LAPPEENRANNAN TEKNILLINEN YLIOPISTO
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Uudelleenkäytettävyys eräässä ohjelmistoprojektissa
Kandidaatintyö

Ohjaaja: Professori Kari Smolander

Lappeenrannassa 5. toukokuuta 2010,

Harri Eronen

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Harri Eronen

Uudelleenkäytettävyys eräässä ohjelmistoprojektissa

Kandidaatintyö

2010

31 sivua, 1 kuva, 3 taulukkoa ja 0 liitettä.

Tarkastaja: Professori Kari Smolander

Hakusanat: ohjelmisto, uudelleenkäytettävyys, sovitettavuus

Tässä työssä kuvataan erästä kaupallista ohjelmistoprojektia, jossa ohjelmistojen uudelleenkäyttöä tavoiteltiin komponenttipohjaisen uudelleenkäytön ja ohjelmistojen sovittamisen kautta. Projektissa toteutettiin matkapuhelinsovellus, jonka ydin eristettiin uudelleenkäytettäväksi ja sovitettavaksi komponentiksi. Ytimen sovitettavuus verifioitiin sovittamalla ydin toiselle ympäristölle ja edelleen toteuttamalla toimiva prototyyppisovellus sovitetun ytimen varaan. Ytimen sovittamisen sekä prototyyppisovelluksen toteuttamisen vaatima työmäärä oli huomattavasti pienempi kuin ytimen tekemiseen alkuperin käytetty työmäärä.

Työssä on osoitettu ohjelmistometriikoiden avulla, että merkittävä osa ohjelmiston toiminnallisuudesta saatiin uudelleenkäytettäväksi sovitettavan ytimen avulla. Lisäksi työssä on kuvattu millaisia kehitysprosessikäytäntöjä projektissa oli käytössä tukemassa uudelleenkäytettävyystavoitetta.

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology Management
The Degree Program of Information Technology

Harri Eronen

Reusability in a certain software project

Bachelor's thesis

2010

31 pages, 1 figure, 3 tables and 0 appendices.

Examiner: Professor Kari Smolander

Keywords: software, reusability, portability

This thesis describes a certain commercial software project which aimed at software reuse by utilizing component-based software reuse and software porting. The project implemented a mobile phone application, which core parts were encapsulated into reusable and portable components. Components portability was verified by porting those to another software platform and creating a functional prototype application top of those. Total effort needed to port the core components and implement prototype application was notably smaller than effort originally needed to implement the core components.

Thesis uses software metrics to show that notable amount of application logic was made reusable with portable core approach. Thesis also describes what kind of development process practices the project used for supporting the reusability goals.

SISÄLLYSLUETTELO

1 JOHDANTO

Tämä työ on kuvaus ohjelmistojen uudelleenkäyttöratkaisusta eräässä matkapuhelinympäristöön suunnatussa ohjelmistotuoteprojektissa. Ohjelmistojen uudelleenkäyttö on tapa kohentaa ohjelmistotuotannon kannattavuutta, ja täten yksi keino menestyä paremmin globaaleilla ohjelmistotuotemarkkinoilla. Lisäksi matkapuhelinmarkkinoiden jakautuminen useamman ohjelmistoalustan kesken kannustaa osaltaan panostamaan ohjelmistojen uudelleenkäytettävyyteen.

1.1 Työn tavoitteet

Tässä työssä keskitytään ohjelmistoprojektin toteutustason uudelleenkäytettävyyteen. Työssä käydään läpi sitä, kuinka uudelleenkäytettävyys miellettiin käsitellyssä ohjelmistoprojektissa:

- Millaisia ohjelmistojen uudelleenkäytettävyystavoitteita projektilla oli?
- Kuinka uudelleenkäytettävyys vaikutti ohjelmiston arkkitehtuuriin ja toteutukseen?
- Millaisia uudelleenkäytön mahdollistavia teknologiavalintoja projektissa tehtiin?
- Millaisia prosessikäytäntöjä projektissa oli käytössä tukemassa uudelleenkäyttötavoitteiden saavuttamista?

1.2 Työn rakenne

Työ jakautuu kuuteen lukuun, joilla on seuraavanlaiset sisällöt:

- Luku 1 esittelee kandidaatintyön aiheen, työn tavoitteet ja rajaukset.
- Luku 2 sisältää kirjallisuus- ja teoriakatsauksen kandidaatintyön aihealueeseen. Luvussa käydään lyhyesti läpi kirjallisuudessa kuvattuja ohjelmistojen uudelleenkäytön lähestymistapoja. Lisäksi luvussa käydään läpi muutamia mittareita, joita voidaan käyttää ohjelmistojen ominaisuuksien mittaamiseen ja esimerkiksi siis sen analysoimiseen kuinka hyvin uudelleenkäytettävyys projektissa toteutui.
- Luku 3 esittelee analysoitavan ohjelmistoprojektin ja sen taustat. Kohteena oleva ohjelmistoprojekti sekä ohjelmistotuote ovat kuvattu sillä tarkkuudella, kuin mitä yrityksen julkaisurajoitteet mahdollistavat.
- Luku 4 käsittelee sitä, kuinka ohjelmiston uudelleenkäytettävyys miellettiin kohteena olevassa ohjelmistoprojektissa. Luvussa kuvataan millaista uudelleenkäytettävyyslähestymistapaa projektissa käytettiin, ja kuinka uudelleenkäytettävyys näkyy ohjelmiston arkkitehtuurissa ja teknologiavalinnoissa. Lisäksi luvussa käydään läpi sitä, millaisia prosessikäytäntöjä projektissa käytettiin tukemaan uudelleenkäytettävyyden saavuttamista.
- Luku 5 analysoi projektin uudelleenkäytettävyystuloksia. Luvussa arvioidaan kuinka suuri osuus ohjelmistosta muodostui uudelleenkäytettäväksi, ja kuinka uudelleenkäytettävyys vaikutti projektin työmääriin.
- Luku 6 esittää yhteenvedon analysoidusta ohjelmistoprojektista sekä löydetyistä tuloksista.

2 OHJELMISTOJEN UUDELLEENKÄYTTÖ

Charles Kruegerin mukaan ohjelmistojen uudelleenkäytössä (*software reuse*) on perimmiltään kyse siitä, että uusi ohjelmisto pohjaa jollain muotoa johonkin jo olemassa olevaan ohjelmistoon. Hyödyntämällä jo olemassa olevia osia pyritään pienentämään uuden ohjelmiston toteuttamiseen tarvittavaa työmäärää ja aikaa. Lisäksi mikäli uudelleenkäyttö pohjaa laadukkaan ohjelmistolähteen käyttöön, periytyy osa laadusta samalla uuteen ohjelmistoon. [1]

Ohjelmistojen uudelleenkäyttökäsitteelle ei vaikuta olevan tiukkaa rajausta ohjelmistotuotannon tutkimuksen sisällä. Tarkasteltavasta lähteestä riippuen tiettyjen hyödyntämistapojen katsotaan joko kuuluvan ohjelmistojen uudelleenkäytön piiriin tai olevan sen ulkopuolella. Esimerkiksi Jeffrey Poulin esittää kirjassaan, että ohjelmistojen uudelleenkäyttöä olisivat vain tiukasti tietyn sovellusalueen ohjelmistojen välillä tapahtuva uudelleenkäyttö, sekä yrityksen sisällä tuotteiden välillä tapahtuva uudelleenkäyttö. Poulinin mukaan esimerkiksi seuraavia tapauksia ei yleisesti miellettä uudelleenkäytöksi:

- Käyttöjärjestelmäpalveluiden käyttö
- Korkeantason ohjelmointikielten käyttö
- Ohjelmistokehitystyökalujen käyttö
- Ohjelmistojen sovittaminen uuteen ympäristöön
- Sovelluskehittimien käyttö
- Yleishyödyllisten kirjastojen käyttö
- Muokattujen komponenttien käyttö
- Uusien ohjelmistoversioiden tuottaminen ylläpidon kautta

[2]

William Frakes ja Kyo Kang määrittelevät ohjelmistojen uudelleenkäytön puolestaan väljemmin toiminnaksi, jossa uusi ohjelmisto tehdään jo olemassa olevaa ohjelmistoa tai ongelma-alueeseen liittyvää tietämystä hyödyntäen. He myös määrittävät, että uudelleenkäytettävä vahvuus (*reusable asset*) on sellaista aihealueen osaamista tai

konkreettista ohjelmistotototeutusta, jota voidaan hyödyntää uuden ohjelmiston tekemisessä. Uudelleenkäytettävävahvuus käsitteelle käytetään myös rinnakkaistermiä uudelleenkäytettävä tuotos tai uudelleenkäytettävä artefakti (*reusable artefact*). Uudelleenkäytettävyys (*reusability*) on puolestaan ohjelmiston rakenteiden ominaisuus, joka kuvaa sitä kuinka helposti nuo rakenteet ovat uudelleenkäytettävissä jossain toisessa käyttötapauksessa. [3]

2.1 Merkittävimmät uudelleenkäytön lähestymistavat

Sajjan G. Shiva ja Lubna Abou Shala listaavat nykyhetken merkittävimmiksi ohjelmistojen uudelleenkäyttötavoiksi komponenttipohjaisen uudelleenkäytön (*component-based software reuse*), arkkitehtuuriratkaisujen uudelleenkäytön (*architecture-based software reuse*) sekä ohjelmistotuotelinjan (*software product line*). Sajjan ja Lubna huomauttavat, että nämä lähestymistavat eivät ole toisiaan poissulkevia, vaan käytännön ohjelmistonkehitysprojekteissa menetelmiä käytetään usein yhdessä. [4]

Näistä kolmesta lähestymistavasta komponenttipohjaisen uudelleenkäytön mittakaava on suppein. Siinä uusi ohjelmisto pyritään koostamaan yksittäisistä jo olemassa olevista ohjelmistokomponenteista. Käytettävät ohjelmistokomponentit ovat etukäteen suunniteltu ja toteutettu varta vasten komponenttipohjaista uudelleenkäyttöä silmälläpitäen. Arkkitehtuuripohjaisessa uudelleenkäytössä uudelleenkäytön mittakaava on puolestaan komponenttipohjaista uudelleenkäyttöä laajempi. Arkkitehtuuripohjaisessa uudelleenkäytössä pyritään hyödyntämään suurempia osakokonaisuuksia joko määrittelyinä tai kokonaisina toteutettuina alijärjestelminä.

Ohjelmistotuotelinja puolestaan edustaa laajinta ohjelmistojen uudelleenkäytön mittakaavaa. Ohjelmistotuotelinjassa toteutetaan tietylle sovellusalueelle joukko ominaisuuksiltaan erilaisia ohjelmistotuotteita siten, että erillisten tuotteiden arkkitehtuuri ja toteutus pohjaavat kuitenkin yhteen ja samaan tuotealustaan. Tuotealustankehitys (*domain engineering*) analysoi mitä yhteneväisyyksiä ja eroja

tuotelinjan eri tuotteilla tulee olemaan. Lisäksi tuotealustankehitys määrittelee ja toteuttaa analyysitulosten pohjalta tuotealustan, josta erilliset tuotteet sitten muodostetaan eriyttämisen (*variation*) avulla.

2.2 Ohjelmistojen sovitettavuus

James Mooney määrittelee ohjelmistojen sovittamisen (*software porting*) yhdeksi ohjelmistojen uudelleenkäytön muodoksi. Ohjelmiston sovittamisessa siirretään jo olemassa oleva ohjelmistotuote tietystä ohjelmistoympäristöstä (*software platform*) käyttöön johonkin toiseen ympäristöön. Sovittamisella pyritään samoihin tavoitteisiin kuin konventionaalisilla uudelleenkäytön lähestymistavoilla. Näitä tavoitteita ovat ohjelmistotuotannon tehokkuuden kasvattaminen ja kustannussäästöt. [5]

Ohjelmiston sovittaminen voi tapahtua kahden tyystin erillisen ohjelmistoympäristön välillä, tai se voi tapahtua ohjelmistoympäristön eri versioiden välillä. Kahden erillisen ohjelmistoympäristön tapauksessa puhutaan lähdeympäristöstä (*source platform*) ja kohdeympäristöstä (*target platform*). Saman ohjelmistoympäristön eri versioiden välillä tapahtuvassa siirtämisessä puhutaan puolestaan eteenpäinsovittamisesta (*forward porting*), mikäli siirtäminen tapahtuu vanhemmasta ympäristöversiosta uudempaan. Mikäli siirtäminen tapahtuu päinvastaiseen suuntaan, uudemmasta ympäristöversiosta vanhempaan, puhutaan taaksepäinsovittamisesta (*backward porting*). Ohjelmiston sovittaminen koostuu ohjelmiston siirtämisestä kohdeympäristöön (*transport*), sekä tarvittavasta ohjelmiston mukauttamisesta kohdeympäristön toiminnan mukaiseksi (*adapt*).

Ohjelmiston sovitettavuus (*portability*) kuvaa sitä, millaisia panostuksia ohjelmiston uuteen kohdeympäristöön sovittaminen vaatii. Mitä parempi sovitettavuus ohjelmistolla on, sitä pienemmällä työllä ja kustannuksilla sovittaminen saadaan tehtyä. Sovittaminen on mielekästä vain mikäli sovitustyön kustannukset jäävät

alhaisemmiksi, kuin uuden vastaavan ohjelmiston kehittäminen kohdealustalle vaatisi.

Ohjelmiston sovitettavuus voidaan jakaa binäärimuotoiseen sovitettavuuteen (*binary portability*) ja lähdekoodin sovitettavuuteen (*source portability*). Binäärimuotoinen sovitettavuus kuvaa tilannetta, jossa lähdeympäristössä suoritettavaan muotoon koostettu ohjelmisto voidaan ottaa sellaisenaan käyttöön kohdeympäristössä. Ohjelmiston koostaminen lukitsee suoritettavan ohjelman sisään monia riippuvuuksia ohjelman suoritusympäristöön. Nuo riippuvuudet asettavat ohjelmistojen binäärimuotoiselle sovitettavuudella tiukkoja reunaehtoja. Esimerkiksi lähde- ja kohdeympäristön prosessorien käskykannan, muistin hallinnan ja käyttöjärjestelmäpalveluiden tulee olla ehdottoman yhteensopivia, jotta binäärimuotoinen ohjelmiston sovitettavuus olisi mahdollista.

Lähdekoodin sovitettavuus puolestaan on tilanne, jossa ohjelmiston lähdekoodin pohjalta voidaan koostaa uusi suoritettavissa oleva ohjelmisto kohdeympäristöä varten. Lähdekoodipohjaisessa sovitettavuudessa reunaehdot eivät ole niin tiukkoja, sillä ohjelmiston lähdekoodia on mahdollista mukauttaa osana käännösprosessia vastaamaan kohdeympäristön ominaisuuksia. Toisaalta lähdekoodipohjainen sovittaminen vaatii enemmän vaiheita kuin binäärimuotoinen sovittaminen. Vähintäänkin lähdekoodin kääntäminen ja suoritettavaan ohjelman koostaminen tarvitsee tehdä. Mahdollisia muita tehtäviä ovat esimerkiksi lähdekoodin muokkaaminen tai uusien osien toteuttaminen, jotta ohjelmisto saadaan mukautettua kohdealustaa vastaavaksi.

Hyvä sovitettavuus mahdollistaa säästöjen saamisen ohjelmistotuotteen elinkaarikustannuksissa. Ohjelmistoihin kohdistuu erilaisia muutosvaatimuksia niiden eliniän aikana ja näiden muutosten toteuttaminen vaatii taloudellisia panostuksia. Ohjelmistoja saatetaan joutua esimerkiksi sovittamaan ohjelmistoympäristöstä toiseen, tai mahdollisesti saman ympäristön uudempiin versioihin. Mikäli sovitettavuus on huomioitu ohjelmiston alkuperäisissä

suunnitelmissa ja rakenteissa, on tuo sovittaminen helpompaa ja siis tehtävissä pienemmillä kustannuksilla.

Sovitettavuuden huomioiminen kasvattaa ohjelmiston kehityskustannuksia, kun taas mahdollisista kustannussäästöistä päästään nauttimaan vasta pitkällä aikavälillä. Käytännössä suurimmalla osalla ohjelmista on pitkä elinikä, jopa pidempi kuin ohjelmistoja alun perin suunniteltaessa arvioidaan. Ohjelmistojen suunnittelun painopisteen tulisikin täten olla elinkaarikustannusten minimoimisessa eikä pelkästään lyhtynäköisessä kehityskustannusten minimoinnissa. David Parnas sanookin, että yksi mittari yrityksen ohjelmistotuotannon prosessien kehittyneisyydestä on se, kuinka hyvin yritys huomioi ohjelmistojen suunnittelussa niiden pitkäaikaisen elinkaaren, eikä vain keskity pelkästään sen hetkiseen versioon. [6]

2.3 Ohjelmiston ominaisuuksien mittaaminen

Ohjelmistojen rakennetta, ominaisuuksia sekä ohjelmistoprosesseja voidaan mitata lukuisilla eri mittareilla. Mittareiden avulla pyritään keräämään esimerkiksi tietoa jo läpi viedyistä ohjelmistoprojekteista, jotta saadaan lähtöarvoja seuraavien ohjelmistoprojektien työmäärien ja kustannusten arviointia varten. Toisaalta tuotettuja ohjelmistoja analysoidaan ja mitataan, jotta saadaan selville esimerkiksi niiden kompleksisuus ja koko ominaisuuksia. Näitä tuloksia voidaan käyttää esimerkiksi arvioitaessa ohjelmistojen sisäistä laatua ja sisältämien virheiden lukumäärää ja sitä kautta edelleen mahdollista ylläpidon tarvetta ja kustannuksia. Tässä työssä on mittareiden avulla analysoitu sitä, kuinka suuri osuus toteutetun ohjelmiston sisällöstä saatiin uudelleenkäytettäväksi, ja siis sitä kuinka hyvin uudelleenkäytettävyystavoitteessa onnistuttiin.

Everald Mills jakaa ohjelmistometriikat tuotemetriikoihin (*product metrics*) ja prosessimetriikoihin (*process metrics*). Prosessimetriikat mittaavat ohjelmiston kehitysprosessia, esimerkiksi ohjelmiston kehitykseen käytettyä aikaa tai

projektiryhmänjäsenten keskimääräistä kokemustasoa. Tuotemetriikat puolestaan mittaavat itse ohjelmiston ja sen lähdekoodin ominaisuuksia. Tuotemetriikat jaotellaan edelleen koko-, kompleksisuus- ja laatumetriikoihin. [7]

Kokometriikat kuvaavat yksinkertaisesti ohjelmiston kokoa, suuruutta tai pienuutta. Koonmittareita ovat esimerkiksi ohjelmiston lähdekoodinrivien lukumäärä (*lines of code, LOC*) ja Halsteadin ohjelmistonpituus (*Halstead's program length*). Koodirivien lukumäärä ilmoittaa ohjelmiston lähdekoodilistauksen sellaisten rivien lukumäärän, jolla on ohjelmistoon kuuluvia ohjelmointikielen lausekkeita. Halsteadin ohjelmistonpituus puolestaan laskee ohjelman sisältämien käskyjen sekä niiden kohteina olevien argumenttien lukumäärän summan. [8]

Kompleksisuusmetriikat pyrkivät puolestaan kuvaamaan ohjelmiston toteutuksen monimutkaisuutta. Kompleksisuusmittareita ovat esimerkiksi syklomaattinen kompleksisuus (*cyclomatic complexity*) useine eri laskennallisine variaatioineen, sekä ohjelmiston sisältämien ehtorakenteiden muodostaman verkon solmujen määrä (*knots*). Syklomaattisesta kompleksisuudesta puhutaan usein McCaben syklomaattisena kompleksisuutena (*McCabe's cyclomatic complexity*). McCabe kehitti syklomaattisen kompleksisuuden alun perin kuvaamaan rakenteisen ohjelmoinnin algoritmien monimutkaisuutta, ilmaisemalla algoritmin toisistaan riippumattomien suorituspolkujen lukumäärän. [9]

Laatumetriikat pyrkivät kuvaamaan ohjelmiston sisältämien virheiden määrää, ohjelmiston luotettavuutta ja ohjelmiston ylläpidettävyyttä. Ohjelman sisältämien virheiden määrää voidaan estimoida esimerkiksi seuraamalla kuinka paljon virheitä löytyy ohjelmiston koodikatselmoinneissa tai ohjelmiston testauksessa. Lisäksi löydettyjen virheiden määrää voidaan tarkastella ajanfunktiona, ja siten nähdä onko ohjelmistosta löytyvien virheiden määrän trendi ajan kuluessa laskeva vai jotain muuta. Jos trendi on laskeva, niin tästä voidaan esimerkiksi ekstrapoloida milloin ohjelmisto olisi riittävän virheetön julkaistavaksi. Trendin perusteella voidaan myös tehdä valintoja sen suhteen, pitääkö testaus- ja virheenkorjausnopeutta kasvattaa,

jotta riittävän suuri osa ohjelmiston virheistä saadaan käsiteltyä tiettyyn julkaisuajankohtaan mennessä.

3 KÄSITELTÄVÄ OHJELMISTOPROJEKTI

Analysoidun ohjelmistoprojektin päämääränä oli asiakas-palvelin -arkkitehtuurin mukaisen viestintäsovelluksen toteuttaminen matkapuhelinympäristöön. Sovellus toteutettiin kahden yrityksen yhteistyönä. Yritys, jossa tämä työ on tehty, vastasi loppukäyttäjän matkapuhelimessa toimivasta asiakasohjelmistosta. Toinen yrityksistä puolestaan vastasi matkapuhelinoperaattorin järjestelmissä toimivasta palvelinohjelmistosta sekä asiakas-palvelin -komponenttien välisen tiedonsiirtoprotokollan määrittelystä. Sovelluksen palvelinohjelmisto integroitui läheisesti matkapuhelinoperaattorin taustajärjestelmiin. Vastaavasti asiakasohjelmisto integroitui tiiviisti loppukäyttäjän matkapuhelimen yleiseen toiminnallisuuteen sekä puhelimen sisäisiin kontaktien- ja viestienkäsittelymekanismeihin.

Sovellusta oltiin toteuttamassa yritysten itse rahoittamana tuotekehityshankkeena ja sovelluksen markkina- ja jakelukanavana nähtiin matkapuhelinoperaattorit. Sovellusta markkinoitiin operaattoreille lisäarvopalveluna. Operaattorit voisivat markkinoida sovellusta valituille asiakasryhmilleen joko jälkiasenteisena palveluna että osana operaattori räätälöityä matkapuhelinta.

Asiakasohjelmistoa oltiin ensisijaisesti tekemässä Nokia S60 -matkapuhelinympäristöön, mutta samalla asiakasohjelmistosta haluttiin mahdollisimman pitkälle uudelleenkäytettävä. Taustalla oli tavoite, että asiakasohjelmisto pystyttäisiin myöhemmin kustannustehokkaasti sovittamaan muille matkapuhelinalustoille operaattori kysynnän mukaan.

3.1 Projektin aikataulutus

Ajallisesti asiakasohjelmiston kehitys aloitettiin alkusyksyllä 2008 määrittely- ja analyysivaiheella. Tästä edettiin tarkempien suunnitelmien tekemiseen ja edelleen prototypointi- ja toteutusvaiheen aloittamiseen loppusyksystä 2008. Alkutalvesta 2009 asiakasohjelmistosta valmistui rajoitetusti toimiva prototyyppiversio Nokia S60 -ympäristöön. Prototyyppi oli luonteeltaan poisheitettävä ja sitä käytettiin

vuoden 2009 aikana myynti- ja markkinointitarkoituksiin. Prototyypin valmistuttua varsinainen tuotteistettu kehitystyö pääsi vauhtiin, ja Nokia S60 -ympäristöön kohdennetun version kehitystyötä jatkettiin vuoden 2009 loppuun asti. Vaikka tekemisen painopiste olikin S60-ympäristössä, tehtiin kesän ja syksyn 2009 aikana ohjelmistosta myös prototyyppisovitus Windows Mobile -ympäristöön.

Kesän ja syksyn 2009 kuluessa ohjelmistoa markkinoitiin prototyyppiversion avulla globaalisti matkapuhelinoperaattoreille. Vuoden 2009 loppuun mennessä jouduttiin kuitenkin myöntämään se, että tuotteen ennakkokysyntä ei ollut lähtenyt toivotussa laajuudessa käyntiin. Käytännössä tuote ei ollut herättänyt operaattoreissa riittävää investointihalukkuutta taloudellisessa laskusuhdanteessa.

Alkutalvesta 2010 Nokia S60 -alustalle sovitettuun versioon saatiin valmiiksi merkittävä osa tuotteen toiminnallisista ominaisuuksista, mutta muutamat alkuperäisistä vaatimuksista sekä lopullinen tuotteistaminen matkapuhelinoperaattorien jakelukanavia varten puuttuivat vielä. Koska tuotteen myynti ei ollut kuitenkaan lähtenyt riittävissä määriin käyntiin, tehtiin yrityksessä päätös keskeyttää sovelluksen jatkokehitys toistaiseksi.

3.2 Projektin hallinnointi

Sovelluksen toteutusprojektin ohjaus perustui Scrum-kehyksen sekä muiden ketterien kehitysmenetelmien käyttöön. Scrum-kehyksen mukaisesti ohjelmistolla oli priorisoitu tekemättömien töiden lista (*product backlog list*), joka sisälsi kaikki ohjelmistoon kehitettävät asiat. Tekemättömien töiden lista oli priorisoitu sen mukaan, kuinka tärkeinä eri asioiden mukanaolo ohjelmistossa nähtiin. Lisäksi kullekin asialle oli määritetty kokoarvio, joka kuvasi asian toteuttamisen vaatimaa työmäärää. Tekemättömien töiden lista on siis yhdistelmä projektisuunnitelmasta ja ohjelmiston vaatimusmäärittelystä. Työlistan runko muodostettiin projektin alussa yhteistyössä yhteistyöyrityksen kanssa. Työlistaa päivitettiin projektin kuluessa sitä

mukaa kun ymmärrys ohjelmiston vaatimuksista ja tehtävien työmääristä tarkentuivat.

Ohjelmistoa toteutettiin keskimäärin kahden viikon mittaisissa iteraatioissa (*sprint*). Kukin lyhyt iteraatio piti sisällään perinteisen ohjelmistoprojektin vaiheet: aluksi vaatimusten analysointi, tämän jälkeen tekninen suunnittelu, toteutus ja testaus sekä lopuksi toimitus. Kunkin iteraation alussa poimittiin priorisoidulta tekemättömientöidenlistalta ne tehtävät, mitkä otettiin mukaan kyseisen iteraation sisältöön. Tämän jälkeen mukaan poimittuja tehtäviä analysoitiin tarkemmin, niiden toteutuksesta tehtiin riittävät tekniset suunnitelmat, ja lopulta tehtävät toteutettiin ja testattiin. Iteraation lopussa ohjelmistosta tehtiin kooste, joka toimitettiin tarpeen mukaan projektin eri osapuolille käyttöön.

4 UUDELLEENKÄYTTÖ PROJEKTISSA

Ohjelmistoa suunniteltaessa asiakassovelluksen mahdollisina kohdeympäristöinä nähtiin Nokia S60, Windows Mobile sekä Google Android -ympäristöt. Näiden lisäksi projektin kuluessa saadusta markkinointipalautteesta nousi esille RIM Blackberry alusta. Näistä Nokia S60 -ympäristö valikoitui ensisijaiseksi asiakassovelluksen kehitysalustaksi ensinnäkin yrityksestä löytyvän vahvan S60- ja Symbian-osaamisen johdosta, ja toisaalta siksi että Nokia S60 -ympäristö oli projektin aloitushetkellä matkapuhelinalustojen selkeä markkinajohtaja. Koska sovelluksella oli kuitenkin useita potentiaalisia kohdealustoja, otettiin uudelleenkäyttönäkökulma mukaan projektin tavoitteisiin. Määrittelyvaiheessa nähtiin mahdolliseksi, että sovellus halutaan aikaa myöten käyttöön sekä muille matkapuhelinympäristöille, että uudemmille S60-ympäristöille sitä mukaa kuin niitä julkaistaan.

Projektin alussa uudelleenkäytön eri vaihtoehtoja arvioitiin niiden teknisen toteuttavuuden ja tarjoamien mahdollisuuksien kannalta. Melko nopeasti päädyttiin tavoittelemaan pelkän sovellusytimen uudelleenkäyttöä lähdekoodin sovittamisen kautta. Sovellusytimen sisältämästä bisneslogiikasta (*business logic*) nähtiin muodostavan sellainen vahvuus, joka tulee määrittämään ohjelmiston leimalliset ominaisuudet ja joka siis kannattaa monistaa eri ympäristöjen käyttöön.

Matkapuhelinympäristöjen erilaisuuden asettamat rajoitteet ohjasivat myös osaltaan lähdekoodin sovittamisen ja pelkän ohjelmistoytimen uudelleenkäytön suuntaan. Koska mahdolliset kohdeympäristöt erosivat melko suuresti niiden tarjoamien ohjelmointirajapintojen ja käyttöliittymäkonseptien osalta, ei nähty kannattavaksi tavoitella koko asiakasohjelmiston sovitettavuutta, sillä tällöin tarvittavan mukauttamistyön määrä olisi kasvanut varsin suureksi. Sovelluksen ytimellä sen sijaan arvioitiin olevan varsin rajallinen liittymäpinta ympäröivään ohjelmaympäristöön, joten se olisi huomattavasti pienemmällä työllä sovitettavissa.

Käytettävän ohjelmointikielen valintaa puolestaan ohjasivat matkapuhelinympäristöjen tukemat kehityskielet, ja se kuinka tiiviisti kielellä sai sovelluksen integroitua ympäristön tarjoamiin palveluihin ja sisäisiin sovelluksiin. Käytännössä kieli vaihtoehtoja olivat Java, C ja C++. Java olisi ollut kielivaihtoehdoista kehittynein, mutta koska kaikki ympäristöt eivät tarjonneet sille riittävää integraatiotukea, karsiutui Java vaihtoehdoista pois.

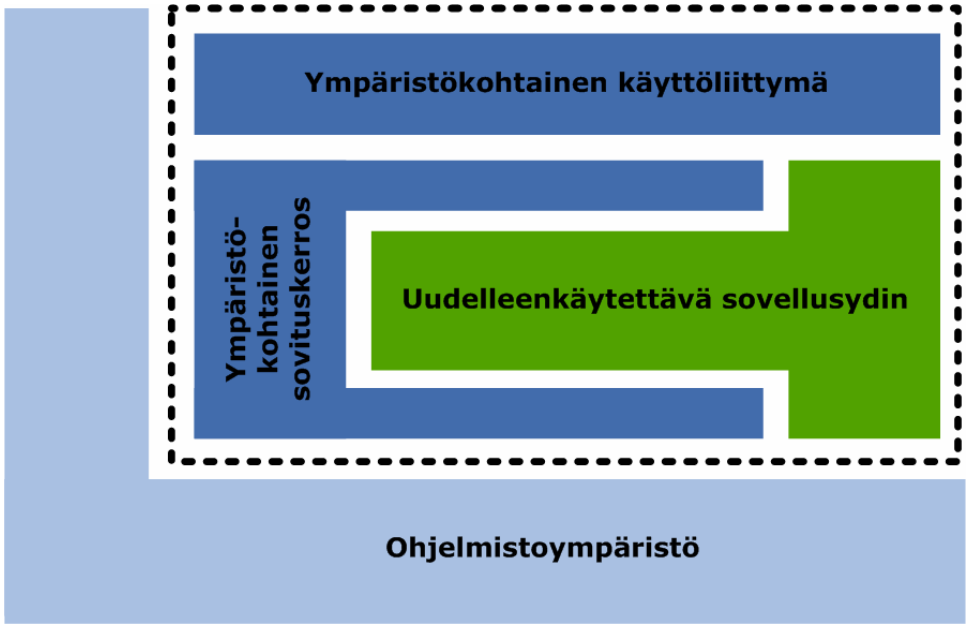
Myös C-kieltä harkittiin sovitettavan osuuden toteutuskieleksi sen yksinkertaisuuden takia, mutta lopulta päädyttiin C++:aan, olio-ohjelmoinnin tarjoaman korkeamman abstraktiotason vuoksi [10]. C++:n käyttöä puolsi myös se, että se on useimpien matkapuhelinympäristöjen sisäinen toteutuskieli ja siis kieli millä alustojen tarjoamat ohjelmointirajapinnat pääsääntöisesti julkaistaan. Käyttämällä C++:aa uudelleenkäytettävän ohjelmistoytimen toteutuskielenä, saatiin ytimen kohdeympäristöön mukauttaminen hieman yksinkertaisemmaksi, kuin mitä se olisi ollut C-kielisen ytimen tapauksessa. Lisäksi C++:n abstraktiotasoa ja ilmaisuvoimaa bisneslogiikkatyyppisten tehtävien ratkaisemiseen voitiin kasvattaa käyttämällä STL-kirjaston palveluita [11]. Tuki STL-kirjastolle löytyi kaikista C++ kehitystä tukevista kohdealustoista.

Uudelleenkäytön lähestymistavaksi valikoitui siis eräänlainen komponenttipohjaisen uudelleenkäytön ja ohjelmiston sovittamisen kombinaatio. Uudelleenkäytettävä toiminnallisuus haluttiin paketoida selkeästi eristetyksi kokonaisuudeksi, joka voitaisiin ottaa osaksi kullekin ympäristölle erikseen tehtävää sovelluskokonaisuutta. Koska mahdolliset kohdeympäristöt poikkesivat toisistaan, uudelleenkäytettävä toiminnallisuus haluttiin jaella lähdekoodimuodossa, jotta se voidaan tarvittaessa mukauttaa vastaamaan kunkin kohdeympäristön ominaisuuksia.

4.1 Uudelleenkäytettävyyden vaikutukset sovelluksen arkkitehtuuriin

Asiakasohjelmiston arkkitehtuuriksi valittiin kerrosarkkitehtuuri, jossa kontrollihierarkiassa ylimpänä on ohjelmistoympäristökohtainen käyttöliittymäkerros

ja sen alapuolella osittain limitettyinä uudelleenkäytettävä ohjelmistoydin ja ympäristökohtainen sovituskerros. Uudelleenkäytettävä ohjelmistoydin eristettiin ohjelmistoympäristöstä abstraktiorajapintojen taakse. Ydin itsessään määritti tarvittavat abstraktiorajapinnat ja nuo rajapinnat toteutettiin sovituskerroksessa, edelleen ohjaamalla abstraktiorajapinnat soveltuviin ympäristökohtaisiin rajapintoihin. Mikäli asiakasohjelmistolla ei olisi ollut uudelleenkäyttötavoitteita, ohjelmiston arkkitehtuurista olisi todennäköisesti muodostunut kaksikerroksinen ratkaisu, sisältäen pelkästään ympäristökohtaiset käyttöliittymä- ja sovellusydinkerrokset. Kuva 4-1 esittää asiakasohjelmiston toteutuneen kolmikerroksisen kerrosarkkitehtuurin.



Kuva 4-1: Asiakasohjelmiston yleinen kerrosarkkitehtuuri.

Ohjelmiston toiminnallisuuksien sijoittelu eri kerroksiin suunniteltiin niin, että uudelleenkäytettävän ytimen toiminnallisuuden määrää pyrittiin maksimoimaan ja käyttöliittymä- ja sovituskerroksen toiminnallisuuden määrä minimoimaan. Samalla ytimen toiminnallisuudessa pyrittiin sellaisiin yleisiin ratkaisumalleihin, jotka olisivat sovitettavissa eri ympäristöjen käyttöön.

Uudelleenkäytettävä sovellusydin sekä alustakohtaiset osuudet jaettiin edelleen useampaan loogiseen moduuliin, joilla kullakin oli selkeästi määritetty tehtävä. Lisäksi sovellusytimen ja alustakohtaisten moduulien välillä päädyttiin käyttämään dynaamista sidontaa. Tällöin moduulien välille ei luotu ohjelmiston käännös- ja koostamisaikana staattisia riippuvuuksia, vaan ohjelmiston sisältämistä moduuleista koostettiin tarvittava oliohierarkia dynaamisesti ajonaikana.

Modularisoinnilla pyrittiin parantamaan ohjelmistoytimen selkeyttä ja parantamaan siten ohjelmiston ylläpidettävyyttä jatkossa. Lisäksi modularisoinnin ja moduulien dynaamisen sidonnan nähtiin jatkossa helpottavan ohjelmiston sovitustyötä eri ympäristöihin, koska näin työssä voidaan edetä moduuli kerrallaan, ja jakaa sovitustyö siis pienempiin ja paremmin hallittaviin alikokonaisuuksiin. Joustava arkkitehtuuri mahdollistaa sen, että sovitustyön toiminnallisuuden testaamiseksi ei tarvitse toteuttaa kaikkia sovitettavia osuuksia kerralla valmiiksi, vaan testaus voidaan aloittaa jo valmistuneista osista ja tarjota puuttuvat riippuvuudet dynaamisesti mallikomponentteina.

4.2 Uudelleenkäytettävyyden huomioiminen kehitysprosessissa

Ohjelmiston kehitysprosessi pohjasi ketterien kehitysmenetelmien kuten Scrum-kehyksen käyttöön. Ketterät kehitysmenetelmät soveltuivat projektiin varsin hyvin, koska kyseessä oli suhteellisen pieni projekti (suurimmillaankin noin kymmenkunta kehittäjää) ja kyseessä oli yrityksen sisäinen tuotekehitysprojekti, jolloin projektin kanssa tiiviistä yhteistyötä tekevä asiakas löytyi yrityksen sisältä. Lisäksi ketterien kehitysmenetelmien joustavuus oli hyödyksi, koska kehitysprosessia ja -käytäntöjä kyettiin suhteellisen vaivattomasti korjaamaan projektin aikana, sitä mukaa kuin uudelleenkäytettävän ohjelmiston tekemisestä opittiin lisää.

Monissa kehitysvaiheen iteraatiokierroksissa kävi niin, että kaikkia kierrokselle mukaan poimittuja tehtäviä ei saatu kierroksen kuluessa valmiiksi. Projektin alussa projektiryhmällä ei ollut vielä juurikaan kokemusta sovitettavan ohjelmiston

tekemisestä, ja siis kykyä luotettavasti analysoida tekemättömien töiden listalla olevien tehtävien kokoa sekä sitä kuinka paljon yhden iteraation aikana ehditään toteuttaa asioita. Toisaalta tekemättömien töiden listalla olevat asiat olivat sisällöltään varsin laajoja, jolloin niiden koon analysointi ja vaaditun työmäärän hahmottaminen hankaloitui entisestään.

Iteraation sisällön lipsumisongelmaa pyrittiin korjaamaan projektin kuluessa analysoimalla tekemättömien töiden listalla olevia tehtäviä aktiivisesti, ja yrittämällä pilkkoa niitä mahdollisuuksien mukaan pienempiin osakokonaisuuksiin. Lisäksi listalla olevien tehtävien kuvauksia ja niiden kokoarvioita tarkennettiin projektin kuluessa useampaan otteeseen. Toisaalta iteraatioprosessia ohjattiin niin, että iteraatioille sovituista loppuajankohdista pidettiin kiinni, mutta hyväksyttiin se, että kaikki tehtävät eivät välttämättä valmistu iteraation loppuun mennessä. Iteraation vaihdoksessa keskenjääneiden tehtävien tilanne analysoitiin ja tarvittavilta osin tehtävät siirrettiin jatkettavaksi seuraavaan iteraatiokierrokseen.

Projektilla oli korkeat laatutavoitteet, koska muuten koko sovitettavan ohjelmistoytimen idea olisi valunut hukkaan. Ohjelmiston laatua pyrittiin pitämään korkealla käyttämällä muun muassa seuraavia käytäntöjä: automatisoitu testaus, testauslähtöinen kehitystapa (*test driven development*), jatkuva integrointi (*continuous integration*), pariohjelmointi (*pair programming*), yhteisesti sovitut koodauskonventiot (*coding convention*), koodin yhteisomistajuus ja tiukka tehtävien valmistumissäännöstö (*definition of done*). Lisäksi kehitysryhmässä oli erikseen nimetty käyttöliittymätestaaja, joka huolehti ohjelmiston käyttöliittymän kautta tapahtuvan testauksen määrittelystä sekä testitapausten suorittamisesta.

Automatisoiduilla testeillä pyrittiin sekä korkeaan koodikattavuuteen että sovitettavan ohjelmistoytimen testaamiseen niissä kohdeympäristöissä, joihin ohjelmistoydin sovitetaan. Tällöin automatisoituja testejä kyetään jatkossa hyödyntämään ytimen sovituksen toimivuuden verifiointiin.

Sovellusytimen sovitettavuutta verifioitiin säännöllisesti projektin aikana. Käytännössä verifiointi tapahtui osana jatkuvaa integrointia. Jatkuvassa integroinnissa koko ohjelmisto käännettiin ja koostettiin Nokia S60-alustalle. Lisäksi sovellusydin käännettiin ja testattiin automatisoiduilla testeillä erikseen Linux-ympäristössä. Ytimen sovitettavuuden verifioinnilla pyrittiin saamaan varmistus siitä, että sovitettavaan ytimeen tehty toiminnallisuus toimii muissakin ympäristöissä kuin ensisijaisessa Nokia S60 -kehitysympäristössä. Linux-ympäristön käyttämiseen sovitettavuuden varmistamisessa päädyttiin, koska Linux oli varsin suoraviivaisesti yhdistettävissä osaksi käytettyä jatkuvan integroinin hallintatyökalua. Lisäksi Linux-ympäristölle löytyi hyviä verifikaatiotyökaluja, joita voitiin hyödyntää esimerkiksi ytimen muistinhallinnan laadun tarkkailussa.

5 UUDELLEENKÄYTÖN TULOKSET

Projektin uudelleenkäyttötavoitteiden onnistumista on seuraavassa analysoitu metriikoiden avulla, tarkastelemalla kuinka suuri osuus ohjelmiston toiminnallisuudesta on uudelleenkäytettävässä sovellusytimessä. Analyysissä on tarkasteltu metriikoiden tuottamien tunnuslukujen suhteita ohjelmiston eri osien välillä. Prosessimetriikoiden analyysiä (esimerkiksi käytetty kehitysaika eri osien välillä) ei tehty, koska ongelmaksi muodostui sellaisen lähtödatan hankkiminen, josta olisi luotettavasti pystynyt erottamaan eri osien toteuttamiseen käytetyn kehitysajan. Sen sijaan tuotemetriikoita varten lähtödatan muodostaa ohjelmiston lähdekoodi. Siitä tuotemetriikoiden tunnusluvut määritettiin kaupallisen Understand C++ -koodianalysaattorin avulla [12].

5.1 Uudelleenkäytettävissä olevan toiminnallisuuden määrä

Uudelleenkäytettävän toiminnallisuudesta määrän tarkastelua varten Nokia S60 -ohjelmistoversion lähdekoodi jaettiin kolmeen osaan, jotka ovat uudelleenkäytettävä ohjelmistoydin, S60-ympäristön sovituskerros ja S60-ympäristön käyttöliittymäkerros. Osien suhteellista kokoa on tarkasteltu C++ esittely- (*declarative statements*) ja suorituslauseiden (*executable statements*) yhteenlaskettujen määrien avulla. C++ lauseisiin pohjaavaa tarkastelua on käytetty, koska uudelleenkäytettävällä ytimellä ja S60-kohtaisella lähdekoodilla oli käytössään hieman toisistaan poikkeavat koodauskonventiot. Poikkeavat koodauskonventiot vaikuttivat esimerkiksi siihen kuinka koodi rivitetään riveille ja täten teki rivipohjaisesta koodimäärien vertailusta epäluotettavaa.

Taulukko 1 sisältää ohjelmiston osille mitatut C++ lauseiden kappalemäärät sekä näiden kappalemäärien prosentuaaliset osuudet ohjelmiston C++ lauseiden kokonaismäärästä. Noin 40% ohjelmiston kaikista C++ lauseista on uudelleenkäytettävässä ohjelmistoytimessä. Lisäksi uudelleenkäytettävä ydin sisältää yli kolme kertaa sen määrän C++ lauseita kuin mitä S60-sovituskerroksessa on.

	Ohjelmistoydin	Sovituskerros (S60)	Käyttöliittymä-kerros (S60)
C++ lauseiden kappalemäärä	6058	1848	7397
Prosentuaalinen osuus C++ lauseiden kokonaismäärästä	39,6	12,1	48,3

Taulukko 1: Ohjelmiston osien C++ lauseiden kappalemäärät sekä prosentuaaliset osuudet lauseiden kokonaismäärästä.

Ohjelmiston osien sisältämän logiikan määrää on arvioitu käyttäen McCaben syklomaattista kompleksisuutta. Alkujaan rakenteelliselle koodille esitettyä metriikkaa voidaan käyttää myös oliokielten analysointiin, koska oliokielissäkin ohjelman kontrollirakenne muodostaa vastaavanlaisen verkon, kuin rakenteellisen ohjelmoinnin algoritmit ja aliohjelmat muodostavat [13]. Taulukko 2 sisältää ohjelmiston osille mitatut McCaben syklomaattisen kompleksisuuden kokonaisarvot sekä kunkin ohjelmiston osan prosentuaalisen osuuden syklomaattisen kompleksisuuden kokonaismäärästä.

Syklomaattisen kompleksisuuden jakautuminen ohjelmiston osien välillä noudattaa hyvin tarkasti osien suhteellisia kokoja. Noin 40% ohjelmiston logiikasta on uudelleenkäytettävässä ohjelmistoytimessä. Uudelleenkäytettävä ydin sisältää myös reilut kolme kertaa enemmän logiikkaa kuin S60-sovituskerros. Toisin sanoen hyödyntämällä uudelleenkäytettävää ohjelmistoydintä ja tekemällä vaaditun sovituskerroksen, saa ytimestä kolme kertaa enemmän toiminnallisuutta takaisin kuin mitä sovituskerroksen tekeminen itsessään vaatii.

Lisäksi uudelleenkäytettävän ohjelmistoytimen sisältöä voidaan pitää koko ohjelmiston toiminnallisuuden kannalta merkittävämpänä kuin mitä ympäristökohtaiset kerrokset ovat. Sovellusytimen logiikka on ohjelmiston sisäisen toiminnan ja ominaisuuksien kannalta oleellisinta ja sen kehittäminen vaatii syvällistä aihealueen tuntemusta. Alustakohtaisten kerrosten toiminnallisuus on pitkälti ohjelmistoympäristön käyttöliittymä ja muihin rajapintoihin yhteydessä olevaa yleisluontoisempaa logiikkaa. Alustakohtaisten kerrosten toteuttaminen ei vaadi niin

syvällistä aihealueen tuntemusta, vaan siihen riittää yleisempi ohjelmistoympäristön rajapintojen sekä uudelleenkäytettävän ohjelmistoytimen rajapintojen tuntemus.

	Ohjelmistoydin	**Sovituskerros (S60)**	**Käyttöliittymä-kerros (S60)**
McCaben syklomaattinen kompleksisuus	1392	444	1643
Prosentuaalinen osuus syklomaattisen kompleksisuuden kokonaismäärästä	40	12,8	47,2

Taulukko 2: Ohjelmiston osien McCaben syklomaattisen kompleksisuuden arvot sekä prosentuaaliset osuudet syklomaattisen kompleksisuuden kokonaismäärästä.

5.2 Aikatauluvaikutukset

Ohjelmistoytimen sovitettavuus heijastui myös projektin työmäärään. Sovitettavuus lisäsi ohjelmiston tekemiseen selkeästi sellaisia tehtäviä, joita minimalistisessa yhdelle alustalle kohdistetussa ohjelmistossa ei olisi ollut. Näitä sovitettavuuden mukanaan tuomia lisäosuuksia olivat esimerkiksi laajemmasta arkkitehtuurista johtuvat kerrosten väliset abstraktiorajapinnat sekä alustakohtaisen sovituskerroksen toteuttaminen ja testaaminen.

Myös sovitettavan osuuden toteuttaminen valtaosalle projektin kehittäjistä hieman vieraammalla standardi-C++:lla ja STL-kirjaston avulla nosti työmäärää. Vieraampi ohjelmointikieli vaikutti työkuormaan etenkin projektin alussa, kun varsinaisen kehitystyön lisäksi jouduttiin samalla opettelemaan uuden C++-murteen käyttöä.

Työmäärää kasvatti jossain määrin myös ohjelmistoytimen sovitettavuuden verifiointi osana ohjelmistonkehityssykliä. Vaikka itse verifiointi tapahtuikin automaattisesti testien avulla osana jatkuvaa integrointia, löytyi tuossa verifioinnissa aina välillä ongelmia jo tehdystä ytimen toteutuksesta. Jossain tapauksissa ytimeen vaaditut korjaukset eivät olleet täysin triviaaleja, vaan ongelmien syiden selvittely

sekä korjaavan toteutuksen tekeminen ja testaaminen saattoi viedä kehittäjältä huomattavastikin työaikaa.

Myös sovitettavuuden takia ehkä jopa hieman normaalia korkeammat laatutavoitteet vaikuttivat työmääriin, sillä projektissa toteutettiin itse ohjelmiston lisäksi suuri määrä automatisoituja testejä. Taulukko 3 sisältää ohjelmiston osien ja niitä vastaavien yksikkö- ja moduulitestien C++ suorituslauseiden määrät. Taulukosta nähdään että kutakin ohjelmistoytimen C++ suorituslausetta vastaa noin kolme C++ suorituslausetta testikoodissa. Vastaavasti S60-sovituskerroksella on noin 2,2 ja S60-käyttöliittymäkerroksella noin 1,8-kertainen määrä testikoodilauseita verrattuna itse ohjelmiston koodiin.

	Ohjelmistoydin	**Sovituskerros (S60)**	**Käyttöliittymä-kerros (S60)**
C++ suorituslauseiden kappalemäärä	2664	955	3195
Testikoodin C++ suorituslauseiden kappalemäärä	8137	2130	5813

Taulukko 3: Ohjelmiston osien sekä niitä vastaavien testikoodien C++ suorituslauseiden kokonaismäärät.

5.3 Prototyyppi Windows Mobile -ympäristöön

Kesän ja syksyn 2009 kuluessa asiakasohjelmistosta tehtiin prototyyppi Windows Mobile -ympäristöön. Windows Mobile -sovituksella pyrittiin varmistumaan uudelleenkäytettävän ytimen sovitettavuudesta laajemmassa mittakaavassa. Toisaalta sovelluksesta saatiin samalla Windows Mobile -versio myynti- ja markkinointitarkoituksiin.

Vaikka samanlaista sovitettavuuden varmistustyötä tehtiin varsinaisen kehitystyön rinnalla Linux -alustan ja automatisoitujen testien avulla, eivät ne kuitenkaan antaneet täydellistä kuvaa koko ytimen toiminnallisuuden sovittamisesta, eivätkä siitä

kuinka kokonainen sovellus saadaan toisessa ympäristössä toteutettua sovellusytimen päälle. Windows Mobile -sovituksesta saatiinkin arvokasta palautetta sekä itse ytimen sisäisten toimintojen suunnitteluun, että myös ytimen asiakas- ja sovitusrajapintojen suunnitteluun. Yksi esimerkki Windows Mobile -sovitustyön kautta esille nousseista asioista oli dynaamisen sidonnan käyttöönotto ytimen ja sovituskerroksen moduulien välillä. Joustava dynaaminen tekniikka mahdollisti kevyemmän arkkitehtuurin toteuttamisen Windows Mobile -ympäristöön.

Käytännössä Windows Mobile -sovitustyö onnistui hyvin ja se osoitti sovitettavan lähestymistavan vahvuuden. Sovellusytimen sovittaminen ja Windows Mobile sovitus- ja käyttöliittymäkerrosten toteuttaminen vaati vain murto-osan siitä työmäärästä, jonka uudelleenkäytettävän sovellusytimen kehittäminen itsessään oli vienyt.

6 YHTEENVETO

Tässä kandidaatintyössä on kuvattu erästä matkapuhelinympäristöön suunnattua ohjelmistoprojektia ja sen käyttämää ohjelmistojen uudelleenkäyttöratkaisua. Kirjallisuuden mukaan käytetyimpiä ohjelmistojen uudelleenkäyttöratkaisuja ovat komponenttipohjainen uudelleenkäyttö, arkkitehtuuriratkaisujen uudelleenkäyttö ja ohjelmistotuotelinja. Eräiden lähteiden mukaan ohjelmistoympäristöjen välillä tapahtuva ohjelmistojen sovittaminen lasketaan myös ohjelmistojen uudelleenkäytöksi. Ohjelmistojen sovittamisessa pyritään samoihin ohjelmistotuotannon tehokkuuden kasvattamistavoitteisiin kuin perinteisissä uudelleenkäytön lähestymistavoissa.

Tutkitussa projektissa tehtiin sovellus Nokia S60 -matkapuhelinympäristölle. Sovelluksesta pyrittiin tekemään samalla keskeisiltä osin uudelleenkäytettävä, jotta se kyettäisiin jatkossa sovittamaan kustannustehokkaasti muille matkapuhelinalustoille. Uudelleenkäytön lähestymistavaksi projektissa valikoitui komponenttipohjaisen uudelleenkäytön ja ohjelmiston sovittamisen yhdistelmä. Sovelluksen oleellisen toiminnallisuuden sisältävä ohjelmistoydin eristettiin uudelleenkäytettäviksi moduuleiksi, jotka sovitetaan kohdeympäristöille. Pelkän ohjelmistoytimen uudelleenkäyttöön päädyttiin, koska sen nähtiin tarjoavan paras hyöty sovittamisen vaatimaan työmäärään nähden.

Arkkitehtuurisesti ohjelmisto jakautui kolmeen osuuteen. Kontrollihierarkian ylimmän kerroksen muodosti ympäristökohtainen käyttöliittymäkerros. Sen alapuolella olivat osittain limitettyinä uudelleenkäytettävä ohjelmistoydin ja ympäristökohtainen sovituskerros. Käyttöliittymäkerros hyödynsi ohjelmistoytimen palveluita loppukäyttäjälle näkyvän toiminnallisuuden toteuttamiseen. Ympäristökohtainen sovituskerros puolestaan sovitti uudelleenkäytettävän ohjelmistoytimen ympäröivän ohjelmistoympäristön rajapintoihin.

Noin 40% Nokia S60 -ympäristöön toteutetun sovelluksen lähdekoodista ja logiikasta sijaitsi uudelleenkäytettävässä ohjelmistoytimessä. Uudelleenkäytettävä

ydin sisälsi myös reilut kolme kertaa enemmän logiikkaa kuin S60-sovituskerros. Merkittävä osa sovelluksen sisältämästä toiminnallisuudesta saatiin siis uudelleenkäytettäväksi. Lisäksi uudelleenkäytettäväksi saatu toiminnallisuus oli ohjelmiston kannalta merkittävää, koska se sisälsi syvällistä ohjelmiston ongelma-alueen tietämystä verrattuna yleisluontoisempiin käyttöliittymä- ja sovituksenkerrokseen toimintoihin.

Sovelluksesta tehtiin lisäksi prototyyppisovitus Windows Mobile -ympäristölle. Windows Mobile -sovitus osoitti käytännössä että uudelleenkäytettävästä ohjelmistoytimestä oli konkreettista hyötyä. Windows Mobile -sovitus valmistui murto-osalla siitä työmäärästä, jonka alkuperäisen ohjelmistoytimen kehittäminen itsessään oli vienyt.

LÄHDELUETTELO

[1] Krueger, Charles W. (1992) Software reuse,
ACM Computing Surveys 24 (2), New York, s. 131 – 183

[2] Poulin, Jeffrey S. (1997)
Measuring Software Reuse: principles, practices, and economic models,
Addison-Wesley Longman Inc., Boston

[3] Frakes, William & Kang, Kyo (2005)
Software Reuse Research: Status and Future,
IEEE Transactions on Software Engineering 31 (7), Los Alamitos,
s.529 – 536

[4] Shiva, Sajjan G. & Shala, Lubna Abou (2007)
Software Reuse: Research and Practice,
ITNG Fourth International Conference on Information Technology,
IEEE Computer Society , Washington, s. 603 – 609

[5] Mooney, James D. (1990)
Strategies for supporting application portability,
IEEE Computer 23 (11), s. 59 – 70

[6] Parnas, David (1994) Software Aging,
Proceedings of 16th International Conference on Software
Engineering,
IEEE Computer Society Press, s. 279 – 287

[7] Mills, Everald (1988) Software Metrics,
Software Engineering Institute Curriculum Module 12-1.1,
Software Engineering Institute, Carnegie Mellon University

[8] Halstead, Maurice (1977) Elements of Software Science,
Elsevier, North-Holland

[9] McCabe, Thomas (1976) A complexity measure,
IEEE Transactions on Software Engineering 2(4), s. 308 – 320

[10] Stroustrup, Bjarne (2007)
Evolving a language in and for the real world: C++ 1991-2006,
ACM History of programming languages 3

[11] Alexander Stepanov & Meng Lee (1994)
The Standard Template Library
HP Labs TR HPL-94-34

[12] Scientific Toolworks Inc.
Understand C++ Source Code Analysis & Metrics tool

WWW-sivu, http://www.scitools.com/.
Luettu: 6.3.2010

[13] Tegarden, David; Sheetz, Steven & Monarchi, David (1992)
Effectiveness of Traditional Software Metrics for Object-Oriented
Systems Proceedings: 25th Hawaii International Conference on System
Sciences, s. 359 – 368