Juhani Lammassaari

Mobiilisovellusten siirrettävyys ja toteutuksen abstraktiotaso

Tietotekniikan (ohjelmistotekniikka)
pro gradu -tutkielma
31. lokakuuta 2006

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Juhani Lammassaari
Yhteystiedot: Puhelin: 044 012 4816, sähköposti: nules@iki.fi
Työn nimi: Mobiilisovellusten siirrettävyys ja toteutuksen abstraktiotaso
Title in English: Portability of Mobile Applications and Implementation's Abstraction Level
Työ: Tietotekniikan (ohjelmistotekniikka) pro gradu -tutkielma
Sivumäärä: 86
Tiivistelmä: Tässä pro gradu -tutkielmassa käsitellään mobiilisovellusten siirrettävyyttä sekä toteutuksen abstraktiotasoa. Työssä esitellään mobiilisovelluskehityksen erityispiirteet, joista fragmentaatio, eli mobiililaitteiden ominaisuuksien erilaisuus, vaikuttaa usein eniten sovellusten siirrettävyyteen. Lisäksi työssä tutkitaan, millä keinoin ja kuinka paljon mobiilisovellusohjelmoinnin abstraktiotasoa olisi suositeltavaa nostaa. Abstraktiotason nostolla voidaan piilottaa sovelluskehittäjältä laitteiden fragmentaatiota ja näin tehostaa sekä nopeuttaa sovelluskehitystyötä. Työssä tutkitaan olemassa olevia mobiililaitteiden fragmentaatiosta aiheutuvia ongelmia ratkaisevia työkaluja ja niiden toimintatapoja. Työkalujen hyvien ominaisuuksien pohjalta toteutetaan mobiilisovelluskehitykseen fragmentaatiota piilottava Preppi-liitännäinen Eclipse-sovelluskehitysympäristöön.

English abstract: In this thesis the portability and abstraction level of mobile applications are examined. The special characteristics of mobile software development are introduced, of which the mobile device fragmentation is mainly studied because it affects the most to the portability of mobile software. In this thesis different methods to raise abstraction level of source code are inspected. With risen level of abstraction many problems caused by device fragmentation can be solved. In addition, the present tools to avoid fragmentation problems are evaluated and a Preppi plug-in, optimized for mobile software development, for Eclipse integrated development environment is designed and implemented.
Avainsanat: Mobiiliohjelmointi, abstraktiotaso, mobiililaitteiden fragmentaatio, esikäännös, Eclipse, liitännäinen, Java ME
Keywords: Mobile Software Engineering, Abstraction level, Fragmentation of Mobile Devices, Preprocessing, Eclipse, Plug-In, Java ME

Esipuhe

Tämä työ on tehty Sysline Oy:n ehdottamasta aiheesta. Kiitänkin yrityksen edustajina toimineita Niko Luojumäkeä ja Jukka Matilaista, jotka tarjosivat ohjausta sitä tarvitessani sekä antoivat rakentavaa palautetta työn edistyessä.

Kiitokset tahdon osoittaa myös tutkielmani ohjaajina toimineille professori Tommi Kärkkäiselle sekä filosofian maisteri Ville Tirroselle paneutumisesta työhöni kaikissa sen vaiheissa. Kiitokset lisäksi kaikille ystävilleni, jotka ovat olleet suurena apuna ja jaksaneet aina tarpeen mukaan irroittaa allekirjoittaneen lopputyön hionnasta opiskelijaelämän rentouttaviin rientoihin.

Sanasto

API	eli *Application Programming Interface* (suom. ohjelmointirajapinta) tarkoittaa järjestelmän tai yksittäisen luokan tarjoamien toimintojen luetteloa.
Bluetooth	on lyhyen kantaman radiotekniikkaan perustuva standardoitu langaton tiedonsiirtotekniikka.
C++	on Bjarne Stroustrupin suunnittelema ja toteuttama C-kielen pohjalta kehitetty olio-ohjelmointiominaisuudet sisältävä ohjelmointikieli.
CLDC	lyhenne tulee sanoista *Connected Limited Device Configuration* ja tarkoittaa Java ME -konfiguraatiota, joka määrittelee minimikirjastot, jotka sitä tukevassa laitteessa tulee olla.
CPP	lyhenne tulee sanoista *C Preprocessor* ja tarkoittaa C-kielen esikääntäjää.
GPRS	lyhenne tulee sanoista *General Packet Ratio Service* ja on GSM-verkossa toimiva pakettikytkentäinen tiedonsiirtopalvelu.
CSS	eli *Cascading Style Sheets* on www-standardi, joka määrittelee kotisivujen tyylitiedoston rakenteen.
Eclipse	on graafinen sovellusalusta ja sovelluskehitysympäristö, jossa on työkalut mm. Java-sovellusten kehittämiseen.
EclipseME	on Eclipsen liitännäinen, joka mahdollistaa mobiilisovellusten kehittämisen käyttäen Java ME -ohjelmointikieltä.
EMF	eli *Eclipse Modeling Framework* on Eclipsen malli- sekä lähdekoodin generoimiskomponentti, jonka avulla voidaan toteuttaa Eclipsen työkaluja sekä muita sovelluksia, jotka perustuvat määrättyihin jäsennettyihin tietomallirakenteisiin.

Fragmentaatio	tarkoittaa mobiililaitteiden eroavaisuuksia. Näitä eroavaisuuksia ovat mm. mobiililaitealustojen-, käyttöliittymien, käyttöjärjestelmien ja systeemikirjastojen sekä muiden resurssien erot.
HTML	eli *Hypertext Markup Language* on www-sivujen sisällön ja rakenteen määrittelyyn käytettävä merkkauskieli.
IDE	lyhenne tulee sanoista *Integrated Developement Environment* ja tarkoittaa integroitua sovelluskehitysympäristöä.
JAD	eli *Java Application Descriptor* on tiedosto, joka määrittelee kuinka mobiililaitteen tulee käsitellä siihen liitettyä sovellusta. Käsittelyllä tarkoitetaan mm. asennusta, tunnistusta ja suorittamista.
JAR	lyhenne tulee sanoista *Java Archive* ja tarkoittaa Javan kokoelmatiedostoa, joka sisältää käännetyn sovelluksen, resurssitiedostot sekä manifest-tiedoston.
Java	on Sun Microsystems -yhtiön kehittämä ohjelmointikieli, josta on kehitetty eri versiot mobiililaitteille (Java ME, Java Micro Edition), työpöytätietokoneisiin (Java SE, Standard Edition) sekä liiketoimintaratkaisuihin (Java EE, Enterprise Edition).
JET	tulee sanoista *Java Emitter Templates* ja tarkoittaa Eclipsen työkalua, jolla voidaan sisällyttää määriteltyjä lähdekoodimallitiedostoja lähdekoodiin.
JNI	eli *Java Native Interface* tarkoittaa Java-kielen tekniikkaa liittää Java-sovelluksen toimintaan muilla ohjelmointikielillä toteutettuja sovelluksen toimintoja.
JTWI	tulee sanoista *Java^{TM} Technology for the Wireless Industry* ja tarkoittaa JSR-185 määrittelyä, joka määrittelee standardin alustan Java-teknologiaa tukeville mobiililaitteille.
JVM	lyhenne tarkoittaa Javan virtuaalikonetta (*Java Virtual Machine*), joka suorittaa Java-sovelluksen tavukoodia.
Middleware	tarkoittaa välikomponenttia tai -sovellusta, joka tarjoaa palveluita muiden komponenttien ja sovellusten käytettäväksi.

MIDlet	on Java-sovellus mobiileille päätelaitteille.
MIDP	lyhenne tulee sanoista *Mobile Information Device Profile*. MIDP tarkoittaa kokoelmaa Javan ohjelmointirajapintoja, jotka tarjoavat sovelluskehittäjälle perustoiminnallisuudet Java ME -ympäristössä.
MSA	eli *Mobile Service Architecture* on määritys, joka kokoaa olemassa olevia Java ME -ympäristön määrityksiä yhdeksi tarkoin määritellyksi kokoelmaksi.
MSAA	eli *Mobile Service Architecture Advanced* on MSA-määritystä kehittyneemmille laitteille toteutettava määritys, joka kokoaa olemassa olevia Java ME -ympäristön määrityksiä yhdeksi tarkoin määritellyksi kokoelmaksi.
PDA	lyhenne tulee sanoista *Personal Digital Assistant* ja tarkoittaa kämmenmikroa.
Plug-In	on sovelluksen liitännäinen, joka laajentaa sovelluksen tai muiden sovellukseen liitettyjen liitännäisten toimintaa.
Reflektointi	tarkoittaa sovelluksen kykyä tarkkailla ja mahdollisesti muuttaa itseään.
Symbian OS	on Symbian Ltd:n luoma mobiileihin päätelaitteisiin tarkoitettu käyttöjärjestelmä, josta käytetään myös nimitystä Symbian Platform.
WLAN	tulee sanoista *Wireless Local Area Network* ja tarkoittaa langatonta lähiverkkoa.
Wrapper-luokka	tarkoittaa luokkaa, jonka tehtävänä on yhtenäistää tiettyjen toimintojen suorittaminen riippumatta käytetystä ympäristöstä.
XML	eli *Extensible Markup Language* on rakenteinen ihmisten ja tietokoneiden ymmärtämässä muodossa oleva merkkauskieli, jolla voidaan kuvata erilaisia tietueita käyttötapauskohtaisella merkkauksella.

Sisältö

1 Johdanto

Sovellusten siirrettävyys erilaisten mobiililaitteiden välillä on mobiiliohjelmistotuotannossa välttämätöntä. Ohjelmistoista joudutaan nykyisin mobiililaitteiden suuren fragmentaation, eli ominaisuuksien erilaisuuden, vuoksi räätälöimään eri käännökset lähes jokaiselle mobiililaitemerkille ja -mallille. Tämä aiheuttaa luonnollisesti mobiilisovelluksia kehittäville projekteille lisäkustannuksia vaaditun työajan muodossa sekä saattaa heikentää toteutettujen sovellusten laatua.

Tässä tutkielmassa esittelen mobiilisovelluskehityksen erityispiirteet perinteiseen työasemasovelluskehitykseen verrattuna. Mobiilisovelluskehityksen erityispiirteitä ovat muunmuassa fragmentaation, vähäisten resurssien sekä käytettävyys- ja siirrettävyysstekijöiden huomioon ottaminen.

Lähdekoodin abstrahoinnilla tarkoitetaan määritellyn ongelman ratkaisun yksityiskohtien piilottamista tietyn rajapinnan, jonka käytetty abstraktiotaso määrittelee, taakse. Lähdekoodin abstraktiotasoa kohottamalla voidaan sovelluskehittäjältä piilottaa kohdelaitteistojen eroja ja pyritään joissain tapauksissa kiertämään kohdelaitteistoissa ilmenneitä virheellisiä toimintoja. Käytetty ohjelmointikieli määrittelee lähdekoodin perusabstraktiotason, mutta sovelluskehittäjän toteuttamat aliohjelmat ja -metodit sekä käytetyt kirjastojen toiminnallisuudet kohottavat abstraktiotasoa. Abstraktiotason nosto nopeuttaa sovelluskehitystyötä sekä vähentää sovelluskehittäjän muistamistarvetta, sillä hänen ei tarvitse muistaa niin kattavasti eri kohdelaitteiden eroavuuksia. Hyvin määritelty abstrahointi lähdekooditasolla ja sen oikea käyttö parantavat tuntuvasti sovellusten siirrettävyyttä eri laitealustoille. Lisäksi lähdekoodin luettavuus ja ymmärrettävyys paranevat huomattavasti.

Tutkielmassa tutkin lisäksi sopivia lähdekoodin abstraktiotasoja mobiilisovelluskehityksen alueella, millä keinoin sovelluskehityksen abstraktiotasoa voidaan kohottaa ja mitkä abstraktiotasot ovat soveltuvimpia tämänhetkiseen mobiilisovelluskehitykseen geneerisillä sovelluskehityksen osa-alueilla, joita ovat esimerkiksi grafiikka, äänentoisto, yhteyskäytänteet ja tiedonsyöttövälineet. Tässä tutkielmassa tutkin sopivia lähdekoodin abstraktiotasoja mobiilisovelluskehityksessä sekä abstraktiotason nostamisesta ja käyttämisestä mahdollisesti aiheutuvia ongelmia.

Lähdekoodin abstraktiotasoa voidaan nostaa useilla eri keinoilla. Mobiililaitteiden vähäisten resurssien vuoksi olio-ohjelmointipohjaiset ratkaisut sovellusten siirrettävyyden parantamiseksi ovat vielä nykyisin poissuljettu vaihtoehto, sillä ne kasvattavat

sovelluksen kokoa yleensä liian paljon. Erittäin korkean abstraktiotason kieliä käytettäessä joudutaan yleensä opettelemaan uusi ohjelmointikieli sekä siihen liittyvien työkalujen käyttö. Lisäksi liian korkea ohjelmointikielen abstraktiotaso saattaa rajoittaa sovelluskehittäjän vapautta toteuttaa ja optimoida kehitetyn sovelluksen toimintaa. Sovellusten koon pienenä säilyttämisen ja samalla siirrettävyyden ylläpitämisen vuoksi esikäännöksellä toteutettu abstraktiotason nosto on varsin yleisesti käytetty ja toimivaksi todettu vaihtoehto. Esikääntäjän käytöllä pyritään korvaamaan laitteistokohtaiset lähdekoodin osaset geneerisemmillä, korkeamman abstraktiotason, esikäännöskomennoilla, joiden perusteella esikääntäjä muokkaa annettua lähdekoodia halutulle kohdelaitteistolle sopivaksi.

Lopuksi tutkin Java ME -ympäristöön toteutettuja siirrettävyyttä parantavia, ja mahdollisesti lähdekoodin abstraktiotasoa kohottavia, työvälineitä. Tutkimustulosten perusteella toteutan Eclipse-sovelluskehitysympäristöön Preppi-liitännäisen, joka toimii esikääntäjänä ja mahdollistaa lähdekoodin abstraktiotason noston halutulle tasolle, jolla voidaan piilottaa sovelluskehittäjältä mobiililaitteiden fragmentaation vaatimat räätälöidyt lähdekoodilohkot. Tässä käytännönläheisemmässä vaiheessa tutkin myös mahdollisia EclipseME-liitännäisen käytöstä koituvia ongelmatilanteita.

Tämän tutkielman luvussa 2 esitellään mobiililaite käsitteenä, mobiilisovelluskehityksen erityispiirteitä sekä mobiilisovelluskehityksen tulevaisuudennäkymiä. Luku 3 käsittelee abstraktiotaso-käsitettä ja mobiilisovelluskehitykseen soveltuvaa abstraktiotasoa. Luvussa 4 kerrotaan esikääntäjistä sekä esikäännöksen hyödyistä ja haitoista. Luvussa 5 tutkitaan aikaisemmin fragmentaatio-ongelman ratkaisemiseksi kehitettyjä työkaluja. Luvussa 6 esitellään luvun 5 pohjalta kehitetty Preppi-liitännäinen Eclipse-ympäristöön, jolla voidaan tehokkaasti ja helposti ratkaista fragmentaatiosta aiheutuvia ongelmatilanteita. Luvussa 7 esitetään yhteenveto tutkielman taustoista ja tuloksista.

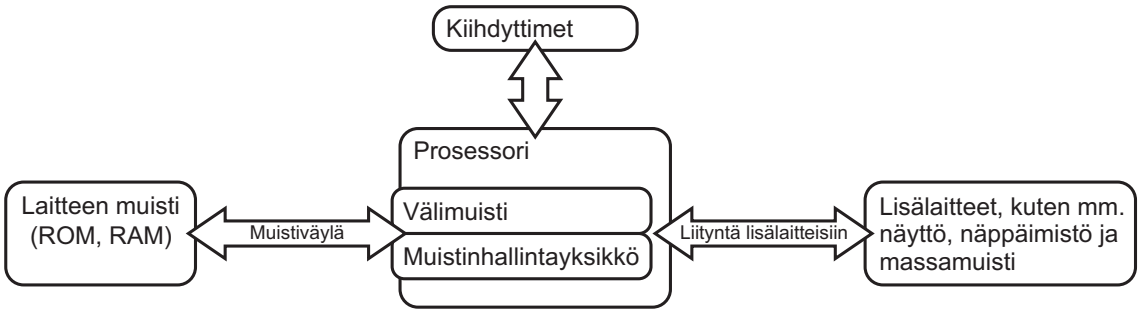
2 Mobiilisovelluskehitys

Vaikka mobiilisovelluskehitys on perusperiaatteiltaan samanlaista kuin sovellusten kehitys työpöytätietokoneille, on siinä myös poikkeavia piirteitä. Erot tulevat ilmi jo sovelluksen määrittely- ja suunnitteluvaiheissa kun joudutaan huomioimaan mobiililaitteen rajalliset resurssit, erilaiset yhteyskäytänteet sekä rajoitettu käytettävyys.

Tässä luvussa esittelen yleisesti mikä mobiililaite on ja mitä eroja mobiilisovelluskehityksessä on perinteisemmällä työpöytätietokoneille suunnatussa sovelluskehityksessä. Tämä luku perustuu pääosin lähteeseen [23].

2.1 Mobiililaite

Mobiililaite on terminä varsin vakiintunut yleiskielen sana, mutta sen määrittely teknisessä mielessä on varsin epätäsmällinen. Ei ole olemassa yksikäsitteistä määritelmää, mikä rajaisi laitteen mobiiliksi tai ei-mobiiliksi. Syynä tähän voidaan pitää mobiilien järjestelmien nuorta ikää. Yleisesti mobiililaitteella tarkoitetaan kuitenkin kaikkia tietojärjestelmän, kuten käyttöjärjestelmän tai muun ohjaussovelluksen, sisältämiä laitteita, joita niiden käyttäjä kuljettaa mukanaan. Näistä ovat esimerkkejä mm. kannettava tietokone, matkapuhelin ja PDA (Personal Digital Assistant).



Kuva 2.1: Mobiililaitteen arkkitehtuuri

Vaikka on olemassa useita erilaisia mobiililaitteita, niin niiden periaatteellinen rakenne on samanlainen. Tätä rakennetta esitellään kuvassa 2.1. Mobiililaitteessa on järjestelmän ytimenä toimiva prosessori, jolla on oma välimuisti ja muistinhallintayksikkö, joka hallitsee laitteen sisäisiä muisteja. Prosessorin kanssa toimivat kiinteästi erilaiset kiihdyttimet, jotka ovat erikoistuneet tiettyihin nopeutta vaativiin tehtäviin kuten esi-

merkiksi puheäänen ja videokuvan pakkaamiseen sekä purkuun videopuhelun aikana. Laitteessa on lisäksi liityntäväylä erilaisiin lisälaitteisiin, joita ovat esimerkiksi näyttö, näppäimistö ja muistikortti.

Jäljempänä tarkoitan mobiililaitteilla vain matkapuhelimia ja PDA-laitteita, koska käsiteltävä aihe koskee vain näiden pieniresurssisten laitteiden alueella tapahtuvaa sovelluskehitystä.

2.2 Mobiilisovelluskehityksen erityispiireet

Ohjelmistotuotanto mobiililaitteelle on usein haastavampaa kuin ohjelmistojen tuottaminen työpöytätietokoneille. Mobiilisovellusten toteuttamisen haasteellisuus aiheutuu mobiililaitteiden varsin rajallisista resursseista ja laitteiden fragmentaatiosta. Työasemaympäristössä resurssit ovat mobiiliympäristöön verrattuna rajattomat eikä vastaavan mobiilisovelluksen tuottaminen työasemaympäristöön tuota yleensä suuriakaan ongelmia, kun mobiililaitteiden erityispiirteitä kuten erilaisia yhteysmuotoja ei huomioida. Luonnollisesti mobiililaitteiden vähäiset resurssit rajoittavat myös mobiilisovellusten kompleksisuutta. Monimutkaisen mobiilisovelluksen toteuttaminen vaatii usein työlään optimointityön, jotta kohdelaitteiden resurssit saadaan riittämään sovelluksen suorittamiseksi.

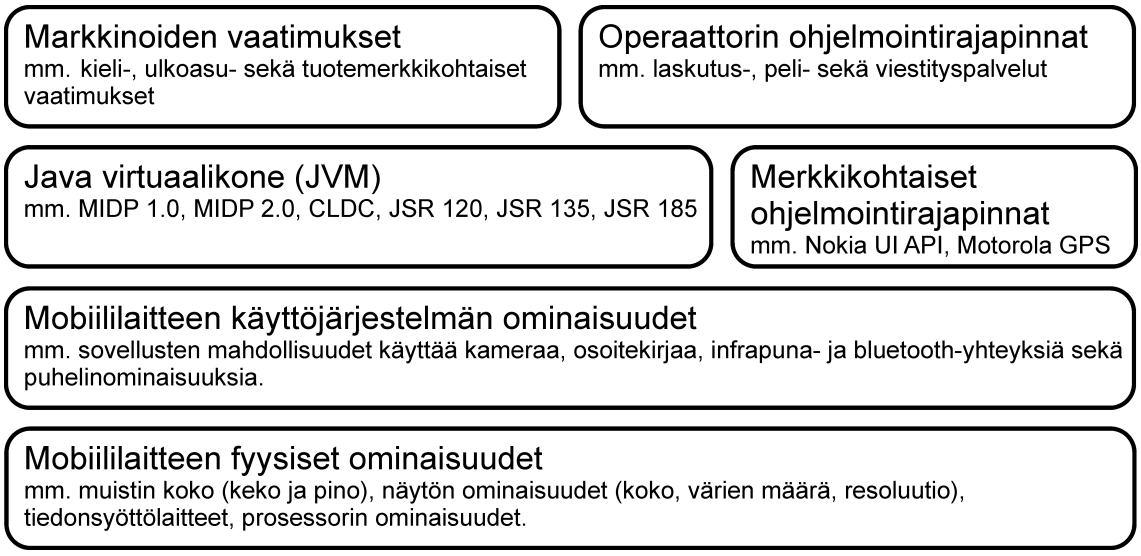
Mobiililaitteen ja sulautetun järjestelmän suuri periaatteellinen ero on se, että sulautetut suunnitellaan tyypillisesti yhtä muuttumatonta tarkoitusta varten, kun taas mobiililaitteen toiminnallisuutta voidaan usein laajentaa lisälaitteilla ja asennettavilla sovelluksilla. Tämä laajennettavuusmahdollisuus saattaa monimutkaistaa mobiilisovellusten tuotantoprosessia verrattuna sulautettujen järjestelmien sovellustuotantoon. Ohjelmistotuotanto sulautettuihin järjestelmiin ja mobiiliympäristöön ei välttämättä nykyisin kuitenkaan poikkea toisistaan kovinkaan paljon, sillä esimerkiksi mikrokontrollereissa voi nykyisin olla käyttöjärjestelmä piiriin integroituna, ja se puolestaan tarjoaa ohjelmoijalle rajapinnan piirille ominaisten toimintojen suorittamiseksi.

2.2.1 Fragmentaatio

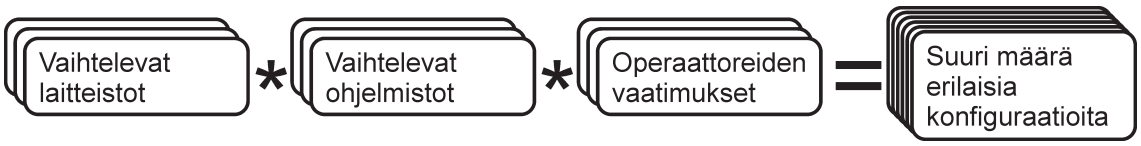
Tämä luku perustuu pääosin lähteeseen [13].

Mobiililaitteiden fragmentaatio, eli ominaisuuksien eroavuus laitteiden välillä, juontaa juurensa kuluttajien toiveista. Koska ihmiset toivovat mobiililaitteilta erilaisia ominaisuuksia, kuten mm. ulkomuoto, tiedonsyöttövälineet, valokuvausmahdollisuudet sekä laitteen sisältämät sovellukset, ovat valmistajat vastanneet markkinoiden tarpeeseen tuottamalla laajan valikoiman erilaisia mobiililaitteita.

Laaja mobiililaitteiden kirjo on lisännyt laitteiden myyntiä, mutta on se aiheuttanut myös ongelmia ratkottavaksi. Mobiililaitefragmentaatio onkin yksi haastavimmista mobiilisovelluskehityksessä ilmenevistä ongelmista. Muun muassa laitevalmistajia, laitealustoja, käyttöjärjestelmiä, tiedonsyöttövälineitä ja -tapoja, laitteiden käyttämiä natiivikieliä ja aakkostoja, paikallisasetuksia, verkko-operaattoreiden vaatimuksia sekä erilaisia mobiililaitestandardeja on todella suuri määrä (ks. kuvat 2.2 ja 2.3). Lisäksi mobiililaitekehityksen nuoren iän vuoksi erilaisia standardeja luodaan erittäin nopealla tahdilla lisää. Sovelluskehittäjän tulisikin pystyä toteuttamaan mobiilisovelluksensa siten, että se olisi siirrettävissä mahdollisimman monelle laitteelle ilman hyötyyn nähden liian suurta rahallista ja ajallista panostusta. Nykyisin mobiililaitteiden suuren fragmentaation myötä joudutaan tarkoin määrittelemään ne laitteet, joissa kehitettävää sovellusta on tarkoitus suorittaa. Tämän rajauksen jälkeen sovellus suunnitellaan ja toteutetaan siten, että sen vaatimat resurssit riittävät kaikissa kohdelaitteissa, mikä käytännössä tarkoittaa heikkoresurssisimman kohdelaitteen resursseja. Pulmaksi tässä toimintatavassa tulee se, ettei kehittyneempien mobiililaitteiden kaikkia ominaisuuksia hyödynnetä, vaikka niistä voisi olla sovelluksen kannalta hyötyä esimerkiksi käytettävyyden parantamisessa.



Kuva 2.2: Mobiililaitteen ja Java ME -ympäristön fragmentaatiota aiheuttavat osa-alueet



Kuva 2.3: Mobiililaitteiden fragmentaatiokonfiguraatiot

Mobiililaitteiden fyysisisissä ominaisuuksissa on useita eri osa-alueita, jotka vaikuttavat kuinka laitteessa suoritettava sovellus toimii. Näitä ominaisuuksia ei ole kyetty standardoimaan edes de facto -tasolla, koska mobiililaitesektorilla on useita keskenään kilpailevia valmistajia, jotka ottavat uusia teknologioita otetaan käyttöön erittäin nopealla tahdilla.

Piiritasolla fyysisiin ominaisuuksiin vaikuttavat mm. laitteen muistin koko ja nopeus, prosessorin tehokkuus, laitteeseen sisällytettävät kiihdytinpiirit, näyttö sekä syöttölaitteet. Usein nämä piiritason ominaisuuksiin vaikuttavat valinnat tehdään taloudellisten sekä teknologiaan perustuvien syiden pohjalta.

Mobiililaitteissa käytetään erilaisten piiritason ratkaisuiden ohella myös suurta määrää erilaisia käyttöjärjestelmiä, joita ovat mm. Nokia OS, Symbian OS, Linux ja Windows CE. Suurin syy eri valmistajien omien käyttöjärjestelmien käyttämiseen on käyttöjärjestelmän ja käytetyn laitteiston yhteistoiminnan ongelmien riskin pienentäminen. Lisäksi laitteistojen kehittyessä valmistajat voivat tarpeen mukaan muokata käyttöjärjestelmäänsä uusien vaatimusten mukaisiksi.

Käyttöjärjestelmä tarjoaa sovelluksille rajapinnan laitteiston käyttämiseksi. Mobiililaitteen Java-virtuaalikone abstrahoi suurimman osan laitteen toiminnoista, kuten mm. erilaisien yhteysmuotojen käytön, Java ME -sovelluksille. Nämä toiminnot ovat kuitenkin vain niin luotettavia ja oikeellisia kuin käytetty käyttöjärjestelmän tarjoama toiminnallisuus. Sovelluskehittäjän tuleekin olla tietoinen mahdollisista mobiililaitteen käyttöjärjestelmässä ilmenevistä puutteista ja vioista sovellusta kehittäessään.

Java-virtuaalikoneen toiminnallisuuden määrittelevä standardi on valitettavasti jossain määrin tulkinnanvarainen, ja täten eri valmistajat toteuttavat Java-virtuaalikoneensa hiukan eri tavalla. Tämän lisäksi virtuaalikoneista tuotetaan eri versioita, jotka kattavat eri ominaisuuksia. Näin Java-virtuaalikoneista, joiden tehtävä on suorittaa Java-sovellusta määrätyllä tavalla käytetystä ympäristöstä riippumatta, on olemassa suuri määrä hiukan eri tavalla toimivia versioita, mikä puolestaan lisää mobiililaitteissa ilmenevää fragmentaatiota.

Mobiililaitefragmentaatio lisääntyy myös mobiilioperaattoreiden vuoksi. Operaattorit muokkaavat usein tarjoamiaan tuotteita markkinoiden vaatimusten mukaan. Operaattorit lisäävät mobiililaitteisiin toiminnallisuutta, joilla he tarjoavat asiakkailleen

Taulukko 2.1: Esimerkkejä kännyköiden Java- ja näyttöominaisuuksien eroista

laitevalmistaja	**malli**	**Java-ominaisuudet**	**näytön resoluutio ja värimäärä**
Motorola	A388	MIDP 1.0, CLDC 1.0	240*320 pikseliä, 2 väriä (1 bit)
Motorola	A835	MIDP 1.0, CLDC 1.0	176*220 pikseliä, 65536 väriä (16 bit)
Nokia	5140	Nokia UI API, JTWI 1.0, MIDP 2.0, CLDC 1.1, MMAPI 1.0, WMA 1.0	128*128 pikseliä, 4096 väriä (12 bit)
Nokia	6270	MIDP 2.0, CLDC 1.1, MMAPI 1.0, WMA 1.0, M3GAPI 1.0, FileConnection & PIM API, Nokia UI API, Mobile 3D Graphics API, Bluetooth API, JTWI 1.0	320*240 pikseliä, 262144 väriä (18 bit)
Sony-Ericsson	P910i	MIDP 2.0, CLDC 1.0, JTWI 1.0, PersonalJava 1.1.1, Bluetooth API, WMA 1.1	208*320 pikseliä, 26244 väriä (18 bit)

erilaisia palveluita. Sovelluskehittäjän tulee olla tietoinen myös näistä operaattorikohtaisista eroista mobiilisovellusta kehittäessään.

Sovelluskehittäjän näkökulmasta helpoin tapa ratkaista fragmentaatiosta aiheutuvia ongelmia on rajoittaa sovelluksen tukemia laitteita ja operaattoreita. Ratkaisun haittapuolena on luonnollisesti sovelluksen pienempi osuus sovellusmarkkinoilla. Tämä tredi on kuitenkin nähtävillä jo nykypäivänä, ja osa Java ME -sovelluksista on toteutettu vain suosituimmilla mobiililaitemalleilla toimiviksi. Fragmentaatio-ongelmia voidaan ratkaista myös luomalla useita eri mobiililaitteilla toimivia versioita sovelluksesta manuaalisesti, käyttäen kehittyneitä sovelluskehitystyökaluja tai ulkoistaa eri versioiden tuottaminen kolmannen osapuolen vastuulle.

Tällä hetkellä toteutettavat mobiilisovellukset joudutaan fragmentaation vuoksi testaamaan jokaisella kohdelaitteella erikseen. Mahdollisesti löydetyt fragmentaation ilmentymät, eli laitealustasta riippuvat erilaisuudet, jotka johtavat usein kehitettävän

sovelluksen virhetilanteisiin, joudutaan kiertämään laitekohtaisella räätälöidyllä lähdekoodilla. Tämä lisää sovelluskehitykseen vaadittavaa aikaa ja kuluja. Lisäksi joudutaan ylläpitämään tietämyskantaa käytettyjen laitteiden standardeista poikkeavista toiminnoista sekä mahdollisista virheenkiertotavoista, mikäli sellaisia on löydetty. [8]

2.2.2 Käyttömukavuus

Työasemaympäristössä käyttäjät ovat tottuneet sovellusten pieneen tahmaisuuteen ja epämukavuuteen ohjelmistoja käyttäessään. Mobiiliympäristössä asia on toinen: ihmiset vertaavat mobiililaitteiden luotettavuutta ja toimintaa puhelimeen, jonka on perinteisesti odotettu toimivan nopeasti ja virheettömästi. Lisäksi mobiililaitteen tehtävä on mahdollistaa tiettyjen toimintojen suorittaminen aina käyttäjän niin halutessa. Tällöin lyhytkin odotusaika tuntuu tuskastuttavalta. Mobiilisovelluksen suhteellinen nopeus on tärkeää myös siksi, että monet tiedonsiirtorajapintoihin ja protokolliin liittyvät asiat saatetaan joutua hoitamaan ohjelmallisesti. Näissä tapauksissa sovelluksen liiallinen hitaus saattaa johtaa esimerkiksi tietoliikenneyhteyden katkeamiseen tai virheelliseen tiedonsiirtoon. Työasemilla tällaisia todella nopeaa toimintaa vaativia tehtäviä on varsin harvoin tai ne toteutetaan laitteistotasolla.

Mobiilisovelluskehityksessä tulee ottaa huomioon myös varsin rajallinen käytettävyys. Yleensä mobiililaitteen näyttö on pieni ja matalaresoluutioinen eikä sovelluksen eri näkymiin voida sijoittaa kovinkaan montaa elementtiä. Myös mobiililaitteen näppäimistö tai muu syöttölaite on laitteen koon rajoittamana varsin pieni. Lisäksi usein näppäimistön näppäimiin on sisällytetty useampia toimintoja kuten esimerkiksi monen eri kirjaimen, numeron tai erikoismerkin syöttötoiminnot. Sovellusta suunniteltaessa onkin pyrittävä siihen, että monimutkaisenkin sovelluksen käyttäminen olisi riittävän helppoa ja nopeaa, jotta käyttäjä ei turhaudu sovellusta käyttäessään.

Käyttömukavuuteen vaikuttavat myös sovellusten käyttöympäristö. Erilaisissa kulttuuriympäristöissä tietyt värit, sanamuodot ja kuvakkeet voivat tarkoittaa eri asioita. Mobiililaitteiden liikkuvuus tulee ottaa huomioon siten, että minimoidaan mahdollisuus, että sovelluksen ulkoasu loukkaisi tai ärsyttäisi käyttäjiä. [19]

2.2.3 Muistinkulutus

Yksi tärkeimmistä asioista mobiilisovellusta suunniteltaessa ja toteutettaessa on muistinkulutuksen minimointi, koska mobiililaitteiden käyttö- ja tallennusmuistin määrä on yleensä varsin rajallinen, johtuen sen korkeasta valmistuskustannuksesta. Lisäksi koska muistipiirit vaikuttavat haitallisesti myös mobiililaitteen energiankulutukseen, on muistia usein käytössä suhteellisen vähän. Mobiilisovelluksen suunnittelussa onkin

syytä kiinnittää huomiota muistinkulutukseen varhaisessa vaiheessa, sillä myöhemmin muistinkulutuksen pienentäminen voi olla vaikeaa ja johtaa tuotekehitysajan pitenemiseen ja joustavuuden heikkenemiseen.

Kaikissa sovelluksissa varattujen muistiresurssien vapauttaminen on erittäin tärkeää. Sovelluksen virhe muistinhallinnassa aiheuttaa niin sanotun muistivuodon ja pidemmällä aikavälillä ohjelmaa suorittavan laitteen muistiresurssit on varattu. Mobiiliympäristössä muistinhallintaan tulee kiinnittää työasemaympäristöä enemmän huomiota myös sen vuoksi, että mobiililaitteet sammutetaan usein huomattavasti työasematietokoneita harvemmin.

Mobiilisovelluksissa saatetaan joutua optimoimaan muistinkäyttöä usealla eri tavalla, mitä ei työasemaympäristössä tarvitsisi edes harkita. Tällainen tapaus on esimerkiksi sovelluksen sisäisten tietorakenteiden tiivistäminen optimoimalla tietorakenteita halutuille laitteille sopivilla tietotyypeillä.

Usein paljon käyttömuistia vaativat sovellukset ovat fyysiseltä kooltaankin suuria. Suurien sovellusten käynnistäminen heikkotehoisilla mobiililaitteilla tuottaa usein käynnistysviivettä, mikä vaikuttaa käyttömukavuuteen. Mobiilisovellukset ladataan mobiililaitteeseen usein erilaisten verkkoyhteyksien kautta, joiden käyttö ei välttämättä ole ilmaista. Sovelluksen lataamisesta aiheutuneet kulut saattavat rajoittaa asiakaskuntaa.

2.2.4 Laskentateho

Mobiililaitteissa on huomattavasti vähemmän laskentatehoa työasematietokoneisiin verrattuna. Tämä tulee myös ottaa huomioon mobiilisovellusta toteutettaessa. Suorituskyvyn parantaminen saattaa kuitenkin aiheuttaa suuremman muistin- ja energiankulutuksen. Tästä syystä mobiilisovelluksen suunnittelu ja toteutus sisältää monimutkaisia optimointitehtäviä, joilla etsitään sopiva energian- ja muistinkulutuksen sekä suorituskyvyn välinen suhde. Mobiililaitteissa joudutaan usein hyödyntämään ns. viivästettyä tai aikaistettua laskentaa. Aikaistetulla laskennalla tarkoitetaan sitä, että sovellus pyrkii ennakoimaan käyttäjän toimenpiteitä ja suorittamaan mahdollisesti prosessointitehoa vaativia toimenpiteitä etukäteen. Viivästetty laskenta toimii päinvastoin: tällöin sovellus suorittaa toimintaa vasta käyttäjän toimenpiteen jälkeen. Mobiilisovellussuunnittelijan täytyykin ottaa huomioon, mitkä toiminnot täytyy tai kannattaa toteuttaa aikaistetulla laskennalla ja mille toiminnoille riittää viivästetty laskenta.

Sovelluksen suorituskykyä on varsin vaikea arvioida ennen kuin itse ohjelma on tehty. Tämän vuoksi on hyvä tehdä riittävän usein proto-versioita tuotettavasta sovelluksesta ja testata niitä eri emulaattoreilla ja laitteistoilla. Suorituskyvyn ongelmat voivat aiheuttaa mm. käytettävyysongelmia ja erilaisten verkkoyhteyksien katkeilua.

Emulaattoreihin tukeutuvaan testaukseen ei kuitenkaan tule kuitenkaan täysin luottaa, sillä niiden tarjoama ympäristö ei välttämättä ole täysin samanlainen määritellyn kohdelaitteen kanssa. Eroja voi ilmetä esimerkiksi sovellukselle tarjotun vapaan muistin määrässä, suoritustehossa sekä muissa ominaisuuksissa kuten yhteyskäytänteissä sekä grafiikka- ja äänentoisto-ominaisuuksissa.

2.2.5 Energiankulutus

Energiankulutus on asia, jota työasemaympäristössä ei yleensä tarvitse ottaa lainkaan huomioon. Mobiilisovelluskehityksessä tämä asia on kuitenkin erittäin tärkeä ja energiankulutusta pyritään pienentämään kaikin mahdollisin keinoin siten, että laitetta pystytään käyttämään mahdollisimman kauan. Energiankulutuksen säätömekanismeja ovat mm. laitteen asettaminen virransäästötilaan, näytön taustavalaistuksen sammuttaminen ja vaativien operaatioiden määrään vähentäminen. [1]

Yksinkertaisimmillaan mobiililaitteen energiankulutusta voi verrata laitteen suorittamien komentojen energiankulutukseen. Tällöin energiankulutusta voidaan simuloida, mutta tällainen yksinkertainen arviointi ei yleensä ole tarkoituksenmukaista. Usein mobiililaitteisiin on sisällytetty omia energiansäästörutiineja. Mobiilisovellusta suunniteltaessa nämä rutiinit tulee ottaa huomioon, koska esimerkiksi mobiililaitteen näytön animointi jonkin toiminnon elävöittämiseksi saattaa osoittautua epäkäytännölliseksi, sillä se voi estää laitteen sisäiset energiansäästötoiminnot kokonaan.

Erilaisten yhteysmuotojen, kuten esimerkiksi GPRS, Bluetooth, infrapuna ja WLAN, käyttö vaikuttaa varsin paljon laitteen yhteisenergiankulutukseen. Yleisenä peukalosääntönä mobiilisovelluksen energiankulutuksen pienennykseksi on kaiken hyödyttömän toiminnan pysäyttäminen ja laitteen siirtyminen ei-aktiiviseen tilaan mikäli laite ei ole suorittanut mitään toimintoa määrätyn ajan kuluessa. Aivan välittömästi laitteen energiansäästörutiineja ei kannata suorittaa, koska sen herättäminen takaisin aktiiviseen tilaan on usein hidasta ja energiaa kuluttavampaa kuin pienen hetken odottaminen aktiivisessa tilassa.

Mobiilisovellusta kehitettäessä on myös huomioitava mahdollinen sovelluksen suorituksen hallitsematon päättyminen mobiililaitteen virran loppuessa. Sovelluksen tulisi pyrkiä mahdollisuuksien rajoissa ennakoimaan virran vähyys ja tallentaa tarpeen vaatiessa tiedot sovelluksen tilasta pysyväismuistiin ennen sovelluksen suorittamisen äkillistä päättymistä.

2.2.6 Rajapinnat

Aiemmin mobiililaitteiden ohjelmointirajapinnat Java ME -ympäristössä ovat olleet varsin rajalliset, mutta parin viime vuoden aikana uusia ominaisuuksia tukevia rajapintoja on kehitetty ja standardoitu erittäin nopealla tahdilla. Nopea standardoituminen on saattanut jättää standardeihin virheitä ja puutteita, jotka vaikeuttavat sovelluskehittäjän työtä esimerkiksi rajapintapuutteiden tai -muutosten muodossa. Myös nopeasti lisääntyvien lisälaitteiden puutteellinen standardoituminen ilmenee näitä tukevien standardirajapintojen puuteena. Ongelmia sovelluskehittäjille saattavat aiheuttaa sovelluksille asetetut vaatimukset, jotka voivat joillain mobiililaitteilla vaatia valmistajan tarjoamien Java-standardia noudattamattomien ohjelmointirajapintojen käyttöä. Näissä tapauksissa epästandardit ohjelman osat joudutaan räätälöimään laitekohtaisesti halutuille laitteille. Laitevalmistajat eivät tarkoituspohjaisesti yritä edistää fragmentaatiota luomalla omia epästandardeja rajapintoja, vaan syynä näihin on laitteiden tarjoamien ominaisuuksien erittäin nopea kehittyminen.

2.2.7 Ylläpidettävyys

Sovelluksen ylläpidettävyydellä tarkoitetaan sen helppoa muokkautuvuutta mahdollisia tulevaisuuden tarpeita varten. Ylläpidettävyys tulee huomioida suunniteltaessa sovelluksia sekä mobiili- että työpöytäympäristöön. Mobiiliympäristössä tämä tulee kuitenkin huomioida työpöytäympäristöä kattavammin, koska mobiililaitteiden teknologiat kehittyvät erittäin nopeasti ja uusia standardeja kehitetään koko ajan. Mobiilisovellusohjelmoija joutuukin ottamaan huomioon tämänhetkiset ja mahdollisesti tulevat vaatimukset. Uusien ominaisuuksien tuki perustuu luonnollisesti jonkin verran arvailuiden varaan, vaikka laitevalmistajat pyrkivätkin antamaan tietoa tulevista tuotteistaan etukäteen.

Valitettavan usein mobiilisovellusten ylläpidettävyydestä joudutaan tinkimään resurssien puutteen vuoksi. Useimmiten ylläpidettävämpi lähdekoodi vaatii enemmän mobiililaitteen resursseja, jotka yleensä tarvitaan varsinaisen sovelluksen toiminnallisuuden suorittamiseen eikä ylläpidettävyyden parantamiseen.

2.2.8 Tietoturva

Mobiililaitteiden tietoturvaseikkoihin on paneuduttu sekä laitteiden että niiden ohjelmistojen suunnittelussa varsin paljon. Koska nykyisin on varsin helppoa ja nopeaa ladata erilaisia mobiilisovelluksia tai niiden osia missä ja milloin tahansa, on erilaisin varmennuskeinoin haluttu varmistua, etteivät laitteeseen asennettavat sovellukset

aiheuta tietoturvariskejä. Esimerkkinä tällaisesta varennuskeinosta ovat Symbian OS -järjestelmälle tuotettujen sovellusten varmenteet. Jotta sovellus olisi asennettavassa muodossa, täytyy se testauttaa riippumattomalla testausyrityksellä, joka varmentaa sovelluksen vaarattoman toiminnan. Ilman tätä yrityksen tarjoamaa varmennetta sovellusta ei voida asentaa Symbian OS -järjestelmällä varustettuun mobiililaitteeseen. Erilaisia varmennetekniikoita, kuten Java ME -ympäristössä käytettyä esivarmennusta (preverifying) ja Symbian OS -järjestelmän yhteydessä käytettyä varmennetekniikkaa, hyödynnetään myös muissa ympäristöissä, joita ovat mm. Microsoft Windows Mobilen käyttämä sisällönvarmennustekniikka (authenticated content signing). Lisäksi useat eri yritykset tarjoavat omia mobiililaitteiden tietoturvaa parantavia sovelluksia, joita ovat esimerkiksi virustorjuntasovellukset, tietojen salaussovellukset ja muut etähallinta- ja seurantasovellukset. [16, 24, 35]

Mobiililaitteiden tietoturvaan vaikuttavat myös käytetyt ohjelmointikielet ja niiden käyttämät tekniikat ja ominaisuudet. Esimerkiksi Java ME -ympäristössä kielestä on joissain tilanteissa karsittu mahdollisille mobiililaiteympäristössä tietoturvaongelmille altistavia elementtejä, jotka työpöytäympäristössä Java-kielestä löytyvät. Nämä elementit ovat: Java Native Interface (JNI), käyttäjän määrittelemät luokkalataajat, säieryhmät sekä reflektointi. Syy näiden elementtien karsimiseen on, etteivät heikkotehoiset mobiililaitteet nykyisin usein kykene suorittamaan kaikkia tietoturvan takaamiseksi vaadittuja toimenpiteitä, vaikka mainitut tekniikat takaavatkin tietoturvan oikeinkäytettynä Java SE -ympäristössä. [4]

Mobiilisovelluksen mahdollisesti tallettamien tietojen tietoturva tulee myös huomioida. Tärkeät ja salaiset tiedot tulisi tallentaa salatussa muodossa, sillä pienet ja kevyet mobiililaitteet voivat kadota tai joutua varastetuksi varsin helposti.

2.2.9 Testaus

Mobiilisovellusten testausta voidaan suorittaa erilaisilla mobiililaite-emulaattoreilla sekä varsinaisilla mobiililaitteilla. Emulaattoreilla testausta voidaan suorittaa sovellusta kehitettäessä, mutta niiden kattavuuteen ei voida luottaa toteutettavan sovelluksen järjestelmä- ja hyväksymistestauksessa. Emulaattorit eivät välttämättä tue kaikkia käytettäviä ominaisuuksia. Lisäksi niillä ei usein pystytä testaamaan sovelluksen käyttäytymistä reaalimaailmaa vastaavalla tavalla, kuten esimerkiksi sovelluksen toimintaa eri nopeuksisilla laitteilla tai erilaisten yhteyskäytäntöjen, kuten Bluetooth tai WLAN, toimintaa. Lisäksi sovelluksen vaatimusten mukaista toimintaa ei voida emulaattoreilla taata, sillä esimerkiksi fragmentaatiota aiheuttavat mobiililaitteiden järjestelmissä olevat virheet vaikuttavat usein sovelluksen toimintaan. Esimerkiksi Java ME -pohjaista

Opera Mini -www-selainta toteuttaessa Opera Software totesi, että mobiililaitteiden fragmentaation vuoksi selainsovelluksen toiminta jouduttiin käytännössä testaamaan jokaisella halutulla kohdelaitteella erikseen, jotta voitiin varmistua sen toimivuudesta vaadituissa laitteissa [8].

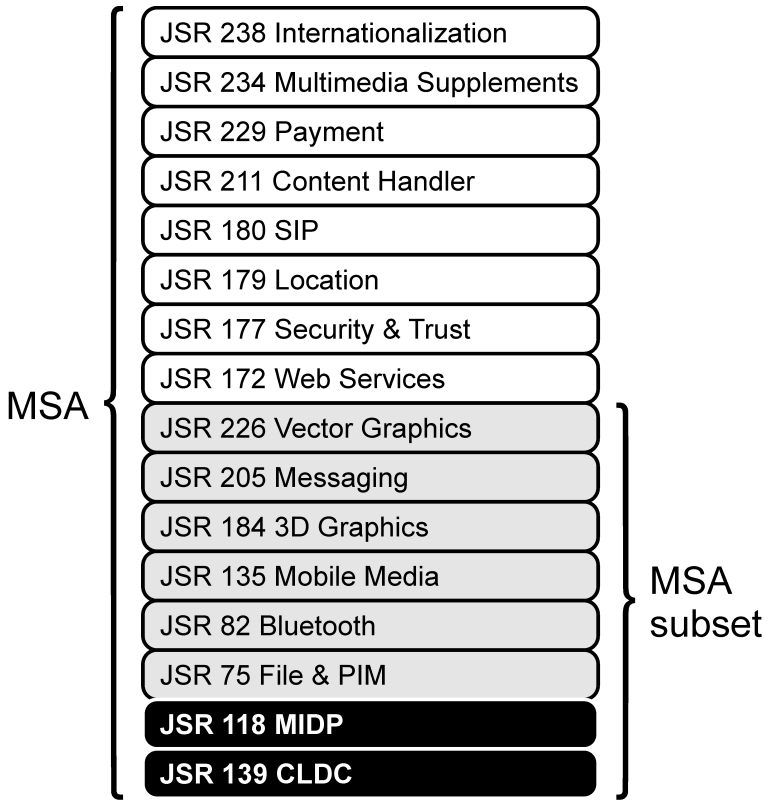
Nykyisin mobiilisovelluksen testaus voidaan suorittaa sovellusta kehittävän yrityksen toimesta, mutta tarjolla on myös useita mobiilisovellusten testaukseen erikoistuneita yrityksiä kuten Babel, Gapgemini, NSTL ja RelQ. Testaukseen erikoistuneiden yritysten palveluita käytettäessä kehitetty sovellus lähetetään testausta suorittavalle yritykselle, joka palauttaa sovellusta kehittävälle osapuolelle kattavan testausraportin ja sovelluksen läpäistyä testit hyväksyntätodistuksen. [32]

2.3 Mobiilisovelluskehityksen tulevaisuudennäkymät

Mobiililaitekannan fragmentaatio tulee varsin todennäköisesti kasvamaan sitä mukaa kun laitevalmistajien ja niiden toteuttamien laitemallien määrä kasvaa. Lisäksi käytössä olevien mobiililaitteiden ohjelmistot monimutkaistuvat, joten ne tulevat luultavasti sisältämään entistä enemmän virheitä. Hyvin määritellyt standardit luonnollisesti ehkäisevät fragmentaatiota, mutta jo tähän päivään mennessä on huomattu, etteivät ne täysin ehkäise sitä. Fragmentaatiota aiheuttavat laitevalmistajien itse kehittämät epästandardit laitteiden laajennokset ja ohjelmointirajapinnat, standarditoimintoihin jääneet virheet sekä laitteiden ominaisuuksista johtuvat erot. Fragmentaation kasvusta aiheutuvat kulut kasvavat ja sovellusten kehitys kattavalle laitekirjolle hyvin testattuna hidastuu, mikäli tähän ongelmaan ei kehitetä tehokkaita pitkälle automatisoituja ratkaisuja.

Lähitulevaisuudessa mobiililaitteisiin tullaan kehittämään dynaamisia komponenttienhallintajärjestelmiä, jotka mahdollistavat mm. itsenäisten komponenttien asentamisen mobiililaitteisiin. Nämä komponentit voivat olla käyttöliittymättömiä palveluita (services), sovelluksia (applications) tai jaettuja kirjastoja (shared libraries). Uusimmissa Symbian OS -järjestelmän versioissa näitä ominaisuuksia on jo toteutettu, mutta esimerkiksi Java-ympäristössä vaaditaan tällä saralla vielä kehitystä [33]. Tämä uudistus muuttaa nykyisen monoliittisen mobiiliympäristön helpommin mukautettavaksi komponenttipohjaiseksi ympäristöksi, johon työasemaympäristössä on jo totuttu. Tämä uudistus mahdollistaa sovelluksien osien kattavamman ja helpomman uudelleenkäytön jaettujen, ja mahdollisesti myös laitteelle räätälöityjen, kirjastojen muodossa, ja ratkaisee näin osittain fragmentaatiosta aiheutuvia ongelmia. Lisäksi tarvittaessa voidaan tuottaa tarvittavia palveluprosesseja, jotka laajentavat sovelluskehittäjien mahdollisuuksia tuottaa mobiililaitteeseen helposti erilaisia lisäpalveluita. [28, 31, 30]

Myös laitevalmistajat pyrkivät yhdessä ehkäisemään fragmentaatiota. JSR 185 *Java^{TM} Technology for the Wireless Industry* (JTWI) -määritystä käytetään nykyisin markkinoilla olevien Java-mobiilisovellusten suoritusympäristönä. Laitteiden ominaisuudet tulevat kuitenkin monipuolistumaan ja tehokkuus kasvamaan ja näiden laitteiden fragmentaation kasvua ehkäistääkseen ovat useat laitevalmistajat kehittämässä yhteistyössä Java ME -ympäristöön JTWI:n pohjalta JSR 248: *Mobile Service Architecture* (MSA) -määritystä. MSA yhdistää useita käytössä olevia, kuitenkin erillisinä tarjottuja, Java-komponentteja yhdeksi määritellyksi kokoelmaksi (ks. kuva 2.4). MSA-määritystä kehittyneempiä mobiililaitteita varten ollaan kehittämässä myös JSR 249: *Mobile Service Architecture Advanced* -määritystä, joka koostaa MSA:ta kattavammin erilaisia Java-komponentteja yhdeksi määrittelyksi. MSAA, MSA ja sen osa MSA subset määrittelevät mobiililaitteille Java-ympäristön ominaisuudet sekä sen yhteensopivuusvaatimukset, joita palveluiden ja sisällöntuottajat voivat käyttää ohjenuorana mobiilisovelluskehityksessä. [29, 30]



Kuva 2.4: Mobile Service Architecture

3 Abstraktiotasot sovelluskehityksessä

Tässä luvussa määrittelen mitä abstraktiotasolla tarkoitetaan ja miten eri abstraktiotasoja hyödynnetään ohjelmistotuotannossa.

3.1 Abstraktiotaso

Abstraktiotasoilla tarkoitetaan yleisesti useita eri tarkkuusasteen määrityksiä tietystä ongelmasta siten, että matalamman abstraktiotason määrityksissä on yksityiskohtien ratkaisutavat esitetty tarkemmin kuin korkeamman abstraktiotason määrityksissä. Nämä abstraktiotasot ovat läsnä ohjelmistotuotannossa niin suunnitteludokumenteissa kuin myös toteutetussa lähdekoodissa.

Sovelluskehityksessä on monta eri toteutusvaihetta, joiden aikana tarvitaan erilaisia abstraktiotasoja. Sovelluskehitysprojekti alkaa määrittelyllä ja suunnittelulla, joiden aikana toteutetaan suunnitteludokumentteja. Näiden dokumenttien tarkoitus on kuvata toteutettava sovellus tai järjestelmä riittävän tarkasti, jotta sen toteutus voidaan aloittaa. Dokumenteissa ei kuitenkaan paneuduta epäoleellisiin toteutusteknisiin asioihin, vaan esitellään toteutettavan järjestelmän looginen rakenne. Näiden dokumenttien abstraktiotaso onkin korkeammalla kuin sovellusta toteuttavan henkilön tuottaman lähdekoodin abstraktiotaso. Yleisesti korkean abstraktiotason merkkaukset kuvaavatkin selkeämmin tuotettavan järjestelmän tai sovelluksen yleisen rakenteen ja pääpiirteisen toiminnan kuin matalan abstraktiotason dokumentit.

Myös erilaiset ohjelmointikielet ja niiden erilaiset toteutustekniset asiat, kuten eritasoiset kirjastokutsut, määrittelevät useita eri abstraktiotasoja. Suunnittelussa ja toteutuksessa ilmenevät abstraktiotasot on kuitenkin syytä pitää jossain määrin erillään, sillä suunnitteludokumenteissa ei ole tarkoituskaan määritellä toteutusteknisiä asioita liian tarkasti.

Tässä tutkielmassa keskityn tästä lähtien tarkemmin abstraktiotasoihin vain ohjelmoinnin ja lähdekoodin näkökulmasta.

3.2 Lähdekoodin abstrahointi

Lähdekoodin abstrahoinnilla sovellusta kehitettäessä tarkoitetaan määritellyn ongelman ratkaisun yksityiskohtien piilottamista tietyn rajapinnan taakse. Näitä rajapintoja edustavat useat erilaiset abstraktiotasot, joita määrittelevät toteutustekniset asiat, kuten käytetty ohjelmointikieli ja siihen liittyvät työkalut sekä lähdekoodin rakenne.

Ohjelmistoja toteutetaan nykyisin käyttämällä jotain korkean abstraktiotason ohjelmointikieltä, mikä parantaa huomattavasti sovelluskehittäjän tuottavuutta, sillä abstraktiotason nosto vähentää tuotettujen virheiden määrää eikä ohjelmoijan tarvitse muistaa abstraktiotason piilottamia ohjelmointiteknisiä yksityiskohtia. Lähdekoodin perusabstraktiotason määrittelee käytetty ohjelmointikieli, mutta sovelluskehittäjä voi kohottaa abstraktiotasoa toteuttamalla aliohjelmia ja -metodeja, jotka toteuttavat tietyn toiminnallisuuden. Lisäksi käytössä olevat tai itse kehitetyt toiminnallisuutta tarjoavat luokat, komponentit ja kirjastot tarjoavat useita eri abstraktiotasojen ohjelmointirajapintoja, jotka oikein käytettynä kohottavat kehitettävän lähdekoodin abstraktiotasoa.

Järkevästi käytetyllä korkean abstraktiotason ohjelmointikielellä toteutettu lähdekoodi, jossa on hyödynnetty muita abstraktiotasoa nostavia tekijöitä, kuten esimerkiksi komponentteja, on huomattavasti selkeämpää, virheettömämpää ja ymmärrettävää kuin matalan abstraktiotason ohjelmakomentoja käytettäessä. Lähdekoodin ymmärrettävyys onkin suoraan verrattavissa sovelluksen ylläpidettävyyteen. Lisäksi korkean abstraktiotason lähdekoodin voi usein kääntää eri laite- ja käyttöjärjestelmäympäristöihin käyttämällä sopivia kääntäjiä ja muita työkaluja [3]. Myös sovelluskehitysympäristö vaikuttaa lähdekoodin abstraktiotasoon, sillä nykyisin useat sovelluskehitysympäristöt tuottavat automaattisesti osan lähdekoodista nopeuttaen näin kehittäjän työtä entisestään. [23, 20]

3.3 Esimerkkejä eri abstraktiotason ohjelmointikielistä

Alhaisimman abstraktiotason kielet ovat ensimmäisen sukupolven ohjelmointikielet, joita ovat laitteiden konekielet. Konekielellä tarkoitetaan ohjelman sitä muotoa, jonka laite pystyy sellaisenaan suorittamaan. Konekielellä hyvin toteutettu sovellus suoriutuu tehtävistään niin nopeasti kuin sitä suorittavan laitteen resurssit sen sallivat. Käytännössä konekieliset ohjelmat kirjoitettiin ensin paperille symbolisesti käyttäen konekäskyistä muistikasnimiä ja sitten ohjelma käännettiin käsin kohdelaitteen konekielelle. Ensimmäisen sukupolven kielet olivat vallalla 1940-luvun lopulta 1950-luvun alkupuolelle. Nykyisin konekielistä ohjelmakoodia tuotetaan erittäin harvoin, sillä sen

ymmärrettävyys ja siirrettävyys ovat todella huonoja. Jo tässä vaiheessa sovellus voitiin jakaa osakokonaisuuksiin, kuten aliohjelmiin, joita voitiin hyödyntää toteutettaessa sovelluksen toiminnallisuutta. Aliohjelmakutsuja hyödyntäen voitiin lähdekoodin abstraktiotasoa kohottaa. [15]

Lähdekoodi 1 on esimerkki Hello World -ohjelmasta erään tietokoneen konekielellä 16-lukujärjestelmällä:

Lähdekoodi 1 Hello World -ohjelma konekielellä

```
980901013d6e18579802010098060003 6f686a656c6d612e6d6d730098070002f4
ff000000000701000000009804000398070007 48656c6c980700076f2c20579807
00076f726c6498070007210a0000980a00ff0000000000000100980b0000203a40
50104040204d20612069026e0100812053207410100272010c82000000980c0008
```

Ohjelmointikielten toinen sukupolvi syntyi, kun kehitettiin ohjelma, joka suorittaa yllämainitun käännöstyön muistikkaista konekielelle. Toinen sukupolvi koostuu siis symbolisista konekielistä. Symbolisen konekielen abstraktiotaso on hiukan konekieltä korkeammalla, ja sen ymmärrettävyys onkin huomattavasti konekieltä parempi. Symbolisen konekielen siirrettävyys on kuitenkin samaa tasoa kuin konekielisten sovellusten siirrettävyys eri laitteille. [15]

Lähdekoodiesimerkki 2 on symbolisella konekielellä eli assemblyllä toteutetusta Hello World -ohjelmasta. Esimerkissä käytetään MS-DOS-käyttöjärjestelmän keskeytyspalveluita hyödyksi:

Lähdekoodi 2 Hello World -ohjelma symbolisella konekielellä

```
title   Hello World Program                    (hello.asm)

dosseg .model small .stack 100h
.data hello_message db 'Hello, World!',0dh,0ah,'$'

.code main  proc
      mov    ax,@data
      mov    ds,ax                   ;data seg. to @data
      mov    ah,9
      mov    dx,offset hello_message
      int    21h                     ;print string
      mov    ax,4C00h
      int    21h                     ;terminate
main  endp  end   main
```

Kolmas ohjelmointikielten sukupolvi syntyi vuonna 1955, kun Fortran-kieli kehitettiin [15]. Lähes kaikki nykyisin ohjelmointikieliksi kutsumamme kielet ovat kolmannen sukupolven kieliä ja niiden abstraktiotaso on jo varsin korkealla. Esimerkiksi C-, C++- ja Java-kielillä toteutetut sovellukset voidaan tietyin rajoituksin siirtää toiseen järjestelmään ongelmitta. Näiden kielien ymmärrettävyyskin on huomattavasti korkeammalla kuin konekielen tai symbolisen konekielen. Näiden kielien abstraktiotason määräävät kielen rakenne-elementit ja niiden kirjastojen tarjoamat aliohjelmat, luokat ja metodit. Lähdekoodit 3,4 ja 5 ovat esimerkkejä C, C++ ja Java-kielillä toteutetuista Hello World-ohjelmista. [15]

Lähdekoodi 3 Hello World -ohjelma C-ohjelmointikielellä

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

Lähdekoodi 4 Hello World -ohjelma C++ -ohjelmointikielellä

```
#include <iostream.h>

int main(void) {
  cout << "Hello World!" << endl;
  return 0;
}
```

Lähdekoodi 5 Hello World -ohjelma Java-ohjelmointikielellä

```
package helloWorld;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Neljäs ja viides sukupolvi esiintyvät puheissa toisinaan, mutta niiden sisältö riippuu täysin puhujasta. Tavallisimmin neljännen tai viidennen sukupolven kielellä tarkoitetaan joko sovelluskehittimiä kuten esimerkiksi Delphi tai ohjelmoitavia sovellusohjelmia kuten Microsoft Excel. [15]

Korkeamman abstraktiotason kielillä on myös rajoituksia, joista merkittävin on aika- ja kokokriittisten sovellusten vaikea yksityiskohtainen optimointi. Optimoinnin tasoon vaikuttaa pääasiassa ohjelmointikielen kääntäjän toteutus eikä niinkään sovelluskehittäjän ratkaisut. Lisäksi korkean abstraktiotason komentoja ei usein ole optimoitu juuri haluttuun tarkoitukseen sopiviksi. Joissain tapauksissa korkean abstraktiotason ohjelmointikielten standardikomennot voivat olla myös virheellisesti toteutettu ja ne voivat aiheuttaa turhaa muistinkulutusta ja tehokkuushäviötä. Edellä mainituissa tapauksissa on joskus pakko pyrkiä mahdollisimman matalan abstraktiotason toteutukseen, jotta järjestelmän resursseja pystyttäisiin käyttämään mahdollisimman tehokkaasti. [23, 3]

3.4 Mobiilisovelluskehityksen abstraktiotasot

Mobiilisovellusten siirrettävyys erilaisten mobiililaitteiden välillä on mobiiliohjelmistotuotannossa välttämätöntä. Nykyisin ohjelmistoista joudutaan kuitenkin räätälöimään omat versionsa lähes jokaiselle mobiililaittemallille laitteiden erilaisuudesta, eli fragmentaatiosta, johtuen. Mikäli käytetyn ohjelmointikielen abstraktiotasoa nostettaisiin siten, että käytetty kieli peittäisi laitteiden väliset erot, helpottaisi ja nopeuttaisi se huomattavasti sovelluskehittäjän työtä. Esimerkiksi Java-ohjelmointikielen yksi päätavoite onkin juuri mahdollistaa saman ohjelmakoodin suorittaminen useassa eri ympäristössä periaatteella "write once, run anywhere". Valitettavasti nykyisin eri valmistajien mobiililaitteiden Java-virtuaalikoneiden (JVM, Java Virtual Machine) sekä ohjelmointirajapintojen toteutuksen virheet ovat romuttaneet jossain määrin tämän tavoitteen saavuttamisen. Luonnollisesti muut fragmentaation ilmentymät kuten esimerkiksi erilaiset tiedonsyöttötavat ja -laitteet, näytön resoluutio sekä äänentoistolaitteisto aiheuttavat tarvetta laitekohtaisesti optimoidulle lähdekoodille.

Lähdekoodin abstraktiotason nostolla pyritään helpottamaan ja nopeuttamaan toteutettavien sovellusten siirrettävyyttä sekä selkeyttämään ja tehostamaan lähdekoodia. Lisäksi abstraktiotasoa nostamalla voidaan jossain määrin valmistautua tulevaisuuden tekniikoiden hyödyntämiseen. Voidaan esimerkiksi luoda toiminnallisuutta, joka voidaan määritellä jätettäväksi vanhoista sovelluksen versioista pois, koska kyseinen toiminnallisuus ei sen hetkisissä mobiililaitteissa ole tuettu. Näistä toiminnallisuuksista esimerkkeinä ovat varsin nopeasti kehittyvät mobiililaitteiden API:t eli ohjelmointirajapinnat.

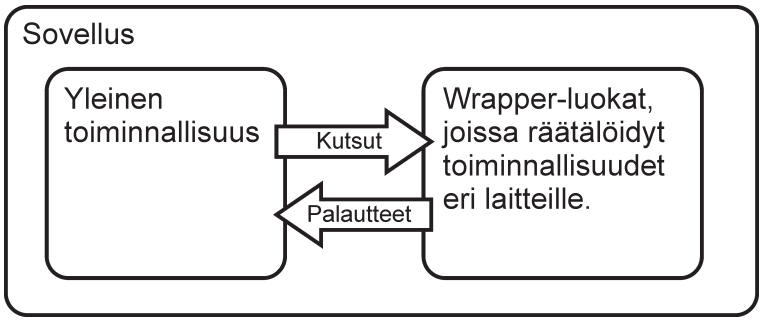
Mobiilisovelluksen lähdekoodin abstraktiotasoa määriteltäessä huomioitavia seikkoja ovat mm. käyttöliittymä, äänentoistoon ja grafiikkaan liittyvät asiat, erilaiset yhteyskäytännöt sekä yleiset osa-alueet, joiden toiminta on täysin tai lähes samanlainen riippumatta käytettävästä laitteesta. Lisäksi sovellusten aika- ja kokokriittisyys tulee ottaa huomioon.

3.4.1 Menetelmät abstraktiotason nostamiseksi

Tässä luvussa tutkin millä keinoin mobiilisovellusohjelmoinnin abstraktiotasoa voitaisiin nostaa. Myöhemmin määrittelen suosituksia sopiville mobiilisovellusten lähdekoodin abstraktiotasoille. Sopivan abstraktiotason lähdekoodilla tarkoitan lähdekoodia, joka on selkeää, tehokasta ja samalla mahdollisimman helposti siirrettävää mahdollisimman monelle mobiililaitealustalle. Mahdollisia mobiilisovellusten lähdekoodin abstraktiotason kohottamisen tapoja tällä hetkellä ovat erilaiset olio-ohjelmointipohjaiset, esikäännökseen pohjautuvat sekä erittäin korkean abstraktiotason välikieleen pohjau-

tuvat ratkaisut. Tulevaisuudessa voidaan hyödyntää myös erilaisia mobiililaitteeseen asennettavia ja suoritettavia palveluita sekä jaettuja kirjastoja abstraktiotason nostossa [28].

Eräs vaihtoehto käsitellä fragmentaatiosta aiheutuvia ongelmia on kehittää ns. middleware-komponentteja ja wrapper-luokkia, jotka yleistävät räätälöidyn toiminnallisuuden korkeammalle abstraktiotasolle. Wrapper-luokan toiminnallisuutta kutsuessa on luokalle annettu tieto halutusta laitealustasta. Kutsun aikana Wrapper-luokka suorittaa vain määrätylle laitteelle tai laiteperheelle toteutetun räätälöidyn ohjelmakoodin (ks. kuva 3.1).



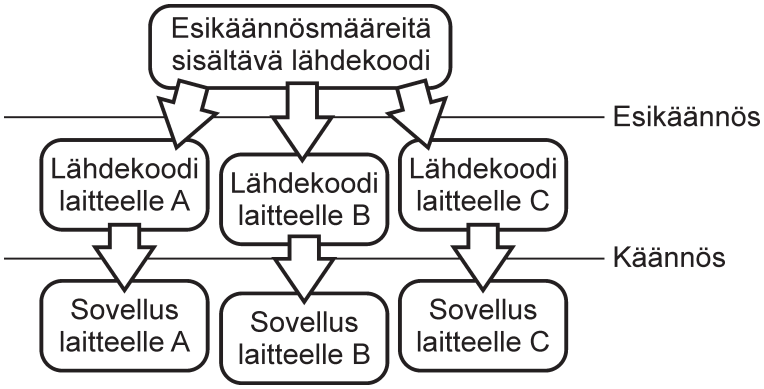
Kuva 3.1: Fragmentaatio-ongelman ratkaisumalli hyödyntäen olio-ohjelmointiteknisiä ratkaisuja

Käytännössä nämä olio-ohjelmointipohjaiset ratkaisut fragmentaation peittämiseen abstraktiotasoa nostamalla ovat kuitenkin mahdottomia toteuttaa pitkällä aikavälillä. Pienissä ja vain muutamalle mobiililaitteelle toteutettavissa sovelluksissa wrapper-luokat tai middleware-komponentit voidaan yleensä toteuttaa vaikka laitteiden resurssit ovatkin rajalliset. Ongelmia ilmenee kuitenkin, kun sovellus halutaan muokata toimivaksi useammilla eri laitekohtaista lähdekoodia vaativilla mobiililaitteilla. Tässä tapauksessa wrapper-luokkien koko kasvaa räätälöityjen toiminnallisuuksien kehittämisen vuoksi ajan kuluessa niin suureksi, etteivät nykyisten mobiililaitteiden vähäiset muistiresurssit riitä massiiviseksi kasvaneen sovelluksen tallentamiseen pysyväismuistiin tai suorittamiseen. Ongelmia syntyy myös heti, kun joku sovelluksen toiminnallisuus vaatii tietylle laitteelle määritellyn epästandardin kirjaston käyttöönottoa. Mikäli tällainen toiminnallisuus toteutetaan wrapper-luokassa, eivät laitteet, joissa mainittua kirjastoa ei ole, voi toteutettua wrapper-luokkaa käyttää.

Nykyisin mobiilisovellusten ohjelmoinnissa räätälöidyt lähdekoodilohkot ovatkin syrjäyttäneet varsin pitkälle käytetyn kielen abstraktiotason noston. Tavoitteena on lähes aina toteuttaa sovellus käyttäen erilaisia esikäännökseen pohjautuvia ratkaisuita, kuitenkin niin, että sovelluskehittäjä ohjelmoi käytännössä tarjotulla tietyn abstraktio-

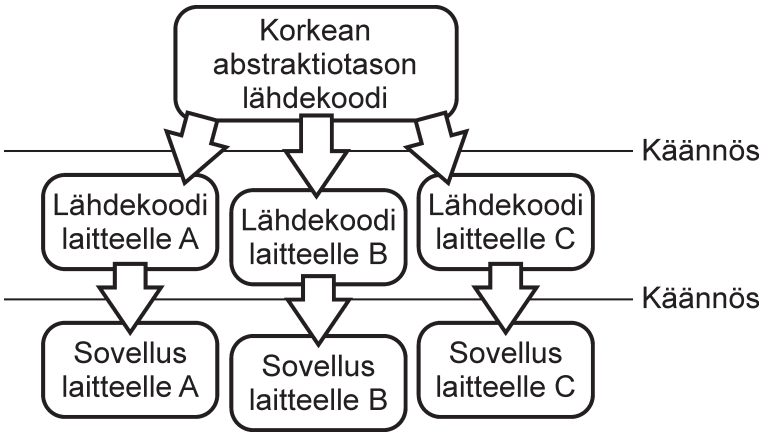
tason ohjelmointikielellä käyttäen ehdollisen esikäännöksen tarjoamia mahdollisuuksia siirrettävyyttä parantaakseen (ks. kuva 3.2). Tällöin fragmentaatio-ongelmien ilmentymät ratkaistaan kirjoittamalla jokaiselle halutulle laitteelle oma lähdekoodilohko sovelluksen ongelmakohdissa. Mikäli sovelluksen halutaan toimivan useissa eri laitteissa, sisältää se varsin paljon erilaisia esikäännösmääreitä ja if-else-rakenteita, ja lähdekoodi tulee kääntää jokaiselle kohdelaitteelle erikseen. Pidemmällä aikavälillä olisikin hyödyllisempää kehittää korkeamman abstraktiotason rakenteita lähdekoodin selkeyden ja samalla siirrettävänä säilyttämiseksi.

Nykyisin on jo olemassa erilaisia esikäännökseen pohjautuvia abstraktiotasoa nostavia järjestelmiä, kuten esimerkiksi J2ME Polish, jossa sovelluksen käyttöliittymän suunnitteluun voidaan käyttää korkean abstraktiotason määrittelyitä, joita sovellus käyttää käyttöliittymää esittäessään. Näitä, jo toteutettuja järjestelmiä, esittelen tutkielmassa myöhemmin.



Kuva 3.2: Fragmentaatio-ongelman ratkaisumalli esikäännöstä hyödyntäen

Kolmantena vaihtoehtona ovat korkeamman abstraktiotason ohjelmointikielet, jotka käännettynä generoivat laitekohtaista lähdekoodia, joka puolestaan voidaan kääntää halutulla laitteella toimivaksi sovellukseksi (ks. kuva 3.3). Tässä vaihtoehdossa ilmaisukykyisillä korkean abstraktiotason kielillä ja tehokkailla kääntäjillä voidaan saada erittäin suorituskykyistä ja räätälöityä lähdekoodia halutulle kohdelaitteelle. Periaatteessa tämä vaihtoehto toimii samoin kuin esikäännökseen perustuva ratkaisu, mutta tässä sovelluskehittäjän tulee yleensä opetella uusi korkeamman abstraktiotason ohjelmointikieli ja siihen liittyvien työkalujen käyttö.



Kuva 3.3: Fragmentaatio-ongelman ratkaisumalli korkean abstraktiotason ohjelmointikieltä ja lähdekoodin generointia hyödyntäen

Näistä abstraktiotasoa nostavista vaihtoehdoista nykyisin esikäännös ja korkean abstraktiotason ilmaisuvoimainen ohjelmointikieli ovat pitkällä aikavälillä toimivia vaihtoehtoja. Hyvin suunnitellun ja hallinnoidun esikäännöksen tai korkean abstraktiotason kielen avulla voidaan toteuttaa räätälöityjä versioita sovelluksista käytännössä kaikille vaadittaville kohdelaitteille. Korkean abstraktiotason kielen vaihtoehdon käyttössä ongelmaksi saattavat muodostua uuden ohjelmointikielen ja uusien, mahdollisesti monimutkaisten, työkalujen opettelu. Olio-ohjelmointipohjaisissa ratkaisuissa, kuten esimerkiksi wrapper-luokissa, joudutaan fragmentaation ja kohdelaitteiden lisääntyessä kirjoittamaan yhä enemmän räätälöityä lähdekoodia haluttuja laitteita varten. Nämä useille eri laitteille tehdyt laitekohtaiset lähdekoodilohkot sisällytetään sovellukseen ja sovelluksen koko kasvaa. Pidemmällä aikavälillä nämä wrapper-luokat kasvavat liian suuriksi vaikka mobiililaitteiden resurssit kasvaisivatkin.

Esitellyistä abstraktiotasoa nostavista menetelmistä pidän esikäännöstä parhaana vaihtoehtona, sillä se vaatii hyvin vähän tai ei lainkaan uuden opettelua sekä sovelluskehittäjä voi jatkaa jo olemassa olevien sovellusten kehittämistä käyttäen samaa ohjelmointikieltä, jota aiemminkin on käytetty. Lisäksi hyvin käytettynä esikäännöksellä pystytään nostamaan abstraktiotasoa mobiilisovellusohjelmoinnin alalla riittävästi peittämään fragmentaatiosta aiheutuvia ongelmia.

3.4.2 Sopivien lähdekoodin abstraktiotasojen määrittely

Tässä luvussa esittelen sovellusten eri osa-alueiden seikkoja, jotka tulee huomioida lähdekoodin abstraktiotasojen määrittelyssä. Sopivalla abstraktiotason lähdekoodilla tarkoitan lähdekoodia, joka on selkeää, tehokasta ja helposti siirrettävää useille eri mobii-

lilaitealustoille. Lähdekoodin oletusabstraktiotasona pidän Java-kielen tarjoamaa abstraktiotasoa, jonka pohjalta teen huomioiden perusteella johtopäätöksiä mahdollisesti paremmista abstraktiotasoista. Lähteenä tässä luvussa esitetyille päätelmille ovat eri mobiililaitteiden järjestelmissä ilmenneet virhetoiminnot, jotka löytyvät lähteistä [6, 34, 22].

Yleisimmin mobiililaitteiden fragmentaatio tulee sovelluskehittäjälle ilmi grafiikka-, äänentoisto- sekä tietoliikenne- ja syötteenhavaitsemisrutiineissa. Grafiikka- ja tietoliikennerutiineissa fragmentaation aiheuttajat ovat mobiililaitteiden tarjoamien toimintojen virheellisyydet sekä laitteiden graafisten ominaisuuksien erot. Näiden syiden vuoksi olisi lähdekoodin abstraktiotason nosto kyseisten toiminnallisuuksien kohdalla suositeltavaa. Virheellisestä toiminnallisuudesta esimerkkinä ovat Nokia 3510i ja 7650 puhelimissa Java-ympäristössä `Graphics.drawString()` ja `Graphics.drawSubString()` -metodeissa esiintyvä virhe, joka aiheuttaa järjestelmän kaatumisen mikäli piirrettävä merkkijono on yli 200 merkkiä pitkä. Tällaisissa virhetilanteissa tulisi lähdekoodin abstraktiotasoa nostaa siten, että merkkijonojen piirtorutiinit yleistettäisiin korkeamman asbtraktiotason komennoiksi. Näitä yleistettyjä komentoja sovelluskehittäjä voisi käyttää riippumatta käytettävästä mobiililaitteesta. Myös mobiililaitteiden graafisten ominaisuuksien eroja voidaan piilottaa sovelluskehittäjältä esimerkiksi luomalla uusia komentoja, joilla voidaan laitealustasta riippumatta saada tietoja näistä ominaisuuksista, kuten esimerkiksi näytön resoluutiosta sekä värien määrästä.

Äänentoistorutiineissa fragmentaatiota aiheuttavat pääasiassa eri mobiililaitteiden tukemat tiedostoformaatit. Näissä tapauksissa abstraktiotason nostolla ei välttämättä saavuteta kovinkaan suurta hyötyä. Tälläkin sovelluskehityksen osa-alueella voidaan kehittää korkeamman abstraktiotason komentoja, jotka mahdollistavat laitteen ääniominaisuuksien tarkastelun. Javan Multimedia API tarjoaa kuitenkin usein riittävät ja varsin virheettömät toiminnallisuudet äänentoisto sekä -nauhoitusta varten, joten yleensä abstraktiotason nosto tällä saralla ei ole tarpeellista [27].

Mobiililaitteiden syötteet voidaan nykyisin tuottaa käyttäen mitä lukuisimpia välineitä, kuten mm. valintarullia, erilaisia näppäimistöjä, kosketusnäyttöä, ääniohjausta ja muita lisälaitteita. Varsinkin näppäimistöjen kohdalla mobiililaitteiden fragmentaatio tulee ilmi. Lähes jokaisella laitevalmistajalla, ja jopa joidenkin valmistajien tuottamilla tuoteperheillä ja laitemalleilla, on erilaiset signaaliarvot eri näppäimille (ks. liite 4). Sovelluksen siirrettävyyttä ja lähdekoodin selkeyttä ajatellen olisi abstraktiotason nostaminen tässä osa-alueella erittäin suotavaa. Hyvänä vaihtoehtona olisi määritellä korkeamman abstraktiotason komentoja, joita kutsumalla saataisiin ennalta määrätyt signaaliarvot riippumatta käytetystä laitealustasta.

Erilaisten yhteyskäytäntöjen fragmentaatio aiheutuu pääosin mobiililaitteiden erilaisista ominaisuuksista. Tässä tapauksessa lähdekoodin abstraktiotason nosto ei vaikuta sovelluskehittäjän työhön millään tavalla, vaan hänen tulee ennalta tietää mitä yhteyskäytäntöominaisuuksia hän voi sovelluksessaan käyttää. Myös yhteyskäytäntöjen saralla on kuitenkin olemassa virheellisiä toteutuksia, kuten esimerkiksi useissa vanhemman ohjelmistoversion omaavissa Motorolan valmistamissa puhelimissa (mallit E770, V3, V3x, V550, V551, V620) viestien lähetyksessä vain 89% tekstiviestin sisällöstä tulee lähetetyksi [7]. Usein tällaiset virheet voidaan ehkäistä abstraktiota nostamalla.

Abstraktiotasoa voidaan myös kohottaa toteuttamalla usein käytettyjä toiminnallisuuksia korkeamman abstraktiotason komennon taakse, minkä jo 3. sukupolven ohjelmointikielet mahdollistivat funktioiden sekä metodien toteutuksina. Varsinkin toiminnallisuuksia, jotka sisältävät tietyille mobiililaitteille räätälöityä lähdekoodia, kannattaa abstrahoida selkeään ja helppokäyttöiseen ohjelmointirajapintaan.

4 Esikäännös

Esikäännös on lähdekoodin käännösprosessin ensimmäinen vaihe. Esikäännöksellä tarkoitetaan matalan tason tekstimuotoista lähdekoodin muokkaamista kuten ehdollista lähdekoodin lisäämistä, poistamista tai muuntamista [40].

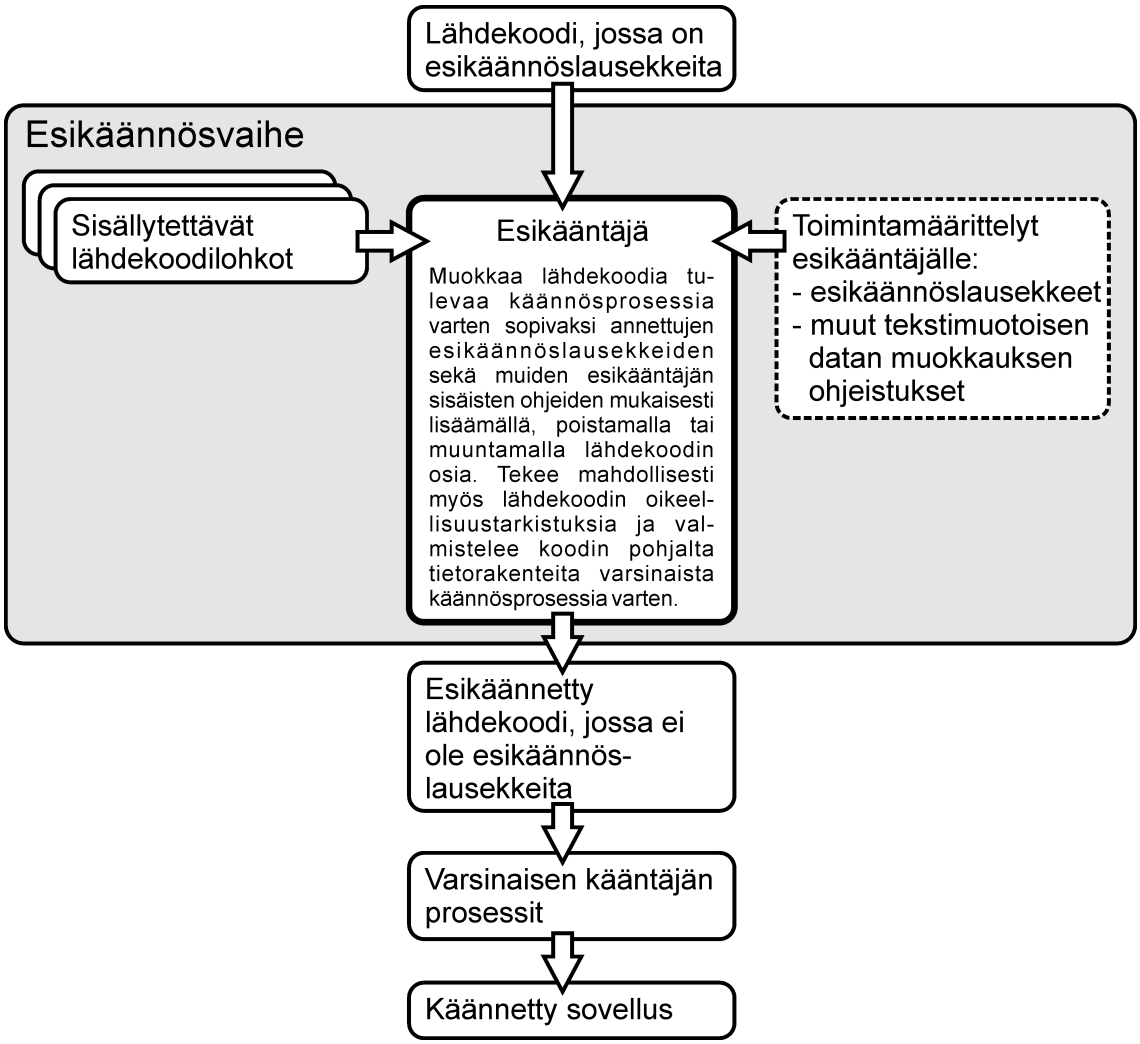
Sovelluskehittäjät ovat käyttäneet esikääntäjiä useita vuosikymmeniä ja esikäännös onkin yksi vanhimmista tavoista nostaa lähdekoodin abstraktiotasoa ja täydentää puutteellisen ohjelmointikielen ominaisuuksia. Esikäännöksellä voidaan parantaa sovellukselle edullisia toiminnallisuuksia, kuten esimerkiksi siirrettävyyttä. Sovelluksilla voikin olla useita esikäännöskonfiguraatiosta riippuvia toimintoja, josta esimerkkinä on mm. sovelluksen kääntäminen useille eri laitealustoille hyödyntäen ehdollista esikäännöstä laitekohtaisten lähdekoodilohkojen osalta. Käännöskonfiguraatiot määritellään esikäännösmääreillä, joita voi olla useita. Esikääntämisellä on myös haittapuolia kuten siitä mahdollisesti aiheutuvat vaikeasti paikannettavat virhetoiminnallisuudet, mutta sitä käytetään varsin laajalti sen lukuisten hyötyjen vuoksi. [40]

4.1 Esikääntäjien arkkitehtuuri ja toiminta

Tämä luku perustuu lähteeseen [11].

Esikääntäjän tehtävänä on suorittaa sarja tekstimuotoisia muunnoksia saamalleen syötemerkkijonolle ennen varsinaisen kääntäjän työvaiheita. Periaatteessa kaikki esikäännöksen vaiheet tapahtuvat määrätyssä järjestyksessä ja annetun syötteen perusteella saatu tulos on seuraavan vaiheen syöte. Käytännössä vaiheet kuitenkin suoritetaan yhtäaikaisesti esikäännösprosessin tehostamiseksi. Esikääntäjiä on toteutettu useille eri ohjelmointikielille sopiviksi ja vaikka niiden käyttämä syntaksi on erilainen, on niiden sisäinen rakenne varsin samanlainen. Kuva 4.1 esittää yleisesti käännösprosessin ja esikäännösvaiheen sijoittumisen tässä prosessissa.

Seuraavaksi esittelen C-standardin mukaisen esikääntäjän toimintaa vaihe vaiheelta. Ensimmäiset kaksi vaihetta ovat tekstimuotoista muunnosta. Ensimmäisessä vaiheessa, mikäli kolmimerkit (trigraphs) ovat käytössä, muunnetaan ne vastaaviksi käytetyn merkistön merkeiksi. Toisessa vaiheessa esikääntäjä lukee lähdekooditiedoston muistiin pilkkoen tekstin riveiksi ja käsitellen erilaiset rivinvaihtomerkkaukset. Lisäksi kaikki lähdekoosin kommentit korvataan välilyöntimerkeillä.



Kuva 4.1: Käännösprosessi ja esikäännösvaihe

Tekstimuotoisten muunnosten jälkeen esikäännöksen kolmannessa vaiheessa syötetiedosto symboloidaan, eli muunnetaan sarjaksi tarkoin määriteltyjä esikäännössymboleita, joita kutsutaan myös tokeneiksi. Nämä symbolit luokitellaan viiteen luokkaan: tunnisteet (identifiers), esikäännösnumerot (preprocessing numbers), merkkijonot (string literals), välimerkit (punctuators) sekä muut (other). Usein nämä symbolit ovat yhtenäisiä varsinaisen kääntäjän symboleille, mutta niissä voi olla poikkeuksia. C-kielen standardin mukaan esikääntäjän symbolit kuitenkin ovat yhtäläisiä varsinaisen kääntäjän symbolien kanssa eikä kääntäjä erikseen muunna esikääntäjän tuotoksia omiksi käännössymboleiksi vaan hyödyntää ne sellaisenaan.

Esikäännöksen tunniste on yleensä samanlainen kuin itse ohjelmointikielen tunniste. C-kielessä tunniste on mikä tahansa merkeistä, numeroista tai alleviivausmerkeistä koostuvat merkkijono, joka alkaa merkillä tai alleviivausmerkillä. Ohjelmointikielen

avainsanat (keywords) eivät ole esikääntäjälle merkityksellisiä vaan nekin käsitellään kuten tunnisteet. Esikääntäjällä voi tosin olla omia avainsanoja, joiden perusteella se käsittelee lähdekoodia sille määrättyjen ohjeiden mukaisesti.

Esikäännösnumeroita ovat kaikki tunnetut kokonais- ja liukulukuvakiot, jotka on määritelty myös kyseisessä ohjelmointikielessä. Merkkijonoihin kuuluvat merkkijonosekä merkkivakiot ja otsikkotiedostojen nimet. Välimerkkejä ovat kaikki ohjelmointikieleen kuuluvat pienet osaset kuten vertailu- ja sijoitusmerkit. Kaikki muut yhden merkin osaset syötteessä käsitellään "muut" luokkaan kuuluviksi ja ne välitetään kääntäjälle muuttumattomina.

Symboloinnin jälkeen symbolit välitetään kääntäjän jäsentäjälle. Mikäli symbolit sisältävät esikäännöslausekkeita tai -määreitä, tulee ne muuntaa esikääntäjän avulla ohjelmakoodilohkoiksi ja symboloida ennen jatkokäsittelyä. Tätä kutsutaan esikäännöksen neljänneksi vaiheeksi, ja useimmiten ihmiset ajattelevat tätä vaihetta varsinaiseksi esikääntäjän tehtäväksi. Esikäännöskieli sisältää suoritettavia lausekkeita, määreitä ja makroja, jotka tulee laajentaa. Nämä ovat C-kielen standardin mukaisesti:

- otsikkotiedostojen sisällyttäminen
- makrojen laajentaminen
- ehdollinen kääntäminen
- rivien hallinta (Esikäännöksen jälkeisen väliaikaisen tiedoston riveihin voidaan lisätä viitetieto alkuperäisen tiedoston riveihin. Näin voidaan virheenetsintätoimintoja monipuolistaa.)
- diagnostiikka, jota hyödyntäen voidaan havaita käännösaikaiset ongelmat ja näyttää virheet sekä varoitukset.

4.2 Esikäännöksen hyödyt ja haitat

Esikääntäjät tukevat yleensä ehdollista kääntämistä ehto- ja määrityslauseiden avulla (ks. liite 1). Niitä käytetäänkin varsin usein esimerkiksi siirrettävyyden parantamiseen, ohjelmakoodin optimoimiseen sekä selkeyttämiseen. Lisäksi esikääntäjällä voidaan tarvittaessa laajentaa puutteellisen ohjelmointikielen ominaisuuksia. Esimerkki tällaisesta ohjelmointikielen täydennyksestä esikääntäjän ominaisuuksia hyväksikäyttäen ovat C-kielen vakiot, jotka määritellään `#define`-esikäännösmääreellä. Hyviä esimerkkejä paljon esikäännöstä hyödyntävistä sovelluksista ovat Mozilla ja Linux-käyttöjärjestelmän

kernel, joissa kummassakin noin 40% lähdekoodiriveistä on esikäsiteltyä. Sekä Mozillassa sekä Linuxissa ehdollista esikääntämistä on hyödynnetty siirrettävyyden parantamiseksi, ja se onkin onnistunut varsin hyvin. [2, 9]

Vaikka ehdollisen esikääntämisen hyödyt ovatkin kiistattomat, voi siitä aiheutua myös paljon harmia. On esimerkiksi erittäin vaikeaa tarkastaa jokaisen mahdollisen esikäännöskonfiguraation oikea toiminnallisuus ja virheettömyys, varsikin kun erilaisia konfiguraatioita voi olla jopa useita satoja. Lisäksi jotkut esikäännösehdot saattavat aiheuttaa sen, ettei tiettyjä ohjelmakoodilohkoja suoriteta koskaan. [2]

Yksi esikäännöksen suurimmista ongelmista on sen usein aiheuttama lähdekoodin ymmärrettävyyden heikentyminen. Ongelma ilmenee siten, että varsinaisen kääntäjän syötteenä saama lähdekoodi on esikäännöksen vuoksi erilainen kuin sovelluskehittäjän tuottama ja sisäistämä lähdekoodi. Ja vaikka esikääntäjien toiminta onkin tarkoin määriteltyä, voi niiden tuottamien muutosten seuraaminen olla todella vaikeaa, mikä hankaloittaa lähdekoodin ymmärtämistä ja sitä myöten myös sovelluksen ylläpitoa. Lisäksi esikäännösmääreet aiheuttavat usein ongelmia erilaisille sovelluskehittäjän apuvälineille, joiden tehtävä on selkeyttää lähdekoodia esimerkiksi värjäämällä tai piilottamalla epärelevantteja lähdekoodilohkoja. [40]

Esikäännösvaiheessa ongelmia saattavat aiheuttaa myös erilaiset esiprosessointitietojen tyyppimääritykset. Usein esikääntäjät eivät yleensä tarkasta muuttujien tyyppimäärityksiä korvatessaan annetun symbolin toisella ja huolettomasti käytettynä esikäännös saattaakin tuottaa sovellukseen todella vaikeasti jäljitettäviä virheitä, koska tietyissä tapauksissa virheelliset tyyppimääreet eivät estä varsinaista käännösprosessia. Toinen varsin yksinkertainen, mutta helposti paljon päänvaivaa aiheuttava ongelma ovat esikäännöksen myötä muunnetun ohjelmakoodin muuttujien näkyvyydet. Esimerkiksi lähdekoodiin lisätään esikäännösvaiheessa uusia lähdekoodin osasia, joka muuttavat sovelluskehittäjän määrittelemien muuttujien arvoja kehittäjän siitä tietämättä. [2]

Nykyisin esikääntäjän tarjoamista ominaisuuksista käytetään yleisesti ehdollista esikääntämistä räätälöityjen käännösversioiden toteuttamiseksi haluttuihin kohdeympäristöihin. Tässä toimintatavassa sovelluskehittäjä joutuu kuitenkin ottamaan selvää tai muistamaan jokaisen käytetyn kohdeympäristön erityispiirteet kyseisen ohjelmakoodin aihealueella. Tämä aiheuttaa ylimääräistä muistirasitetta ohjelmoijalle sekä usein epäselkeitä pitkien ehtolausejonojen koristamaa lähdekoodia sekä hidastaa sovelluksen kehitystyötä. [2]

Jotta esikääntämisestä olisi varsinaista hyötyä mobiililaitteiden fragmentaatiosta aiheutuvien ongelmien ehkäisemiseksi, tulee sovellusta kehittävällä ryhmällä olla tietämyskanta, josta löytyvät tiedot eri laitteiden ominaisuuksista ja virheistä. Ilman näitä

tietoja on laitekohtaisen lähdekoodin tuottaminen mahdotonta. Tässä tutkielmassa en kuitenkaan ota kantaa tietämyskannan sisältöön vaan työkaluihin, joiden avulla näitä tietoja voidaan mahdollisimman helposti ja tehokkaasti hyödyntää.

Seuraavaksi esittelen jo olemassa olevia esikäännökseen pohjautuvia fragmentaatiosta johtuvien ongelmien ratkaisuun kehitettyjä työkaluja ja analysoin niiden hyviä ja huonoja puolia. Lisäksi tutkin erilaisia mahdollisuuksia esikäännöksen haittapuolien ehkäisemiseksi ja uusia näkökulmia esikäännöksen hyödyntämisessä.

5 Toteutettuja järjestelmiä fragmentaatio-ongelmien ratkaisuun

Tässä luvussa esittelen yleisimmin Java ME -ympäristöön toteutettuja fragmentaatiosta aiheutuvia ongelmia ratkaisevia työkaluja sekä niiden toteutus- ja käyttötapoja. Esitellyt työkalut etsin internetistä käyttäen apuna hakurobotteja sekä mobiilisovelluskehitykseen suunnattujen keskustelufoorumeiden viestejä. Esitellyt työkalut valittiin sillä perusteella, että niillä pystyy ainakin jossain määrin ratkaisemaan fragmentaatiosta aiheutuvia ongelmia. Esittelyissä otan esiin työkalujen hyvät ja huonot puolet, joiden perusteella toteutin Preppi-liitännäisen Eclipse Platformille.

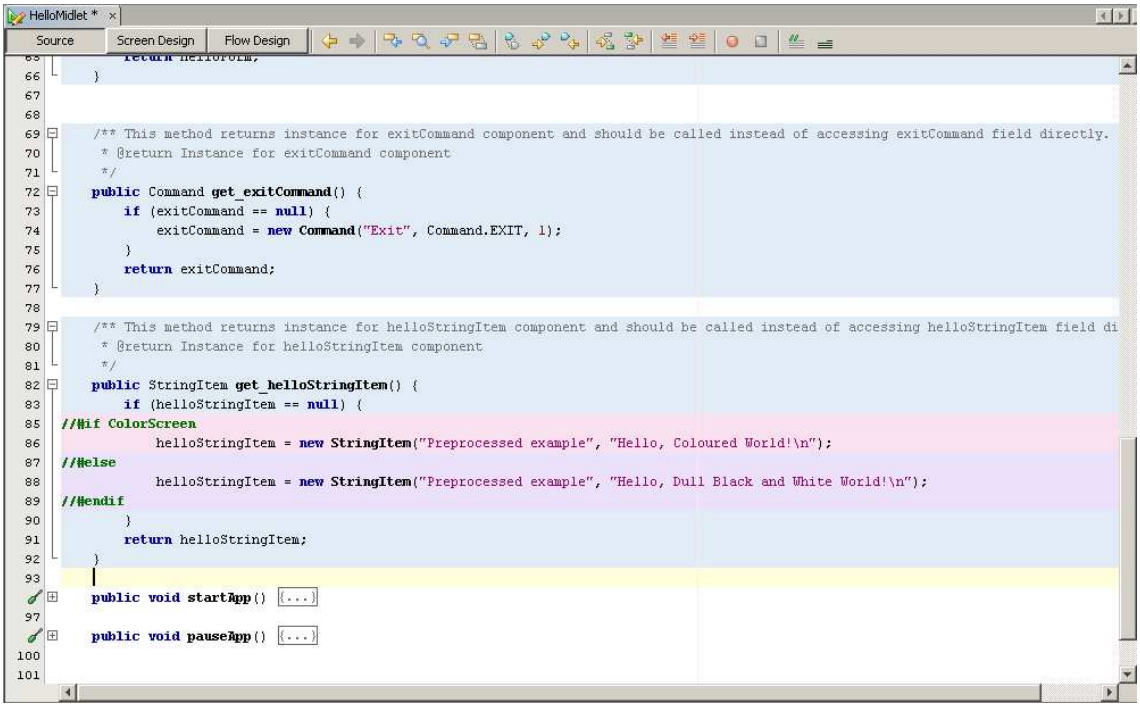
5.1 NetBeans ja NetBeans Mobility Pack

NetBeans on avoimeen lähdekoodiin perustuva sovelluskehitysympäristö, johon kuuluvat NetBeans IDE ja NetBeans Platform. NetBeans muuttui avoimen lähdekoodin järjestelmäksi, kun Sun Microsystems, tällä hetkellä projektin suurin sponsori, julkisti heinäkuussa 2000 NetBeansin lähdekoodit. NetBeans sellaisenaan on Java SE -kehitysympäristö, mutta sen voi laajentaa myös mobiililaiteiden Java-sovelluskehitystyökaluksi asentamalla NetBeans Mobility Packin, joka sisältää kaiken tarvittavan mobiilisovellusten toteuttamiseen, kuten mobiililaiteprofiilit ja emulaattorin. [17]

NetBeans Mobility Pack -laajennos sisältää sovelluskehitystä helpottavia ja toimintaa selventäviä työkaluja kuten esimerkiksi graafisen sovelluksen etenemistä kuvaavan *Flow Designerin* [17]. Nämä työkalut eivät kuitenkaan ota kantaa fragmentaation aiheuttamiin ongelmiin. NetBeans Mobility Packissa fragmentaatio-ongelmat ratkaistaan C-kielen esikääntäjälle tyypillisellä ehdollisella esikäännöksellä, jossa hyödynnetään erilaisia ehto- ja määrittelylauseita (ks. liite 2).

Tässä lähestymistavassa ratkaista fragmentaation aiheuttamia ongelmatilanteita joudutaan halutulle kohdelaitteelle kirjoittamaan laitekohtaista ohjelmakoodia omiin ehdollisiin esikäännöslohkoihin muun lähdekoodin joukkoon. Ongelmaksi tässä tavassa muodostuu lähdekoodin epäselkeys ja jossain määrin mahdollisesti myös toistuvuus esikäännöslohkojen välillä, mikäli fragmentaation aiheuttamat lähdekoodin erot ovat pieniä. Lisäksi, mikäli haluttuja kohdelaitteita on paljon, tulee ehdollisesti esikäännettäviä lähdekoodilohkoja useita peräkkäin, mikä tekee lähdekoodista epäselkeää ja

vaikeammin ylläpidettävää. Kuva 5.1 esittää NetBeans-työpöytänäkymää, jossa on yksinkertainen mallilähdekoodi usealle kohdelaitteelle suunnatusta esikäännettävästä lähdekoodista.



Kuva 5.1: NetBeans -näkymä ja esikäännösmääreet

Käännös usealle kohdelaitteelle NetBeansissa onnistuu kätevästi valitsemalla valikosta käännöksen kaikille määritellyille projektikonfiguraatioille. Tietylle kohdelaitteelle käännös onnistuu samaan tapaan valikosta valitsemalla vain käännös halutulle konfiguraatiolle. Tällainen käytäntö onkin hyvä, sillä usein kehitettävästä sovelluksesta halutaan helposti käännös usealle eri laitteelle. Hyvänä asiana pidän myös NetBeans Mobility Packin tarjoamia värikoodeja lähdekoodin taustalla, jotka selventävät minkä tyyppisestä koodilohkosta on kyse, joita ovat mm. puhdas Java-koodi, esikäännösmääreet ja esikäännöslohkot.

5.2 J2ME Polish

J2ME Polish on saksalaisen Enough Softwaren toteuttama kokoelma erilaisia työkaluja ja valmiita komponentteja, jotka mahdollistavat Java ME -sovellusten kehittämisen mobiililaitteiden fragmentaation huomioiden. J2ME Polish tulee liittää haluttuun sovelluskehitysympäristöön, kuten Eclipse Platformiin tai Netbeans IDE:en, jotta sen antamia mahdollisuuksia voidaan hyödyntää. [5]

J2ME Polishin toiminta perustuu esikäännökseen, määrittelytiedostoihin ja muutamaan valmiina tarjottuun komponenttiin. Komponentit, kuten pelimoottorikomponentti ja käyttöliittymäkomponentti, ovat lisäpalveluita, eikä niiden käyttö ole pakollista. Nämä komponentit on kuitenkin toteutettu fragmentaatio-ongelmat huomioiden ja voivat nopeuttaa mobiilisovelluskehitystä huomattavasti tarjoamalla tarvittaessa valmista toiminnallisuutta toteutettavaan sovellukseen. Ulkoasukomponenttia voidaan käyttää mobiilisovelluksen ulkoasun yleisen tyylin määrittämiseen käyttäen CSS-tyylitiedostoa. J2ME Polish ratkaiseekin fragmentaatiosta aiheutuvia ongelmia ehdollisella esikääntämisellä (ks. lähdekoodi 6) sekä haluttaessa yllämainitulla CSS-määrittelyjä hyödyntävällä käyttöliittymäkomponentilla. Esikääntämisen aikana J2ME Polish käyttää sen rinnalle kehitettyä tietämyskantaa, johon on kerätty mobiililaitteiden fragmentaatioinformaatiota mahdollisimman kattavasti. Tietämyskanta sijaitsee Enough Softwaren J2ME Polish -projektin kotisivuilla ja se päivittyy yksityisihmisten tekemien mobiililaitteiden virhetoimintohavaintojen perusteella. [5]

J2ME Polishin käyttö ei kasvata toteutetun sovelluksen kokoa mikäli hyödynnetään vain esikäännöstoiminnallisuutta. Käyttöliittymäkomponentin tai pelimoottorin ominaisuuksia käytettäessä kasvaa sovelluksen koko noin 5-45 kilotavua, mikä voi joissain tapauksissa olla liikaa - varsinkin vanhemmilla ja heikkoresurssisimmilla mobiililaitteilla. [5]

J2ME Polishin käännöskonfiguraatiot määritellään *build.xml*-tiedostossa, jossa kerrotaan myös mille laitteille kyseinen Java-projekti käännetään, joten käännös joko vain yhdelle tai useammalle kohdelaitteelle onnistuu kerralla. J2ME Polishissa sovelluskehittäjä voi myös tarpeen mukaan määritellä omiin tarpeisiinsa sopivia esikäännöskomentoja ja vakioita, mikä on mielestäni erittäin hyvä lisä työkalun tarjoamien ominaisuuksien lisäksi. Ongelmana näen kuitenkin samanlaisen ehdollisen esikäännöksen vaatimien if-else-rakenteiden tarpeellisuuden kuin NetBeans Mobility Packissa. [5]

Lähdekoodi 6 Malli Java-lähdekoodista, jossa on hyödynnetty J2ME Polish:in ehdollisia esikäännöslausekkeita laiteen ominaisuuksien tutkimiseen ja erilaisille laitteille räätälöidyn lähdekoodin tuottamiseen.

```
//#if polish.api.mmapi || polish.midp2
   ... lähdekoodi, jossa hyödynnetään Mobile Media API:a, mikäli
       käännösvaiheessa valittu kohdelaite tukee sitä. ...
//#else
   ... lähdekoodi, joka otetaan käyttöön, mikäli käännösvaiheessa
       valittu kohdelaite ei tue MMAPI:a. ...
//#endif
```

5.3 Antenna

Antenna on avoimeen lähdekoodiin perustuva työkalu, joka tarjoaa Apache Ant -build-työkalun toimintoja, jotka on laajennettu kattamaan Java ME -ympäristössä toimivien mobiilisovellusten vaatimukset. Ant tarjoaa tuen laitealustasta riippumattomiin konfiguraatioihin, joissa määritellään halutut toimintosarjat määritellyille komennoille. Käyttäjä voi tarpeen vaatiessa muokata määriteltyjä komentoja eri laitealustoilla toimiviksi kokoelmiksi. [36, 14]

Antenna-työkalua käyttäen sovelluskehittäjä voi kääntää (compile), esivarmentaa (preverify), pakata (package), monimutkaistaa (obfuscate), suorittaa (run) MIDP-sovelluksia eli MIDlet:tejä, muokata JAD-tiedostoja sekä muuntaa JAR-tiedostoja PalmOS-järjestelmissä toimivaan PRC-formaattiin. Antenna tarjoaa myös ehdollisen esikäännöksen toimintoja, joilla voidaan tehdä mukautettuja käännöksiä sovelluksesta käyttäen esikäännösmääreitä sisältävää lähdekoodia. Antennan tarjoamat esikäännöskomennot ovat yleisesti tarjottuja määrittely- sekä if-else-rakenteita (ks. liite 3), joiden käytön mahdollistavat mm. NetBeans Mobility Pack sekä J2ME Polish. [14]

Vaikka yleensä Java-ohjelmointityökalut mahdollistavat erilaisten käännöskonfiguraatioiden hallinnan ja hyödyntämisen fragmentaatio-ongelman torjumiseksi, ovat Ant-skriptit riippumattomia käytetystä työympäristöstä. Näin sovelluskehittäjät voivat tarpeen vaatiessa helpohkosti vaihtaa käytettyjä työkaluja ja edelleen käyttää Antennaa sekä jo luotuja Ant-skriptejä Java-projekteissaan. [14]

Antenna ei ole itsenäisesti toimiva kokonaisuus, vaan se on toteutettu pääasiassa Java Wireless Toolkit:in ja Ant:in tarjoaman toiminnallisuuden varaan. Mikäli monimutkaistus (obfucation) halutaan ottaa käyttöön tarvitaan myös muita työkaluja, kuten esimerkiksi RetroGuard tai ProGuard, jotka toteuttavat tämän työvaiheen. [14]

5.4 JaTS - Java Transformation System

Java Transformation System toteutettiin tarkoituksena määritellä ja luoda tehokas tapa tuottaa apuväline, jolla voidaan nopeasti toteuttaa ja testata monimutkaisia Java-kielellä toteutettuja sovelluksia ja täten tehostaa sovelluskehittäjän tuottavuutta. Toteutettu JaTS toimii mallinsovitusta (pattern matching) hyödyntäen ja muokkaa kokonaisia Java-kielen lähdekooditiedostoja annettujen ohjeiden mukaisesti. Nämä ohjeet esitellään kahdessa eri tiedostossa: mallisovitustiedostossa ja kohdetiedostossa. Mallisovitus- ja kohdetiedostot sisältävät Java-lähdekoodia, joissa on JaTS-määreitä, jotka alkavat yleensä #-merkillä. [39]

JaTS-muunnos on kaksivaiheinen. Ensimmäisessä vaiheessa etsitään sopiva mallisovitustiedosto annetulle alkuperäiselle lähdekooditiedostolle. Tällöin alkuperäisen lähdekooditiedoston ja mallisovitustiedoston perusrakenne täytyy olla samanlainen, jotta mallisovitustiedoston määreet voidaan sovittaa alkuperäiseen lähdekooditiedostoon. Toisessa vaiheessa, kun sovitus on tehty, kohdetiedosto tuotetaan määrittelyiden mukaisesti. Alla olevat lähdekoodiesimerkit 7-10 selvittävät JaTS:in toimintaa. [39]

Lähdekoodi 7 Alkuperäinen Java-kielinen lähdekoodi

```
public class TestJaTS extends Test {
    TestJaTS() {}
    public void singleMethod() throws WeakException {}
}
```

Lähdekoodi 8 Alkuperäiseen lähdekoodiin sopiva JaTS-mallisovitustiedosto

```
public class #a extends #b {
    #a() {}
    public void #c() throws #d {}
}
```

Lähdekoodi 9 JaTS-kohdelähdekoodi

```
public class #a extends #b {
    #a(int numberOfTests) {}
    #a() {}
    public synchronized void #c() throws #d,
        SomethingGoneTerriblyWrongException {}
}
```

Lähdekoodi 10 JaTS-muunnosten jälkeen syntynyt lähdekoodi

```
public class TestMatch extends Test {
    TesteMatch(int numberOfTests) {}
    TestMatch() {}
    public synchronized void singleMethod() throws
        WeakException, SomethingGoneTerriblyWrongException {}
}
```

Tutkittuani Java Transformation System -työkalun toimintaa tulin siihen tulokseen, ettei JaTS ole sopiva työkalu mobiililaitteiden fragmentaatiosta aiheutuvien ongelmien ratkaisemiseksi. JaTS on suunniteltu varsin suurten ja monimutkaisten järjestelmien lähdekoodin tehostamiseksi ja selkeyttämiseksi eikä mahdollista lähdekoodin sisällön muokkaamista tavalla, joka olisi käytännöllistä mobiilisovellusten siirrettävyyden parantamiseksi. JaTS tarjoaa olio-ohjelmointipohjaisia muunnostyökaluja luokkien rakenteen muokkaamiseksi, mutta sillä ei voida muokata esimerkiksi vain tietyn metodin osan lähdekoodia.

5.5 Jappo

Jappo on avoimeen lähdekoodiin perustuva Java-esikääntäjä, joka tukee tekstipohjaisia sovelluskehittäjän toteuttamia makroja. Nämä makrot tukevat parametrisointia (ks. lähdekoodi 11). Lisäksi Jappo mahdollistaa muutamien valmiiksi toteutettujen makrojen, kuten päivämäärä- tai rivinumerointimakrojen, ja ehdollisen esikääntämisen käytön. Jappoa voidaan käyttää Ant-työkalun avulla tai komentokehotteesta. [21]

Fragmentaatio-ongelmien ratkaisuun voidaan käyttää Jappo-esikääntäjän tarjoamaa ehdollista esikäännöstoimintoa. Samaa toiminnallisuutta tarjoavat kuitenkin monet muutkin työkalut, kuten esimerkiksi Antenna, NetBeans Mobility Pack sekä J2ME

Polish. Erinomaisena ja muista poikkeavana ajatuksena pidän kuitenkin mahdollisuutta kehittää halutunkaltaisia makroja, joita voidaan tarvittaessa kutsua esikäännösmääreen kautta (ks. lähdekoodit 11 ja 12). Sellaisenaan Jappo-esikääntäjän toiminnallisuutta ei voida kuitenkaan käyttää fragmentaatiosta aiheutuvien ongelmien ratkaisemiseksi, sillä nämä määritellyt makrot ovat aina vakioita riippumatta mille laitteelle sovellus haluttaisiin kääntää. Jappoa käyttääkseen joutuisi sovelluskehittäjä vaihtamaan Jappo-esikääntäjän makrot määrittelevät tiedostot aina kun halutaan käännös erilaiselle kohdelaitteelle. Mikäli tämä työläs välivaihe poistettaisiin, olisi Jappo varsin hyvä fragmentaatiosta aiheutuvien ongelmien ratkasuvaihtoehto. [21]

Lähdekoodi 11 Jappo-makron `!out` määrittely

```
<macro name="!out" class="org.osfry.jappo.macros.TemplateMacro">
  <template>
    <![CDATA[System.out.println(DATA)]]>
  </template>
  <param name="DATA">
</macro>
```

Lähdekoodi 12 Jappo-makron `!out` käyttö ehdollisessa esikäännöksessä

```
#define(_JAVA)

public final class Example {

    public static void main (String args[]) {
        #ifdef _JAVA
            !out("Java is defined");
        #else
            !out("Java is not defined");
        #endif
    }
}
```

5.6 JET

JET (Java Emitter Templates) on Eclipse Modeling Framwork -kokonaisuuteen sisältyviä tehokas lähdekoodin generointityökalu. JET-työkalua käytettäessä voidaan hyödyntää mukautettavia malleja (template), joilla määritellään generoitava lähdekoodi. Eclipse sisältää työkalut mallien luomiseksi, muokkaamiseksi ja poistamiseksi. [25]

Toisin kuin useat lähdekoodin generointiin perustuvat työkalut, JET:in toiminta perustuu esikäännökseen. JET:in käyttämät mallit määrittelevät haluttuja luokkia, joita voidaan käyttää haluttujen toimintojen toteuttamiseksi, joten ne antavat sovelluskehittäjälle varsin suuren vapauden toteuttaa haluamiaan toiminnallisuuksia. Mallit tukevat myös parametrisointia. Työkalua käytettäessä voidaan ratkaista myös fragmentaatioongelmia tekemällä parametrisoituja malleja, jotka sisältävät tietyn toiminnallisuuden toteutuksen useille eri mobiililaitteille. Haluttu toiminnallisuus määrätään mallin esittelemän luokan toiminnallisuuden tarjoavaa metodia kutsuen määritellen kutsuparametrillä minkä laitteen toiminnallisuus otetaan ko. kerralla käyttöön. Esimerkki tällaisesta mallista on lähdekoodiesimerkissä 13. Valitettavasti tällainen mallien parametrisointi JET:iä käytettäessä ajautuu samaan ongelmaan kuin wrapper-luokat ja middleware-komponentit: niiden koko kasvaa käytettyjen laitteiden määrän mukaan, ja pidemmällä aikavälillä niiden käyttö voi osoittautua mahdottomaksi pieniresurssisilla mobiililaitteilla. [25]

Lähdekoodi 13 JET-mallin esittelytiedosto, jossa on räätälöity toiminnallisuus Nokia 7650 -puhelimen drawString-toiminnolle.

```
<%@ jet package="testJET" imports="javax.microedition.lcdui.*"
class="DrawString" %>

<% DrawStringParams params = (DrawStringParams)argument; %>
<% if (params.device.equals("Nokia/7650/rev1")) { %>
chrs[] = params.stringToWrite.toCharArray();
params.graphics.drawChars(chrs,0,chrs.length,x,y,
    Graphics.LEFT | Graphics.TOP);
<% } else { %>
params.graphics.drawString(str,x,y, Graphics.LEFT | Graphics.TOP);
<% } %>
```

JET ei kuitenkaan ole suunniteltu EclipseME-liitännäistä ajatellen, eikä se tue liitännäisen tarjoamia palveluita. Mobiilisovelluskehityksessä JETin käyttö vaatiikin yli-

määräistä työtä, koska esikäännöksen tuottamat mallit käyttävät hyväksi Java SE:n kirjastoja, joita ei Java ME:ssä voida hyödyntää.

Vaikka JET ei tarjoakaan tämän tutkimuksen kannalta täysin toimivaa ratkaisua fragmentaatio-ongelmiin, on siinä myös hyviä ajatuksia, joita voidaan hyödyntää myös tämän aihepiirin ongelmissa. JET-työkalu takaa mm. lähdekoodin parametrien tarkastuksen, joten virheelliset parametrit eivät pääse heikentämään esikäännöksen tuottaman koodin laatua. Lisäksi malleissa voidaan `imports` -määreellä määritellä tarpeen vaatiessa lähdekoodin tarvitsemat kirjastopaketit, joita puhtaasti esikäännökseen perustuvat työkalut yleensä, kuten esimerkiksi J2ME Polish, eivät ota huomioon.

5.7 Yhteenveto

Taulukossa 5.1 esittelen tiivistettynä yhteenvetona käsiteltyjen työkalujen ominaisuuksia, joiden perusteella tulen toteuttamaan mahdollisimman mukautettavan, siirrettävän ja helppokäyttöiden Preppi-liitännäisen. Liitännäisen tarkoituksena on esiteltyjä työkaluja kattavammin ja helpommin mahdollistaa fragmentaatiosta aiheutuvien ongelmien ratkaiseminen.

Taulukko 5.1: Yhteenveto esiteltyjen fragmentaatio-ongelmia ratkovien työkalujen ominaisuuksista

Työkalu	**A**	**EE**	**O**	**MS**	**PK**	**MK**	**S**	**UL**	**VL**	**ML**
Netbeans Mobility Pack	X	X						X	X	X
J2ME Polish	X	X						X		X
Antenna	X	X					X	X		X
JaTS				X						
Jappo	X	X			X	X	X			
JET		X	X		X	X				

Lyhenteiden merkitykset ovat:

- A — perustuu avoimeen lähdekoodiin
- EE — hyödyntää ehdollista esikäännöstä
- O — hyödyntää olio-ohjelmointiin perustuvia tekniikoita
- MS — hyödyntää mallinsovitusta esikäännöksessä
- PK — mahdollistaa parametrisoidut esikäännöskomennot
- MK — mahdollistaa esikäännöskomentojen muokkauksen
- S — esikäännösmääreitä sisältävä lähdekoodi on siirrettävissä muihin sovelluskehitysympäristöihin.
- UL — mahdollisuus suorittaa käännös usealle kohdelaitteelle vähin vaivoin
- VL — lähdekoodin värjäystoiminto eri koodilohkoille
- ML — mahdollistaa käännöstoiminnot esikäännöksineen lopputuotteeksi asti

6 Toteutettu Preppi-liitännäinen

Tässä luvussa esittelen tutkielmani käytännönläheisemmän osan, jossa toteutin Eclipse Platformin versiolle 3.1 (tai uudempi) Preppi-nimisen liitännäisen, jonka toiminnallisuus perustuu pääosin lähdekoodin esikääntämiseen. Aluksi esittelen toteuttamani fragmentaatio-ongelmiin ratkaisuja tuovan liitännäisen taustat ja tavoitteet. Tämän jälkeen esittelen lyhyesti liitännäisen käyttämän alustan, Eclipse Platformin, jonka jälkeen kerron toteuttamani liitännäisen ominaisuuksista ja uusista fragmentaatio-ongelmien ratkaisutapojen näkökulmista.

6.1 Taustat ja tavoitteet

Päätökseni uuden työkalun toteuttamiseksi fragmentaatio-ongelmien ratkaisemiseksi syntyi tutkittuani ongelmiin jo toteutettuja työkaluja sekä pohdittuani niissä hyväksi ilmenneitä toimivia ratkaisuja, puutteita ja muita huonoja puolia. Hyvät puolet yhdistämällä ja niiden toiminnallisuutta laajentamalla ajatusteni ja kokemusteni pohjalta sekä huonoja puolia karsimalla saatiin aikaiseksi tehokkaampi, mukautettavampi sekä sovelluskehittäjän kannalta helppokäyttöisempi työkalu fragmentaation aiheuttamien ongelmien ratkaisemiseksi.

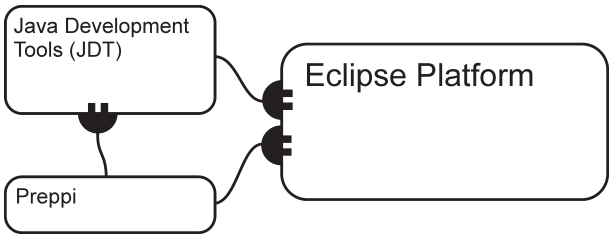
Preppi-liitännäisen alustaksi valitsin Eclipse Platformin sen suuren käyttäjäryhmän, avoimen lähdekoodin sekä helpon laajennettavuuden perusteella. Lisäksi Eclipse tarjoaa erinomaiset työkalut uusien liitännäisten toteuttamiseksi [37].

Liitännäisen käyttämä ratkaisutapa fragmentaation aiheuttamiin ongelmiin on esikäännös, jolla saadaan varsin kattavasti useat mobiililaitteiden erilaisuudesta johtuvat ongelmat niiden sovelluksissa ratkaistua. Esikäännöksen valitsin sen käytännöllisyyden vuoksi. Nykyisin olio-ohjelmointipohjaiset ratkaisut, kuten wrapper-luokat ja middleware-komponentit ovat vielä tänä päivänä toistaiseksi liian raskaita rakenteita räätälöidyn lähdekoodin suorittamiseksi erilaisilla ominaisuuksilla varustetuilla mobiililaitteilla. Korkean abstraktiotason kielillä toteutettavat sovellukset, jotka käännettynä tuottaisivat alemman abstraktiotason lähdekoodia, voisivat ratkaista fragmentaatiosta aiheutuvia ongelmia, mutta tämä vaatisi sovelluskehittäjältä uuden korkean abstraktiotason ohjelmointikielen sekä siihen liittyvien työkalujen opettelun. Lisäksi korkean abstraktiotason kielen käännöksenä saadun lähdekoodin tehokkuuteen ei sovelluskehittäjä voi paljoakaan vaikuttaa.

Suunnitellessani ja toteuttaessani pyrin alkuperäisten tavoitteiden mukaisesti valitsemaan sellaiset ratkaisutavat, jotta liitännäisen käyttö on sovelluskehittäjän näkökulmasta helppoa ja tehokasta sekä tarjoaa riittävän vapaat kädet sovelluksen kehittämiseen. Yhtenä suurimmista esikäännöksen ongelmista ovat olleet pitkät ja vaikeaselkoiset ehdollisen esikäännöksen määrittely- sekä if-else-rakenteet. Suuri esikäännöksen ongelmatekijä on myös esikäännöksen jälkeisen lähdekoodin näkymättömyys sovelluskehittäjälle. Muun muassa nämä kaksi seikkaa aiheuttavat usein vaikeasti paikannettavia virheitä perinteisellä tyylillä käytettyä ehdollista esikäännöstä hyödynnettäessä. Preppi-liitännäisen tulisikin mahdollistaa erilaisille mobiililaitteille suunnattujen räätälöityjen lähdekoodilohkojen käyttö ilman if-else-rakenteita. Liitännäisen tarjoaman esikäännöksen tuotteena syntynyt lähdekoodi tulisi olla myös aina sovelluskehittäjän nähtävissä, jotta esikäännöksessä syntyneiltä, joskus vaikeastikin paikannettavilta, sovelluksen virheiltä vältyttäisiin.

Nykyisin jotkin Java-työkalujen esikääntäjät tarjoavat mukautettavia, sovelluskehittäjän luomia, esikäännösmakroja, jotka voidaan myös parametrisoida. Toteutettavassa Preppi-liitännäisessä tulisi myös olla mahdollisuus luoda ja käyttää mukautettavia ja parametrisoituja esikäännöskomentoja, joiden käytön esimerkiksi Jappo-esikääntäjä mahdollistaa. Makrojen käyttöä tulee kuitenkin laajentaa Jappo-esikääntäjän tarjoamasta toiminnallisuudesta siten, että se mahdollistaa helposti useiden eri mobiililaitteiden räätälöityjen lähdekoodilohkojen sisällyttämisen sovelluksen lähdekoodiin ilman suurta työpanosta. Lisäksi Preppi-liitännäisen tulisi tarkistaa lähdekoodimakrojen parametrien tyyppitarkistusta, jotta erilaisten tyyppivirheiden mahdollisuus minimoitaisiin.

Pyrin toteuttamaan Preppi-liitännäisen itsenäiseksi Eclipsen työkaluksi, joka ei vaadi alustan tarjoamien ominaisuuksien lisäksi muita aputyökaluja, kuten esimerkiksi Ant-työkalua toimiakseen. Tämä yksinkertaistaa liitännäisen käyttöönottoa eikä liitännäisen toiminta tällöin ole riippuvainen tietyistä apusovelluksista ja niiden mahdollisesti tarkoin rajatuista versioista, mikäli niiden toiminnallisuus muuttuu ajan saatossa. Preppi-liitännäisen liittyminen Eclipse Platformiin on esitetty kuvassa 6.1.



Kuva 6.1: Preppi-liitännäisen liittyminen Eclipse Platformiin

6.1.1 Vaatimukset

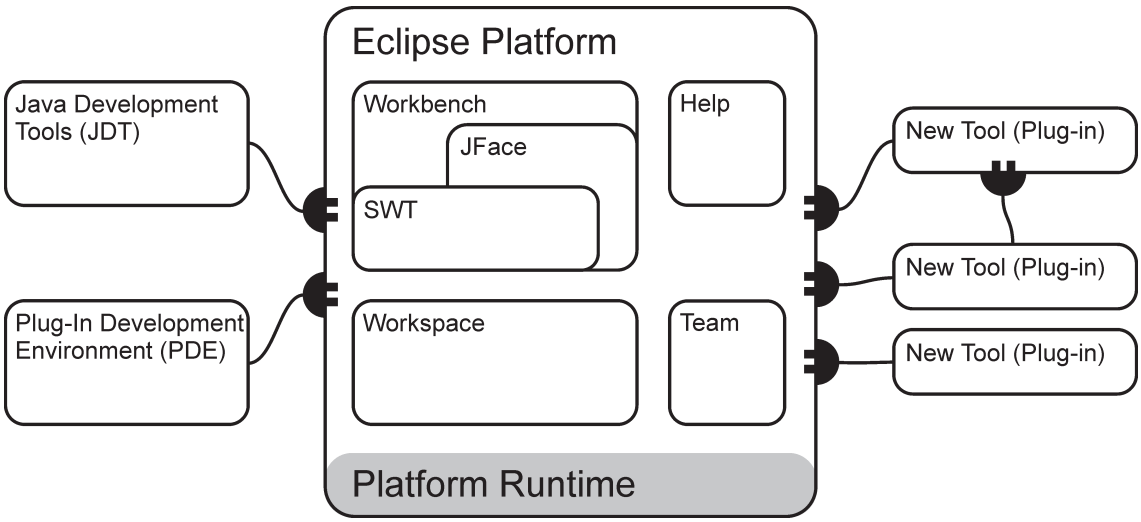
Taulukossa 6.1 esittelen Preppi-liitännäisen yleiset ominaisuudet, jotka tulee toteuttaa, jotta liitännäinen todella helpottaisi sovelluskehittäjän työtä fragmentaatiosta ja esikäännöksistä aiheutuvien ongelmien alueella ja olisi helposti käytettävä.

Taulukko 6.1: Preppi-liitännäisen vaaditut ominaisuudet

#	**Vaatimus**
1	Fragmentaatiosta aiheutuneiden ongelmien ratkaisutapana käytetään esikäännökseen perustuvaa tekniikkaa.
2	Lähdekoodista tulee olla aina mahdollista nähdä sekä esikääntämätön että esikäännetty versio.
3	Haluttu kohdelaite, jolle määriteltyä toiminnallisuutta esikäännöskomennon kohdalla käytetään, tulee olla mahdollista valita.
4	Esikäännöskomentoja tulee olla mahdollista luoda, muokata ja poistaa.
5	Esikäännöskomennot tulee olla mahdollista parametrisoida.
6	Liitännäinen mahdollistaa käännöstoiminnot esikäännöksineen lopputuotteeksi asti.
7	Määritellyille esikäännöskomennoille tulee olla mahdollista määritellä versionhallinta, jota hyödyntäen voidaan tarvittaessa käyttää aiemmin määriteltyjä esikäännöskomentoja.
8	Esikäännöskomentojen toiminnallisuus tulee voida sisällyttää komennon paikalle tai metodikutsun muodossa. Metodikutsun tapauksessa metodin määreet, kuten `public`, `private` ja `static`, tulee voida määritellä.
9	Esikäännösvaiheessa tulee esikäännöskomentojen parametrien ja paluuarvojen tietotyyppien oikeellisuus tarkastaa.

6.2 Eclipse Platform

Eclipse Platform (ks kuva. 6.2) on geneerinen sovelluskehitysalusta, joka on suunniteltu integroitujen ohjelmointiympäristöjen, IDE:jen, toteuttamiseen. Eclipsessä on Java-ohjelmointikielen käännöstoiminnot mukana Java Developement Tools (JDT) -liitännäisenä, mutta mikään ei estä muiden ohjelmointikielten käyttöä. Eclipse tarjoaa sovelluskehittäjälle mahdollisuuden muokata, kääntää ja suorittaa kehitettäviä sovelluksia ja Eclipse Platformin toimintaa laajentavia liitännäisiä. Lisäksi Eclipse mahdollistaa mm. ajonaikaisen virheenjäljityksen kehitettävästä sovelluksesta. [37].



Kuva 6.2: Eclipse Platformin rakenne

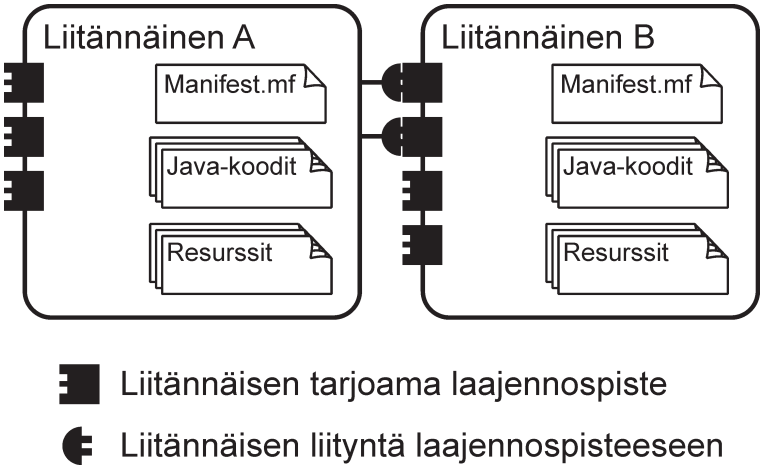
Eclipse Platform on avoin alusta, jota on helppo laajentaa sovelluskehittäjien tarpeisiin sopivaksi. Teknisemmältä kannalta Eclipse on vain sovelluskehys (framework), josta saadaan erittäin käytännöllinen työkalu siihen liitettyjen toimintoja tarjoavien liitännäisten avulla. [37].

IBM:n tytäryhtiö Object Technology International (OTI) sai ajatuksen Eclipse-projektista. Tällä projektilla IBM pyrki vähentämään tarjoamiensa, toisiinsa sopimattomien, kehitysympäristöjen määrää. Eclipse ei syntynyt silmänräpäyksessä, vaan sitä edelsivät useat eri sovelluskehitysympäristöt, kuten VisualAge for Smalltalk^{TM} ja IBM VisualAge for Java^{TM}. Aiemmat järjestelmät olivat toimivia, mutta niiden laajennettavuus oli kehnonlainen, koska sitä ei ollut huomioitu suunnittelussa riittävän kattavasti. [37].

Pieni asiantuntijaryhmä analysoi ihmisten kokemukset aiemmista sovelluskehitysjärjestelmistä ja toteutti Eclipse Platformin, jossa laajennettavuus oli liitännäisten muodossa otettu paremmin huomioon. Myöhemmin vuonna 2001 ryhmä yrityksiä muodosti Eclipse Consortiumin ja Eclipse siirtyi avoimen lähdekoodin ohjelmistoksi, jonka käyttäjäkunta alkoi kasvaa erittäin nopeasti. Eclipse-projektin tuotteita ovat Eclipse Platform, Java Development Tools (JDT) ja Plug-In Development Environment (PDE). Nämä kolme tuotetta muodostavat täydellisen Java-sovelluskehitysympäristön, joka mahdollistaa myös uusien liitännäisten toteuttamisen Eclipse Platformiin. [37].

Liitännäinen, Plug-In, on Eclipse Platformin pienin toiminnallisuutta tarjoava kokonaisuus. Yleensä yksinkertaiset työkalut toteutetaan yhden erillisen liitännäisen muodossa, mutta monimutkaisemmissa tapauksissa voidaan toiminnallisuus toteuttaa useammalla keskenään kommunikoivalla liitännäisellä (ks. kuva 6.3). Lukuun ottamatta

Platform Runtimeä, Eclipse Platformin ydintä (kernel), kaikki Eclipsen toiminnallisuus sijaitsee Java-kielellä toteutetuissa liitännäisissä. Tyypillinen liitännäinen sisältää Java-koodia sekä resurssitiedostoja, kuten esimerkiksi kuvia ja natiivikielisiä ohjelmakirjastoja. On olemassa myös liitännäisiä, jotka eivät sisällä yhtään lähdekoodia. Tällaisia ovat esimerkiksi ohjetiedostoihin laajennoksia tarjoavat liitännäiset. Jokaiselle liitännäiselle on määritelty XML-pohjaisessa `manifest.mf`-tiedostossa sen riippuvuussuhteet (depencies) ja laajennospisteet (extension points) muihin liitännäisiin. Näitä laajennospisteitä käyttäen liitännäiset voivat kommunikoida keskenään ja Eclipse Platformin kanssa ja näin laajentaa alustansa tarjoamaa toiminnallisuutta. [12]



Kuva 6.3: Eclipse-liitännäisten rakenne

Käynnistyessään Platform Runtime löytää asennetut liitännäiset, lukee niiden manifest-tiedostot ja rakentaa näiden tietojen pohjalta rekisterin käytettävissä olevista liitännäisistä. Liitännäisten ohjelmakoodia ei ladata tai suoriteta ennen kuin käyttäjä aktivoi niiden toiminnallisuuden. [12]

6.3 Esikäännös

Preppi-liitännäisen tarjoama esikäännöstoiminto perustuu täysin esikäännöskomentoihin ja lähdekoodimalleihin, jotka esittelen tarkemmin seuraavissa luvuissa. Esikääntäjä etsii annetusta lähdekooditiedostosta esikäännöskomennot ja niihin mahdollisesti liittyvät versio- tai sisällytysmääreet. Esikäännöskomennon löytäessään esikääntäjä hakee tätä komentoja ja toivottua ensisijaista kohdelaitetta vastaavan lähdekoodimallin, jonka se sisällyttää esikäännettyyn lähdekoodiin joko metodina tai esikäännösmääreen paikalle sijoitettavalla lähdekoodimallin kohdelaitteelle räätälöidyllä koodilla. Esikäännöskomentoa etsitään lähdekoodimallin nimen, parametrien sekä käytettävän

kohdelaitteen perusteella. Haku tapahtuu nimen ja kohdelaitteen perusteella kirjaimellisesti. Parametrien perusteella haku tapahtuu kirjaimellisesti perustietotyyppien ja -rakenteiden, kuten esimerkiksi `int`, `double` ja `String`, perusteella, mutta liitännäiselle tuntemattomien tyyppien kohdalle Preppi hyväksyy minkä tahansa muuttujan. Mikäli esikäännöskomentoa vastaavaa lähdekoodimallia ei ensisijaiselle kohdelaitteelle löydy, yrittää Preppi löytää lähdekoodimallin halutuille prioriteettijärjestyksen omaaville toissijaisille kohdelaitteille. Mikäli tällöinkään ei esikäännöskomentoa vastaavaa lähdekoodimallia pystytty löytämään, käyttää liitännäinen oletuslaitteelle luotua lähdekoodimallia. Oletuslaite on nimeltään `Default Device / Default Model /`. Mikäli lähdekoodimallia ei löydy oletuslaitteellekaan, lisää esikääntäjä esikäännettyyn lähdekoodiin virheilmoituksen, joka ilmaisee sovelluskehittäjälle, ettei kyseessä olevalle esikäännöskomennolle kyetty löytämään vastaavaa lähdekoodimallia.

Esikäännöskomennon paikalle lisää Preppi-liitännäinen joko metodikutsun tai sisällytetyn lähdekoodimallin määrittelemän lähdekoodin. Mikäli käytetään metodityyppisiä rakenteita, lisää liitännäinen esikäännetyn lähdekoodin loppuosaan luotua metodikutsua vastaavan metodin toteutuksen, joka sisältää lähdekoodimallin tarjoaman toiminnallisuuden lähdekoodin. Metodin nimi muodostetaan lähdekoodimallin nimen, version sekä sen parametrien tyyppinimiä käyttäen, jotta esikäännöskomentojen ylikuormitus on mahdollista, eikä useita eri toimintoja toteuttavat metodit olisi saman nimisiä. Lisäksi Preppi sisällyttää lähdekoodiin kommenttiriveinä tiedon, mikä esikäännöskomento on korvattu metodikutsulla tai lähdekoodin sisällytyksellä sekä kertoo perustiedot esikäännöksestä, kuten esimerkiksi mikä oli käytetty ensisijainen kohdelaite ja minkä laitteen ja version lähdekoodimallia esikäännöksessä käytettiin.

Esikäännä ja suorita ja *esikäännä projektiin* -toiminnot, jotka esittelen tarkemmin käyttöliittymää esitellessäni, suorittavat jokaiselle valitussa projektissa olevalle Java-lähdekooditiedostolle esikäännöksen, jonka jälkeen Preppi hyödyntää Eclipsen tarjoamaa automaattista käännöstoiminnallisuutta. Tässä vaiheessa saa sovelluskehittäjä mahdollisista virheistä standardit Eclipsen tarjoamat virheilmoitukset, joiden perusteella sovelluskehittäjän on helppo muokata alkuperäistä esikääntämätöntä lähdekoodia. Mikäli virheitä ei esikäännöksen jälkeen ole ollut ja kehittäjä suoritti *esikäännä ja suorita* -toiminnon käynnistää, Eclipse aktiivisena olevan valitun projektin.

6.4 Lähdekoodimallit

Paljon käytetyn, mutta usein myös paljon ongelmia aiheuttavan, ehdollisen esikäännöksen sijaan käytin lähdekoodimalleja (template), joita voidaan verrata Jappo-esikääntäjän lähdekoodimakroihin. Lähdekoodimalleja käyttäen ratkaisin fragmentaation aiheut-

taman tarpeen kohdelaitteelle räätälöidyn lähdekoodin käyttöön ilman monimutkaisia ja pitkiä esikäännöksen if-else-rakenteita. Tutkittuani erilaisia esikäännöstyylejä tulin siihen tulokseen, ettei if-else-rakenteita lähdekoodimalleja hyödynnettäessä tarvita, eikä Preppi-liitännäinen myöskään tue niitä. Nämä rakenteet piilotetaan sovelluskehittäjältä tarjoamalla käyttöön korkeamman abstraktiotason tarjoamia esikäännöskomentoja. Esikäännöskomennot muutetaan esikääntäjän toimesta halutulle kohdelaitteelle räätälöidyksi halutun toiminnallisuuden toteuttavaksi lähdekoodiksi lähdekoodimallien mukaisesti. Saman toiminnallisuuden toteuttavia, mutta mahdollisesti laitteiden erilaisuudesta aiheutuvia ongelmia ratkaisevia, räätälöityä lähdekoodia sisältäviä malleja voidaan toteuttaa useille eri laitteille. Kun lähdekoodimalleista on olemassa versiot useille eri laitteille, voidaan esikääntäjän määrityksistä valita käytettävä kohdelaite, jolle toteutettuja lähdekoodimalleja esikäännöksessä käytetään.

Lähdekoodimalli toteuttaa yhden räätälöidyn toiminnallisuuden yhdelle määritellylle mobiililaitteelle. Laitteet yksilöidäkseni valitsin käytettäviksi tiedoiksi laitteen valmistajan nimen, laitteen mallin sekä version, jolla tarkoitetaan laitteen ohjelmiston versiota. Nämä tiedot riittävät nykyisin hyvin kohdelaitteiden yksilöimiseksi. Vain yksi lähdekoodimalli ei anna sovelluskehittäjälle lisäarvoa fragmentaatiosta aiheutuvien ongelmien ratkaisemiseen, mutta useista malleista koostuva lähdekoodimallikirjasto tuottaakin jo varsin hyvät mahdollisuudet fragmentaatio-ongelmien kiertämiseksi. Lähdekoodimalleja käyttämällä voi sovelluskehittäjä varsin vapaasti kohottaa käyttämäänsä lähdekoodin abstraktiotasoa. Preppi-liitännäinen tarjoaa sovelluskehittäjälle työkalut lähdekoodimallien luomiseksi, muokkaamiseksi sekä poistamiseksi. Näin voidaan aina luoda tarpeen vaatiessa markkinoille tuleville uusille laitteille lähdekoodimalleja, joilla voidaan määritellä laitteen ominaisuuksia tai kiertää laitteessa esiintyviä virheellisiä toiminnallisuuksia. Preppi-liitännäisen tapauksessa tietämyskanta erilaisten laitteiden ominaisuuksista sisällytetään lähdekoodimallikirjastoon lähdekoodimallien muodossa. Liitännäinen mahdollistaa lähdekoodimallin tarjoaman esikäännöskomennon käytön heti kun malli on luotu, eikä sovelluskehitysympäristössä tarvitse erikseen päivittää tietoja tai käynnistää sitä uudestaan.

Lähdekoodimalli sisältää kaikki esikäännöskomennon käyttämiseksi tarvittavat tiedot. Nämä tiedot vaaditaan, jotta Preppi-liitännäinen osaa eritellä eri laitteille toteutetut toiminnallisuudet kohdelaitekohtaisesti ja pystyy esikäännöksen aikana muokkaamaan lähdekoodia siten, että lopputulos on oikeellista Java-koodia. Lisäksi käytettävyyttä ja lähdekoodimallin ylläpidettävyyttä silmällä pitäen lähdekoodimalli sisältää kommentti- ja leijuntaohjetiedot. Lähdekoodimallin sisältämät tiedot ovat:

- esikäännöskomennon prototyyppi
- käytettävä kohdelaite
- kommentit
- leijuntaohje, joka esitetään sovelluskehittäjälle lähdekoodieditorissa hänen siirrettyään hiiren osoitin kyseessä olevan esikäännöskomennon päälle (hover help).
- lähdekoodin vaatimat kirjastot
- tieto onko lähdekoodi esikäännöskomennon paikalle sisällytettävää vai suoritetaanko se metodikutsulla
- kohdelaitteelle räätälöity lähdekoodi
- tieto onko lähdekoodi aina inline-muotoista vai voidaanko mallin tarjoama toiminnallisuus toteuttaa myös metodikutsuna.
- metatiedot, kuten lähdekoodimallin luoja sekä luontiaika.

Lähdekoodimallin tiedot tallennetaan erillisiin XML-formaatissa oleviin tekstitiedostoihin, jotka sijaitsevat liitännäisen asetuksissa määrätyssä hakemistossa. Nämä tiedostot ovat helposti muokattavissa myös tavallisella tekstieditorilla. Alla on esimerkki `graphics.drawString`-esikäännöskomennon tarjoamasta lähdekoodimallista, joka tarjoaa korjauksen Nokia 7650 -puhelimessa esiintyvään Java-virtuaalikoneen merkkijonon kirjoitus -virheeseen.

```
<?xml version="1.0"?>
  <preppitemplate prototype="void graphics.drawString(
      Graphics g, String str, int x, int y)" version="1">
  <device brand="Nokia" model="7650" version="ver1" />
  <creation time="212443" date="20060721" author="nules">
    <comments>
        Workaround for Nokia 7650 mobile phone. If str is over 200
        characters long, the usual drawString method crashes.
        drawChars is used instead.
    </comments>
  </creation>
```

```
  <hoverhelp>
      Draws a string to a given graphics. With Nokia 7650
      Graphics.drawChars method is used to draw to avoid method crash.
  </hoverhelp>
  <importlibraries>javax.microedition.lcdui.Graphics</importlibraries>
  <inlineonly>false</inlineonly>
  <sourcecode>
      char chrs[] = str.toCharArray();
      g.drawChars(chrs,0,chrs.length,x,y, Graphics.LEFT | Graphics.TOP);
  </sourcecode>
</preppitemplate>
```

6.5 Esikäännöskomennot

Preppi-liitännäisen esikäännös tapahtuu esikäännöskomentojen perusteella. Esikääntäjälle merkkinä esikäännöstä vaativasta rivistä on rivin alussa oleva `//#` merkki, jota käytetään yleisesti myös muiden esikääntäjien, kuten J2ME Polishin, komennoissa. Esikäännöskomentoja voidaan kirjoittaa tavallisen lähdekoodin joukkoon haluttuihin kohtiin, kuten lähdekoodista 14 voimme huomata.

Lähdekoodi 14 Java-lähdekoodia, jossa on Preppi-esikäännöskomentoja

```
/* Clears the mobile device's screen with light gray and
   draws Hello World! to the top left corner of the screen
   with black colour. */

 protected void paint(Graphics g) {
    /* Clear the screen */
    g.setColor(200,200,200);
    //# int width = devicedb.constants.screenWidth();
    //# int height = devicedb.constants.screenHeight();
    g.fillRect(0,0,width,height);
    /* Draw the string */
    g.setColor(0,0,0);
    //# graphics.drawString(g,"Hello World!",0,0);
}
```

Esikäännettävä rivi voi sisältää tavanomaista Java-lähdekoodia ennen esikäännöskomentoa. Tästä esimerkkinä ovat lähdekoodiesimerkin 14 esikäännöskomennot `//# int width = devicedb.screenWidth();`. Komennossa esitellään lähdekoodi `int width =` ennen esikäännöskomentoa. Lähdekoodia voidaan myös halutessa kirjoittaa esikäännöskomennon jälkeen, vaikka tällaista ei esimerkkilähdekoodissa näykään.

Suuren ongelman esikäännöksen käytössä nykyisin aiheuttavat erilaiset tyyppivirheet, joita ei tarkasteta. Varsinkin heikosti tyypitetyissä kielissä, kuten C ja C++, voidaan esikäännöksellä luoda erittäin vaikeasti jäljitettäviä virheitä. Esikäännöksessä lähdekoodiin syntynyt virhe ei välttämättä estä käännösprosessin etenemistä, ja tietyissä tapauksissa, kuten virheellisessä osoitintyyppisien muuttujien käsittelyssä, voi kääntäjä suoriutua tehtävästään ilman virheitä, vaikka tuotettu sovellus voi sisältää erittäin vakavia virheitä. Vahvasti määritellyissä kielissä, kuten Object Pascalissa tai Javassa, ei esikääntäjien tyyppitarkastuksen puute aiheuta tällaisia ongelmia, sillä kääntäjä katkaisee käännösprosessin mikäli käytetty muuttujan tai olion tyyppi on väärä.

Preppi-liitännäistä suunnitellessani ja toteuttaessani pyrin toteuttamaan esikääntäjään tyyppitarkastustoimintoja, jotta mahdolliset virhetilanteet voidaan havaita aiemmin. Preppi-liitännäisen esikääntäjä tarkistaakin esikäännöskomentojen perustietotyyppien oikeellisuuden ja ilmoittaa virhetilanteista esikäännetyssä lähdekoodissa. Tyyppitarkastus mahdollistaa myös esikäännöskomentojen ylikuormittamisen, jota Preppi-liitännäistä käytettäessä voidaan hyödyntää. Aiemmin esitellyissä työkaluissa ylikuormituksta ei voitu hyödyntää, koska ne perustivat toimintansa pääasiassa ehdolliseen esikääntämiseen.

Halutessaan sovelluskehittäjä voi määritellä esikäännöskomentoriville mitä versiota lähdekoodimallista käytetään joko esittelemällä halutun mallin versionumeron tai määrittelemällä halutun version esittelemällä halutun päivämäärän, josta tuorein valittuun päivämäärään mennessä luotu malli otetaan käyttöön. Haluttu versionumero esitellään `$[versionumero]` tai `%[päivämäärä]` esikäännösmääreillä, joista löytyy esimerkit lähdekoodissa 15. Mikäli haluttua mallia ei löydy käytetään ensisijaisesti aiempaa versiota valitun laitteen lähdekoodimallista ja seuraavaksi esikääntäjälle määriteltyjen toissijaisten laitteiden malleista. Mikäli mallia ei edelleenkään ole löydetty, käytetään oletuslähdekoodimallia.

Lähdekoodi 15 Preppi-esikäännöskomentojen manuaalinen versiohaku

```
//# graphics.drawString(g,"Hello World! (version)",0,0); $[ver2]
//# graphics.drawString(g,"Hello World! (date)",0,0); %[22.7.2006]
```

Tavallisesti Preppi esikääntäjä tuottaa esikäännöskomentojen perusteella käsiteltävään luokkaan metodeja ja metodikutsuja, mutta halutessaan esikäännöskomennon määreillä voidaan ohjeistaa esikääntäjää sisällyttämään lähdekoodimallin koodisisältö esikäännöskomennon paikalle (inline inclusion). Tämä tapahtuu lisäämällä `source:` tai `src:` määre esikäännöskomennon alkuun, kuten lähdekoodiesimerkissä 16 on tehty. Lähdekoodin sisällyttäminen voi selventää esikäännettyä lähdekoodia, mutta voi myös aiheuttaa ongelmatilanteita. Useimmiten ongelmia aiheuttavat muuttujien nimet, joita on hyödynnetty sekä alkuperäisessä lähdekoodissa että sisällytettävässä lähdekoodissa. Näissä tapauksissa voi sisällytetty lähdekoodi muuttaa joidenkin muuttujien arvoja sovelluskehittäjän siitä tietämättä. Näitä ongelmia ehkäistääkseen Preppi-liitännäinen tarjoaa näkymän, josta sovelluskehittäjä voi tarkastella esikäännettyä lähdekoodia.

Lähdekoodi 16 Preppi-esikäännöskomentoja inline-määreellä

```
//# src:graphics.drawString(g,"Hello World!",0,0);
//# source:graphics.drawString(g,"Hello World! (date)",0,0);
```

Ilman erillisiä määreitä esikäännöskomennot käsitellään metodityyppisinä rakenteina, eli metodikutsuina sekä metodin toteutuksina. Metodin julkisuusaste ja tyyppi, jotka voidaan asettaa esimerkiksi `public` ja `static` -määreillä, voidaan esikäännöskomennossa määritellä samaan tapaan kuin lähdekoodin sisällytys. Mikäli julkisuusastetta tai tyyppiä ei ole määritelty, käsitellään metodia `private` julkisuusasteella ilman tyyppimäärityksiä. Preppi-esikäännöskomennossa halutut julkisuus- ja tyyppimääreet kirjoitetaan yhteen yhdistäen ne yhdysviivalla. Mikäli määreitä käytetään, tulee julkisuusaste esitellä aina. Esimerkkejä erilaisista julkisuus- ja tyyppimääreellisistä metodityyppisistä esikäännöskomennoista on lähdekoodiesimerkissä 17.

Lähdekoodi 17 Preppi-esikäännöskomentoja julkisuus- ja tyyppimääreillä

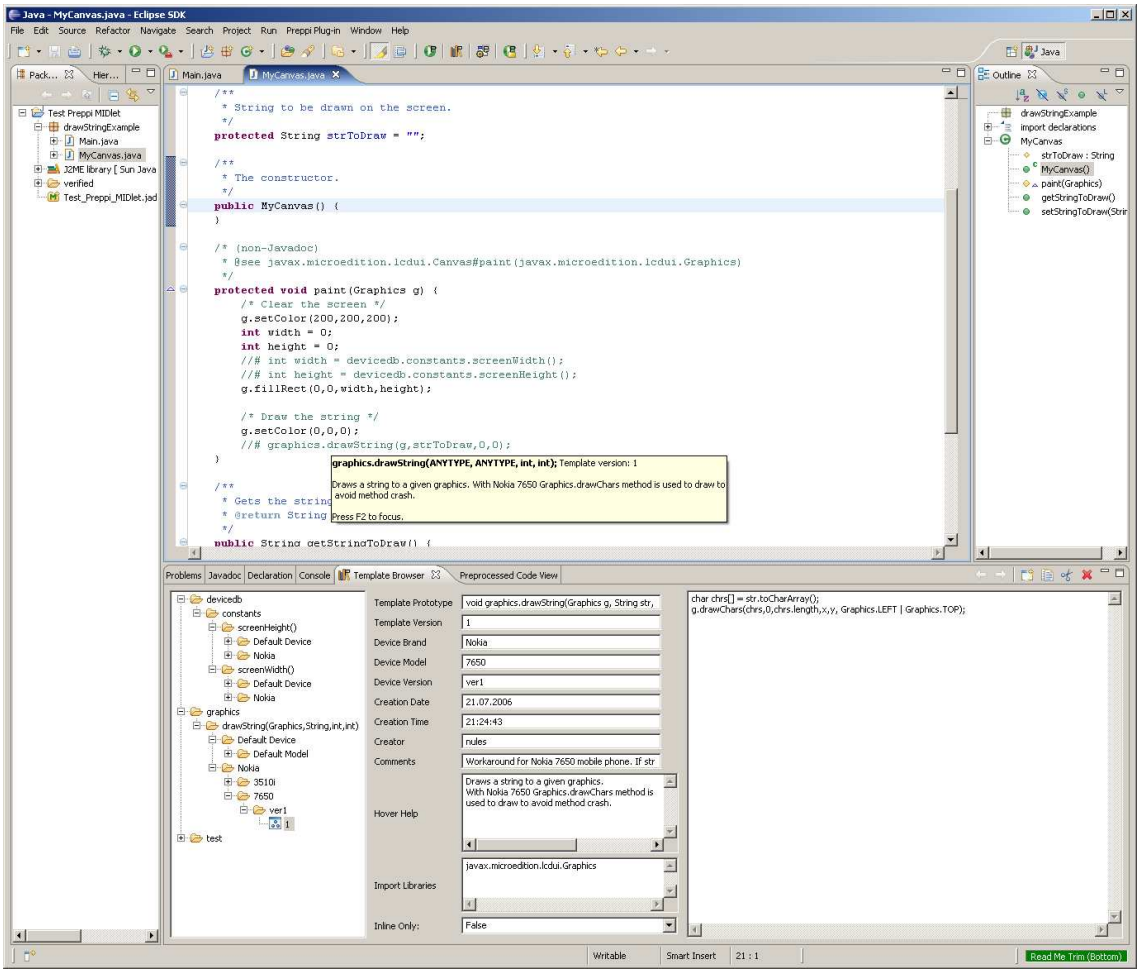
```
//# public:graphics.drawString(g,"Hello World!",0,0);
//# public-static:graphics.drawString(g,"Hello World! (date)",0,0);
//# protected:graphics.drawString(g,"Hello World! (date)",0,0);
```

6.6 Käyttöliittymä

Toteuttamani Preppi-liitännäisen käyttöliittymän pyrin tekemään riittävän monipuoliseksi, mutta samalla myös yksinkertaiseksi. Käyttöliittymä ei kuitenkaan ollut tämän tutkimuksen pääaihe, joten en työssä keskittynyt sen suunnitteluun ja toteutukseen.

Preppi-liitännäisen käyttöliittymän kaksi ydinnäkymää ovat *Template Browser* -näkymä jossa käyttäjä voi selata, luoda, muokata ja poistaa lähdekoodimalleja, sekä *Preprocessed Code View* -näkymä, joka nimensä mukaisesti näyttää sovelluskehittäjälle sillä hetkellä muokatun lähdekoodin esikäännöksen jälkeen.

Kuvassa 6.4 Eclipse-sovelluskehitysympäristössä Template Browser -näkymä on sijoitettu kuvan alalaitaan muiden näkymien joukkoon. Näkymän paikkaa voidaan muuttaa halutessa Eclipse Platformin tarjoamin keinoin, kuten esimerkiksi näkymän vetäminen hiirtä käyttäen eri kohtiin sovelluskehitysympäristöä.



Kuva 6.4: Template Browser -näkymä Eclipsessä

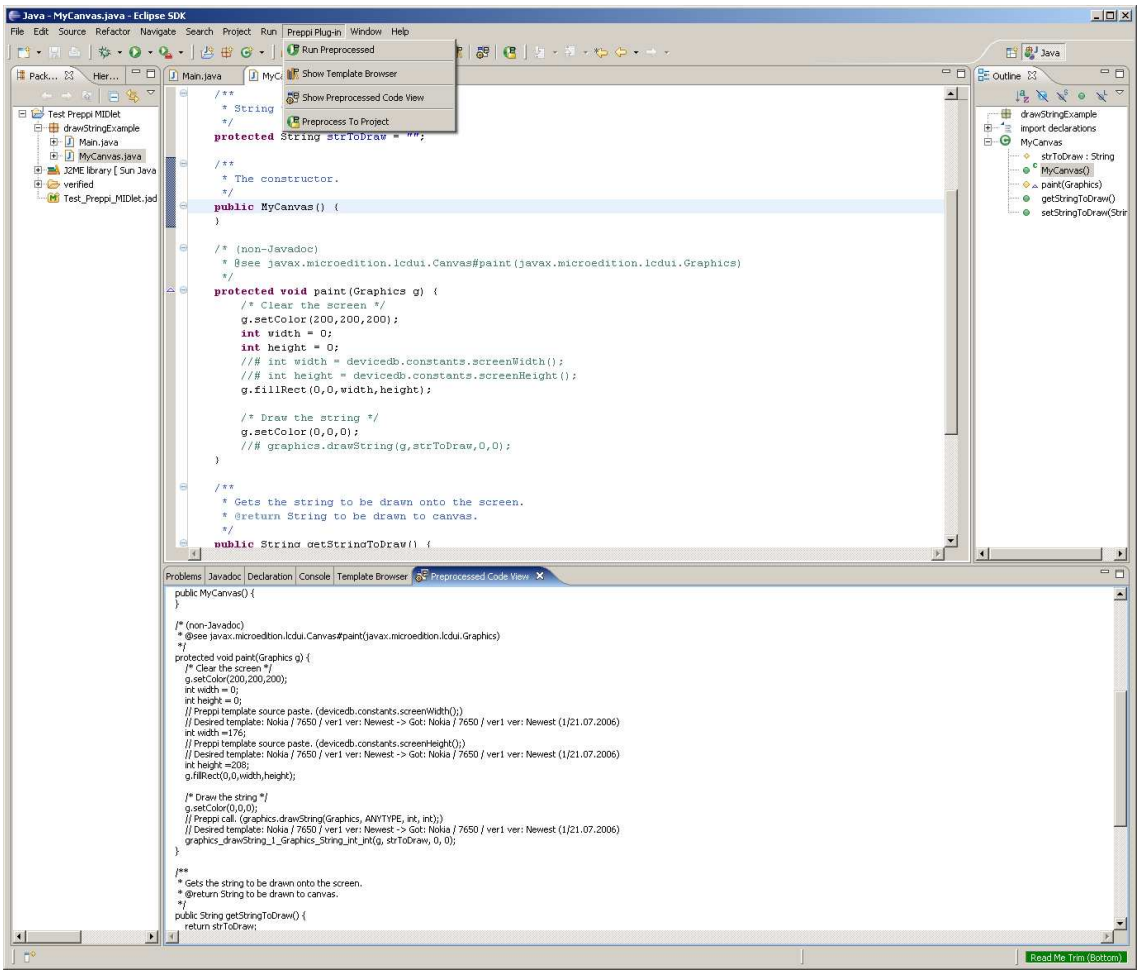
Template Browser tarjoaa lähdekoodimallien selailuun puurakenteena esitetyn lähdekoodimallilistauksen. Lähdekoodimallit voidaan nimetä halutulla tavalla. Mallit voidaan jaotella erilaisiin kokonaisuuksiin. Kokonaisuuksiin jako tapahtuu mallien nimien perusteella. Nimessä piste-merkki toimii kokoelmaerottimena. Kuvassa 6.4 nähdään, että Template Browser -näkymässä valittu esikäännöskomento `drawString` on sijoittunut `graphics` -kokoelmaan nimensä "`graphics.drawString`" mukaisesti.

Template Browser -näkymän puurakenteisen lähdekoodimallihakemiston oikealla puolella ovat kaikki lähdekoodimallin sisältämät tiedot, jotka esittelen tarkemmin luvussa 6.4.

Preppi-liitännäinen tarjoaa sovelluskehittäjälle myös mahdollisuudet tarkastella esikäännöskomennon toiminnallisuutta sekä siihen mahdollisesti liittyviä huomautuksia siirtämällä hiirenosoittimen haluamansa esikäännöskomennon päälle hetkeksi. Liitännäinen on laajentanut Eclipsen `JavaUI`-liitännäisen toimintaa siten, että se näyttää ns. leijuntaohjeikkunan (hover help) perinteisen Java-lähdekoodin lisäksi Preppi-esikäännöskomentojen kohdalla, kuten kuvasta 6.4 nähdään.

Kuvassa 6.5 esitellään *Preprocessed Code View* -näkymä, jonka tehtävä on näyttää sovelluskehittäjälle, miltä hänen lähdekoodinsa näyttää esikäännöksen jälkeen. Tämä ehkäisee yhden esikäännöksen suurimmista ongelmienaiheuttajista, joka on esikäännetyn lähdekoodin näkymättömyys, sovelluskehittäjälle.

Preppi-liitännäinen laajentaa myös Eclipsen työkalurivin sekä valikon toiminnallisuutta. Liitännäinen lisää näihin toimintovalinnat ja -painikkeet, joilla yllä esitetyt näkymät saadaan käyttöön ja joilla voidaan lisäksi suorittaa *Esikäännä ja suorita* (Run Preprocessed) sekä *Esikäännä projektiin* (Preprocess To Project) -toiminnot. Lisäksi Preppi-liitännäinen lisää asetussivun Eclipsen asetustenhallintaan, jossa voidaan määritellä mm. halutut kohdelaitteet, lähdekoodimallien sijainti tiedostojärjestelmässä, halutut lähdekoodimallien versiot sekä projektin nimi mikäli Esikäännä projektiin -toimintoa käytetään. Asetuksista kerron lisää seuraavassa luvussa.



Kuva 6.5: Preprocessed Code -näkymä Eclipsessä

6.7 Asetukset

Toteutettu liitännäinen hyödyntää toiminnassaan sille määriteltyjä asetuksia, joita ovat:

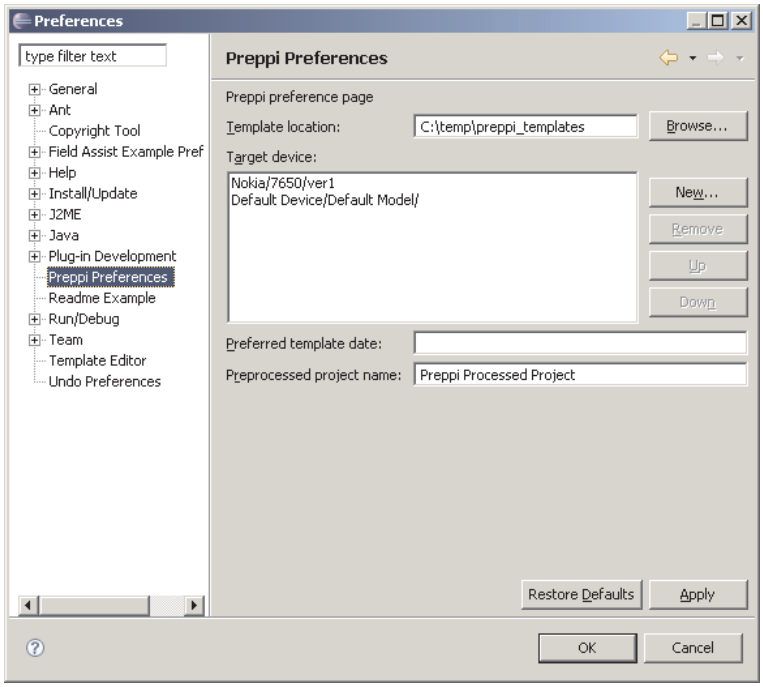
- Lähdekoodimallien sijaintipaikka
- Ensisijaisesti käytettävä ja toissijaisesti käytettävät kohdelaitteet
- Toivottu lähdekoodimallien päivämäärä
- Esikäännetyn projektin nimi

Jotta Preppi-liitännäisen esikäännöstoiminnot voisivat toimia, täytyy sen asetuksiin määritellä lähdekoodimallien sijaintipaikka. Lisäksi on määriteltävä ensisijaisesti

käytettävä kohdelaite. Sovelluskehittäjä voi halutessaan valita myös toissijaisesti käytettävät kohdelaitteet, joiden lähdekoodimalleja käytetään, mikäli ensisijaiselle kohdelaitteelle tiettyä lähdekoodimallia ei ole olemassa. Toissijaisilla laitteilla on prioriteettijärjestys, jota käyttäen etsitään ensimmäinen haluttua esikäännöskomentoa vastaava lähdekoodimalli. Kohdelaitemäärityksillä voidaan tarvittaessa myös luoda erilaisia tuoteperhemalleja, sillä mikään ei esimerkiksi estä luomasta laitetta nimeltä *Generic Nokia Colour Phone*.

Liitännäinen mahdollistaa tarvittaessa aiempien lähdekoodimallien versioiden käytön esikäännöksessä. Asetuksissa voidaan määritellä toivottu lähdekoodimallien päivämäärä, jota tuoreempia lähdekoodimalleja ei esikäännöksessä oteta huomioon.

Esikäännös voidaan suorittaa myös uuteen projektiin, jossa sovelluskehittäjä voi halutessaan tarkastella ja suorittaa esikäännettyä lähdekoodia. Esikäännetyn projektin nimi -asetus määrittelee minkä nimiseen projektiin esikäännösvaiheessa esikäännetyt tiedostot siirretään, mikäli käyttäjä suorittaa liitännäisen tarjoaman *Esikäännä projektiin* -toiminnon (Preprocess To Project).



Kuva 6.6: Preppi-liitännäisten asetukset

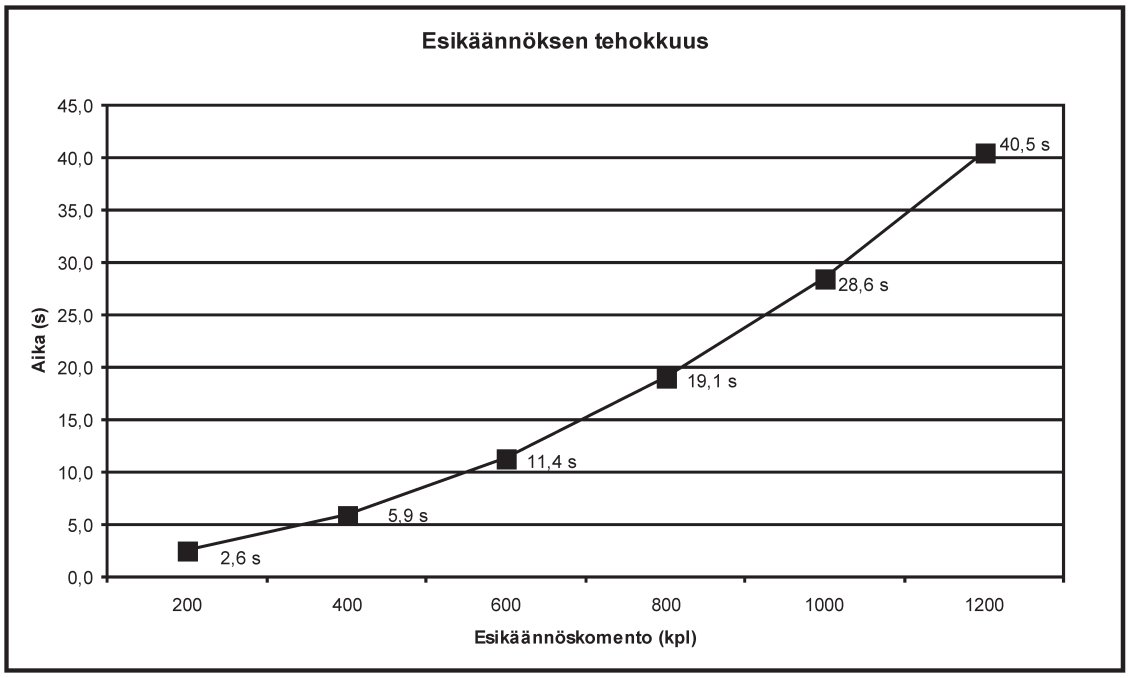
6.8 Arviointi

Preppi-liitännäinen saavutti sille taulukossa 6.1 asetetut yleiset vaatimukset, jotta sitä voidaan käytännössä hyödyntää fragmentaatiosta aiheutuvien ongelmien torjumiseen sekä lähdekoodin abstraktiotason kohottamiseen. Preppi-liitännäisen lähdekoodimalleja hyödyntäen on sovelluskehittäjällä mahdollisuus itse valita käytetty lähdekoodin abstraktiotaso määrittelemällä halutun abstraktiotason määritteleviä lähdekoodimalleja.

Preppi-liitännäistä hyödyntäen sovelluskehittäjä pystyy helposti tuottamaan selkeää ja helposti muokattavaa lähdekoodia, josta voidaan vähällä vaivalla tehdä käännökset useille eri kohdelaitteille hyödyntäen mukautettavia lähdekoodimalleja ja niiden esittelemiä esikäännöskomentoja. Preppi on myös helposti käytettävä, vaikka siinä onkin käytettävyyttä heikentäviä seikkoja, jotka esittelen luvussa 6.8.2.

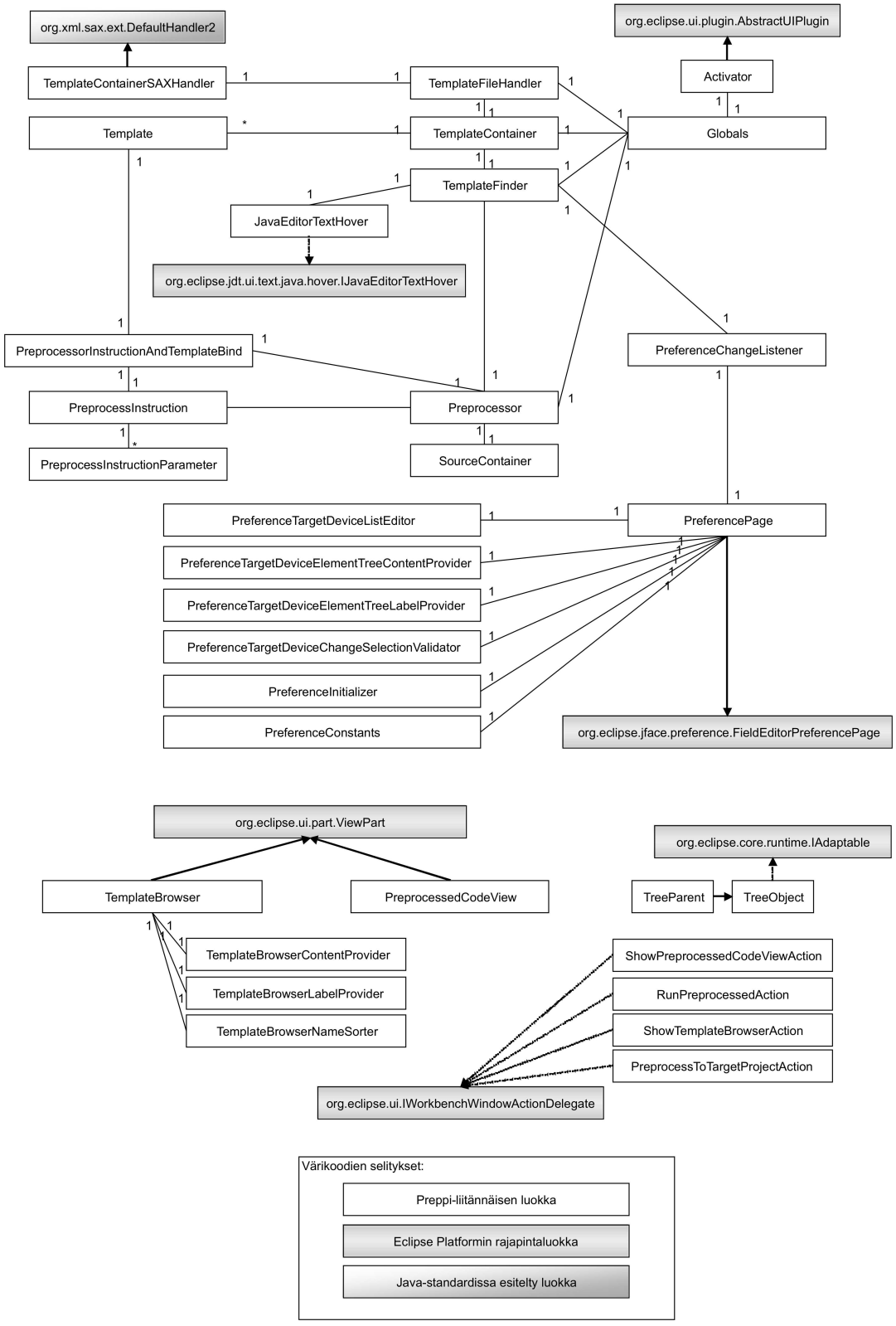
Preppi-liitännäinen itsessään on sidottu Eclipse Platformiin eikä sitä pystytä tällä hetkellä siirtämään muihin sovelluskehitysympäristöihin. Liitännäisen esikäännöstoiminnallisuus on lisäksi sidottu Java-kieleen.

Liitännäisestä toteutettiin perustoiminnallisuutta esittelevä versio, jonka tehokkuutta pystytään parantamaan optimoimalla lähdekoodimallien haku- ja vertausalgoritmeja. Tällä hetkellä lähdekoodimallit on tallennettu listamuotoiseen tietorakenteeseen, joten kun käytössä on suuri määrä erilaisia lähdekoodimalleja, voi hakujen suoritusaika kasvaa huomattavasti, mikä heikentää liitännäisen käytettävyyttä. Kuvassa 6.7 esitetään Preppi-liitännäisen esikäännöksen kulunutta aikaa käyttäen sekä parametrisoimattomia että parametrisoituja esikäännöskomentoja. Liitteessä 6 esitetään käytetty mittausympäristö sekä mittaustapa.



Kuva 6.7: Preppi-liitännäisen esikäännöksen tehokkuuden kuvaaja

Liitännäisen rakenne on pyritty pitämään mahdollisimman selkeänä pitämällä mielessä MVC-arkkitehtuuri (model-view-controller). Liitännäisen säiliöluokat eivät sisällä toiminnallisuutta, vaan toiminnallisuus on toteutettu erillisissä toiminnallisuutta sisältävissä luokissa. Jossain määrin käyttöliittymää tarjoavien luokkien kohdalla MVC-ajattelutavasta on lipsuttu jättämällä niihin yksinkertaista toiminnallisuutta. Tarkemmin liitännäisen rakennetta on esitelty luokkakaaviokuvassa 6.8.



Kuva 6.8: Luokkakaavio Preppi-liitännäisen rakenteesta

6.8.1 Mihin Preppi ei riitä

Vaikka Preppi-liitännäinen pyrkii ehkäisemään, ja varsin usein ratkaiseekin, fragmentaatiosta aiheutuvia ongelmia, ei puhdas esikäännös ilman lisäominaisuuksia tai sovelluskehittäjän tarkkaavaisuutta ratkaise kaikkia fragmentaation aiheuttamia virhetilanteita. Esikäännöksellä ei pystytä ratkaisemaan esimerkiksi kaikkia virhetilanteita, jotka syntyvät mobiililaitteen Java-toteutuksen virheistä. Tällaisia virheitä ovat esimerkiksi mobiililaitteen ennalta arvaamaton toiminta tiettyjen olioiden luonti tai niiden metodien kutsuminen laitteen kannalta väärässä kohdassa sovellusta. Esimerkiksi useissa Sony-Ericssonin valmistamissa puhelimissa `Canvas.setFullScreenMode`-metodin kutsuminen aiheuttaa virhetilanteen, mikäli kutsu ei tapahdu `Canvas`-luokan `paint`-metodissa, vaikka Java- standardi ei tällaista rajoitusta määritäkään [6]. Vastaavia sovelluksen rakenteesta johtuvia virhetilanteita, jotka varsin usein koskevat graafisia toiminnallisuuksia, ei Preppi sellaisenaan ratkaise.

Alla on lista erilaisista virhetyypeistä, joita Preppi-liitännäisen esikäännöksellä ei voida ratkaista. Virhetyypit on kerätty lähteen [6] perusteella.

- käännetyn MIDletin käsittelyn rajoitteet
- lähdekoodin rakenteellisista ominaisuuksista aiheutuvat virheet
- laitteiden yhteyskäytäntöjen protokollaerot
- muistinhallinnan virheet

6.8.2 Kehitysideoita

Tässä tutkielmassa keskityin fragmentaatio-ongelman ratkaisemiseen enkä niinkään toteutetun liitännäisen hyvän käytettävyyden toteuttamiseen. Tällä hetkellä lähdekoodimallien sisällön tarkastelu sekä muokkaaminen täytyy tehdä Template Browser -näkymän tarjoamassa editorissa. Parempi käytäntö olisi mahdollistaa kehityksen alla olevan lähdekoodin esikäännöskomentojen tuottaman sisällön tarkastelu ja muokkaaminen käytettävässä Java-editorissa hyödyntäen lähdekoodin piilotusta (code folding). Lähdekoodimallin tuottama sisältö voisi olla piilotettuna esikäännöskomennon kohdalla. Sovelluskehittäjä voisi paljastaa sisällön ja halutessaan myös muokata sitä.

Sovelluskehityksen nopeuttamiseksi lähdekoodintäydennystoiminnallisuus tulisi laajentaa tukemaan myös toteutetun liitännäisen esikäännöskomentoja. Tällä hetkellä tällaista ominaisuutta ei ole liitännäiseen ole toteutettu.

Mielestäni tärkeänä, toistaiseksi kuitenkin puuttuvana, ominaisuutena pidän liitännäisen tarjoamia varoituksia lähdekoodin rakenteen aiheuttamista virheistä. Luvussa

6.8.1 esittelin yksittäisen tapauksen, jossa virhe seuraa mobiililaitteen Java-tulkin virheellisyydestä. Tällaisia virheitä on valitettavasti useita, eikä esikäännös sellaisenaan pysty niitä kiertämään. Preppi voisi kuitenkin ohjastaa sovelluskehittäjää erilaisin ilmoituksin ja varoituksin välttämään toteuttamasta virhetilanteita aiheuttavaa lähdekoodia. Liitännäinen voisi myös suositella olemassa olevista esikäännöskomennoista, jotka toteuttavat halutun toiminnallisuuden, mikäli hän kirjoittaa lähdekoodiin käytetyn kielen komentoja, jotka voivat aiheuttaa virhetilanteita joillain mobiililaitemalleilla.

Liitännäisessä olisi hyvä olla myös erillinen näkymä, josta sovelluskehittäjä voisi tarvittaessa tarkastaa eri mobiililaitteiden ominaisuuksia kuten tuetut yhteyskäytännöt, erilaiset tiedonsyöttölaitteet ja tuetut ohjelmointirajapinnat. Näiden tietojen perusteella Preppi voisi myös varoittaa sovelluskehittäjää, mikäli hän käyttää ominaisuuksia, joita valittu kohdelaite ei tue.

Preppi-liitännäinen on itsenäinen komponentti eikä se ole yhteydessä EcliseME-liitännäiseen. Tämä aiheuttaa sen, ettei Preppi pysty suorittamaan Esikäännä ja suorita -toimintoa MIDlet-projektien osalta. Sovelluskehittäjän on ensin esikäännettävä valittu projekti uuteen projektiin ja suoritettava MIDlet esikäännetystä projektista. Mikäli Preppi liitettäisiin EclipseME:n projektin käännös- sekä suoritusrutiineihin, olisi sen käyttö huomattavasti helpompaa Java ME -sovelluksia kehitettäessä.

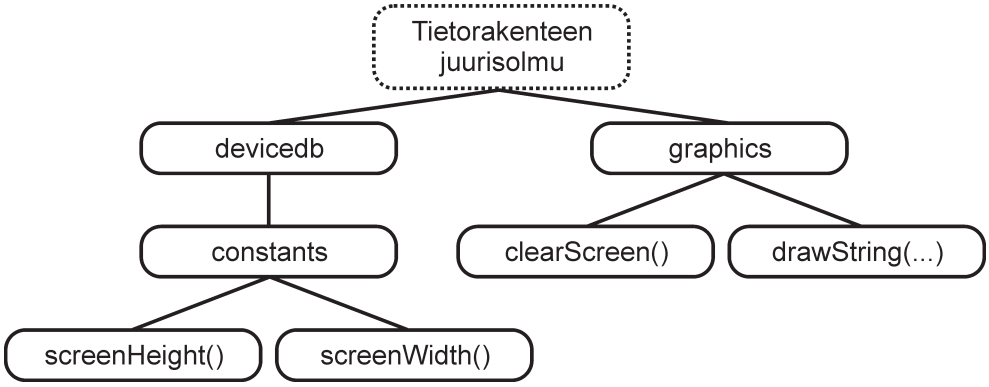
Preppi tarkastaa esikäännöskomentojen parametrien tyypit vain perustyyppien ja -luokkien, kuten esimerkiksi `int`, `double` ja `String`, mukaan. Tämä vaikuttaa myös esikäännöskomentojen ylikuormittamisen onnistumiseen, sillä oikean lähdekoodimallin löytyminen riippuu sekä lähdekoodimallin nimestä että parametrien tyypeistä. Liitännäisen puutteena tällä hetkellä on erotella saman nimiset lähdekoodimallit, mikäli niillä on sama määrä parametrejä, jotka eivät ole perustyyppisiä.

Esikäännöksen aikana esikäännöskomentoja verrataan lähdekoodimalleihin, jotka on tallennettu listarakenteeseen, josta halutun esikäännösmallin haku on vaativuudeltaan O(n). Tämä voi aiheuttaa liitännäisen esikäännöstoiminnallisuuden heikkoa tehokkuutta. Lähdekoodimallit, jotka myös määrittelevät esikäännöskomennot, voitaisiin esikäännösprosessin tehokkuuden parantamiseksi tallentaa puu-muotoiseen tietorakenteeseen ryhmitellen ne kokonaisuuksiensa mukaisesti, kuten kuvassa 6.9 on esitetty. Tämä nopeuttaisi lähdekoodimallien hakua ja siten myös Preppi-liitännäisen toteuttamaa esikäännösprosessia. Lisäksi liitännäisen esikäännöksen aikaista lähdekoodin analysointia tulisi optimoida, koska tutkimukseni tuloksena syntynyttä fragmentaatio-ongelmia ratkaisevaa esikäännöstä hyödyntävää tapaa esittelevä Preppi-liitännäinen on liian hidas sellaisenaan keskisuurissa ja suurissa sovellusprojekteissa käytettäväksi.

Määritellyt lähdekoodimallit:

devicedb.constants.screenHeight()
devicedb.constants.screenWidth()
graphics.clearScreen()
graphics.drawString(int x, int y, String str)

Puu-muotoinen tietorakenne:



Kuva 6.9: Puu-muotoinen lähdekoodimallien tietorakenne

6.8.3 Tilaajan kommentit

Tämän tutkielman tilaajana toimi Sysline Oy, jonka edustajina toimivat Niko Luojumäki ja Jukka Matilainen. Tilaajan edustajien mielestä tutkielman tarjoama lähestymismalli mobiililaitteiden fragmentaatiosta aiheutuvien ongelmien hallintaan antoi hyvän, rinnakkaisen, vaihtoehdon olemassa oleville ratkaisumalleille. Tutkielmassa esitetty tietous antaa hyvät perustiedot ja lähtökohdat tulevien ratkaisujen sunnittelulle, mikä oli lopputyön alkuperäinen tavoite.

Lopputyön aikana toteutettu Preppi-liitännäinen esitteli toteutetun ratkaisumallin hyvin käytännössä. Tämän hetkisellä ratkaisulla yritys kykenee hyvin hahmottelemaan käyttötapauksia sekä vaadittavia toiminnallisuuksia jatkossa kehitellyille ratkaisuille. Lisäksi toteutetulla liitännäisellä kyetään esitelty lähestymistapa fragmentaatioongelmien hallitsemiseksi esittelemään hyvin nopeasti muille toimijoille. Kokonaisuudessaan tutkielma vastasi yrityksen odotuksia varsin hyvin.

7 Yhteenveto

Tässä tutkielmassa esittelin mobiilisovelluskehityksen erityispiirteitä, joista keskityin mobiililaitteiden ominaisuuksien eroihin, eli fragmentaatioon, ja sen aiheuttamiin ilmiöihin.

Tutkielmassa esittelin mitä lähdekoodin abstraktiotasolla tarkoitetaan ja kuinka sitä voisi kohottaa sekä tutkin mikä olisi sopiva abstraktiotaso mobiilisovelluskehityksessä eri osa-alueilla, kuten grafiikka, ääni ja tietoliikenne. Sopivan abstraktiotason ohjelmointikielellä tarkoitan kieltä, joka peittää mahdollisimman kattavasti fragmentaation sovelluskehittäjältä ja samalla antaa hänelle mahdollisimman vapaat kädet toteuttaa sovellusta.

Sovelluskehityksessä lähdekoodin abstraktiotason kohottamisella voidaan ehkäistä useat fragmentaatiosta aiheutuvat ongelmat. Tutkielmassa tutkin eri metodeja mobiilisovellustekehityksen abstraktiotason nostamiseksi. Wrapper-luokkien ja middleware-komponenttien, esikäännöksen sekä korkean abstraktiotason välikielien analysoinnin jälkeen päädyin tulokseen, että nykyisin esikäännös on yksinkertaisin, käytetyin sekä usein myös tehokkain tapa nostaa abstraktiotasoa. Wrapper-luokkien ja middleware-komponenttien koko tulee kasvamaan fragmentaation myötä liian paljon, jotta niiden tarjoamaa toiminnallisuutta voitaisiin rajallisin ominaisuuksin varustetuissa mobiililaitteissa käyttää. Korkean abstraktiotason kielten käytössä haasteeksi muodostuvat uuden ohjelmointikielen ja työkalujen opettelu. Lisäksi korkean abstraktiotason kielillä sovelluskehittä ei usein pysty optimoimaan lähdekoodia riittävällä tarkkuudella, vaan toteutetun sovelluksen optimoinnin taso riippuu täysin käytettävistä käännöstyökaluista.

Työssä analysoin jo toteutettuja mobiililaitteiden fragmentaatiosta aiheutuvien ongelmien ratkaisemiseksi kehitettyjä työkaluja, joita ovat mm. J2ME Polish sekä NetBeans Mobility Pack. Näiden työkalujen tarjoamien ominaisuuksien sekä puutteiden perusteella toteutin Eclipse Platformille Preppi-liitännäisen, joka tarjoaa helppokäyttöiset työkalut fragmentaatio-ongelmien ratkaisuun. Toisin kuin useat muut työkalut, Preppi hyödyntää lähdekoodimalleja räätälöidyn lähdekoodin tuottamisessa esikäännettyyn lähdekoodiin. Lähdekoodimalleja käyttäen säilyy alkuperäinen esikäännöskomentoja sisältävä lähdekoodi selkeänä, toisin kuin pitkiä if-else-rakenteita käytettäessä. Preppi myös antaa sovelluskehittäjälle erittäin hyvät mahdollisuudet määritellä lähdekoodin abstraktiotaso käyttäen lähdekoodimalleja ja esikäännöskomentoja.

Tutkielman lopussa esittelin toteutetun Preppi-liitännäisen puutteita ja tiettyjä fragmentaation ilmentymiä, joita esikäännöksellä ei sellaisenaan pystytä ratkaisemaan.

8 Viitteet

[1] Aarnio Tommi, Pulli Kari, Roimela Kimmo ja Vaarala Jani, *Designing Graphics Programming Interfaces for Mobile Devices*. Computer Graphics and Applications, IEEE vuosikerta 25, numero 6, marras-joulukuu 2005, sivut 66-75. Saatavilla www-muodossa: <URL: `http://ieeexplore.ieee.org/iel5/38/32639/01528436.pdf`>, viitattu 27.3.2006.

[2] Aversano Lerina, Di Penta Massimiliano ja Baxter Ira, *Handling Preprocessor-Conditioned Declarations*. Source Code Analysis and Manipulation, Second IEEE International Workshop 1 lokakuuta 2002, sivut 83-92. Saatavilla www-muodossa <URL: `http://ieeexplore.ieee.org/iel5/8211/25179/01134108.pdf`>, viitattu 12.3.2006.

[3] Bagge Otto Skrove, Kalleberg Kalr Trygve ja Haveraaren Magne, *Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs*. Source Code Analysis and Manipulation, Third IEEE International Workshop 26.-27. syyskuuta 2003, sivut: 65-74. Saatavilla www-muodossa <URL: `http://ieeexplore.ieee.org/iel5/8773/27776/01238032.pdf`>, viitattu 12.3.2006.

[4] Debbabi Mourad, Saleh Mohammed, Talhi Chamseddine ja Zhioua Sami, *Java for Mobile Devices: A Security Study*. Communications Magazine, IEEE vuosikerta 43, numero 9, syyskuu 2005, sivut 80-85. Saatavilla www-muodossa: <URL: `http://ieeexplore.ieee.org/iel5/10467/33214/01565251.pdf`>, viitattu 28.5.2006.

[5] Enough Software, *J2ME Polish.org -kotisivu*. Saatavilla www-muodossa: <URL: `http://www.j2mepolish.org`>, viitattu 10.7.2006.

[6] Enough Software, *J2ME Polish.org - The Known Issues Database*. Saatavilla www-muodossa: <URL: `http://www.j2mepolish.org/devices/issues.html`>, viitattu 26.7.2006.

[7] Enough Software, *J2ME Polish.org - The Known Issues Database: ReducedMessageLength*. Saatavilla www-muodossa: <URL:

http://www.j2mepolish.org/devices/issue-ReducedMessageLength.html>, viitattu 24.8.2006.

[8] Eriksson Erik, *Managing Java fragmentation, Opera Software's Java ME browser client*. Saatavilla www-muodossa: <URL: http://developer.sonyericsson.com/site/global/newsandevents/latestnews/newsjune06/p_opera_mini_java_casestudy.jsp>, viitattu 14.7.2006.

[9] Ernst Michael, Badros Greg ja Notkin David, *An Empirical Analysis of C Preprocessor Use*. Software Engineering, IEEE Transactions vuosikerta 28, numero 12, joulukuu 2002, sivut 1146-1170. Saatavilla www-muodossa: <URL: http://ieeexplore.ieee.org/iel5/32/25950/01158288.pdf>, viitattu 11.7.2006.

[10] Favre Jean-Marie, *Preprocessors from an Abstract Point of View*. Software Maintenance 1996, International Conference 4.-8. marraskuuta 1996, sivut 329-338. Saatavilla www-muodossa <URL: http://ieeexplore.ieee.org/iel3/4204/12288/00565036.pdf>, viitattu 12.3.2006.

[11] Free Software Foundation Inc., *The C Preprocessor*. Saatavilla www-muodossa: <URL: http://computing.ee.ethz.ch/sepp/gcc-3.0.1-mo/cpp.html>, viitattu 17.4.2006.

[12] International Business Machines Corp., *Eclipse Platform Technical Overview*. Saatavilla www-muodossa: <URL: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf>, viitattu 16.7.2006.

[13] Lau Allen, *The Fragmentation Effect*, Java World. Saatavilla www-muodossa: <URL: http://www.javaworld.com/javaworld/jw-05-2004/jw-0524-fragment.html, viitattu 15.10.2006.

[14] Jörg Pleumann, *Antenna - An Ant-to-End Solutions For Wireless Java*. Saatavilla www-muodossa: <URL: http://antenna.sourceforge.net>, viitattu 14.7.2006.

[15] Kaijanaho Antti-Juhani, *"Ohjelmointikielten periaatteet - Syksy 2004" -luentomoniste*. Saatavilla www-muodossa: <URL:

http://www.mit.jyu.fi/antkaij/opetus/okp/2004/moniste/okp2.pdf>, viitattu 18.4.2006.

[16] Kontio Mikko, *Mobile Java™ with J2ME*, IT Press, Gummerus Kirjapaino Oy, Finland 2002.

[17] NetBeans.org, *NetBeans.org -kotisivu*. Saatavilla www-muodossa: <URL: http://www.netbeans.org>, viitattu 9.7.2006.

[18] Nokia Corporation, *Developing JavaTM Games for Platform Portability - Case Study: Miki's World*. Saatavilla www-muodossa: <URL: http://sw.nokia.com/id/63bba335-60bd-4063-972e-42dcaa9c7e2e/Java_Game_Porting_v1_0.pdf>, viitattu 18.4.2006.

[19] Nokia Corporation, *Usability, Culturally Speaking*. Saatavilla www-muodossa: <URL: http://sw.nokia.com/id/5679f372-7597-4371-b37d-dc0341b77642/Tip_of_the_Month_Usability_Culturally_Speaking_v1_0_en.pdf>, viitattu 27.7.2006.

[20] O'Shea Pamela ja Exton Chris, *An Investigation of Java Abstraction Usage for Program Modifications*. Program Comprehension, 2005. IWPC 2005. 13th International Workshop 15.-16. toukokuu 2005, sivut 65-74. Saatavilla www-muodossa <URL: http://ieeexplore.ieee.org/iel5/9727/30705/01421016.pdf>, 15-16. toukokuuta 2005, viitattu 10.3.2006.

[21] Open Source Finland, *Jappo Java Preprocessor -kotisivu*. Saatavilla www-muodossa: <URL: http://jappo.opensourcefinland.org/index.html>, viitattu 25.7.2006.

[22] Ortiz Enrique, *Unofficial Device Issues List*. Saatavilla www-muodossa: <URL: http://www.j2medeveloper.com/wiki/Wiki.jsp?page=UnofficialDeviceIssuesList>, viitattu 29.7.2006.

[23] Mikkonen Tommi, *Mobiiliohjelmointi*. Talenum Media Oy, Helsinki, 2004.

[24] PalmInfocenter.com, *Palm Infocenter*. Saatavilla www-muodossa: <URL:http://software.palminfocenter.com/platformMain.asp?category=1&subcategory=108>, viitattu 18.9.2006.

[25] Remko Popma, Azzurri Ltd., *JET Tutorial Part 1 (Introduction to JET)*. Saatavilla www-muodossa: <URL:

http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html>, viitattu 26.7.2006.

[26] Stroustrup Bjarne, *C++ -ohjelmointi*. Teknolit Oy, Jyväskylä, 2. painos, 2000

[27] Sun Microsystems, *The Java Community Process: JSR 135: Mobile Media API*. Saatavilla www-muodossa: <URL: http://www.jcp.org/en/jsr/detail?id=135>, viitattu 29.7.2006.

[28] Sun Microsystems, *The Java Community Process: JSR 232: Mobile Operational Management*. Saatavilla www-muodossa: <URL: http://www.jcp.org/en/jsr/detail?id=232>, viitattu 24.5.2006.

[29] Sun Microsystems, *The Java Community Process: JSR 248: Mobile Service Architecture*. Saatavilla www-muodossa: <URL: http://www.jcp.org/en/jsr/detail?id=248>, viitattu 4.8.2006.

[30] Sun Microsystems, *The Java Community Process: JSR 248: Mobile Service Architecture Advanced*. Saatavilla www-muodossa: <URL: http://www.jcp.org/en/jsr/detail?id=249>, viitattu 4.8.2006.

[31] Sun Microsystems, *The Java Community Process: JSR 271: Mobile Information Device Profile 3*. Saatavilla www-muodossa: <URL: http://jcp.org/en/jsr/detail?id=271>, viitattu 21.7.2006.

[32] Sun Microsystems, *The Java Verified^{TM} Program*. Saatavilla www-muodossa: <URL: http://www.javaverified.com>, viitattu 15.7.2006.

[33] Symbian Ltd, *Symbian OS 9.1 Developer Library*. Saatavilla www-muodossa: <URL: http://www.symbian.com/developer/techlib/v9.1docs/index.asp>, viitattu 28.7.2006.

[34] Symbian Ltd, *Symbian OS Developer Knowledgebase*. Saatavilla www-muodossa: <URL: http://www.symbian.com/developer/faq/index.html>, viitattu 28.7.2006.

[35] Symbian Ltd, *Symbian Signed FAQ -kotisivu*. Saatavilla www-muodossa: <URL: https://www.symbiansigned.com/app/page/faq>, viitattu 24.5.2006.

[36] The Apache Software Foundation, *Apache Ant -kotisivu*. Saatavilla www-muodossa: <URL: http://ant.apache.org>, viitattu 14.7.2006.

[37] The Eclipse Foundation, *Eclipsepedia*. Saatavilla www-muodossa <URL: http://wiki.eclipse.org>, viitattu 19.3.2006.

[38] The Eclipse Foundation, *EMF - Eclipse Modeling Framework*. Saatavilla www-muodossa: <URL: http://www.eclipse.org/emf/>, viitattu 26.7.2006.

[39] UFPE, *JaTS - Java Transformation System*. Saatavilla www-muodossa: <URL: http://www.cin.ufpe.br/~jats/>, viitattu 19.7.2006.

[40] Vidács Lászlo, Beszédes Árpád, Ferenc Rudolf, *Columbus Schema for C/C++ Preprocessing*. Software Maintenance and Reengineering, 2004. CSMR 2004. Eighth European Conference 2004, sivut 75-84. Saatavilla www-muodossa: <URL: http://ieeexplore.ieee.org/iel5/9013/28613/01281408.pdf>, viitattu 28.3.2006.

Liite 1: Ehdollisen esikäännöksen määritys- ja ehtolauseet

Ehdollisessa esikääntämisessä hyödynnetään erilaisia määritys- ja ehtolauseita. Yleisimpiä erikäännösmääritys- ja ehtolauseita ovat mm. *#define*, *#elsif*, *#else*, *#endif*, *#if*, *#ifdef*, *#ifndef*, *#include* ja *#undef* [26]. Alla on esimerkki mainittujen lausekkeiden käytöstä C Preprocessorin ymmärtämässä syntaksissa:

```
#if ehto1
  /* koodi, joka liitetään mikäli ehto1 evaluoituu tosi-arvoon. */
#elsif ehto2
  /* koodi, joka liitetään mikäli ehto2 evaluoituu tosi-arvoon. */
#else
  #ifdef maaritys1
    /* tapaukset, joissa ehto1 ja ehto 2 evaluoituvat epätosi-arvoon
       sekä maaritys1 määritelty. */
  #else
    /* tapaukset, joissa ehto1 ja ehto 2 evaluoituvat epätosi-arvoon
       ja maaritys1 määrettä ei ole määritelty. */
  #endif
#endif
```

Liite 2: NetBeans Mobility Packin esikäännöksen lauseet

NetBeans Mobility Packin esikäännöslauseet ovat: *#ifdef*, *#ifndef*, *#elifdef*, *#elifndef*, *#if*, *#else*, *endif*, *#condition*, *#debug*, *#mdebug*, *#enddebug*, *#define* ja *#undefine* [17]. Alla on lyhyt esimerkki mainittujen lausekkeiden käytöstä NetBeans-kehitysympäristössä

```
//#if mmedia
   //#if nokia
         //#if s60_ver=="1.0"
        import com.nokia.mmapi.v1
        //#elif s60_ver=="2.0"
        import com.nokia.mmapi.v2
        //#else
       import com.nokia.mmapi.def
        //#endif
    //#else
        import javax.microedition.mmapi
    //#endif
//#endif
```

Liite 3: J2ME Polishin esikäännöslauseet

J2ME Polishin esikäännöslauseet ovat: *#ifdef*, *#ifndef*, *#elifdef*, *#elifndef*, *#endif*, *#else*, *#if*, *#else*, *#elif*, *#endif*, *#debug*, *#mdebug*, *#enddebug*, *#condition*, *#define*, *#undefine*, *#foreach*, *#next* ja *#message* [5]. Alla on lyhyt esimerkki mainittujen lausekkeiden käytöstä.

```
//#if !basicInput && (polish.hasPointerEvents || polish.mouseSupported)
    doSomething();
    //#if polish.BitsPerPixel >= 8
        doSomethingColorful();
    //#else
        doSomethingDull();
    //#endif
//#elifdef doWildStuff
    doWildStuff();
//#endif

String format;
//#foreach format in polish.SoundFormat
    format = "${ lowercase( format ) }";
    System.out.println( "The audio-format " + format +
        " is supported by this device." );
//#next format
```

Liite 4: Mobiililaitteiden näppäinten signaaliarvoja Java-ympäristössä

Alla olevassa taulukossa on muuaman laitevalmistajan laitemallin osalta eri syöttönäppäinten signaaliarvoja Java-ympäristössä.

mobiililaite	**Y**	**A**	**V**	**O**	**K**	**VV**	**OV**	**T**	**S**	**LO**	**TA**	**VAL**
LG, U8290	-1	-2	-3	-4	-5	-6	-7	-16	-10	LS	EIP	EIP
LG, KG800	-1	-2	-3	-4	-5	-202	-203	-204	-200	LS	EIP	EIP
Motorola, Razr V550	-1	-6	-2	-5	-20	-21	-22	EIP	-10	LS	EIP	-23
Motorola, A1000					-10	-11						
Nokia, S40-laitteet	-1	-2	-3	-4	-5	-6	-7	EIP		LS	EIP	EIP
Nokia, S60-laitteet	-1	-2	-3	-4	-5	-6	-7	-8		TA	EIP	
Siemens, SF65					-5	105	106	EIP		LS	EIP	
Sony-Ericsson	-1	-2	-3	-4	-5	-6	-7	-8			-11	

Lyhenteiden merkitykset ovat:

Y	ylös-näppäin
A	alas-näppäin
V	vasen-näppäin
O	oikea-näppäin
K	keskinäppäin
VV	vasen valintanäppäin (left softkey)
OV	oikea valintanäppäin (right softkey)
T	tyhjennä-näppäin
S	soittonäppäin
LO	soiton lopetus -näppäin
TA	takaisin-näppäin
VAL	valikko-näppäin
LS	lopettaa sovelluksen suorittamisen
EIP	näppäintä ei ole laitteessa
TA	siirtää sovelluksen suorittamisen taustalle

Liite 5: Esimerkki esikääntämättömästä ja Preppi-liitännäisellä esikäännetystä lähdekoodista

Esikääntämätön lähdekoodi, joka sisältää Preppi-liitännäisen esikäännösmääreitä.

```
package drawStringExample;

import javax.microedition.lcdui.*;

 /**
 * Example canvas, which draws given string to mobile device's screen.
 *
 * @author Juhani Lammassaari
 * @version 1.0
 */
public class MyCanvas extends Canvas {

    /**
     * String to be drawn on the screen.
     */
    protected String strToDraw = "";

    /**
     * The constructor.
     */
    public MyCanvas() {
    }

    /* (non-Javadoc)
     * @see javax.microedition.lcdui.Canvas#paint(
     *     javax.microedition.lcdui.Graphics)
     */
    protected void paint(Graphics g) {
        /* Clear the screen */
        g.setColor(200,200,200);
        //# int width = devicedb.constants.screenWidth();
        //# int height = devicedb.constants.screenHeight();
        g.fillRect(0,0,width,height);
```

```
        /* Draw the string */
        g.setColor(0,0,0);
        //# graphics.drawString(g,strToDraw,0,0);
    }

    /**
     * Gets the string to be drawn onto the screen.
     * @return String to be drawn to canvas.
     */
    public String getStringToDraw() {
        return strToDraw;
    }

    /**
     * Sets the string to be drawn onto the screen.
     * @param strToDraw String to be drawn to canvas.
     */
    public void setStringToDraw(String strToDraw) {
        this.strToDraw = strToDraw;
    }

}
```

Yllä esitelty lähdekoodi Preppi-liitännäisen toteuttaman esikäännöksen jälkeen. Esikäännöksessä ensisijaisena kohdelaiteena on käytetty Nokia 7650 -puhelinta.

```
package drawStringExample;

import javax.microedition.lcdui.*;

/**
 * Example canvas, which draws given string to mobile device's screen.
 *
 * @author Juhani Lammassaari
 * @version 1.0
 */
public class MyCanvas extends Canvas {
```

```
/**
 * String to be drawn on the screen.
 */
protected String strToDraw = "";

/**
 * The constructor.
 */
public MyCanvas() {
}

/* (non-Javadoc)
 * @see javax.microedition.lcdui.Canvas#paint(
 *     javax.microedition.lcdui.Graphics)
 */
protected void paint(Graphics g) {
    /* Clear the screen */
    g.setColor(200,200,200);
    // Preppi template source paste. (devicedb.constants.screenWidth();)
    // Desired template: Nokia / 7650 / ver1 ver: Newest
    // -> Got: Nokia / 7650 / ver1 ver: Newest (1/21.07.2006)
    int width =176;
    // Preppi template source paste. (devicedb.constants.screenHeight();)
    // Desired template: Nokia / 7650 / ver1 ver: Newest
    // -> Got: Nokia / 7650 / ver1 ver: Newest (1/21.07.2006)
    int height =208;
    g.fillRect(0,0,width,height);
    /* Draw the string */
    g.setColor(0,0,0);
    // Preppi call. (graphics.drawString(Graphics, ANYTYPE, int, int);)
    // Desired template: Nokia / 7650 / ver1 ver: Newest
    // -> Got: Nokia / 7650 / ver1 ver: Newest (1/21.07.2006)
    graphics_drawString_1_Graphics_String_int_int(g, strToDraw, 0, 0);
}

/**
```

```
     * Gets the string to be drawn onto the screen.
     * @return String to be drawn to canvas.
     */
    public String getStringToDraw() {
        return strToDraw;
    }

    /**
     * Sets the string to be drawn onto the screen.
     * @param strToDraw String to be drawn to canvas.
     */
    public void setStringToDraw(String strToDraw) {
        this.strToDraw = strToDraw;
    }



//   ====== Preppi Preprocess method calls below ======

    // Template version: 1 for Nokia / 7650 / ver1
    private void graphics_drawString_1_Graphics_String_int_int(
            Graphics g, String str, int x, int y) {
        char chrs[] = str.toCharArray();
        g.drawChars(chrs,0,chrs.length,x,y, Graphics.LEFT | Graphics.TOP);
    }
}
```

Esikäännöksessä käytetyt lähdekoodimallit ovat:

graphics.drawString

```
<?xml version="1.0"?>
<preppitemplate prototype="void graphics.drawString(
    Graphics g, String str, int x, int y)" version="1">
  <device brand="Nokia" model="7650" version="ver1" />
  <creation time="212443" date="20060721" author="nules">
```

```
    <comments>Workaround for Nokia 7650 mobile phone. If str is over
      200 characters long, the usual drawString method crashes.
      drawChars is used instead.
    </comments>
  </creation>
  <hoverhelp>
    Draws a string to a given graphics. With Nokia 7650
    Graphics.drawChars method is used to draw to avoid
    method crash.
  </hoverhelp>
  <importlibraries>javax.microedition.lcdui.Graphics</importlibraries>
  <inlineonly>false</inlineonly>
  <sourcecode>
    char chrs[] = str.toCharArray();
    g.drawChars(chrs,0,chrs.length,x,y, Graphics.LEFT | Graphics.TOP);
  </sourcecode>
</preppitemplate>
```

devicedb.constants.screenHeight

```
<?xml version="1.0"?>
<preppitemplate prototype="int devicedb.constants.screenHeight()"
      version="1">
  <device brand="Nokia" model="7650" version="ver1" />
  <creation time="220427" date="20060721" author="nules">
    <comments></comments>
  </creation>
  <hoverhelp>Returns screen height.</hoverhelp>
  <importlibraries></importlibraries>
  <inlineonly>true</inlineonly>
  <sourcecode>
    208
  </sourcecode>
</preppitemplate>
```

devicedb.constants.screenWidth

```
<?xml version="1.0"?>
```

```
<preppitemplate prototype="int devicedb.constants.screenWidth()"
      version="1">
  <device brand="Nokia" model="7650" version="ver1" />
  <creation time="220427" date="20060721" author="nules">
    <comments></comments>
  </creation>
  <hoverhelp>Returns screen width.</hoverhelp>
  <importlibraries></importlibraries>
  <inlineonly>true</inlineonly>
  <sourcecode>
    176
  </sourcecode>
</preppitemplate>
```

Liite 6: Preppi-liitännäisen esikäännöksen tehokkuuden arvioinnissa käytetyt lähdekoodit

Preppi-liitännäisen esikäännöksen tehokkuuden arvioimiseksi toteutin parametrittoman lähdekoodimallin lisäksi yksi, kaksi, ja kolme parametria vastaaottavat lähdekoodimallit, jotka määrittelivät seuraavat esikäännöskomennot:

```
test.testEfficiency();
test.testEfficiency(int ivalue);
test.testEfficiency(int ivalue, String svalue);
test.testEfficiency(int ivalue, String svalue, double dvalue);
```

Mittasin Preppi-liitännäisen esikäännökseen kuluvan ajan millisekunteina siten, että jokainen esikäännöskomento esikäännettiin 200, 400, 600, 800, 1000 ja 1200 kertaa. Jokainen mittaus toistettiin kolme kertaa ja saaduista lähdekoodimallikohtaisista ajoista saatiin keskiarvoistamalla aikavaativuus. Kuvassa 6.7 esitetyt aikavaativuudet ovat niiden esikäännöskomentojen aikavaativuuden keskiarvo, joiden esikäännösmäärä on sama. Taulukossa L6.1 on esitetty määriteltyjen esikäännöskomentojen aikavaativuudet. Mittaukset on toteutettu 3 GHz Pentium 4 -tietokoneella, jossa on yksi gigatavu keskusmuistia. Käyttöjärjestelmänä tietokoneessa oli Windows XP SP2. Mittaukset tehtiin Eclipse-ympäristössä lisäämällä Preppi-liitännäisen lähdekoodiin konsolitulostukset esikäännöksiin kuluneista ajoista.

Esikäännöskomentojen aikavaativuudet millisekunteina

parametriton	**yksi param.**	**kaksi param.**	**kolme param.**	**Keskiarvo**
2525	2313	2573	2917	2582
5156	4953	5276	8292	5919
10067	8151	9334	17922	11369
17963	14119	14536	29646	19066
28838	23266	21620	40599	28581
38188	36756	34412	52812	40542