Mika Pakarinen

SOVELLUSOHJELMAN JA SEN GRAAFISEN KÄYTTÖLIITTYMÄN SIIRRETTÄVYYS JAVASSA

Tietojärjestelmätieteen
pro gradu -tutkielma
5.3.2001

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Pakarinen, Mika Tapani
Sovellusohjelman ja sen graafisen käyttöliittymän siirrettävyys Javassa / Mika Pakarinen
Jyväskylä: Jyväskylän yliopisto, 2001.
101 s.
Pro gradu -tutkielma

Tässä tutkielmassa tarkastellaan sovellusohjelman ja sen graafisen käyttöliittymän siirrettävyyttä Javassa sovelluskehittäjän näkökulmasta. Tärkeimmät tutkimustavoitteet ovat seuraavat: Päämääränä on ensinnäkin muodostaa kokonaisnäkemys sovellusohjelman ja sen graafisen käyttöliittymän siirrettävyydestä sekä konkretisoida sitä viitekehyksen muodossa. Työn toisena tavoitteena on kuvata ja arvioida Javan siirrettävyysominaisuuksia kehitetyn viitekehyksen pohjalta. Tarkastelu on käsitteellisteoreettista pohjautuen kirjallisuuteen sekä artikkeleihin.

Tutkimuksen keskeisimpinä tuloksina voidaan pitää seuraavia tuotoksia ja arvioita: Työssä on kehitetty siirrettävyyden yleinen viitekehys, jota voidaan hyödyntää muidenkin ohjelmointikielten siirrettävyyden arvioinnissa. Varautumalla sovellusohjelman siirrettävyyteen riittävän ajoissa voidaan saavuttaa huomattavia kustannussäästöjä. Javassa on selvästi huomioitu siirrettävyys hyvin laajasti. Javan keskeisiä siirrettävyysmekanismeja ovat Java-alusta ja sen tiukka määrittely, sijoitteluhallitsimet, kytkettävän käyttöliittymätyylin arkkitehtuuri sekä kansainvälistämisen tuki. Silti Javasta löytyy siirrettävyysongelmia, jotka ilmetessään saattavat aiheuttaa siirtämistyötä.

AVAINSANAT: Siirrettävyys, graafinen käyttöliittymä, Java

SISÄLLYS

1 JOHDANTO ... 3
2 SOVELLUSOHJELMAN SIIRRETTÄVYYS ... 7
2.1 Siirrettävyyden keskeinen käsitteistö ... 7
2.2 Sovellusohjelman siirrettävyysongelmia ... 15
2.3 Sovellusohjelman siirrettävyyden ratkaisutapoja ... 18
3 GRAAFINEN KÄYTTÖLIITTYMÄ JA SIIRRETTÄVYYS ... 23
3.1 Graafisten käyttöliittymien kehittyminen ... 23
3.2 Graafisen käyttöliittymän piirteitä ... 26
3.3 Vuorovaikutteisen järjestelmän rakenne ... 32
3.4 Siirrettävyys graafisissa käyttöliittymissä ... 35
3.5 Graafisen käyttöliittymän kansainvälistäminen ja paikallistaminen ... 37
3.6 Ratkaisuja graafisen käyttöliittymän kansainvälistämiseen ... 42
4 SIIRRETTÄVYYS OHJELMISTOPROSESSISSA ... 46
4.1 Ohjelmistoprosessi ... 46
4.2 Siirrettävyysongelmat ja ohjelmistoprosessi ... 50
4.3 Siirrettävyystekniikat ohjelmistoprosessissa ... 52
5 SIIRRETTÄVYYS JAVASSA ... 55
5.1 Javan historia ... 55
5.2 Siirrettävyyden perusmekanismit Javassa ... 57
5.2.1 Java ohjelmointikielenä siirrettävyyden näkökulmasta ... 57
5.2.2 Java-alusta ... 62
5.2.3 Lisänäkökohtia Javan siirrettävyyden perusmekanismeihin ... 65
5.3 Java-ohjelmoinnin siirrettävyysongelmia ja niiden ratkaisukeinoja ... 67
5.4 Graafinen käyttöliittymä Javassa ... 72
5.4.1 Sovelmat ja sovellukset ... 72
5.4.2 AWT ... 74
5.4.3 Kevyet käyttöliittymäkomponentit ... 75
5.4.4 Siirrettävyyden huomioiminen Javan graafisessa käyttöliittymässä .. 76
5.5 Kansainvälistäminen Javassa ... 79

5.5.1 Perusmekanismit 79
5.5.2 Merkkijonojen vertailu, sanavälit ja merkistöt 82
5.5.3 Rajoituksia 84
5.6 Johtopäätökset Javan siirrettävyydestä 85
5.6.1 Keskeisimmät siirrettävyysongelmat ja niiden ratkaisukeinot 86
5.6.2 Arvio Javan siirrettävyydestä 89
6 YHTEENVETO 93
LÄHTEET 96

1 JOHDANTO

Sovellusohjelman kehittämisvaiheessa tai sen elinkaaren myöhemmässä vaiheessa, kuten ylläpidon aikana, sovelluksesta halutaan yhä useammin tehdä siirrettävä. Taustamotiiveja on monia, kuten esimerkiksi tavoite myydä sovellusta monissa maissa erikielisinä ja eri maiden kulttuurin huomioivana. Toisaalta motiivina voi olla se, että myytävän sovelluksen halutaan toimivan useissa käyttöjärjestelmissä, kuten Windowsissa ja Unixissa. Edelleen motiivina voi WWW-pohjaisen sovelluksen tapauksessa olla sovelluksen toimiminen useissa selaimissa. Käytännön sovelluskehitystyössä siirrettävyys ja sen huomioonottaminen koetaan usein lisätaakaksi. Sellainen siitä saattaa todellisuudessa muodostuakin, mikäli ei tiedetä eikä ymmärretä, mitä kaikkia asioita sovelluskehitystyössä täytyisi huomioida.

Tässä tutkielmassa siirrettävyys nähdään Mooneyn (1995, 151) määritelmän mukaan mahdollisuutena siirtää ohjelmayksikkö toiseen ympäristöön. Ohjelmayksikkö rajataan tässä työssä hieman alkuperäistä määritelmää suppeammaksi käsittämään sovellusohjelman tai ohjelman komponentin. Ympäristö on Mooneyn määritelmässä poikkeuksellisen laaja, mikä antaa tälle työlle mielenkiintoisen pohjan siirrettävyyden tarkasteluihin.

Ohjelmien kehittämisessä siirrettävyys on perinteisesti liittynyt ohjelmien tekemiseen ohjelmointikielten standardoitujen osajoukkojen, kuten muun muassa ANSI-C:n mukaisesti. Nämä standardit eivät kuitenkaan ota kantaa graafisiin käyttöliittymiin. Graafinen käyttöliittymä on osa sovellusohjelmaa, mutta virallista standardia graafisten käyttöliittymien tekemiseen ei ole. Sen sijaan on olemassa erilaisia teollisuusstandardeja, kuten Windows- tai Motif-tyylien mukaiset graafiset käyttöliittymät. Näissä ongelmana on siirrettävyys käyttöjärjestelmäympäristöstä toiseen. Sovellusta tehtäessä olisi kuitenkin usein tarpeellista varautua sen toteuttamiseen moneen eri ympäristöön. Miksi ei siis jo ennakoitaisi tarpeita ja rakennettaisi graafisen käyttöliittymänkin osalta siirrettäviä ohjelmia? Mutta mitä sitten siirrettävän käyttöliittymän rakentamisessa tulisi huomioida jo

ennen sen rakentamista? Mitkä ovat ne periaatteet, joiden mukaan rakentaminen tulisi tehdä, jotta käyttöliittymästä tulisi siirrettävä?

Yksi keino siirrettävän graafisen käyttöliittymän rakentamiseksi on valita sovelluksen toteutuskieleksi jokin sellainen ohjelmointikieli, joka lupaa siirrettävyyttä ja jossa on kehittynyt käyttöliittymäosa. Eräs tällainen ohjelmointikieli on Java, jonka markkinoinnissa käytetään tuttua iskulausetta: "kirjoita kerran, aja kaikkialla", ja jonka käyttöliittymäosa on kehittynyt paljon aivan viime aikoina. Javaa on myös alettu käyttää yhä enenevässä määrin uusissa ohjelmistoprojekteissa. Siten Java on valittu myös tämän työn tarkastelun kohteeksi.

Siirrettävyys on ollut tutkimuksen kohteena jo pitkään (mm. Tanenbaum ym. 1978, Wallis 1982; Filipski 1985; Henderson 1988; Mooney 1990; Rowley 1996). Siirrettävyyden tutkimusta on tehty muun muassa seuraavista osa-alueista: sovellusten luokittelu siirrettävyyden näkökulmasta, siirrettävyysvaatimusten määrittely, siirrettävyyden huomiointi suunnitteluprosessissa sekä siirrettävyyden ja sen aiheuttamien kustannusten mittaus (Mooney 1995, 150).

Graafisia käyttöliittymiä on tutkittu visualisoinnin ja käytettävyyden (usability) näkökulmista (esim. Mullet & Sano 1995; Nielsen 1993). Nämä osa-alueet eivät kuitenkaan ole tämän tutkimuksen kannalta keskeisimmässä roolissa. Näiden sijaan tämän työn kannalta kiintoisampaa on tutkimus, joka kohdistuu graafisten käyttöliittymien suunnitteluperiaatteisiin ja -tekniikoihin (esim. Galitz 1996) sekä toteuttamiseen (esim. Bass & Coutaz 1992).

Java on varsin uusi ohjelmointikieli, joten on luonnollista, että sitä koskevaa tutkimusta (mm. Horstmann & Cornell 1997; Lewandowski 1998; Sood 1998; Johnson 1999) on tehty vasta viime vuosina. Javan kehittäjäyhtiö Sun Microsystems on julkaissut melko paljon Javaan liittyviä teknisiä dokumentteja (mm. Sun Microsystems Inc. 2000a, 2000b, 2000c, 2000d, 1999a, 1999b, 1999c). Javan siirrettävyyden tutkimus on tähän mennessä ollut vähäistä.

Siirrettävyydellä on läheisiä yhteyksiä muihin tutkimuksellisesti kiintoisiin käsitteisiin, kuten ohjelmien uudelleenkäyttöön (Mooney 1995, 150), yhteentoimivuuteen (Mooney 1990, 61; Pressman 1997, 67) sekä laatuun (McCall ym. 1977, Pressmanin 1997, 535 mukaan).

Tässä tutkimuksessa rajaudutaan sovellusohjelman ja sen graafisen käyttöliittymän siirrettävyyteen liittyvään ongelmakenttään. Tutkimusongelmat ovat seuraavat:

1. Millaista käsitteistöä siirrettävyyteen liittyy?
2. Minkälaisia perinteisiä ongelmia ja niiden ratkaisukeinoja siirrettävyyteen liittyy?
3. Mitkä ovat graafisen käyttöliittymän relevantit siirrettävyyskysymykset?
4. Miten siirrettävyys tulisi huomioida ohjelmistoprosessissa?
5. Miten siirrettävyyttä voitaisiin tutkia yksittäisen ohjelmointikielen osalta?
6. Miten siirrettävyysongelmia on Javassa ratkaistu?
7. Miten ohjelman ja sen graafisen käyttöliittymän siirrettävyys toteutuu Javassa?

Tutkimus on käsitteellisteoreettista ja perustuu alan kirjallisuuteen sekä artikkeleihin. Näkökulma tutkimukseen on ohjelmistotekninen sovelluskehittäjän näkökulma.

Luvun 2 tavoitteena on pyrkiä muodostamaan siirrettävyydestä selkeä ja looginen kuva käsitteellisellä tasolla. Luvussa perehdytään aiempaan siirrettävyyden tutkimukseen ja määritellään siirrettävyyden yleinen viitekehys jatkotarkastelujen pohjaksi. Lisäksi kartoitetaan keskeisimpiä siirrettävyysongelmia ja niiden ratkaisutapoja.

Luku 3 käsittelee graafista käyttöliittymää ja sen siirrettävyyttä. Luvun päämääränä on selvittää graafisen käyttöliittymän relevantit siirrettävyyskysymykset. Luvussa perehdytään graafisen käyttöliittymän kehittymiseen, piirteisiin ja rakenteeseen. Lisäksi tarkastellaan siirrettävyyttä sekä erityisesti kansainvälistämistä ja paikallistamista graafisissa käyttöliittymissä.

Luvussa 4 havainnollistetaan siirrettävyyskysymysten laajuutta yleisessä ohjelmistoprosessissa. Tarkastelun pohjaksi valitaan yleinen ja kattava ohjelmistoprosessimalli, jonka

avulla havainnollistetaan siirrettävyysongelmien laajuutta ja suhteutetaan siirrettävyys-tekniikoita ohjelmistoprosessiin.

Luvussa 5 tutkitaan ohjelman ja sen graafisen käyttöliittymän siirrettävyyttä Javassa. Tavoitteena on selvittää ja arvioida, miten Javassa on huomioitu siirrettävyys. Luvussa tarkastellaan Javan lyhyttä historiaa, siirrettävyyden perusmekanismeja sekä siirrettä-vyysongelmia ja niiden ratkaisukeinoja. Lisäksi tutkitaan Javan graafista käyttöliittymää ja sen siirrettävyyttä, sekä miten kansainvälistäminen on huomioitu Javassa. Luvussa esitetään myös arvio Javan siirrettävyydestä yleisen siirrettävyyden viitekehyksen avul-la.

Lopuksi luvussa 6 tehdään yhteenveto tutkimuksen tuloksista.

2 SOVELLUSOHJELMAN SIIRRETTÄVYYS

Tässä luvussa tarkastellaan siirrettävyyttä koskevia aikaisempia tutkimuksia. Aluksi määritellään siirrettävyyteen liittyvä keskeinen käsitteistö ja muodostetaan siirrettävyydelle yleinen viitekehys. Seuraavaksi esitellään yleisimpiä sovellusohjelman siirrettävyyteen liittyviä ongelmia. Lopuksi tarkastellaan erilaisia tapoja mahdollisten ongelmien eliminoimiseksi tai niiden vaikutusten pienentämiseksi.

2.1 Siirrettävyyden keskeinen käsitteistö

Siirrettävyydellä tarkoitetaan Mooneyn (1995, 151) mukaan mahdollisuutta ohjelmayksikön siirtämiseen toiseen ympäristöön. Ohjelmayksikkö voi tarkoittaa sovellusohjelmaa, systeemitason ohjelmaa tai ohjelman komponenttia (Mooney 1995, 151). Tämän tutkimuksen tapauksessa ohjelmayksikkö tarkoittaa sovellusohjelmaa tai ohjelman komponenttia. Ympäristö puolestaan tarkoittaa kaikkien siirretyn ja asennetun ohjelman kanssa vuorovaikutuksessa olevien tekijöiden joukkoa (Mooney 1995, 151). Tyypillisesti ympäristö käsittää prosessorin ja käyttöjärjestelmän mutta voi myös sisältää I/O-laitteet, kirjastot, tietokoneverkot ja laajemman inhimillisen tai fyysisen järjestelmän (Mooney 1995, 151).

Sommervillen (1995, 410) mielestä siirrettävyyden tavoitteena on järjestelmän kehittäminen siten, että järjestelmä on siirrettävä erilaisten alustojen (platform) välillä. Tässä järjestelmää ei ole tarkemmin määritelty, mutta edellä esitettyyn Mooneyn (1995) ohjelmayksikköön verrattuna se voisi olla lähellä sovellusohjelmaa. Alustaa voidaan ajatella Mooneyn siirrettävyyden määritelmän mukaisena ympäristönä, tosin sitä suppeampana käsitteenä.

Erittäin tiukka ja vaativa, ohjelman muuttamiseen kantaa ottava määritelmä siirrettävästä sovelluksesta on NAG:n (Numerical Algorithms Group) vaatimus, jonka mukaan siirrettävä sovellus on liikutettavissa ilman muutosta yhdestä ympäristöstä toiseen (Cowell 1977). Tämä voi olla sovelias määritelmä numeerisille sovelluksille, joilta usein edellytetään täysin samanlaista toiminnallisuutta eri ympäristöissä.

Edellä esitetyistä siirrettävyyden määritelmistä ensimmäisenä esitelty Mooneyn määritelmä soveltuu parhaiten tämän työn tarpeisiin, koska se on kattavin määritelmä.

Monet tutkijat ovat määritelleet siirrettävän ohjelman myös kustannus- tai työmääräperustaisesti. Ohjelma on Mooneyn (1995, 151) mukaan siirrettävä, jos ohjelman siirtämiskustannukset ovat vähäisempiä kuin sen uudelleenkehittämisen aiheuttamat kustannukset. Ohjelmayksikkö olisi täydellisesti siirrettävä, mikäli se voitaisiin siirtää ilman kustannuksia. Tämä ei kuitenkaan ole Mooneyn (1995, 151) mielestä käytännössä koskaan mahdollista. Sommerville (1995, 410) puolestaan vertaa ohjelman siirtämiskustannuksia sen alkuperäisen kehitystyön määrään. Jos työtä tarvitaan huomattavasti vähemmän verrattuna alkuperäisen kehitystyön määrään, ohjelma on hänen mukaansa siirrettävä. Wallis (1982, 1) pitää sovellusta siirrettävänä, mikäli se voidaan siirtää toiseen tietokoneeseen paljon pienemmällä vaivalla, kuin mitä olisi tarvittu täysin uuden sovelluksen kehittämiseen.

Siirrettävyydellä on läheisiä yhteyksiä muihin tutkimuksellisesti kiintoisiin käsitteisiin. Yksi niistä on ohjelmien uudelleenkäyttö. Mooneyn (1995, 150) mukaan siirrettävyys on eräs uudelleenkäytön muoto: siirrettävyys on tyypillisesti kokonaisen sovelluksen uudelleenkäyttöä uudella alustalla. Eräs toinen läheinen käsite on yhteentoimivuus, joka liittyy kahden sovelluksen väliseen toimivaan kommunikointiin. Pressman (1997, 67) esittää lisäksi, että uuden sovelluksen tulee sopeutua olemassa olevan järjestelmän tai tuotteen ennalta asettamiin rajoitteisiin. Mooney (1990, 61) puolestaan käyttää hieman suppeampaa määritelmää, jonka mukaan yhteentoimivuus tarkoittaa datan jakamista erityyppisten sovellusten kesken. Lähellä siirrettävyyttä on myös sovellusohjelmien laatu. Pressmanin (1997, 535) mukaan McCall ym. (1977) ovat kuvanneet siirrettävyyden yhtenä sovellusohjelman laatutekijöistä.

Siirrettävyyden päätavoitteena on Mooneyn (1990, 59) mukaan helpottaa sovelluksen siirtämistyötä sovelluksen alkuperäisestä ympäristöstä uuteen ympäristöön. Siirtämistyö voidaan jakaa kahteen osaan, jotka ovat kuljettaminen (transportation) ja mukauttaminen (adaptation) (Mooney 1990, 59).

Kuljettaminen tarkoittaa sovelluksen koodin ja siihen liittyvän datan fyysistä toimittamista uuteen ympäristöön. Fyysisen kuljettamisen ongelmiin kuuluvat yhteensopivan tiedonvälitys- tai viestintäkanavan käyttäminen ja erityyppisten esitysmuotojen muutosten tekeminen. Kuljettamiseen liittyvät ongelmat, jotka olivat etenkin 1970- ja 1980-luvuilla merkittävässä roolissa, ovat Sommervillen (1995, 410) mielestä merkitykseltään vähentyneet. Kaupalliset paineet ovat pakottaneet valmistajat tuottamaan järjestelmiä, jotka lukevat toisten valmistajien tekemiä levykkeitä ja nauhoja. Laajalle levinneen verkottumisen myötä elektroninen ohjelmien ja datan vuoropuhelu voi korvata fyysisen kuljettamisen.

Mukauttamisella Mooney (1990, 59) tarkoittaa informaation muuttamista, jota tarvitaan, jotta sovellus toimisi tyydyttävästi uudessa ympäristössä. Kuljettamiseen verrattuna mukauttaminen käsittää korkeamman tason muutoksia, jotka saattaisivat olla tarpeen ohjelman sopeuttamiseksi toimimaan uuden ympäristön piirteiden mukaisesti. Nämä piirteet ovat joko tarkoituksellisesti tai välttämättömästi erilaisia kuin aiemmassa ympäristössä. Voidaan myös ajatella, että mukautus on mikä tahansa muutos, joka täytyy tehdä sovelluksen alkuperäiseen versioon. Jonkinlaista mukautusta voidaan tehdä automaattisesti muunnosohjelmien avulla. Tästä esimerkkinä on ohjelman lähdekoodin kääntäminen uudelleen eri kääntäjällä. Yleensä kuitenkin mukauttaminen täytyy tehdä käsin.

Sovellusohjelman tai ohjelman komponentin siirrettävyyteen liittyvien tekijöiden ymmärtämiseksi kannattaa pyrkiä jäsentämään ympäristö, jossa sovellusohjelma tai ohjelman komponentti toimivat. Sommerville (1995, 410) määrittelee sovellusohjelman toimintaympäristön koostuvan seuraavista osista:

- laitteistojärjestelmästä (system hardware),
- käyttöjärjestelmästä,
- kirjastoista sekä

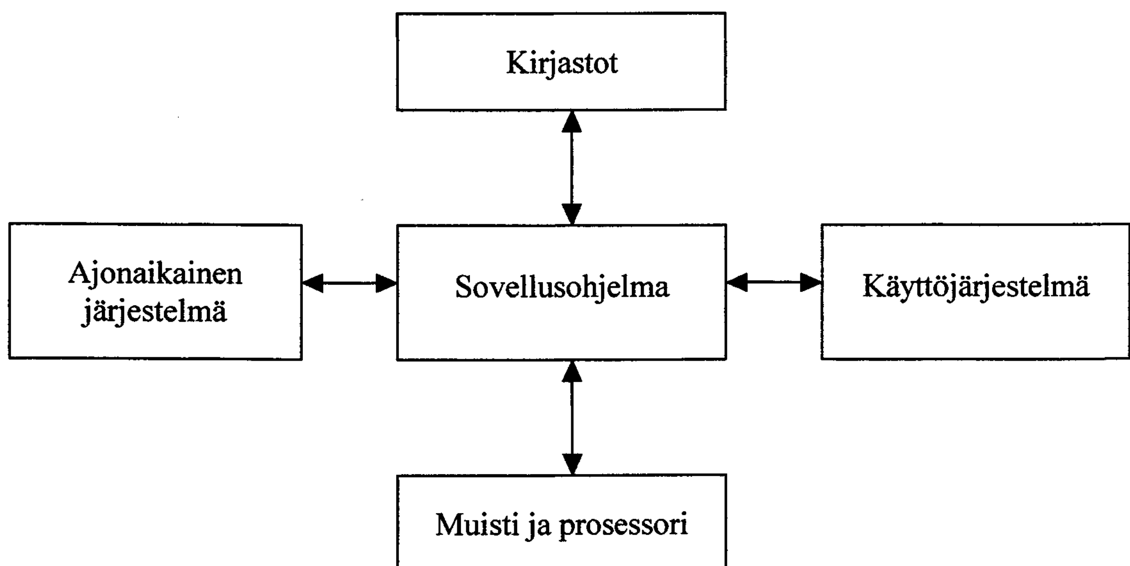
- kielen ajonaikaisesta tukijärjestelmästä (language run-time support system).

Tämä näkemys on suppeampi kuin edellä esitetty Mooneyn näkemys, jossa ympäristöön oli liitetty myös laajempi inhimillinen tai fyysinen järjestelmä sekä tietokoneverkot. Lisäksi Mooney on maininnut I/O- laitteet erikseen. Toisaalta Sommerville on Mooneystä poiketen käyttänyt yhtenä ympäristön käsitteenä kielen ajonaikaista tukijärjestelmää.

Sovelluksen ympäristön muuttuessa siitä riippuvat ohjelman osat täytyy tunnistaa ja mahdollisesti muuttaa, jotta sovellus saadaan mukautettua uuteen ympäristöönsä. Sovellus kommunikoi ympäristön kanssa rajapintojen kautta, jotka Sommerville (1995, 410—411) nimeää seuraavasti:

- muisti ja prosessori,
- käyttöjärjestelmä,
- kirjastot sekä
- ajonaikainen järjestelmä (run-time system).

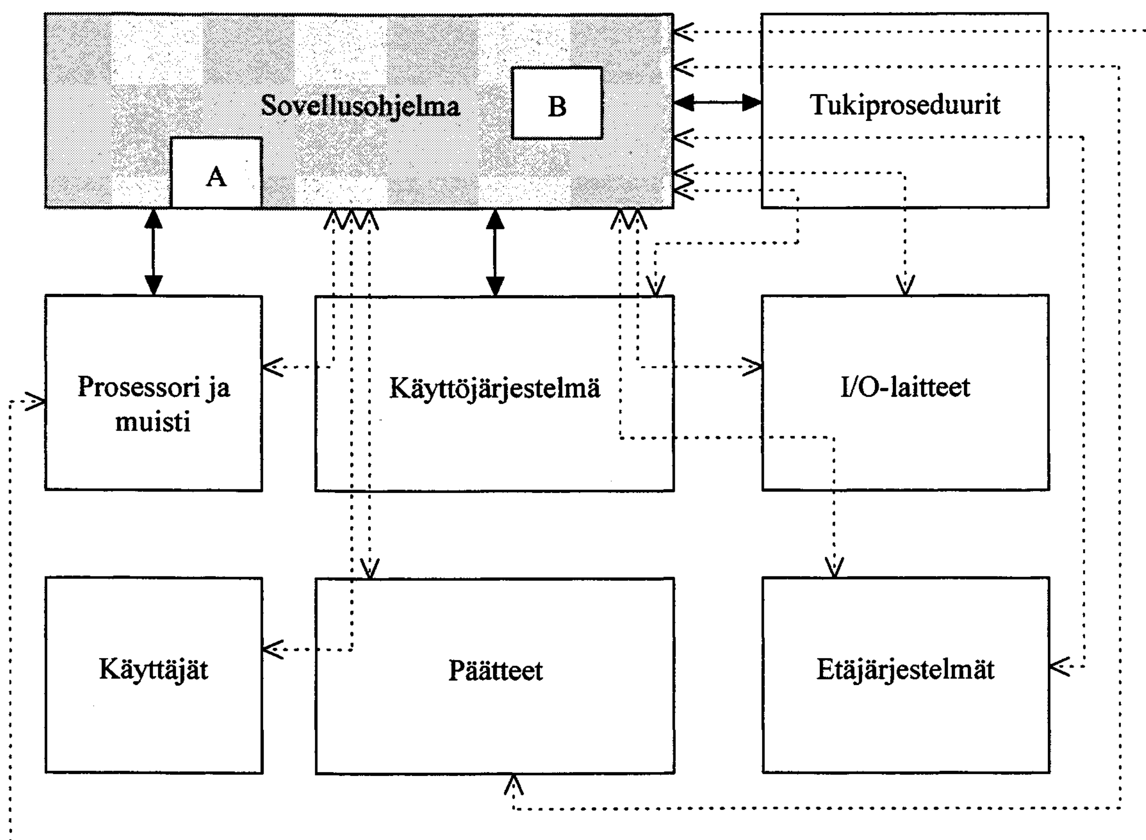
Nämä rajapinnat on esitetty kuviossa 1.



KUVIO 1. Sovellusohjelman rajapinnat Sommervillen (1995, 411) näkemyksen mukaisesti.

Ohjelma kommunikoi käyttöjärjestelmän, kirjastojen ja ajonaikaisen järjestelmän kanssa proseduuri- tai funktiokutsujen kautta (Sommerville 1995, 410). Epäsuorempaa ohjelman kommunikointi on muistin ja prosessorin kanssa. Sommerville (1995, 410) käyttää näistä myös nimitystä konearkkitehtuuri. Tämä sisältää tiedon esitysmallin, jota pohjimmiltaan käytetään huolimatta siitä, millä ohjelmointikielellä sovellus toteutetaan.

Mooneyn (1990, 61—62) näkemys on laajempi myös rajapintojen osalta. Hän jakaa sovelluksen ja sen tyypillisen ympäristön rajapinnat suoriin ja epäsuoriin rajapintoihin. Suora rajapinta on kyseessä silloin, kun sovellus kommunikoi suoraan jonkin ympäristön komponentin kanssa. Epäsuorassa rajapinnassa sovelluksella ei ole välitöntä rajapintaa ympäristön komponentin kanssa vaan välissä on jokin toinen ympäristön komponentti. Kommunikointia sovelluksen ja ympäristön komponentin kanssa on havainnollistettu kuviossa 2 siten, että yhtenäiset nuolet tarkoittavat suoraa sovelluksen ja ympäristön komponentin välistä kommunikointia ja katkonaiset nuolet epäsuoraa, välillistä kommunikointia.



KUVIO 2. Sovellusohjelman sekä sen komponenttien rajapinnat. (Mooney 1990, 61; Mooney 1995, 153, mukaeltu)

Tyypillisen sovelluksen kolme suoraa ajonaikaista rajapintaa ovat siis (Mooney 1990, 61—62):

- prosessori ja muisti,
- ajonaikaiset tukiproseduurit sekä
- käyttöjärjestelmä.

Sovellus kommunikoi prosessorin ja muistin kanssa konekäskyjen kautta. Ajonaikaisten tukiproseduurien käyttö puolestaan tapahtuu proseduurikutsujen kautta. Käyttöjärjestelmää sovellus käyttää pääasiassa järjestelmäkutsujen avulla. (Mooney 1990, 61—62.)

Neljä tyypillisen sovelluksen epäsuoraa rajapintaa ovat (Mooney 1990, 62):

- päätteet,
- käyttäjät,
- I/O-laitteet sekä
- etäjärjestelmät.

Epäsuorien rajapintojen käyttöä voidaan havainnollistaa muutaman Mooneyn (1990, 62) esimerkin avulla. Sovellus saattaa käyttää epäsuoraa rajapintaa esimerkiksi silloin, kun se käyttää I/O-laitetta. Tällöin välittäjänä on useimmiten käyttöjärjestelmä. Käyttöjärjestelmä tarjoaa myös rajapinnan muistinhallintafunktioihin ja tiettyihin prosessoriresursseihin, kuten ajoitusmekanismeihin. Korkeammalla tasolla monien sovellusten tarjoama interaktiivinen käyttöliittymä välitetään sekä käyttöjärjestelmän että käyttäjän päätelaitteen kautta. Pääsy etäjärjestelmiin on mahdollista käyttöjärjestelmän ja tietoliikennelaitteiden kautta. Mikä tahansa ohjelman ympäristön komponenteista, käyttäjää lukuunottamatta, saattaisi olla käytettävissä myös tukiproseduurien avulla.

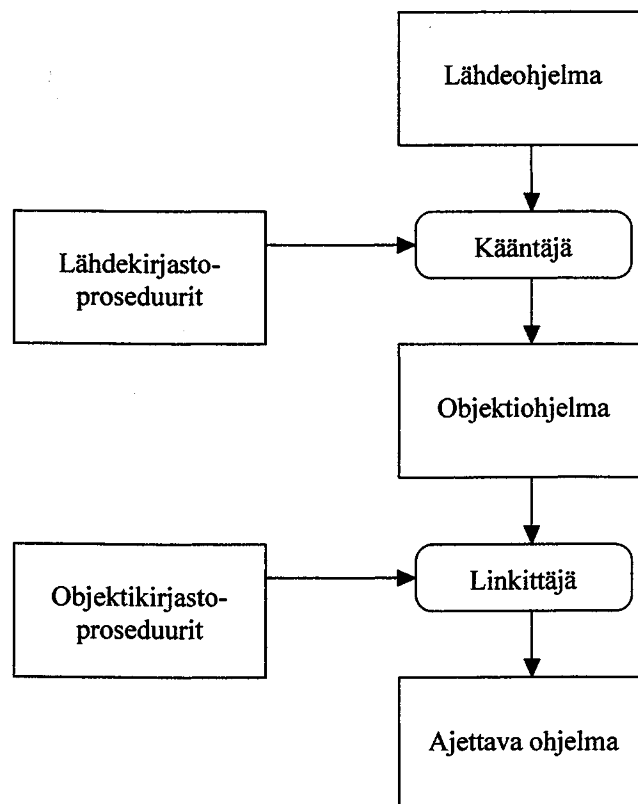
Myös uudelleenkäytettäviä komponentteja on tarkasteltu siirrettävyyden näkökulmasta (Mooney 1995, 153). Komponentit voidaan nähdä sovelluksen osina, kuten komponentit A ja B kuviossa 2. Komponenttien yksittäiset siirrettävyyskysymykset osoitetaan niiden kontaktirajapintojen avulla. Kuviossa 2 on havainnollistettu komponenttien kontaktirajapintoja siten, että sovellusohjelman reunaan sijoitetulla komponentilla A on suora systeemirajapinta, kun taas täysin sovellusohjelman sisällä sijaitsevalla komponentilla B

ei ole suoria systeemiriippuvuuksia. Kontaktirajapintojen tunnistaminen voisi auttaa suunniteltaessa komponentteja, joilla on hyvin määritellyt siirrettävyyspiirteet.

Edelliset Mooneyn määrittelemät rajapinnat edustavat ajonaikaista näkemystä sovelluksesta. Tämän lisäksi Mooney (1990, 62) erittelee ympäristön lisätekijät ja ohjelman pääesitysmuodot, jotka ovat relevantteja ennen sovelluksen ajamista. Tyypillisen ohjelman pääesitysmuodot ovat:

- lähdeohjelma,
- objektiohjelma ja
- ajettava ohjelma.

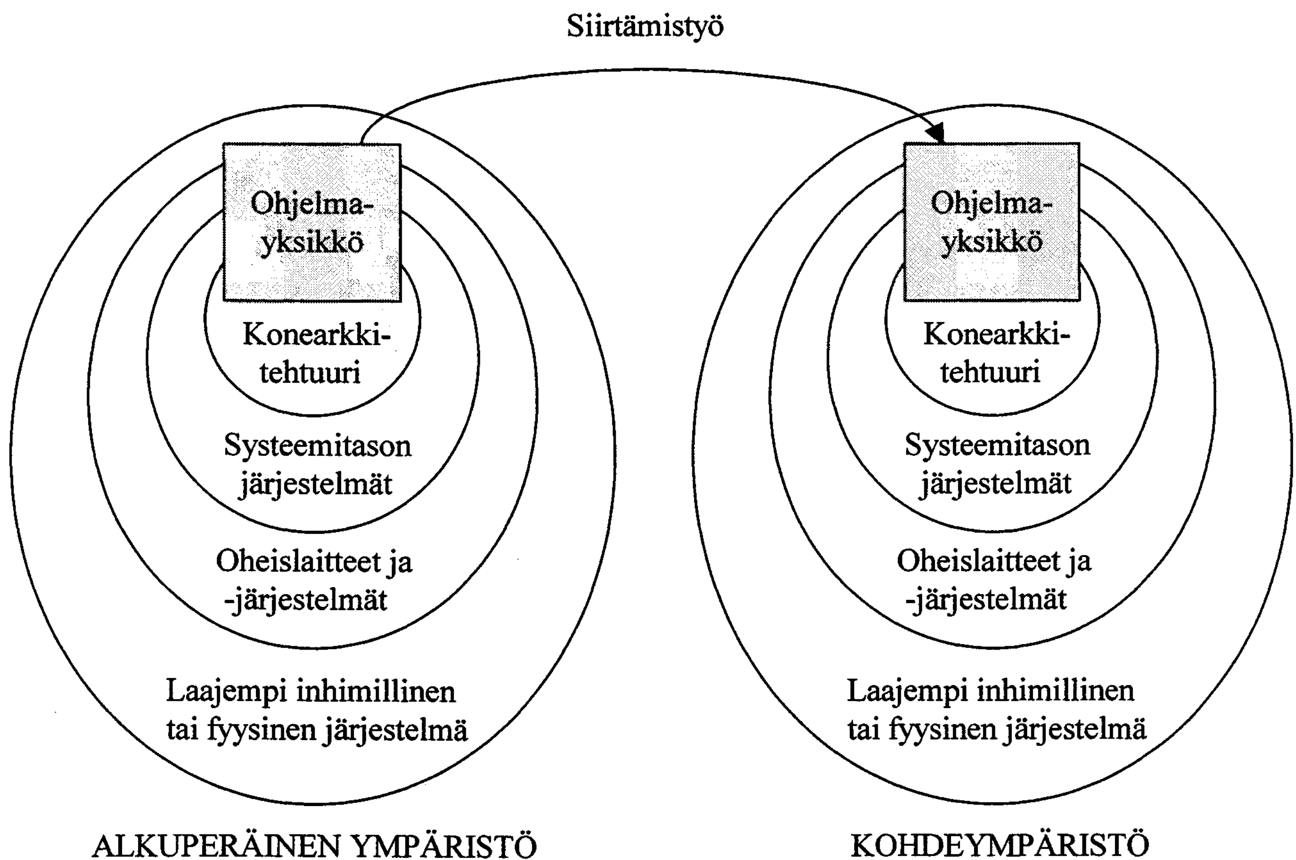
Ympäristön lisätekijöillä tarkoitetaan muuntajia, jotka muuntavat ohjelman esitysmuodosta toiseen. Yllä esiteltyjen esitysmuotojen muuntamiseen tarvitaan kääntäjä ja linkittäjä. Kuviossa 3 on havainnollistettu pääesitysmuotojen ja muuntajien rooleja, jotka ilmenevät tyypillisen ohjelmankehitystyön aikana. On kuitenkin huomattava, ettei ohjelman kehitys kaikilla ohjelmointikielillä, kuten Javalla, tapahdu tämän mallin mukaisesti.



KUVIO 3. Tyypillisen ohjelman kehittäminen ja sen esitysmuodot (Mooney 1990, 62).

Voidaan kysyä, miten tämä voisi havainnollistaa ohjelman siirrettävyyteen liittyviä asioita. Mooney (1990, 62) toteaa, että ohjelman esitysmuodot ovat ohjelman vaihtoehtoisia ilmentymiä sovelluksen ja sen ympäristötekijöiden välisiin suoriin rajapintoihin nähden. Lisäisin tähän vielä, että nämä ohjelman esitysmuodot ovat eri abstraktiotasoilla. Kunkin siirrosaskeleen aikana on Mooneyn mukaan olemassa lisämahdollisuuksia hallita rajapintojen yhteensopimattomuutta. Siten myös ohjelman esitysmuodon abstraktiotason kohottaminen voi antaa lisäkeinoja tähän.

Edellä esitettyjen eri tutkijoiden näkemysten pohjalta voidaan johtaa siirrettävyyden yleisempi viitekehys, joka havainnollistaa siirrettävyyden perusasetelman. Tämä viitekehys on esitetty kuviossa 4.



KUVIO 4. Siirrettävyyden yleinen viitekehys.

Ohjelmayksikkö voidaan nähdä edellä esitetyn Mooneyn (1995) näkemyksen mukaisesti. Siirtämistyö puolestaan vastaa aiempaa Mooneyn (1990) esitystä. Siirrettävän ohjelmayksikön alkuperäinen ympäristö ja kohdeympäristö on kuviossa 4 esitetty kehien avulla. Ohjelmayksikön ja sen ympäristön osatekijöiden välillä tapahtuu suoraa tai epäsuoraa kommunikointia. Ympäristön osatekijöistä konearkkitehtuuri on Sommervillen (1995) näkemyksen mukainen. Systeemitason järjestelmiin kuuluu esimerkiksi käyttöjärjestelmä. Oheislaitteita ja -järjestelmiä ovat muun muassa päätteet, I/O-laitteet ja etäjärjestelmät. Laajempaan inhimilliseen tai fyysiseen järjestelmään voidaan luokitella muun muassa käyttäjät, ryhmät ja organisaatiot.

Perinteisesti siirrettävyyden tutkimus on painottunut etenkin konearkkitehtuurin ja systeemitason järjestelmien tasoille sekä jonkin verran oheislaitteiden ja -järjestelmien tasolle. Kuitenkin tämän työn puitteissa siirrettävyyttä tarkastellaan näitä painotuksia kattavammin myös laajemman inhimillisen tai fyysisen järjestelmän tasolla.

Siirrettävyyden yleistä viitekehystä voidaan hyödyntää siirrettävyysongelmien ja niiden ratkaisutapojen luokittelussa. Yksittäisen ohjelmointikielen siirrettävyyttä voidaan tutkia viitekehyksen siirrettävyysympäristön osatekijäkohtaisesti. Valittua ohjelmointikieltä tutkittaessa kukin siirrettävyysympäristön osatekijä voidaan ottaa yksi kerrallaan tarkasteltavaksi, kun vakioidaan siirrettävyysympäristön muut osatekijät. Näin ohjelmointikielen siirrettävyyden tasoa voidaan paremmin arvioida tietyn siirrettävyysympäristön osatekijän suhteen. Lisäksi ohjelmointikielen siirrettävyydestä voidaan viitekehystä käyttämällä muodostaa jäsentyneempi ja perustellumpi näkemys. Siirrettävyyden yleistä viitekehystä käytetään jatkossa tässä työssä siirrettävyyteen liittyvien tarkastelujen perustana.

2.2 Sovellusohjelman siirrettävyysongelmia

Seuraavassa sovellusohjelman siirrettävyysongelmien tarkastelu aloitetaan teknisemmistä ongelmista ja päädytään inhimilliseen sekä fyysiseen järjestelmään liittyviin ongel-

miin. Teknisempiä ongelmia lähestytään Sommervillen (1995) luokittelujen kautta. Inhimilliseen ja fyysiseen järjestelmään liittyvistä ongelmista tuodaan esille esimerkkejä.

Sommerville (1995, 411, 413) luokittelee siirrettävyysongelmia sovellusohjelman ympäristörajapintojen mukaan seuraavasti:

1. konearkkitehtuuriongelmat,
2. käyttöjärjestelmäongelmat,
3. kirjasto-ongelmat sekä
4. ajonaikaisen järjestelmän ongelmat.

Konearkkitehtuuriongelmia aiheutuu silloin, kun ohjelma tekee joitakin oletuksia tietokoneen käyttämästä tiedon esittämisestä ja nämä oletukset eivät päde kaikilla koneilla. Esimerkiksi tietokoneen käyttämä sanan pituus vaikuttaa suoraan tietokoneessa saatavilla olevan kokonaisluku-tietotyypin (integer) kokoon, reaalilukujen tarkkuuteen ja merkkien määrään, jotka voidaan pakata yksittäiseen sanaan. Käyttöjärjestelmäongelmat tarkoittavat ohjelmakutsuja käyttöjärjestelmän ominaisuuksiin, jotka eivät ole saatavilla kaikissa mahdollisissa isäntäkoneissa. Kirjasto-ongelmia aiheutuu, kun ohjelma käyttää kirjastoja, jotka eivät ole saatavilla kaikissa mahdollisissa isäntäkoneissa. Tämän ongelman tavallinen epäsuora ilmentymä on kyseessä silloin, kun ohjelma luottaa tiettyyn kirjaston versioon, joka ei ole kaikkialla saatavilla. Ajonaikaiset ongelmat liittyvät siihen, että ohjelma käyttää erityisiä ajonaikaisen järjestelmän piirteitä, jotka eivät ole yleisiä. Vaihtoehtoisesti ohjelma voi luottaa joihinkin ajonaikaisen järjestelmän käyttäytymispiirteisiin, joita ei ole kaikissa toteutuksissa. Esimerkiksi ajonaikana käytettävät I/O-kutsut vaihtelevat eri koneissa ja käyttöjärjestelmissä. Tosin korkean tason ohjelmointikielissä nämä vaihtelut on piilotettu.

Sommervillen (1995, 412—413) mielestä siirrettävyysongelmat ilmenevät todennäköisimmin silloin, kun ohjelma luottaa johonkin tiedon esityksen erityiseen piirteeseen tai se käyttää käyttöjärjestelmäkutsuja, jotka tarjoavat kohdealustalta puuttuvia ominaisuuksia. Tiedon esitykseen luottaessaan ohjelma saattaa esimerkiksi sisältää olettamuksia bittien järjestyksestä datasanassa tai tuottaa koneen tukemia äärimmäisiä tietoarvoja. Esimerkkejä kohdealustakohtaisista kutsuista ovat usein tiedostojärjestelmäkutsut tai

kutsut, jotka tarjoavat erityisen prosessinhallinnan metodin. Erityinen prosessinhallinnan metodi on esimerkiksi Unixin fork, joka käynnistää uuden prosessin.

Oheislaitteisiin liittyviä siirrettävyysongelmia on otettava esille myös eksplisiittisemmin, vaikka ne kätkeytyivät Sommervillen (1995) luokituksessa lähinnä ajonaikaisen järjestelmän alle. Siirrettävässä sovellusohjelmassa ei voida välttämättä luottaa siihen, että kohdeympäristössä on kaikki samat oheislaitteet kuin alkuperäisessä ympäristössä. Kohdeympäristöstä saattaa esimerkiksi puuttua levyasema. Myöskään samantasoisten oheislaitteiden olemassaoloon kohdeympäristössä ei voida välttämättä luottaa. Kohdeympäristössä saattaa olla esimerkiksi heikkotasoisempi äänikortti, jolloin kohdeympäristössä ei välttämättä päästä yhtä hyvään äänentoiston laatuun kuin alkuperäisessä ympäristössä. Toinen esimerkki on kohdeympäristön alkuperäistä ympäristöä hitaampi verkkoyhteys, kun verkkoon ollaan yhteydessä hitaan modeemin kautta. Näppäimistöissä saattaa olla mukana järjestelmäkohtaisia näppäimiä, joita ei tule käyttää siirrettävissä sovelluksissa. Päätteisiin ja osoitinlaitteisiin liittyviä siirrettävyysongelmia käsitellään luvussa 3.4.

Eräs siirrettävyysongelmatilanne saattaa olla sovelluksen siirtäminen kulttuuriltaan ja kieleltään erilaisten käyttäjien käyttöön. Tällöin esimerkiksi päivämäärän esitysmuodon pitäisi olla paikallisen käytännön mukainen. Mikäli tämänkaltaisia asioita ei ole huomioitu sovelluksessa etukäteen, saatetaan siihen joutua tekemään myöhemmin suurempia muutoksia. Kuitenkaan tämäntyyppisiä ongelmia ei pystytä sijoittamaan edellä esitettyyn siirrettävyysongelmien kategorisointiin, joten Sommervillen (1995) luokittelu ei ole riittävän kattava. Tämä näyttää johtuvan hänen määrittelemän sovellusohjelman ympäristön suppeudesta. Määritelmässä ei ole otettu huomioon käyttäjää sovellusohjelman ympäristötekijänä.

Siirrettävyysongelmana voisi olla saman sovelluksen siirtäminen useamman organisaation käyttöön siten, että sovelluksessa huomioitaisiin kunkin organisaation tavat. Siirrettävyys voisi olla ongelmana myös ryhmätasolla, jolloin yhden sovelluksen siirtäminen useamman ryhmän käyttöön edellyttäisi kunkin ryhmän erityisten tarpeiden tyydyttämistä. Tästä voisi olla esimerkkinä erilaisten ryhmätyöskentelytapojen tukeminen. Toisaalta

edellä mainituissa konteksteissa ja siirrettävyysongelmissa tullaan lähelle käytettävyyden ongelmia, joista eräs on Nielsenin (1993, 23) mukaan sovelluksen saaminen käyttäjien tarpeita vastaavaksi. Tässä työssä ei tarkastella sovelluksen siirrettävyyttä ryhmäympäristössä eikä organisaatioympäristössä, vaan siirrettävän sovelluksen ympäristö rajataan tältä osin käsittämään vain yksittäinen sovelluksen käyttäjä. Tähän liittyvään problematiikkaan palataan luvussa 3 graafisen käyttöliittymän siirrettävyyden tarkastelun yhteydessä.

2.3 Sovellusohjelman siirrettävyyden ratkaisutapoja

Seuraavassa esiteltävät perinteiset sovellusohjelman siirrettävyystekniikat ovat suuntautuneet konearkkitehtuurin, systeemitason järjestelmien sekä oheislaitteiden erilaisuudesta aiheutuvien ongelmien ratkaisuun. Myös jäljempänä käsiteltävä standardointi keskittyy näihin osa-alueisiin liittyvien siirrettävyysongelmien vähentämiseen.

Perinteisiä tekniikoita sovellusohjelman siirrettävyyden saavuttamiseksi ovat (Sommerville 1995, 410):

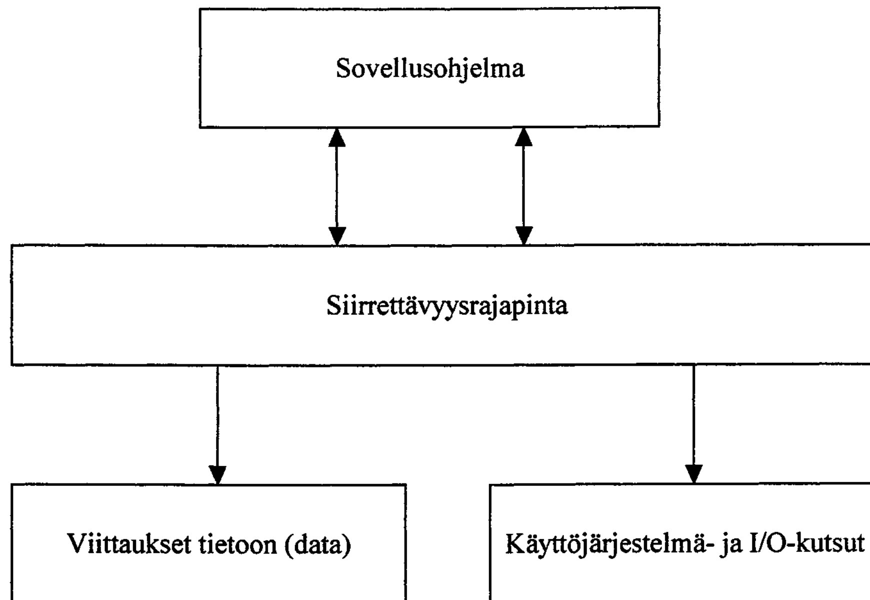
- Yhden koneen emulointi toisen koneen kautta mikrokoodia käyttäen.
- Ohjelman kääntäminen jonkun abstraktin koneen kielelle ja tämän abstraktin koneen toteuttaminen erilaisiin tietokoneisiin.
- Esiprosessoreiden käyttäminen koodin muuntamiseksi yhdestä ohjelmointikielen murteesta toiseen.

Emuloinnilla pyritään olemassa olevien ohjelmien siirrettävyyteen (Henderson 1988, 71). Siinä tarjotaan uusi ympäristö, jossa sovelluksia voidaan ajaa. Tämä ympäristö sisältää mikrokoodilla toimivan prosessorin ja vähintään I/O-toiminnot. Abstraktilla koneella tarkoitetaan prosessoria, jota käytetään abstraktin koneen käskyillä ja joka muuttaa ne kohdekoneen käskyjoukoksi (Tanenbaum ym. 1978, 682). Abstraktin koneen käskyjoukosta on myös käytetty nimitystä välikieli (Wallis 1982, 108). Esiprosessorin avulla sovelluksen tekstuaalinen koodi voidaan muuntaa ohjelmointikielen toiselle

murteelle tai kokonaan toiselle korkean tason ohjelmointikielelle (Wallis 1982, 75). Esiprosessoinnin huono puoli on, että se saattaa lisätä huomattavasti koodin määrää.

Sommerville (1995, 411—412) tarkastelee mahdollisia siirrettävyysongelmien ratkaisukeinoja esittämiensä ympäristörajapintojen mukaan. Ajonaikaisen järjestelmän ongelmat ja kirjasto-ongelmat ovat Sommervillen mielestä erityisen vaikeita ratkaista. Ajonaikaiset järjestelmät ja kirjastot ovat tavallisesti laajoja ja monimutkaisia eikä niiden lähdekoodia ole saatavilla. Standardeja pitäisi käyttää niin paljon kuin mahdollista, mutta usein standardit eivät ole kuitenkaan täydellisesti määriteltyjä. Eri toteutusten välisiä eroja on vielä olemassa. Usein tapahtuu myös niin, että kaikki eri standardikirjastojen uusien versioiden toteutukset eivät ole saatavilla yhtä aikaa eri koneille. Tästä esimerkkinä on mainittu X-ikkunointijärjestelmä. Siksi ohjelma, joka perustuu viimeiseen versioon, saattaa olla siirtokelvoton, koska kohdekoneella on vain vanhempi standardikirjastojen versio.

Konearkkitehtuuri- ja käyttöjärjestelmäriippuvuudet voidaan erottaa kehittämällä siirrettävyysrajapinta, jota on havainnollistettu kuviossa 5 (Sommerville 1995, 412). Suorien käyttöjärjestelmäkutsujen ja I/O-proseduurien käyttämisen sijaan sovellusohjelmasta kutsutaan näiden proseduurien abstrakteja versioita. Nämä abstraktit proseduurit on toteutettu käyttäen käyttöjärjestelmän ominaisuuksia. Samoin kieleen sisäänrakennettujen tietotyyppien käyttämisen sijaan pitäisi kaikki tietotyypit esittää abstrakteina tietotyyppeinä, joita sitten käytettäisiin sovellusohjelmasta. Nämä abstraktit tietotyypit toteutetaan käyttäen sisäänrakennettuja tyyppejä, kuten kokonaislukuja (integer) ja merkkijonoja (string). Kun ohjelma siirretään toiseen käyttöjärjestelmään tai konearkkitehtuuriin, vain siirrettävyysrajapinta tarvitsee toteuttaa uudelleen. Kaikki alustasta riippuvat viittaukset on ohjattu siirrettävyysrajapinnan kautta.



KUVIO 5. Siirrettävyysrajapinta (Sommerville 1995, 412).

Tämän lähestymistavan ongelmana on huomattavien kustannusten aiheutuminen kaikkien viittausten ohjaamisessa siirrettävyysrajapinnan kautta (Sommerville 1995, 412). Tällöin kustannukset kaksinkertaistuvat kutsutasolla käytettäessä käyttöjärjestelmä- tai I/O-funktioita, kun täytyy tehdä kaksi proseduurikutsua yhden sijaan. Datan käyttökustannukset alkuperäisistä tietotyypeistä ovat jopa edellisiä suurempia, kun kaikki käyttö vaatii vähintään yhden proseduurikutsun abstraktin tietotyyppirajapinnan kautta. Siksi käytännössä järjestelmäsuunnittelijat eivät yleensä eristä kaikkia käyttöjärjestelmän ja konearkkitehtuurin riippuvuuksia siirrettävyysrajapinnan taakse. Sen sijaan he käyttävät tietämystään sovelluksesta ja kohdealustasta tunnistaakseen, mitkä ohjelman osat todennäköisesti aiheuttavat siirrettävyysongelmia, kun ohjelma siirretään toiseen koneeseen. Nämä osat piilotetaan siirrettävyysrajapinnan taakse.

Yksi keskeinen siirrettävyysongelmia vähentävä tekijä on standardointi. Toimivien standardien aikaansaanti ja niiden laaja hyväksyntä sovellusten kehitysyhteisössä on Sommervillen (1995, 414) mielestä helpottanut siirrettävien sovellusten kehittämistä 1970- ja 1980-lukujen ajoista. Rowley (1996, 81) jakaa standardit kahteen luokkaan, teollisuusstandardeihin (de facto) ja virallisiin standardeihin (de jure). Teollisuusstan-

dardit määräytyvät markkina-aseman mukaan. Sen sijaan viralliset standardit on hyväksytty jossakin standardointijärjestössä. Esimerkki teollisuusstandardista on Microsoftin Win-32 -standardi. Tämän luokituksen mukainen virallinen standardi on esimerkiksi ANSI-C -standardi.

Sommerville (1995, 414, 416) luokittelee kehitetyt standardit seuraavasti:
1. ohjelmointikielistandardit,
2. käyttöjärjestelmästandardit,
3. verkkostandardit ja
4. ikkunointijärjestelmästandardit.

Virallisia ohjelmointikielistandardeja on hyväksytty C:n lisäksi muun muassa Adalle, COBOL:lle, Pascalille ja C++:lle. Sommerville (1995, 414) pitää näiden standardien mukaisia sovelluksia helposti siirrettävinä mille tahansa kääntäjälle, joka noudattaa standardeja. Kuitenkin siirrettävyysongelmia aiheutuu yhä eroista kielen kääntäjien välillä (Sommerville 1995, 414).

Microsoftin Windows-käyttöjärjestelmä on vallitseva teollisuusstandardi nykyisissä PC-laitteissa. MacOS on ollut vastaavassa asemassa Macintosh-laitteissa, jotka ovat tällä hetkellä suhteellisen vähäisen käyttäjäryhmän suosiossa. Unix on työasemissa käytetty käyttöjärjestelmä, joka on alunperin kuulunut Hendersonin (1988, 63) mukaan teollisuusstandardeihin. Kuitenkin Unixin ympärillä on ollut virallisiakin standardointipyrkimyksiä (Rowley 1996, 82—83). POSIX, joka on yleisesti käytetty nimitys IEEE:n standardista 1003.1-1988 (Portable Operating System Interface for Computer Environments), kehitettiin erilaisten Unix-versioiden yhtenäistämiseksi tunnistamaan ja määrittelemään standardi sovellusohjelmointirajapinta käyttöjärjestelmälle Unixin järjestelmäkutsujen pohjalta (Mooney 1990, 67). POSIX tarjoaa Mooneyn mukaan (1990, 67) yksityiskohtaisen määrityksen melko erityisille ympäristöille. Siten POSIX:in mahdolliset ympäristöt ovat rajallisia, mutta mahdollisten järjestelmien lukumäärä on silti suuri. Sopiville kohteille POSIX mahdollistaa korkean tason siirrettävyyden (Mooney 1990, 67). Eri Unix-toteutuksista Sommerville (1995, 416) kuitenkin huomauttaa, että niiden välillä on pieniä eroja, jotka johtuvat eri valmistajien toteuttamista järjestelmälaajen-

nuksista. Suurtietokoneilla (mainframe computer), joita käytetään muun muassa tapahtumien suorittamiseen, ei ole standardia käyttöjärjestelmää (Sommerville 1995, 416).

Joukko verkkostandardien kansainvälisiä standardointihankkeita on käynnissä kaiken aikaa. Verkkostandardeista Rowley (1996, 81) pitää TCP/IP-protokollastandardia sekä teollisuusstandardina että virallisena standardina, koska se on alunperin saanut alkunsa standardointitahojen ulkopuolelta.

X-ikkunointijärjestelmä on saavuttanut laajan suosion työasemien graafisten käyttöliittymien pohjana (Scheifler & Gettys 1986, 79). Motif-työkalusettiä (toolkit) on yleisesti käytetty X-ikkunointijärjestelmän päällä erityisenä ikkunointialustana. Luokituksessa näitä voidaan pitää teollisuusstandardeina. Microsoft Windows on teollisuusstandardi PC-koneiden enemmistössä. Näiden järjestelmien käyttöliittymätyylit (look and feel) muistuttavat toisiaan. Kuitenkin niiden mukaisia sovelluksia toteutetaan eri tavoin. Kun alunperin tiettyyn ikkunointijärjestelmään toteutettu sovellus joudutaan muuttamaan jonkin toisen ikkunointijärjestelmän mukaiseksi, aiheutuu siitä kohtalaisen korkeat kustannukset (Sommerville 1995, 416).

Nämä standardit vähentävät huomattavasti siirrettävyysongelmia, mutta kuten tarkastelusta ilmeni, standardeista huolimatta monentyyppisiä siirrettävyysongelmia esiintyy edelleen. Tämän vuoksi siirrettäviin sovelluksiin pyrittäessä sovelluskehittäjien onkin tärkeää aktiivisesti ja kriittisesti pohtia siirrettävyyskysymyksiä, vaikka he hyödyntäisivätkin sovelluskehityksessä jotakin siirrettävyyttä lupaavaa virallista standardia tai teollisuusstandardia.

3 GRAAFINEN KÄYTTÖLIITTYMÄ JA SIIRRETTÄVYYS

Tässä luvussa tarkastellaan graafista käyttöliittymää ja sen siirrettävyyttä yleisellä tasolla. Aluksi katsotaan menneisyyteen ja kerrotaan, kuinka graafiset käyttöliittymät ovat kehittyneet. Tämän jälkeen kuvataan graafiselle käyttöliittymälle tyypillisiä piirteitä. Sen jälkeen tarkastellaan vuorovaikutteisen järjestelmän rakennetta, joka on olennainen myös graafisen käyttöliittymän kannalta. Seuraavaksi graafista käyttöliittymää katsotaan siirrettävyyden valossa. Tämän jälkeen luodaan katsaus graafisen käyttöliittymän kansainvälistämiseen ja paikallistamiseen. Lopuksi tarkastellaan ratkaisuja graafisen käyttöliittymän kansainvälistämiseksi.

3.1 Graafisten käyttöliittymien kehittyminen

Seuraava graafisten käyttöliittymien kehittymisen tarkastelu vuoteen 1995 asti perustuu lähinnä Galitzin (1996, 8—9, 28—29) esitykseen. Ensimmäisenä tietokonegrafiikkaa esitti vuonna 1962 Ivan Sutherland Sketchpad-ohjelmassaan. Viivoja, ympyröitä ja pisteitä voitiin piirtää näytöllä valokynän avulla. Xerox kehitti kädessä pidettäviä osoitinlaitteita 1960-luvulla ja patentoi pyörien avulla toimivan hiiren 1970. Vuonna 1974 Xerox patentoi nykyisen pallolla toimivan hiiren. Ensimmäinen markkinoille tullut järjestelmä, jossa pääasiallisena ihmisen ja tietokoneen välisenä kommunikointimetodina olivat hiiri, osoittaminen ja valitseminen, oli vuonna 1980 julkistettu Xeroxin STAR-järjestelmä. Nämä järjestelmät esittelivät myös graafisen käyttöliittymän nykyisenkaltaisena.

Apollo julkaisi vuonna 1981 Domain-käyttöjärjestelmän (Bell 1988, 22). Domain oli Unix-pohjaisen työaseman käyttöjärjestelmä ja hajautetun tietojenkäsittelyn pioneeri.

Xerox ei koskaan kyennyt markkinoimaan STAR:ia menestyksellisesti. Apple sen sijaan otti konseptin nopeasti Macintoshiinsa. Se julkaistiin vuonna 1984 ja oli ensimmäinen menestyksekäs massamarkkinoille suunnattu järjestelmä. Sen menestystekijöihin kuuluu järjestelmän yhdenmukainen ja käyttäjäkeskeinen liittymä. Käyttöliittymän yksinkertaisuus tekee sen helposti opittavaksi, mutta samalla käyttöliittymä on joustamaton kokeneille käyttäjille.

1980-luvun puolivälissä markkinoilla oli monia työasemia. Useat näitä tarjonneet pienet yritykset eivät kuitenkaan menestyneet pitkään. (Bell 1988, 34.)

Next esitteli NeXTStepin vuonna 1988. Järjestelmä esitti ensimmäisenä simuloidun kolmiulotteisen ulkoasun komponenteilleen. Sen yksinkertainen ja käyttäjäkeskeinen liittymä ei vaadi käyttäjiltään tekniikkaan perehtymistä. Käyttöliittymä soveltuu siten kokemattomillekin käyttäjille. Se tarjoaa työkaluja, joilla käyttäjä voi muuttaa ulkoasun piirteitä mieleisikseen.

AT&T ja Sun Microsystems kehittivät ja julkaisivat vuonna 1989 Open Lookin Unix V.4 -järjestelmän toimintaympäristöksi. Macintoshiin ja NeXTStepiin verrattuna se tarjoaa enemmän toiminnallisuutta ja mahdollistaa siten enemmän tehoa ja joustavuutta kokeneelle käyttäjälle. Tämän kustannuksena kokemattomalle käyttäjälle kohdistuvat oppimisvaatimukset lisääntyvät. Open Lookille kehitettiin monia innovatiivisia ulkoasuun ja käyttäytymiseen liittyviä piirteitä, koska kehittäjät pyrkivät välttämään muiden tahojen oikeudellisia toimia. Open Look tarjoaa myös toiminnallisen spesifikaation ja tyylioppaan. Sun Microsystems käyttää Open Lookia työasemissaan.

IBM ja Microsoft kehittivät yhteistyössä OS/2 Presentation Managerin vuonna 1987. Se oli tarkoitettu MS-DOS:n korvaavaksi graafiseksi käyttöjärjestelmäksi. Myöhemmin IBM kehitti yksin vuonna 1992 julkaistun OS/2 Workplace Shellin, kun Microsoft ja IBM purkivat kaupallisen yhteistyösuhteensa. OS/2 Workplace Shell on OS/2:n graafinen käyttöliittymä versiosta 2.0 lähtien.

Digital Equipment Corporationin ja Hewlett-Packardin kehittämä, vuonna 1989 julkaistu OSF/Motif on ikkunanhallitsin ja käyttöliittymän työkalusetti. Sen ulkoasu ja käyttäytyminen perustuvat OS/2 Presentation Managerin pohjalle. Ulkoasun piirteiden muuttaminen on mahdollista. Tyyliopas tarjoaa joitakin tyylillisiä neuvoja. NeXTStepin tavoin se tarjoaa simuloidun kolmiulotteisen ulkoasun.

Microsoftin Windows luotiin alunperin vuonna 1985 graafisuuteen keskittyvänä MS-DOS:n vaihtoehtona. Ennen tätä Digital Researchin valmistama Atarin GEM-ikkunointiympäristö (Graphic Environment Management) oli ensimmäinen PC-laitteissa saatavilla ollut graafinen ympäristö (M. Sakkinen, henkilökohtainen tiedonanto 5.4.2000). Windows oli alunperin rajoittunut DOS:n suunnitteluperiaatteisiin, mutta sitä on laajennettu vuosien ajan. Vuonna 1989 julkistettiin Windows 3.0 ja vuonna 1992 versio 3.1. Näitä voidaan käyttää tehokkaasti myös näppäimistöltä, millä on pyritty helpottamaan kokeneiden DOS:n käyttäjien siirtymistä. Kokemattomille käyttäjille käyttöliittymä ei ollut kuitenkaan niin helppo kuin olisi haluttu. Microsoftin Windows 95 julkaistiin vuonna 1995. Se käsittää käyttöjärjestelmä- ja verkkopiirteitä. Windows 95 suunniteltiin ratkaisemaan monia Windows 3.1:n käytettävyysongelmia. Sen yleiset tavoitteet olivat tehdä oppiminen helpommaksi kokemattomille käyttäjille, saada se tehokkaammaksi ja muokattavammaksi kokeneille käyttäjille sekä mahdollistaa oliokeskeinen lähestymistapa.

Vuoden 1995 jälkeen Microsoft Windows 95, 98 ja 2000 -versioineen sekä Windows NT 4.0 ovat dominoineet PC-puolen markkinoita. Kannettavia tietokoneita ja langattomia päätelaitteita varten on kehitelty graafisia käyttöjärjestelmiä: muun muassa Microsoft on vuonna 1996 julkistanut Windows CE-käyttöjärjestelmän, joka pyrkii toimimaan molemmissa edellä mainituissa, ja Symbian on kehittänyt langattomiin päätelaitteisiin tarkoitetun EPOC-käyttöjärjestelmän.

3.2 Graafisen käyttöliittymän piirteitä

Käyttöliittymää voisi kuvata eräänlaisena liittymärajapintana sovellusohjelman ja käyttäjän välillä. Käyttöliittymä mahdollistaa käyttäjän ja sovellusohjelman välisen vuorovaikutuksen. Mikä sitten tekee käyttöliittymästä graafisen? Selkeimpiä tunnistetekijöitä ovat graafiset elementit eli oliot, joiden kautta käyttäjä voi tehdä operaatioita eli toimintoja (Galitz 1996, 13). Lisäksi graafisen käyttöliittymän ensisijainen vuorovaikutuksessa käytettävä apuväline on osoitinlaite.

Aikaisemmat graafiset järjestelmät etenkin PC-puolella olivat tekstipohjaisten järjestelmien jatkeita. Myös niiden taustalla olevat suunnitteluperiaatteet olivat lähtöisin aiemmista lähtökohdista. Komentopohjaisessa tekstuaalisessa liittymässä annetaan tavallisesti ensin komento "mitä tehdään" ja sitten annetaan kohde "mille toiminto tehdään". Galitz (1996, 26) käyttää tästä lähestymistavasta nimitystä sovelluskeskeisyys. Graafisten järjestelmien kehittyessä kehittäjät ajattelivat niitä tavallisesti sovellusten omien termien kautta. Kun käyttäjän toimintatapoja opittiin tuntemaan paremmin, selvisi lopulta, että ihmiset ajattelevat tehtävien termien kautta eivätkä sovelluksen termien avulla. He valitsevat ensin olioita ja sitten tekevät niille toimintoja. Sovelluskeskeistä lähestymistapaa edustaa esimerkiksi Microsoftin Windows 3.1, kun taas MacOS ja Microsoftin Windows 95 edustavat oliokeskeistä lähestymistapaa.

Sovelluskeskeinen lähestymistapa käyttää toiminto—olio -käsiteparia seuraavasti (Galitz 1996, 27):

Toiminto> 1. Sovellus avataan (esimerkiksi tekstinkäsittely).

Olio> 2. Tiedosto tai jokin muu olio valitaan (esimerkiksi dokumentti).

Oliopohjainen lähestymistapa puolestaan toimii vastaavasti (Galitz 1996, 27):

Olio> 1. Olio valitaan (dokumentti).

Toiminto> 2. Sovellus valitaan (tekstinkäsittely).

Monet kokeneet käyttäjät saattavat kokea vaikeuksia yhdestä lähestymistavasta toiseen vaihtamisessa, kun entinen vuorovaikutustyyli täytyy unohtaa ja oppia uusi lähestymis-

tapa. Missä tahansa liittymässä on kriittistä, että ylläpidetään yhdenmukaisesti joko oliopohjaista tai sovelluskeskeistä lähestymistapaa. (Galitz 1996, 27.)

Shneiderman (1982) on kuvannut graafisten järjestelmien vuorovaikutusta suoran käsittelyn (direct manipulation) avulla. Hän kutsuu graafisia järjestelmiä suoran käsittelyn järjestelmiksi. Suoran käsittelyn järjestelmä esitetään todellisen maailman laajennuksena. Siinä oletetaan, että käyttäjän kiinnostuksen kohteena olevan ympäristön oliot ja toiminnot ovat hänelle jo ennalta tuttuja. Järjestelmä yksinkertaisesti toistaa ja esittää ne eri medialla, näytöllä. Käyttäjällä on pääsy ja muokkausvaltuudet ikkunoissa oleviin olioihin. Käyttäjän työskentely tapahtuu tällöin tutussa ympäristössä ja tutulla tavalla keskittyen dataan sovelluksen ja työkalujen sijasta. Järjestelmän käyttäjälle tuntematon fyysinen rakenne on piilotettu näkyvistä eikä se ole siten suuntaamassa ajatuksia epäolennaisuuksiin. Käyttäjän älyn kohdistamista tehtäviin työkalujen sijaan Rutkowski (1982) on kuvannut läpinäkyvyydellä. (Galitz 1996, 13—14.)

Oliot ja toiminnot ovat jatkuvasti näkyvillä suoran käsittelyn järjestelmässä. Tämä rohkaisee käyttämään ihmisen voimakkaampaa tunnistusmuistia. Ongelma ”poissa näkyvistä, poissa mielestä” eliminoituu. Sama analogia on löydettävissä kirjoituspöydältä ja toimistosta, jossa kohteet ovat jatkuvasti näkyvillä. Suoritettavista toiminnoista on myös muistuttajia nähtävillä. Näitä ovat esimerkiksi nimiöidyt painikkeet, jotka korvaavat monimutkaiset syntaktiset komentonimet. Kohdistimen toiminta ja liike tapahtuvat fyysisesti selvinä ja intuitiivisesti luonnollisella tavoin. Näkyvyyteen liittyy käsite WYSIWYG (What You See Is What You Get) (Hatfield 1981). Käsitteellä tarkoitetaan sitä, että käyttäjän saatavilla olevat vaihtoehdot joko esitetään näytöllä tai voidaan saavuttaa näytöllä esitetyn kautta. (Galitz 1996, 13—27.)

Suoran käsittelyn järjestelmässä toiminnot ovat välittömiä, ja toimintoja sekä tulosten näkyvää esittämistä on enemmän. Toimintojen tulokset näytetään välittömästi visuaalisesti ruudulla niiden uudessa muodossa. Palaute voitaisiin tarjota myös äänen avulla. Käyttäjä voi nähdä toimintojen vaikutukset nopeasti. Toimintoja on myös mahdollista suorittaa toisessa järjestyksessä. Lisäksi toiminnot voidaan jättää helposti tekemättä, jos niiden havaitaan olevan virheellisiä tai eitoivottuja. (Galitz 1996, 14.)

Galitzin (1996, 14) mielestä suoran käsittelyn käsitettä olisi voitu käyttää jo ennen graafisia järjestelmiä. Varhaisimmilla täyden kuvaruudun tekstieditoreilla oli suoraan käsittelyyn viittaavia piirteitä. Niiden tekstinäytöt muistuttivat kirjoituspöydällä olevaa paperia, joka voitiin täyttää ja sitten tarkastella sitä kokonaisuutena. Muokkaus ja uudelleenjärjestely kyettiin suorittamaan helposti ja tulokset nähtiin välittömästi. Toiminnot voitiin myös tarvittaessa kääntää. Kuitenkin vasta graafisten järjestelmien tulo vakiinnutti suoran käsittelyn käsitteen.

Käytännössä kaikkien näytön olioiden ja toimintojen suora käsittely ei ole järkevää seuraavista syistä johtuen (Galitz 1996, 14—15):

- Operaatio voi olla vaikea käsitteellistää graafisessa järjestelmässä.
- Järjestelmän graafiset kyvyt saattavat olla rajoittuneita.
- Käsittelykontrollien sijoittamiseen tarvittava tila saattaa olla rajallinen ikkunassa.
- Ihmisten saattaa olla vaikeaa oppia ja muistaa kaikki tarvittavat operaatiot ja toiminnot.

Edellä esitettyjen seikkojen vuoksi tarvitaan myös epäsuoraa käsittelyä. Se korvaa symbolit sanoilla ja tekstillä, kuten alasvedettävillä tai ponnahtavilla valikoilla, sekä osoittamisen kirjoittamisella. Ikkunointijärjestelmät ovat yleensä suoran ja epäsuoran käsittelyn yhdistelmiä. Esimerkiksi valikko saadaan käyttöön osoittamalla valikkokuvaketta ja valitsemalla se. Itse valikko on kuitenkin tekstuaalinen operaatioiden lista. Kun operaatio valitaan listasta osoittamalla tai kirjoittamalla, järjestelmä suorittaa sen käskynä. (Galitz 1996, 15.)

”Poimi ja napsauta” on tyypillinen vuorovaikutustekniikka käyttäjän ja graafisen järjestelmän välillä (Galitz 1996, 24). Graafisen näytön elementit, joiden avulla jokin toiminto suoritetaan, täytyy ensin tunnistaa. Liikkumisaktiviteettiin, joka käyttäjältä vaaditaan elementin osoittamiseksi, on yleensä viitattu sanalla ”poimi”. Signaalin antoa toiminnon suorittamiseksi on kutsuttu napsauttamiseksi. Yleisimmin käytetty mekanismi tämän ”poimi ja napsauta” -toiminnon suorittamiseksi on hiiri ja sen painikkeet. Käyttäjä siirtää hiiren osoittimen relevanttiin elementtiin ja toiminto käynnistetään napsauttamalla.

Toissijainen mekanismi näiden toimintojen suorittamiseen on näppäimistö. Suurin osa järjestelmistä sallii vastaavan toiminnan suorittamisen näppäimistöä käyttämällä.

Graafiseen käyttöliittymään liittyy keskeisesti visuaalinen esittäminen. Tämä käsittää sen, mitä ihmiset näkevät näytöllä. Visuaalinen esitys on liittymän kuvallinen näkökulma. Esittämällä erikoistuneita graafisia kuvauksia helpotetaan mielikuvien muodostamista. Mielikuvien muodostaminen puolestaan on kognitiivinen prosessi, joka mahdollistaa liian runsauden tai abstraktiuden vuoksi vaikeasti havaittavan informaation ymmärtämisen. Mielikuvien muodostaminen pitää sisällään informaation asteittaisen muuttumisen, joka alkaa kohteen esityksestä ja päättyy alla olevan järjestelmän tai prosessin rakenteen tai toiminnan paljastumiseen. Toiminnan paras visuaalinen esityskeino riippuu siitä, mitä ihmiset yrittävät oppia datasta. Päämääränä ei ole tuottaa uudelleen realistinen graafinen kuva todellisuudesta vaan sellainen, joka ilmaisee relevanteimman informaation. Tehokkaat visualisoinnit voivat helpottaa älyllisiä oivalluksia, lisätä tuottavuutta sekä edistää datan nopeampaa ja tarkempaa käyttöä. (Galitz 1996, 23—24.)

Graafisessa järjestelmässä on Galitzin (1996, 24) mukaan tavoitteena heijastaa visuaalisesti käyttäjän reaalimaailmaa näytöllä niin todellisesti, tarkoituksenmukaisesti, yksinkertaisesti ja selkeästi kuin mahdollista. Graafisen järjestelmän edistynyt visuaalinen esittäminen mahdollistaa viivojen sekä kuvakkeiden näyttämisen. Se sallii myös monenlaisten merkkien kirjasintyyppien (font) näyttämisen sekä niiden koot ja tyylit. Joillakin näytöillä 16 miljoonan värin esittäminen on mahdollista. Grafiikka sallii myös animaation, kuvien ja liikkuvan kuvan esittämisen. Yleisimpiä käyttäjälle visuaalisesti esitettäviä graafisen käyttöliittymän elementtejä esitellään taulukossa 1.

TAULUKKO 1. Yleisimmät graafisen käyttöliittymän elementit (Fournier 1999, 398).

Elementti	Kuvaus
Ikkunat	Mahdollistavat käyttäjälle informaation näkemisen ja muokkaamisen sekä toimintojen käynnistämisen.
Staattinen teksti	Tekstuaalinen informaation kuvaus.
Yhden rivin editori	Mahdollistaa käyttäjälle tekstin näkemisen tai syöttämisen yhdellä rivillä.
Monen rivin editori	Mahdollistaa käyttäjälle tekstin näkemisen tai syöttämisen monella rivillä.
Listalaatikko	Esittää informaatiota vertikaalisesti listassa.
Alasvetolistalaatikko	Esittää informaatiovaihtoehtoja, kun käyttäjä painaa alaspäin osoittavaa nuolta laatikon oikealla puolella.
Kuva	Esittää informaatiota graafisessa muodossa.
Kuvakkeet (icon)	Pieniä graafisia esityksiä päätoiminnoista, sovelluksista ja tiedostoista.
Asetuspainike (check box)	Esittää kyllä—ei -informaatiota, josta käyttäjä voi tehdä valinnan.
Valintapainike (radio button)	Käyttäjä valitsee toisensa poissulkevaa informaatiota painamalla pyöreää painiketta.
Hiiren osoitin	Näyttää hiiren osoitinlaitteen sijainnin.
Vieritinpalkit	Vierittävät ikkunaa ylös tai alas, vasemmalle tai oikealle.
Komentopainike	Suorittaa painettaessa toiminnon vasteena tapahtumaan.
Kuvio	Esittää informaatiota graafista muotoa käyttäen.
Alasvetovalikko	Käyttäjä käynnistää toimintoja ikkunassa valitsemalla niitä vertikaalisesta listasta.
Työkalupalkki	Esittää kuvakepainikkeita, jotka vastaavat useimmin valittuja valikon toimintoja.
Ponnahtava valikko	Mahdollistaa käyttäjälle toimintojen käynnistämisen ikkunassa valitsemalla niitä vertikaalisesta listasta.
Vasteikkuna (response window)	Kehottaa käyttäjää ottamaan huomioon esitettävää informaatiota ja reagoimaan siihen.
Merkkikansio (tab folder)	Säiliö merkityille sivuille, jotka esittävät muita graafisia elementtejä.

Graafinen järjestelmä koostuu olioista ja toiminnoista (Galitz 1996, 25). Oliot ovat ruudulla ihmisten nähtävissä. Niitä käsitellään kuin yksittäisiä yksiköitä. Hyvin suunniteltu järjestelmä kohdistaa käyttäjien huomion olioihin eikä siihen, kuinka suorittaa toimintoja (Galitz 1996, 25). Oliot voivat koostua myös aliolioista. Esimerkiksi dokumentti voi olla olio. Dokumentin kappale, lause, sana tai kirjain voivat olla aliolioita. Tärkeäksi oliopiirteeksi graafisen järjestelmän kannalta Galitz (1996, 26) nimeää pysyvyyden. Py-

syvyys tarkoittaa tilan ylläpitoa siitä lähtien, kun se on muodostettu. Olion tilan tulisi aina automaattisesti säilyä, kun käyttäjä muuttaa sitä. Esimerkkejä olioiden tilasta ovat ikkunan koko, kohdistimen tai vierittimen sijainti. Olioilla on myös ominaisuuksia eli attribuutteja, jotka ovat olion yksilöllisiä piirteitä. Ominaisuudet auttavat olion kuvaamisessa ja käyttäjät voivat muuttaa niitä. Esimerkkejä ominaisuuksista ovat tekstityylit, kirjasintyyppien koot tai ikkunan taustavärit. Samantyyppisillä olioilla on samanlaiset piirteet ja käyttäytyminen.

Olioiden tyyppejä voidaan määritellä myös niiden välisten suhteiden avulla, kuten esimerkiksi Microsoftin Windows 95:ssä. Oliot voivat olla olemassa toisten olioiden ympäristössä ja saattavat vaikuttaa toisten olioiden ilmestymis- tai käyttäytymistapaan. Näitä suhteita kutsutaan kokoelmiksi, rajoitteiksi, koosteiksi ja säiliöiksi. Yksinkertaisin suhde on kokoelma, johon kuuluvat oliot jakavat yhteisen näkemyksen. Kokoelma saattaisi olla kyselyn tulos tai olioiden monivalinta. Operaatioita voidaan soveltaa olioiden kokoelmaan. Rajoite on vahvempi oliosuhde. Olion muuttaminen joukossa vaikuttaa johonkin toiseen joukon olioon. Esimerkki rajoitteesta on sivuiksi järjestetty dokumentti. Kooste on olemassa siellä, missä olioiden välinen suhde tulee niin merkitykselliseksi, että kokonaisuus itse voidaan tunnistaa oliona. Esimerkkejä ovat järjestettyjen taulukkolaskentasolujen joukko tai sanojen kokoelma, joka on järjestetty kappaleeksi. Säiliö on olio, jossa muut oliot ovat olemassa. Säiliöolio voisi olla esimerkiksi tekstiä sisältävä dokumentti. Säiliö vaikuttaa usein sisältönsä käyttäytymiseen. Se saattaa lisätä tai vaimentaa olioidensa tiettyjä ominaisuuksia tai operaatioita. Säiliö saattaa myös kontrolloida pääsyä sisältöönsä tai hyväksymiinsä olioihin. (Galitz 1996, 25—26.)

Graafisessa järjestelmässä käyttäjät kohdistavat operaatioita olioille. He käsittelevät olioita komennoilla tai muuttavat olioiden ominaisuuksia. Komentoja käynnistetään monin tavoin, kuten edellä kuvattujen suoran tai epäsuoran käsittelyn kautta. Komennot suoritetaan välittömästi, kun ne on käynnistetty. Esimerkkejä komennoista ovat dokumentin tulostaminen tai ikkunan sulkeminen. Ominaisuuksien määritystoiminnot muodostavat tai muokkaavat olioiden ominaisuuksia. Kun ominaisuuksien määrityksiä asetetaan, jää niiden vaikutus voimaan siihen asti, kunnes ne valitaan uudestaan. Esimerkkejä

ovat alaspudotettujen ikkunoiden näyttäminen, tietty tekstityyli tai tietty väri. Seuraavassa on tyypillinen ominaisuuden määritysjärjestys:

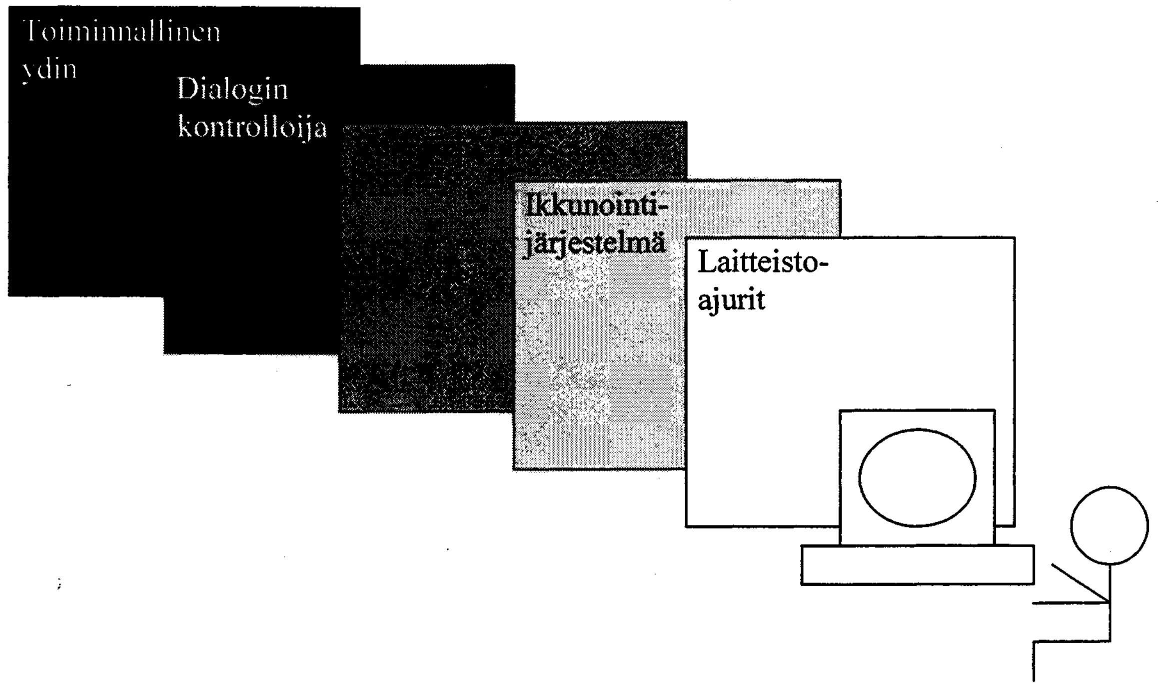
1. Käyttäjä valitsee olion, kuten esimerkiksi rivin tekstistä.
2. Käyttäjä valitsee sitten tälle oliolle sovellettavan toiminnon, kuten kursivoinnin.
3. Valittu rivi kursivoidaan ja se jää kursivoiduksi, kunnes se valitaan ja muutetaan jälleen. (Galitz 1996, 26.)

Valitulle oliolle voidaan suorittaa toimintosarjoja. Niiden suorittaminen olioista lähtien mahdollistaa järjestelmän oppimisen tutkimalla ja myös rohkaisee käyttäjää siihen. (Galitz 1996, 26.)

3.3 Vuorovaikutteisen järjestelmän rakenne

Vuorovaikutteisella järjestelmällä Bass ja Coutaz (1992, 6) tarkoittavat ihmisen kanssa vuorovaikutuksessa olevaa järjestelmää. Vuorovaikutteinen järjestelmä ei rajoitu pelkästään käyttöliittymään, vaan Bassin ja Coutazin (1992, 6) mielestä se käsittää koko toiminnallisen järjestelmän. Vaikka kyseessä pitäisi olla koko toiminnallisen järjestelmän rakenne, vuorovaikutteinen järjestelmä on keskittynyt etenkin käyttöliittymän kannalta keskeisiin rakenteisiin ja on siten tämän työn kannalta relevantti. Seuraavassa tarkastellaan vuorovaikutteisen järjestelmän rakennetta tarkemmin Bassin ja Coutazin (1992, 6—9) esityksen pohjalta.

Käyttöliittymä esittää abstraktia konetta, jonka kanssa ihminen on vuorovaikutuksessa. Abstraktien koneiden hierarkia on yksi tapa ohjelman jäsentämiseen. Kukin abstrakti kone tarjoaa tietyt palvelut yläpuolellaan oleville tasoille ja vaatii tietyt palvelut alapuolisilta tasoiltaan. Tasot tarkoittavat eri abstraktiotasoja ja ne voidaan toteuttaa eri tavoin. Eräs tapa on toteuttaa kukin taso erikseen. Vuorovaikutteinen järjestelmä jaetaan kuvion 6 mukaisiin tasoihin. (Bass & Coutaz 1992, 6, 8.)



KUVIO 6. Vuorovaikutteinen järjestelmä (Bass & Coutaz 1992, 8, mukaeltu).

Laitteistoajuri kontrolloi fyysistä laitetta. Se lähettää esimerkiksi näytölle pikselitasoisia ohjeita kohdistimen näyttämiseksi halutussa kohdassa. Laitteistoajuri parantaa ikkunointijärjestelmän riippumattomuutta yksittäisistä päätteistä. Ikkunointijärjestelmä hallitsee fyysisten laitteiden resursseja. Se kontrolloi todellisia vuorovaikutukseen kuuluvia resursseja sekä tarjoaa vuorovaikutukseen tarvittavia abstraktioita. Esimerkkinä ikkunointijärjestelmästä on X-ikkunointijärjestelmä. Vuorovaikutteinen olio on kohde, jonka käyttäjä voi havaita ja jota hän voi käsitellä fyysisten vuorovaikutuslaitteiden, kuten hiiren ja näppäimistön avulla. Vuorovaikutteinen olio käsittää sekä esityksen että vuorovaikutuksen. Esimerkki vuorovaikutteisesta oliosta on valikko. Vuorovaikutteiset oliot saadaan käyttöön esimerkiksi työkalusettien (toolkit) tarjoamien kirjastojen avulla. Dialogin kontrolloija on käyttöliittymäohjelman osa, joka kontrolloi käytettävää välitysmediaa ja käyttäjän toimintojen sarjoja. Se huolehtii toimintojen logiikasta, kuten milloin olio on käyttäjän saatavilla. Dialogin kontrolloija kontrolloi myös vuorovaikutuksen tyyliä, kuten esimerkiksi käytetäänkö valikkoja vai komentorivejä. (Bass & Coutaz 1992, 7—8, 97.)

Toiminnallinen ydin toteuttaa kohdealueen tietämyksen (Bass & Coutaz 1992, 8). Esimerkiksi elektronisen kaupankäynnin sovelluksessa toiminnallisen ytimen tulisi toteuttaa kaupankäyntiin liittyvät normit. Ihannetapauksessa toiminnallisen ytimen pitäisi Bassin ja Coutazin mukaan (1992, 8—9) jäädä täysin tietämättömäksi siitä, miten tietorakenteet ja toiminnot näytetään käyttäjälle vuorovaikutteisten olioiden kautta. Toiminnallisen ytimen pitäisi olla tietämätön myös syöttöön ja tulostamiseen käytetyistä välineistä sekä informaation välitysmuodosta (Bass & Coutaz 1992, 9). Sen ei pitäisi esimerkiksi tietää, käytetäänkö ulosannissa ikkunointijärjestelmää vai äänentoistolaitteita. Toiminnallisen ytimen ei tulisi myöskään tietää sitä, mitä kieltä käytetään tekstin esittämiseen tai tuleeko syöte valikoista vai komentoriveiltä. Toisaalta Bassin ja Coutazin (1992, 9) mielestä toiminnallisella ytimellä tulisi olla jonkinlainen välineestä riippumaton tietämys käyttäjän olemassaolosta. Tämä olisi tarpeen esimerkiksi sovellusalueeseen liittyvien virheiden ja tiettyjen operaatioiden kohteiden tunnistamisessa.

Vuorovaikutteisen järjestelmän käyttäjä kommunikoi toiminnallisen ytimen kanssa käyttöliittymätasojen eli laitteistoajurin, ikkunointijärjestelmän, vuorovaikutteisten olioiden ja dialogin kontrolloijan kautta. Käyttäjästä lähtien tasojen abstraktius kasvaa. Kullakin välitasolla, joita ovat laitteistoajurien ja toiminnallisen ytimen väliset tasot, on kaksi toimintaa. Välitason täytyy kontrolloida omaa osuuttaan koko käyttöliittymästä sekä tarjota sovitus yläpuolisille ja alapuolisille tasoilleen. Siten tasojen väliset liittymät ovat tärkeitä missä tahansa toteutuksessa. (Bass & Coutaz 1992, 9.)

Edellä esitetty jäsennys soveltuu esimerkiksi X-ikkunointijärjestelmän päälle perinteisellä tavalla rakennettavan Motif-pohjaisen käyttöliittymän kuvaukseen. Tulee kuitenkin huomata, että jäsennyksen kerroksia ei ole toteutettu näin selkeästi erillisinä kaikissa muissa graafisissa järjestelmissä. Esimerkiksi Windowsissa ikkunointijärjestelmä ja perustyökalusetti ovat yhtenäisenä osana.

3.4 Siirrettävyys graafisissa käyttöliittymissä

Graafinen käyttöliittymä on osa sovellusohjelmaa, joten graafisen käyttöliittymän siirrettävyyteen sisältyvät sovelluksen normaalia siirrettävyyttä koskevat asiat. Keskeistä graafisen käyttöliittymän siirrettävyyttä tarkasteltaessa on inhimillinen ja fyysinen ympäristö, koska graafinen käyttöliittymä on tiiviissä vuorovaikutuksessa käyttäjän kanssa. Siten Mooneyn (1995) esittämät kulttuuriseen mukauttamiseen liittyvät ongelmat ovat relevantteja graafisen käyttöliittymän siirrettävyyden tarkastelussa.

Kulttuurinen mukauttaminen (cultural adaptation) tarkoittaa sitä, että sovelluksen käyttäytyminen täytyy mukauttaa uuden ympäristön ja aiottujen käyttäjien tapoihin (Mooney 1995, 155). Kulttuurisen mukauttamisen osa-alueita on lueteltu seuraavassa. Ensiksikin hyvin tunnettua on kansainvälistämisen (internationalization) tarve, jotta voitaisiin tukea tietyn kansan tai kulttuurin mieltymyksiä muun muassa kielen, päiväyksen, valuutan sekä värien käytössä. Toiseksi voi olla tarve mukauttaa käyttöliittymä kohteena olevan tietokonejärjestelmän tapoihin tai resursseihin. Tästä ovat esimerkkinä eri järjestelmissä käytetyt erilaiset ikkunointijärjestelmät, joiden mukaiseksi sovellus saatetaan joutua mukauttamaan. Kolmanneksi käyttäytyminen voidaan joutua mukauttamaan uudelle kokeneelle käyttäjäryhmälle. Tässä käyttäjäryhmällä tarkoitetaan yksittäisiä kokeneita käyttäjiä eikä kyseessä ole ryhmätyösovellus. Mukauttaminen kokeneille käyttäjille voisi tapahtua tarjoamalla heille oma toimintatapa, josta Mooney käyttää nimitystä "eksperttimoodi".

Keskeisimpiä graafisen käyttöliittymän siirrettävyyskysymyksiä oheislaitteiden puolella ovat näyttöjen ja osoitinlaitteiden erilaisuus. Näyttöjä on monenkokoisia ja niiden käyttämät erottelutarkkuudet eivät ole samoja. Tästä aiheutuu ongelmia sovelluksen graafiselle käyttöliittymälle, jos asiaa ei ole huomioitu sitä tehtäessä. Huonossa tapauksessa saattaa käydä niin, että sovelluksen graafinen käyttöliittymä ei mahdu kokonaisuudessaan pienelle näytölle tai sitten koko ruudun kokoiseksi avautuvan graafisen käyttöliittymän elementit liikkuvat mielivaltaisesti paikoiltaan, kun sovellus avataan alkuperäistä suuremmalle näytölle. Paras ratkaisu olisi tehdä graafinen käyttöliittymä skaalautuvaksi eikä kiinnittää sitä tiettyyn näytön erottelutarkkuuteen. Tällöin suuremmissa näytöissä

voitaisiin hyödyntää ruudun kokoa näytön normaaliasetuksilla. Jos graafisen käyttöliittymän koko joudutaan jostakin syystä kiinnittämään, kannattaa kohdenäytön erottelutarkkuus valita riittävän pieneksi, jotta sovellus on näytettävissä useilla näytöillä. Tällöin esimerkiksi erottelutarkkuus 640x480 tai 600x800 voisi olla käyttökelpoinen. Näyttöjen käyttämät värimäärät vaihtelevat. Tämä ongelma koskee etenkin graafisessa käyttöliittymässä esitettäviä kuvia, ikoneja ja muuta grafiikkaa. Jos kohdekoneessa on vähemmän värejä käytössä kuin mitä kuvat käyttävät, niin kuvat näyttävät alkuperäisiä huonolaatuisemmilta. Asia kannattaa huomioida kuvia, ikoneja ja muuta grafiikkaa tuotettaessa vähentämällä niissä käytettävien värien määrää.

Osoitinlaitteista yleisin on hiiri. Hiiren painikkeiden määrä ei ole vakio kaikissa koneissa. Esimerkiksi Macintosh-koneissa käytetään tyypillisesti vain yhden painikkeen hiiriä. Sen sijaan PC-koneissa hiiren painikkeita saattaa olla 2 tai 3. Lisäksi käytössä saattaa olla painikkeiden keskelle sijoitettu ”rulla”, jolla voi kelata esimerkiksi tekstiä alaspäin rullattavassa tekstikentässä. Nämä eroavaisuudet tulisi huomioida siirrettäviä sovelluksia tehtäessä.

Näytöillä esitettäviin kirjasintyyppeihin liittyy siirrettävyysongelmia. Alkuperäisessä ympäristössä käytettyjen kirjasintyyppien nimet saattavat olla erilaisia kohdejärjestelmässä. Jos kirjasintyypin tekninen esitystapa on pikseliperustainen, se voi näyttää erilaiselta näytön erottelutarkkuuden vaihtuessa (Bass & Coutaz 1992, 68). Tällöin myös tulostuksen tarkkuus rajoittuu kirjasintyypin merkkien bittikarttojen erottelutarkkuuteen. Sen sijaan muotoperustaiset kirjasintyypit toimivat Bassin ja Coutazin (1992, 68) mukaan paremmin erilaisilla näytöillä. Ne tarjoavat myös paremman erottelutarkkuuden tulostuksessa.

3.5 Graafisen käyttöliittymän kansainvälistäminen ja paikallistaminen

Galitz (1996, 581) määrittelee *kansainvälistämisen* (internationalization) prosessiksi, joka erottaa kulttuurikohtaiset elementit tuotteesta. *Paikallistamisen* (localization) Galitz (1996, 581) näkee prosessina, jossa lisätään tietty kulttuurikonteksti aiemmin kansainvälistettyyn tuotteeseen. *Paikallisuus* (locale) määrittää paikan, joka käsittää kielen ja maan tai kulttuurin (Arnold & Gosling 1998, 335). ISO (International Standard Organization) on standardoinut paikallisuuden kieli- ja maakoodit. Seuraavassa keskitytään tarkastelemaan kansainvälistämistä lähinnä Nielsenin (1993, 237—254) esityksen pohjalta. Hän tarkastelee ongelmaa kansainvälisen käyttöliittymän käsitteen kautta: Kansainväliset käyttöliittymät on tarkoitettu käytettäviksi useammassa kuin yhdessä maassa. Kansainvälisten käyttöliittymien suunnittelu voi sisältää kielen kääntämistä, mutta sen pitäisi huomioida myös muiden maiden ja kulttuurien erityiset tarpeet. Tämän perusteella Nielsenin näkemys on lähellä Mooneyn ja Galizin näkemyksiä.

Graafisessa käyttöliittymässä käytettävät kuvakkeet ja värien herättämät assosiaatiot eivät välttämättä ole yleismaailmallisia (Nielsen 1993, 239). Kuvakkeet voidaan luokitella kolmeen kategoriaan niiden graafisen suunnittelun perusteella (mm. Lodding 1983, Nielsenin 1993, 239 mukaan):

1. samannäköiset kuvakkeet,
2. viittaavat kuvakkeet ja
3. mielivaltaiset kuvakkeet.

Samannäköiset kuvakkeet muistuttavat fyysistä kohdetta, jota kuvakkeen on tarkoitus esittää. Tästä esimerkkinä on kirjekuoren kuva, jota käytetään sähköpostiviestin kuvaamiseen. Yleensä samannäköiset kuvakkeet tunnistetaan monissa maissa ainakin silloin, kun ne kuvaavat yleistä kohdetta. *Viittaavat kuvakkeet* kuvaavat jotakin kohdetta, joka edustaa viittauksen tai analogian kautta jotakin käsitettä. Viittaavia ikoneja on kutsuttu joskus myös symbolisiksi ikoneiksi. Vaihtelevat kansalliset tavat saattavat vähentää viittaavien kuvakkeiden käyttökelpoisuutta. Esimerkki viittaavasta kuvakkeesta on puristimen kuvan käyttö kuvaamaan tiedoston kompressointia. *Mielivaltaiset kuvakkeet*

ovat mielivaltaisia hahmoja, joilla on tarkoitus ainoastaan kansallisten tai kansainvälisten tapojen kautta. Tämän tyyppiset kuvakkeet ovat käyttäjälle vaikeimpia oppia, elleivät ne ole laajasti käytössä. Esimerkki laajasti kansainvälisessä käytössä olevista mielivaltaisista kuvakkeista ovat liikennemerkit, jotka ovat siten hyviä tietokoneessa käytettävien kuvakkeiden lähteitä. Samannäköisten kuvakkeiden käyttö kansainvälistämisessä on suositeltavampaa kuin viittaavien ja mielivaltaisten kuvakkeiden käyttäminen.

Nielsen (1993, 241) väittää, että eräs perinteinen ongelma graafisten käyttöliittymien kansainvälistämisessä on graafisten suunnittelijoiden puuttuminen siirtämisprosessista. Graafisia suunnittelijoita tarvittaisiin kuvakkeiden suunnittelussa, koska niiden uudelleensuunnittelu vaatii graafisen suunnittelun taitoa.

Grafiikka, joka ei ole kuvakkeen muodossa, voi Nielsenin (1993, 241) mukaan johtaa ongelmiin kansainvälistämisessä. Esimerkiksi X-merkkiä on Japanissa käytetty normaalisti tarkoittamaan, että jotakin ei haluta. Siten kyseisen merkin käyttö valintalaatikossa asetuksen käyttöönoton kuvaamiseen on japanilaisille harhaanjohtava. Toinen esimerkki liittyy kursivoinnin käyttöön tekstin korostamiseen graafisessa käyttöliittymässä. Kursivoinnin käyttö ei ole soveliasta silloin, kun teksti on kanjia (kiinalaisperäinen merkistö japaninkielessä). Sen sijaan joihinkin Japanissa myytyihin tekstinkäsittelyohjelmiin on lisätty muotoilupiirre, jossa tekstien korostukseen käytetään varjostettua laatikkoa.

Kynäpohjaisissa tietokoneissa ja joissakin virtuaalitodellisuuden järjestelmissä käytetyt eleet saattavat Nielsenin (1993, 241) mielestä tarvita tarkistusta kansainvälistä käytettävyyttä varten. Esimerkiksi oikolukijoiden merkkejä käytetään usein editointiin kynäliittymissä, vaikka niillä on juurensa typografisissa perinteissä, jotka saattavat vaihdella eri maissa.

Merkistöön liittyvät eroavaisuudet ovat hyvin relevantteja kansainvälistämisessä. Monissa maissa on merkkijoukkoja, jotka ovat laajempia kuin englanninkielessä ja alkuperäisessä ASCII-merkkijoukossa käytetyt aakkoset A—Z väliltä. Monissa maissa, kuten erityisesti Aasian maissa, on hyvin laajat merkistöt, joita ei voi koodata kahdeksan bitin sallimaan 256 arvoon (Lunde 1993, Nielsenin 1993, 248 mukaan). Yleensä vähintään 16

bittiä käytetään kaikkien merkkien esittämiseen ja sovelluksen täytyy pystyä käsittelemään tämä (Nielsen 1993, 248). Useissa Euroopan maissa, kuten Suomessa, merkistöt ovat laajempia kuin alkuperäinen seitsemän bitin ACSII-koodi, mutta silti ne sopivat kahdeksan bittisen sanan sisään. Nielsenin (1993, 248) perusohje on mukautua paikalliseen merkistöön. Lisäohje on kohdella paikallisia lisämerkkejä kuin "ensimmäisen luokan kansalaisia" ja sallia niiden tasa-arvoinen käyttö esimerkiksi muuttujien nimissä ja tiedostonimissä.

Nielsen (1993, 248) toteaa Beckeriin (1984) viitaten, että merkistöjen käsittely tulee vielä monimutkaisemmaksi, mikäli on tarpeen esittää useita kieliä samassa dokumentissa tai samassa tietokoneessa. Useiden kielien esittäminen samassa dokumentissa on yleinen tarve muun muassa useille monikansallisten yhtiöiden työntekijöille. Kaksi lähestymistapaa todellisiin kansainvälisiin merkistöihin pyrittäessä ovat Nielsenin (1993, 248) mukaan Unicoden 16 bitin standardi ja ISO 10646 48 bitin standardi.

Oikeanlainen eri merkistöjen kansainvälistetty käsittely vaatii, että merkkikoodien numeerisia arvoja ei käytetä (Nielsen 1993, 248—249). Esimerkiksi merkkien vaihtaminen pienistä isoiksi kirjaimiksi ei saa tapahtua lisäämällä 26 merkin numeeriseen arvoon, koska A—Z kirjaimien ulkopuolisien kirjaimien isoja ja pieniä kirjaimia ei välttämättä ole koodattu 26 merkin päähän toisistaan.

Lajittelun tulee ottaa erikoismerkit huomioon (Nielsen 1993, 249). Joissakin maissa ö lajitellaan kuten o, kun taas toisissa maissa, kuten esimerkiksi Suomessa, se lajitellaan aakkosten loppuun kuuluvana erillisenä kirjaimena. Joissakin kielissä on myös hienoisia eroja siinä, ovatko lajiteltavat nimiä vai normaaleja sanoja (Nielsen 1993, 249). Esimerkiksi kanjin nimet lajitellaan ääntämisen mukaan. Jotta tämän pystyy tekemään, täytyy arvioida, kuinka merkki äännetään kussakin nimessä. Nämä seikat viittavat siihen, että kullekin maalle saatetaan tarvita erilaiset lajittelufunktiot (Nielsen 1993, 249). Valitettavasti jopa saman maan sisällä eri järjestelmien valmistajien käyttämissä lajittelujärjestyksissä (collation table) on pieniä eroja ja ne eivät välttämättä aina vastaa tarkasti kansallisia standardeja. Siten sanojen joukko saattaa olla aakkostettu erilailla riippuen siitä, millä alustalla sovellusta suoritetaan, vaikka sovellus noudattaa soveltuvia

paikallistamissääntöjä ja lajittelujärjestyksiä. Nielsen (1993, 249) suosittaa, että kunkin maan paikallistamissääntöjen noudattaminen on todennäköisesti paras ratkaisu sovelluksissa, jotka ovat tarkoitettu käytettäviksi useiden alustojen päällä.

Eräs ongelma kansainvälistämisessä on esitettävien merkkijonojen yhdistely (Davis 1998). Eri kielissä lauseen osat ovat erilaisessa järjestyksessä. Jos johonkin esitettävään lauseeseen yhdistetään merkkijono ja muuttuja, eikä muuttujan paikkaa voi vaihtaa lauseen sisällä, ei lauseen paikallistaminen onnistu. Siten myös lauseen osien sisäistä järjestystä tulisi pystyä muuttamaan sovellusta paikallistettaessa.

Merkkien kirjoitussuunta vaihtelee eri kielissä ja jopa samankin kielen sisällä. Lähi-idän kielistä muun muassa arabia ja hebrea kirjoitetaan oikealta vasemmalle, mutta myös vasemmalta oikealle suuntautuvan tekstin lisääminen muun tekstin joukkoon on Davisin (1998) mukaan sallittua. Vasemmalta oikealle suuntautuvat näissä kielissä esimerkiksi numerot ja englanninkielinen teksti. Tätä piirrettä kutsutaan kaksisuuntaisuudeksi (bidirectional, BIDI) (Davis 1998). Lisäksi arabialaiset merkit saattavat muuttaa täysin muotoaan kontekstin mukaan. Joissakin kielissä merkit voidaan kirjoittaa myös pystysuoraan, jolloin vaakatasossa edetään yleensä oikealta vasemmalle (Davis 1998). Tällainen on mahdollista kaukoidän kielissä kuten esimerkiksi kiinassa. Ongelmana on tällöin Davisin (1998) mukaan myös joidenkin merkkien mahdollinen kiertyminen tai niiden muodon muuttuminen pystysuorassa kontekstissa. Nämä piirteet vaatisivat hyvin erityistä käsittelyä tekstin asettelussa.

Merkkien kirjoitussuunnan vaihtelut täytyisi huomioida myös käyttöliittymässä yleensä. Esimerkiksi oikealta vasemmalle kirjoitettavissa kielissä myös kohteiden yleinen suunta tulee Davisin (1998) mielestä olla oikealta vasemmalle. Muita huomioitavia asioita ovat muun muassa sarkainvälit tekstissä ja valintalaatikon ilmestymissuunta valintalaatikkokomponentissa. Tekstin tulisi myös olla oikealta tulostuvaa vasemmalta tulostumisen sijaan (Davis 1998). Nämä seikat täytyy huomioida komponenttikohtaisesti.

Sanojen erottaminen toisistaan ei ole kaikissa kielissä samanlaista. Esimerkiksi thain kieli vaatii erityistä sanankatkaisun hallintaa, koska sanojen erottamiseen ei käytetä tyhjiä välejä (Davis 1998).

Joidenkin kielten esittämisessä tarvittaisiin selitystekstejä niiden oikeaa ääntämistä varten. Esimerkiksi kirjoitetun kanjin lukija ei yleensä tiedä tietyn symbolin ääntämistä (Davis 1998). Siten pieniä ääntämissymboleja sijoitetaan usein kirjoitettujen symbolien yläpuolelle.

Numeroille käytetään erilaista merkintätapaa eri maissa (Nielsen 1993, 250). Todennäköisenä pääerona Nielsen (1993, 250) pitää desimaalierottimena käytettyä merkkiä, joka on pilkku tai piste. Merkkiä, jota ei käytetä desimaalierottimena, käytetään tyypillisesti tuhansien erottimena, mutta myös tyhjää merkkiä käytetään joskus tuhansien erottimena (Nielsen 1993, 250). Jotkut maat käyttävät myös erityisiä symboleja valuutan merkitsemiseen, kuten esimerkiksi $ ja £. Toiset maat käyttävät puolestaan lyhenteitä, joista esimerkkinä ovat mk ja kr. Nämä symbolit ja lyhenteet tulevat joko ennen numeroita tai numeroiden jälkeen. Esimerkiksi yksitoista tuhatta valuuttayksikköä voidaan kirjoittaa Englannissa £11,000.00 ja Tanskassa 11.000,00 kr. Lisäksi muutaman maan valuutoissa käytetään Nielsenin (1993, 250) mukaan kolmea desimaalia ja joillakin mailla on vain täydet valuuttayksiköt käytössä ilman desimaaleja.

Käyttöliittymien tulee hallita eri mittajärjestelmiä (Nielsen 1993, 250). Näistä tärkeimmät ovat SI-järjestelmä (Système International) ja amerikkalainen järjestelmä, johon kuuluvat muun muassa tuuma, jalka, maili ja Fahrenheit.

Kellonaika kirjoitetaan joskus 24 tunnin merkitsemistavalla, kuten esimerkiksi 23:00, ja toisinaan 12 tunnin merkitsemistavalla, josta esimerkkinä 12:00 PM.

Päivämäärien kirjoittamisessa on myös käytössä erilaisia merkitsemistapoja, joita on esitetty seuraavassa (Nielsen 1993, 250):

- D/M/Y, josta esimerkkinä on 3/1/00 (3. tammikuuta 2000).
- D/M-Y, josta esimerkkinä on 3/1-00.

- M/D/Y, josta esimerkkinä on 1/3/00 tai 01/03/00.
- Y.M.D, josta esimerkkinä on 2000.01.03.
- Y-M-D, josta esimerkkinä on 2000-01-03.

Näistä Y-M-D on ISO:n standardoima merkintätapa. Edellisestä luettelosta puuttuu muun muassa Suomessa, Norjassa ja Islannissa käytetty merkitsemistapa D.M.Y, esimerkiksi 03.01.2000. Lisäksi ainakin Tanskassa käytetään merkitsemistapaa D-M-Y, joka esitetään siis 03-01-2000.

Nämä merkintätavat voivat olla harhaanjohtavia. Y voi tarkoittaa koko vuosilukua tai vuoden kahta viimeistä numeroa. Kansainvälisissä yhteyksissä olisi tärkeää käyttää aina koko vuosilukua, koska tällöin voitaisiin välttää väärinkäsityksiä ja samalla vältyttäisiin ongelmilta vuosisatojen ja vuosituhansien vaihtuessa. Kansainvälisissä yhteyksissä myös kuukauden kirjoittaminen kirjaimilla on suositeltavampaa kuin antaa se numerona. Yksittäisen maan sisällä numeerisia esitysmuotoja saatetaan suosia niiden tiiviyden ja aikajaksojen helpon arvioinnin vuoksi. Sovelluksen pitäisikin siis sallia paikalliset ajan esitysmuodot sekä kuukausien ja viikonpäivien paikallisten nimien käyttö (Nielsen 1993, 250—251).

Joissakin maissa on vaihtoehtoiset kalenterit gregoriaaniselle kalenterille, joten tällaisessa tapauksessa sovelluksen kansainvälistämisessä saatettaisiin joutua rakentamaan tuki erilaisille kalentereille. Esimerkiksi arabimaissa on käytössä kalenteri, jonka ajanlasku alkaa islamin uskon perustajan Muhammedin ajalta noin puoli vuosituhatta myöhemmin kuin kristittyjen ajanlasku.

3.6 Ratkaisuja graafisen käyttöliittymän kansainvälistämiseen

Yksi päätavoista parantaa kansainvälistämistä käytännössä on Nielsenin (1993, 251—252) mukaan erottaa käyttöliittymä ja järjestelmän toiminnallisuus toteutuksessa. Perinteisillä tavoilla kirjoitetussa sovelluksessa käyttöliittymämääritys on vahvasti kytköksis-

sä muuhun koodiin normaalisti siten, että huomautusten ja virheiden merkkijonot ovat proseduurikutsujen osana niissä osissa sovellusta, jotka tarvitsevat niitä. Sellaisen ohjelman siirtäminen on melko vaikeaa ja vaatii pääsyn lähdekoodiin sekä sen tarkkaa tutkimista.

Uudemmissa järjestelmissä on alettu tallentaa käyttöliittymämääritys erillisinä resursseina, jotka yhdistetään muuhun koodiin sovelluksen ajonaikana (Nielsen 1993, 252). Esimerkiksi kaikkien käyttäjän näkemien tekstimerkkijonojen tallentaminen yhteen paikkaan helpottaa niiden kääntämistä kieleltä toiselle.

Myös ikkunoiden ja dialogien sijoittelumääritykset voitaisiin Nielsenin (1993, 252) mielestä tallentaa resurssitietokantoihin. Ilmiasumääritys voisi sisältää informaatiota ikkunan koosta ja kunkin elementin suhteellisesta sijainnista, sisältäen tekstimerkkijonot, käyttäjän syöttökentät ja kuvakkeet. Kuvakkeet voitaisiin tallentaa erikseen siten, että niitä voitaisiin editoida vastaamaan paikallisia graafisia tapoja. Nielsen (1993, 252) suosittelee käyttämään näytön editointiin työkalua, jonka avulla siirtäjä voisi järjestää uudelleen dialogin elementit sopimaan paikalliseen kieleen ja muuttaa tekstikenttien kokoja lisätilan tekemiseksi lisäsanoille tai pidemmille sanoille. Jos koon muutokset eivät ole mahdollisia, yleinen suositus alkuperäisten tekstikenttien suunnitteluun on jättää siirrettäviin versioihin noin 30 % tilaa ylimääräisille merkeille, kun alkuperäinen kieli on englanti (Nielsen 1993, 252).

Joissakin systeemitason järjestelmissä monet paikalliset tavat, kuten esimerkiksi merkkien lajittelujärjestys ja tarkoituksenmukainen tapa kirjoittaa päivämääriä sekä kuukausien nimiä, tallennetaan keskitettyyn resurssikantaan, jota voidaan käyttää mistä tahansa sovelluksesta (Nielsen 1993, 253). Jos yksittäiset sovellukset yhdistävät tällaiset systeemitason resurssit omiinsa, ne voivat Nielsenin (1993, 253) mukaan saavuttaa melko korkean tason kansainvälistämisen ilman minkäänlaista lisätyötä. Tällöin käyttöliittymä ottaa paikalliset merkit huomioon käyttämällä paikalliseen systeemiin tallennettuja resursseja. Käytettävä paikallisuus ja monia erilaisia paikallisuuksiin liittyviä, valmiiksi määriteltyjä erityispiirteitä voidaan muuttaa esimerkiksi Windows 95:ssa.

Hyvin laajoja merkistöjä sisältäviä Kaukoidän kieliä, kuten kiinaa, japania ja koreaa, varten ei ole mielekästä tehdä näppäimistöä, jossa olisi näppäimet kaikille merkeille. Sen sijaan käytetään ohjelmallisia syöttölaitteita, joita kutsutaan syöttömetodikoneiksi (Input Method Engine, IME) tai joskus edustaprosessoriksi (Front-End Processor, FEP) (Davis 1998). Näiden avulla käyttäjä voi kirjoittaa merkit käyttäen pientä joukkoa äänteellisiä tai muita merkkejä. Muutamat kohdealustat tarjoavat syöttötukea laajoille merkistöille.

Mahdollista syöttötukea on kolmentyyppistä: "pois kohdalta" (off-the-spot), "kohdan päällä" (over-the-spot) ja "kohdalla" (on-the-spot). Näistä kukin tarjoaa eritasoiset mahdollisuudet ja vaatii eriasteisia muutoksia varsinaiseen sovellusohjelmaan. (Davis 1998.)

"Pois kohdalta" -lähestymistavassa käyttäjän kirjoittaessa merkin sovellusohjelmassa erillinen ikkuna ilmestyy tavallisesti näytön alaosaan. Tässä ikkunassa käyttäjä on vuorovaikutuksessa syöttömetodikoneen kanssa. Kun käyttäjä lopettaa, näppäimistötapahtumien sarja syötetään ohjelmalle. Tämän lähestymistavan arvo käyttäjälle on vähäinen, koska käyttäjä joutuu syöttämään merkit erillisessä ikkunassa näytön toisessa osiossa. Varsinaiseen sovellusohjelmaan ei tarvittaisi juurikaan muutoksia. (Davis 1998.)

"Kohdan päällä" -syöttötuessa käyttäjän kirjoittaessa merkin sovellusohjelmassa erillinen ikkuna ilmestyy suoraan sen paikan yläpuolelle, jossa käyttäjä oli kirjoittamassa. Muutoin tämä syöttötuki toimii vastaavasti kuin edellinen. Tällä syöttötuella on enemmän lisäarvoa käyttäjälle kuin edellisellä lähestymistavalla. Tekstillä on usein sama kirjasintyyppi ja koko. Lisäksi tässä käyttäjällä on parempi tuntuma siitä, että hän on kirjoittamassa suoraan dokumenttiin. Tätä lähestymistapaa käytettäessä sovellusohjelmassa täytyisi olla enemmän tukea edelliseen lähestymistapaan verrattuna. (Davis 1998.)

"Kohdalla" -lähestymistavassa käyttäjän kirjoittaessa merkin sovellusohjelmassa merkki menee suoraan dokumenttiin. Erityinen korostus näkyy tekstissä ja muutokset heijastuvat välittömästi pitäen sisällään sanojen vierityksen. Tämän lähestymistavan arvo käyttäjälle olisi edellisiä lähestymistapoja suurempi ja se vaatisi sovellusohjelmaan myös kaikkein monimutkaisimpia mekanismeja. (Davis 1998.)

Systeemitason resursseihin ja sovelluskohtaisten resurssien siirtämiseen pohjautuvan kansainvälistämisen on Nielsenin (1993, 253) mielestä soveliasta tukea yksittäisen kulttuurikohtaisen liittymän käyttöä. Oletetaan, että käyttäjä on esimerkiksi Espanjassa asuva espanjalainen, joka kommunikoi vain muiden espanjalaisten kanssa Espanjassa. Hänen ongelmiensa pitäisi ratketa, jos kaikkien hänen käyttämiensä sovelluksien kehittäjät olisivat tehneet kunnon työtä järjestelmiensä paikallistamisessa Espanjaan. On kuitenkin käyttäjiä, jotka tarvitsevat monipaikallistettuja (multilocale) liittymiä (Nielsen 1993, 253). Esimerkkejä heistä voisivat olla henkilöt, jotka muuttavat vieraaseen maahan tai vierailevat ulkomailla. Myös henkilöillä, jotka kommunikoivat tai vaihtavat dataa toisen maan kansalaisten kanssa, voisi olla tarvetta monipaikallistettuihin liittymiin.

Nielsenin (1993, 253) mukaan ideaalitilanteessa kaikki käytettävät liittymät ja tiedostot pitäisi kytkeä paikallisuuteen, joka tunnistaa oikean senhetkisen käyttäjän kanssa tarvittavan paikallistamisen. Jos uusi käyttäjä aloittaa sovelluksen käyttämisen tai dataa siirretään toiseen maahan, pitäisi olla mahdollista valita uusi paikallisuus ja muuttaa liittymä sekä tiedon tulkinta vastaavasti. Oletetaan esimerkiksi, että myyjällä on olemassa tiettyjen tuotteiden hintatietokanta Englannissa. Tuotteen hinta saattaisi olla £1,499.90 ja se pitäisi esittää tällaisena, kun paikallisuuden asetus on "Iso-Britannia". Jos hinnat sisältävä tiedosto lähetettäisiin sähköpostina asiakkaalle Saksaan, paikallisuus vaihtuisi ja hinta pitäisi näyttää seuraavasti: £1.499,90. Vaikka järjestelmän muuttujan "paikallinen valuuttasymboli" -arvo muuttuisi £:sta DM:ksi paikallisuuden vaihtuessa Iso-Britanniasta Saksaksi, järjestelmän ei pitäisi vaarantaa datan yhtenäisyyttä vaihtamalla mittayksikköä jo annetulle datalle. Kuitenkin järjestelmän pitäisi huolehtia desimaalipisteen ja tuhansien erottimen vaihtamisesta. Samoin muun muassa tietokantakomennot ja virheilmoitukset pitäisi vaihtaa englannista saksaksi paikallisuuden muuttuessa. Jos saksalaisella käyttäjällä on suomalainen vierailija, vierailijalle pitäisi olla mahdollista väliaikaisesti asettaa paikallisuudeksi "Suomi" ja käyttää järjestelmää suomenkielellä.

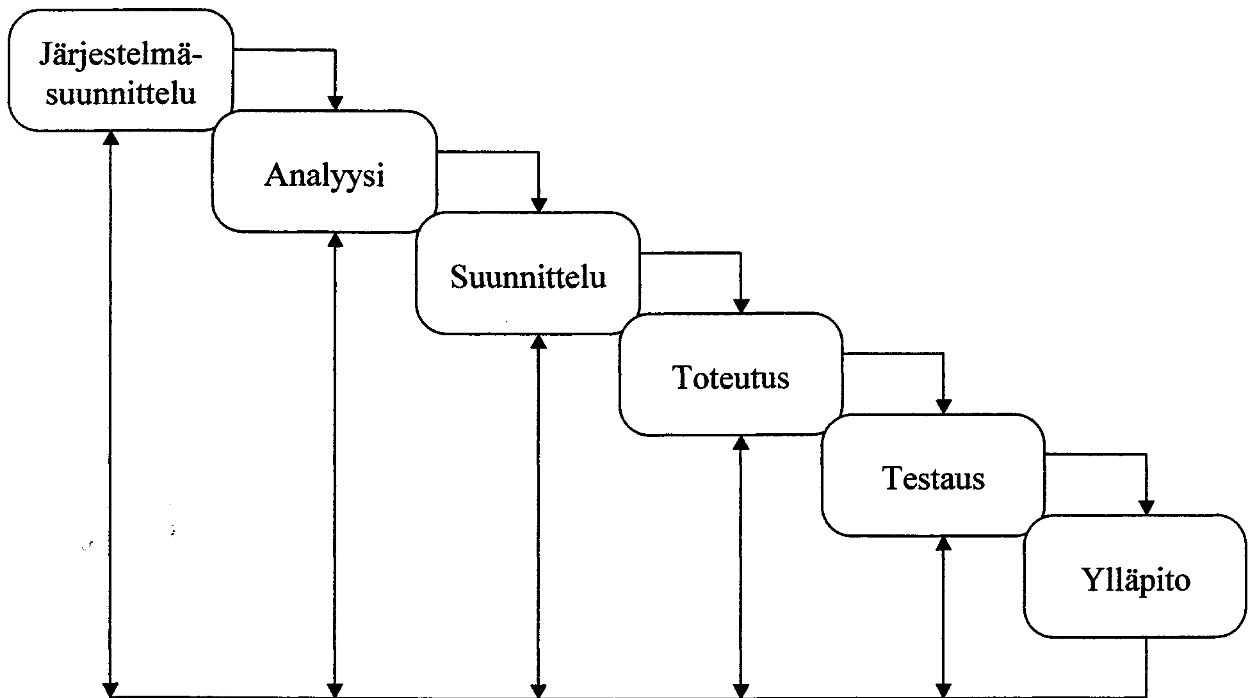
4 SIIRRETTÄVYYS OHJELMISTOPROSESSISSA

Luvun tarkoituksena on suhteuttaa siirrettävyys yleiseen ohjelmistoprosessiin. Aluksi valitaan yleisin ohjelmistoprosessia kuvaava prosessimalli. Tämän jälkeen valittua prosessimallia tarkennetaan, jotta se havainnollistaisi paremmin siirrettävyyskysymyksiä. Prosessimallista muodostetaan viitekehys jatkotarkastelujen pohjaksi. Viitekehyksen avulla havainnollistetaan siirrettävyysongelmien laajuutta ja suhteutetaan edellä esitettyjä siirrettävyystekniikoita ohjelmistoprosessiin.

4.1 Ohjelmistoprosessi

Tavoitteena on valita yleisin koko ohjelmistoprosessia kuvaava prosessimalli, jotta siirrettävyys voidaan suhteuttaa yleiseen ohjelmistoprosessiin. Klassinen elinkaarimalli (classic life cycle) soveltuu tähän tarkoitukseen, koska muun muassa Pressman (1994, 25) pitää sitä vanhimpana ja laajimmin ohjelmistokehitykseen käytettynä mallina. Lisäksi klassinen elinkaarimalli kattaa koko ohjelmistoprosessin. Klassinen elinkaarimalli esitetään Pressmanin (1994) näkemyksen pohjalta.

Klassinen elinkaarimalli, jota on kutsuttu myös vesiputousmalliksi (waterfall model), tarjoaa Pressmanin (1994, 24—25) mukaan systemaattisen peräkkäisen lähestymistavan ohjelmistokehitykseen. Klassisen elinkaarimallin vaiheet on kuvattu kuviossa 7.



KUVIO 7. Klassinen elinkaarimalli (vrt. Pressman 1994, 24).

Ohjelmistoprosessi alkaa järjestelmäsuunnittelulla. Järjestelmätason suunnittelua tarvitaan, koska sovellusohjelma on aina osa laajempaa järjestelmää tai toimintaa. Tässä vaiheessa määritellään vaatimukset kaikille järjestelmän osille ja kohdennetaan näistä tarkoituksenmukainen alijoukko sovellusohjelmalle. Sovellusohjelman vaatimusanalyysi puolestaan keskittyy itse sovellusohjelman vaatimusten keräämiseen ja määrittelyyn. Suunnittelu on prosessi, joka keskittyy neljään sovellusohjelman kohteeseen: tietorakenteisiin, ohjelman arkkitehtuuriin, proseduraalisiin yksityiskohtiin ja rajapintoihin. Toteutusvaiheessa suunnitelmat muunnetaan koneen ymmärtämään muotoon. Testausprosessi keskittyy ohjelman sisäiseen logiikkaan varmistaen, että kaikki koodi testataan. Toinen testausprosessin keskeinen tavoite liittyy ohjelman ulkoiseen toiminnallisuuteen, jotta virheet saataisiin paljastettua testauksin ja varmistettua, että määritellyt syötteet tuottavat vaatimukset täyttävät tulokset. Sovellusohjelman ylläpito kohdistuu asiakastoimituksen jälkeisten muutosten tekemiseen. Ylläpito hyödyntää edellisiä vaiheita, jotta muutokset olemassa olevaan sovellukseen saadaan tehdyksi. (Pressman 1994, 24—25.)

Klassista elinkaarimallia vastaan on myös esitetty kritiikkiä (Pressman 1994, 25—26). Malli perustuu vaiheiden peräkkäisyyteen, mutta todelliset projektit etenevät harvoin peräkkäisesti. Asiakkaan on yleensä vaikea määrittää heti aluksi vaatimuksia, vaikka klassinen elinkaarimalli sitä edellyttääkin. Asiakkaalla täytyy olla myös kärsivällisyyttä, koska sovellusohjelman toimiva versio saadaan vasta projektin myöhäisessä vaiheessa. Kritiikistä huolimatta klassisella elinkaarimallilla on Pressmanin (1994, 26) mukaan tärkeä paikka ohjelmistokehitystyössä. Se tarjoaa mallin, johon voidaan sijoittaa teknii- kat analyysiä, suunnittelua, koodausta, testausta ja ylläpitoa varten.

Edellä esitetty kritiikki näyttää kohdistuvan prosessimallin toimivuuteen ohjelmistokehityksessä, mutta ei sen sijaan siihen, löytyykö todellisesta ohjelmistoprosessista kaikki klassisen elinkaarimallin vaiheet. Tästä voidaan todeta, että järjestelmätason suunnittelua ei tarvita, mikäli kyse on aivan pienestä yksittäisestä sovelluksesta, joka ei ole osa laajempaa toimintaa tai järjestelmää, ja sovellus tekee vain sille kuuluvan tehtävän. Esimerkki tällaisesta sovelluksesta voisi olla vaikkapa pieni pelisovellus. Kysymys klassisen elinkaarimallin vaiheiden kattavuudesta voidaan esittää myös toisinpäin: kattaako klassinen elinkaarimalli kaikki todellisen ohjelmistoprosessin vaiheet? Klassinen elinkaarimalli on esitetty niin yleisellä tasolla, että joitakin ohjelmistoprosessin osavaiheita on jätetty esityksen ulkopuolelle. Tällainen on esimerkiksi ohjelmiston käyttöönotto, joka voisi olla omana vaiheenaan testauksen ja ylläpidon välissä. Kuitenkin yleisellä tasolla klassinen elinkaarimalli sisältää ohjelmistoprosessin vaiheet sekä kattaa koko ohjelmistoprosessin.

On kuitenkin huomattava, että klassinen elinkaarimalli esittää ohjelmistoprosessin perusjäsennyksen, mutta usein käytännössä ohjelmistokehitys etenee erilaisena prosessina. Eri vaiheissa saattaa tapahtua iteraatioita, jolloin ohjelmistokehityksessä tehdään osittain jo tarkemman abstraktiotason toimintaa. Taustamotiivina saattaa olla esimerkiksi ohjelmistokehityksen riskien minimointi tai ohjelmiston käytettävyyden parantaminen. Tästä esimerkkinä analyysivaiheessa saatetaan tuottaa prototyyppi graafisesta käyttöliittymästä sovelluksen käytettävyyden parantamiseksi, jolloin tehdään jo suunnittelua ja toteutusta.

Edellisen tarkastelun perusteella klassisen elinkaarimallin roolina on olla yleisen tason ohjelmistokehityksen prosessimalli, koko ohjelmistoprosessin abstrakti kuvaus. Niinpä sitä tullaan tässä työssä pitämään pohjana, jonka päälle rakennetaan siirrettävyyden tarkastelussa tarvittava viitekehys.

Tarkasteltaessa klassisen elinkaarimallin vaiheita voidaan havaita kunkin vaiheen sisältävän joukon aktiviteetteja. Esimerkiksi toteutusvaiheessa kirjoitetaan ohjelman suunnitelmista ohjelmakoodia. Jotta kirjoitetusta ohjelmakoodista saataisiin varsinainen ajettava ohjelma, täytyy esimerkiksi C-kielinen ohjelmakoodi lisäksi kääntää ja linkittää. Ohjelman ajaminen on myös oma aktiviteettinsa, jota tarvitaan esimerkiksi ohjelman toimivuuden testaamisessa. Nämä tarkemmalla tasolla olevat aktiviteetit on relevanttia nostaa esiin siirrettävyyden tarkastelua varten, kuten jo aiemmin luvussa 2.1 ilmeni. Täten jaamme toteutuksen pienempiin osiin, joita ovat siis:

- koodaaminen,
- kääntäminen,
- linkittäminen sekä
- ajaminen.

Aktiviteeteista ainakin ajaminen kuuluisi myös testaukseen, mutta koska aktiviteetit eivät ole testausvaiheessa keskeisessä roolissa, testausvaihetta ei ole tarkennettu vastaavasti. Toiminnot kuuluisivat myös ylläpitovaiheeseen, mutta ylläpitovaihe jätetään tämän tarkastelun ulkopuolelle, koska ylläpito on aina edeltävien vaiheiden suorittamista pienoiskoossa. Huomionarvoinen seikka on lisäksi, että lisäaktiviteeteista ainakin ajaminen liittyy myös ohjelmiston käyttöönottoon. Kuitenkaan itse käyttöönottoakaan ei ole erillisenä vaiheena mukana ja tämän työn kannalta relevantin informaation havainnollistamisessa sen lisäämistä ja tarkentamista ei tarvita.

Seuraavassa kohdassa on tavoitteena tarkastella siirrettävyysongelmien ja ohjelmistoprosessin suhdetta yleisellä tasolla. Tarkastelun viitekehyksenä käytetään edellä tarkennettua klassista elinkaarimallia.

4.2 Siirrettävyysongelmat ja ohjelmistoprosessi

Siirrettävyysongelmat ovat erilaajuisia. Jotkut ongelmat edellyttävät enemmän siirtämistyötä kuin toiset. Mitä enemmän siirtämistyötä joudutaan tekemään, sitä enemmän siitä aiheutuu kustannuksia. Miten sitten kustannuksia voitaisiin pienentää? Ensinnäkin siirrettävyysongelmat täytyisi tiedostaa riittävän aikaisessa ohjelmistoprosessin vaiheessa. Siirrettävyysongelmien laajuus ohjelmistoprosessissa tulisi ymmärtää, jotta ongelmiin ja niiden mahdolliseen eliminoimiseen osattaisiin paremmin varautua. Yksi keino ongelmien laajuuden selvittämiseen ja havainnollistamiseen on suhteuttaa ne yleiseen ohjelmistoprosessiin. Taulukossa 2 siirrettävyysongelmien laajuutta on havainnollistettu esimerkkien avulla. Taulukosta voidaan havaita, kuinka eritasoiset siirrettävyysongelmat liittyvät ohjelmistoprosessin vaiheisiin. Siirrettävyysongelman sijoittaminen tiettyyn ohjelmistoprosessin vaiheeseen osoittaa, että siirrettävyysongelma aiheuttaa siirtämistyötä kyseisessä ohjelmistoprosessin vaiheessa.

TAULUKKO 2. Esimerkkejä eritasoisten siirrettävyysongelmien liittymisestä ohjelmistoprosessin vaiheisiin. Merkintä * tarkoittaa, että abstraktia konetta käytettäessä sovellusohjelmalle ei välttämättä suoriteta erillistä linkittämistoimenpidettä.

	ESIMERKKEJÄ ERITASOISISTA SIIRRETTÄVYYSONGELMISTA					
VAIHE	Sovellus täytyy ajaa uudessa ympäristössä. Ideaalitapaus: sovelluksen binäärikoodi voidaan ajaa sellaisenaan.	Sovellus täytyy ajaa uudessa ympäristössä. Ideaalitapaus: sovelluksen välikielinen koodi voidaan ajaa abstraktilla koneella.	Sovellus täytyy kääntää uuteen ympäristöön.	Sovelluksessa on käytetty laajennuksia, joita uuden ympäristön kääntäjä ei hyväksy.	Sovelluksessa on käytetty käyttöjärjestelmäkohtaista palvelua, kuten fork:ia.	Perinteinen graafinen käyttöliittymä muutetaan toisen ikkunointijärjestelmän ja käyttöliittymätyylin mukaiseksi.
Järjestelmäsuunnittelu						
Analyysi						x
Suunnittelu					x	x
Koodaaminen				x	x	x
Kääntäminen			x	x	x	x
Linkittäminen		x*	x	x	x	x
Ajaminen	x	x	x	x	x	x
Testaus	(x)	(x)	x	x	x	x

Siirrettävyysongelmana saattaa olla, että sovellus pitää ajaa uudessa ajoympäristössä. Ideaalitapauksessa sovelluksen binäärikoodi voitaisiin ajaa sellaisenaan uudessa ympäristössä ilman minkäänlaista ylimääräistä siirtämistyötä. Tällaisessä tapauksessa tarvittavia aktiviteetteja olisivat vain sovelluksen ajaminen ja mahdollinen testaus sen varmistamiseksi, että sovellus toimii uudessa ympäristössä. Kuitenkin tämä on mahdollista vain joissakin hyvin lähellä toisiaan olevissa ympäristöissä. Toinen ideaalitapaus siirrettävyyden kannalta olisi, että sovelluksen välikielinen koodi voitaisiin ajaa sellaisenaan uuteen ympäristöön toteutetulla abstraktilla koneella. Tällöin itse sovelluksen siirtämiseen ei parhaassa tapauksessa tarvittaisi siirtämistyötä. Aktiviteetit ovat samat kuin edellisessä tapauksessa, jos abstraktia konetta käytettäessä sovellusohjelmalle ei suoriteta erillistä linkittämistoimenpidettä.

Kun sovellusta ei voida ajaa sellaisenaan uudessa ympäristössä, joudutaan se vähintään kääntämään ja linkittämään uudessa ympäristössä edellyttäen, että kääntäjä ja linkittäjä on toteutettu kohdeympäristössä. Tällöin sovellus tulee myös testata uudessa ympäristössä. Mikäli sovelluksessa on käytetty joitakin laajennuksia, joita uuden ympäristön kääntäjä ei hyväksy, täytyy lisäksi tehdä uudelleenkoodausta.

Siirrettävyysongelmana voi olla sovelluksen siirtäminen toisen käyttöjärjestelmän sisältävään ympäristöön. Jos siirrettävässä sovelluksessa on käytetty suoraan jotakin vain alkuperäisessä käyttöjärjestelmässä ollutta palvelua, kuten esimerkiksi Unixin fork:ia, aiheutuu siitä lisää työtä. Tällöin voidaan joutua suunnittelemaan toiminnon korvaaminen esimerkiksi uuden käyttöjärjestelmän tarjoamilla palveluilla. Tämän lisäksi sovellus täytyy koodata, kääntää, linkittää, ajaa ja testata.

Mikäli siirrettävyysongelmana on muuttaa perinteisen graafisen käyttöliittymän sisältävä sovellus toisen ikkunointijärjestelmän ja sen sisältämän käyttöliittymätyylin mukaiseksi, on kyseessä edellisiä laajemmalle ulottuva ongelma. Sovelluksen graafinen käyttöliittymä määritellään usein jo analyysivaiheessa. Tuolloin saatetaan laatia spesifikaatiot, jonka osaksi tulee kuvallinen visualisointi käyttöliittymän näytöistä. Määrityksissä tulee huomioida myös, että käyttöliittymän visualisointi ja toiminnallisuus ovat kohteena olevan ikkunointijärjestelmän tyylioppaan mukaisia. Tällaista menettelytapaa käytetään

esimerkiksi Ari Jaaksin OMT-menetelmän (Object Modeling Technique) pohjalta kehittämässä OMT++-menetelmässä. Tällöin näitä analyysivaiheen määrityksiä joudutaan muuttamaan. Lisäksi tarvitaan suunnittelua, miten käyttöliittymä toteutetaan uuden ikkunointijärjestelmän tarjoamilla palveluilla. Näiden lisäksi tarvitaan koodausta, kääntämistä, linkittämistä, ajamista ja testausta.

Edellisen tarkastelun pohjalta voidaan todeta, että mitä aikaisemmalle ohjelmistoprosessin tasolle siirrettävyysongelman laajuus ulottuu, sitä enemmän aktiviteetteja tarvitaan ohjelman siirtämiseksi. Edelleen mitä enemmän aktiviteetteja tarvitaan, sitä enemmän siirtämiskustannukset kasvavat.

Koska siirrettävyysongelmat liittyvät jo melko aikaisiin ohjelmistoprosessin vaiheisiin, tulee sovelluksen siirrettävyyteen varautua ajoissa. Mitä aiemmin ohjelmistoprosessissa varaudutaan siirrettävyyteen, sitä paremmin on mahdollista huomioida siirrettävyys erilaisissa valinnoissa, päätöksissä ja ratkaisuissa. Tämä lisää myös siirtämiskustannusten säästömahdollisuuksia.

4.3 Siirrettävyystekniikat ohjelmistoprosessissa

Myös siirrettävyystekniikat kohdistuvat ohjelmistoprosessin tiettyihin vaiheisiin. Niiden kattavuuden hahmottamisesta on hyötyä siirrettävyyteen varautumisessa. Taulukossa 3 on esitetty yleisimpien siirrettävyystekniikoiden kohdistuminen ohjelmistoprosessissa. Kohdistuvuudella tarkoitetaan sitä, että siirrettävyystekniikka tukee juuri kyseisessä vaiheessa suoritettavaa siirtämistyötä tai siirtämiseen varautumista.

TAULUKKO 3. Siirrettävyystekniikoiden kohdistuminen ohjelmistoprosessissa. Merkintä * tarkoittaa, että abstraktia konetta käytettäessä sovellusohjelmalle ei välttämättä suoriteta erillistä linkittämistoimenpidettä.

	YLEISET SIIRRETTÄVYYSTEKNIIKAT					
VAIHE	Emulointi	Abstrakti kone	Esiprosessori	Siirrettävyysrajapinta	Standardointi	Kansainvälistäminen
Järjestelmäsuunnittelu						
Analyysi					x	x
Suunnittelu				x	x	x
Koodaaminen				x	x	x
Kääntäminen			x		x	
Linkittäminen		x*			x	
Ajaminen	x	x			x	
Testaus						

Emuloinnilla pyritään jo olemassa olevan sovelluksen siirrettävyyteen, kuten luvussa 2.3 esitettiin. Siinä tarjotaan uusi ympäristö, jossa sovellusta voidaan ajaa. Siten tämä siirrettävyystekniikka kohdistuu lähinnä sovelluksen ajamiseen.

Abstrakti kone muuttaa välikieliset käskyt kohdekoneen käskyjoukoksi. Sovellusta voidaan ajaa sellaisessa ympäristössä, johon sovelluksen käyttämällä välikielellä toimiva abstrakti kone on toteutettu. Abstraktia konetta käytettäessä ei aina tarvita erillistä linkittämisoperaatiota. Siten tämä siirrettävyystekniikka kattaa ajamisen ja joskus myös linkittämisen. Esiprosessorin käyttö kohdistuu selkeästi sovelluksen kääntämiseen.

Siirrettävyysrajapinnalla pyritään erottamaan sovelluksen ympäristöriippuvuudet rajapinnan taakse. Siten tämä tekniikka kohdistuu ohjelmistoprosessin suunnittelu- ja koodaamisvaiheisiin. Siirrettävyystekniikkana standardointi esiintyy laajasti ohjelmistoprosessissa, mikäli teollisuusstandardit otetaan myös huomioon. Toteutusvaihetta tukevat esimerkiksi ohjelmointikielistandardit, ja suunnitteluun vaikuttavat muun muassa käyttöjärjestelmästandardit. Lisäksi analyysivaiheeseen vaikuttavia standardeja ovat esimerkiksi ikkunointijärjestelmästandardit.

Kansainvälistämisen keinoin erotetaan tuotteesta kulttuurikohtaiset elementit, kuten luvussa 3.5 määriteltiin. Yksi keskeisistä keinoista on tallentaa sovelluksen käyttöliittymämääritys erillisissä resursseissa. Lisäksi tulee hyödyntää systeemitason tarjoamia re-

sursseja. Sovelluksen kansainvälistäminen toteutetaan ohjelmistoprosessin koodaamisvaiheessa. Tätä ennen tulee kuitenkin määritellä, mitä sovelluksesta paikallistetaan, ja suunnitella, mitä kansainvälistämistekniikoita käytetään. Määritykset voitaisiin tehdä jo käyttöliittymäspesifikaatiossa analyysivaiheessa, mikäli käytetään esimerkiksi OMT++-menetelmää.

Edellisissä tarkasteluissa on saanut vähän huomiota se, milloin käytettävien siirrettävyystekniikoiden valinta tulee ohjelmistoprosessin aikana tehdä. Edellisessä luvussa havainnollistettujen siirrettävyysongelmien laajuuden vuoksi nämä valinnat kannattaa tehdä jo järjestelmäsuunnittelu- tai analyysivaiheessa. Tällöin siirrettävyystekniikoiden avulla voidaan varautua sovelluksen siirrettävyyteen ajoissa ja siten säästää myös siirrettävyyskustannuksissa.

5 SIIRRETTÄVYYS JAVASSA

Tässä luvussa tarkastellaan sovellusohjelman ja graafisen käyttöliittymän siirrettävyyttä käytännön kontekstissa. Tarkastelun kohteeksi on valittu Java. Se on mielenkiintoinen ohjelmointikieli siirrettävyyden tarkastelun kannalta, koska se lupaa siirrettävyyttä ja sen käyttöliittymäosa on kehittynyt paljon aivan viime aikoina. Tämän työn puitteissa Javan siirrettävyyden tarkastelu keskittyy periaatetasolle, joten esimerkkejä ja ratkaisuja ei esitetä ohjelmakooditasolla.

Aluksi tässä luvussa esitellään Javan lyhyttä historiaa. Sen jälkeen kuvataan Javan tarjoamat siirrettävyyden perusmekanismit. Tämän jälkeen perehdytään Java-ohjelmoinnin siirrettävyysongelmiin ja niiden ratkaisukeinoihin. Sitten tarkastellaan graafista käyttöliittymää ja sen siirrettävyyttä Javassa. Tämän jälkeen luodaan katsaus siihen, miten kansainvälistäminen on Javassa toteutettu. Lopuksi tehdään yhteenveto Javan siirrettävyydestä.

5.1 Javan historia

Seuraava Javan historian tarkastelu perustuu pääasiassa Horstmannin ja Cornellin (1997, 14—16) esitykseen. Vuonna 1991 ryhmä Sunin insinöörejä alkoi suunnitella pientä ohjelmointikieltä, jota voitaisiin käyttää kulutuselektroniikkalaitteissa, kuten kaapelitelevision kytkentälaatikoissa. Koska näissä laitteissa ei ole paljon tehoa eikä muistia, kielen täytyi olla pieni ja tuottaa hyvin tiivistä koodia. Koska eri valmistajat saattoivat käyttää erilaisia prosessoreja, oli tärkeää, ettei kieli ollut sidottu mihinkään tiettyyn arkkitehtuuriin.

Nämä vaatimukset saivat Sunin insinöörit kaivamaan esiin mallin, jota UCSD Pascal -kielessä kaupallisesti kokeiltiin PC:n alkuaikoina ja jonka uranuurtaja ennen sitä oli Niklaus Wirth. Kyseessä oli abstrakti kone ja siirrettävä välikieli. Abstraktia konetta on kutsuttu myös virtuaalikoneeksi, josta on peräisin myös Java-virtuaalikone (Java Virtual Machine, JVM) -nimitys. Kielen alunperin kehittänyt James Gosling antoi sille nimen Oak (tammi). Myöhemmin selvisi, että sen niminen ohjelmointikieli oli jo olemassa, joten kielen nimeksi vaihdettiin Java.

Vuosina 1992-1994 Sun kehitteli kielensä avulla muutamia kulutuselektroniikan tuotteita, kuten älykkään kaukosäätimen ja kaapelitelevisioon liitettävän, lisäpalveluja käsittelevän lisälaitteen. Kulutuselektroniikkayhtiöt eivät olleet kuitenkaan kiinnostuneita niistä. Samalla eräs Internetin osa, WWW kasvoi valtavasti. WWW:n ydin on selain, joka ottaa hypertekstisivun ja muuttaa sen näytöllä esitettävään muotoon. Vuonna 1994 käytetyin selain oli Mosaic, joka oli Illinoisin yliopistossa 1993 valmistunut ei-kaupallinen selain.

Vuoden 1994 keskivaiheilla Java-kielen kehittäjät oivalsivat, että he voisivat rakentaa sillä selaimen. He havaitsivat, että Internetin asiakas—palvelin -arkkitehtuurissa tarvittiin niitä ominaisuuksia, joita he olivat jo kehittäneet kieleensä, kuten esimerkiksi arkkitehtuurineutraalisuutta ja turvallisuutta. Nämä eivät olleet yhtä tärkeitä asioita työasemien tapauksessa. He rakensivat Javalla HotJava-selaimen, joka pystyi tulkitsemaan välikieltä. Tämä oli tärkeää, jotta sovelmia (applet) voitiin suorittaa selaimessa. Tämä teknologia esiteltiin toukokuussa 1995.

Läpimurto Javan laajaan käyttöön tapahtui syksyllä 1995, kun Netscape päätti alkaa tukea Javaa selaimessaan. Netscapen 2.0 -selain julkistettiin tammikuussa 1996, mistä lähtien selaimen versiot ovat tukeneet Javaa. Muita Java-teknologian lisensoijia olivat esimerkiksi IBM, Symantec ja Borland (nykyisin Inprise). Jopa Microsoft on lisensoinut sen ja tukee Javaa, joten muun muassa Internet Explorer on tukenut Javaa versiosta 3.0 lähtien. Tosin Sun syytti myöhemmin Microsoftia oikeudessa lisenssiehtojen rikkomisesta. Oikeuskiista ratkesi vuoden 2001 tammikuussa, jolloin Microsoft sitoutui jatkossa täyttämään tuotteissaan Sunin yhteensopivuustestit (Sun Microsystems Inc. 2001).

Sun julkisti Javasta ensimmäisen version 1.0 syksyllä 1995 (Cadenhead 1999, 46). Muutaman kuukauden kuluttua Javasta julkistettiin versio 1.02 (Horstmann & Cornell 1997, 15). Tämän version mukainen Java ei kuitenkaan ollut vielä kypsä laajamittaiseen sovelluskehitykseen.

Tämän jälkeen Javasta on tehty kaksi suurempaa julkistusta (Cadenhead 1999, 46). Keväällä 1997 esiteltiin versio 1.1, jonka keskeisimmät parannukset liittyivät käyttöliittymän luomiseen ja käsittelyyn. Suuria muutoksia tehtiin esimerkiksi tapahtumankäsittelyyn. Syksyllä 1998 julkistettiin versio 1.2, jota kutsutaan nimellä Java 2. Tämä on yli kolme kertaa laajempi kuin Javan ensimmäinen versio. Huhtikuussa 2000 julkistettiin versio 1.3. Tästä julkistuksesta käytetään nimitystä J2SE-alusta (Java 2 Standard Edition) versio 1.3. Nämä parannukset tasavertaistavat kielen asemaa kilpailussa muiden yleiskäyttöisten ohjelmointikielten kanssa.

5.2 Siirrettävyyden perusmekanismit Javassa

Jotta Javan siirrettävyyden perusmekanismejä voidaan tarkastella, täytyy selvittää ensin Javan perusteita. Mikä Java sitten on? Java on sekä ohjelmointikieli että alusta (Campione & Walrath 1998, 3). Alusta on laitteisto- tai sovellusympäristö, jossa ohjelma suoritetaan. Kramer (1996) pitää alustoina esimerkiksi Microsoft Windowsia ja Macintoshia. Java-alusta eroaa Campionen ja Walrathin (1998, 4) mukaan useimmista muista alustoista siinä, että se on pelkkä sovellusalusta, joka sijoittuu muiden laitteistopohjaisten alustojen päälle.

5.2.1 Java ohjelmointikielenä siirrettävyyden näkökulmasta

Siirrettävyyden kannalta tarkasteltuna Java-ohjelmointikielen suunnittelutavoitteina on ollut arkkitehtuuririippumattomuus, siirrettävyys, tulkittavuus, dynaamisuus, hyvä suori-

tuskyky ja monisäikeisyys, joita muun muassa Gosling (1996) sekä Horstmann ja Cornell (1997) tarkastelevat.

Java on Goslingin (1996) mukaan suunniteltu tukemaan ohjelmia, joita tullaan jakamaan heterogeenisessä verkkoympäristöissä. Tällaisissa ympäristöissä ohjelmia täytyy kyetä suorittamaan erilaisissa laitteistoissa sekä käyttöjärjestelmissä. Jotta toimintaympäristöjen erilaisuuteen mukauduttaisiin, Java-kääntäjä tuottaa tavukoodia (bytecode), joka on arkkitehtuurineutraali, siirrettävä välikieli. Tavukoodi on suunniteltu ohjelman koodin kuljettamiseksi moniin erilaisiin laitteisto- ja sovellusalustoihin, sillä sama Javan tavukoodi voidaan suorittaa millä tahansa alustalla, jossa on Javan ajonaikainen alusta. Tavukoodi on suunniteltu helposti tulkittavaksi missä tahansa koneessa ja helposti lennossa käännettäväksi konekielelle. Horstmannin ja Cornellin (1997, 8) mukaan tavukoodikäskyjoukko toimii hyvin yleisimmissä tietokonearkkitehtuureissa. Lisäksi koodit on suunniteltu helposti muutettaviksi todellisiksi konekäskyiksi. Horstmann ja Cornell (1997, 8) muistuttavat kuitenkin, että suorituskyky on pääroolissa tavukoodeja käytettäessä.

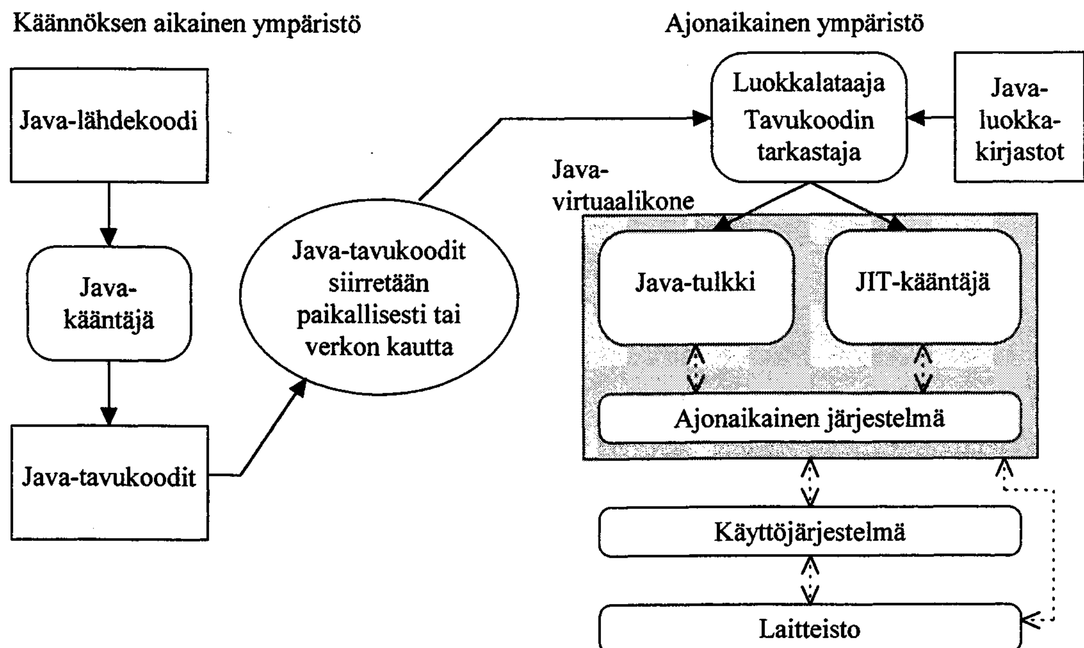
Java on tiukka peruskielen määrittelyssä: Javan määrityksissä ei Goslingin (1996) mukaan pitäisi olla toteutusriippuvia piirteitä, toisin kuin esimerkiksi C:ssä tai C++:ssa. Javassa määritellään eksplisiittisesti perustietotyyppien koot ja aritmeettisten operaatioiden käyttäytyminen. Tämän vuoksi Java-ohjelmissa ei ole tietotyyppien yhteensopivuusongelmia eri laitteisto- ja sovellusarkkitehtuureissa (Gosling 1996). Esimerkiksi tietotyyppi kokonaisluku (integer) on kooltaan aina 32-bittinen. C++:ssa kokonaisluku voi olla 16-bittinen, 32-bittinen tai minkä tahansa muun kokoinen, jonka kääntäjän tekijä määrittää. Javan merkkijonot tallennetaan standardissa Unicode-muodossa (Horstmann & Cornell 1997, 9). Myös binäärinen data tallennetaan kiinnitetyssä muodossa.

Java-tulkki voi suorittaa tavukoodia suoraan missä tahansa koneessa, johon tulkki ja ajonaikainen järjestelmä on siirretty (Gosling 1996). Java-ympäristössä erillistä kääntämisen jälkeistä linkittämisvaihetta ei tarvita. Linkittämisen, joka on varsinaisesti uusien luokkien lataamisprosessi, tekee luokkalataaja. Näin tapahtuva linkittäminen on vä-

hittäisempi ja kevyempi prosessi kuin normaali linkittäminen, joten kehitysprosessi voi olla nopea ja tutkiva. Tästä saattaa olla etua sovelluskehityksessä, tosin kehitysprosessin etuja on Horstmannin ja Cornellin mielestä (1997, 9) liioiteltu.

Java-kääntäjä rajoittuu käännöksen aikaiseen staattiseen tarkastukseen. Kieli ja ajonaikainen järjestelmä ovat dynaamisia niiden linkitysvaiheissa (Gosling 1996). Luokkia linkitetään vain tarvittaessa. Uudet koodimodulit voidaan linkittää vaadittaessa useista eri lähteistä, jopa verkon yli. Java on monin tavoin dynaamisempi kieli kuin C tai C++. Se suunniteltiin mukautumaan kehittyvään ympäristöön. Luokkakirjastoihin voidaan vapaasti lisätä uusia metodeja ja ilmentymämuuttujia ilman minkäänlaista vaikutusta niiden asiakkaisiin. Tämä on tärkeä piirre niissä tilanteissa, kun koodia tarvitsee lisätä suorituksessa olevaan ohjelmaan (Horstmann & Cornell 1997, 11). Tällaisia päivitystarpeita voisi ilmetä esimerkiksi koodissa, joka ladataan Internetistä suoritettavaksi selaimessa.

Javan käännöksen ja ajonaikaisia ympäristöjä havainnollistetaan kuviossa 8 Kramerin (1996) esityksen pohjalta. Ajonaikaisella ympäristöllä tarkoitetaan Javan ajonaikaista alustaa. Kuviossa 8 yhtenäiset nuolet kuvaavat koodin fyysistä siirtymistä ja katkonaiset nuolet kommunikointia.



KUVIO 8. Käännöksen ja ajonaikaiset ympäristöt (vrt. Kramer 1996).

Java-ohjelman kehittäminen on toisenlaista kuin perinteisen ohjelman (vrt. kuvio 3). Kehittäjä kirjoittaa Java-kielellä lähdekoodia, eli .java-päätteisiä tiedostoja, ja kääntää ne tavukoodeiksi, joita ovat .class-päätteiset tiedostot. Kun sovellus (application) tai sovelma käynnistetään suoritukseen, luokkalataaja lataa tavukooditiedostot muistiin. Välittömästi tämän jälkeen virtuaalikoneessa tulkki tulkitsee tavukoodit. Vaihtoehtoisesti tulkin sijaan voidaan käyttää JIT-kääntäjää (Just-in-Time) kääntämään tavukoodit konekoodiksi. Tulkki ja JIT-kääntäjä toimivat ajonaikaisen järjestelmän ympäristössä. Javan luokkakirjastojen luokat ladataan dynaamisesti, kun sovellus tai sovelma tarvitsee niitä.

Vaikka suorituskyky, johon tavukoodien tulkitsemisella päästään, ylittää Goslingin (1996) mielestä yleensä käyttötarkoitukseen tarvittavan suorituskyvyn, on olemassa tilanteita, joissa parempaa suorituskykyä tarvitaan. Tavukoodit voidaan kääntää suorituksen aikana konekoodiksi. Kuitenkaan arvio "hyvä suorituskyky" ei ole Horstmannin ja Cornellin (1997, 10) mukaan oikeaan osuva kuvaamaan Java-tulkin käyttämistä Java-pohjaisen ohjelman suoritukseen. Kuvaavampi arvio olisi keskinkertaisen ja huonon välillä. JIT-kääntäjillä suorituskyvyssä päästään lähemmäksi kääntäjien suorituskykyä (Horstmann & Cornell 1997, 10; Andersson ym. 1999, 4). JIT-kääntäjät toimivat kääntämällä ohjelman tavukoodit kokonaisuudessaan konekielelle suuremmissa pätkissä ja tuloksena syntyvä koodi ajetaan kohdeympäristön koneessa (Andersson ym. 1999, 4). Java-koodi voidaan kääntää myös perinteisesti kohdealustaa varten (Andersson ym. 1999, 292). Esimerkiksi Symantec tarjoaa tällaisen kääntäjän, jonka avulla voitaisiin tuottaa Java-ohjelmia, joilla heidän mukaansa olisi samanlainen suorituskyky kuin C++-ohjelmilla. Kuitenkin tällaisessa kohdealustalle tehtävässä kääntämisessä katoaa Javan riippumattomuus alustasta eli siirrettävyys alustojen välillä. Alustakohtainen sovelma ei ole siirrettävä Internetissä tai sellaisessa intranetissä tai verkossa, joka koostuu erilaisista tietokonearkkitehtuureista.

Javan versiossa 1.3 ajonaikaiseen ympäristöön on tehty monia suorituskykyyn liittyviä parannuksia. Java 1.3:ssa käytetään Java HotSpot -teknologiaan perustuvia asiakas- ja palvelinvirtuaalikoneita. Asiakasvirtuaalikone on viritetty maksimoimaan suorituskyky asiakasjärjestelmissä parantamalla suorituskykyä käynnistyksen ja muistinhallinnan

alueilla. Palvelinvirtuaalikone on suunniteltu maksimoimaan ohjelman ajonaikainen suorituskyky ja se on tarkoitettu palvelinsovelluksille, joiden kannalta nopein mahdollinen toimintanopeus on paljon tärkeämpää kuin nopein mahdollinen käynnistysaika. (Sun Microsystems Inc. 2000b.)

Seuraavassa on joitakin Java HotSpot -teknologian molemmille virtuaalikonetoteutuksille yhteisiä piirteitä:

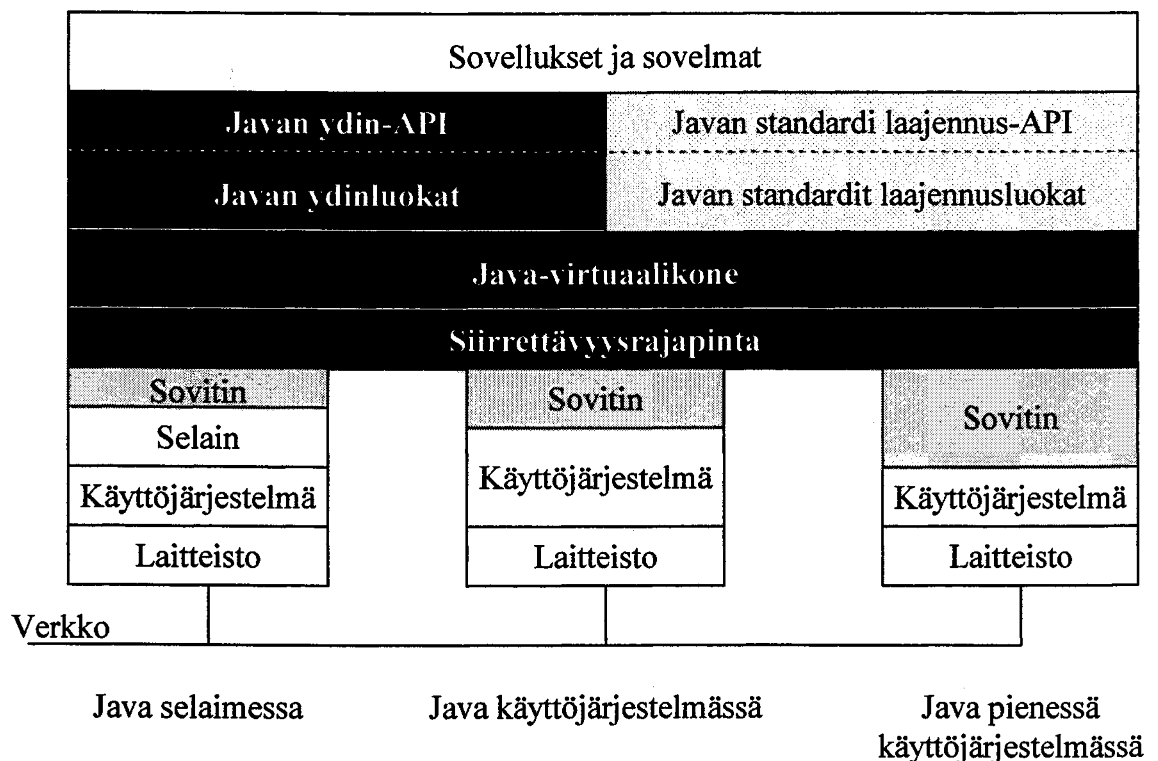
- Kääntäjä on adaptiivinen. Sovellukset käynnistetään normaalilla tulkilla, mutta ohjelman suorituksen aikana koodia tutkitaan, jotta voitaisiin tunnistaa pullonkaulat tai "kuumat kohdat" suorituskyvyn kannalta. Java HotSpot -asiakasvirtuaalikone kääntää nämä suorituskyvyn kannalta kriittiset koodin osat tehostaakseen suorituskykyä ja välttääkseen harvoin käytetyn koodin kääntämistä, mitä on suurin osa koodista. Asiakasvirtuaalikone käyttää myös mukautuvaa kääntäjää päättämään lennossa, miten käännetyn koodin voisi parhaiten optimoida käyttäen erilaisia tekniikoita, kuten esimerkiksi in-line -tekniikkaa. Kääntäjän suorituksen aikainen analyysi parantaa mahdollisuuksia osoittaa, mitkä optimoinnit tuottavat suurimmat suorituskykyedut.
- Muistin varaamista ja roskien keruuta on parannettu. Java HotSpot -asiakasvirtuaalikone varaa muistia olioille paljon nopeammin kuin klassinen virtuaalikone. Asiakasvirtuaalikoneessa roskien keruu on nopeampaa ja tehokkaampaa kuin klassisessa virtuaalikoneessa.
- Parannuksia on tehty säikeiden synkronointiin. Java-ohjelmointikieli mahdollistaa useiden samanaikaisten ohjelman suorituspolkujen, säikeiden, käytön. Java HotSpot -asiakasvirtuaalikone tarjoaa nopeamman säikeiden käsittelykyvyn, joka on suunniteltu skaalautumaan helposti suuriin jaettua muistia käyttäviin moniprosessoripalvelimiin. (Sun Microsystems Inc. 2000d.)

Monisäikeisyys (multithreaded) tarkoittaa yhden ohjelman kykyä tehdä enemmän kuin yhtä asiaa samaan aikaan (Horstmann & Cornell 1997, 10). Esimerkiksi ohjelma voisi tulostaa dokumenttia ja samaan aikaan ottaa vastaan faksin. Javan monisäikeisyys mahdollistaa ohjelmien rakentamisen, joissa toimii monia samanaikaisia tehtäväsäikeitä (Gosling 1996). Täten loppukäyttäjälle olisi mahdollista tarjota korkean tason vuorovaikutteisuus. Huonona puolena Javan monisäikeisyydessä on Horstmannin ja Cornellin

(1997, 10) väittämän mukaan se, että varsinaiset säietoteutukset vaihtelevat laajasti tärkeimmillä alustoilla ja tässä suhteessa Java ei pyri olemaan alustasta riippumaton. Sun Microsystems (2000c) myöntää, että säikeiden ajastus saattaa vaihdella eri alustoilla. Varmuudella ainakin koodi, jota tarvitaan kutsumaan monisäikeisyyttä, pysyy samana kohdealustasta riippumatta (Horstmann & Cornell 1997, 10). Säikeitä ei tulla tässä työssä tarkastelemaan kovin tarkasti, koska kyseisestä aiheesta on tekeillä toinen pro gradu - tutkielma, josta voi katsoa lisätietoja.

5.2.2 Java-alusta

Java-alustalla on kaksi keskeistä osaa: Java-virtuaalikone ja Java-sovellusliittymä (Java Application Programming Interface, Java API) (Campione & Walrath 1998, 4). Java-alustan rakenne on esitetty kuviossa 9.



KUVIO 9. Java-alustan rakenne (vrt. Kramer 1996).

Java-virtuaalikone on alustan ydin. Siirrettävyysrajapinta sijaitsee virtuaalikoneen ja käyttöjärjestelmän tai selaimen välissä. Tällä siirrettävyysrajapinnalla on alustariippumaton osa, joka on esitetty mustalla värillä, ja alustariippuva osa eli sovittimet, jotka on kuvattu harmaalla värillä. Käyttöjärjestelmä tarjoaa muun muassa ikkunointi-, arkistointi- ja verkkotoiminnallisuuden. Eri koneet voidaan yhdistää verkolla, kuten kuviossa 9 on esitetty. (Kramer 1996.)

Java-virtuaalikone on "mukautuva" tietokone, joka voidaan toteuttaa sovelluksella tai laitteistolla (Kramer 1996). Se on abstrakti kone, joka on suunniteltu toteutettavaksi olemassa olevien prosessorien päälle. Java-virtuaalikone pohjautuu Goslingin (1996) mukaan pääasiassa POSIX:n standardiin siirrettävän systeemirajapinnan määritykseen. Java-virtuaalikone mahdollistaa riippumattomuuden allaolevasta käyttöjärjestelmästä ja laitteistosta (Kramer 1996). Se piilottaa allaolevan käyttöjärjestelmän Javalla tehdyiltä sovelluksilta ja sovelmilta. Lisäksi virtuaalikone määrittelee käskyjoukon, jota voidaan käyttää koneesta riippumattomissa tavukoodeissa. Siirrettävyysrajapinta ja sovittimet helpottavat Java-virtuaalikoneen siirtämistä uuteen käyttöjärjestelmään tai selaimeen ilman, että se jouduttaisiin kirjoittamaan kokonaan uudestaan. Java-virtuaalikoneen toteutus uusissa arkkitehtuureissa on suhteellisen suoraviivainen tehtävä, mikäli kohdeympäristö tukee perusvaatimuksia, kuten esimerkiksi monisäikeisyyttä (Gosling 1996).

Java-sovellusliittymä sisältää sekä ydin-API:n että laajennus-API:n. Luokat ovat tämän sovellusliittymän toteutus. Java-sovellusliittymä muodostaa teollisuusstandardin rajapinnan sovelluksille ja sovelmille allaolevasta käyttöjärjestelmästä riippumatta. Java-sovellusliittymä on järjestetty ryhmiin tai joukkoihin. Kukin niistä voidaan toteuttaa yhtenä tai useampana pakettina, jotka ovat omia nimiavaruuksia. Kukin paketti ryhmittää joukon luokkia ja rajapintoja, jotka määrittävät joukon niihin liittyviä kenttiä, muodostimia ja metodeja. (Kramer 1996.)

Javan ydin-API käsittää perusluokkakirjastot, joita käyttämällä ohjelma voidaan suorittaa missä tahansa ilman lisäluokkakirjastojen tarvetta. Java-alustan lisensoijat ovat teh-

neet sopimuksen ydin-API:n sisällyttämisestä Java-alustan toteutuksiinsa. (Kramer 1996.)

Javan laajennus-API lisää ydin-API:n tarjoamia mahdollisuuksia. Sitä määritellään JavaSoftin ja muiden Sunin yhteistyökumppaneiden toimesta. Jotkut näistä laajennuksista siirtyvät lopulta ydin-API:in. Laajennus-API:in kuuluvat osat ovat teollisuusstandardeja ja ne muodostavat julkaistun, yhdenmukaisen sekä avoimen API:n, jonka kuka tahansa voi toteuttaa. Kerran määriteltynä laajennukset voidaan lisätä, mutta takaisinpäin yhteensopivuuden ylläpitämiseksi niitä ei muuteta sellaisella tavalla, että kutsut niihin epäonnistuisivat. Myös sovelma, sovellus tai allaoleva käyttöjärjestelmä voivat tarjota muita laajennettuja, tosin ei-teollisuusstandardeja, API-kirjastoja. (Kramer 1996.)

Java-alustoja on kolme eri versiota. Kuviossa 9 mustalla värillä esitetty osio sekä harmaat, sovittimiksi nimetyt osiot muodostavat Javan perusalustan (Kramer 1996). Javan perusalusta on tarkoitettu suorittamaan Javalla tehtyjä sovelluksia ja sovelmia. Nykyisin Javan perusalustaa kutsutaan J2SE:ksi (Java 2 Standard Edition) (Thomas 1999, 6). J2SE määrittelee teollisuusstandardin Java-kokoonpanon, joka tarvitaan yleiskäyttöisissä Java-ohjelmissa. J2SE-alusta sisältää täyden tuen Javan ydinluokkien sovellusliittymille.

Kun edelliseen kokoonpanoon lisätään teollisuusstandardeja API-kirjastoja, muodostuu J2SE:tä laajempi J2EE-alusta (Java 2 Enterprise Edition). J2EE on J2SE:n ylijoukko, joka määrittelee laajennetun enterprise-tason ohjelmistoja tukevan Java-kokoonpanon (Thomas 1999, 7). Termi "enterprise" tarkoittaa Thomasin (1999, 1) mukaan äärimmäisen vikasietoista tietojenkäsittelyä. Enterprise-ohjelmistot tukevat ydinliiketoimintaa, ja virheet näissä ohjelmistoissa aiheuttavat liiketoiminnan keskeytymisen.

Kolmas Java-alusta on J2ME (Java 2 Micro Edition). J2ME on Java 2 alustan versio, joka on suunnattu kulutuselektroniikkaan ja sulautettuja ohjelmistoja sisältäviin laitteistoihin (Sun Microsystems Inc. 2000a). Esimerkkejä näistä laitteista ovat matkapuhelimet ja tulostimet. Tällaisilla laitteilla on usein erityisiä rajoitteita, kuten esimerkiksi pieni muisti. J2ME koostuu virtuaalikoneesta ja joukosta tarvittavia sovellusliittymiä,

jotta kohdelaitteelle voidaan tarjota sovelias ajonaikainen ympäristö (Sun Microsystems Inc. 2000a). J2ME:n avulla kirjoitetut sovellukset ovat ylöspäin skaalautuvia J2SE:hen ja J2EE:hen nähden (Sun Microsystems Inc. 2000a).

5.2.3 Lisänäkökohtia Javan siirrettävyyden perusmekanismeihin

Java-sovellusliittymän toteutuksien pitäisi tehdä sitä, mikä on tarkoituksenmukaista kullakin suoritusalustalla sen laitteistorajoitusten puitteissa (Kramer 1996). Esimerkiksi käyttöjärjestelmät, joilla on mahdollisuus käyttää kaiutinta, voivat tuottaa ääntä. Kuitenkin suurtietokoneen käyttöjärjestelmä, jolla ei ole mahdollisuutta käyttää kaiutinta, tekee vastaavasti "ei operaatiota" -ilmoituksen tai jonkun muun hyvin määritellyn käyttäytymisen, kuten antaa poikkeuksen.

Siirrettävyyden näkökulmasta tällainen ei-toteutettu toiminta voitaisiin ohjelmoida ohjelmaan ja ohjelma voitaisiin kääntää ilman ongelmia. Ohjelma voitaisiin myös suorittaa kohdeympäristössä, mutta se ei toimisikaan halutulla tavalla. Siten tällaiset toteuttamattomat ominaisuudet rikkovat ohjelman täydellisen siirrettävyyden. Voidaan kuitenkin kysyä, olisiko tällainen ohjelma alunperin haluttu ottaa käyttöön kohdeympäristössä, jos tiedetään, ettei siellä ole mahdollista tuottaa ääntä. Ehkä koko ominaisuutta ei olisi haluttu ohjelmoida ohjelmaan, jos olisi tiedetty, ettei kyseinen ominaisuus toimi kaikissa kohdeympäristöissä. Tässä tullaan valintatilanteeseen: Java-sovellusliittymä mahdollistaisi laajempien toimintojen ohjelmoimisen ohjelmaan kuin mitä kaikissa kohdeympäristöissä voidaan toteuttaa. Siten sovelluskehittäjän tulisi ensinnäkin tietää, mitkä Java-sovellusliittymän palveluista ovat toteutettavissa ja toteutettu kohdeympäristöön tai kaikkiin kohdeympäristöihin. Sitten hänen tulisi valita Java-sovellusliittymän palveluista vain ne, jotka varmuudella ovat olemassa kohdeympäristössä.

Java-kielen sisältä sovelluskehittäjät voivat myös kutsua paikallisia, järjestelmästä riippuvaisia metodeja nopeuden tai erityisen toiminnallisuuden saavuttamiseksi (Kramer 1996). Näitä ovat esimerkiksi alustasta riippuvaiset C-, C++- tai muun kieliset metodit,

jotka kääntyvät tiettyyn allaolevaan käyttöjärjestelmään. Nämä paikalliset metodit toteutetaan käyttäen virtuaalikoneen tekijöiden tarjoamaa sovellusliittymää (Arnold & Gosling 1998, 57). Tällainen sovellusliittymä on esimerkiksi C-ohjelmoijille tarkoitettu JNI (Java Native Interface). Paikallisia metodeja käytettäessä sovellus tai sovelma ei ole sellaisenaan siirrettävissä toiseen käyttöjärjestelmäympäristöön.

Sun Microsystems (2000c) tarjoaa varmennusohjelman (100% Pure Java Certification Program), joka on tarkoitettu mahdollisuudeksi todistaa Javalla tehdyn tuotteen siirrettävyys yksilöidyllä ja yhdenmukaisella tavalla. Varmennuksen läpäisseessä Java-ohjelmassa voidaan käyttää "100% Pure Java" -logoa. Sun Microsystemsin (2000c) mukaan varmennuksen pitäisi todentaa, että ohjelma olisi täysin siirrettävä. Täsmällisemmin Sun Microsystems (2000c) kuvaa varmennuksen takaavan, että varmennetun ohjelman koodi riippuu vain dokumentoidusta ja määritellystä Java-alustasta. Se takaa myös sen, että ohjelma suorittuu millä tahansa tietokoneella, jossa on Java-yhteensopiva ohjelmaympäristö (Sun Microsystems Inc. 2000c). Varmennuksen normina ei ole varmennettavan ohjelman identtinen käyttäytyminen vaan toiminnallinen vastaavuus kohdealustalla (Sun Microsystems Inc. 2000c). Tässä valossa vaikuttaa edelleen pätevältä Lewandowskin (1998, 17) arvio, jonka mukaan eri virtuaalikoneiden välillä on lukuisia eroavaisuuksia, jotka vaikuttavat Java-ohjelman samanlaiseen suoritukseen ja käyttäytymiseen eri kohdealustoilla. Graafisen käyttöliittymän suhteen varmennuksen normina on, että ohjelma näyttää samat elementit kohdealustoilla, eikä se, että näillä elementeillä olisi sama ulkoasu (Sun Microsystems Inc. 2000c). Todellisuudessa varmennusprosessi keskittyykin siis ohjelman "puhtauden" (purity) testaamiseen täyden siirrettävyyden sijaan. Ohjelman "puhtaus" mitataan tarkemmin ohjelman alarajalta eli alustarajalta kuin ylärajalta eli käyttäjärajalta (Sun Microsystems Inc. 2000c). Tätä mitattua "puhtautta" Sun Microsystems (2000c) pitää hyvänä siirrettävyyden ennustajana siten, että ohjelmien puhtauden tarkastus johtaa ohjelmien parempaan siirrettävyyteen.

5.3 Java-ohjelmoinnin siirrettävyysongelmia ja niiden ratkaisukeinoja

Java-alustan luontainen siirrettävyys ei yksin takaa siirrettävyyttä jokaiselle Java-ohjelmalle. Monet "ansat" voivat vaikuttaa epäedullisesti ohjelmien siirrettävyyteen. Sovelluskehittäjien on hyödyllistä tietää niistä ja kuinka näitä "ansoja" voitaisiin kiertää. Seuraavassa on tarkasteltu Java-ohjelmoinnin keskeisimpiä siirrettävyysongelmia ja niiden ratkaisukeinoja.

Säikeiden harkitsematon käyttö saattaa vaarantaa ohjelman siirrettävyyden. Kuten kohdassa 5.2.1 todettiin, säikeiden ajastus saattaa erota eri alustoilla. Ohjelma ei ole siirrettävä, jos ohjelmassa luotetaan säikeiden prioriteetteihin eli tärkeysjärjestyksiin, kun estetään kahden säikeen yhtäaikainen pääsy samaan olioon (Sun Microsystems Inc. 2000c). Sun Microsystems (2000c) esittää erääksi keinoksi välttää näitä ongelmia metodien pakottamisen synkronoiduiksi. Tällöin datan yllättävän turmeltumisen sijaan saatetaan saada synkronointivirheitä, jotka näkyvät ilmeisinä lukkiutumina, mikäli synkronointia ei suunnitella ja toteuteta huolellisesti.

Paikallisten järjestelmäriippuvaisten metodien kutsuminen ei ole luonnostaan siirrettävää. Ratkaisuna tähän on yksinkertaisen protokollan määrittäminen paikalliseen palveluun ja sitten Java-ohjelman kirjoittaminen tämän protokollan asiakkaaksi. Tämän jälkeen paikallinen metodi tulisi kirjoittaa uudelleen Javaa käyttäen. Paikalliset metodit voidaan kuitenkin joutua rajaamaan yhteen luokkaan ja tarjoamaan sitten tälle luokalle toteutukset kullakin Java-alustalla. JNI helpottaa paikallisen koodin siirtämistä mutta sen avulla ei voida tehdä paikallista koodia alustariippumattomaksi. (Sun Microsystems Inc. 2000c.)

Runtime-luokan palvelut ovat järjestelmäriippuvaisia. Tämän luokan kautta päästään käsiksi systeemitason resursseihin. Java-ohjelman siirrettävyys uhrataan, jos Java-ohjelman olioista lähetetään viestejä suoraan Runtime-oliolle. Runtime-luokka edustaa ohjelman ajonaikaista ympäristöä ja se on läheisessä yhteydessä Java-tulkin ja Java-virtuaalikoneen toteutusten sekä isäntäkäyttöjärjestelmän kanssa. Runtime-olio kom-

munikoi suoraan edellisten kanssa vastaanottamalla informaatiota ja kutsumalla järjestelmästä riippuvaisia metodeja. Esimerkiksi Unixissa Runtime-olio saattaisi tukea getenv() ja setenv() -metodeja. Toinen Runtime-olio esimerkiksi Macintoshissa ei varmaankaan tukisi näitä metodeja, koska niitä ei ole isäntäkäyttöjärjestelmässä. Kuitenkin Macintoshissa Runtime-olio saattaisi tukea muita palveluja. (Campione & Walrath 1998, 267, 291—292.)

Yleisesti käytetty java.lang.Runtime.exec() -metodi ei ole yleensä siirrettävä. Kaikilla alustoilla ei ole ajettavia ohjelmia. Siten Runtime.exec() -metodin aiheuttamat virheet kannattaa käsitellä kunnolla. Etenkin sellainen virhe, joka aiheutuu siitä, että pyydetty ohjelma ei ole saatavilla. Suositeltavaa on myös antaa ohjelman käyttäjän määrittää kutsuttava ulkopuolinen ohjelma. (Sun Microsystems Inc. 2000c.)

System-luokan kautta voidaan käyttää joitakin systeemitason resursseja, mutta kaikkiin sen tarjoamiin palveluihin ei pidä luottaa siirrettävyyteen pyrittäessä. System.in, System.out tai System.err -virtojen käyttö ei ole täysin siirrettävää, koska kaikilla Java-alustoilla ei ole olemassa syöttö- ja tulostusvirtoja (Sun Microsystems Inc. 2000c; Campione & Walrath 1998, 270). Näiden virtojen käytön sijaan käyttäjälle voitaisiin Campionen ja Walrathin (1998, 270) mukaan tarjota graafinen käyttöliittymä syötteiden saamiseksi ja tekstien näyttämiseksi. Myös System-luokan tarjoamien systeemiasetusten muuttaminen ei ole suositeltavaa siirrettävyyteen pyrittäessä, sillä tietyissä tilanteissa niiden muuttaminen saattaa tuottaa ennustamattomia lopputuloksia (Campione & Walrath 1998, 276). System-luokka pyrkii kuitenkin eristämään käytettävät systeemitason resurssit vähän samaan tapaan kuin luvussa 2.3 esitelty Sommervillen (1995) siirrettävyysrajapinta.

Komentorivin syntaksi ja käyttötavat ovat melko erilaisia eri alustoilla. Siirrettävyyden kannalta parasta olisi olla käyttämättä komentoriviä lainkaan, koska sitä ei edes ole olemassa kaikilla Java-alustoilla. Mikäli komentoriviä kuitenkin käytetään, suositeltavaa olisi tehdä se laajasti ymmärrettyjen POSIX:in tapojen mukaisesti. (Sun Microsystems Inc. 2000c; Campione & Walrath 1998, 265.)

Eri koneilla on erilaiset sisäiset tekstin esitysmuodot. Java-alusta käyttää sisäisesti Unicodea, joka on kansainvälinen standardipohjainen ratkaisu ongelmaan. Silti tekstiä täytyy kirjoittaa tiedostoihin ja lukea tiedostoista. Javan 1.0 version syöttö- ja tulostusluokat eivät ole siirrettäviä sellaisiin laitteistoarkkitehtuureihin, joissa käytetään muita kuin ASCII-merkistön mukaisia tiedostoja. Näiden sijaan suositeltavaa onkin käyttää Javan versiosta 1.1 alkaen löytyviä luku- ja kirjoitusluokkia. Kuitenkin ongelmia voi aiheutua jopa silloin, kun luetaan tai kirjoitetaan ASCII-tiedostoja, koska ASCII-standardi ei ole tarkka rivin katkaisumerkin suhteen. Siirrettävä tapa kirjoittaa tiedostoon on käyttää println() -metodeja. Tekstin kirjoittamiseen voidaan käyttää myös writeLine() -metodia. Järjestelmästä saatavaa rivierotinta voidaan myös hyödyntää. Tekstin lukemiseen kannattaa käyttää BufferedReader -luokan readLine() -metodia. (Sun Microsystems Inc. 2000c.)

Koodiin kiinnitetyissä tiedostonimissä saattaa ilmetä siirrettävyysongelmia. Siirrettävyysongelmia aiheuttavat myös koodiin kiinnitetyt polut, joissa on hakemistonimi ja tiedostonimi. On huomattava, että käsite "absoluuttinen polku" on myös järjestelmäriippuva, sillä esimerkiksi Unixissa absoluuttiset polut alkavat "/"-merkillä, kun taas Windowsissa absoluuttiset polut voivat alkaa millä tahansa kirjaimella. Siirrettäviä ratkaisuja ovat järjestelmäasetusten määrittämien paikallisen tiedoston erottimen ja lähtöhakemiston käyttäminen tai tiedostodialogin käyttäminen tiedostonimen kysymiseen käyttäjältä. (Sun Microsystems Inc. 2000c.)

Jos ohjelman koodiin kiinnitetään tietyn JDBC-ajurin (Java Database Connectivity) nimi, ohjelma on yhtä siirrettävä kuin tämä ajuri. Tällaisessa tapauksessa on suositeltavaa määrittää tarkka JDBC-ajurin nimi konfiguroitavaksi siten, että ajurin nimi on käyttäjän valittavissa joko asennuksessa tai ajonaikana. Tämä voidaan tehdä esimerkiksi käyttämällä asetustiedostoa tai järjestelmäasetusta, jolla saadaan selville järjestelmästä saatavilla olevat JDBC-ajurit. (Sun Microsystems Inc. 2000c.)

Ongelmia saattaa aiheuttaa myös java.net.InetAddress.getHostName() -metodin palauttaman merkkijonon muoto. Joissakin tapauksissa metodi palauttaa täysimittaisen alueen (domain) nimen, kun toisissa tapauksissa saadaan vain isäntä (host) -osa tästä nimestä.

Tästä johtuen sellaisissa tapauksissa, joissa tämä nimi täytyy välittää eri alueelle, saattaa olla parasta antaa isäntä -nimen lisäksi IP-numero (Internet Protocol). (Sun Microsystems Inc. 2000c.)

Tietyt Javan ydin-API:n metodit on merkattu vältettäviksi. Vaikka nämä metodit toimivat senhetkisessä versiossa, ne on merkitty poistettaviksi jossakin vaiheessa. Siten onkin suositeltavaa korvata nämä metodit API-dokumentaatiossa ehdotetuilla muilla metodeilla. (Sun Microsystems Inc. 2000c.)

Asennettaessa ohjelmaa erilaisille alustoille tai yritettäessä ajaa asennettua ohjelmaa voi ilmaantua useita tiedostonimirajoitteista johtuvia ongelmia. Nämä ongelmat saattavat häiritä Java-virtuaalikoneen luokkatiedostojen paikannusta, joka on riippuvainen yksinkertaisesta yhteydestä luokannimen ja tiedostonimen välillä. Ongelmallisia rajoitteita ovat tiedostonimen pituus, isojen ja pienten kirjainten erottamattomuus, epätäydellinen Unicode-tuki sekä erikoiskäyttöön tarkoitetut tiedostonimet. Tiedostonimen pituuden rajallisuus on ongelma erityisesti sisäisten luokkien kanssa, jotka esitetään yhdistetyn nimen avulla luokkatiedostoina. Joillakin alustoilla ei erotella isoja ja pieniä kirjaimia tiedostonimiä vertailtaessa. Jos ohjelmassa on kaksi luokkaa tai pakettia, jotka eroavat vain isojen ja pienten kirjaimien perusteella, ei ohjelma ole siirrettävä tämän tyyppisillä alustoilla. Ohjelman luokkanimet saattavat sisältää Unicode-merkkejä, joita ei kuitenkaan voi käyttää tiedostonimissä kaikilla alustoilla. Joillakin alustoilla on määritelty jokin erityinen tarkoitus muutamille tiedostonimille, joita ovat esimerkiksi "LPT" tai "con". Näitä tiedostonimiä ei voi käyttää paketin nimen osana luokille, jotka asennetaan tiedostoina tällaisiin järjestelmiin. Ratkaisuna edellisiin ongelmiin on pakata ohjelman luokat .jar -päätteisiin arkistotiedostoihin. Tämä pakkaustapa on JDK:n mukana versiosta 1.1 alkaen. Tämä ratkaisee pitkiin tiedostonimiin sekä isojen ja pienten kirjainten erotteluun liittyvät ongelmat. Se ei kuitenkaan ratkaise epätäydellisen Unicoden tukiongelmaa. (Sun Microsystems Inc. 2000c.)

Javan eri pääversioilla, kuten esimerkiksi 1.0, 1.1, 1.2 ja 1.3, käännettyjen luokkien sekoittaminen toistensa kanssa ei ole suositeltavaa, koska jotkut harvinaiset tilanteet, virhekorjaukset tai vähäiset epäyhteensopivuudet versioiden välillä saattavat aiheuttaa siir-

rettävyysongelmia. Siten ohjelman käyttämät luokat tulisikin kääntää samalla Java-alustan versiolla, jolla itse ohjelmaa aiotaan suorittaa. (Sun Microsystems Inc. 2000c.)

Uusimmilla Javan versioilla kehitettyjen ohjelmien siirrettävyys vaarantuu, kun Javan eri versioiden toteutuksia ei julkisteta samanaikaisesti kaikkiin Javaa tukeviin kohdeympäristöihin. Esimerkiksi Javan version 1.3 toteutus tuli saataville ensimmäisenä Solarikselle ja Windowsille. Sen sijaan HP-UX:ille se ilmestyi useita kuukausia myöhemmin. Osaltaan tähän vaikuttaa se, että Sunin lisäksi monet muutkin yhtiöt kehittelevät Java-toteutuksia ja näiden aikataulut eivät ole yhteneviä. Siten rakennettaessa ohjelmia uusimmilla Javan versioilla, kannattaa ottaa selville onko kyseisen version toteutus saatavilla kohdeympäristössä tai mikä on sen luvattu julkistusajankohta. Kyseessä on kirjasto-ongelma, jota käsiteltiin luvussa 2.2.

Joissakin Java-alustan toteutuksissa on virheitä. Pyrittäessä siirrettävyyteen tulee nämä virheet ohittaa jollakin tavoin. Joissakin tapauksissa ohjelmaan täytyy lisätä eri koodia eri kohdealustoja varten. Korjaus voidaan ottaa käyttöön joko ennakoivasti haarautumalla järjestelmämääritysten avulla, kuten saatavan käyttöjärjestelmän nimen mukaan, tai reagoivasti pistämällä korjaus poikkeuksen käsittelijään. Valinta riippuu korjauksen luonteesta. Jos sillä on vaikutuksia ohjelman rakenteeseen tai pitkäkestoisiin datarakenteisiin siten, että peruuttamattomia muutoksia saattaisi aiheutua ennen ongelman havaitsemista, on ehkä parasta noudattaa ennakoivaa tapaa. Sen sijaan jos ratkaisulla on suhteellisen paikallisia vaikutuksia, reagoiva tapa voi olla helpointa koodata ja ymmärtää. (Sun Microsystems Inc. 2000c.)

Horstmannin ja Cornellin (1997, 10) mukaan perinteisesti on tarvittu suuria ohjelmointiponnistuksia, kun ongelmana on kirjoittaa sovellus, joka näyttäisi hyvältä Windowsissa, Macintoshissa ja kymmenessä erilaisessa Unixissa. Tähän liittyen Javan kirjastot määrittelevät siirrettävät rajapinnat (Gosling 1996). Esimerkiksi ikkunaluokka on määritelty abstraktiksi ja sille on tehty toteutukset Unixille, Windowsille ja Macintoshille. Kuitenkin Horstmann ja Cornell (1997, 10) kirjoittavat Javan versiosta 1.1, että Javan suunnittelijat eivät olleet ainakaan vielä tuolloin ratkaisseet ongelmaa. Kaikki mitä silloin oli olemassa, käsitti kirjaston, joka suurella työllä saattoi antaa juuri ja juuri hyväk-

syttäviä tuloksia eri järjestelmissä. Lisäksi eri alustojen graafisista toteutuksista löytyi usein erilaisia virheitä. Kuitenkin Java 1.1:n graafiset kirjastot ovat paljon parempia kuin Javan aiemmissa versioissa. Kun graafisen käyttöliittymän siirrettävyyden suhteen oli havaittu ongelmia Javan versiossa 1.1 ja sitä aiemmissa versioissa, onkin mielenkiintoista tarkastella tarkemmin graafista käyttöliittymää Javassa ja mitä uutta siihen on tullut Java 1.2:n myötä.

5.4 Graafinen käyttöliittymä Javassa

Seuraavassa tarkastellaan graafisen käyttöliittymän perusteita Javassa. Aluksi esitetään kaksi erilaista tapaa tehdä Java-ohjelmia. Sitten tarkastellaan graafisen käyttöliittymän kahta erilaista ikkunointityyppiä, joiden pohjalle Javan graafinen käyttöliittymä voidaan rakentaa. Lisäksi pohditaan, miten graafinen käyttöliittymä tulisi rakentaa, jotta se olisi siirrettävä.

5.4.1 Sovelmat ja sovellukset

Javalla voidaan kehittää käyttöliittymiä kahdenlaisiin ohjelmiin: sovelmiin ja sovelluksiin. Sovelmat vaativat suorittuakseen selaimen. Selaimessa on sovelmaa varten sovelmaympäristö (AppletContext) (Sun Microsystems Inc. 2000c). Javan versiossa 1.0 sovelmaympäristön Applet-protokollan määritys ei Sun Microsystemsin (2000c) mukaan ollut vielä kovin tarkka, joten eri selaimet kutsuivat sovelman käynnistys- ja lopetusmetodeja eri aikaan. Siten esimerkiksi sovelma, joka toimi normaalisti yhdellä selaimella, saattoi juuttua toisella selaimella. Lisäksi yhdellä selaimella hyvin käyttäytynyt sovelma saattoi viedä paljon resursseja toisella selaimella. Protokollan määritys on kuitenkin tarkentunut Javan 1.1 versiosta lähtien (Sun Microsystems Inc. 2000c).

WWW-sivulla sovelma merkitään käyttämällä upotettua HTML-kielistä (Hyper Text Markup Language) <applet> -merkintää, johon sijoitetaan muun muassa suoritettavan ohjelman nimi. Kun käyttäjä avaa tällaisen WWW-sivun Internetin tai yhteisön sisäisen intranetin kautta, sovelma latautuu automaattisesti palvelimelta ja suorittuu asiakkaan koneessa (Kramer 1996). Koska sovelmat ladataan, ne kannattaa suunnitella pieniksi ja modulaarisiksi, jotta vältettäisiin pitkiä latausaikoja. Sovellukset eivät vaadi selainta suorittuakseen. Niillä ei ole vastaavaa sisäänrakennettua latausmekanismia (Kramer 1996). Kun sovellusta kutsutaan, se suorittuu, mutta suorittuakseen se tarvitsee Java-alustan.

Kramerin (1996) mukaan sovelmissa ja sovelluksissa voidaan käyttää suurimmaksi osaksi yhdenvertaisesti Java-kielen palveluja. Sovelluksilla on kuitenkin suurempi vapaus siinä, että niillä on täydet oikeudet käyttää systeemitason palveluja. Sovelmien sen sijaan täytyy suorittua turvallisuushallitsimen (security manager) kontrollissa (Sun Microsystems Inc. 2000c). Esimerkiksi sovelluksella voi sovelmasta poiketen olla normaalit luku- ja kirjoitusoikeudet millä tahansa levyllä oleviin tiedostoihin. Kun sovelma voi mahdollisesti olla ladattu epäluotettavalta WWW-sivulta, sillä on rajoitetut luku- ja kirjoitusoikeudet tiedostojärjestelmiin lukuunottamatta sen palvelimen tiedostojärjestelmää, josta sovelma on lähtöisin. Tätä rajoitusta voidaan lieventää merkitsemällä sovelma digitaalisella allekirjoituksella. Se mahdollistaa loppukäyttäjän varmistumisen siitä, että sovelma on ladattu muuttumattomana luotettavasta lähteestä. Turvallisuushallitsimille ei ole Sun Microsystemsin (2000c) mukaan olemassa standardia profiilia. Siksi käyttäjä voi määritellä turvallisuushallitsimensa kieltämään sovelmalta minkä tahansa pääsymahdollisuuden systeemitason resursseihin (Sun Microsystems Inc. 2000c; Campione & Walrath 1998, 219). Siten sovelmissa tulisikin varautua käsittelemään kaikki mahdolliset turvallisuuspoikkeukset tilanteeseen soveltuvalla tavalla.

5.4.2 AWT

Abstrakti ikkunointisetti (Abstract Window Toolkit, AWT) on Javan perinteinen graafisen käyttöliittymän kirjoittamiseen tarkoitettu työkalusetti. AWT:n avulla voidaan kirjoittaa graafisia käyttöliittymiä kaikkiin Java-järjestelmiin. AWT:llä rakennettujen käyttöliittymien pitäisi Arnoldin ja Goslingin (1998, 359) mielestä myös suorittua tarkoituksenmukaisella tavalla kaikissa Java-järjestelmissä. AWT esittää käyttöliittymäkomponentteja käyttäen allaolevan alustan käyttöliittymätyyliä. AWT näyttää esimerkiksi Macintoshin painikkeita Macintoshissa, Windowsin painikkeita Windowsissa ja Motifin painikkeita X-ikkunointijärjestelmässä.

Seuraavassa tarkastellaan, kuinka Java AWT voi toimia eri alustoilla. Perusidea on yksinkertainen: joka kerta kun AWT-komponentti luodaan, AWT luo automaattisesti sille vastineen (peer) kohdeympäristön ikkunointijärjestelmässä (Horstmann & Cornell 1997, 451). Esimerkiksi AWT:n painike yhdistetään kohdeympäristön käyttöjärjestelmän painikkeeseen. Kutakin vastinoliota hallitaan Java-rajapinnan kautta. Esimerkiksi kun käytetään java.awt.Component -luokan metodia setBackground(), AWT kutsuu lopulta kohdeympäristön ikkunointijärjestelmän metodia, joka asettaa ikkunalle taustavärin.

AWT-komponenttien ja allaolevan ikkunointijärjestelmän vastineitten yksi yhteen -vastaavuudella on Horstmannin ja Cornellin (1997, 359, 451) mukaan muutamia etuja: Java-ohjelmien käyttöliittymä näyttää tutulta kunkin kohdeympäristön käyttäjille. Piirre mahdollistaa Javan versiosta 1.0 lähtien "leikkaa ja liimaa" -toiminnot tekstikomponenteissa, koska tekstikomponentin vastine tietää, kuinka nämä toiminnot tehdään. Kuitenkin vastinelähestymistavalla on seuraavia ongelmia ja kustannuksia (Horstmann & Cornell 1997, 451—452):

- Vastineet vaativat ikkunan luomisen allaolevaan järjestelmään.
- AWT:n komponenttikokoelma on suppea.
- Sovellukset näyttävät erilaisilta kullakin alustalla.

Tietyissä tapauksissa, kuten X-ikkunointijärjestelmässä, monien kohdejärjestelmän ikkunoiden luominen on todellinen kustannus. Esimerkiksi joka kerta, kun luodaan pa-

neeli komponenttien järjestämiseksi, AWT luo sille vastineen kohdeympäristön ikkunointijärjestelmään. Yleensä paneeleita tarvitaan komponenttien järjestämiseen useita jo yhdessäkin käyttöliittymän näytössä, joten tässä kulutetaan resursseja.

AWT:n komponenttikokoelma on suppea, koska se voi käyttää vain niitä komponentteja, jotka ovat saatavilla kaikissa kohdeympäristöissä. Esimerkiksi Windows 95:ssa on käyttökelpoisia yleisiä kontrolleja. Kun niitä ei ole muilla alustoilla, vastinelähestymistapaa ei voida käyttää niiden toiminnallisuuden tuomiseksi Javaan.

Sovelluksella ei voi AWT:ta käytettäessä olla yhtenäistä, alustasta riippumatonta käyttöliittymätyyliä. Kullakin alustalla käytettävän alustakohtaisen tyylin kanssa on ilmennyt ongelmia. Johnsonin (1999, 90) mukaan vastinelähestymistapa aiheuttaa eri alustoilla vähäisiä mutta ongelmallisia vaihteluita käyttöliittymän käyttäytymisessä ja tyylissä. Sood (1998, 111) mainitsee vastineitten käytön erääksi ongelmaksi alustakohtaiset viat. Näitä ongelmia saattaa aiheuttaa vastinelähestymistavan alustakohtaisen toteutuksen suuri osuus.

5.4.3 Kevyet käyttöliittymäkomponentit

Toinen tapa Java-pohjaisen käyttöliittymän rakentamiseen on käyttää kevyitä komponentteja (lightweight components) (Horstmann & Cornell 1997, 359). Kevyet käyttöliittymäkomponentit eivät käytä alkuperäisiä, allaolevan ikkunointijärjestelmän käyttöliittymäelementtejä vastineina (Sun Microsystems Inc. 1998a, 1—10; Sood 1998, 111; Horstmann & Cornell 1997, 359). Tästä johtuen kevyiden käyttöliittymäkomponenttien ei tarvitse olla yhtä rajoitteisia kuin AWT-komponenttien. Komponenttivalikoima on laajempi, koska komponenttien ei tarvitse olla kaikissa kohdeympäristöissä saatavilla. Kevyiden käyttöliittymäkomponenttien ulkoasua sekä käyttäytymistä voidaan myös muokata ja laajentaa AWT-komponentteja helpommin (Sun Microsystems Inc. 1998a, 1—10).

Kevyet käyttöliittymäkomponentit piirretään Javan omalla piirtografiikalla (Sun Microsystems Inc. 1998a, 1—10). Näiden komponenttien uudelleenpiirtäminen vie vähemmän muistitilaa kuin AWT-komponenttien uudelleenpiirtäminen. Kevyet käyttöliittymäkomponentit hyödyntävät kaksoispuskurointitekniikkaa, jossa ylläpidetään taustavarastoa kuvaoliossa ja uudelleenpiirtämiset tehdään kopioimalla pikselit taustavarastosta. AWT-komponentit ovat kuitenkin hieman kevyitä käyttöliittymäkomponentteja nopeampia.

Sun ja sen yhteistyökumppanit ovat rakentaneet kevyiden käyttöliittymäkomponenttien kokoelman, jota kutsutaan Swingiksi. Swing-komponentteja on kehitetty JFC-luokkakirjastoon (Java Foundation Classes), joka oli saatavilla Javan versioon 1.1.x erillisenä pakettina. Javan versiosta 1.2 lähtien Swing-komponentit ovat olleet sen osana. Käyttämällä Swing-komponentteja voidaan ohjelmalle antaa yhdenmukainen käyttöliittymätyyli erilaisilla alustoilla (Sood 1998, 111; Horstmann & Cornell 1997, 359). Lisäksi voidaan käyttää myös kohdealustan omaa käyttöliittymätyyliä (Sun Microsystems Inc. 1998a, 1—7).

Kevyitä komponentteja ei suositella käytettäväksi yhdessä AWT-komponenttien eikä muidenkaan AWT:n kaltaisten vastineita käyttävien komponenttien kanssa (Campione & Walrath 1999; Sun Microsystems Inc. 1998a, 2—5). Yhteiskäytöstä tulee ongelmia, kun AWT-komponentit ja kevyet komponentit menevät limittäin, jolloin AWT-komponentit piirretään aina päällimmäisiksi.

5.4.4 Siirrettävyyden huomioiminen Javan graafisessa käyttöliittymässä

Jotta käyttöliittymä toimisi useissa kohdeympäristöissä, täytyy harkita uudestaan, kuinka komponentit sijoitellaan graafisessa käyttöliittymässä. Sen sijaan, että pohdittaisiin, mihin tietty komponentti tarkalleen ottaen pitäisi näytöllä sijoittaa, tulisi pohtia, miten se pitäisi sijoittaa suhteessa muihin komponentteihin. Komponenttien absoluuttinen sijoittelu ei toimi siirrettävästi, koska muun muassa painikkeen koko ei ole vakio erilai-

sissa järjestelmissä (Arnold & Gosling 1998, 359; Campione & Walrath 1998, 504). Kun absoluuttisella sijoittelulla rakennettua käyttöliittymää käytetään toisessa järjestelmässä kuin missä se alunperin suunniteltiin, käyttöliittymä rikkoutuu. Tämä ilmenee muun muassa siten, että jotkut painikkeet menevät päällekkäin ja toisten painikkeiden välille tulee ylimääräisiä välejä. Lisäksi kun näyttöjä on monenkokoisia, suhteellinen komponenttien sijoittelu on tällöin absoluuttista komponenttien sijoittelua parempi vaihtoehto (Sun Microsystems Inc. 1997, 2000c).

Vaikka Java-ohjelman käyttöliittymä voidaan rakentaa käyttämällä absoluuttista sijoittelua, sitä ei pidä tehdä siirrettäviin ohjelmiin pyrittäessä (ks. Arnold & Gosling 1998, 359; Campione & Walrath 1998, 504). Sen sijaan kannattaa käyttää sijoitteluhallitsimia (layout manager). Sijoitteluhallitsimet käyttävät suhteellista komponenttien sijoittelua: komponenttien paikka ja koko asetetaan suhteessa muihin komponentteihin. Kun esimerkiksi asetetaan komponentti näyttökehyksen sisään, kehyksen sijoitteluhallitsin määrää, mihin komponentti sijoitetaan. Useimmat sijoitteluhallitsimet ovat melko yksinkertaisia ja joustamattomia, kuten esimerkiksi FlowLayoutManager, joka lisää komponentteja samalle riville niin kauan kuin niitä sille mahtuu. Mutta on myös olemassa eräs erittäin joustava ja monimutkainen sijoitteluhallitsin nimeltään GridBagLayoutManager.

Kirjasintyyppien koko ja saatavuus vaihtelee näytöstä toiseen jopa samalla laitealustalla asennuksesta riippuen. Tekstien esityskokojen kiinnittäminen ohjelmakoodiin ei ole siirrettävää. Tekstielementeille pitäisi sallia niiden luonnollinen koko sijoittelussa. Asetettaessa ohjelmaan Javassa valmiiksi määritellyistä kirjasintyypeistä poikkeavaa kirjasintyyppiä tulee toteuttaa myös toipuminen siltä varalta, että kirjasintyyppiä ei löydy. Toteutettaessa valikkoa kirjasintyypeille tulee kirjasintyyppien nimet hakea java.awt.Toolkit.getFontList() -metodin avulla. (Sun Microsystems Inc. 2000c.)

Näytön koko ja saatavilla olevien värien määrä saattavat vaihdella eri kohdealustoilla tai käyttäjillä. Tämä voi tehdä ulkoasun epäselväksi tai painikkeet voivat hävitä, jos ne eivät mahdu näytölle. Siten ikkunan oletuskoko tai kirjasintyypin koko saattaa olla tarpeen mukauttaa näytön erottelukyvyn mukaan. Ohjelman värit voidaan harmonisoida käyttäjän kohdealustan mukaan Javan versiosta 1.1 lähtien käyttämällä ja-

va.awt.SystemColor -luokkaa. Oman värimallin käyttämisessä tulee olla varovainen, sillä näytöt eroavat suuresti tietylle RGB-arvolle (Red Green Blue) esitetyssä värissä. Ohjelman ulkoasu tulee paljon siirrettävämmäksi, jos käytetään java.awt.Color -luokan valmiiksi nimettyjä värikenttiä numeeristen värien sijaan. (Sun Microsystems Inc. 2000c.)

Javan ydin-API:n toteutukset sisältävät luokkia ja paketteja, jotka eivät kuulu dokumentoituun rajapintaan. Siirrettävissä ohjelmissa ei tule olla riippuvuuksia näihin dokumentoimattomiin luokkiin. Dokumentoimattomissa luokissa saattaa olla eroavaisuuksia eri Java-alustan toteutuksissa siten, että luokat saattavat puuttua kokonaan tai sitten niiden toiminnallisuudessa on vähäisiä, mutta vaarallisia eroja. Esimerkiksi AWT-komponenttien java.awt.peer -paketissa määritellyt vastinerajapinnat on tarkoitettu vain AWT-toteuttajien käyttöön ja vastineluokkien kanssa käytettävä vuorovaikutusprotokolla on alustakohtainen, joten siirrettävän ohjelman tulee vain käyttää AWT:tä eikä toteuttaa sitä. (Sun Microsystems Inc. 2000c.)

AWT:n metodeille java.awt.Component.paint() ja java.awt.Component.update() annetaan grafiikkaolio parametrina. Tätä oliota ei pitäisi säilyttää, koska se saattaa olla lyhytaikainen olio, joka on olemassa vain piirtämisen ajan. AWT:n toteutus voi tuhota grafiikkaolion sen jälkeen kun piirtometodista palataan. (Sun Microsystems Inc. 2000c.)

Javan ydin-API:ssa on tiettyjä metodeja, joita täytyy kutsua tai toteuttaa tietyn mallin mukaan. Jos näitä malleja ei noudateta, on mahdollista kirjoittaa ohjelma, joka on syntaktisesti oikea, mutta joka ei kuitenkaan ole siirrettävä. Esimerkiksi JavaBeans-komponenttiprotokollaa ja -malleja täytyy noudattaa, jotta saadaan tuotettua siirrettävä Java-komponentti. Toinen esimerkki liittyy tapahtumankäsittelyyn, sillä Javan versiosta 1.1 lähtien tapahtumankäsittelymalli on erilainen kuin sitä aiemmissa versioissa. Ohjelmat, jotka on kirjoitettu Java-version 1.02 tapahtumankäsittelymallilla toimivat Javan 1.1 tai sitä myöhemmällä versiolla, mutta kahden tapahtumankäsittelymallin sekoittaminen yhdessä ohjelmassa ei toimi. Siten ohjelman tulee noudattaa vain yhtä tapahtumankäsittelymallia. (Sun Microsystems Inc. 2000c.)

Kytkettävän käyttöliittymätyylin (Pluggable Look and Feel, PLAF) arkkitehtuuri, joka on rakennettu Javan version 1.2 Swing-luokkiin ja Javan version 1.1 JFC-laajennukseen, mahdollistaa eri käyttöliittymätyylien yhtenäisen käyttöönoton ikkunoissa, dialogeissa ja muissa graafisissa käyttöliittymäkomponenteissa. Kaikki käyttöliittymätyylit eivät kuitenkaan ole saatavilla kaikilla kohdealustoilla. Jotkut käyttöliittymätyylit saattavat olla tuettuja vain siinä käyttöjärjestelmässä, jonka ikkunointijärjestelmän käyttämää käyttöliittymätyyliä kyseinen käyttöliittymätyyli jäljittelee. Siirrettävissä ohjelmissa tulee varmistaa ennen tietyn käyttöliittymätyylin käyttämistä, että kyseinen käyttöliittymätyyli on sekä tuettu että saatavilla kohdeympäristössä. Java-käyttöliittymätyyli on saatavilla jokaisella Java-alustan toteuttavalla kohdealustalla, siten se on myös siirrettävin käyttöliittymätyyli. (Sun Microsystems Inc. 2000c.)

5.5 Kansainvälistäminen Javassa

Java tarjoaa useita työkaluja ohjelman kansainvälistämiseen. Yksi peruselementti on mukana itse kielessä: Javan merkkijonot ovat Unicode-muotoisia (Arnold & Gosling 1998, 335). Unicode Consortiumin (1998) mukaan Unicode-standardi määrittää maailman nykyisten pääkielten merkkikoodit. Lisäksi Java tarjoaa kansainvälistämiseen arkkitehtuurin, jonka avulla voidaan esittää viestejä, numeroita, päivämääriä ja valuuttoja eri maiden paikallisessa muodossa (Davis 1998).

5.5.1 Perusmekanismit

Esitettävät merkkijonot täytyy erottaa muusta ohjelmakoodista, jotta ne voidaan muuttaa helpommin toiselle kielelle. Javassa tämä voidaan tehdä resurssinippujen (ResourceBundle) avulla (Davis 1998). Nämä tarjoavat yleisen mekanismin, joka mahdollistaa merkkijonojen ja muiden kansainvälistettävien resurssien, kuten esimerkiksi kuvakkeiden, käytön paikallisuuden mukaan.

Paikallisuuden määrittämiseksi Javassa käytetään kielestä ISO-639 standardin mukaisia koodeja ja maasta ISO-3166 standardin mukaisia koodeja (Sun Microsystems Inc. 1998a). Esimerkiksi englannin kielestä käytetään lyhennettä en ja ranskasta vastaavasti fr. Maakohtaiset esimerkkikoodit ovat US Yhdysvalloille ja FR Ranskalle. Lisäksi paikallisuudessa voi olla mukana myös yksi ylimääräinen muuttuja, johon voidaan määritellä esimerkiksi kohdealusta. Tätä ylimääräistä muuttujaa on käytetty määrittelemään muun muassa euro-valuutan tuki. Käyttöjärjestelmissä käytetään myös paikallisuutta ja sama paikallisuus voidaan ottaa käyttöön myös Java-ohjelmassa. Java-ohjelmalle voidaan määritellä myös oma paikallisuus.

Kansainvälistettävät resurssit nimetään yksilöivällä avaimella, joka on merkkijono. Resurssiniput tarjoavat avaimen ja paikallisuuden yhdistämisen paikallistettuihin arvoihin (Davis 1998). Ne tarjoavat myös periytyvyyden paikallisissa resursseissa, mikä mahdollistaa toisteisuuden minimoinnin paikallisuuksien välillä. Kun ohjelmaan tehdään uutta paikallisuutta, tulee vain tarvittavat resurssit muuttaa kohdepaikallisuuteen. Jos esimerkiksi oletuspaikallisuutena on englannin kieli ja Iso-Britannia, voidaan paikallisuuteen saksan kieli ja Saksa paikallistettaessa käyttää samoja kuvakkeita ja kääntää vain tekstit uudelleen. Kun ohjelmaa käytetään tämän uuden paikallisuuden mukaan, ohjelma näyttää paikallistettuja saksan kielisiä tekstejä ja oletuspaikallisuuden mukaisia kuvakkeita.

Mahdollisia resurssinippuja ovat ominaisuusresurssinippu ja listaresurssinippu (Davis 1998). Lisäksi voidaan rakentaa myös oma resurssinippu. Ominaisuusresurssinippua käytettäessä avain ja arvo -parit sijoitetaan resurssitiedostoon. Listaresurssinippua hyödynnettäessä resurssit sijoitetaan sen sijaan omiin luokkiinsa (Sun Microsystems Inc. 1999a). Tällöin voidaan merkkijonojen lisäksi palauttaa myös olioita (Davis 1998). Oma resurssinippu voidaan saada tekemällä uusi luokka, joka perii resurssinipun. Jos resurssinipusta tulee liian laaja, se voidaan jakaa pienempiin resurssinippuihin.

Javalla voidaan kansainvälistää numeroiden, valuuttojen, päivämäärien ja kellonaikojen esitysmuodot. Numeromuodot (NumberFormat) käsittelevät yleisiä numeroita ja valuuttoja. Päivämäärämuodot (DateFormat) kattavat sekä päivät että kellonajat. Ohjelman

kansainvälistämiseksi korvataan näiden implisiittiset muotoilut eksplisiittisesti määritellyllä muotoilumallilla ja sijoitetaan tämä malli resurssinippuun. (Davis 1998.)

Esitettävien merkkijonojen yhdistely voidaan tehdä Javassa käyttämällä viestimuotoilua (MessageFormat), joka mahdollistaa paikallistettaessa muuttujatiedon sijoittamisen tarkoituksenmukaisesti. Perusmerkkijonosta tehdään mallimerkkijono, johon sijoitetaan muuttujat oikeille paikoilleen käyttäen merkintätapaa {0}. Edellisessä kaarisulkujen sisällä oleva numero tarkoittaa muuttujan järjestystä muuttujataulukossa, jonka alkiot voidaan myös kansainvälistää. Lisäksi tarvittaessa muuttujan muoto voidaan määritellä tarkemmin mallimerkkijonossa. Tämä voidaan tehdä lisäämällä avainsanoja tai muotoilumalleja numeron perään. (Davis 1998.)

Java tarjoaa tukea laajojen merkistöjen syöttöön. Sun Microsystems Inc. (1998b) määrittelee syöttömetodit (input method) ohjelmistokomponenteiksi, jotka tulkitsevat käyttäjän suorittamia operaatioita, kuten esimerkiksi kirjainten kirjoittamisen, muodostaakseen tekstuaalisen syötteen sovelluksille. Siten syöttömetodi vastaa luvussa 3.6 mainittua Davisin (1998) syöttömetodikonetta.

Java 1.1:ssä syöttötuki tarjotaan AWT:n tekstikenttä- ja tekstialuekomponenttien avulla (Sun Microsystems Inc. 1998c). Näitä käytettäessä syöttötuen toiminnallisuus riippuu täysin kohdealustan tai selaimen toteutuksesta (Davis 1998). Java 1.2:een on lisätty syöttömetodikehys (Sun Microsystems Inc. 1998b). Se sisältää luokkia ja rajapintoja tekstiä muokkaaville komponenteille tekstin saamiseksi syöttömetodien kautta. Java 1.2 ja Swing-komponentit tarjoavat tuen "kohdalla" -syöttötavalle syöttömetodien kehyksen avulla (Sun Microsystems Inc. 1998c). Kun käyttäjä syöttää tekstiä tämän syöttötavan mukaisesti, syöttömetodit sijoittavat tekstin suoraan tekstikomponenttiin. Sovelluskehittäjän ei tarvitse koordinoida syöttömetodien ja tekstikomponenttien välisiä toimintoja, vaan nämä hoidetaan automaattisesti, kun syöttötapa on asetettu mahdolliseksi tekstikomponentille. Kehittäjien pitäisi huolehtia kuitenkin siitä, että ohjelmassa kutsutaan InputContext-luokan endComposition()-metodia silloin, kun koko teksti täytyy hyväksyä. Tämä täytyy tehdä esimerkiksi ennen dokumentin tallentamista tai tulostamista. Java 1.2 käyttää kohdealustan syöttömetodeja, koska se ei sisällä omaa syöttömetodiko-

netta (Sun Microsystems Inc. 1998c). Siten Java 1.2:n syöttötuki riippuu Java-alustan toteutusten laadusta kohdealustoilla. Java 1.3.een on tullut syöttömetodikoneelle palvelurajapinta (Service Provider Interface, SPI), joka mahdollistaa syöttömetodikoneiden kehittämisen Javalla (Sun Microsystems Inc. 2000b). Lisäksi Java 1.3 tarjoaa tuen "kohdan päällä" -syöttötavalle (Sun Microsystems Inc. 2000b).

Javalla voitaisiin Davisin (1998) mukaan rakentaa myös monikielinen (multilingual) ohjelma, joka mahdollistaisi monen paikallisuuden samanaikaisen käytön. Tämä on lähellä Nielsenin (1993) esittämää monipaikallistamista. Davis (1998) erottaa monikielisyydelle kaksi alikäsitettä: monikielinen data ja monikielinen käyttöliittymä. Monikielinen data tarkoittaa sitä, että käyttäjät voivat syöttää dataa tai asettaa datalle muodon monien paikallisuuksien mukaan. Esimerkkinä tästä ovat eri solujen erilaiset muotoilut taulukkolaskentaohjelmassa. Monikielinen käyttöliittymä puolestaan tarkoittaa sitä, että käyttäjät voivat vaihtaa ohjelmassa näytettävää paikallisuutta ajonaikana. Paikallisuus voidaan Davisin (1998) mukaan asettaa eksplisiittisesti monissa kansainvälistämiseen liittyvissä luokissa. Näitä ovat esimerkiksi muotoilu- ja lajitteluluokat. Sama voidaan tehdä myös Component-luokassa.

Mikäli ohjelmaan rakennetaan monikielinen käyttöliittymä, Davis (1998) esittää kaksi vaihtoehtoista tapaa tukea sitä ohjelmassa. Ensimmäisessä tavassa käyttäjän valitessa erilaisen käyttöliittymäpaikallisuuden esimerkiksi valikosta tulisi oletuspaikallisuus asettaa uudelleen ja sitten kutsua käyttöliittymän rakentavaa koodia rakentamaan käyttöliittymä kokonaan uudelleen. Toinen vaihtoehto olisi tuottaa koodia, joka kävisi käyttöliittymän elementtikohtaisesti läpi korvaten kunkin elementeistä uusilla resursseilla.

5.5.2 Merkkijonojen vertailu, sanavälit ja merkistöt

Tehtäessä kansainvälistämistä Javalla on tärkeää huomioida muutamia erityisiä seikkoja, jotta kansainvälistäminen saataisiin rakennettua toimivaksi. Seuraavaksi käsiteltäviä asioita voitaisiin Javalla tehdä kansainvälistämisen näkökulmasta väärin.

Javassa merkkijonoluokan (String) vertailuoperaatio (compareTo()) tekee Unicode-merkkien binäärisen vertailun kahden merkkijonon sisällä (Campione ym. 1998). Kuitenkaan tähän binääriseen vertailuun ei voida luottaa useimmissa kielissä, koska Unicode-arvot eivät vastaa merkkien suhteellista järjestystä (Campione ym. 1998). Myös Davis (1998) pitää merkkijonoluokan vertailuoperaatioita vääränä tapana merkkijonojen vertailuun kansainvälistettävissä ohjelmissa. Campione ym. (1998) ja Davis (1998) painottavatkin, että kansainvälistettävässä ohjelmassa merkkijonojen vertailuun pitää käyttää lajitteluluokkaa (Collator). Vertailua tarvitaan esimerkiksi järjestettäessä käyttäjälle esitettävää listaa aakkosjärjestykseen.

Jos merkkijonoja vertaillaan usein, tulee käyttää lajitteluavainluokkaa (CollationKey). Tämä esiprosessoi merkkijonon ottaen huomioon kansainvälistämisen ja muuntaa sen sisäiseen muotoon, jota voidaan vertailla yksinkertaisella binäärisellä vertailulla. Tämä nopeuttaa merkkijonojen vertailua. (Davis 1998.)

Koodissa ei pidä tehdä oletusta, että kaikki merkit kuuluvat vain ASCII-merkistöön (Davis 1998). Yksittäisten merkkien vertailussa tulee käyttää Unicode-standardia noudattavia metodeja (Campione ym. 1998). Näitä ovat esimerkiksi merkkiluokan (Character) valmiit metodit.

Joissakin kielissä sanavälejä ei määritellä vain tyhjien välien avulla. Merkkien rajatkaan eivät aina ole yksikäsitteisiä, sillä jonkun maan paikallisille käyttäjille yksi merkki saattaa käsittää useamman kuin yhden Unicode-merkin. Sana-, merkki- ja lauserajojen etsimiseen voidaan Javassa käyttää erilaisia väli-iteraattoreita (BreakIterator). (Davis 1998.)

Rakennettaessa pelkkiä Java-ohjelmia, jotka käyttävät Unicode-merkistöä, ei tarvitse välittää muista maailmassa käytetyistä tuhansista erilaisista merkistöistä. Jos kuitenkin käsitellään muuta dataa, tarvitaan muunnoksia Unicoden ja toisten merkistöjen välillä. Javan sovellusrajapinta on Davisin mukaan (1998) melko rajoittunut merkkikoodimuunnosten tekemisessä. Uudelleenkoodaaminen voidaan määritellä virtaan, kuten syöttövirtaan (InputStreamReader) tai tulostusvirtaan (OutputStreamWriter). Eräs tapa

on määritellä uudelleenkoodaaminen merkkijonoluokkaan (String), kun merkkijonoa muodostetaan tavukooditaulukosta tai kun käytetään merkkijonoluokan getBytes() -metodia merkkijonon muuntamisessa tavukoodiksi.

5.5.3 Rajoituksia

Davisin (1998) mielestä Java 1.1 tarjoaa hyvän kansainvälistämistuen eurooppalaisille kielille ja maille. Kuitenkin Kaukoitää varten Java 1.1:ssä on vain minimaalinen kansainvälistämistuki. Lähi-itää ja Kaakkois-Aasiaa varten, johon luetaan mukaan Intia, Java 1.1 ei anna riittävää kansainvälistämistukea. Lisäksi Java 1.1:n kirjasintyyppien tuki on hyvin heikko jopa englanninkielelle.

Todelliset kohdealustasta riippumattomat kirjasintyypit ovat mahdollisia vain, kun kirjasintyypit ovat sisäänrakennettuja eli ne tarjotaan Java-alustan mukana (Sun Microsystems Inc. 1999b). Toinen mahdollinen tapa kohdealustasta riippumattomiin kirjasintyyppeihin olisi muodostaa ne matemaattisesti tai ohjelmallisesti (Sun Microsystems Inc. 1999b). Kuitenkin Java-alustan mukana tulevia kirjasintyyppejä on vähän. Lisäksi Java ei tue matemaattisesti muodostettuja kirjasintyyppejä (Sun Microsystems Inc. 1999b). Siten Java-ohjelmissa käytetään yleensä kohdealustojen kirjasintyyppejä. Tällöin eri maiden erikoismerkkejä voidaan näyttää, jos kohdejärjestelmästä löytyy kirjasintyypin tuki niiden näyttämiseksi. Kaikki kohdealustat eivät kuitenkaan osaa esittää Unicode-merkkejä (Sun Microsystems Inc. 2000c). Sun Microsystems (2000c) suosittaakin, että käyttöliittymän oletusteksteinä käytettäisiin vain ASCII-merkistöön kuuluvia merkkejä. Muiden merkkien käyttö voitaisiin sallia paikallistetuissa resurssinipuissa ja käyttäjältä saatavassa tekstissä.

Java 1.1:n AWT-kirjastossa olevat tekstialue- ja tekstikenttäkomponentit käyttävät kohdejärjestelmän vastineita tekstin muokkaamiseen. Jos kohdejärjestelmä ei tue Unicodea, tehdään muunnos johonkin toiseen kohdealustan tukemaan merkistöön (Davis 1998). Tällaisessa tapauksessa Unicoden monet symbolit ja välimerkit hylätään. Lisäksi jotkut

toteutukset tekevät muunnoksen vielä takaisin Unicodeen (Davis 1998). Käytettäessä AWT-komponenttien sijaan Swing-komponentteja ei vastaavia ongelmia pitäisi ilmetä, koska Swing-komponenteissa ei käytetä vastineita.

Java 1.2:ssa ei ole valmista tukea monille kalentereille (Davis & Shih 1998). Siinä on valmis tuki vain gregoriaaniselle kalenterille. Merkkikoodikonversiota varten Javan sovellusrajapinnassa on melko vähän palveluita (Davis 1998).

Syöttömetodien tuen kehittyminen Javan versioissa 1.2 ja 1.3 on osaltaan parantanut mahdollisuuksia kansainvälistää Java-ohjelmia Kaukoitään. Tekstin asetteluluokka (TextLayout), joka Javassa on versiosta 1.2 lähtien, tukee Davisin ym. (1998) mukaan kaksisuuntaisia kirjoitusjärjestelmiä, kuten arabiaa ja hebreaa. Tämä parantaa mahdollisuuksia kansainvälistää ohjelmia Lähi-itään. Sen sijaan ainakaan Java 1.2:ssa ei vielä ole riittävää tukea intialaisille kirjoitusjärjestelmille ja thaille eikä myöskään pystysuoralle merkkien näyttämiselle (Davis ym. 1998).

Tuettujen paikallisuuksien määrä saattaa vaihdella Java-alustan eri toteutuksien välillä (Sun Microsystems Inc. 1999c). Ei ole määritelty, mitä paikallisuuksia kaikkien Java-alustan toteutuksen tulisi tukea. Siten suunniteltaessa kansainvälistämistä johonkin tiettyyn maahan ja kieleen tulee varmistaa, tukeeko kohdeympäristön Java-alustan toteutus kyseistä paikallisuutta.

5.6 Johtopäätökset Javan siirrettävyydestä

Seuraavassa tehdään yhteenveto Javan siirrettävyydestä tämän työn löydösten pohjalta. Ensin tehdyistä havainnoista kootaan Javan siirrettävyyteen keskeisimmin liittyneet ongelmat ja keinot niiden eliminoimiseksi tai niihin varautumiseksi. Tämän jälkeen Javan siirrettävyydestä esitetään arvio yleisen siirrettävyyden viitekehyksen avulla.

5.6.1 Keskeisimmät siirrettävyysongelmat ja niiden ratkaisukeinot

Javassa on siirrettävyyteen negatiivisesti vaikuttavia ongelmia. Jos näihin ongelmiin ei varauduta jo ennalta Java-ohjelman kehittämisprosessissa, voi niiden ilmeneminen aiheuttaa paljonkin siirtämistyötä. Seuraavassa taulukossa 4 on nostettu esiin tässä tutkimuksessa havaitut Javan keskeisimmät siirrettävyyttä haittaavat ongelmat.

TAULUKKO 4. Javan keskeisimmät siirrettävyyteen liittyvät ongelmat.

Vakavuus	Siirrettävyysongelma
***	Joidenkin Javan palveluiden käyttäminen vaarantaa ohjelman siirrettävyyden.
***	Jotkut Javan palvelut toimivat eri tavoin eri kohdeympäristöissä.
**	Uudet Javan versioiden toteutukset julkistetaan eri aikaan eri kohdeympäristöissä.
**	Jotkut Javan palvelut eivät ole yhtä kattavia kaikissa Javan kohdealustatoteutuksissa.
**	Java saattaa tarjota enemmän palveluja kuin kaikissa kohdeympäristöissä voidaan toteuttaa.
**	Kansainvälistämistuki on vasta kehittymässä.
*	Javan kohdealustatoteutuksissa on virheitä.
*	Turvallisuushallitsimen standardiprofiilia ei ole määritelty.
*	Kaikki käyttöliittymätyylit eivät ole saatavilla kaikissa kohdeympäristöissä.

Taulukossa 4 on arvioitu ongelmien vakavuusastetta siirrettävyyden kannalta. Mitä enemmän ongelmalla on "*" -merkkejä, sitä merkittävämpi ongelma on ohjelman siirrettävyyden kannalta ja sitä enemmän ongelma voi ilmetessään aiheuttaa siirtämistyötä. Siirrettävyysongelma ja siirtämistyön määrä voi olla huomattava (***), kohtalainen (**) tai vähäinen (*).

Ohjelman siirrettävyys voi vaarantua käytettäessä joitakin Javan palveluita. Esimerkiksi paikallisia järjestelmäriippuvia metodeja käytettäessä ohjelma ei ole sellaisenaan siirrettävä. Näiden metodien käyttöä tulisikin välttää siirrettäviin ohjelmiin pyrittäessä. Mikäli

paikallisten järjestelmäriippuvien metodien käyttöä ei voida välttää, kannattaa ne erottaa erilleen muusta ohjelmasta, jotta vain ne voidaan ohjelmoida uudelleen ilman muutoksia koko ohjelmaan. Toinen tämän ongelman ilmentymä ovat toisentyyppiset Javan palvelut, jotka on toteutettu puhtaasti Javalla, mutta joita ei kuitenkaan tulisi käyttää siirrettävyyteen pyrittäessä. Esimerkiksi käyttöliittymiä voidaan rakentaa pikseliperustaisesti asettamalla elementeille kiinteät koot, mutta siirrettävyyden kannalta suositeltavampaa on kuitenkin käyttää sijoitteluhallitsimia. Samoin kansainvälistettäessä täytyy monien Java-palveluiden sijaan käyttää toisia Java-palveluita, jotka huomioivat kansainvälistämisen tarpeet. Kansainvälistäminen kannattaa huomioida koko ohjelman suunnittelussa ja toteutuksessa, sillä jo valmiin sovelluksen kansainvälistäminen vaatii paljon työtä. Huomioimalla ohjelman kansainvälistäminen ajoissa voidaan säästää aikaa ja rahaa.

Ohjelman siirrettävyyden kannalta merkittävä ongelma on myös se, että jotkut Javan palvelut tai yksittäisen palvelun osa, kuten esimerkiksi säikeiden ajastus, toimii eri tavoin eri kohdeympäristöissä. Tällaiset palvelut ovat salakavalia, sillä yhdessä ympäristössä niiden varaan on saatettu rakentaa valmiiksi keskeisiä toimintoja, mutta kun ohjelmaa testataankin toisessa ympäristössä, se toimiikin yllättäen aivan erilailla. Tällaisten palvelujen käyttö tulisi olla hyvin suunniteltua ja varovaista.

Java-ohjelman siirrettävyys vaarantuu, jos kehityksessä käytetylle uudelle Javan versiolle ei ole saatavilla toteutusta kohdeympäristössä. Mikäli oma toimitusaikataulu on kireä eikä tukea ole vähään aikaan tulossa, voidaan sovellus joutua mukauttamaan jonkun kohdeympäristöstä löytyvän vanhemman Java-version mukaiseksi. Tämä saattaa vaatia kohtalaisen paljon siirtämistyötä etenkin silloin, jos mukauttaminen joudutaan tekemään Javan pääversioiden välillä. Siten rakennettaessa ohjelmia uusimmilla Javan versioilla kannattaa ottaa selville, onko kyseisen version toteutus saatavilla kohdeympäristössä tai mikä on sen luvattu julkistusajankohta.

Eräs ilmeinen ohjelman siirrettävyyteen vaikuttava ongelma on, että jotkut Javan palvelut eivät ole yhtä kattavia kaikissa Javan kohdealustatoteutuksissa. Esimerkiksi ei ole määritelty, mitä paikallisuuksia Javan kohdealustatoteutusten tulee tukea. Tällaisten

palvelujen saatavuus kannattaakin varmistaa kohdeympäristön Java-alustan toteuttavalta taholta.

Ohjelman siirrettävyyttä saattaa haitata se, että Java tarjoaa enemmän palveluja kuin kaikissa kohdeympäristöissä voidaan toteuttaa. Esimerkiksi syöttö- ja tulostusvirtojen puuttuminen kohdeympäristöstä saattaa aiheuttaa siirrettävyysongelmia. Eräs keino varautua tällaisiin ongelmiin on käsitellä metodikutsujen mahdollisesti aiheuttamat poikkeukset asianmukaisella tavalla.

Käyttäjätasolla Java-ohjelman eräs siirrettävyysongelma on kansainvälistämistuen keskeneräisyys. Java-ohjelmaa ei voida tällä hetkellä kansainvälistää kaikkiin maailman kulttuureihin tai määriteltyihin paikallisuuksiin. Tämä osa-alue on kuitenkin kehittymässä kaiken aikaa. Tällä hetkellä kannattaa käyttää jo olemassa olevia kansainvälistämismekanismeja.

Ohjelman siirrettävyyden vaarantavat kohdealustatoteutusten virheet. Tästä esimerkkinä mainittiin edellä AWT-kirjaston alustakohtaisissa toteutuksissa olevat virheet. Siirrettävyyteen tähdättäessä hyödyllisiä strategioita virheiden korjaukseen ovat ennakoiminen ja reagoiminen. Ennakointi voidaan tehdä haarautumalla esimerkiksi järjestelmämääritysten avulla, kuten käyttöjärjestelmän nimen mukaan. Reagointi mahdollistuu sijoittamalla korjaus poikkeuksen käsittelijään. Kohdealustatoteutuksien virheitä ei kuitenkaan löytynyt kovin paljon tämän työn yhteydessä. Tämä havainto ei silti anna todellista kuvaa tilanteesta, koska työn pääpaino ei ole ollut kohdealustatoteutuksien virheiden kartoituksen tasolla. Näitä virheitä löytyisi todennäköisesti enemmän, mikäli niistä tehtäisiin järjestelmällinen kohdealustakohtainen kartoitus tai jos Javan siirrettävyyttä testattaisiin kattavan konstruktion avulla useissa kohdeympäristöissä.

Sovelmien siirrettävyyteen vaikuttaa ongelmallisesti turvallisuushallitsimen standardiprofiilin määrittelemättömyys. Siten jopa selainten oletusprofiilit voivat olla hyvin erilaisia. Tähän ongelmaan kannattaa varautua käsittelemällä sovelmassa kaikki mahdollisesti aiheutuvat turvallisuuspoikkeukset tarkoituksenmukaisella tavalla.

Java mahdollistaa useiden käyttöliittymätyylien käyttämisen rakennettaessa käyttöliittymää kevyistä käyttöliittymäkomponenteista. Kuitenkin siirrettävyyden kannalta ongelmana on, että kaikki käyttöliittymätyylit eivät ole saatavilla kaikissa kohdeympäristöissä. Ainoa käyttöliittymätyyli, joka on saatavilla kaikissa Java-alustan kohdeympäristöissä, on Java-käyttöliittymätyyli. Muita Javan mahdollistamia käyttöliittymätyylejä käytettäessä tulee ajoissa varmistaa, että kyseinen tyyli on käytettävissä kohdeympäristössä.

5.6.2 Arvio Javan siirrettävyydestä

Tässä kohdassa arvioidaan Javan siirrettävyyttä käyttäen luvussa 2 kehitettyä yleistä siirrettävyyden viitekehystä. Kutakin siirrettävyysympäristön osatekijää kohden nostetaan esille Javasta löytyneet siirrettävyyttä edistävät mekanismit. Kirjallisuudesta löytyneet maininnat Java-ohjelmayksikön siirrettävyysongelmista jaetaan siirrettävyysympäristön osatekijäkohtaisesti sillä perusteella, mihin osatekijään ongelma vaikuttaa suoraan. Kun tietyllä siirrettävyysympäristön tasolla tapahtuu muutos, tästä ilmenevät siirtämisongelmat kuuluvat samalle tasolle, jolla muutos tapahtui. Jos esimerkiksi oheislaitteiden ja -järjestelmien tasolla näyttö vaihtuu, tästä aiheutuvat ongelmat kuuluvat siten oheislaitteisiin ja -järjestelmiin. Tarkastelussa ei huomioida epäsuoria vaikutuksia, joiden johdosta sisemmillä tasoilla, kuten esimerkiksi systeemitason järjestelmissä, suoraan ilmenevät siirrettävyysongelmat vaikuttavat usein myös ulommilla tasoilla, etenkin laajemman inhimillisen ja fyysisen järjestelmän tasolla. Joissakin tapauksissa siirrettävyysongelma johtuu useammalla siirrettävyysympäristön tasolla tapahtuvista muutoksista. Tällöin se on pyritty kohdistamaan sille tasolle, jonka muutoksesta se pääasiallisesti ilmenee. Löydettyjen siirrettävyysongelmien kohdistumisten määriin otetaan kantaa karkeasti.

Lisäksi esitetään yleisarvio Java-ohjelmayksikön siirrettävyyden tasosta kunkin siirrettävyysympäristön osatekijän suhteen. Mitä enemmän siirrettävyysympäristön osatekijän kohdalla on "+"-merkkejä, sitä parempi Java-ohjelmayksikön siirrettävyys on kyseisen

tekijän suhteen. Asteikko on seuraava: erinomainen (++++), hyvä (+++), tyydyttävä (++) ja välttävä (+). Arvio on esitetty taulukossa 5.

TAULUKKO 5. Yhteenveto Javan siirrettävyyttä koskevista havainnoista.

Siirrettävyysympäristön osatekijä	**Siirrettävyyttä edistävä mekanismi**	**Kirjallisuudessa mainitut suorat siirrettävyysongelmat**	**Arvio siirrettävyydestä**
Konearkkitehtuuri	Java-alusta ja sen tiukka määrittely	Ei mainintoja	++++
Systeemitason järjestelmät	Java-alusta ja sen tiukka määrittely	Muutamia mainintoja	+++
Oheislaitteet ja -järjestelmät	Java-alusta ja sen tiukka määrittely Sijoitteluhallitsimet	Joitakin mainintoja	+++
Laajempi inhimillinen ja fyysinen järjestelmä	Kytkettävän käyttöliittymätyylin arkkitehtuuri Kansainvälistämisen tuki	Joitakin mainintoja	++

Konearkkitehtuuritasolla siirrettävyyttä edistävä mekanismi on itse Java-alusta ja sen tiukka määrittely. Siirrettävyyden kannalta Java-alustan Java-ohjelmayksikölle tarjoama abstraktio toimii hyvin konearkkitehtuuritasolla. Tämän työn puitteissa ei havaittu konearkkitehtuuriin kohdistuvia siirrettävyysongelmia. Siten arvio Java-ohjelmayksikön siirrettävyydestä on erinomainen konearkkitehtuurin suhteen.

Systeemitason järjestelmien suhteen tärkein siirrettävyyttä edistävä mekanismi on myös Java-alusta ja sen tiukka määrittely. Java-alusta tarjoaa abstraktion käyttöjärjestelmiin nähden. Kuitenkin systeemitason järjestelmiin havaittiin kohdistuvan suoraan muutamia siirrettävyysongelmia (esim. säikeiden ajastus, paikalliset järjestelmäriippuvat metodit, kohdeympäristön tiedostonimirajoitteet, komentoriviongelmat, syöttö- ja tulostusvirtojen puuttuminen kohdeympäristöstä). Tosin suurimpaan osaan ongelmista löytyi jokin eliminointi- tai hallintakeino (esim. huolellinen säieohjelmointi, paikallisten metodien välttäminen, luokkatiedostojen pakkaaminen, POSIX:in tapojen noudattaminen, poikkeusten käsittely). Arvio Java-ohjelmayksikön siirrettävyydestä systeemitason järjestelmien suhteen on hyvä.

Oheislaitteiden ja -järjestelmien tasolla tärkeimmät siirrettävyyttä edistävät mekanismit ovat Java-alusta ja sen tiukka määrittely sekä erikseen mainittuna Java-kieleen rakennetut sijoitteluhallitsimet. Java-alusta tarjoaa abstraktioita oheislaitteisiin ja -järjestelmiin nähden. Sijoitteluhallitsimet puolestaan tarjoavat mahdollisuuden rakentaa graafisia käyttöliittymiä, jotka huomioivat komponenttien asettelussa erilaisista näytöistä aiheutuvia vaihteluita. Oheislaitteisiin ja -järjestelmiin havaittiin kohdistuvan suoraan joitakin siirrettävyysongelmia (esim. pikseliperustaisuus graafisissa käyttöliittymissä, JDBC-ajurien kiinnittäminen), mutta useimpiin niistä löytyi myös eliminointi- tai hallintakeino (esim. sijoitteluhallitsimet, ajurien konfigurointi). Siten arvio Java-ohjelmayksikön siirrettävyydestä oheislaitteiden ja -järjestelmien suhteen on hyvä.

Laajemmasta inhimillisestä ja fyysisestä järjestelmästä on tässä työssä rajauduttu tarkastelemaan vain yksittäistä käyttäjää. Keskeisimmät siirrettävyyttä edistävät mekanismit käyttäjätasolla ovat kytkettävän käyttöliittymätyylin arkkitehtuuri ja kansainvälistämisen tuki. Kytkettävän käyttöliittymätyylin arkkitehtuuri mahdollistaa eri käyttöliittymätyylien yhtenäisen käyttöönoton ikkunoissa, dialogeissa ja muissa graafisissa käyttöliittymäkomponenteissa. Kaikki käyttöliittymätyylit eivät kuitenkaan ole saatavilla kaikilla kohdealustoilla. Javaan sisäänrakennettu kansainvälistämisen tuki tarjoaa jo useita keinoja kansainvälistämiseen. Kuitenkin kansainvälistämisen tuki on Javassa vielä keskeneräinen. Käyttäjätasolle havaittiin kohdistuvan suoraan joitakin siirrettävyysongelmia (esim. kansainvälistämistä tukemattomat palvelut, tuettujen paikallisuuksien sekä käyttöliittymätyylien vaihtelu eri alustatoteutuksilla, Unicode-merkkien esitysongelmat eri kohdeympäristöissä). Tosin useisiin ongelmiin löytyi eliminointi- tai hallintakeino (esim. kansainvälistämistä tukevat palvelut, ASCII-merkkien käyttäminen oletuksena käyttöliittymässä). Arvio Java-ohjelmayksikön siirrettävyydestä laajemman inhimillisen ja fyysisen tason suhteen on tyydyttävä.

Kaiken kaikkiaan Javaan on rakennettu jokaiselle siirrettävyysympäristön osatekijän tasolle jokin siirrettävyyttä edistävä mekanismi ja edelliset arviot Java-ohjelmayksikön siirrettävyydestä ovat jokaisen siirrettävyysympäristön osatekijän suhteen vähintäänkin tyydyttäviä, joten Javassa on selvästi huomioitu siirrettävyys hyvin laajasti. Tämä tarkas-

telu on kuitenkin rajoittunut käsitteellisteoreettiselle tasolle, joten sen painoarvo on lähinnä suuntaa-antava.

6 YHTEENVETO

Tämän tutkimuksen yhtenä tavoitteena on ollut selvittää siirrettävyyteen liittyvästä tutkimuksesta, millaista käsitteistöä siirrettävyyteen liittyy. Sovellusohjelman siirrettävyyden havaittiin useiden määritelmien perusteella olevan suhteellinen käsite. Havaittiin myös, että sovellusohjelman tai ohjelman komponentin siirrettävyyteen liittyvien tekijöiden ymmärtämiseksi kannattaa pyrkiä tunnistamaan niiden ympäristöä sekä ympäristörajapintoja.

Tutkimuksen toisena tavoitteena oli selvittää, minkälaisia perinteisiä ongelmia ja ratkaisukeinoja siirrettävyyteen liittyy. Perinteisesti sovellusohjelman siirrettävyyden ongelmia on kartoitettu konearkkitehtuurin, systeemitason järjestelmien ja oheislaitteiden yhteydessä. Sen sijaan laajempaan inhimilliseen ja fyysiseen järjestelmään liittyvät siirrettävyysongelmat ovat jääneet siirrettävyyden näkökulmasta vähemmälle huomiolle. Muutamia yleisiä ratkaisukeinoja löydettiin, kuten emulointi, abstrakti kone ja esiprosessori. Lisäksi konearkkitehtuuri- ja käyttöjärjestelmäriippuvuudet voidaan erottaa Sommervillen (1995, 412) esittämän siirrettävyysrajapinnan avulla. Edelleen yksi keskeinen siirrettävyysongelmia vähentävä tekijä on standardointi. Kuitenkin standardeista huolimatta monentyyppisiä siirrettävyysongelmia esiintyy edelleen. Siten siirrettäviin sovelluksiin pyrittäessä sovelluskehittäjien olisikin tärkeää aktiivisesti ja kriittisesti pohtia siirrettävyyskysymyksiä, vaikka he hyödyntäisivätkin sovelluskehityksessä jotakin siirrettävyyttä lupaavaa virallista standardia tai teollisuusstandardia.

Kolmantena selvityksen kohteena olivat graafisen käyttöliittymän relevantit siirrettävyyskysymykset. Graafinen käyttöliittymä on sovellusohjelman osa, joten graafisen käyttöliittymän siirrettävyyteen sisältyvät sovelluksen normaalia siirrettävyyttä koskevat asiat. Keskeistä graafisen käyttöliittymän siirrettävyyttä tarkasteltaessa on inhimillinen ja fyysinen ympäristö, koska käyttöliittymä on tiiviissä vuorovaikutuksessa käyttäjän kanssa. Siten Mooneyn (1995) esittämät kulttuuriseen mukauttamiseen liittyvät ongel-

mat ovat relevantteja. Näihin kuuluvat kansainvälistämisen tarve, käyttöliittymän mukauttaminen kohteena olevan tietokonesysteemin tapoihin tai resursseihin sekä sovelluksen käyttäytymisen mukauttaminen uudelle kokeneelle käyttäjien ryhmälle. Keskeisimpiä graafisen käyttöliittymän siirrettävyyskysymyksiä oheislaitteiden puolella ovat näyttöjen ja osoitinlaitteiden erilaisuus.

Neljäntenä tavoitteena oli selvittää, miten siirrettävyys tulisi huomioida ohjelmistoprosessissa. Suositeltavaa on, että sovelluksen siirrettävyyteen varaudutaan aikaisessa vaiheessa. Mitä aiemmin ohjelmistoprosessissa varaudutaan siirrettävyyteen, sitä paremmin on mahdollista huomioida siirrettävyys erilaisissa valinnoissa, päätöksissä ja ratkaisuissa. Näin siirtämiskustannuksissakin voidaan säästää.

Tutkimuksen viidentenä tavoitteena oli pohtia, miten siirrettävyyttä voitaisiin tutkia yksittäisen ohjelmointikielen osalta. Yksittäisen ohjelmointikielen siirrettävyyttä voidaan tutkia siirrettävyyden yleisen viitekehyksen avulla. Ohjelmointikielen siirrettävyyttä voidaan tarkastella viitekehyksen siirrettävyysympäristön osatekijäkohtaisesti. Kukin siirrettävyysympäristön osatekijä voidaan ottaa tarkasteluun yksi kerrallaan ja vakioida tällöin siirrettävyysympäristön muut osatekijät. Näin ohjelmointikielen siirrettävyyden tasoa voidaan paremmin arvioida tietyn siirrettävyysympäristön osatekijän suhteen. Lisäksi viitekehyksen avulla ohjelmointikielen siirrettävyydestä voidaan muodostaa jäsentyneempi ja perustellumpi näkemys. Tätä viitekehystä voidaan hyödyntää periaatteessa minkä tahansa ohjelmointikielen siirrettävyyden tutkimisessa.

Kuudentena tavoitteena oli selvittää, miten siirrettävyysongelmia on Javassa ratkaistu. Keskeisiä siirrettävyyttä tukevia mekanismejä ovat Java-alusta ja sen tiukka määrittely, sijoitteluhallitsimet, kytkettävän käyttöliittymätyylin arkkitehtuuri sekä kansainvälistämisen tuki. Silti Javasta löytyy siirrettävyysongelmia, jotka ilmetessään saattavat aiheuttaa siirtämistyötä. Siten nämä ongelmat on hyvä tiedostaa ja niihin kannattaa varautua ajoissa.

Seitsemäntenä tavoitteena oli arvioida, miten ohjelman ja sen graafisen käyttöliittymän siirrettävyys toteutuu Javassa. Siirrettävyyden perusmekanismit ovat olemassa, sillä jo-

kaiselle siirrettävyysympäristön osatekijän tasolle on rakennettu jokin mekanismi edistämään siirrettävyyttä. Arviot Java-ohjelmayksikön siirrettävyydestä ovat jokaisen siirrettävyysympäristön osatekijän suhteen vähintäänkin tyydyttäviä, joten Javassa on selvästi huomioitu siirrettävyys hyvin laajasti. Silti Javasta löytyy yksittäisiä siirrettävyysongelmia, kuten edellä mainittiin. Tässä työssä Javan siirrettävyyden tarkastelu on kuitenkin rajoittunut käsitteellisteoreettiselle tasolle, joten arviot Javan siirrettävyydestä ovat lähinnä suuntaa-antavia.

Jatkotutkimuksena voisi olla mielenkiintoista konstruktiivisesti testata ja tutkia yleistä siirrettävyyden viitekehystä käyttäen Java-ohjelmayksikön todellisen siirrettävyyden tasoa useissa erilaisissa kohdeympäristöissä.

LÄHTEET

Andersson L., Java 2 -ohjelmointi, Suom. R. Torkkeli, Pagina, Espoo, 1999.

Arnold K. & Gosling J., The Java Programming Language Second Edition, Addison-Wesley, Yhdysvallat, 1998.

Bass L. & Coutaz J., Developing Software for the User Interface, Addison-Wesley, Yhdysvallat, 1992.

Becker J. D., Multilingual word processing, Scientific American, Vol. 251, No. 1, 1984, 82—93.

Bell C. G., Toward a History of (Personal) Workstations. Teoksessa: Goldberg A. (toim.), A History of Personal Workstations, Addison-Wesley, Yhdysvallat, 1988.

Cadenhead R., Java 2 Trainer, Suom. E. Suominen, IT Press, Helsinki, 1999.

Campione M. & Walrath K., The Java Tutorial Second Edition, Addison-Wesley, Yhdysvallat, 1998.

Campione M. & Walrath K., The JFC Swing Tutorial [online], Addison-Wesley, Yhdysvallat, 1999 [viitattu 16.4.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/docs/books/tutorial/uiswing/index.html >

Campione M., Walrath K., Huml A. & Tutorial team, The Java Tutorial Continued [online], Addison-Wesley, Yhdysvallat, 1998 [viitattu 22.4.2000]. Saatavilla www-muodossa: <URL: http://java.sun.com/docs/books/tutorial/overview/index.html >

Cowell W. (toim.), Portability of Numerical Software, Springer-Verlag, Berlin Heidelberg, 1977.

Davis M., Java Cookbook: Creating Global Applications [online], Yhdysvallat, 1998 [viitattu 24.4.2000]. Saatavilla www-muodossa:
<URL: http://www-4.ibm.com/software/developer/library/globalapps>

Davis M., Felt D. & Raley J., International Text in JDK 1.2 [online], Yhdysvallat, 1998, [viitattu 30.4.2000]. Saatavilla www-muodossa:
<URL: http://www-4.ibm.com/software/developer/library/international-text/>

Davis M. & Shih H., The Java International API: Beyond JDK 1.1 [online], Yhdysvallat, 1998 [viitattu 29.4.2000]. Saatavilla www-muodossa:
<URL: http://www-4.ibm.com/software/developer/library/intljava.html>

Galitz W., The Essential Guide to User Interface Design, John Wiley & Sons Inc., Yhdysvallat, 1996.

Gosling J., The Java Language Environment [online], Yhdysvallat, 1996, [viitattu 26.3.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/docs/white/langenv/index.html>

Filipski A., A Case Study in Software Portability: The Unix Symbolic Debugger, Proceedings of the ACM Sigsmall Symposium on Small Systems [online], 1985, [viitattu 23.11.1999]. Saatavilla www-muodossa:
<URL: http://www.acm.org/pubs/articles/proceedings/small/317164/p85-filipski/p85-filipski.pdf>

Fournier R., A methodology for client/server and web application development, Yourdon Press Computing Series, 1999.

Hatfield D., Conference on Easier and More Productive Use of Computer Systems, Ann Arbor, 1981.

Henderson J., Software Portability, Gower Technical Press Ltd, Englanti, 1988.

Horstmann C. & Cornell G., Core Java Volume I — Fundamentals, Prentice Hall Inc., Yhdysvallat, 1997.

Johnson D., Comparing WFC and JFC., Dr. Dobbs Journal: Software Tools for the Professional Programmer, Vol. 24, No. 2, 1999, 90—95.

Kramer D., The JavaTM Platform [online], Yhdysvallat, 1996, [viitattu 24.3.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/docs/white/platform/javaplatformTOC.doc.html>

Lewandowski S., Frameworks for Component-Based Client/Server Computing, ACM Computing Surveys, Vol. 30, No. 1, 1998, 3—27.

Lodding K. N., Iconic interfacing, IEEE Computer Graphics and Applications, Vol. 3, No. 2, 1983, 11—20.

Lunde K., Understanding Japanese Information Processing, O'Reilly and Associates Inc., 1993.

McCall, J., Richards P. & Walters G., Factors in Software Quality, NTIS, 1977.

Mooney J., Portability and Reusability Common Issues and Differences, Proceedings of the ACM 23rd Annual Computer Science Conference on Computer Science Conference [online], 1995, [viitattu 23.11.1999]. Saatavilla www-muodossa:
<URL: http://www.acm.org/pubs/articles/proceedings/csc/259526/p150-mooney/p150-mooney.pdf>

Mooney J., Strategies for Supporting Application Portability, IEEE Computer, Vol. 23, No. 11, 1990, 59—70.

Mullet K. & Sano D., Designing Visual Interfaces, Prentice Hall Inc., Yhdysvallat, 1995.

Nielsen J., Usability Engineering, Academic Press Inc., Yhdysvallat, 1993.

Pressman R., Software Engineering: a Practitioner's Approach, The McGraw-Hill Companies Inc., Yhdysvallat, 1994.

Pressman R., Software Engineering: a Practitioner's Approach, 4th ed., The McGraw-Hill Companies Inc., Yhdysvallat, 1997.

Rowley D., The Business of Application Portability [online], StandardView, Vol 4, No. 2, 1996, [viitattu 23.11.1999]. Saatavilla www-muodossa: <URL: http://www.acm.org/pubs/articles/journals/standardview/1996-4-2/p80-rowley/p80-rowley.pdf>

Rutkowski C., An introduction to the human applications standard computer interface, part 1: Theory and principles, BYTE, Vol. 7, No. 11, 1982, 291—310.

Scheifler R. & Gettys J., The X Window System, ACM Transactions on Graphics, Vol. 5, No. 2, 1986, 79—109.

Shneiderman B., The future of information systems and the emergence of direct manipulation, Behaviour and Information Technology, 1982, 237—256.

Sommerville I., Software Engineering, 5th ed., Addison-Wesley, Harlow, 1995.

Sood M., What Is Swing?, Dr. Dobbs Journal: Software Tools for the Professional Programmer, Vol. 23, No. 9, 1998, 111—114.

Sun Microsystems Inc., GUI Construction With Java Foundation Classes, Student Guide, Colorado, Yhdysvallat, 1998a.

Sun Microsystems Inc., Input Method Framework Design Specification [online], Yhdysvallat, 1998b, [viitattu 27.4.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/products/jdk/1.2/docs/guide/intl/spec.html>

Sun Microsystems Inc., JavaTM 2 Platform, Micro Edition (J2METM) [online], Yhdysvallat, 2000a, [viitattu 19.8.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/j2me>

Sun Microsystems Inc., JavaTM 2 Platform, Standard Edition, v1.2.2 API Specification [online], Yhdysvallat, 1999a, [viitattu 27.4.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/products/jdk/1.2/docs/api/index.html>

Sun Microsystems Inc., JavaTM 2 SDK, Standard Edition, version 1.3, Summary of New Features and Enhancements [online], Yhdysvallat, 2000b, [viitattu 12.8.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/j2se/1.3/docs/relnotes/features.html>

Sun Microsystems Inc., JDK Internationalization Overview [online], Yhdysvallat, 1998c, [viitattu 27.4.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/products/jdk/1.2/docs/guide/internat/intlTOC.doc.html>

Sun Microsystems Inc., Porting Considerations and Wrap-Up, Java Programming Workshop -kurssimateriaali, Colorado, Yhdysvallat, 1997.

Sun Microsystems Inc., Programmer's Guide to the JavaTM 2D API: Enhanced Graphics and Imaging for Java [online], Yhdysvallat, 1999b, [viitattu 1.5.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/products/jdk/1.2/docs/guide/2d/spec/j2d-title.fm.html>

Sun Microsystems Inc., Pure JavaTM Certification Guide [online], Yhdysvallat, 2000c, [viitattu 27.8.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/100percent/pjcg.pdf>

Sun Microsystems Inc., Sun and Microsoft Settle Lawsuit; Settlement Protects Integrity of Java Platform [online], Yhdysvallat, 2001, [viitattu 14.02.2001]. Saatavilla www-muodossa:
<URL: http://www.sun.com/smi/Press/sunflash/2001-01/sunflash.20010123.1.html>

Sun Microsystems Inc., Supported Locales [online], Yhdysvallat, 1999c, [viitattu 18.11.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/j2se/1.3/docs/guide/intl/locale.doc.html>

Sun Microsystems Inc., The Java HotSpotTM Client and Server Virtual Machines [online], Yhdysvallat, 2000d, [viitattu 20.8.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/j2se/1.3/docs/guide/performance/hotspot.html>

Tanenbaum A., Klint P. & Bohm W., Guidelines for Software Portability, Software-Practice and Experience, Vol. 8, 1978, 681—698.

Thomas A., JavaTM 2 Platform Enterprise Edition [online], Yhdysvallat, 1999, [viitattu 18.8.2000]. Saatavilla www-muodossa:
<URL: http://java.sun.com/j2ee/white/index.html>

Tietotekniikan liitto ry, Atk-sanakirja [online], Suomen ATK-kustannus, 1999, [viitattu 9.4.2000]. Saatavilla www-muodossa: <URL: http://www.ttlry.fi>

Unicode Consortium, The Unicode Standard: A Technical Introduction [online], Yhdysvallat, 1998, [viitattu 26.4.2000]. Saatavilla www-muodossa:
<URL: http://www.unicode.org/unicode/standard/principles.html>

Wallis P., Portable Programming, The Macmillan Press Ltd, Englanti, 1982.