjaajana. Projektipäällikkö vastaa myös sovelluksen vaatimuksista ja päättää projektin laajuudesta. Lisäksi projektissa käytetään hyväksi muiden organisaatiossa työskentelevien kehittäjien osaamista ja apua

mahdollisuuksien mukaan. Projektin sidosryhmään kuuluvat aikaisemmin mainitut johtoryhmä ja myyntihenkilöstö. Näiden kahden lisäksi sovelluksesta on myös kiinnostunut yrityksen Dynamics CRM -projekteista vastaava ryhmä. Sisäisten sidosryhmien lisäksi sovellukselle voi myös olla kiinnostusta usealla eri asiakkaalla.

Tekniset vaatimukset

Sovelluksen kehittämiseen tarvitaan tietokone, jossa on Windows 8 -käyttöjärjestelmä. Tietokoneella pitää myös olla asennettuna Windows 8- ja Windows Phone 8 -ohjelmistokehityspaketit. Sovelluksen toteuttamiseen käytetään Visual Studio 2012 -kehitysympäristöä. Asiakkuudenhallintajärjestelmiä varten tarvitaan Dynamics CRM- ja Salesforce-ympäristöt. Dynamics CRM -järjestelmän asentaminen vaatii palvelimen tai erittäin tehokkaan tietokoneen, johon on asennettu virtuaaliympäristö. Aikaisemman kokemuksen mukaan asiakkuudenhallintajärjestelmän suorittaminen kehitysympäristössä heikentää koneen suorituskykyä, joten palvelinasennus on suositeltavaa. Dynamics CRM -järjestelmän asentaminen on pitkä operaatio, joten projektissa päädyttiin käyttämään pilvessä toimivaa kokeiluympäristöä. Salesforce-ympäristöksi valittiin pilvessä toimiva kehittäjille tarkoitettu kokeiluympäristö.

4.2 Sovelluksen suunnittelu ja toteutus

Sovelluksen suunnittelu alkoi tutustumisvaiheella, jonka aikana tutkittiin Windows Runtime- ja Windows Phone -ympäristöjä ja sovelluskehitystä niille. Lisäksi tutkittiin sovelluskehitystä Portable Class Libraryn avulla ja sen tuomia rajoituksia sekä Dynamics CRM:n toteuttajien julkaisemaa ohjelmistokehityspakettia (Dynamics CRM SDK). Se sisältää runsaasti toteutusesimerkkejä ja työkaluja, joiden avulla voi kehittää laajennuksia ja sovelluksia, jotka käyttävät Dynamics CRM -järjestelmää. Tietoa kerättiin enimmäkseen internetistä ja kollegoilta, joilla oli aikaisempaa kokemusta mainituista asioista. Lisäksi tutustuttiin sovelluksen vaatimuksiin ja projektin alustavaan laajuuteen. Tutustumisvaiheessa saaduilla tiedoilla lähdettiin pohjustamaan sovelluksen arkkitehtuuria ja projektirakennetta Visual Studioon.

Sovellusarkkitehtuuri noudattaa sivulla 20 kuviossa 11 esitettyä mallia, jota on myös hyödynnetty Visual Studion projektirakenteeseen. Aluksi projektirakenteen muodosti eri sovelluksen vastuualueiden jakaminen omiin aliprojekteihin. Näihin aliprojekteihin kuu-

luivat esimerkiksi palvelukerros, tietomalli ja bisneslogiikka ja ViewModel- ja View-kerrokset. Se todettiin kuitenkin liian monimutkaiseksi näin pienelle projektille ja sovellukselle. Projektirakenne kävi läpi vielä muutaman iteraation, joilla sitä pyrittiin yksinkertaistamaan, ja lopulta päädyttiin projektirakenteeseen, jossa Portable Class Libraryyn toteutetaan sovelluksen yleiskäyttöinen osa eli sovelluslogiikka, tietomalli ja palvelurajapinnat. Ne toimivat MVVM-arkkitehtuurimallin Model-kerroksena, jonka lisäksi ViewModel-kerroksen luokat toteutetaan myös Portable Class Libraryyn. Näin pyrittiin maksimoimaan koodin uudelleenkäyttö View-kerroksessa, jota varten Visual Studioon lisättiin Windows Store- ja Windows Phone -sovellusprojektit. Näin sovellusarkkitehtuuri noudattaa myös sivulla 26 kuviossa 14 esitettyä arkkitehtuuria. Windows Store- ja Windows Phone -sovellukset toimivat siis insinöörityössä toteutetun asiakkuudenhallintasovelluksen ilmentyminä eri alustoilla. Sovellusarkkitehtuurin ja Visual Studion projektirakenteen suunnittelun jälkeen sovelluksen toteutus alkoi sen yleiskäyttöisen osan, Dynamics CRM -palvelurajapinnan ja Windows Store -sovelluksen kehittämisellä.

Sovellusarkkitehtuurissa hyödynnettiin kerrosarkkitehtuurin mukaisesti jaettuja toimintoja, jotka määriteltiin rajapintoina Portable Class Libraryssa. Jaettuihin toimintoihin kuuluivat sovelluksessa välimuistin käyttö, tiedostonkäsittely, navigointi (navigointipalvelu) ja käyttäjän todentaminen, valtuutus sekä tallentaminen (käyttäjäpalvelut). Navigointipalvelu hoitaa sovelluksen sisäisen navigoinnin. Esimerkiksi silloin, kun käyttäjä haluaa katsoa tiettyä mahdollisuutta, sovellus navigoi mahdollisuusnäkymään. Käyttäjäpalvelulla hoidetaan käyttäjän todentamisen ja valtuutuksen asiakkuudenhallintajärjestelmän kautta ja tallennetaan kirjautuneen käyttäjän tiedot laitteelle. Myös asiakkuudenhallintajärjestelmien web-palveluiden käyttö toteutettiin rajapintoja vasten, koska sovelluksen vaatimuksiin kuului tuki sekä Dynamics CRM- että Salesforce-asiakkuudenhallintajärjestelmälle. Koodista saatiin yksinkertaisempaa käyttämällä rajapintaa kahden eri toteutuksen sijaan. Jaetuista toiminnoista alustariippuvaisia olivat navigointi, välimuistin käyttö, tiedostonkäsittely ja käyttäjän tallentaminen, jotka toteutettiin erikseen Windows Store- ja Windows Phone -sovelluksissa. Käyttäjäpalvelut pystyttiin toteuttamaan Portable Class Libraryssä, koska sillä ei ollut suoria alustariippuvaisuuksia.

Koska useat kriittiset toiminnot ohjelmoitiin rajapintoja vasten, niiden toteutukset pitää syöttää niistä riippuvaisille luokille. Esimerkiksi käyttäjäpalvelu on riippuvainen palvelurajapinnasta, jota tarvitaan käyttäjän todentamiseen ja valtuutukseen, sekä tiedostonkäsittelystä, koska käyttäjän tiedot tallennetaan laitteelle. Myös ViewModel-luokat ovat riippuvaisia jaetuista toiminnoista ja palvelurajapinnasta. Riippuvuuksien hallintaan

käytettiin luvussa 3.3 esitettyä IoC-ohjelmointitekniikkaa ja IoC container -luokkakirjastoa. Käytössä ollut IoC container -luokkakirjasto syöttää riippuvuudet automaattisesti luokan konstruktorille, jos alustettava luokka ja sen riippuvuudet on määritelty container-objektiin. Näin objektien alustuksesta saatiin yksinkertaisempaa tekemällä se container-objektin kautta.

Sovelluksen ViewModel-luokista ja tietomallista yritettiin tehdä mahdollisimman yksinkertaiset. Jokaiselle näkymälle toteutettiin oma ViewModel-luokka, joka täyttää näkymän tiedolla, kun siihen navigoidaan. ViewModel-luokat hakevat näkymään tarvittavan tiedon palvelurajapinnan kautta. Pääasiassa tarvittavat tiedot ovat asiakkaan tai mahdollisuuden tiedot. Sovelluksen tietomalli toteutettiin pitkälti Dynamics CRM:n tietomallin pohjalta. Sovelluksen yleiskäyttöisen osan eli sovelluslogiikan, tietomallin ja ViewModel-luokkien toteutus Portable Class Libraryssä sujui hyvin. Vain Dynamics CRM -palvelurajapinnan toteutuksessa kohdattiin ongelmia.

Dynamics CRM -palvelurajapinnat suunniteltiin aluksi toteutettavaksi Portable Class Libraryssä, ja tarkoituksena oli käyttää sen tarjoamaa REST-palvelurajapintaa (Representational State Transfer). Valitettavasti kehitystä sen eteen ehdittiin tehdä jonkin aikaa, kunnes huomattiin, että REST-palvelut olivat käytettävissä vain Dynamics CRM -portaalista tai sinne tehdyistä laajennuksista. Virheeltä olisi voitu välttyä, jos dokumentaatiota olisi luettu tarkemmin. Sen jälkeen päädyttiin käyttämään toista web-palvelurajapintaa, jota käytetään Dynamics CRM SDK:n luokkakirjaston avulla. SDK:n huono puoli oli, että siinä oli erilliset luokkakirjastot Windows Runtimelle ja Windows Phone 8:lle, joista kumpikaan ei ollut PCL-yhteensopiva. Tämä oli myös osasyy, miksi REST-palvelurajapinta valittiin alun perin. Toinen syy oli se, että Dynamics CRM SDK:n luokkakirjaston käyttö oli todettu hankalaksi aikaisemmassa projektissa.

Dynamics CRM:n palvelurajapinnan kehittäminen siirrettiin uuteen projektiin, joka oli tyypiltään Windows Runtime -luokkakirjastoprojekti, jotta se toimisi ainakin Windows Store -sovelluksen kanssa. Tämä toi kuitenkin esille kysymyksen, kuinka koodia voidaan käyttää uudelleen Windows Phone 8 -sovelluksessa. Käytännössä se on mahdollista, jos käytetty koodi on siitä osasta Windows Runtimen luokkakirjastoja, jotka ovat myös Windows Phone Runtime -luokkakirjastoissa (sivu 10 kuvio 6). Myös Dynamics CRM SDK:n luokkakirjastot Windows Store- ja Windows Phone -sovelluksia varten saattavat olla erilaisia. Siihen asti, kunnes Windows Phone 8 -sovelluksen kehitys al-

kaa, jää vielä avoimia kysymyksiä, eli toimivatko Dynamics CRM SDK:n luokkakirjastot samalla tavalla, ovatko samat luokat käytössä ja pystyykö koodia jakamaan.

Palvelurajapinnan toteuttamiseen otettiin mallia Dynamics CRM SDK:n esimerkkikoodeista. Palveluoperaatiot suoritettiin kokonaan luokkakirjastossa saatavilla olevan luokan kautta. Luokan metodit eivät ole asynkronisia, joten kehityksessä käytettiin hyväksi Dynamics CRM -tiimin julkaisemaa palveluoperaatioden async/await-toteutusta, joka ei tule ohjelmistokehityspaketin mukana. Tämä oli olennaista, koska web-palveluita käytettäessä vasteajat saattavat olla pitkiä, joten palveluoperaatiot pitää suorittaa asynkronisesti.

Palvelurajapinnan koodin luettavuuden ja laadun parantamiseksi päädyttiin käyttämään Dynamics CRM SDK:n vahvasti tyypitettyjä luokkia. Toinen vaihtoehto on käyttää näiden luokkien kantaluokkaa, mutta sen käyttäminen on koodin kannalta huonompi vaihtoehto, koska jäsenet kirjoitetaan tekstinä ja virheet huomataan vasta suorituksen aikana. Vahvasti tyypitettyjä luokkia on kuitenkin huomattava määrä. Jos kaikki luokat sijoitetaan yhteen tiedostoon, sen kooksi tulee yli 170 000 riviä. Määrän rajaamiseen käytettiin internetistä löydettyä esimerkkikoodia, jonka avulla pystyy suodattamaan pois ne luokat, joita ei tarvitse sovelluksessa. Koodi käyttää hyväksi SDK:n mukana tulevaa työkalua, joka luo vahvasti tyypitetyt luokat asiakkuudenhallintajärjestelmän web-palvelusta löytyvän WSDL-rajapintakuvauksen (Web Service Description Language) perusteella.

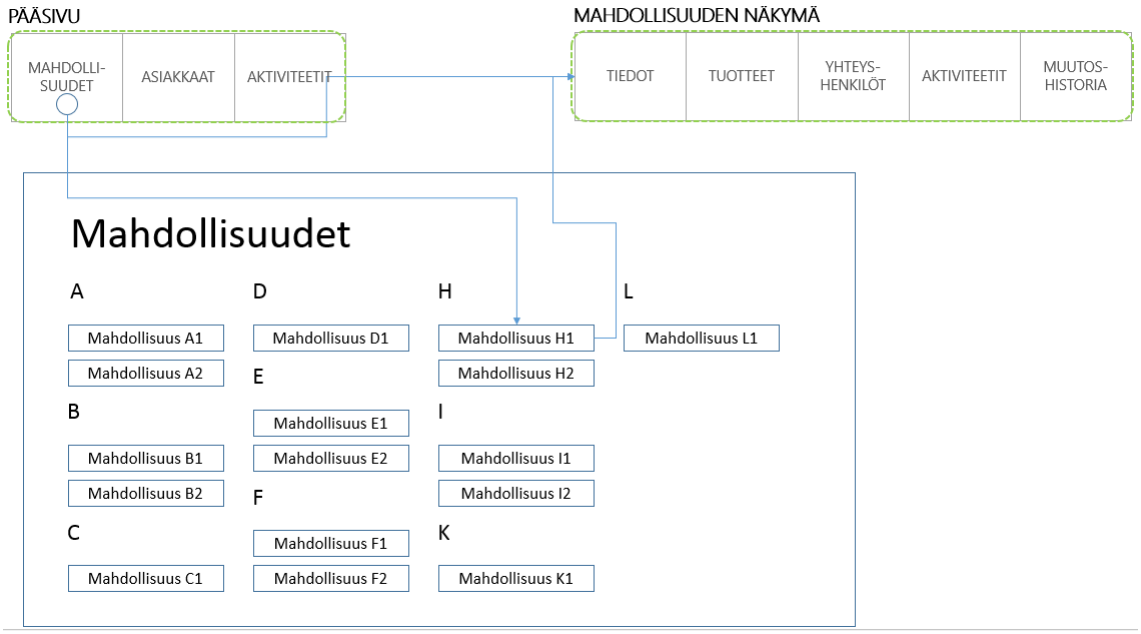
Kun ensimmäisiä tiedon hakuoperaatioita alettiin testata, palvelurajapinnan toiminnassa huomattiin outoa käytöstä, sillä palveluoperaatio saattoi palauttaa edellisen kyselyn tulokset. Tästä syntyi poikkeuksia, kun objekteja yritettiin muuntaa väärään luokkaan. Olin kohdannut aikaisemmassa projektissa vastaavia ongelmia, kun palveluoperaatio palautti vanhaa dataa päivityksen jälkeen. Silloin ongelman todettiin johtuvan välimuistin käytöstä, joka tapahtuu Dynamics CRM SDK:n luokkakirjaston sisällä. Välimuistia käyttävää ominaisuutta ei kuitenkaan saatu suljettua tällä kertaa, joten ongelma ratkaistiin luomalla jokaista kyselyä varten uusi instanssi luokkakirjaston luokasta, jolla palvelupyynnöt suoritetaan.

Vaatimuksiin kuului, että asiakkaiden ja mahdollisuuksien tietoa pitää pystyä päivittämään sovelluksen kautta. Päivitysoperaatioissa otettiin käyttöön vahvasti tyypitettyjen luokkien kantaluokka. Sen etu päivityksessä on, verrattuna vahvasti tyypitettyihin luok-

kiin, että sille pystyy määrittämään ne jäsenet, jotka lähetetään palvelupyynnössä järjestelmälle. Tämä on lähes pakollista päivitysoperaatiossa Dynamics CRM -webpalvelun kanssa, koska se päivittää tiedot palvelupyynnössä tulleen objektin jäsenten mukaisesti. Siksi tiedon päivittäminen Dynamics CRM -web-palvelulla vaatii objektien muutosten seurantaa, joka toteutettiin itse. Muutoksen seuranta toteutettiin niin, että objektit ovat itse tietoisia alkuperäisistä ja muuttuneista arvoistaan. Sen avulla päivityksessä lähetetään vain ne kentät, joita käyttäjä on muuttanut.

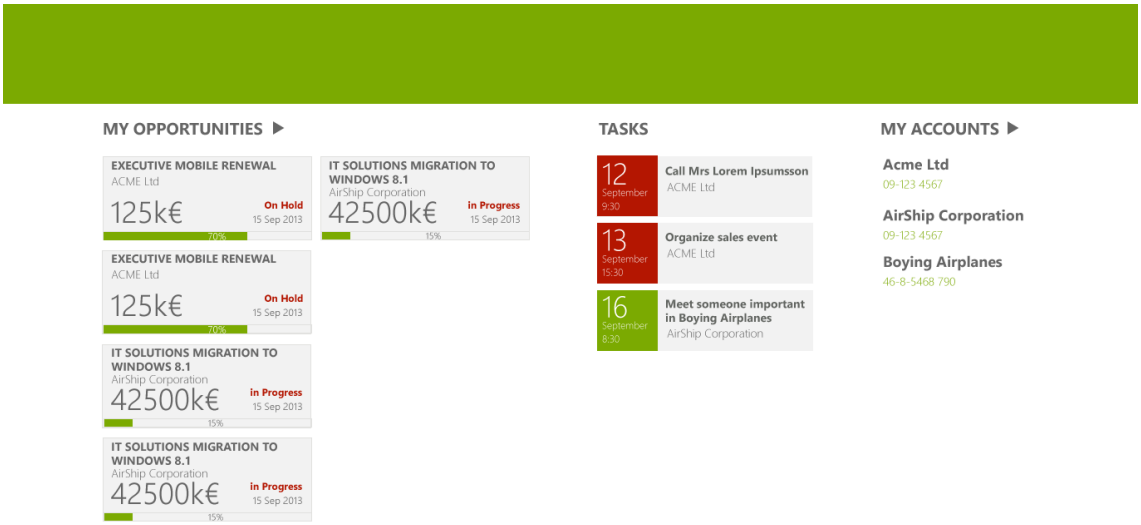
Myös uusien asiakkaiden, mahdollisuuksien ja tehtävien luominen kuului vaatimuksiin. Luontioperaatioiden toteutuksessa käytettiin hyväksi muutoksen seurantaa ja päivitysoperaatioita varten toteutettua koodia. Koska objektit ovat tietoisia alkuperäisistä arvoistaan, vain ne kentät, jotka käyttäjä on täyttänyt, lähetetään palvelupyynnössä järjestelmään.

Sovelluksen yleiskäyttöisen osan ja Dynamics CRM -palvelurajapinnan toteutus tehtiin yhtä aikaa Windows Store -sovelluksen kanssa. Windows Store -sovelluksen toteutus tehtiin noudattamalla liitteen 2 navigointimallia, jossa havainnollistetaan käyttäjälle mahdollinen siirtyminen sovelluksen näkymien välillä. Sovellusta pyrittiin kehittämään yksi näkymä kerrallaan. Liitteen 2 navigointimalliin lisättiin pääsivulta siirtyminen asiakkaiden tai mahdollisuuksien ryhmitettyyn näkymään. Kuviossa 18 on esitetty mahdollisuuksien ryhmitetty näkymä. Ryhmitetyssä näkymässä asiakkaat tai mahdollisuudet on ryhmitetty esimerkiksi nimen perusteella.



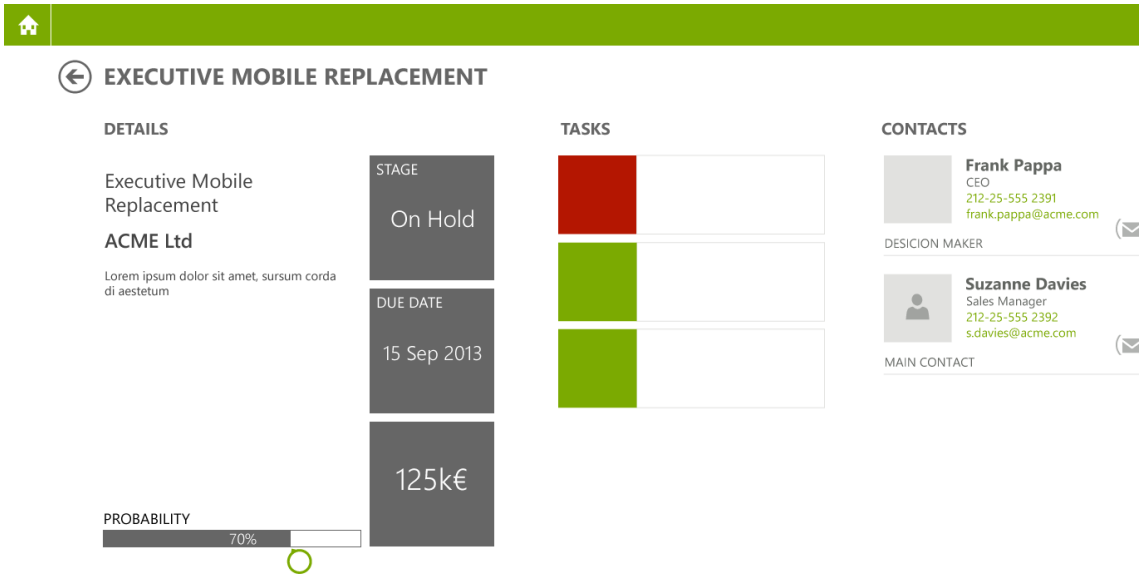
Kuvio 18. Ryhmitetty näkymä mahdollisuuksille.

Osa näkymistä toteutettiin luonnoksien perusteella. Niistä selviää sovelluksen graafinen ilme eli se, miltä käyttöliittymäkomponenttien tulee näyttää sovelluksessa. Luonnokset hahmottavat myös käytettäviä käyttöliittymäkomponentteja. Kuvassa 8 on esitetty pääsivun luonnos. Siitä voi huomata, että tarvittavat käyttöliittymäkomponentit ovat listoja.



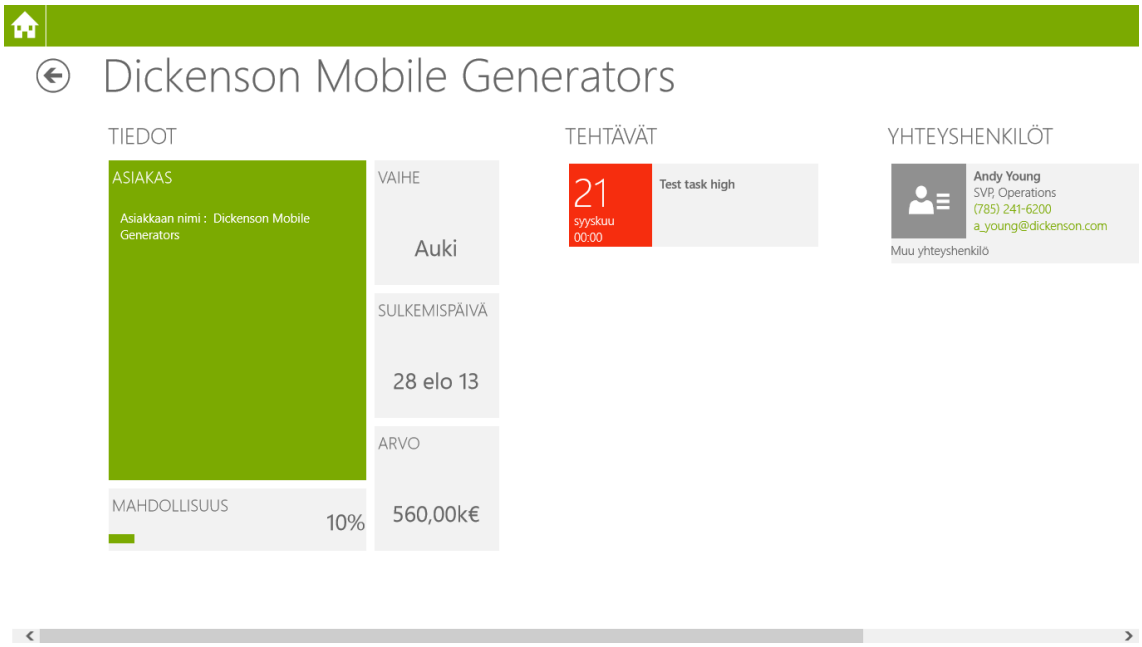
Kuva 8. Pääsivun viimeinen luonnos.

Mahdollisuusnäkymän luonnos on esitetty kuvassa 9. Näkymässä painotetaan mahdollisuuden tietoja. Asiakasnäkymälle ei tehty erikseen luonnosta, koska siinä näytettävät tiedot ovat lähes samat kuin mahdollisuusnäkymässä. Myöskään ryhmitetyille näkymille ei ollut luonnoksia.

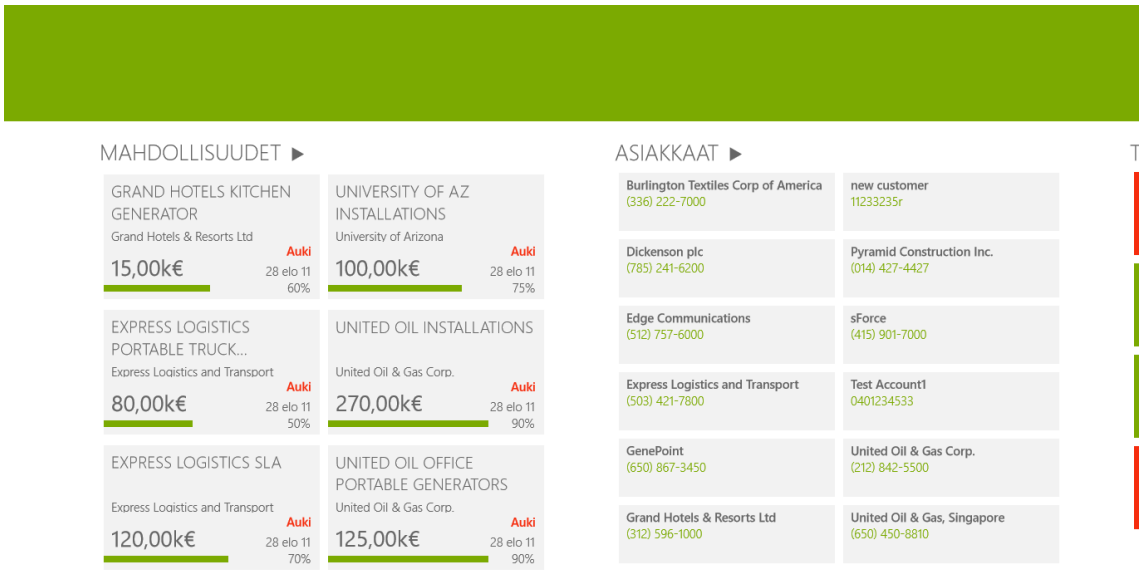


Kuva 9. Mahdollisuussivun viimeinen luonnos.

Windows Store -sovellukseen toteutettiin kaikki vaatimuksen mukaiset käyttötapaukset ja näkymät, joista kuvassa 10 on esitetty sovelluksen mahdollisuusnäkymä ja kuvassa 11 sovelluksen pääsivu. Lisäksi Windows Store -sovellukseen toteutettiin jako- ja hakutoiminnot. Jakotoiminnolla käyttäjä voi antaa mahdollisuuden luku- tai muutosoikeudet toiselle käyttäjälle. Hakutoiminnolla käyttäjä voi hakea asiakkaita ja mahdollisuuksia käyttämällä Windows RT:n hakua.



Kuva 10. Windows Store -sovelluksen mahdollisuusnäkymä.



Kuva 11. Windows Store -sovelluksen pääsivu.

Salesforce-asiakkuudenhallintajärjestelmään varten kehitettiin palvelurajapinnat siinä vaiheessa, kun Windows Store -sovellukseen oli toteutettu suurin osa vaatimuksen mukaisista käyttötapauksista. Tähän käytettiin taustajärjestelmänä Salesforcen kehittäjäversiota, joka toimii pilvipalveluna. Salesforcen tarjoamista palvelurajapinnoista valittiin REST, jota voi käyttää Portable Class Libraryssä Microsoftin Http-luokkakirjastolla.

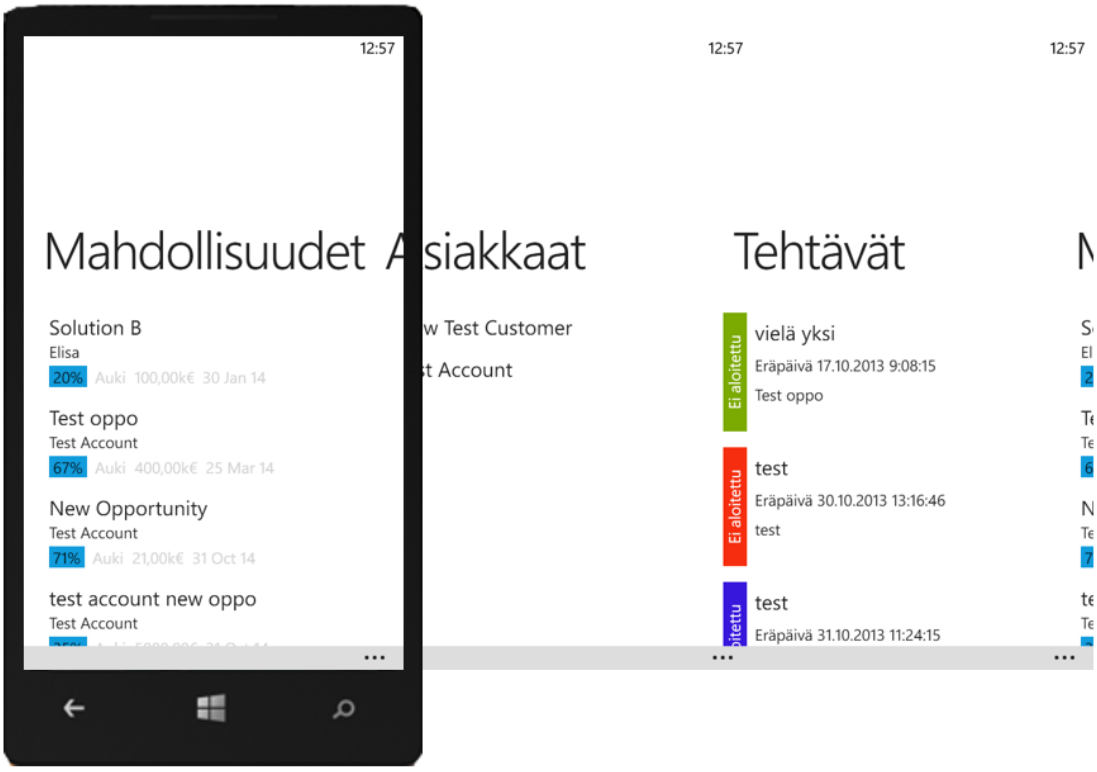
Suurin ongelma palvelurajapintojen käytössä oli ymmärtää kyselykieltä SOQL (Salesforce Object Query Language) ja saada kyselyt toimimaan. Salesforcen kyselykielen dokumentaatioista löytyi kuitenkin ohjeet kielen käyttöön, minkä jälkeen alun vaikeudet poistuivat. Tässä vaiheessa palveluoperaatioita määrittävässä rajapinnassa oli jo suurin osa vaatimuksien mukaisista käyttötapauksista, joten Salesforcen palvelurajapintojen toteutus sujui lopulta hyvin nopeasti verrattuna Dynamics CRM -palvelurajapintojen toteutukseen.

Myös Windows Phone 8 -sovelluksen kehitys alkoi siinä vaiheessa, kun Windows Store -sovellukseen oli toteutettu suurin osa vaatimuksista. Portable Class Libraryn koodi toimi odotetusti ilman ongelmia, mutta Dynamics CRM -palvelurajapinnat oli toteutettu Windows Runtime -luokkakirjastoprojektissa. Tässä vaiheessa selviää, toimiiko Windows Store -sovellusta varten kehitetty koodi Windows Phone -sovelluksessa. Dynamics CRM SDK:n Windows Phone 8 -luokkakirjaston käytössä ei havaittu ongelmia, ja koodin jakaminen käyttäen Windows Runtime- ja Windows Phone Runtime -luokkakirjaston yhteistä osaa toimi hyvin, vaikka olin ollut epävarma sen toimivuudesta.

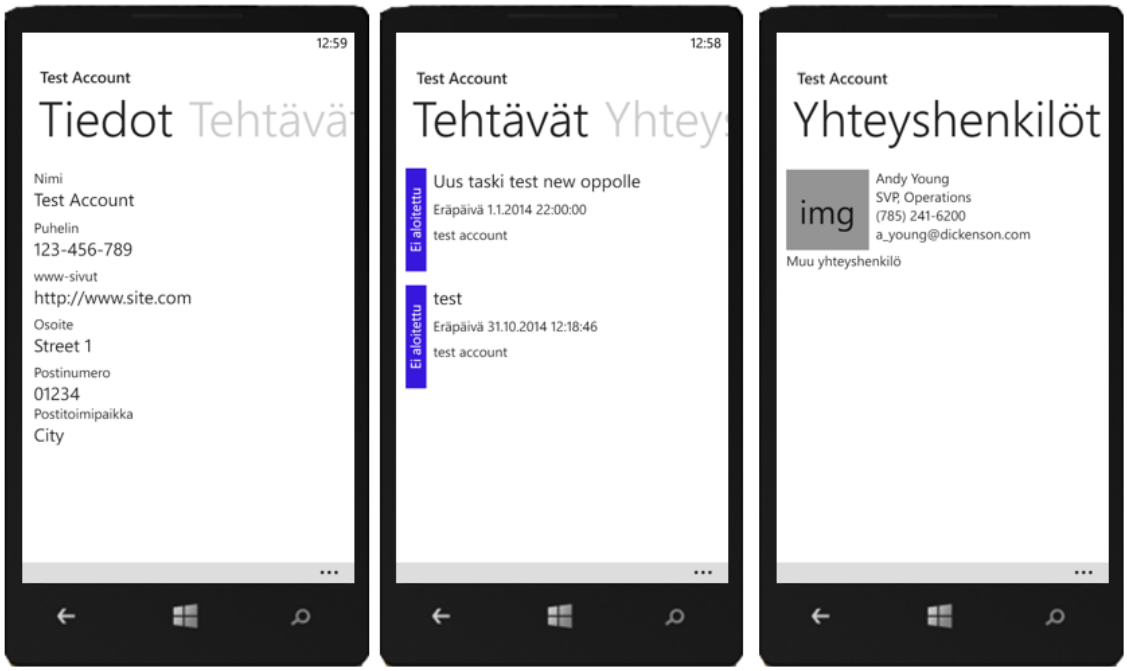
Isoin ongelma Windows Phone -sovelluksen kehityksessä muodostui navigoinnin toteutuksessa. Navigoinnin vastuu oli toteutettu ViewModel-luokissa, ja se pohjautui Windows Runtimen malliin. Windows Runtimen navigoinnissa parametriksi voi antaa objektin, mutta Windows Phonella voi antaa vain tekstiä. Ongelma ratkaistiin hyvin nopeasti huomioimalla molemmat mahdollisuudet ViewModel-luokissa. Tästä kaikesta muodostui kuitenkin turhaa koodia, koska ViewModel-luokissa piti tarkistaa parametrin tyyppi. Toinen pienempi ongelma tuli ryhmitettyjen näkymien ViewModel-luokkien kanssa. Windows Store -sovelluksessa ne toteutettiin tavalla, jota ei ollut järkevää käyttää Windows Phonella, jossa ryhmitetyn näkymän tekeminen oli yksinkertaisempaa. Ongelma ratkaistiin perimällä alkuperäiset ViewModel-luokat ja ohittamalla niiden metodit, jotka hakevat tiedon ja siirtävät ne näkymiin.

Windows Phone -sovelluksessa testattiin lähinnä koodin uudelleen käyttöä, joten sen käyttöliittymään panostettiin huomattavasti vähemmän. Siihen toteutettiin suurin osa vaadituista ominaisuuksista. Windows Phone -sovelluksella ei voi kuitenkaan lisätä uusia asiakkaita, eikä siihen toteutettu ylimääräisiä jako- ja hakutoimintoja, jotka toteutettiin Windows Store -sovellukseen. Windows Phone -sovelluksen graafiseen ilmee-

seen ei käytetty yhtä paljon työaikaa kuin Windows Store -sovelluksen. Kuvassa 12 esitetään Windows Phone -sovelluksen pääsivu ja kuvassa 13 asiakasnäkymä.



Kuva 12. Windows Phone -sovelluksen pääsivu.



Kuva 13. Windows Phone -sovelluksen asiakasnäkymä.

4.3 Tulokset, jatkokehitystarpeet ja kehitysideat

Oletin projektin olevan haastava, koska en ollut kehittänyt Windows Store- ja Windows Phone -sovelluksia aikaisemmin. Projektin alussa oli myös selvää, että Portable Class Libraryn onnistunut käyttö vaatii jokaisen valitun alustan tuntemista ja sovellusarkkitehtuuri pitää suunnitella tarkasti sen mukaan, mille alustoille sovellus julkaistaan. Sovellus saatiin kuitenkin toimimaan molemmilla alustoilla odotettua paremmin. Käytännössä suurin syy onnistumiseen oli rajapintoja vasten ohjelmointi. Esimerkiksi asiakkuudenhallintajärjestelmien web-palveluiden toteuttaminen rajapintoja vasten onnistui mielestäni hyvin. Palvelurajapintojen toteutus Salesforce-asiakkuudenhallintajärjestelmää varten aloitettiin noin puolessavälissä projektia, jolloin palvelurajapintaan oli määritelty jo huomattava määrä toteutettavia metodeita. Siitä huolimatta Salesforce-palvelut saatiin Dynamics CRM -palveluiden tasolle muutamassa päivässä.

Suurin osa projektiin käytetystä työajasta kului Dynamics CRM -palvelurajapinnan ja Windows Store -sovelluksen kehitykseen. Kun Windows Phone -sovellusta aloitettiin, sen perusrakenne näkymiä lukuun ottamatta oli valmiina parin työpäivän kuluessa. Näkymiin käytettiin myös muutamia päiviä, minkä jälkeen se oli ominaisuuksiltaan lähes samalla tasolla Windows Store -sovelluksen kanssa. Vastaavasti Windows Store -sovelluksen parissa käytettiin useita viikkoja kokonaisuuden aikaansaamiseksi. Windows Phone -sovelluksen kehittämisessä ei tarvinnut juurikaan miettiä ViewModel-luokkia, tietomallia ja palveluita, koska ne olivat jo lähes valmiita. Tämä suuri ero käytetyssä ajassa kuvastaa mielestäni Portable Class Libraryn hyötyjä alustariippumattoman .NET-sovelluksen tekemisessä. Aikahyödyn lisäksi se pakottaa sovellusarkkitehtuurin noudattamaan todettuja hyviä periaatteita.

Insinöörityössä toteutettua prototyyppiä ja sen lähdekoodia hyödynnetään jo toisessa prototyypissä, jonka tarkoituksena on osoittaa räätälöityjen asiakkuudenhallintajärjestelmäsovellusten hyötyä yritysten liiketoiminnassa ja sisäisissä prosesseissa. Uudessa demossa alkuperäiseen sovellukseen lisätään käyttötapauksia, jotka tukevat ja tehostavat myyntikokousten läpivientiä sekä kokousten välillä tapahtuvaa mahdollisuuksien seurantaa ja niiden edistämistä. Uusilla käyttötapauksilla sovellukseen tuodaan mukaan asiakkuudenhallintaan liittyvä päätöksenteko muun muassa seuraamalla mahdollisuuksia ja asettamalla tehtäviä niiden vastuuhenkilöille.

Sovelluksen jatkokehitystarpeet

En ole vielä täysin omaksunut kaikkia Microsoftin esittämiä designperiaatteita, koska tämä oli ensimmäinen kerta, kun kehitin Windows Store- ja Windows Phone -sovelluksia. Esimerkiksi Windows Phone -sovelluksen graafinen ilme on täysin toteuttamatta eikä sitä voi helposti yhdistää samaksi sovellukseksi. Tarkoituksena oli myös toteuttaa välimuistin käyttö niin, että palvelurajapinta noutaa tiedon välimuistista, jos se on tallennettu sinne äskettäin, ja muussa tapauksessa asiakashallintajärjestelmän web-palvelusta. Tätä ei kuitenkaan ehditty toteuttaa loppuun asti.

Yksikkötestien hyötyjä ja etua ei voi kiistää, jos sovellusta kehitetään eteenpäin. Järjestelmätestejä tulisi toteuttaa ainakin kyselyiden osalta. Sovelluksen toimivuus pitää voida tarkistaa järjestelmätesteillä, että voi todeta sen yhteensopivaksi asiakkaan järjestelmään. Sekä Dynamics CRM että Salesforce tukevat erittäin laajaa muokkausta. Järjestelmävalvojat voivat esimerkiksi sulkea joitakin tietomallin osia pois web-palveluiden rajapinnoista tai lisätä uusia objekteja tietomalliin. Lisäksi web-palveluiden ”takana", saattaa olla työnkulkuja, liitännäisiä ja muokkauksia, jotka voivat käyttäytyä odottamattomasti.

Yhtenä jatkokehitystarpeena pitää ottaa huomioon demon muunneltavuus asiakkaan graafiseen ilmeeseen. Tarve saattaa tulla hyvin yllättäen, ja aikatauluksi annetaan usein vain muutamia päiviä. Käyttöliittymien graafisen ilmeen muokkauksen pitäisi olla siis helppoa ja nopeaa. Esimerkiksi yhteen tiedostoon voitaisiin määritellä rajattu määrä käytettäviä värejä ja logot, jotka tekisivät sovelluksesta asiakkaan graafiseen ilmeeseen sopivan.

5 Yhteenveto

Microsoftin Windows-käyttöjärjestelmien eroavaisuuksien ja käytössä olevien eri .NET -sovelluskehysten myötä kehittäjät joutuvat toteuttamaan sovelluksen erikseen jokaiselle käyttöjärjestelmälle, jolle he haluavat julkaista sovelluksensa. Helpottaakseen kehittäjien työtä Microsoft tarjoaa lähes samanlaiset sovelluskehykset, ohjelmointimallit ja työkalut sovellusten kehittämiseen. Microsoft tarjoaa myös alustariippumattomien .NET-sovellusten kehittämiseen Portable Class Libraryn. Sen käytössä on kuitenkin huomioitava tiettyjä ohjelmointiin liittyviä periaatteita ja tekniikoita, joita tässä työssä esiteltiin.

Noudattamalla kerrosarkkitehtuuria ja SoC-periaatetta sovelluksen vastuualueet jaetaan kerroksiin. Eri kerroksiin suunnitelluilla komponenteilla on tietyt roolit ja ominaisuudet, joten komponentit eivät tule olemaan päällekkäisiä muiden kerrosten komponenttien kanssa. XAML-pohjaisissa sovelluksissa kerrosarkkitehtuurin toteuttamiseen voi hyödyntää MVVM-arkkitehtuurimallia, jonka avulla käyttöliittymä erotetaan sovelluksen logiikasta ja tietomallista. Lisäksi MVVM-arkkitehtuurimallissa hyödynnetään datan sidosta tiedon päivittämiseksi näkymiin automaattisesti.

Rajapintoja vasten ohjelmointi on Portable Class Libraryn kanssa hyvä ohjelmointitekniikka, koska se tekee sovelluksen rakenteesta modulaarisemman, jolloin parannetaan sovelluksen testattavuutta ja ylläpidettävyyttä. Se on myös osittain pakollista. Tietyt alustariippuvaiset toiminnot vaativat luokkia, joita ei voi käyttää Portable Class Libraryssä. Nämä luokat abstrahoidaan rajapintojen taakse, jotka toteutetaan jokaiselle alustalle erikseen.

Edellä mainituilla periaatteilla ja tekniikoilla insinöörityössä luotiin asiakkuudenhallintajärjestelmän mobiilikäyttöliittymän prototyyppi, joka toimii sekä Windows RT- että Windows Phone 8 -käyttöjärjestelmillä. Sen lisäksi sovellus pystyy käyttämään kahden eri asiakkuudenhallintajärjestelmän web-palveluita. Työn toteuttamiseen käytettiin paljon aikaisemmin opittua tietotaitoa, mutta yhtä lailla insinöörityötä varten hankittua tietotaitoa. Sovelluksen suunnittelu ja toteutus sujui projektisuunnitelmassa määritetyn aikataulun puitteissa. Insinöörityössä toteutetun sovelluksen lähdekoodia on jo hyödynnetty toisen prototyypin kehittämisessä, ja molemmat prototyypit ovat herättäneet paljon kiinnostusta yrityksen Dynamics CRM -projekteista vastaavan ryhmän sisällä.

Portable Class Libraryn avulla pystytään säästämään sekä aikaa että vaivaa, kun kehitetään .NET-sovelluksia usealle eri alustalle. Portable Class Library kuitenkin edellyttää tiettyjen .NET-sovelluskehikkojen ja ohjelmointimallien käyttämistä. Nämä rajoitukset voidaan myös käsittää hyödyllisenä piirteenä, koska ne johtavat modulaarisen sovellusarkkitehtuurin toteuttamiseen.

Lähteet

1 Asiakkuuden hallinta. Verkkodokumentti. Tietoyhteiskunnan kehittämiskeskus ry. <http://www.tieke.fi/display/ashal/Asiakkuuden+hallinta>. Luettu 1.11.2013

2 Asynchronous Programming with Async and Await. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx>. Luettu 5.10.2013.

3 Brumfield, B., Cox, G., Hill, D., Noyes, B., Pulelo, M. & Shifflett, K. 2012. Modular Application Development. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40).aspx>. Päivitetty 28.8.2012. Luettu 1.9.2013.

4 Bugnion, Laurent. 2013. Messenger and View Services in MVVM. Verkkodokumentti. <http://msdn.microsoft.com/en-us/magazine/jj694937.aspx>. Julkaistu 3/2013. Luettu 19.10.2013.

5 Burns, Kyle. 2012. Beginning Windows 8 Application Development: XAML Edition. USA: Apress.

6 Byrne, A. & Rothaus, D. 2012. How to Leverage your Code across WP8 and Windows 8. Verkkodokumentti. <http://channel9.msdn.com/Events/Build/2012/3-043>. Katsottu 8.9.2013.

7 Chakrabarty, I. 2013. Exam Ref 70-484: Essentials of Developing Windows Store Apps Using C#. USA: Microsoft Press.

8 Common Language Runtime. 2012. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/8bs2ecf4.aspx>. Luettu 14.9.2013.

9 Cross-Platform Development with the .NET Framework. 2012. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/gg597391.aspx>. Luettu 19.9.2013.

10 Data Binding Overview. 2012. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/ms752347.aspx >. Luettu 14.10.2013.

11 Design library for Windows Phone. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/design/hh202915(v=vs.105).aspx>. Luettu 8.9.2013.

12 Devey, Ben. 2012. Getting started with Metro Style Apps: A Guide to the Windows Runtime. USA: O'Reilly Media.

13 Dijkstra, Edsger W. 1974. On the role of scientific thought. Verkkodokumentti. <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html>. Tarkistettu 2010. Luettu 31.8.2013.

14 Edwards, Brent. 2013. Writing a Cross-Platform Presentation Layer with MVVM. Verkkodokumentti. <http://msdn.microsoft.com/en-us/magazine/dn385706.aspx>. Julkaistu 3/2013. Luettu 20.10.2013.

15 Fowler, Martin. 2004. Patterns of Enterprise Application Architecture. 6th Printing. Addison-Wesley.

16 Getting started with Blend for Visual Studio 2012. 2012. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/vstudio/jj171012.aspx>. Luettu 5.10.2013.

17 Getting started with developing for Windows Phone. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402529(v=vs.105).aspx>. Luettu 8.9.2013.

18 Haack, Phil. Manage Project Libraries with NuGet. 2011. Verkkodokumentti. <http://msdn.microsoft.com/en-us/magazine/hh547106.aspx> Julkaistu 11/2011. Luettu 5.10.2013.

19 How to create your first app for Windows Phone. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402526(v=vs.105).aspx>. Luettu 8.9.2013.

20 How to implement property change notification. 2012. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/ms743695.aspx>. Luettu 19.10.2013.

21 How to register your phone for development. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff769508(v=vs.105).aspx>. Luettu 8.9.2013.

22 Hunt, A. & Thomas, D. 2008. The Pragmatic Programmer. 22nd Printing. USA: Addison-Wesley.

23 INotifyDataErrorInfo Interface. 2012. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/system.componentmodel.inotifydataerrorinfo.aspx>. Luettu 19.10.2013.

24 Kaskela, Lauri. 2005. CRM-sovellusratkaisun rakenne. Verkkodokumentti. <http://www.tieke.fi/display/ashal/CRM-sovellusratkaisun+rakenne>. Luettu 1.11.2013.

25 Kovacs, James. 2008. Tame Your Software Dependencies for More Flexible Apps. Verkkodokumentti. <http://msdn.microsoft.com/en-us/magazine/cc337885.aspx>. MSDN Magazine 4/2008. Luettu 1.9.2013.

26 Language Independence and Language-Independent Components. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207014(v=vs.105).aspx>. Luettu 5.10.2013.

27 Launching, resuming, and multitasking for Windows Phone. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/12a7a7h3.aspx>. Luettu 15.9.2013.

28 LeBlanc, Brandon. 2012. Announcing the Windows 8 Editions. Verkkodokumentti. <http://msdn.microsoft.com/en-us/magazine/jj651570.aspx>. Julkaistu 26.4.2012. Luettu 15.9.2013.

29 Likness, Jeremy. 2012. Windows Runtime Components in a .NET World. Verkkodokumentti. <http://blogs.windows.com/windows/b/bloggingwindows/archive/2012/04/16/announcing-the-windows-8-editions.aspx>. Luettu 29.9.2013.

30 Lovell, Martyn. 2011. Lap around the Windows Runtime. Verkkodokumentti. <http://channel9.msdn.com/Events/BUILD/BUILD2011/PLAT-874T>. Julkaistu 14.9.2011. Katsottu 15.9.2011.

31 Martin, Robert C. 1996. The Dependency Inversion Principle. Verkkodokumentti. <http://objectmentor.com/resources/articles/dip.pdf>. Julkaistu 5/1996. Luettu 1.9.2013.

32 McConnel, Steven. 2004. Code Complete. 2nd Edition. USA: Microsoft Press.

33 Meier, J.D., Hill, D., Homer, A., Taylor, J., Bansode, P., Wall, L. Boucher, R. & Bogawat, A. 2009. Microsoft® Application Architecture Guide, 2nd Edition. USA: Microsoft Press.

34 Microsoft design principles. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-US/library/windows/apps/hh781237>. Luettu 5.10.2013.

35 Microsoft Unveils Windows Phone 8. 2012. Verkkodokumentti. Microsoft. <http://www.microsoft.com/en-us/news/press/2012/oct12/10-29windowsphone8pr.aspx>. Luettu 8.9.2013.

36 Mitchell, R. J. 1990. Managing Complexity in Software Engineering. London: Peter Peregrinus.

37 Multiple Platform Support. 2013. Verkkodokumentti. Microsoft. <http://www.microsoft.com/net/multiple-platform-support>. Luettu 15.9.2013.

38 .NET Framework Class Library Overview. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/hfa3fa08.aspx>. Luettu 14.9.2013.

39 .NET for Windows Store apps overview. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windows/apps/br230302.aspx>. Luettu 29.9.2013.

40 .NET Technology Guide for Business Applications. 2013. USA: Microsoft Corporation.

41 Overview of the .NET Framework. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/zw4w595w.aspx>. Luettu 14.9.2013.

42 Plaisted, Daniel. 2012. How to Make Portable Class Libraries Work for You. Verkkodokumentti. <http://blogs.msdn.com/b/dsplaisted/archive/2012/08/27/how-to-make-portable-class-libraries-work-for-you.aspx>. Julkaistu 27.8.2012. Luettu 20.10.2013.

43 Programming Concepts. 2013. Verkkodokumentti. <http://msdn.microsoft.com/en-us/library/vstudio/dd460655.aspx>. Luettu 14.9.2013.

44 Raffaele, Garofalo. 2011. Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern. Sebastopol, California: O'Reilly Media.

45 Rahman, Mohammad. 2011. Expert C# 5.0: with the .NET 4.5 Framework. USA: Apress.

46 Reade, Chris. 1989. Elements of Functional Programming. USA: Pearson Education.

47 Smith, Josh. 2009. WPF Apps With The Model-View-ViewModel Design Pattern. Verkkodokumentti. <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx#id0090006>. Julkaistu 2/2009. Luettu 13.10.2013.

48 Snell, M. & Powers, L. 2012. Microsoft Visual Studio 2012 Unleashed. USA: Pearson Education.

49 The MVVM Pattern. 2012 Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/hh848246.aspx>. Luettu 13.10.2013.

50 UX patterns. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/library/windows/apps/hh770552>. Luettu 6.10.2013.

51 What's new in Windows Phone SDK 8.0. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj206940(v=vs.105).aspx>. Luettu 8.9.2013.

52 Windows Phone API Reference. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff626516(v=vs.105).aspx>. Luettu 8.9.2013.

53 Windows Phone Design. 2013. Verkkodokumentti. Microsoft. <http://developer.windowsphone.com/en-us/design/principles>. Luettu 8.9.2013.

54 Windows Phone .NET API. 2013. Verkkodokumentti. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207211(v=vs.105).aspx>. Luettu 8.9.2013.

55 Windows Phone platform compatibility. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj206947%28v=vs.105%29.aspx>. Luettu 8.9.2013.

56 Windows Phone Runtime API. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207212(v=vs.105).aspx>. Luettu 8.9.2013.

57 Windows Phone Toolkit. 2013. Verkkodokumentti. Microsoft. <http://phone.codeplex.com/>. Päivitetty 15.8.2013. Luettu 8.9.2013.

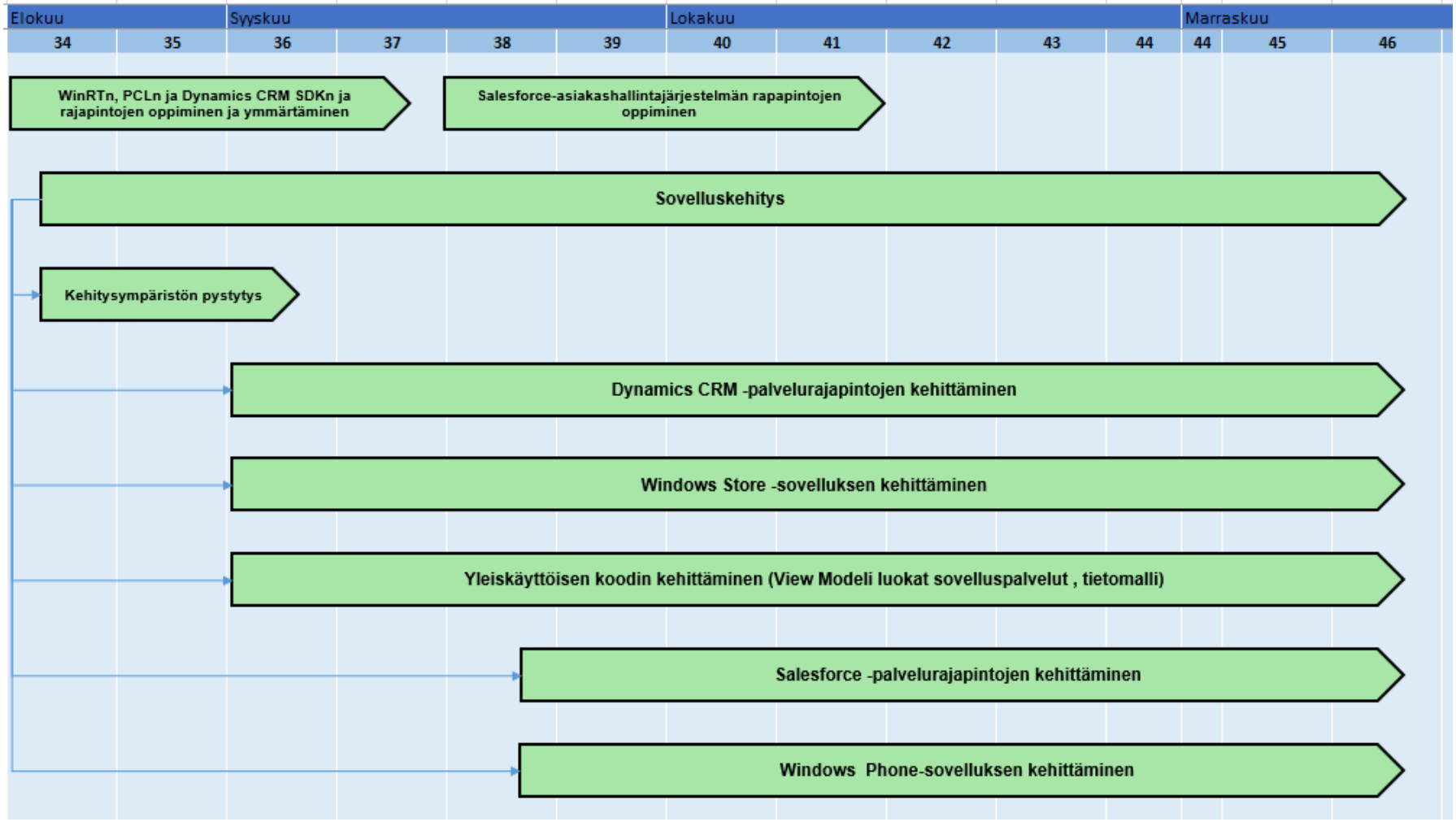
58 Windows 8 Arrives. 2013. Verkkodokumentti. Microsoft. <http://www.microsoft.com/en-us/news/Press/2012/Oct12/10-25Windows8GAPR.aspx>. Julkaistu 25.10.2012. Luettu 15.9.2013.

59 Windows 8 SDK. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-US/windows/apps/hh852363>. Luettu 5.10.2013.

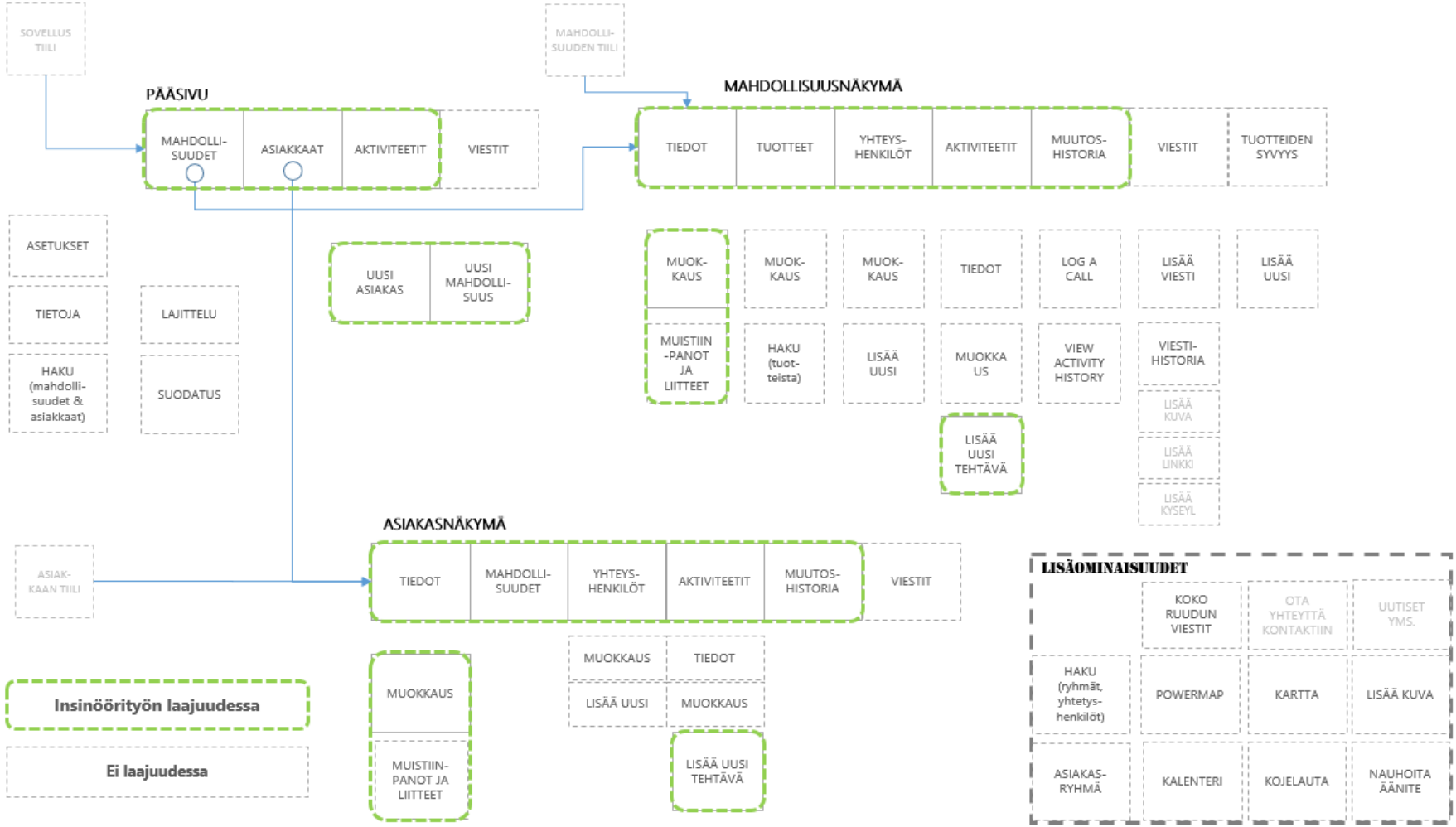
60 Win32 and COM API for Windows Phone 8. 2013. Verkkodokumentti. Microsoft. <http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207198(v=vs.105).aspx>. Luettu 8.9.2013.

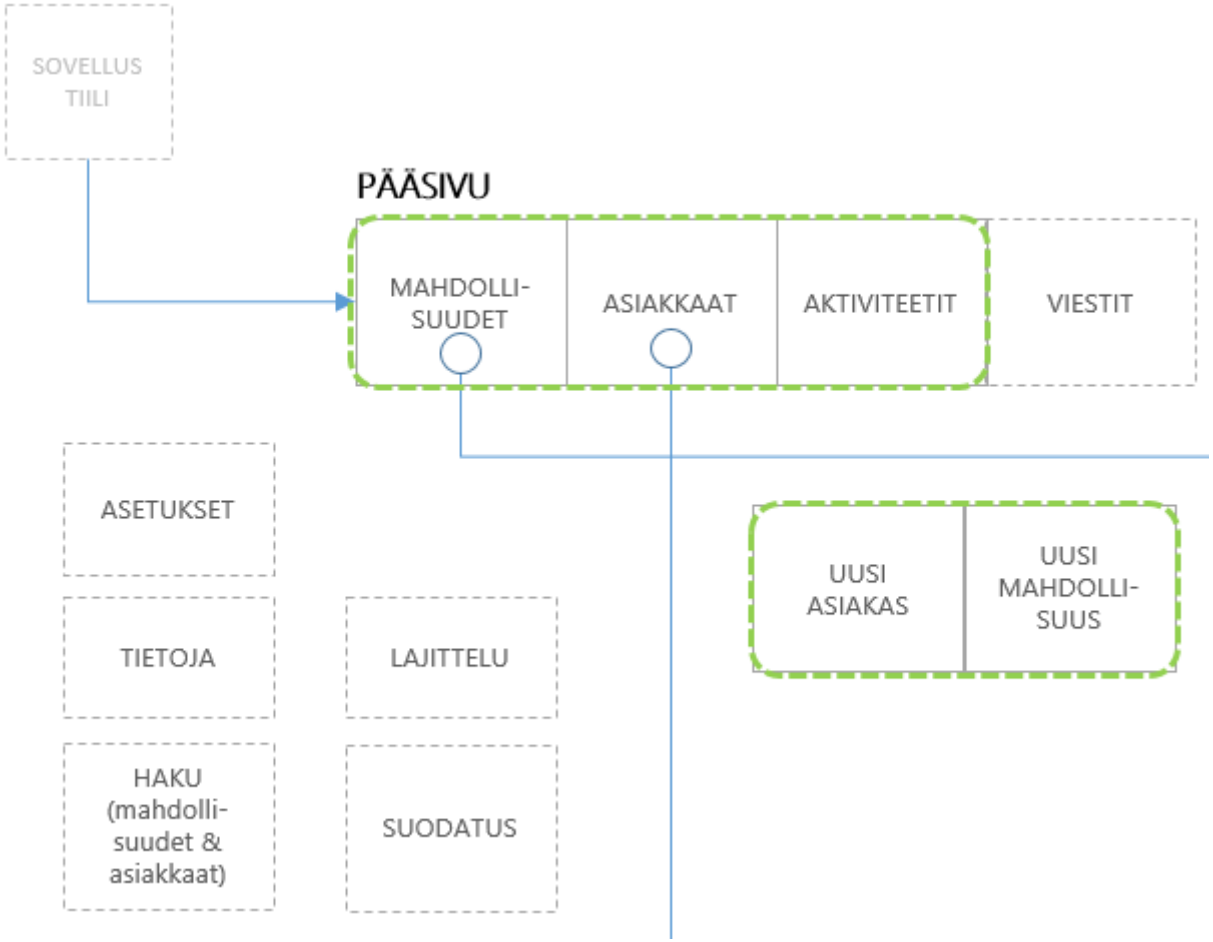
61 Gossman, John. 2005. How to Make Portable Class Libraries Work for You. Verkkodokumentti. <http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>. Julkaistu 8.10.2005. Luettu 20.10.2013.

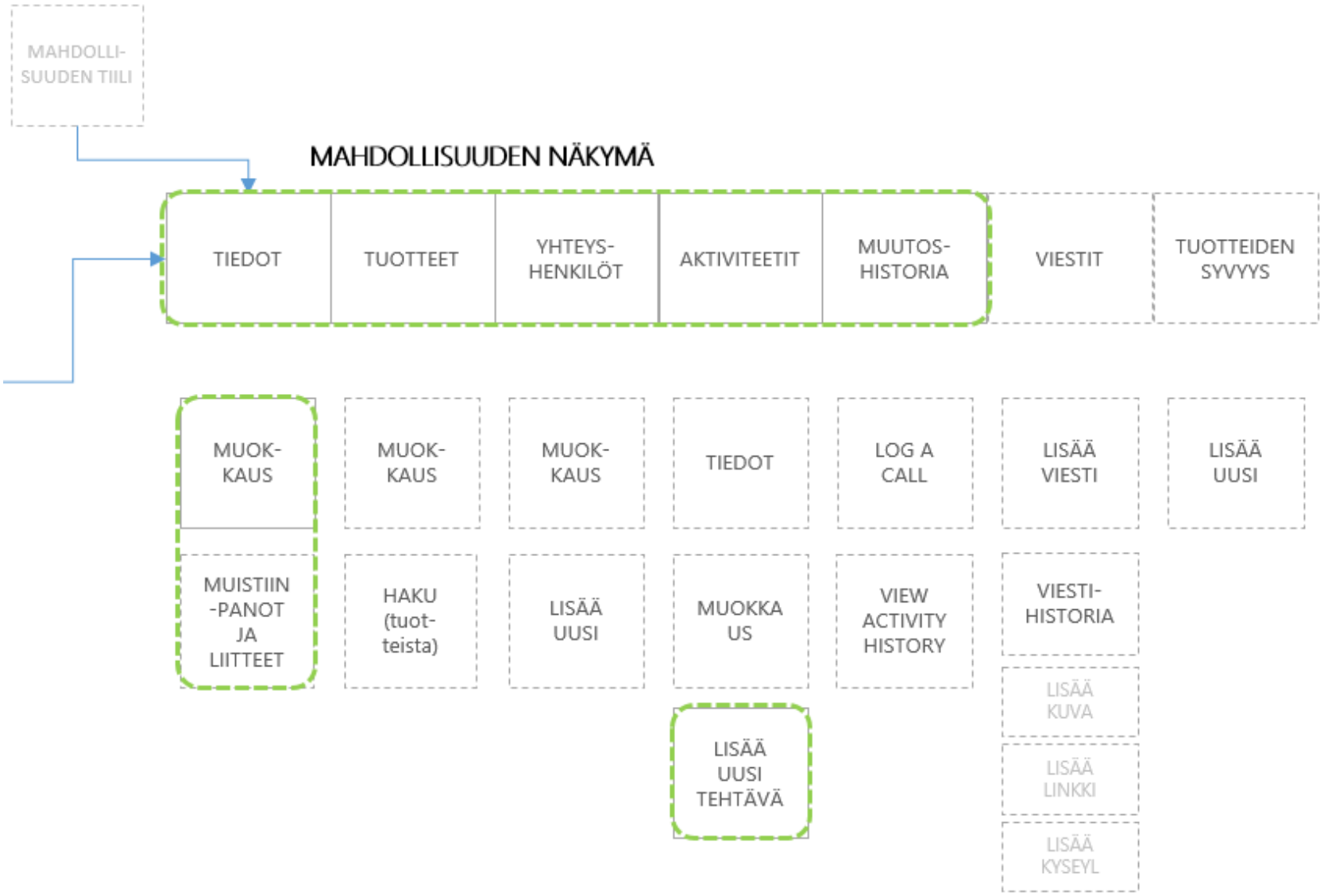
Projektin aikajana

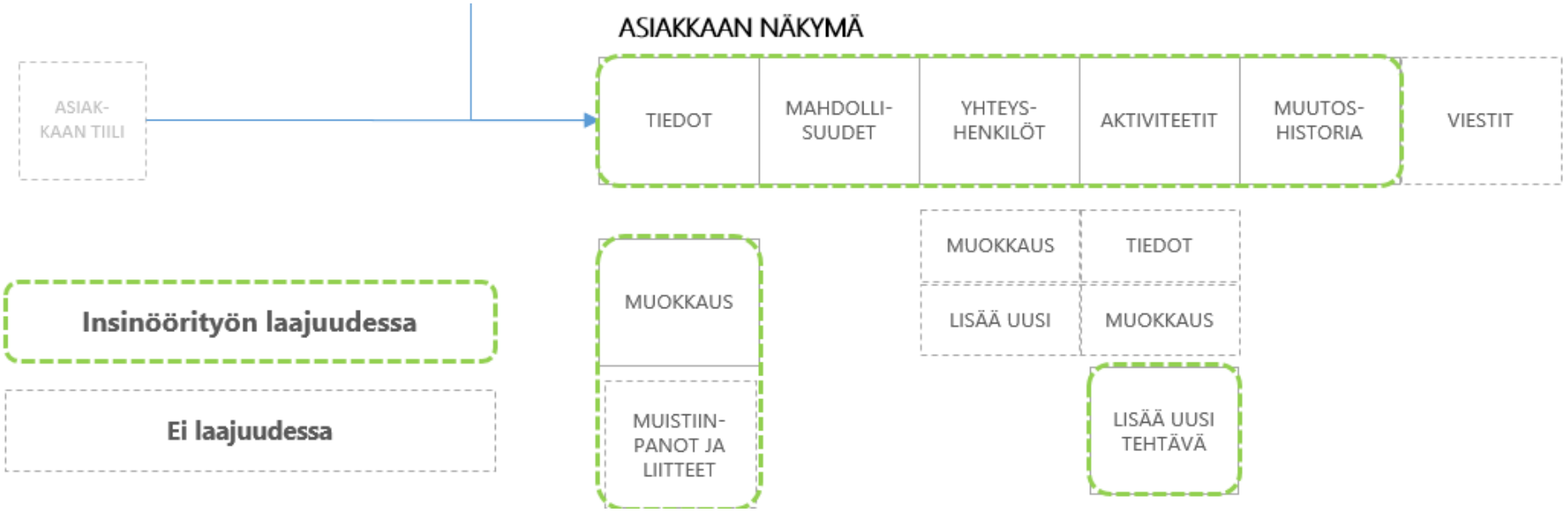


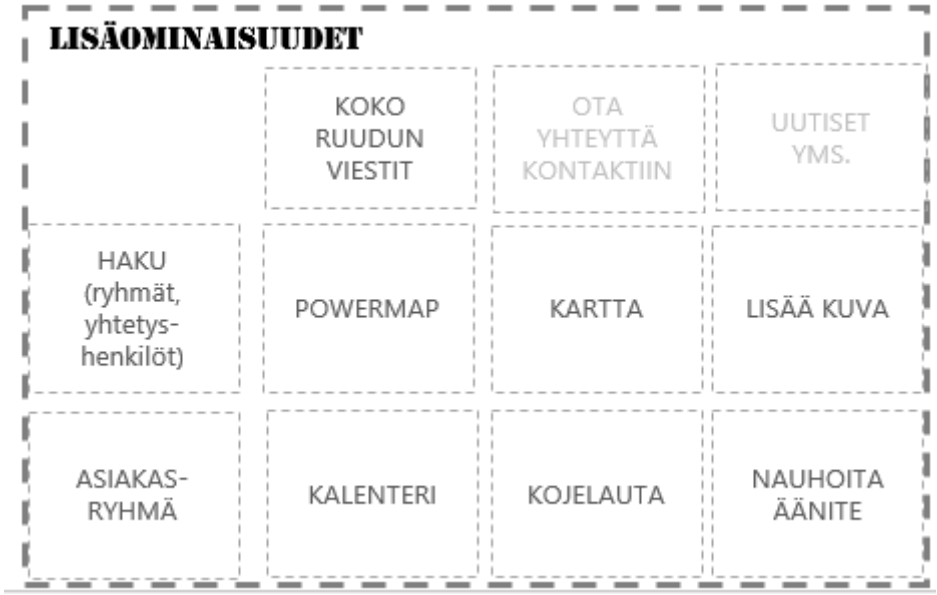
Näkymien, ominaisuuksien ja navigoinnin vaatimukset




SOVELLUS
TIILI
PÄÄSIVU
MAHDOLLI-
SUUDET
ASIAKKAAT
AKTIVITEETIT
VIESTIT
ASETUKSET
TIETOJA
HAKU
(mahdolli-
suudet &
asiakkaat)
LAJITTELU
SUODATUS
UUSI
ASIAKAS
UUSI
MAHDOLLI-
SUUS


MAHDOLLI-
SUUDEN TIILI
MAHDOLLISUUDEN NÄKYMÄ
TIEDOT
TUOTTEET
YHTEYS-
HENKILÖT
AKTIVITEETIT
MUUTOS-
HISTORIA
VIESTIT
TUOTTEIDEN
SYVYYS
MUOK-
KAUS
MUOK-
KAUS
MUOK-
KAUS
TIEDOT
LOG A
CALL
LISÄÄ
VIESTI
LISÄÄ
UUSI
MUISTIIN
-PANOT
JA
LIITTEET
HAKU
(tuot-
teista)
LISÄÄ
UUSI
MUOKKA
US
VIEW
ACTIVITY
HISTORY
VIESTI-
HISTORIA
LISÄÄ
KUVA
LISÄÄ
UUSI
TEHTÄVÄ
LISÄÄ
LINKKI
LISÄÄ
KYSEYL


ASIAKKAAN NÄKYMÄ
ASIAK-
KAAN TIILI
TIEDOT
MAHDOLLI-
SUUDET
YHTEYS-
HENKILÖT
AKTIVITEETIT
MUUTOS-
HISTORIA
VIESTIT
MUOKKAUS
MUISTIIN-
PANOT JA
LIITTEET
MUOKKAUS
TIEDOT
LISÄÄ UUSI
MUOKKAUS
LISÄÄ UUSI
TEHTÄVÄ
Insinöörityön laajuudessa
Ei laajuudessa


LISÄOMINAISUUDET
KOKO RUUDUN VIESTIT
OTA YHTEYTTÄ KONTAKTIIN
UUTISET YMS.
HAKU (ryhmät, yhtetys-henkilöt)
POWERMAP
KARTTA
LISÄÄ KUVA
ASIAKAS-RYHMÄ
KALENTERI
KOJELAUTA
NAUHOITA ÄÄNITE