Eräiden ohjelmointiympäristöjen internationalisointiominaisuuksista

Teemu Mäki

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Pro gradu –tutkielma
Joulukuu 2007

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Teemu Mäki: Eräiden ohjelmointiympäristöjen internationalisointiominaisuuksista
Pro gradu -tutkielma, 61 sivua, 4 liitesivua
Joulukuu 2007

Tutkielmassa on vertailtu muutamaa yleistä ohjelmointiympäristöä niiden internationalisointiominaisuuksien perusteella. Tutkittuihin kieliin kuului eri aikoina kehitettyjä ohjelmointikieliä, joiden voidaan olettaa sisältävän eritasoista tukea kansainvälisten sovellusten laatimiseen. Tutkimus kartoitti miten suuria nämä erot käytännössä ovat. Tutkimus suoritettiin arvioimalla ohjelmointiympäristöjä ISO 9126 –standardiin pohjautuvan arviointikehyksen avulla. Lisäksi tutkimuksessa arvioitiin eroja lokalisoidun ja ei-lokalisoidun sovelluksen välillä pääasiallisesti resurssikuormituksen suhteen. Tulokset tukevat jossakin määrin sitä käsitystä, että internationalisointi on mielekästä pitää osana normaalia ohjelmistokehitysprosessia.

Avainsanat ja -sanonnat: Internationalisointi, lokalisointi, ohjelmointiympäristöt.
CR-luokat: D.2.3 [Software Engineering]: Coding Tools and Techniques, D.2.8 [Software Engineering]: Metrics

Sisällysluettelo

1. Internationalisoinnista 4
   1.1. Peruskäsitteitä 4
   1.2. Internationalisointitekniikoista 6
   1.3. Internationalisoinnin sisäsyntyiset ongelmat 8
   1.4. Internationalisoinnin erityiskysymyksiä: kalenterijärjestelmät 9
   1.5. Internationalisoinnin erityiskysymyksiä: syöttömetodieditori 10
2. Tutkimuksen tavoitteista ja rakenteesta 11
3. Ohjelmointikielistä 13
   3.1. C 13
      3.1.1. Merkkijonot 13
      3.1.2. Internationalisointiratkaisut 14
   3.2. C++ 17
      3.2.1. Merkkijonot 17
      3.2.2. Paikanne 18
      3.2.3. Muita internationalisointiratkaisuja 18
   3.3. C# ja .NET 19
   3.4. Symbian 20
   3.5. Java 21
4. Tutkimusmetodi 22
   4.1. Viitesovellus 22
   4.2. ISO 9126-standardista 23
   4.3. Toiminnallisuus 23
   4.4. Luotettavuus ja käytettävyys 24
   4.5. Tehokkuus 27
   4.6. Ylläpidettävyys ja siirrettävyys 27
5. Ohjelmointiympäristöjen vertailu 28
   5.1. wxWidgets 28
   5.2. C-kirjasto ja Sunin lokalisointikirjastot 34
   5.3. C++ 39
   5.4. C# ja .NET 44
   5.5. Java 51
6. Yhteenveto 57
   6.1. Tuloksista 57
   6.2. Aiheita jatkotutkimukseen 58

1. Internationalisoinnista

1.1. Peruskäsitteitä

Internationalisoinnilla (I18N) tarkoitetaan ohjelmakoodin suunnittelemista ja ohjelmoimista sillä tavalla yleiseksi, että ohjelmaa voidaan käyttää eri kielillä ja eri kulttuuriympäristössä toimivissa järjestelmässä. Internationalisoinnin kaksi päätavoitetta on mahdollistaa ohjelmistotuotteen lokalisointi teknisellä tasolla ja varmistaa, että se kelpaa kansainvälisille markkinoille (LISA, 2007, 17, Esselink, 2000, 35).

Vaihetta, joka seuraa internationalisointivaihetta, ja jonka valmistelusta internationalisoinnissa on kyse, kutsutaan lokalisoinniksi (L10N). Lokalisointivaiheessa kulttuurisidonnaiset osat määritellään geneeriseen ohjelmistoon. Merkittävänä, mutta ei ainoana osana lokalisointia on teksti-informaation kääntäminen kohdekielille.

Joskus erotetaan omana käsitteenään multilingualisointi (M20N). Multilingualisoinnilla tarkoitetaan ohjelmiston internationalisoimista siten, että mahdollistetaan useamman kuin yhden kielen käyttäminen samanaikaisesti.

Silloin tällöin käytetään internationalisoinnin synonyyminä, tai kattokäsitteenä internationalisoinnille ja lokalisoinnille, termiä globalisointi (G11N). Termi on sikäli harhaanjohtava, että läheskään aina tavoitteena ei ole tehdä ohjelmistoja täysin globaaleille markkinoille.

Paikanteella (*locale*) tarkoitetaan määritystä, joka kertoo ohjelmistolle, missä kulttuuriympäristössä sen on määrä toimia. Kaikissa nykyaikaisissa käyttöjärjestelmissä käyttäjä pystyy valitsemaan halutun oletuspaikanteensa. Yleensä paikanteena käytetään kieli-maa -yhdistelmää, jolloin esimerkiksi suomenruotsalainen normaalisti käyttäisi paikannetta ”se-FI”, kun taas riikinruotsalainen käyttäisi paikannetta ”se-SE”. Paikanteessa voi olla enemmän kenttiä kuin kaksi, edellä mainittujen lisäksi esimerkiksi merkistö niitä tapauksia varten, joissa samaa kieltä saatetaan käyttää samassa maassa useammalla eri translitteraatiolla[1]. Tämä paikanteen nimeämistapa on määritelty standardissa RFC 3066, ja tämän standardin muodostavat maa- ja kielikoodin on määritelty standardeissa ISO 639-1 (kielet) ja ISO 3166 (maat).

---

[1] Tästä esimerkkinä Azerbaidžanissa puhuttu azeri, jossa on käytössä rinnakkain latinalainen ja kyrillinen merkistö, sekä paikanteet ”az-AZ-Latn” ja ”az-AZ-Cyrl”.

Suomen kielen sana ”kääntäjä” on moniselitteinen sikäli, että sillä voidaan tarkoittaa joko ohjelmointikieltä kääntävää ohjelmaa (*compiler*) tai kielenkääntäjänä työskentelevää ihmistä (*translator*). Tässä työssä edellistä tarkoitetaan jatkossa termillä *kääntäjä*, kun taas jälkimmäisestä käytetään termiä *kielenkääntäjä*. Yleisesti internationalisointi käsitetään ohjelmointikehityksenä, jossa erotetaan kulttuurisidonnaiset ohjelmiston osat kulttuuririippumattomista ominaisuuksista. Rajanvetoon kulttuurisidonnaisuuden ja kulttuuririippumattoman välillä vaikuttaa pitkälti haluttu internationalisointiaste. Kokkotos ja Spyropoulos (1997, 16) esittävät perusteellisessa internationalisointiarkkitehtuurissaan, että tähän ohjelmistoytimeksi (*program core*) kutsutussa osassa sijaitsee ainoastaan ohjelman peruslogiikka. Toisaalta, jos esimerkiksi tiedetään, että ohjelmiston ei tarvitse tukea muita syötetapoja kuin länsimainen näppäimistö, voisi perussyötekäsittely sisältyä ohjelmaytimeen.

Internationalisointi voidaan jakaa käyttöliittymän lokalisoinnin ja toiminnallisuuden lokalisoimisen mahdollistamiseen. Esimerkkinä ohjelmasta, jonka käyttöliittymä on lokalisoitu, mutta toiminnallisuutta ei, voisi olla kuvitteellinen tekstinkäsittelyohjelma, jonka valikot ja viestit on kirjoitettu käyttäjän äidinkielellä, mutta ohjelma pystyy käsittelemään ainoastaan englanninkielistä tekstiä. Esimerkistä käy selväksi, että käyttöliittymän lokalisointi vaikuttaa käyttökokemukseen, mutta toiminnallisuuden lokalisointi koko ohjelman käyttöalueeseen.

Kulttuurisidonnaisissa ohjelmiston osissa voidaan edelleen ajatella olevan kahdentyyppisiä osia: kulttuuririippuvaista dataa ja kulttuuririippuvaisia algoritmeja. Edelliseen kuuluvat asiat, jotka voidaan määritellä ilman ohjelmalogiikan muuttamista, kuten käyttäjille näytettävät tekstit (ilmoitukset, käyttöliittymäkomponenttien nimiöt), käyttöliittymäkomponenttien geometria, fonttiinformaatio, notaatiokonventiot, numeroerottimet, valuuttasymbolit sekä aika- ja päivämäärämerkintöjen muoto. Jälkimmäiseen kuuluvat asiat, joiden koodaaminen edellyttää erilaista ohjelmalogiikkaa, kuten tavutus, lajittelu, isojen ja pienten kirjainten erottelu, oikeinkirjoituksen tarkistus jne. (Esselink, 43, Kano, 34-36)

Käpyaho (2001, 11) huomauttaa, että usein on järkevintä toteuttaa perustavanlaatuisen internationalisoinnin edellyttämät algoritmit käyttöjärjestelmätasolla, jolloin sovelluksen internationalisoinnissa riittää huolehtia siitä, että näitä käyttöjärjestelmän tarjoamia palveluja käytetään tarjotun rajapinnan kautta.

Kulttuuririippumattomiin osiin kuuluvat myös sellaiset tekstit, joiden kääntäminen ei ole mielekästä ohjelman toimivuuden kannalta, esim. käyttöjärjestelmäkomennot ja hakemistopolut. Makrokielten lokalisoimisen mielekkyydestä voidaan olla eri mieltä. Makrokielen lokalisoiminen saattaa alentaa kynnystä

kielen opetteluun, mutta erikielisten makroversioiden olemassaolo vaikeuttaa niiden siirrettävyyttä. Visual Basic for Applications -makrokieli on lokalisoitu, ja siirrettävyysongelma on ratkaistu käyttämällä sovelluskohtaisia sanakirjoja erikielisille makroille (Kano, 45).

1.2. Internationalisointitekniikoista

Taylor (1992, 63-76) jakaa internationalisoinnin kolmeen eri luokkaan sen mukaan, missä ohjelman rakentamisen vaiheessa kulttuurisidonnaisten osien erottaminen tapahtuu.

Käännösaikaisessa internationalisoinnissa (compile-time internationalization) eri kulttuureille tarkoitetut osat kirjoitetaan lähdekoodiin. Teknisesti tämä voidaan hoitaa joko niin, että jokainen kieliversio kirjoitetaan omaan lähdekooditiedostoonsa tai siten, että erotetaan kääntäjän esikäsittelijäkomennoilla eri kieliset osiot:

```
void hello_world(void)
{
#if LANGUAGE=FINNISH
  printf ("%s \n", "Moro maailma");
#elif LANGUAGE=ENGLISH
  printf ("%s \n", "Hello world");
#endif
exit(0);
}
```

Lähestymistavan ongelmana on se, että lähdekoodista tulee monimutkaisempaa ja vaikeasti ylläpidettävää. Tekstejä muutettaessa lähes identtistä lähdekoodia joudutaan kääntämään uusiksi yhtä monta kertaa kuin kieliversioita on. Lisäksi joko lokalisoijan täytyy ymmärtää lähdekoodia tai tämän täytyy työskennellä yhdessä ohjelmoijan kanssa. (Taylor, 1992, 63-68)

Astetta modulaarisemmassa ratkaisussa tulostuksesta huolehtiva rivi voitaisiinkin kirjoittaa geneerisesti:

```
printf ("%s \n", TERVEHDYS);
```

ja määritellä erillisissä otsikkotiedostossa paikalliset tekstit:

```
#ifndef FINNISH_H
#define FINNISH_H
#define TERVEHDYS "Moro maailma"
```

```
#endif
```

jne.

Tästä muutoksesta on se hyöty, että otsikkotiedostot voidaan esim. erikseen lähettää kielenkääntäjille, jolloin näiden ei tarvitse koskea varsinaiseen lähdekoodiin.

Linkitysaikaisessa internationalisoinnissa (link-time internationalization) kulttuurisidonnaiset osat on erotettu erillisiin lähdekooditiedostoihin, jotka linkitysvaiheessa yhdistetään geneeriseen ohjelmakoodiin. Etuna on se, että geneeristä osaa ei tarvitse kääntää kuin kerran, joten kieliversioita muutettaessa riittää kielispesifisten lähdekoodien uudelleen kääntäminen ja linkittäminen. Linkitysaikaisessa internationalisoinnissa on edelleen monia käännösaikaisen linkittämisen ylläpito-ongelmista. (emt., 68-71)

Mikäli internationalisointi määritellään Käpyahon tapaan kapeasti toiminnaksi, jonka tavoitteena on aikaansaada yhdellä koodiperustalla toteutettavissa oleva ohjelmisto (Käpyaho, 2001, 7), ei linkitys- tai käännösaikaisessa internationalisoinnissa ole vielä lainkaan kyse internationalisoinnista.

Ajonaikaisessa internationalisoinnissa (run-time internationalization) kulttuurispesifisten osien määrittely on tehty kokonaan riippumattomaksi lähdekoodista. Tällöin käytettävän paikanteen valinta tapahtuu käytännössä vasta ohjelman käynnistysvaiheessa. Geneerinen ohjelmakoodi tunnistaa käyttäjän haluaman paikanteen jollakin mekanismilla (esim. sovelluskohtaisesta asetustiedostosta tai tiedustelemalla sitä käyttöjärjestelmältä) ja lataa tämän jälkeen dynaamisesti kulttuurispesifiset osat erillisestä tiedostosta tai tietokannasta. (Taylor, 1992, 71-76) Käyttäjälle näkyvien lokalisoitavien tekstien yhteydessä tällaisesta tietokannasta käytetään tässä tutkielmassa yhtenäisyyden mukaisesti nimitystä viestiluettelo (*message catalogue*).

Ajonaikaista internationalisointia pidetään usein parhaimpana ratkaisuna. Useimmat seuraavissa luvuissa esitetyistä internationalisointiratkaisuista perustuvat jonkinlaiseen ajonaikaiseen mekanismiin. Ajonaikaisen internationalisoinnin haittapuoleksi voidaan katsoa lisääntynyt ajonaikaisten resurssien kulutus. Joissakin käyttöyhteyksissä, joissa näiden resurssien rajallisuus on korostunut (esim. mobiilisovellukset), saattaa olla perusteltua käyttää ennemmin jompaakumpaa ensiksimainituista ratkaisuista. Toisaalta ajonaikaista internationalisointia käyttäessä on mahdollista ajonaikaisesti poistaa muistista tarpeettomia resursseja, millä on mahdollista tehostaa muistinkäyttöä.

Nokia (2006, 10) esittää, että lokalisointi on mahdollista ottaa käyttöön kolmessa vaiheessa: käännösaikaisesti, asennuksen aikana tai ajonaikaisesti. Käännösaikainen lokalisointi tuottaa yhden asennuspaketin kutakin paikannetta varten. Tätä suositellaan, mikäli pyrkimyksenä on saada asennuspaketeista

mahdollisimman pienikokoisia, mistä on etua etenkin mobiiliympäristössä asennettavia ohjelmia ajatellen. Asennuksenaikaisessa lokalisoinnissa tuotetaan yksi asennuspaketti, joka sisältää kaikki tarvittavat resurssitiedostot, joista valitaan käyttöön tarvittavat resurssit asennuksen yhteydessä. Ajonaikaisessa lokalisoinnissa asennuspaketti on samankaltainen kuin edellisessä tapauksessa, mutta kaikki tuetut resurssit asennetaan, ja tarvittavat resurssit ladataan ajonaikaisesti.

Taylor ei tee eroa lokalisointi- ja internationalisointitoimien välillä. He ja muut (2002, 91-92) ehdottavat vielä tarkempaa jakoa siten, että internationalisoinnin ja lokalisoinnin voidaan katsoa tapahtuvan joko samassa tai eri vaiheessa. Näin tarkkaan jakoon en näe kuitenkaan tarvetta tämän tutkimuksen yhteydessä.

1.3. Internationalisoinnin sisäsyntyiset ongelmat

Seuraavassa tarkastellaan niitä seikkoja, jotka tekevät internationalisoinnista vaikeasti ymmärrettävän tehtäväalueen riippumatta ohjelmointiteknisistä asioista.

Monet internationalisointi- ja lokalisointiongelmat johtavat siitä yksinkertaisesta syystä, että tarve ohjelmistojen internationalisoimiselle on vielä kohtalaisen nuori. Näin ollen internationalisointinäkökohdat tuntuvat monille ohjelmistoalalla työskenteleville vierailta. Internationalisointia opetetaan korkeakoulujen ohjelmistotuotantoa käsittelevillä kursseilla parhaimmillaankin kursorisesti. Hoganin ja muiden (2002) mukaan tilanne on jonkin verran parantumassa, mutta internationalisointi ei siltikään ole tulossa kiinteäksi osaksi ohjelmistoalan opetusohjelmaa. McKenna (2002, 10) toteaa lisäksi, että opetuksen vähyydestä johtuen ohjelmistokehitys on yleensä ongelmalähtöistä; toteutuksessa lähdetään liikkeelle ydintoiminnallisuuden toteuttamisesta ja internationalisointi nähdään lisäominaisuutena, joka voidaan lisätä jälkikäteen. Tämän on todennut myös Zhao (2003, 10).

Usein internationalisointitarve havaitaan liian myöhäisessä vaiheessa, kun ohjelmistoa yritetään markkinoida globaalisti, huonolla menestyksellä. Ongelma tuntuisi olevan pahin englanninkielisessä maailmassa ja erityisesti Yhdysvalloissa, johon ohjelmistoteollisuus edelleen on vahvasti keskittynyt. Kysyttäessä XML-tekniikan huonon internationalisointitilanteen syitä XML-standardien uranuurtaja Tim Brayltä, hän vastasi: "It's not an XML problem, it's a problem of American psychology. Too many smart, good people who are Americans have trouble seeing past their borders and realizing that we Anglophones are a minority in the big picture." (Dumbill, 2001.)

Bos ja Dürst mainitsevat joitakin asioita, jotka tekevät internationalisoinnista vaikeaa. Internationalisointiin liittyvä tietämys on yleensä implisiittistä,

usein käytännössä opittua. Internationalisointi perustuu ihmisen käyttäytymiseen, joka ei ole säännöllistä ja näin ollen vaikeasti automatisoitavissa. Ylpeydellä ja politiikalla on myös sijansa. Jotkut internationalisointiin liittyvät asiat, kuten kaksisuuntaiset kirjoitusjärjestelmät (esim. heprea ja arabia), ovat lähtökohtaisesti kompleksisia. Monet internationalisoinnissa huomioitavat asiat, kuten vieraskieliset merkistöt, ovat tuntematonta maaperää useimmille kohdekulttuurin ulkopuolelta tuleville ihmisille. (Bos and Dürst, 1998)

Nämä huomiot korostavat, miksi on tärkeää, että internationalisointi ja lokalisointi on voitava erottaa erillisiksi prosesseiksi, jotka edellyttävät ohjelmistokehittäjästä erillisen asiantuntijan työtä, sekä puoltavat internationalisoinnin integroimista ohjelmistokehitysprosessiin heti sen alkuvaiheista lähtien.

1.4. Internationalisoinnin erityiskysymyksiä: kalenterijärjestelmät

Perinteisesti ohjelmointiympäristöjen ajan ja päivämäärien käsittelyfunktioiden pääasiallinen tarkoitus on toimia rajapintana ohjelmoijan ja järjestelmäkellon välillä hetkellisen kellonajan selvittämiseksi. Ajan käsittely on kuitenkin monisäikeinen, kompleksinen ja usein aliarvioitu ongelmakenttä. Garland (2005) käsittelee yksityiskohtaisesti ajan käsittelyyn liittyviä ongelmia C++-vakiokirjastossa. Seuraavassa käydään lyhyesti läpi tärkeimmät kulttuuriset näkökohdat kalenterijärjestelmiin liittyen.

Ohjelmistojen sovittamisessa globaalin käyttöön voi olla tarpeellista huomioida eri maissa käytössä olevat erilaiset kalenterijärjestelmät. Maailman yleisin nykyään käytössä oleva kalenterijärjestelmä on länsimaissa käytettävä gregoriaaninen kalenteri. Gregoriaanisen kalenterin edeltäjää kutsuttiin juliaaniseksi kalenteriksi. Juliaanisen kalenterin vuosi oli hieman liian pitkä, joten ajanlaskua piti tarkentaa tarkemmin tähtitieteellistä vuoden pituutta vastaavaksi. Gregoriaaninen kalenteri on korvannut juliaanisen eri aikaan eri maissa. Ensimmäisenä gregoriaaniseen kalenteriin on siirrytty Espanjassa, Portugalissa ja muutamassa muussa Länsi-Euroopan maassa vuonna 1582, viimeisenä Turkissa vuonna 1926. Juliaanisella kalenterilla on nykyään merkitystä lähinnä historiallisten päivämäärien merkitsemisessä ja astronomisissa laskelmissa. Sekä gregoriaaninen että juliaaninen kalenteri ovat ns. aurinkokalentereita. Muita laajassa käytössä olevia kalenterijärjestelmiä ovat mm. kiinalainen aurinkokuukalenteri (*lunisolar calendar*) ja islamilainen kuukalenteri (*hijri*-kalenteri). Vähemmän laajassa nykyaikaisessa käytössä olevia kalenterijärjestelmiä ovat mm. persialainen Bahá'í-kalenteri, bengalikalenteri (Bangladesh), buddhalainen kalenteri (Kaakkois-Aasian maat), heprealainen aurinko-kuukalenteri, iranilainen kalenteri (Iran, Afganistan), malayalam-kalenteri (Etelä-Intian osavaltio Kerala), nepalilainen Bikram Samwat-kalenteri, thaimaalainen Suriyakati-aurinkokalenteri ja tiibetiläinen aurinko-kuukalenteri. Monien marginaalisten

kalenterijärjestelmien käyttöalue rajoittuu uskonnollisten juhlapyhien määrittämiseen.

Toinen päivämäärien ja aikojen käsittelyssä huomioitava asia on se, että samankin kalenterijärjestelmän sisällä eri maissa on käytössä erilaisia notaatioita päivämäärille ja kellonajan. Nämä erot sisältyvät paikannejärjestelmään.

1.5. Internationalisoinnin erityiskysymyksiä: syöttömetodieditori

Eräs internationalisoitujen ohjelmistojen erityiskysymyksistä on ns. syöttömetodieditorin (*input method editor, IME)* käyttö. Syöttömetodieditorin tarve perustuu siihen, että tietokonejärjestelmät on perinteisesti suunniteltu käyttämään länsimaisia sekä muita foneettisia eli äännepohjaisia kirjoitusjärjestelmiä. Aasiassa on kuitenkin käytössä ideogrammeihin pohjautuvia kirjoitusjärjestelmiä, joissa tarvittavien merkkien määrä voi kasvaa tuhansiin. Lisäksi käytössä saattaa olla rinnakkain sekä foneettisia että ideogrammipohjaisia kirjoitusjärjestelmiä. Tästä merkittävin esimerkki on japani, jossa on käytössä kaksi foneettista kirjoitusjärjestelmää (hiragana, katakana) sekä ideogrammijärjestelmä (kanji). (Kano, 1995, 197-200) Tekstiä kirjoittaessa on siis tarpeen pystyä sekä syöttämään merkkejä kohtuullisella määrällä näppäimiä (jopa länsimaisella 101-näppäimisellä näppäimistöllä) että vaihtamaan käytettävää merkkijärjestelmää saman tekstin sisällä. Tätä tarvetta varten käyttöjärjestelmien aasialaisiin versioihin sisältyy syöttömetodieditori. Syöttömetodieditori on pieni sovellus, joka huolehtii näppäinpainallusten muuttamisesta halutuiksi kirjoitusmerkeiksi. Editori sisältää myös logiikkaa, joka pyrkii ennakoimaan, minkä merkin useista vaihtoehdoista käyttäjä aikoo syöttää seuraavaksi. Eri vaihtoehdot esitetään näytöllä, ennenkuin näppäintapahtuma lähetetään isäntäsovellukselle. Windowsin syöttömetodieditori toimii myös sellaisten sovellusten kanssa, jotka eivät toteuta IME-tukea. Isäntäsovelluksen hylätessä IME-spesifiset ikkunointijärjestelmäviestit editori lähettää valittua merkkiä vastaavat perinteiset näppäinsyöteviestit. Syöttömetodieditoria tukevat sovellukset voivat käyttää käyttöjärjestelmän tarjoamaa editoria sellaisenaan tai toteuttaa uudelleen sen käyttöliittymäosat, sallien editorin integroimisen isäntäsovellukseen saumattomasti. (emt., 212-225)

2. Tutkimuksen tavoitteista ja rakenteesta

Lokalisointi ja internationalisointi ovat ohjelmistokehitysprosessina suhteellisen uusia käsitteitä, eikä paljon tutkimusta ole vielä tehty näiltä alueilta. Yleisimmät tutkimukset keskittyvät joko kansainvälisten ohjelmien lokalisointiprosessin ominaisuuksiin (ts. kielenkääntäjien työhön) tai itse prosessiin yleisellä tasolla. Immonen ja Sajaniemi (2003, 12-13) ovat kartoittaneet mm. internationalisoitujen ohjelmistoprojektien toteuttamiskielten ja muiden työkalujen jakaumaa Suomen ohjelmistoteollisuudessa. Gross (2006) käy opinnäytetyössään läpi lokalisoijan työkaluja ja lokalisointiprosessin kysymyksiä. Kirjoittajalle ei kuitenkaan tullut kirjallisuudessa vastaan tutkimuksia, jotka olisivat keskittyneet internationalisointiin ohjelmoijan työkalujen eli ohjelmointiympäristöjen näkökulmasta.

Tutkielman tarkoituksena on luoda katsaus siihen, miten eri ohjelmointityökalut vastaavat edellämainitun kaltaisiin monikielisten ja globaalisti käytettävien ohjelmistojen kehitystarpeisiin. Ohjelmointikielen ja -työkalujen valintaa ohjaavat pääosin muut seikat kuin internationalisoitavuustekijät, mutta tutkimustuloksia voi mahdollisesti hyödyntää arvioidessa tämän valinnan vaikutusta ohjelmistokehitysprosessin internationalisointi- ja lokalisointivaiheisiin. Selvitämme myös, miten paljon ohjelmiston internationalisointi kuormittaa toisaalta ohjelmoijaa, toisaalta järjestelmää, jossa ohjelmistoa käytetään. Tämän perusteella arvioidaan, voidaanko internationalisointia pitää lisävaatimuksena, joka on järkevää harkita tapauskohtaisesti, vai ovatko internationalisoinnin vaikutukset niin pieniä, että se kannattaa ottaa osaksi jokaista ohjelmistokehitysprosessia.

Toisena tavoitteena tutkimuksessa on osoittaa ISO 9126 -standardin soveltuvuus internationalisoitavuuden arvioimiseen. ISO 9126 on tarkoitettu alunperin pääosin arviointikehykseksi ohjelmistoille yleisessä merkityksessä. Yleensä ohjelmistojen käyttäjiksi mielletään niiden loppukäyttäjät eli kuluttajat. Ohjelmointityökalujen ollessa tutkimuskohteena myös käyttäjän määrittely on tehtävä uudelleen. Ohjelmointiympäristön ominaisuudet eivät suoraan vaikuta loppukäyttäjään, mutta sitäkin enemmän ohjelmistokehittäjän työhön. Tarkoituksena on luoda yksi mahdollinen arviointitekniikka internationalisointiominaisuuksille, joka auttaa luokittelemaan sekä antamaan joitakin numeerisia tunnuslukuja internationalisointirajapinnoille.

Seuraavassa tarkastelussa huomaamme, että kansainvälisyysvaatimukset ovat jatkuvasti kasvaneet. Samoin toteamme, että eri ohjelmointiympäristöjen kehitys on tapahtunut eri ajanjaksoina. Selvitämme myös sitä, miten paljon pa-

remmin uudet ohjelmoijan työvälineet vastaavat kasvaneeseen tarpeeseen verrattuna iäkkäämpiin vastineisiinsa.

Seuraavassa luvussa luonnehditaan yleisellä tasolla tarkastelun kohteena olevia ohjelmointiympäristöjä, niiden tärkeimpiä internationalisointiin vaikuttavia ominaisuuksia ja viitteeksi joitakin työn ulkopuolelle jääviä ympäristöjä. Kaikki tutkimuksessa tarkasteltavat kirjastot ovat kyseisen ohjelmointikielen vakiokirjastoja, lukuunottamatta wxWidgets-kirjastoa, joka on monialustainen avoimen lähdekoodin käyttöliittymä- ja sovelluskehityskirjasto.

Luvussa 4 esitellään se teoreettinen kehys, jota ohjelmointiympäristöjen vertailussa käytetään. Teorian ytimen muodostaa ISO 9126 -standardi, mutta koska se ei suoraan sovellu ohjelmointikirjastojen arviointiin, määritetään ne poikkeavuudet standardista, jotka ovat tarpeellisia mielekkään arvioinnin kannalta.

Luvussa 5 on esitelty tarkastelun kohteena olevat ympäristön arviointikehyksen kriteerien mukaisesti. Kukin kappale sisältää taulukon, jossa on kuvattu ympäristön keskeinen internationalisointirajapinta, joka on määritetty todellisen viitesovelluksen avulla.

Lopuksi tehdään päätelmiä näiden tulosten perusteella sekä arvioidaan metodin soveltuvuutta ja tutkimuksen vahvuuksia ja puutteita.

3. Ohjelmointikielistä

Seuraavassa tarkastellaan joitakin yleisimpiä ohjelmointikieliä sekä niiden internationalisointiin vaikuttavia ominaisuuksia. Erityisesti keskitytään merkkijonojen ja muiden resurssien käsittelyyn. Joidenkin kielien kohdalla tarkastellaan myös standardikirjaston kehittämisen jälkeen luotuja vaihtoehtoisia kirjastoja tai työkaluja.

3.1. C

C-kielen kehitys on alkanut 1960-luvulla, ja sen nykyistä muistuttava versio valmistui 1972. C-kieli kehitettiin alunperin korvaamaan assembler-kielet ja mahdollistamaan laitteistoriippumattomampi koodi. C-kielellä on vielä nykyäänkin iso merkitys etenkin UNIX-maailmassa ja järjestelmäohjelmoinnissa.

C-kääntäjä sisältää erityisen esikäsittelijäsyntaksin, jolla voidaan määritellä esimerkiksi ehdollisia käännöksiä sekä erilaisia makroja. Tämä mahdollistaa kohdassa 1.2 esitellyt käännös- tai linkitysaikaiset internationalisointitekniikat sekä koodin selkeyttämisen makroilla, kuten alakohdan 3.1.2. esimerkeissä tehdään. (Kernighan and Ritchie 1988, 228-233, ISO9899, §6.10)

3.1.1. Merkkijonot

C:ssä merkkijonoja käsitellään perinteisesti kokonaislukutaulukoina, jotka päättyvät nollaan. Ohjelmakoodin sisällä merkkijonoliteraali määritellään kirjoittamalla se lainausmerkkien väliin (`"esimerkki"`). Merkkijonoille, kuten muillekin taulukoille C:ssä, voi varata ainoastaan kiinteän tilan. Taulukolle varatulla muistialueella pysymistä ei kuitenkaan valvota mitenkään, mikä mahdollistaa ns. puskuriylivuodot. Alkuperäisessä C-standardissa taulukon alkiot ovat 8-bittisiä, joten tällaisilla merkkijonoilla voidaan kuvata ainoastaan 8-bittistä merkistöä käyttäviä merkkijonoja. C89-standardissa on määritelty myös ns. leveät merkkijonot (*wchar_t*), joiden merkit voivat olla yli yhden tavun mittaisia ja sitä vastaava literaalimerkintä (`L"esimerkki"`). Standardi ei määrittele merkin leveyttä, vaan tämä on jätetty toteuttajan vastuulle. (Kernighan and Ritchie, 194)

C-standardikirjastossa on joukko funktioita, jotka käsittelevät nollaloppuisia merkkijonoja (esim. strcat, strcmp). Näiden funktioiden toteutus ei kuitenkaan ole kovin älykäs, joten esimerkiksi strcat-funktiota käytettäessä ohjelmoijan täytyy huolehtia siitä, että yhdistetty merkkijono mahtuu sille tarkoitettuun taulukkoon. Lisäksi on huomattava, että nämä funktiot toimivat oikein ainoastaan ASCII-tekstiä käsiteltäessä, eivätkä silloinkaan huomioi tiettyjä erityiskysymyksiä, kuten isojen ja pienten kirjainten erottelua ja välimerkkien sijaintia lajittelujärjestyksessä. (emt., 249-250)

3.1.2. Internationalisointiratkaisut

Alkuperäinen C-standardikirjasto ei sisällä valmiiksi internationalisointia auttavia funktioita. Tämän puutteen korjaamiseksi aikojen saatossa on kehitetty useita kilpailevia omistettuja ja avoimia kirjastoja. Näistä tarkastellaan lyhyesti Hewlett-Packardin NLS-kirjastoa ja GNU-projektin gettext-työkaluketjua. C99-standardin kirjastossa on mukana paikanteenhallintaan liittyviä funktioita (*locale.h*), paikanteen huomioivia päivämäärän muotoilufunktioita (*time.h*) ja monitavuisten merkkien (*"wide character"*) käsittelyyn liittyviä funktioita (*wchar.h, wctype.h*) (ISO 9899, §7.11, §7.22-§7.24).

Hewlett-Packard on kehittänyt internationalisointikirjaston, josta käytetään nimeä Native Language Support System (NLS). NLS-kirjasto sisältää korvaavia rutiineja C-standardin vastaaville funktioille, jotka huomioivat käytössäolevan paikanteen. NLS-kirjaston funktiot käyttävät 8-bittisiä merkistöjä, ja osaavat esimerkiksi lajitella merkkijonoja eri kielten lajittelukonventioiden mukaisesti. Näiden lisäksi kirjasto sisältää työkaluja, joita voidaan käyttää hyödyksi viestiluetteloa ja paikannetietokantaa luodessa ja ylläpitäessä.

NLS-kirjasto on suunniteltu siten, että olisi mahdollisimman helppoa muuttaa perinteistä C-standandikirjastoa käyttävä ohjelmakoodi käyttämään NLS:ää. Kirjaston setlocale-funktio hakee järjestelmän ympäristömuuttujista tiedon käytettävästä paikanteesta. Kirjasto tukee eri paikanteen käyttämistä kielen, lajittelujärjestyksen, isojen ja pienten kirjanten erittelyn, rahayksikön ja luku- ja kellonaikajärjestelmän suhteen siten, että kukin näistä määritellään omassa ympäristömuuttujassaan.

HP:n viestiluettelon käyttö edellyttää viestiluettelon kääntämistä koneella luettavaksi tiedostoksi. Ratkaisun taustalla on tarkoitus parantaa tekstin eriyttämistä koodista piilottamalla lokalisoidut viestit muilta kuin NLS-ylläpitäjiltä. C-standardikirjastolle ohjelmoidun rutiinin muuttaminen NLS-tyylistä viestiluetteloa käyttäväksi tapahtuu pääpiirteissään seuraavaa proseduuria käyttämällä (Taylor, 1992, 213-216):

1. Eristetään merkkijonoliteraalit koodista käyttämällä *findstr* -työkalua.
2. Tarkistetaan lista ja varmistetaan, että joukossa ei ole ei-lokalisoitavia tekstejä.
3. Määritetään numero jokaiselle luettelon viestille ja korvataan koodissa merkkijonoviittaukset *catgets*-funktion kutsuilla. Tähän tarkoitukseen on olemassa *insertmsg*-työkalu.
4. Lisätään ohjelman alkuun koodi, joka ottaa viestiluettelon käyttöön.
5. Luodaan viestiluettelo *gencat*-työkalulla.

Keskeinen osa viestien käsittelyssä on siis *catgets*-funktiolla. Sillä on seuraavanlainen syntaksi: catgets (catd, set, message, default_msg), jossa *catd* on viite luettelotiedostoon, *set* on viestijoukon numero, message on viestin numero ja *default_msg* on merkkijonoliteraali, joka palautetaan siinä tapauksessa, että luettelossa ei ole lokalisoitua versiota viestistä (Sun 2005, 139). Numeroiden käyttö viestin tunnuksena ei ole omiaan lisäämään koodin selkeyttä. Oletetaan seuraavassa, että funktio `void popup(char *viesti)` näyttää käyttäjälle parametrinä saamansa merkkijonon ponnahdusikkunassa. Tällöin internationalisoidun sovelluksen koodirivi voisi näyttää seuraavalta:

```
popup(catgets(catd, 1, 2, ”Tiedostoa ei löydy”));
```

Koodia voidaan hieman selkeyttää määrittelemällä erillisessä otsikkotiedostossa toisaalta kääntäjämakro, joka vähentää parametrien määrää yhdellä, ja toisaalta korvaamalla numerot loogisilla nimillä, jotka noudattavat tiettyä konventiota, esim:

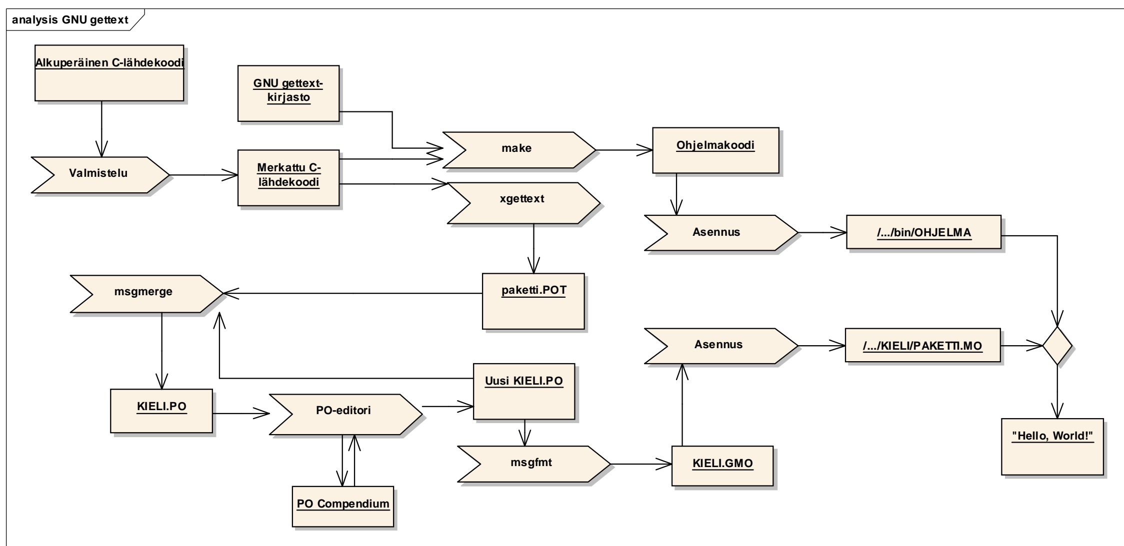
```
#define getmsg(a, b) catgets (catd, a, b)

#define ERROR_MSG 1

#define POP_FILENOTFOUND 2, „Tiedostoa ei löydy‟
```

jolloin edellinen esimerkki selkeytyy huomattavasti:

```
popup (getmsg(ERROR_MSG, POP_FILENOTFOUND));
```



Kuva 1. Internationalisointiprosessi GNU gettext-työkaluketjulla. (GNU, 2001)

GNU-projektin yhteydessä on kehitetty työkaluketju (ks. Kuva 1.), jota usein kutsutaan keskeisen funktion mukaisesti nimellä *gettext.* Gettext-lähestymistavan perusajatus on sama kuin HP:n NLS:ssä, eli tavoitteena on pyrkiä minimoimaan lähdekoodin muutokset lokalisoimattomaan koodiin verrattuna. Gettext-lähestymistavassa on pyritty vähentämään työvaiheita, joita tarvitaan viestiluettelojen käyttämisessä ohjelmasta käsin. Catgets-menetelmän

käyttöaskeleet muistuttavat yleistä tiedostojen käsittelytapaa ("avaa-käytä-sulje"). Gettextissä on pyritty suoraviivaisempaan käyttötapaan. Gettext-menetelmässä viestiluetteloita käsitellään käännösvaiheessa ihmisen luettavissa PO (*"portable object"*)-tiedostoissa, jotka käyttöönoton yhteydessä käännetään binäärisiksi MO (*"machine object"*)-tiedostoiksi. NLS:n tavoin gettext-työkaluketjussa on työkalu (*xgettext*), joka luo tyhjän viestiluettelon (.POT) koodista löytämistään merkkijonoliteraaleista. Tyypillisimmissä käyttötapauksessa xgettextin käyttö edellyttää sitä, että merkkijonot on merkitty lokalisoitaviin ja ei-lokalisoitaviin merkkijonoihin ennen xgettextin ajamista. Konventiona on merkitä lokalisoitava merkkijono `_("esimerkki")` ja ei-lokalisoitava `N_("esimerkki").` Näin ollen koodi muuttuu lähdekooditasolla lokalisoimattomaan koodiin nähden vain 3-4 merkkiä merkkijonoa kohden. Jotta edellinen notaatio ei muuttaisi koodin toimivuutta, lisätään otsikkotiedostoon seuraavat määritykset:

```
#define _(String) (String)

#define N_(String) String
```

Kun lokalisointi on suoritettu, ja viestiluettelot ovat käytettävissä MO-muodossa, muutetaan määrittelyt seuraaviksi:

```
#include <libintl.h>

#define _(String) gettext (String)

#define gettext_noop(String) String

#define N_(String) gettext_noop (String)
```

Notaatio `_("merkkijono")` tarkoittaakin nyt siis internationalisointikirjaston gettext()-funktion kutsua. gettext_noop()-variaatiota käytetään sellaisissa yhteyksissä, jossa funktiokutsu ei ole mahdollinen (esim. alustuslistoissa). Makro ei tee käännösaikana mitään, mutta toimii käännettävän merkkijonon merkkinä xgettext-työkalulle. (GNU, 2001)

gettext-funktio vastaa toiminnaltaan catgets-funktiota; se siis palauttaa viestiluettelosta paikannetta vastaavan merkkijonon. Rajapinta on kuitenkin yksinkertaisempi: gettext() saa parametrikseen merkkijonotyyppisen viestitunnuksen. Kuten ylläolevista määrityksestä ja merkitsemiskonventiosta käy ilmi, viestitunnuksena käytetään alkuperäistä lokalisoimatonta merkkijonoa. Lähestymistapa on ohjelmoijaystävällinen, koska päästään eroon epähavainnollisista numerokoodeista, mutta ongelmia syntyy siinä tapauksessa, että alkuperäistä esimerkiksi englanninkielistä termiä vastaakin muissa kielissä useita eri käännöksiä käyttöyhteydestä riippuen. Ongelmaa havainnollistaa tositapaus, jossa avaruusaiheisen näytönsäästäjän nimi "Space" oli käännetty ruotsiksi lokali-

soidussa versiossa ”Mellanslag” (välilyönti). Myöhemmissä gettext-toteutuksissa tällaisten tilanteiden ratkaisemiseksi on kirjastoon lisätty funktioita, jolla voidaan asettaa kulloinkin käytössä oleva käyttöalue. ”Space”-esimerkissä saattaisi olla esimerkiksi käyttöalueet ”screensaver” ja ”keyboard”. Sunin C-kirjastossa on gettextin lisäksi käytössä seuraavat funktiot:

- char *dgettext(const char *domainname, const char *msgid)
- char *dcgettext(const char *domainname, const char *msgid, int category)
- char *textdomain(const char *domain)
- char *bindtextdomain(const char *domainname, const char *dirname).

Tällöin voidaan säilyttää selkeys asettamalla kulloinkin käytettävissä oleva käyttöalue, tai vaihtoehtoisesti, jos moniselitteisiä tapauksia on vähän, hakea oikea teksti eksplisiittisesti dgettext- tai dcgettext-funktion avulla. (Sun, 2005)

Gettext-metodia pidetään nykyisellään de facto-standardina erityisesti avoimen lähdekoodin ohjelmistojen resurssitiedostojen käsittelyyn.
Internationalisointi sekä NLS- että gettext-lähestymistapaa käyttäen on suhteellisen yksinkertaista, jos koodi on valmiina ennen käännösten aloittamista. Koodia muutettaessa ja uusien viestien syntyessä tai vanhojen muuttuessa tulee ylimääräistä päänvaivaa tekstien ylläpidosta. Pahimmassa tapauksessa koodi ja viestiluettelo eivät ole toistensa kanssa synkronoituja. Koodin ja tekstiluetteloiden päivittämiseen on olemassa työkaluja, mutta ne ratkaisevat vain osan ongelmista, eikä manuaaliselta työltä vältytä. Tämä puoltaa sitä näkemystä, että kielenkääntäjien ja muiden lokalisointiasiantuntijoiden työ pitäisi pyrkiä aloittamaan siinä vaiheessa, kun koodi on mahdollisimman kypsässä tilassa.

3.2. C++

C++ on C-kielen oliolaajennos. C++ on hybridikieli, joten olio-ominaisuuksista huolimatta se mahdollistaa edelleen proseduraalisen ohjelmoinnin C:n tapaan.

C++ sisältää kaikki C:n tietotyypit ja näiden lisäksi luokkatyypin. Osoittimien lisäksi C++ tarjoaa niitä turvallisemman vaihtoehdon, viitetyypin. C++ mahdollistaa tietotyypistä riippumattomien geneeristen rakenteiden määrittelyn kaavaimien avulla. C++-vakiokirjastoon sisältyy Standard Template Library (STL), joka sisältää hyödyllisiä säiliökaavaimia sekä joitakin algoritmeja. Yleistäen voidaan todeta, että C++:ssa tehdään enemmän tyyppitarkistuksia kuin C:ssä.

3.2.1. Merkkijonot

Koska C++ on enimmäkseen takaperin yhteensopiva C:n kanssa, voidaan C++-koodissa käyttää C-tyylisiä nollaloppuisia merkkitaulukoita merkkijonoina. Hyvin usein näin tehdäänkin. C++-vakiokirjastossa on kuitenkin määritelty

turvallisempi ja helppokäyttöisempi merkkijonotyyppi (std::string). Merkkijonotyyppien metodit korvaavat epäturvalliset strcat- ym. funktiot. String ja wstring ovat itseasiassa esimääriteltyjä tyyppimäärityksiä merkkijonon toteuttavalle kaavaimelle. Kaavain on suunniteltu siten, että on helposti mahdollista lisätä omia merkkityyppejä. Useimmille ohjelmoijille esimääritellyt merkkijonotyypit toimivat halutulla tavalla. (Stroustrup, 1997, 580-582)

3.2.2. Paikanne

C++:ssa internationalisointi on huomioitu C:tä paremmin ottamalla käyttöön paikanteen käsite ja sitä vastaava luokka C++:n standardikirjastossa (std::locale). C++:n paikanteella ei ole paljoakaan tekemistä C:n paikanteen kanssa sen käyttötarkoitusta lukuunottamatta. C++:n locale-luokassa määritellään internationalisointisemantiikka käyttäen ns. facet-luokkia. C++:n esimääritellyt facetit vastaavat kokolailla C-vakiokirjaston paikannefunktioiden tarjoamia palveluita. Yksi vakiofaceteista toteuttaa alakohdassa 3.1.2 esitellyn NLS-standardin viestiluettelojen käsittelyn. Tälle on myös toteutettu käyttöä helpottavia kaavaimia. C++:n facetit tarjoavat mielekkään tavan toteuttaa periaatteessa mikä tahansa internationalisoinnin vaatima palvelu laajentamalla paikannetta. (RWS, 1996) Tärkeä ero C:hen nähden on se, että C++-paikanne on luokka, kun taas C:n paikannefunktiot perustuvat globaaleihin tietorakenteisiin. Näin ollen C++-paikanteesta voidaan luoda mielivaltainen määrä toisistaan riippumattomia ilmentymiä, mikä helpottaa huomattavasti monikielisten sovellusten luomista. Paikannetuki on huomioitu C++-vakiokirjaston luokissa, joten internationalisoinnista huolehtiminen voidaan hoitaa käyttämällä locale-luokkaa ilman, että vakiokirjaston luokkien toteutuksiin tarvitsee vaikuttaa. Tyypillisimmillään paikannetta käytetään syötteissä yhdistämällä paikanne syötteeseen syöteolion imbue()-metodilla. (Stroustrup, 1997, 649-650)

3.2.3. Muita internationalisointiratkaisuja

On selvää, että vakiokirjaston tarjoama internationalisointiratkaisu ei tyydytä kaikkia kehittäjiä. Yleensä vakiokirjaston ulkopuoliset ratkaisut ovat syntyneet tarpeesta käyttää yhdenmukaisia resurssitiedostoja muiden kehitysympäristöjen kanssa. Microsoftin Visual C++:ssa käytetään Windowsin resurssitiedostoja (.RC) etenkin Microsoftin Microsoft Foundation Classes (MFC) ja Active Template Library (ATL) -kirjastoon perustuvien sovellusten internationalisoimiseen. Resurssitiedostoon voidaan sijoittaa mm. viestiluetteloita (StringTable), ikoneita, UI-komponentteja ym. Resurssitiedostojen muokkaamiseen on Visual Studiossa tähän tarkoitukseen tehty resurssieditori. Resurssit käännetään erilliseen DLL-kirjastoon, josta sovellus lataa tarvitsemiaan resursseja ajonaikana. Lähestymistavan etuna on helppo siirrettävyys eri Windows-kehitysympäristöjen vä-

lillä, mutta vaikeuksia tulee, jos on tarkoitus portata sovellus muihin käyttöjärjestelmiin.

Avoin monialustainen wxWidgets-kirjasto on suunniteltu helpottamaan helposti siirrettävien sovellusten luomista. Periaatteessa kirjasto mahdollistaa sovelluksen kääntämisen samalla koodipohjalla eri alustoilla.[2] Kirjaston käyttö mahdollistaa erilaisten internationalisointiratkaisujen hyödyntämisen. wxWidgetsissä on oliopohjaiset toteutukset C-tyyppisille paikanteille sekä domain-käsitettä tukevalle gettext-tyylisten viestikirjastojen käsittelyrutiineille. Näiden käyttö noudattelee samaa prosessia kuin C-versiossa, ja wxWidgets-versio voi suoraan hyödyntää jo käännettyjä .po-resursseja. Lisäksi wxWidgetsissä on oma, Microsoftin RC-tiedostoja muistuttava XML-pohjainen resurssitiedostojärjestelmä (XRC). XRC-resurssit voivat Microsoftin vastineensa tapaan sisältää sekä tekstejä että käyttöliittymäresursseja. Tyypillisesti XRC-resurssit ladataan dynaamisesti, mutta ne on myös mahdollista kääntää binääritiedostoksi tai C++-koodiksi, jolloin resurssit voidaan staattisesti linkittää ajettavaan tiedostoon. XRC-resurssien käsittelyyn on olemassa sekä kaupallisia että vapaita työkaluja. Kirjaston mukana tulee myös työkalu, joka kääntää Windowsin .RC-tiedostoja XRC-resursseiksi. (Smart *et al.*, 2007)

3.3. C# ja .NET

.NET on Microsoftin tuote- ja teknologiaperhe, joille yhteistä on Microsoft .NET Framework -kehyksen käyttäminen. Teknologian ytimenä on Common Language Runtime (CLR) -virtuaalikoneen käyttö. CLR:n etuina suoraan konekoodin tuottamisen sijasta ovat mm. helpotettu muistin ja muiden resurssien käyttö (roskienkeruu), säietoteutukset, poikkeushallinta ja turvallisuusominaisuudet. Vaikka CLR-tavukoodin tuottamiseen on eri ohjelmointikieliin perustuvia ratkaisuja, on C#-oliokieli näistä omimmillaan .NET-ympäristössä ja niiden kehitys on kulkenut käsi kädessä. Microsoftin Visual Studio 2005 –kehitysympäristöistä on eri ohjelmointikieliin perustuvat versiot, ja Microsoft on laajentanut kieliä niiden standardisyntaksin ulkopuolelle saadakseen paremmin tuettua CLR:n ominaisuuksia. Microsoftin CLR:lle kohdistettu versio C++-kielestä on nimeltään C++/CLI ja vastaava Javan laajennus on J#. Näiden lisäksi kielenä voidaan käyttää Microsoftin Visual Basicin .NET-versiota (VB.NET).

.NET-kehyksessä on huomioitu internationalisointivaatimuksen ottamalla käyttöön laaja-alainen paikannejärjestelmä, jota .NET-ympäristössä kutsutaan kulttuureiksi (*cultures*). Kulttuurien käsittelyssä keskeinen luokka on *Sys-*

---

[2] Ns. ”Write once, compile anywhere”-periaate. Vertaa Javan ylioptimistinen slogan “Write once, run anywhere”.

tem.Globalization.CultureInfo. Multilingualisointi on huomioitu siten, että jokaisella säikeellä voi olla oma oletuskulttuurinsa. .NET-kehyksen resurssimalli on melko samanlainen kuin muissa Windows-ympäristöissä. Ainoa merkittävä ero on, että .NET käyttää erityisiä koodikirjastoja, joita kutsutaan nimellä *assembly.* Vain resurssikäyttöön tarkoitettuja erityisiä assemblyjä kutsutaan satelliiteiksi (*satellite assembly*). Satelliittiin ei ole mahdollista sisällyttää suoritettavaa koodia, eikä Fusion-koodilataaja huomioi satelliitteja ollenkaan. Sen sijaan satelliittien sisältämät resurssit ladataan ajonaikaisesti *System.Resources.ResourceManager*-luokkaa käyttämällä. (Singh, 2003)

3.4. Symbian

Symbian OS on yleinen avoin käyttöjärjestelmä kannettaville laitteille, kuten älypuhelimille ja kämmentietokoneille. Symbian OS mahdollistaa ohjelmistojen kehittämisen tiettyä C++-osajoukkoa käyttäen. Koska kohdelaitteistot ovat lähtökohtaisesti globaaleille markkinoille suunnattuja laitteita, on internationalisointinäkökohdat pyritty huomioimaan Symbian OS:n ja erityisesti S60-alustan kehityksessä. Pääosa internationalisointituesta löytyy AVKON-komponentista. Näistä tärkeimmät luokat ovat *TLocale,* joka sisältää paikannetuen tarvitsemia funktioita, sekä *CharConv*, joka vastaa tuettujen koodausten välisistä muunnoksista. Lisäksi AVKON tarjoaa syöttömetodieditorituen useille eri syöttötavoille, joista osa on kulttuurispesifisiä (esim. kiinan PinYin ja ZhuYin-syöttötavat). Ohjelmoijan tarvitsee vain luoda tarvittava editorikomponentti; AVKON huolehtii siitä, että nämä tukevat tarvittavia syöttötapoja. (Nokia 2006, 18-23)

Lokalisoitavien resurssien osalta Symbian tukee pääsääntöisesti kahdentyyppisiä resursseja, merkkijonoja ja grafiikkatiedostoja, joille on omat työkaluketjunsa resurssien kääntämiseksi asennuspakettiin tai ROM-imagoon meneviksi binäärimuodoiksi. Merkkijonot määritellään ihmisenluettavissa RLS-resurssitiedostoissa, jotka käännetään *StringLoader*-luokan ymmärtämään RSC-muotoon. Grafiikkatiedostot käännetään useita bittikarttoja sisältäviin binääritiedostoihin (aiemmissa Symbian-versioissa .mbm, S60 3.1:stä lähtien myös .mif). Yleiskäyttöisten MBM-bittikarttojen lisäksi Symbianilla on erityinen formaatti sovellusikoneita varten (Application Information File, AIF). (Edwards *et al.*, 2004, 42-44)

Lisäksi on huomattava, että merkkijonokäsittely Symbian OS:ssä perustuu ns. deskriptoreihin. C-tyyliset nollaloppuiset merkkijonot eivät ole sallittuja Symbian-koodissa. Deskriptorit ovat merkkitaulukkoja monipuolisempia ja turvallisempia mm. puskuriylivuotojen suhteen. Ohjelmoijan käytössä on muokattavia ja ei-muokattavia, sekä pinosta että keosta varattavia deskriptorityyppejä. (Symbian, 2007)

Symbian OS:n internationalisointiominaisuuksia on käsitelty mm. Käpyahon (2001) ja Zhaon (2003) pro gradu -töissä. Symbian OS:n käyttöalue poikkeaa jossakin määrin muista ohjelmointiympäristöistä. Koska sen internationalisointiominaisuudet eivät ole helposti vertailtavissa muihin, niin niiden yksityiskohtainen tarkastelu jää tämän työn ulkopuolelle.

3.5. Java

Javassa internationalisoitavuus on ollut alusta lähtien yksi johtavista periaatteista. Unicodea tuetaan paitsi tulosteissa, myös siten, että Java-lähdekoodissa on mahdollista käyttää muitakin alfanumeerisia merkkejä tunnisteina kuin ASCII-merkkejä. Javassa on myös C++:n paikannetta muistuttava paikanteen käsite. Java sallii usean eri paikanteen käyttämisen samassa sovelluksessa, mikä mahdollistaa monikielisten sovellusten tekemisen.

Javassa on myös resurssien käsittelymekanismi, jonka resurssikäsite tunnetaan nimellä ResourceBundle. ResourceBundleihin voi tallettaa mielivaltaisia luokkia edustavaa dataa. Resursseille, jotka sisältävät vain merkkijonomuotoista informaatiota, voi käyttää tekstitiedostoissa kuvattavia ResourceBundleja. Muille tietotyypeille käytetään ListResourceBundle-luokan periviä Java-luokkia.

Java-ympäristöjä on kirjoitushetkellä kolmenlaisia. Java Standard Edition (Java SE) on tarkoitettu yleisimpiin käyttötarkoituksiin. Java Enterprise Edition (Java EE) on tarkoitettu monikerrosarkkitehtuuriin perustuville järjestelmille lähinnä yritysympäristössä. Java Micro Edition (Java ME) on tarkoitettu pienitehoisille järjestelmille, kuten älypuhelimille, kämmentietokoneille ja digisovittimille. Tässä tutkimuksessa tarkastellaan ainoastaan Java Standard Editionin versiota 6. Käpyahon pro gradu -työssä on käsitelty Java Micro Editionin internationalisointiominaisuuksia.

4. Tutkimusmetodi

4.1. Viitesovellus

Tutkimus on pääosin kirjallisuuteen perustuva vertailututkimus, jossa lasketaan myös joitakin kvantitatiivisia tunnuslukuja. Jotta tutkimus vastaisi mahdollisimman paljon todellisen maailman tarpeita, käytetään internationalisointiin vaikuttavien ominaisuuksien arvioimisessa viitekehyksenä osittain fiktiivistä, osittain todellista sovellusta, jolla on internationalisointia edellyttäviä vaatimuksia. Viitesovelluksen avulla määritetään ne toiminnot, jotka ovat tarpeellisia tyypillisen sovelluksen internationalisoinnin toteuttamiseen. Jokaisesta ohjelmointikielestä ja kirjastosta pyritään erotamaan tätä rajapintaa vastaava osajoukko vertailua varten.

Viitesovelluksen internationalisointivaatimukset ovat:

- Lokalisoitava UI-tekstiluettelo
- Paikannekohtaisesti mukautettavat UI-komponentit (sisältäen grafiikkaresurssit)
- Käsiteltävä data sisältää eri merkistökoodauksia, sisältäen multibyte- ja Unicode-koodauksia
- Sovellus käsittelee aikaleimoja (päivämääriä ja kellonaikoja)
- Sovellus sisältää lajittelutoimintoja
- Sovellus käsittelee useamman kuin yhdenkielistä dataa samassa istunnossa
- Sovellus sisältää IME-tukea edellyttäviä toimintoja.

Todellisena viitesovelluksena käytetään avoimen lähdekoodin FTP-asiakasohjelmaa nimeltä FileZilla 3.0.0. FileZilla perustuu wxWidgets-kirjastoon. Pääasiallisena ohjelmointikielenä on käytetty C++:aa, mutta ohjelmisto sisältää myös C-kielellä toteutetun komponentin (PuTTY-pääteohjelma). (Filezilla, 2007)

FileZillan viestiluettelo perustuu wxWidgetsin gettext-toteutukseen, ja muut UI-resurssit on toteutettu XRC-metodilla. Näitä tekniikoita tarkastellessa tukeudutaan FileZillan toteutukseen, muissa tekniikoissa käytetään viitteenä fiktiivistä näillä tekniikoilla toteutettua FileZillan vastinetta. Viiterajapintojen operaatioiden valinnassa oletetaan, että tarkoitus on käyttää samaa paikannetta "globaalisti", kuten FileZillan ja wxWidgetsin paikannejärjestelmä toimii, huolimatta siitä mahdollistaako ko. ympäristö esimerkiksi paikanteen asettamisen säiekohtaisesti.

4.2. ISO 9126-standardista

ISO 9126 on kansainvälinen standardi ohjelmistojen arviointikriteereille. Standardi koostuu neljästä osasta: laatumallista, ulkoisista metriikoista, sisäisistä metriikoista ja quality in use -metriikoista. Laatumalli (ISO 9126-1) sisältää useita kategorioita, jotka edelleen jakautuvat pienempiin alikategorioihin. Pääkategoriat ovat toiminnallisuus, luotettavuus, käytettävyys, tehokkuus, ylläpidettävyys ja siirrettävyys. Standardissa määritellyt alikategoriat jakautuvat vielä attribuutteihin, joilla tarkoitetaan yksittäisiä mitattavissa olevia laatuun vaikuttavia ohjelmiston ominaisuuksia. Standardi ei määrittele näitä ominaisuuksia, sillä attribuuteiksi soveltuvat ominaisuudet vaihtelevat ohjelmistotyypistä toiseen. Seuraavassa pyrimme määrittelemään standardia soveltaen attribuutteja, joilla voidaan mitata ohjelmointikielten ja -työkalujen soveltuvuutta internationalisoitavien ohjelmistojen kehittämiseen. Lisäksi pitää huomata, että standardi on tarkoitettu yleiseksi kaikenlaisten ohjelmistojen arviointikehykseksi, eivätkä kaikki osat ole sovellettavissa ohjelmointityökaluihin. Näistä poikkeavuuksista mainitaan seuraavassa käsittelyssä.

4.3. Toiminnallisuus

ISO 9126 määrittelee toiminnallisuuden seuraavasti:

> A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs. (ISO 9126: 1991, 4.1)

Internationalisoitavan ohjelmiston kehitystyökalun toiminnallisuutta kuvaavat attribuutit määrittyvät siis internationalisointitarpeiden kautta. Tarpeiden määritystä varten tehdään tyypillisesti vaatimusanalyysiä. Käytämme näiden tarpeiden määritykseen Esselinkin tarkistuslistaa. Kansainväliseksi tarkoitetun ohjelmiston kehittämiseen ovat tarpeellisia seuraavat toiminnot (Esselink, 2000, 28-29):

- Paikannetuki
- Kalenterituki
- Multi-byte enabled
- Lajittelu paikanteen mukaan
- Koodisivu- / merkistötuki. Tähän kuuluvat koodaustuen lisäksi muunnokset eri merkistöjen välillä.
- IME-tuki
- Monikielisyystuki (multilingualisation)

Paikannejärjestelmien toiminnallisuuden laajuutta määritetään kahden ominaisuuden perusteella: toisaalta millä tarkkuudella paikanne on erotettavista (asteikolla kieli-maa-variantti), toisaalta miten paikanteen vaikutuspiiri on

määritettävissä. Laajin paikanteen vaikutus alue on ns. globaali (sovelluksen laajuinen) paikanne, tarkimmillaan paikanne voidaan määrittää säie- tai oliokohtaisesti.

Edelleen näiden toiminnallisuuksien toteutusta voimme tarkastella standardin määrittelemien alikategorioiden tasolla. Alikategoriat ovat soveltuvuus, tarkkuus, yhteentoimivuus, säännönmukaisuus ja turvallisuus.

Soveltuvuudella (*suitability*) tarkoitetaan attribuutteja, jotka mittaavat toiminnallisuuksien soveltumista määritettyihin tehtäviin. Tarkkuudella (*accuracy*) tarkoitetaan attribuutteja, jotka mittaavat sitä, miten hyvin toiminnallisuuden tulokset vastaavat odotuksia. Soveltuvuuden ja tarkkuuden määrittämisen edellyttämät vaatimusmääritykset olisivat laajemman tutkimuksen aihe, ja jätetään tämän tarkastelun ulkopuolelle.

Yhteentoimivuus (*interoperability*) tarkoittaa attribuutteja, jotka määrittävät ohjelmiston kykyä toimia yhdessä muiden järjestelmien kanssa. Säännönmukaisuus (*compliance*) tarkoittaa attribuutteja, jotka kuvaavat sitä, miten hyvin ohjelmisto noudattaa standardeja, konventioita ja muita sääntöjä. Tutkimuksessa tarkastellaan sekä hyväksyttyjä standardeja että de facto -standardeja. Paikannejärjestelmistä tutkitaan, noudattavatko ne RFC 3066 -standardia, joka määrittelee kielten kaksi- ja kolmikirjaimiset tunnukset. Viestiluetteloista gettext- ja catgets-tyyppiset viestiluettelot ovat muodostuneet de facto-standardeiksi, kuten myös Javan properties-notaatio. Näiden de facto -standardien noudattaminen on taulukoitu omana tekijänään.

ISO 6129 –standardin viimeinen toiminnallisuuden alikategoria on turvallisuus. Internationalisointiominaisuudet eivät suoraan vaikuta ohjelmiston turvallisuuteen, joten tätä kategoriaa ei tarkastella tutkimuksessa.

4.4. Luotettavuus ja käytettävyys

Standardi määrittelee luotettavuuden osa-alueiksi kypsyyden, virheensietokyvyn ja toipumiskyvyn. Nämä tekijät ovat pitkälti toteutustason seikkoja, joita ei tässä tutkimuksessa huomioida.

ISO 6129 määrittelee käytettävyyden seuraavasti:

> *A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.* (ISO 9126: 1991, 4.3)

Käyttäjänä on tässä yhteydessä käsitettävä internationalisoitavan ohjelmiston loppukäyttäjän sijasta ohjelmoija, joka käyttää ohjelmointikieltä tai -työkalua ja toisaalta lokalisoija, joka käsittelee resurssitiedostoja. On myös huomattava, että käytettävyys laajemmassa merkityksessään sisältää myös te-

hokkuuden, joka ISO 9126 -terminologiassa käsitellään omana kategorianaan, eikä näin ollen sisälly käytettävyyskategoriaan.

Käytettävyyden attribuutteina käytetään järjestelmän ymmärtämiseen, oppimiseen ja käyttämiseen tarvittavaa työmäärää. Voitaneen sanoa, että näitä työläys on verrannollinen eri hallittavien asioiden määrään. Ohjelmointikielten internationalisointiominaisuuksien osalta seuraavia hallittavia asioita voidaan tarkastella kvantitatiivisesti:

- Tarvittavien kirjastojen tai osasovellusten määrä
- Tarvittavien funktioiden määrä
- Rajapintojen kompleksisuus (parametrien lukumäärä).

Rajapinnan kompleksisuuden mittarina käytetään funktiopisteanalyysiin kuuluvaa unadjusted function point count (UFC) –metriikkaa.

UFC-arvo lasketaan kaavalla:

$$UFC = \sum((N_i) \times p_i),$$

jossa N_i on luokan i tekijöiden lukumäärä ja p_i painotusarvo luokalle i. Painoarvot ja tekijöiden luokat on määritelty Taulukon 1 mukaisesti.

	Painotuskerroin		
Tekijä	**Yksinkertainen**	**Keskinkertainen kompleksisuus**	**Kompleksinen**
Ulkoiset syötteet	3	4	6
Ulkoiset tulosteet	4	5	7
Ulkoiset kyselyt	3	4	6
Ulkoiset tiedostot	7	10	15
Sisäiset tiedostot	5	7	10

Taulukko 1. UFC-metriikan painotuskertoimet

Yksinkertaisuuden vuoksi oletamme, että kaikki tekijät ovat kompleksisuusluokaltaan keskinkertaisia. Ulkoisia kyselyjä ei ole, sillä tarkastelemamme internationalisointirajapinnat ovat ei-interaktiivisia. Täten saamme UFC-arvon laskukaavaksi seuraavan:

UFC = 4A + 5B + 10C + 7D ,

jossa:

A = Ulkoisten syötteiden (parametrien) lukumäärä

B = Ulkoisten tulosteiden (paluuarvojen) lukumäärä

C = Ulkoisten tiedostojen lukumäärä

D = Sisäisten tiedostojen lukumäärä.

Tämä metriikka on valittu sen yksinkertaisuuden vuoksi, ja siksi että tarkasteltavana on sekä oliopohjaisia että ei-oliopohjaisia tekniikoita. Tavallisesti

UFC-metriikasta lasketaan funktiopistearvo kertomalla arvo toisella metriikalla nimeltä technical complexity factor (TCF). (Fenton and Pfleeger, 1997)

Yksinkertaisuuden vuoksi oletamme tässä tutkimuksessa, että eri internationalisointiratkaisujen tekninen kompleksisuus on riittävän lähellä toisiaan, että voimme olettaa TCF-arvon olevan vakio. Todellisella TCF:n arvolla ei ole tämän tutkimuksen kannalta merkitystä, koska metriikkaa käytetään tässä vain eri ratkaisujen kompleksisuuden vertailuun, eikä varsinaisiin laajuuslaskelmiin, kuten yleensä funktiopisteitä laskettaessa.

Lisäksi näihin tekijöihin vaikuttavan kvalitatiiviset asiat, kuten se, miten helposti internationalisointiominaisuudet ovat käsitteellisesti ymmärrettävissä ja miten paljon ratkaisut poikkeavat muista ohjelmoijan käyttämistä rajapinnoista ja sovelluksista. Näiden kvalitatiivisten attribuuttien määrittely on tämän tutkimuksen puitteissa hankalaa, joten ne jätetään tarkastelun ulkopuolelle.

Sebesta (1999) käyttää ohjelmointikielten käytettävyydelle kahta käsitettä: luettavuus (*readability*) ja kirjoitettavuus (*writability*). Luettavuudella tarkoitetaan sitä, miten helppoa ihmisen on koodia tulkita. Kirjoitettavuudella tarkoitetaan sitä, miten helppoa ohjelmoijan on määritellä haluamansa ongelma käyttäen kielen syntaksia. Luettavuuteen vaikuttavat kielen yksinkertaisuus ja ortogonaalisuus, kontrollirakenteet, tietotyypit ja -rakenteet ja syntaksi.

Ortogonaalisuudella tarkoitetaan sitä, että kielessä suhteellisen pieni määrä primitiivirakenteita voidaan yhdistää suhteellisen pienellä määrällä eri tapoja toisiinsa. Näistä yhdisteistä muodostuvat kielen monimutkaisemmat kontrolli- ja tietorakenteet. Toinen tapa ajatella ortogonaalisuutta on se, että ortogonaalisuusaste on kääntäen verrannollinen kielen rakenteissa tarvittavien poikkeussääntöjen määrään. Tarkastelun yhteydessä pohditaan mahdollisia ortogonaalisuuspuutteita tai liiallisen ortogonaalisen aiheuttamia ongelmia, mikäli sellaisia havaitaan.

Kontrollirakenteilla ei ole suurta merkitystä internationalisoinnin kannalta, joten ne jätetään tarkastelun ulkopuolelle. Tietotyypeistä suurin vaikutus internationalisointiin on eri merkkijonotyypeillä. Tutkimuksessa käsitellään mahdollisia tietotyyppien vaikutuksia luettavuuteen, mikäli sellaisia havaitaan. Samaten havainnoidaan mahdollisia kielen syntaksin luettavuutta huonontavia vaikutuksia.

Kirjoitettavuuteen vaikuttavat kaikki samat tekijät kuin luettavuuteen, ja näiden lisäksi kielen abstraktiotuki ja ilmaisuvoima. Abstraktiolla tarkoitetaan tässä kykyä kuvata korkeamman tason rakenteita yksinkertaisemmilla rakenteilla, jolloin yksityiskohtia ei tarvitse toistaa rakenteita käytettäessä. Tutki-

muksessa selvitetään abstraktiotukeen ja rakenteiden ilmaisuvoimaan liittyviä asioita, joilla on vaikutusta internationalisointiin. (Sebesta, 1999, 8-17)

4.5. Tehokkuus

ISO 9126:n määritelmä tehokkuudelle on

> *A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.* (ISO 9126: 1991, 4.4)

Tehokkuuden osa-alueiksi standardissa nimetään ajallinen käyttäytyminen ja resurssikäyttäytyminen. Ajallisella käyttäytymisen attribuutteja ovat tyypillisesti vasteaika, prosessointiaika ja yleinen suoritusteho. Resurssikäyttäytymisellä tarkoitetaan ohjelmiston käyttämien resurssien määrää kuvaavia attribuutteja. Standardi määrittelee resurssit varsin laajasti, mutta tässä tutkimuksessa keskitytään laitteistoresursseihin (muistitila ja levytila). Sekä ajallisen käyttäytymisen että resurssikäyttäytymisen attribuutteina käytetään suhdetta internationalisoidun ja internationalisoimattoman ohjelmiston välillä. Koska internationalisoidun ja vastaavan ei-internationalisoidun ohjelman välistä suhdetta on usein vaikea määritellä, tutkitaan tässä suhteessa vain yksinkertaisesti määriteltävissä olevat tapaukset, kuten lokalisoidut näyttötekstit vs. lokalisoimattomat näyttötekstit (dynaaminen lataus viestiluettelosta vs. kovakoodaus). Nämä mittaukset suoritetaan wxWidgetsin tapauksessa käytettävästä viitesovelluksesta, muutoin kullakin kirjastolla toteutetun minimaalisen "Hello World"-ohjelman lokalisoidusta ja lokalisoimattomasta versiosta (liitteessä 2). Lokalisoitu versio mitataan käyttämällä ohjelman oletuspaikannetta (ns. C- tai POSIX-paikanne, ts. amerikanenglanti). Resurssikäyttäytymisen arvioinnissa pyritään huomioimaan sekä staattinen että dynaaminen resurssikuorma.

4.6. Ylläpidettävyys ja siirrettävyys

ISO 9126 määrittelee ylläpidettävyyden seuraavasti:

> *A set of attributes that bear on the effort needed to make specified modifications.* [- -] *Modifications may include corrections, improvements or adaptation of software to changes in environment, and in requirements and functional specifications.* (ISO 9126: 1991, 4.5)

Ylläpidettävyyden alikategoriat ovat analysoitavuus, muutettavuus, vakaus ja testattavuus.

Useimmat ylläpidettävyystekijät ovat suoraan kytköksissä käytettävyystekijöihin. Siksi ylläpidettävyystekijänä mainitaan ainoastaan, onko vakio-ohjelmointiympäristössä erityisiä ylläpidettävyyttä helpottavia työkaluja.
ISO 9126 määrittelee siirrettävyyden seuraavasti:

> *A set of attributes that bear on the ability of software to be transferred from one environment to another.* [- -] *The environment may include organizational, hardware or software environment.* (ISO 9126: 1991, 4.6)

Tässä tutkimuksessa ympäristö ymmärretään kapeasti laitteisto- käyttöjärjestelmä- ja ohjelmointikieliympäristönä. Siirrettävyydellä määritetyt alikategoriat ovat mukautettavuus, asennettavuus, säännönmukaisuus ja korvattavuus.

Siirrettävyyden arvioimiseksi tutkimuksessa selvitetään seuraavia kysymyksiä:

- Onko internationalisointikoodi siirrettävissä eri ympäristöön?
- Ovatko resurssitiedostot siirrettävissä eri ympäristöön?
- Jos eivät, miten suuria muutoksia edellytetään?

5. Ohjelmointiympäristöjen vertailu

Seuraavassa esitetään vertailtujen ohjelmointiympäristöjen ominaisuudet aiemmin esiteltyä arviointikehystä vasten arvioituna. Tutkimuksen tuloksista on yhteenveto liitteessä 1 ja aikakäyttäytymismittauksessa käytetyt ohjelmakoodit ovat liitteessä 2.

5.1. wxWidgets

wxWidgetsin paikannetuen voidaan katsoa olevan varsin laaja. Koska kirjasto pohjautuu C++-kieleen, se perii sekä C-tyyppiset paikanteet (*locale.h*) että C++:n locale-facetit ja määrittelee näiden lisäksi vielä oman paikannetoteutuksen (wxLocale). Puhdasverisessä wxWidgets-ohjelmoinnissa on suositeltavaa käyttää viimeksimainittua siirrettävyyden säilyttämiseksi. wxLocalen käyttö mahdollistaa myös pysymisen puhtaasti olio-ohjelmointiparadigmassa. Lisäksi wxLocale-luokka kapseloi kätevästi sekä locale.h-moduulin tarjoamat palvelut että gettext-tyylisten viestiresurssien lataamisen. On tosin mainittava, että wxWidgetsin mukana ei toimiteta itse gettext-työkaluja .po-tiedostojen käsittelyyn, joten nämä on hankittava erikseen mikäli käytössä oleva käyttöjärjestelmä ei valmiiksi sisällä gettext-pakettia. (Smart *et al.,* 2007) Tämä ei kuitenkaan ole iso ongelma, sillä kuten mainittu, gettext-metodi nauttii jonkinasteisen de facto-standardin asemaa, joten saatavilla olevia työkaluvaihtoehtoja on runsaasti.

Kalenterituki wxWidgetsin 2.8.5-versiossa ei ole vielä kovin laaja. Kirjasto tarjoaa täyden tuen gregoriaaniselle kalenterille ja hyvin rajallisen tuen juliaaniselle kalenterille. Monet kalenterifunktiot ottavat parametrina kalenterityypin, mutta useimmat näistä toimivat vain gregoriaaniselle kalenterille. Esimerkkinä sekä gregoriaanista että juliaanista kalenteria tukevasta metodista mainittakoon wxDateTime::IsLeapYear, joka kertoo onko määrätty vuosi kar-

kausvuosi ko. kalenterijärjestelmässä. Lisäksi kirjastossa on vakioita, jotka sisältävät tiedon siitä, minä päivänä kussakin maassa siirryttiin juliaanisesta kalenterista gregoriaaniseen. Ajanlaskentaan liittyvien funktioiden lisäksi käyttöliittymäluokkien kalenterikomponentti on rajoittunut gregoriaaniseen kalenteriin.

FileZilla-sovelluksessa on paljon aikaleimojen käsittelyyn tarvittavaa koodia. Sovelluksen on sopeuduttava siihen, että FTP-palvelin saattaa käyttää lokalisoituja esitysmuotoja aikaleimoissaan. Tämä johtuu siitä, että toisaalta FTP-protokollan määrittelevä RFC 959 ei määrittele missä muodossa aikaleima tulee palauttaa, toisaalta siitä, että aikaleimoja ei alunperin suunniteltu koneella luettavaksi. Oletuksena riittää se, että FTP-ohjelmaa käyttävä ihminen ymmärtää aikamerkinnät[3]. wxWidgetsin aikaleimojen jäsentämisfunktiot eivät riitä kaikkien näiden eksoottisten ajan ja päivämäärän esitysmuotojen jäsentämiseen, joten kehittäjät ovat päättäneet kirjoittaa täysin oman luokan aikaleimojen parsimiseen. Sisäisesti aikaleimat käsitellään UNIX-standardin mukaisesti sekuntikokonaislukuarvoina (muodossa jossa stat()-järjestelmäkutsu aikaleiman palauttaa). Näin ollen aikaleimojen käsittelyssä ei käytetä lainkaan wxDateTime-luokan internationalisointiominaisuuksia.

wxWidgetsin Unicode-tuki riippuu sen alla toimivan alustan Unicode-tuesta. Kirjastosta saattaa olla käytössä joko ANSI-versio tai Unicode-versio. Tämän vuoksi kirjaston merkki- ja merkkijonotyypeille on määritelty natiivit tyypit wxString ja wxChar, sekä vastaaville literaaleille makro wxT(). Nämä tulkitaan läpinäkyvästi joko ANSI- tai Unicode-merkeiksi ja -merkkijonoiksi riippuen siitä, onko Unicode-tuki käytössä. Muita multibyte-koodauksia varten on olemassa muunnosfunktioita, esimerkiksi wxString-olion *str* luominen UTF-8-koodattua dataa sisältävästä merkkijonosta *input_data* onnistuu kopiorakentimella: `wxString str(input_data, wxConvUTF8);`

WxWidgetsin vertailufunktiot palautuvat pohjalla olevan C-kirjaston merkkityyppien vertailufunktoreihin, joten paikannekohtaisuus riippuu näiden funktoreiden paikannekohtaisuudesta. Niinikään kirjastossa ei ole aktiivista IME-tukea, passiivinen tuki riippuu käytössä olevasta ikkunointijärjestelmästä. Unicode-kirjastoa käytettäessä tekstikomponentit osaavat näyttää kaksisuuntaisen tekstin oikein, kunhan teksti sisältää kirjoitussuunnan merkitseviä ohjaus-

[3] Tämä on hyvä esimerkki tapauksesta, jossa lokalisointi toimii käytettävyyttä huonontavana tekijänä. Uudempi, ehdotetun standardin asteella oleva RFC-määrittely RFC 2640 tarkentaa muita FTP:n internationalisointinäkökohtia, erityisesti merkistötukea, mutta ei tarjoa ratkaisua tähän ongelmaan. Lisäksi olisi vaarallista olettaa, että kaikki palvelimet tukevat muuta kuin RFC 959:n mukaista FTP-protokollaa.

merkkejä. Esimerkki 1 on Filezillan tilaikkunan toteutuksesta. Tietyt lokiviestityypit sisältävät oletusarvoisesti englanninkielistä tekstiä. Ohjelmoija on itse huolehtinut siitä, että tilaikkunan tekstiin lisätään kirjoitussuuntamerkit tämäntyyppisten viestien eteen. m_rtl-lippu on alustettu tiedustelemalla aktiivisen laitekontekstin (*wxDC*-luokka) käytössä oleva tekstisuunta.

```
if (m_rtl)
      {
            // Unicode control characters that control reading direction
            const wxChar LTR_MARK = 0x200e;
            //const wxChar RTL_MARK = 0x200f;
            const wxChar LTR_EMBED = 0x202A;
            //const wxChar RTL_EMBED = 0x202B;
            //const wxChar POP = 0x202c;
            //const wxChar LTR_OVERRIDE = 0x202D;
            //const wxChar RTL_OVERRIDE = 0x202E;

            if (messagetype == Command || messagetype == Response || mes-
sagetype >= Debug_Warning)
            {
                  // Commands, responses and debug message contain English
text,
                  // set LTR reading order for them.
                  prefix += LTR_MARK;
                  prefix += LTR_EMBED;
                  lineLength += 2;
            }
      }
```

Esimerkki 1. FileZilla-ohjelmiston kaksisuuntaisen tekstin hallintakoodia

Kirjastossa on osittain tietoisesti luovuttu yhteentoimivuudesta siirrettävyyden hyväksi. Windowsin resurssitiedostojen käyttäjälle kirjastossa on edellämainittu muunto-ohjelma, jolla on kuitenkin rajoituksensa. XRC-resurssiformaattiin muunto ei edes teoriassa voi toimia täydellisesti, sillä XRC on resurssijärjestelmänä hieman suppeampi kuin Microsoftin resurssijärjestelmä. XRC-tiedostot eivät esimerkiksi voi sisältää itsenäisiä viestiluetteloita (vrt. Microsoftin string table-tyyppi), vaan merkkijonot ovat aina jonkin wxWidgets-olion XRC-vastineen attribuutteina.

wxWidgets käyttää paikanteiden nimeämiseen itse määrittelemäänsä enumeraatiota *wxLanguage*, esimerkiksi suomenruotsin paikanne on wxLANGUAGE_SWEDISH_FINLAND ja järjestelmän oletuspaikanne wxLANGUAGE_DEFAULT. wxLanguage-paikanne on palautettavissa standardinmukaiseksi RFC 3066 -nimeksi (wxLocale::GetCanonicalName), sekä päinvastoin (wxLocale::FindLanguageInfo). wxWidgets viittaa myös viestiluetteloihin sisäisesti standardimuotoisella viitteellä. Kuten edellä mainittiin, vies-

tiluetteloina käytetään de facto -standardin (.po) mukaisia tiedostoja. Kirjasto on siis näiltä osin varsin säännönmukainen sekä hyväksyttyihin standardeihin että de facto -standardiin nähden.

WxWidgets voidaan rakentaa joko monoliittisenä tai ns. Multilib-buildinä, jolloin voidaan linkittää mukaan vain tarvittavat osakirjastot ja koko sovelluksen koko pysyy pienempänä. Kirjaston keskeisin osakirjasto on wxBase, jota vasten kaikkien wxWidgets-pohjaiset sovellukset on linkitettävä. Viestiluettelojen käsittely sisältyy wxBase-kirjastoon, joten mikäli ohjelmoija ei aio käyttää XRC-resurssijärjestelmää, hän selviää samoilla osakirjastoilla, joita muutenkin tarvitsee sovelluksen linkittämiseen. Jos XRC-resurssit ovat käytössä, tarvitaan lisäksi wxXRC-osakirjasto sekä wxXML-, wxCore-, wxAdvanced- ja wxHTML-kirjastot, joista edellämainittu on riippuvainen. WxWidgetsin yleisenä ongelmana on, että kirjastot ovat isokokoisia ja valmiista sovelluksesta tulee varsin isokokoinen. Minimaalinenkin wxWidgets-sovellus on helposti megatavun kokoluokkaa, ja XRC-resurssien vaatimien kirjastojen mukaan linkittäminen pahentaa ongelmaa entisestään.

Filezillan lähdekoodiin lisättiin ajanmittauskoodia viestiluettelojen aikakäyttäymisen mittaamiseksi kahteen kohtaan: Viestiluettelot lataavan LoadLocales()-funktion ympärille, sekä koodiin, joka alustaa asetusdialogin kaikki sivut. Jälkimmäisessä tapauksessa testi toistettiin kahdesti siten, että tekstit haettiin ensin ladatusta viestiluettelosta ja toisella kerralla sivut alustettiin kovakoodatuista merkkijonoista (_()-makrot vaihdettiin _T()-makroiksi). Mittauskoodissa käytettiin wxWidgetsin ajanlaskentaluokkia, jotka toimivat millisekuntin tarkkuudella. Mittauksien tarkkuus on siis yksi millisekunti. Viestiluettelojen lataamisajaksi saatiin 12 ms ja sivujen alustamisajaksi 2 ms kummallakin alustustavalla. Näistä absoluuttisesta ajoista voidaan jo huomata, että käyttöliittymäkomponenttien lokalisoinnilla on hyvin pieni vaikutus normaalikokoisen sovelluksen suoritusaikaan.

Filezillan internationalisointiin on käytetty Taulukon 2 funktioita:

Funktio /metodi	**Tehtävä**
wxLanguageInfo* wxLocale::FindLanguageInfo (int language)	Palauttaa kieli-informaatiotietueen käytössä olevasta UI-kielestä.
wxLocale::wxLocale()	wxLocale-olion oletusrakennin. Ei alusta paikannetta, tätä varten pitää kutsua vielä Init()-metodia.
bool wxLocale::Init (int language, int flags)	Asettaa paikanneolion oletuspaikanteeksi ja muuttaa sovel-

	luksessa käytössä olevan kielen.
bool wxLocale::IsOk()	Palauttaa tiedon siitä, onnistuiko paikanteen asettaminen
void wxLocale::AddCatalogLookupPathPrefix (const wxString& prefix)	Lisää viestiluetteloiden juurihakemiston hakupolkuun.
wxLocale::~wxLocale()	wxLocale-olion oletuspurkaja. Palauttaa oletuspaikanteen ja sovelluksen kielen edellistä Init()-kutsua edeltävään tilaan.
int wxLocale::GetLanguage()	Palauttaa paikannetta vastaavan kielen.
const wxChar* ::wxGetTranslation (const wxChar* str)	Käytetty _(str)-makron kautta. Palauttaa lokalisoidun version merkkijonosta, tai alkuperäisen merkkijonon, jos käännöstä ei löydy ladatuista viestiluetteloista.
wxXmlResource* wxXmlResource::Get()	Palauttaa osoittimen globaaliin resurssiolioon ja luo sen, jos sitä ei ole vielä olemassa (vrt. Ainokainen-suunnittelumalli)
int wxXmlResource::GetFlags()	Palauttaa resurssiolion liput: wxXRC_USE_LOCALE: resurssijärjestelmä käyttää _()-makroja viestiluettelon käännösten lataamiseen wxXRC_NO_SUBCLASSING: resurssitiedostojen oliosolmujen aliluokkaominaisuus jätetään huomioimatta
void wxXmlResource::SetFlags(int flags)	Asettaa resurssiolion liput (ks. yllä).
void wxXmlResource::AddHandler (wxXmlResourceHandler* handler)	Alustaa määrätyn resurssikäsittelijän.
bool wxXmlResource::Load (const wxString& filemask)	Lataa tiedostomääritystä filemask vastaavat resurssitiedostot.

wxDialog* wxXmlResource::LoadDialog (wxWindow* parent, const wxString& name)	Lataa dialogiolion nimetystä resurssitiedostosta ja asettaa sen emoikkunan.
wxMenu* wxXmlResource::LoadMenu (const wxString& name)	Lataa valikko-olion nimetystä resurssitiedostosta.
wxMenuBar* wxXmlResource::LoadMenuBar (const wxString& name)	Lataa valikkopalkkiolion nimetystä resurssitiedostosta.
wxToolBar* wxXmlResource::LoadToolBar (wxWindow* parent, const wxString& name)	Lataa työkalupalkkiolion nimetystä resurssitiedostosta ja asettaa sen emoikkunan.
wxPanel* wxXmlResource::LoadPanel (wxWindow* parent, const wxString& name)	Lataa paneeliolion ja asettaa sen emoikkunan.
int wxXmlResource::GetXRCID (const wxChar* str_id)	Palauttaa merkkijonossa str_id nimettyä XML-resurssissa käytettyä merkkijonotunnistetta vastaavan numeerisen tunnisteen.
wxCSConv::wxCSConv (wxFontEncoding encoding)	Rakennin, joka ottaa parametriksi merkistön, jonka muunnoksia muunninolio suorittaa.
size_t wxCSConv::MB2WC (wchar_t* buf, const char* psz, size_t n)	Muuntaa multibyte-merkistöä käyttävän puskurin Unicode-puskuriksi. Palauttaa kohdepuskuriin kirjoitettujen merkkien lukumäärän.
size_t wxCSConv::WC2MB (char* buf, const wchar_t* psz, size_t n)	Kuten yllä, mutta muuntaa Unicode-puskurista multibyte-puskuriin.

Taulukko 2. wxWidgets-kirjaston internationalisointirajapinta

wxWidgetsin internationalisointirajapinnat ovat olioparadigman tyyliin helposti ymmärrettäviä ja luettavia. Eri toiminnot on erotettu eri metodeihin.

UFC-laskelmaa varten yllä kuvatusta rajapinnasta saadaan 33 ulkoista syötettä (parametria) ja 19 ulkoista tulostetta (palautusarvoa). Rakentimien palautusarvoksi katsotaan luotu olio, purkajilla ei katsota olevan palautusarvoa. wxCSConv-luokan metodien viiteparametrit katsotaan sekä syötteiksi että tulosteiksi. Järjestelmän tarvitsemat kaksi tiedostotyyppiä, viestiluettelotiedosto ja XRC-resurssitiedosto lasketaan ulkoisiksi tiedostoiksi, sillä niiden tulee olla luettavissa muilla sovelluksilla (esim. resurssieditorit). Sisäisiä tiedostoja ei ole

tunnistettavissa. Tälle rajapinnalle saadaan UFC-arvoksi UFC = 4 * 33 + 5 * 19 + 10 * 2 = 247.

WxWidgetsin viestiluettelojärjestelmä on periaatteessa helposti ymmärrettävissä, etenkin jos gettext-lokalisointimenetelmä on entisestään tuttu. Kovakoodattuihin merkkijonoihin verrattuna erot ovat pieniä, sillä ohjelmoijan pitää vain lisätä _( )-, wxPLURAL( )-, tai _T( )-makro literaalin ympärille. Samoin argumenttien käsittely on C:n printf-funktion käyttäjälle tuttu (johtuen siitä, että wxWidgets käyttää C-rajapintaa suoraan). Sekaannusta saattaa aiheuttaa se, että _( ) on makro wxGetTranslation-metodille ja _T( ) puolestaan ei-lokalisoitavalle merkkijonolle, vaikka makrojen ulkonäöstä saattaisi päätellä päinvastoin. Tätä hämmennystä on mahdollista välttää käyttämällä valmiiksi määriteltyä vaihtoehtoista makroa wxTRANSLATE( ), mutta tämä vaihtoehto puolestaan pidentää koodirivejä merkittävästi.

Edellämainituista kirjastojen kokoasioista johtuen wxWidgets-pohjaiset sovellukset vievät suhteellisen paljon muistitilaa sekä levyllä että muistiin ladattuna. Aikakäyttäytymisessä kirjasto korvaa suurta resurssikuormaa. Koodi on C/C++-koodin tavoin tehokasta, ja wxWidgetsin tehokkuutta lisäävistä ominaisuuksista, kuten viitelaskennasta sekä merkkijonojen tehostusominaisuuksista on lisähyötyä.

Koska wxWidgets perustuu C++-kieleen ja koska monien asioiden määrittelyt perustuvat makroihin, ovat useammat internationalisointiominaisuudet siirrettävissä muihin (ainakin C++-pohjaisiin) järjestelmiin suhteellisen pienillä muutoksilla, esim. sopivilla makromäärittelyillä. Merkkijonojen määrittelyyn käytettävät makrot voidaan määrittää kohdejärjestelmän mukaisiksi, esim. wxT –makrot voitaisiin määritellä std::string –luokan alustuksiksi[4].

5.2. C-kirjasto ja Sunin lokalisointikirjastot

C-standardikirjaston käsittelyssä käytetään viitteenä Sun Microsystemsin Solaris 9 (SunOS 5.9) -version C-kirjastoa. Kirjasto on aavistuksen laajempi kuin standardinmukainen ISO C99-kirjasto, mutta erot ovat niin pieniä, että ei ole mielekästä käsitellä niitä erikseen.

C:n kehittämisajankohta ja painotus systeemiohjelmointiin näkyvät siinä, että vakiokirjastossa ei ole mukana kaikkea tässä tutkimuksessa käsiteltäviä internationalisointiominaisuuksia. Paikannejärjestelmä sisältää kuitenkin oleellisimmat funktiot. Solaris 9:n standardinmukaiset internationalisointifunktiot on siirretty erillisistä kirjastoista libc-kirjastoon. Ohjelmaa ei tarvitse siis enää linkittää eri kirjastoja vasten internationalisoituja sovelluksia kirjoittaessa. Tämä

---

[4] Esimerkki on sikäli triviaali, että wxWidgetsissä siirrytään tulevaisuudessa todennäköisesti käyttämään std::string-tyyppisiä merkkijonoja wxString-luokan sijaan.

tarkoittaa sitä, että internationalisoinnista on tullut normaali tapa toimia sen sijaan, että se olisi poikkeus. Erilliset otsikkotiedostot ovat edelleen olemassa takaperin yhteensopivuuden vuoksi, mutta ne sisältävät vain tynkäviittauksia libc-otsikoihin.

Viestiluettelojen käsittelyyn on Solaris 9:n libc-kirjastossa sekä (d)gettext- että catgets-funktioperheiden rajapinnat. Alkuperäiseen catgets-malliin verrattuna mukana on genmsg-niminen työkalu, joka toimii samaan tyyliin kuin xgettext, eli käy läpi lähdekoodin ja luo löytämiinsä merkkijonoresursseihin perustuvan viestiluettelokaavaimen. Tämä vähentää manuaalisia työvaiheita viestiluettelon valmisteluvaiheessa. (Sun, 2002, 45-47, 52-53) Muuten kaikki tyypilliset catgets- ja gettext-luetteloiden käsittelyyn tarvittavat työkalut ovat Solariksessa mukana.

Kalenterituki löytyy vain gregoriaaniselle kalenterille. Paikannejärjestelmä muokkaa aikojen esitystavan paikanteen mukaisesti. Mitään edistyneempiä päivämäärien käsittelyfunktioita vakiokirjastosta ei löydy.

Paikannefunktioden tapaan myös monitavuisten merkkien käsittelyyn tarkoitetut funktiot on Sunin C-kirjastossa siirretty libc-kirjastoon. ISO/IEC 9899:1990 -standardi määrittelee laajennuksen ANSI C:hen, joka sisältää monitavuisten merkkien käsittelyyn tarvittavat ominaisuudet. Tätä laajennusta kutsutaan myös nimellä MSE (*multi-byte support environment*). MSE mahdollistaa monitavuisten merkkien käsittelyn loogisina yksikköinä käyttäen samaa ohjelmointimallia kuin yksitavuisilla merkeillä. Monille alkuperäisille merkkienkäsittelyfunktiolle on monitavuisia merkkejä tukeva vastine, jonka nimi eroaa alkuperäisestä lisätyllä w-kirjaimella. Esimerkiksi funktio towupper() muuntaa monitavuisen merkin pienistä kirjaimista isoksi kirjaimeksi. (Sun, 2002, 42-44, 48).

Lajittelua varten vakiokirjastossa on strcoll()-funktio, joka vastaa toiminnaltaan vanhempaa strcmp()-funktiota, mutta ottaa huomioon paikanteen lajittelusäännöt. Vaihtoehtoinen tapa toteuttaa paikannekohtainen lajittelu on muuntaa merkkijono paikanteen lajittelusääntöjen mukaan normalisoituun muotoon strxfrm()-funktiolla. strxfrm()-funktiolla muunnettujen vertailu strcmp()-funktiolla tuottaa saman tuloksen kuin vertailu suoraan strcoll()-funktiolla. strxfrm-muunnos voi säästää suoritusaikaa, jos tarvitsee suorittaa useita peräkkäisiä vertailuja.

Merkistömuunnoksiin Sunin vakiokirjastossa on iconv()–rajapinta. Tämä rajapinta oli alunperin HP-UX-käyttöjärjestelmässä, mutta se on nykyisin hyväksytty osaksi C-standardia. (Sun, 2002, 46, HP, 2001, 112)

IME-tuki ei sisälly C-kirjastoon. C-pohjaisissa ohjelmissa tukeudutaan käytössä olevan käyttöjärjestelmän ja ikkunointijärjestelmän IME-rajapintoihin.

Myöskään varsinaisia monikielisyyttä tukevia ominaisuuksia ei ole. Setlocale()-funktio asettaa aina koko sovelluksen globaalin paikanteen, joten monikielisiä sovelluksia tekevän ohjelmoijan on itse huolehdittava paikanteen vaihtamisesta tarpeen mukaan. Setlocale-funktiolle on tosin mahdollista antaa parametri, joka kertoo, mille kategorialle paikanne asetetaan. Tämä mahdollistaa eri paikanteiden käytön eri kategoriassa, esimerkiksi siten, että käytetään englanninkielisiä näyttötekstejä, mutta ajan esitysmuoto on suomalainen.

C-kielen yhteentoimivuutta muiden kielten kanssa voidaan pitää korkeana. C-kirjasto on osajoukkona C++-vakiokirjastoissa ja monille muille kielille ja kirjastoille on C-sidontoja (esim. Python, wxWidgets ja Javan JNI-rajapinta) Samoin yhteentoimivuutta lisää se, että molemmat vakiokirjaston funktioiden ymmärtämät viestiluettelot ovat yleisesti käytettäviä standardeja.

C-paikanteet on nimetty standardinmukaisesti muodossa <ISO639-kielikoodi>_<ISO3166-maakoodi>.<merkistötunnus>. Esimerkiksi fi_FI.ISO8859-15 on Suomea, suomen kieltä ja euro-merkillä laajennettua skandinaavista merkistöä vastaava paikanne.

Internationalisointiin tarvittavien kirjastojen määrä riippuu siitä, mitä C-ympäristöä käytetään. Perinteisemmissä ympäristöissä internationalisointifunktiot ovat erillisissä kirjastoissa (locale, libintl, libw), mutta kuten aiemmin todettiin, nämä on esim. Sunin C-ympäristössä siirretty libc-kirjastoon, joten näitä kirjastoja ei tarvitse linkittää erikseen mukaan.

Tutkimuksesta olleista kielistä C:llä toteutetun sovelluksen suoritusaika oli kaikista nopein. Ero lokalisoimattoman ja C-paikannetta käyttävän version välillä oli varsin pieni (28,14% enemmän). Versio, joka latasi tekstiresurssin levyltä käyttäen fi_FI-paikannetta, käytti aikaa 60,59% enemmän kuin lokalisoimaton versio.

Seuraavassa esitellään C-vakiokirjastosta löytyvät funktiot, jotka suorittavat viitesovelluksessa tehtävät internationalisointioperaatiot. Taulukossa 3 on lueteltu sekä gettext- että catgets-funktioperheiden rajapinta. Näistä käytetään sovelluksessa tyypillisesti vain jompaakumpaa. Koska C-vakiokirjasto ei sisällä resurssijärjestelmää, nämä funktiot puuttuvat taulukosta.

Funktio	**Tehtävä**
char * setlocale (int category, NULL)	Palauttaa osoittimen merkkijonoon, jossa on tämänhetkisen käytössäolevan paikanteen nimi.
char * setlocale (int category, const char *locale)	Asettaa uuden paikanteen. Palauttaa osoittimen merkkijonoon, jossa on käyttöön tulleen paikanteen nimi, tai nollaosoittimen, jos paikanteen aset-

	taminen epäonnistui. (locale != NULL)
char * bindtextdomain (const char *domain_name, const char *dir_name)	Asettaa merkkijonon *domain_name osoittaman käyttöalueen hakupoluksi merkkijonon *dir_name osoittaman hakemiston. Palauttaa osoittimen merkkijonoon, joka kertoo asetetun hakupolun tai nollaosoittimen, jos asettaminen epäonnistui.
char * textdomain (const char *domain_name)	Asettaa sen käyttöalueen, jonka viestiluetteloa käytetään käyttöalueparametrittomien gettext-kutsujen käyttöalueena.
char * gettext (const char *msgid)	Palauttaa käytössä olevan paikanteen ja käyttöalueen mukaisen käännöksen merkkijonolle *msgid, tai alkuperäisen merkkijonon, jos käännöstä ei löydy. Yleensä määritelty makroksi _(msgid)
nl_catd catopen (const char *name, int oflag)	Avaa catgets-tyyppisen viestiluettelon. Palauttaa nl_catd-tyyppisen viestiluettelodeskriptorin, jota käytetään muissa rajapinnan funktioiden kutsuissa, tai (nlcatd) -1, jos avaaminen epäonnistui. Epäonnistuessa errno-muuttujaan asetetaan virhekoodi.
char * catgets (nl_catd catd, int set_num, int msg_num, const char *s)	Palauttaa viestiluettelodeskriptoria catd, kategoriaa set_num ja viestitunnusta msg_num vastaavan käännöksen, tai merkkijonon *s, jos käännöstä ei ole saatavilla.
int catclose (nl_catd catd)	Sulkee viestiluettelon catd. Jos nl_catd on toteutettu tiedostodeskriptorilla, myös tämä suljetaan. Palauttaa arvon 0, jos sulkeminen onnistui tai virhekoodin, jos sulkeminen epäonnistui.
iconv_t iconv_open (const char *tocode, const char *fromcode)	Palauttaa deskriptorin merkistömuuntimelle, joka muuntaa merkki-

	jonon *fromcode kuvaamasta merkistöstä merkkijonon *tocode kuvaamaan merkistöön. Parametrien mahdolliset arvot riippuvat toteutuksesta[5]. Muuntimen avaamisen epäonnistuessa palauttaa arvon (iconv_t) -1 ja asettaa errno-muuttujaan virhekoodin.
size_t iconv(iconv_t cd, const char **inbuf, size_t *inbytesleft, char **outbuf, size_t *outbytesleft)	Muuntaa muunnindeskriptorin cd mukaisten merkistöjen välillä lukien merkkijonosta **inbuf ja kirjoittaen merkkijonoon **outbuf. Palauttaa suoritettujen muunnosten lukumäärän ja päivittää tilamuuttujia osoittimissa *inbytesleft ja *outbytesleft. Jos koko **inbuf-merkkijono voitiin muuntaa onnistuneesti, asetetaan tilamuuttujaan *inbytesleft arvo 0. Muunnos voi myös päättyä ennenaikaisesti muuntimen joutuessa epänormaaliin tilaan esim. jos lukupuskurin **inbuf sisältö ei vastaa ko. merkistössä sallittuja arvoja. Tällöin tilamuuttujaan *inbytesleft asetetaan nollasta poikkeava arvo, ja errno-muuttujaan asetetaan virhekoodi. Varsinaisessa virhetilanteessa palautetaan (size_t) -1 ja errno-muuttujaan asetetaan virhekoodi.
int iconv_close(iconv_t cd)	Sulkee muunnindeskriptorin cd. Sulkemisen onnistuessa palauttaa arvon 0, epäonnistuessa palauttaa arvon -1 ja asettaa errno-muuttujaan virhekoodin.

Taulukko 3. C-vakiokirjaston internationalisointirajapinta

C-kirjaston rajapinta kärsii huonosta luettavuudesta. Monia funktioita käytetään sekä arvojen asettamiseen että lukemiseen antamalla nollaparametri (set-

[5] Katso esim. Sun, 2002, 165-192.

locale(), textdomain(), bindtextdomain()). Osoitintyyppiset parametrit ja palautusarvot toisaalta huonontavat luettavuutta, toisaalta altistavat osoitinsemantiikalle tyypillisille virheille, kuten vain luettavaksi tarkoitettujen arvojen muuttamiseen vahingossa ja puskuriylivuodoille. Erityisesti iconv-rajapinta sisältää paljon osoitintyyppisiä viiteparametrejä. Toisaalta tämä rajapinta antaa ohjelmoijalle mahdollisuuden hallita merkkimuunnoksia tehokkaasti esimerkiksi määrittelemällä haluamansa toipumisstrategian ongelmatilanteisiin.

Kuten aikaisemmin, viiteparametrit (ei-const-tyyppiset osoitintyypit) lasketaan sekä syötteiksi että tulosteiksi. Funktioille, jotka asettavat errno-virhekoodin, on myös tämä laskettu tulosteeksi. Vaikka setlocale-funktion eri kutsutyypit on taulukossa selkeyden vuoksi esitetty erikseen, lasketaan ne UFC-laskelmassa yhdeksi ja samaksi rajapinnaksi. Internationalisointirajapintaan kuuluu yksi ulkoinen tiedosto (viestiluettelo, sekä catgets- että gettext-rajapinnassa). Sisäisiä tiedostoja ei ole erotettavissa. Gettext-tyyppistä viestiluetteloa käytettäessä rajapinnassa on 15 ulkoista syötettä ja 15 ulkoista tulostetta. Catgets-tyyppistä viestiluetteloa käytettäessä ulkoisia syötteitä on 18 ja ulkoisia tulosteita 15. Täten rajapinnan UFC-arvoksi saadaan gettext-rajapintaa käytettäessä UFC = 4 * 15 + 5 * 15 + 10 * 1 = 145 ja catgets-rajapintaa käytettässä UFC = 4 * 18 + 5 * 15 + 10 * 1 = 157.

Myös C-kirjaston internationalisointirajapinta on suhteellisen helposti omaksuttavissa. Sen hankaluudet ovat samoja kuin yleisesti C-ohjelmoinnissa yleensä, joten tärkein uusi omaksuttava asia on lokalisointimakrojen käyttö kovakoodattujen merkkijonoliteraalien sijasta. Tähänkin edellä mainittu xgettext-työkalu tuo helpotusta.

Internationalisoitujen C-ohjelmien siirrettävyyttä helpottaa se, että etenkin gettext-viestiluettelojen käsittelyyn sisältyvät työkalut ja funktiot löytyvät käytännöllisesti katsoen jokaisesta ohjelmointiympäristöstä. Gettext-työkalut osaavat lisäksi muuntaa .PO-resurssit suoraan Javan properties-tiedostoksi.

5.3. C++

Kuten edellä wxWidgetsin yhteydessä mainittiin, C++-vakiokirjasto sisältää C-kirjaston ja sen internationalisointirajapinnan. C++-kirjastossa on kuitenkin oma, elegantimpi oliopohjainen paikannejärjestelmä. C++-paikannetta voidaan tarkastella säiliöluokkana, joka sisältää mielivaltaisen kokoelman facet-luokkia. Kukin facet määrittelee tietyn paikannekohtaisen toiminnon, kuten viestiluettelon käsittelyn, päivämäärämerkintöjen muotoilun jne. Paikanneolioita voidaan luoda joko C-paikanteiden nimien perusteella tai kopioimalla oletuspaikanne locale::classic(), joka määrittelee amerikanenglantilaisen ASCII-ympäristön. Paikanneoliot ovat ei-muutettavia, joten tarvittavat facetit täytyy määritellä oliota muodostettaessa. Tämä paikanteiden ominaisuus tekee niistä myös hy-

vin turvallisia käyttää. Facet-luokkia ei ole tarkoitus instantioda suoraan, vaan käyttämällä use_facet<>-kaavainta. Tämän lisäksi kirjastossa on has_facet<>-kaavain, jolla voi tutkia onko tietty facet-luokka käytettävissä tietyssä paikanteessa. Viimeksimainittu on käytännössä tarpeeton, sillä kaikki vakiokirjaston paikanteet sisältävät kaikki vakiofacetit. (RWS, 1996, 1.4)

Viestiluetteloiden toteutusta varten on standardissa määritelty messages-facet. Facet on varsin löyhästi määritelty (Kosnik, 2001). Rajapinta osoittaa, että facet on primäärisesti ajateltu catgets-tyyppisten viestiluettelojen käsittelyyn, mutta käytännössä on mahdollista määritellä messages-facet käyttäen mitä tahansa viestiluettelotekniikkaa. Eri C++-toteutuksissa on yleensä messages-facet catgets- ja gettext-tyyppisille viestiluetteloille.

C++-vakiokirjaston ajankäsittelyoperaatiot eivät sisällä tukea eri kalenterijärjestelmille, ainoastaan gregoriaaniselle kalenterille, ja niiden voidaan katsoa olevan puutteellisia jopa tällä käyttöalueella. Eräs pahimmista ortogonaalisuuden puutteista on se, että time_get- ja time_put-facetien syötteet ja tulosteet eivät ole keskenään yhteensopivia. (Garland, 2005, 15-16)

Merkistömuunnoksia varten kirjastossa on codecvt<wchar_t, char, mbstate_t> -facet, johon on kapseloitu samantyyppiset merkkimuunnosoperaatiot kuin C-kirjaston iconv()-funktioissa.

Paikanneherkkä merkkijonovertailu on C++-vakiokirjastossa niinikään toteutettu tähän erikoistuneen facet-luokan avulla. Facet on prototyypiltään std::collate <CharT>. Facetin julkinen rajapinta sisältää käytännössä C:n strcoll() ja strxfrm() -funktioiden toiminnallisuuden ja näiden lisäksi jäsenfunktion, joka palauttaa merkkijonon hajautuskoodin.

C++-vakiokirjaston paikanteet perustuvat C-paikanteisiin ja näin ollen niiden nimeäminen noudattaa samaa standardinmukaista linjaa.

Paikannejärjestelmän käyttö ei edellytä ylimääräisten kirjastojen linkitystä, sillä paikanneluokka sisältyy vakiokirjastoon. Tämän huomion jälkeen on hyvä todeta, että C++-vakiokirjasto (etenkin STL:ää käytettäessä[6]) on varsin suuri (n. megatavun luokkaa). Runsas kaavainten käyttö kasvattaa kohdekoodin kokoa. [uSTL, 2007] On huomattavaa, että em. facetien käyttö edellyttää kahden sisäkkäisen kaavaimen käyttöä. Toisaalta tämä malli säästää ajonaikaisia resursseja mm. siten, että kaavaimeen perustuvat operaatiot on muodostettu staattisesti kääntämisen yhteydessä.

Aikakäyttäytymisen mittaamiseen käytetty ohjelma käännettiin täsmälleen samassa laite- ja ohjelmistoympäristössä sekä samalla kääntäjällä kuin yllämai-

---

[6] Vakio-STL:n korvaajaksi ollaan kehittämässä paremmin optimoituja vaihtoehtoja, ks. esim. [uSTL, 2007].

nittu C-versio. Siksi oli hieman yllättävää, että ohjelman absoluuttinen suoritusaika oli yli kaksinkertainen vastaavaan C-ohjelmaan. Sen sijaan erot lokalisoitujen ja ei-lokalisoidun version välillä olivat huomattavasti pienempiä. C-paikannetta käyttävä versio ohjelmasta käytti aikaa vain 1,75% enemmän kuin lokalisoimaton. Messages-facetia ja fi-FI-paikannetta käyttävä versio käytti vain 24,20% enemmän suoritusaikaa kuin lokalisoimaton ohjelma.

Myöskään C++-vakiokirjastossa ei ole muunlaisten resurssien kuin merkkijonojen käsittelyyn tarkoitettuja luokkia. Teoriassa resurssijärjestelmä voitaisiin toteuttaa lisäämällä paikanteisiin resurssijärjestelmästä vastaava facet-luokka, mutta useimmat kirjastonkehittäjät ovat päätyneet toisenlaiseen ratkaisuun. Taulukossa 4 listataan ne C++-vakiokirjaston internationalisointifunktiot, jotka vastaavat viiterajapintaa. Yhdenmukaisesti viiterajapinnan kanssa taulukossa on käytetty nimettyjen paikanteiden xxx_byname-faceteja. Oletuspaikanteessa käytetään vastaavaa facetia ilman _byname-päätettä. Merkittävää on facetien jäsenfunktioiden samankaltaisuus vastaavien C-funktioiden kanssa. Koska standardien (ISO / ANSI) määrittelemässä rajapinnassa ei ole resurssikäsittelyfunktioita, on seuraavaan taulukkoon lisätty Microsoftin Win32-APIn viiterajapintaa vastaavat funktiot. Irrallisia menuresursseja käyttäessään ohjelmoijan on huolehdittava niiden hävittämisestä kutsumalla DestroyMenu-funktiota. Menuresursseissa, jotka kuuluvat toiseen menuun tai pääikkunalle, ei ole tätä huolta, sillä ne hävitetään rekursiivisesti emokomponentin hävittämisen yhteydessä.

Metodi	**Tehtävä**	**Saatavuus**
std::locale::locale (const char * s)	Paikanneluokan rakennin, joka luo merkkijonon *s mukaista C-paikannetta vastaavan paikanneolion. Heittää poikkeuksen std::runtime_error, mikäli nimetty paikanne ei kelpaa.	ISO / ANSI, Visual C++
const std::Facet& std::use_facet<std::messages_byname<class charT> >	Palauttaa viitteen facet-luokkaan, joka käsittelee charT-tyyppisistä merkeistä (käytännössä char tai w_char) koostuvia viestiluetteloita.	ISO / ANSI, Visual C++
catalog std::messages_base::open(const basic_string <char>& fn, const locale& loc)	Palauttaa viestiluettelodeskriptorin, joka vastaa merkkijonossa fn nimettyä viestiluetteloa. Loc viittaa siihen paikanteeseen, jonka codecvt<>-facetia käyte-	ISO / ANSI, Visual C++

	tään muuntamaan tarpeelliset merkistömuunnokset. Jos nimettyä luetteloa ei ole saatavilla, palauttaa arvon -1.	
string_type std::messages_base::get (catalog c, int set, int msgid, const string_type & dfault)	Palauttaa viestiluettelodeskriptoria c, kategoriaa set ja viestitunnusta msgid vastaavan käännöksen, tai merkkijonon dfault, jos käännöstä ei ole saatavilla.	ISO / ANSI, Visual C++
void std::messages_base::close (catalog c)	Sulkee deskriptoria c vastaavan viestiluettelon.	ISO / ANSI, Visual C++
const std::Facet& std::use_facet<std::codecvt_byname<class internT, class externT, class stateT> >	Palauttaa viitteen facet-olioon, joka muuntaa sisäisen tyypin internT ja ulkoisen tyypin externT välillä ja tallettaa muuntimen tilan stateT-tyyppiseen muuttujaan.	ISO / ANSI, Visual C++
result std::codecvt_byname::in(state_type &state, const extern type *from, const extern_type *from_end, const extern_type*& from_next, intern_type *to, intern_type *to_limit, intern_type*& to_next)	Muuntaa merkkejä ulkoisesta tyypistä sisäiseen tyyppiin puskurista, joka sijaitsee välillä from-from_end ja sijoittaa nämä puskuriin, joka alkaa osoittimesta *to siten, että *to_limit-osoitinta ei ylitetä. Muunnoksen (täydellisen tai epätäydellisen) jälkeen osoittimet *&from_next ja *&to_next osoittavat viimeksi onnistuneesti muunnettua merkkiä seuraavaan merkkiin. Palautusarvo on jokin seuraavista: ok (kaikki lähdepuskurin merkit muunnettiin onnistuneesti), partial (muunnos tarvitsisi enemmän tilaa kuin mitä kohdepuskurissa on), error (lähdepuskurissa olevaa merkkiä ei voitu muuntaa merkistömuunnoksen mukaisesti), no-	ISO / ANSI, Visual C++

	conv (sisäinen ja ulkoinen tyyppi ovat samat, joten muunnosta ei tarvittu)	
result std::codecvt_byname::out(state_type& state, const intern_type *from, const intern_type *from_end, const intern_type*& from_next, extern_type *to, extern_type *to_limit, extern_type*& to_next)	Muuntaa merkkejä sisäisestä tyypistä ulkoiseen tyyppiin, ks. yllä.	ISO / ANSI, Visual C++
HMODULE WINAPI LoadLibrary(__in LPCTSTR lpFileName)	Palauttaa modulikahvan DLL-kirjastoon, joka on nimetty merkkijonossa lpFileName	Visual C++
HMENU LoadMenu(HINSTANCE hInstance, LPCTSTR lpMenuName)	Lataa merkkijonon lpMenuName osoittaman menu-resurssin modulikahvan hInstance osoittamasta resurssitiedostosta.	Visual C++
BOOL DestroyMenu(HMENU hMenu)	Tuhoaa menuresurssin hMenu ja sen mahdolliset alimenut.	Visual C++
DWORD WINAPI FormatMessage(__in DWORD dwFlags, __in LPCVOID lpSource, __in DWORD dwMessageId, __in DWORD dwLanguageId, __out LPTSTR lpBuffer, __in DWORD nSize, __in va_list* Arguments)	Hakee viestitunnistetta dwMessageId ja kielitunnistetta dwLanguageId vastaavan merkkijonoresurssin modulikahvan lpSource osoittamasta resurssitiedostosta, sijoittaa mahdolliset argumentit argumenttilistasta Arguments merkkijonon muotoilukoodeihin ja kopioi lopputuloksen puskuriin lpBuffer, jonka maksimikoko on nSize. dwFlags sisältää useita lippuja, jotka muuttavat funktion toimintaa. Resurssitiedostoja käsitellessä täytyy lipun FORMAT_MESSAGE_FROM_HMODULE olla päällä.	Visual C++

Taulukko 4. C++-vakiokirjaston internationalisointirajapinta

C++-vakiokirjaston paikannejärjestelmä ei ole kovin helppo lukea eikä kirjoittaa. Runsas kaavainten ja makrojen käyttö tekee siitä vaikeaselkoista. Jopa Stroustrup itse myöntää, että paikannejärjestelmän yksityiskohtainen kuvaus on ainoastaan kokeneiden C++-ohjelmoijien ymmärrettävissä (Stroustrup, 2004):

> "Please note that I still consider this detailed description of locales beyond the needs of most C++ programmers. It is written with experienced programmers in mind and novices will do best to avoid it. Even experienced C++ programmers will find parts of this appendix [Appendix D: Locales, The C++ Programming Language (special Edition)] hard to read."

Windowsin resurssirajapinta on sitä vastoin aavistuksen selkeämpi. wxWidgetsiin verrattuna voidaan huomata, että käyttöliittymäresurssien lataaminen suoritetaan aina samalla funktiolla (LoadMenu), kun wxWidgetsissä jokaisella komponenttityypillä on oma metodinsa. Toisaalta tämä yksinkertaisuus latausvaiheessa siirtyy mutkikkaampaan resurssien käsittelyyn myöhemmissä vaiheissa.

UFC-laskelmaa varten teemme samat oletukset kuin aiemmissa kohdissa. Viiteparametrit lasketaan sekä syötteiksi että tulosteiksi. Tulosteihin lasketaan paluuarvojen lisäksi heitetyt poikkeukset. Lisäksi kaavainten parametrit lasketaan syötteisiin. Ulkoiset tiedostotyypit ovat käytännössä samat kuin wxWidgets-rajapinnassa: viestiluettelo ja resurssitiedosto. Sisäisiä tiedostoja ei ole erotettavissa. Syötteitä rajapinnassa on näin ollen 40 ja tulosteita 17. Saadaan siis: UFC = 4 * 40 + 5 * 17 + 10 * 2 = 265.

5.4. C# ja .NET

Kuten aikaisemmin mainittiin, C#-ohjelmointikieli on kehittynyt yhtä kättä .NET-kehyksen kanssa. Tietyllä tavalla .NET-kehystä voidaan siis pitää C#:n vakiokirjastona. Tarkemmin sanottuna vakiokirjasto on ns. Base Class Library (BCL) –kirjasto, joka sisältää .NET-kehyksen nimiavaruudet poislukien Microsoft.* -alkuiset nimiavaruudet. Tässä kappaleessa käsitelty rajapinta on pätevä myös muissa Microsoftin CLI-laajennetuissa ohjelmointikielissä ko. kielen syntaksiin sovitettuna, mutta esimerkeissä käytetään C#:n syntaksia. Itse asiassa, C# ei tue kaikkia .NET-kehyksen ominaisuuksia, joita pystyy käyttämään esimerkiksi C++/CLI:stä (Stroustrup, 2003), mutta näillä eroilla ei ole merkitystä internationalisointiominaisuuksien kannalta.

.NET-kehykselle on ominaista, että monille perinteisille sovelluskehitykseen liittyvälle termille on keksitty uusi omaperäinen nimi. Paikanteen sijasta .NET-terminologiassa puhutaan kulttuurista. Kyse on kuitenkin täysin samasta

asiasta. Seuraavassa käsittelyssä käytämme termiä "paikanne" käyttöjärjestelmän käyttämästä paikanteesta ja termiä "kulttuuri", kun puhutaan .NET-kehyksen sisäisistä rakenteista. Kulttuurijärjestelmä kiteytyy CultureInfo-luokkaan. Kulttuurit on nimetty RFC-3066 -standardin mukaisesti. .NET-kehyksessä voidaan käyttää kolmenlaisia kulttuureita. Näistä yksinkertaisin on invarianttikulttuuri, joka itse asiassa edustaa internationalisoimatonta toiminnallisuutta. Invarianttikulttuuri on tarkoitettu käytettäväksi tilanteissa, joissa pitää varmistaa, että yleensä kulttuurisidonnainen toiminnallisuus, kuten lajittelu, toimii yhdenmukaisella tavalla käytössä olevasta paikanteesta huolimatta. Tällaisia kriittisiä tapauksia voivat olla esimerkiksi turvallisuusominaisuudet. Kulttuuria, johon liittyy kieli, mutta ei maata, kutsutaan termillä "neutraali kulttuuri", ja kulttuuria, johon liittyy sekä kieli että maa, kutsutaan "spesifiseksi kulttuuriksi". Tällä hierarkialla on merkitystä resursseja ladatessa. Tiettyä resurssia ladatessa yritetään ensiksi ladata spesifiselle kulttuurille määritelty resurssi. Mikäli tätä ei ole saatavilla, yritetään seuraavaksi ladata spesifisen kulttuurin kielikomponenttia vastaava neutraalin kulttuurin resurssi. Mikäli tätäkään ei ole saatavilla, turvaudutaan sovelluksen sisäänrakennettuihin resursseihin. Tietyt internationalisointiominaisuudet, kuten päiväysten ja kellonaikojen muotoilu ja lukujen muotoilu, sisältyvät ainoastaan spesifisiin kulttuureihin. Kulttuuri on mahdollista asettaa säiekohtaisesti, mikä yksinkertaistaa monikielisten sovellusten laatimista.

.NET-kehyksessä resurssijärjestelmä on kokonaan uusittu. Resurssimalli perustuu ns. satellittiassemblyihin, joka tunnetaan myös ”napa ja puola” -mallina. Tässä mallissa oletusresurssit ovat upotettuna pääassemblyyn, joka sisältää myös suoritettavan koodin. ”Pyörän kehällä” ovat satelliittiassemblyt, jotka sisältävät ainoastaan resursseja, eikä suoritettavaa koodia. Resurssiassemblyjen välillä vallitsee samanlainen hierarkkinen suhde kuin kulttuurienkin välillä: mikäli satelliittiassemblyä ei ole käytettävissä, takaudutaan oletusresurssien käyttöön. Resurssijärjestelmä on myös muodollisesti uusittu, eikä se ole takaperin yhteensopiva Microsoftin vanhemman resurssijärjestelmän kanssa. Resurssitiedostot luodaan (melkein) ihmisen luettavassa XML-formaatissa, jolle käytetään tiedostopäätettä .resx. ResX-muoto muistuttaa tietyssä mielessä wxWidgetsin XRC-järjestelmää, sillä sekin on XML-perustainen ja voi sisältää mielivaltaisia UI-resursseja sekä merkkijonoresursseja. Toinen vaihtoehto on käyttää raakatekstimuotoa (.txt), mutta tällaisiin resurssitiedostoihin voidaan sijoittaa ainoastaan merkkijonoresursseja. Nämä ihmisenluettavat formaatit käännetään binääriformaattiin resgen-työkalun avulla. Käännöksen tuloksena syntynyttä .resources-päätteistä tiedostoa voidaan käyttää sellaisenaan .NET-

sovelluksesta käsin, mutta tyypillisin tapa on linkittää .resources-tiedostot (satelliitti)assemblyihin.

.NET-kehyksen kalenteritukijärjestelmä on vertailun laajin. Gregoriaanisen kalenterin lisäksi versiossa 2.0 tuettuna ovat juliaaninen, heprealainen, persialainen, hijri-, Um-Al-Qura-, japanilainen, korealainen, taiwanilainen ja thaibuddhist-kalenterit sekä aurinko-kuu-vaihtoehdot kaakkoisaasialaisille kalenterijärjestelmille. Erityistä .NETin kalenterijärjestelmässä on, että se määrittelee vuotta pidemmän yksikön nimeltä aikakausi (*era*). Aikakausilla on suurin merkitys japanilaisessa kalenterissa. Muissa kalenterijärjestelmissä aikakaudet saattavat olla määriteltynä, mutta nykyinen toteutus tukee kuitenkin vain kuluvaa aikakautta (esim. gregoriaanisessa kalenterissa vain vuodet jälkeen ajanlaskun alun).

.NET-kehys tukee Unicodea, tarkemmin sanottuna merkkijonojen natiivi muoto on UTF-16. Joissakin tapauksessa käytetään sisäisesti UTF-8:aa. Muilla merkistöillä koodatun tekstin muuntamista varten kehyksessä on luokka System.Text.Encoding. Unicodelle, UTF-8:lle, UTF-7:lle ja ASCII:lle on määritelty erityinen erikoistus Encoding-luokasta, ja 8-bittisiä koodisivuja varten saa luotua Encoding-luokan ilmentymiä GetEncoding-metodilla.

Myös .NETin vertailufunktioiden internationalisointituki on laaja. Kulttuurisidonnaiset lajittelusäännöt on toteutettu System.Globalization.CompareInfo-luokassa, joka sisältyy CultureInfo-luokkaan. Muutamassa spesifisessä kulttuurissa (etenkin kiinassa) on olemassa rinnakkaisia lajittelusääntöjä. Näissä .NET-kehys mahdollistaa valinnan oletus- ja vaihtoehtoisen lajittelujärjestyksen välillä.

.NET-kehyksen lomakekomponentit tukevat kiinan, japanin ja korean syöttömetodieditorin hallintaa jossakin määrin. Komponenteilla on ImeMode-ominaisuus, jonka avulla voidaan esim. estää syöttömetodieditorin käyttö määrätyissä komponenteissa kokonaan tai asettaa editori tiettyyn tilaan, kun komponentti aktivoidaan.

Kaksisuuntaisen tekstin tuki .NET-kehyksessä on hieman rajallista. Käyttöliittymäkomponentit tukevat kaksisuuntaisen tekstin käyttöä, mutta mikäli komponenttien oletustoiminta ei miellytä, ei kaksisuuntaisen tekstin käsittelyalgoritmeihin pääse käsiksi, sillä ne on rajapinnassa määritelty yksityisiksi metodeiksi. Halutessaan ohjelmoija pystyy kiertämään suojauksen käyttämällä reflektiota. Esimerkiksi halutessaan kirjoittaa oman bidi-algoritmin on ohjelmoijan tarpeellista selvittää onko tietty merkki kaksisuuntaisuuden kannalta neutraali, vasemmalta oikealle tai oikealta vasemmalle kirjoittavaan kirjoitusjärjestelmään kuuluva, erotinmerkki jne. Tällainen metodi, nimeltään Sys-

tem.Globalization.CharUnicodeInfo.GetBidiCategory on kehyksessä yksityisenä, ja sen käyttöönotto edellyttää esim. seuraavanlaista reflektiokoodia:

```
Type typeCharUnicodeInfo = Ty-
pe.GetType("System.Globalization.CharUnicodeInfo");

BindingFlags bf = BindingFlags.NonPublic |
BindingFlags.Static | BindingFlags.Instance |
BindingFlags.InvokeMethod;

MethodInfo getBidiCategory =
typeCharUnicodeInfo.GetMethod("GetBidiCategory",
bf);
```

Tämän jälkeen metodi on kutsuttavissa metodin System.Reflection.MethodInfo.Invoke() kautta. (Kaplan, 2007)

Koska .NET-sovellukset perustuvat tavukoodiin eivätkä natiiviin konekoodiin, ovat ne riippuvaisia .NET-kehyksen ajokirjaston asennuksesta. Ajokirjaston koko on melko suuri; versio 2.0 vaatii levytilaa 280 MB (x86-versio) tai 610 MB (x64-versio). System.Globalization-nimiavaruuden luokkien osuus kirjastosta on vain hieman yli 100 kilotavua. Teknisesti on mahdollista linkittää myös .NET-sovellus monoliittisesti sisältäen vain tarvittavat osat kirjastosta, mutta tämä ei onnistu Microsoftin vakiotyökaluilla.

C#-kielinen .NET-ohjelma suoriutui aikakäyttäytymismittauksesta vertailussa olleista kirjastosta absoluuttiselta arvoltaan hitaiten. Ohjelman suoritus kesti Java-vastinettaan yli 3 kertaa kauemmin ja miltei 200-kertaisesti vertailun nopeimpaan C-kielellä toteutettuun natiivisovellukseen. Hitaus selittyy sillä, että .NET-sovellukset ajetaan virtuaalikoneessa, jonka muodostaminen ja purkaminen vie suurimmat osan suoritusajasta. Lokalisoidun ja lokalisoimattoman sovelluksen suhteellinen ero jää siksi testissä hyvin pieneksi.

Taulukossa 5 on listattu .NET-kehyksen viiterajapintaa vastaava rajapinta. Kaikki metodit ovat julkisia, joten lyhyyden vuoksi metoditunnisteesta on jätetty avainsana ”public” pois. C#:lle tyypillistä on ylikuormittaa olennaisten jäsenmuuttujien asetus ja palautus (set/get-metodit), jolloin niihin on mahdollista viitata ominaisuuksien tapaan (piste-notaatio). On myös syytä huomata, että rajapinnassa ei ole mainittu purkajia, koska C#:ssa on automaattinen roskienkeruu.

Metodi	Tehtävä
System.Globalization.CultureInfo.CultureInfo (string culture)	CultureInfo-luokan oletusrakennin. Luo nimettyyn kulttuuriin perustuvan CultureInfo-olion. Rajapinnassa on myös vaihtoehtoinen rakennin, joka ottaa parametrikseen Int32 -tyyppisen kulttuuritunnisteen, mutta tämän käyttöä ei suositella, sillä se ei tue kustomoituja kulttureja.
static CultureInfo System.Globalization .CultureInfo.InstalledUICulture { get; }	Palauttaa käyttöjärjestelmän asennettua paikannetta vastaavan CultureInfo-olion.
CultureInfo System.Threading.Thread .CurrentCulture { get; set; }	Asettaa tai palauttaa säikeeseen liittyvän oletuskulttuurin.
virtual string System.CultureInfo.Name { get; }	Palauttaa kulttuurin nimen muodossa <ISO639-kielikoodi>-<ISO3166-maa/aluekoodi>[7]
System.Resources.ResourceManager (string baseName, Assembly assembly)	ResourceManager-luokan rakennin, joka luo resurssinhallintaolion, joka käyttää nimettyyn nimipohjaan perustuvia resursseja nimetylle assemblylle. Tyypillisesti halutaan luoda resurssinhallintaolio käynnissä olevalle assemblylle. Tämän voi tehdä yksinkertaisesti reflektion avulla: `ResourceManager resmgr = new ResourceManager ("OmaSovellus.OmaResurssi", System.Reflection.Assembly .GetExecutingAssembly());`
virtual string System.Resources.ResourceManager .GetString (string name, CultureInfo culture)	Palauttaa viestitunnistetta name vastaavan ja kulttuurille culture lokalisoidun merkkijonon viestiluettelosta

[7] Tämä metodi vastaa käyttötarkoitukseltaan eniten wxWidgetsin wxLocale::GetLanguage()-metodia. CultureInfo-luokassa on myös muita metodeja, jotka palauttavat kulttuurin kielen nimen englanniksi tai joko .NET-kehyksen asennuskielen tai nykyisen säikeen oletuskulttuurin kielellä.

	(satelliittiassemblystä). Jälkimmäisen parametrin voi jättää pois, jolloin käytetään suoritettavan säikeen oletuskulttuuria. Mikäli viestin lataaminen ei onnistu, palautetaan nollaviite.
virtual ResourceSet System.Resources .ResourceManager.GetResourceSet (CultureInfo culture, bool createIfNotExists, bool tryParents)	Palauttaa ResourceSet-olion, joka edustaa kaikkia kulttuurin culture lokalisoituja resursseja resurssihallintaolion hallinnoimassa resurssitiedostossa. Bool-tyyppiset liput määrittävät, ladataanko ResourceSet, jos sitä ei ole vielä olemassa, sekä voidaanko resurssijoukon sijasta ladata sen edeltäjä (Parent-ominaisuus). Mikäli annetuilla asetuksilla minkään resurssijoukon lataaminen tai palauttaminen ei onnistu, palautetaan nollaviite.
virtual Object System.Resources.ResourceSet .GetObject(string name)	Palauttaa merkkijonossa name nimetyn resurssiolion resurssijoukosta.
virtual Decoder System.Text.Encoding .GetDecoder()	Palauttaa koodausta vastaavan purkajan, joka muuntaa kyseisestä merkistöstä .NET-merkkijonoiksi. Metodia kutsutaan Encoding-luokan aliluokalle, joka edustaa kyseistä merkistökoodausta.
virtual void System.Text.Decoder.Convert (byte* bytes, int byteCount, char* chars, int charCount, bool flush, out int bytesUsed, out int charsUsed, out bool completed)	Muuntaa puskurissa bytes sijaitsevia tavuja maksimissaan parametrin byteCount osoittaman määrän .NET-merkeiksi. Muunnetut merkit tallennetaan puskuriin chars. Parametri charCount ilmaisee maksimimäärän merkkejä, jota voidaan käyttää kohdepuskurissa. Flush-lippu kertoo muuntimelle, onko kyseessä viimeinen Convert-metodin kutsu, jolloin muunnoksen jälkeen Decoder-olio palautetaan alkutilaan. Metodi palauttaa muunnoksen tilasta arvon kolmessa viiteparametrissa. Parametriin

	riin bytesUsed tallennetaan lähdepuskurista kulutettujen tavujen määrä. Parametriin charsUsed tallennetaan kohdepuskuriin lisättyjen merkkien määrä. Parametri completed asetetaan todeksi, jos muunnettiin onnistuneesti parametrin byteCount ilmaisema määrä tavuja.
virtual Encoder System.Text.Encoding .GetEncoder()	Kuten GetDecoder(), mutta palauttaa Encoder-olion, joka muuntaa .NET-merkeistä tavuiksi.
virtual void System.Text.Encoder.Convert (char* chars, int charCount, byte* bytes, int byteCount, bool flush, out int charsUsed, out int bytesUsed, out bool completed)	Kuten Decoder.Convert(), mutta muuntaa .NET-merkeistä tavuiksi.

Taulukko 5. .NET-kehyksen internationalisointirajapinta C#-kielessä

.NET-kehyksen internationalisointirajapinnassa on onnistuttu säilyttämään sekä hyvä luettavuus että kirjoitettavuus. Tämä on pitkälti C#-kielen syntaksin ansiota. Ominaisuus-syntaksin ansiosta on mahdollista kirjoittaa intuitiivisia lausekkeita, jotka muistuttavat olion julkisten kenttien suoraa käsittelyä muissa ohjelmointikielissä, kuitenkin säilyttäen olioparadigman turvallisuuden, kapseloinnin ja toteutuksen piilotuksen. Paikannejärjestelmä on yksityiskohtaisempi kuin vanhemmissa ohjelmointiympäristöissä, joten hallittavia asioita on periaatteessa enemmän. Useimmissa tapauksissa kuitenkin spesifisten kulttuurien käyttö riittää, ja tämä muistuttaa muiden ympäristöjen paikanteiden käyttöä.

UFC-laskelmassa on tehty seuraavat oletukset: Syötteitä ovat kaikki parametrit (myös viiteparametrit) ja ”setterit”. Tulosteita ovat palautusarvot, viiteparametrit ja ”getterit”. Ulkoisia tiedostoja on vain yksi: satelliittiassembly, joka sisältää sekä lokalisoidut merkkijonot että muuntyyppiset resurssit. Koska Decoder- ja Encoder-luokat ovat toistensa semanttisia vastakappaleita, on näistä laskettu vain toinen. Täten UFC-arvoksi saadaan UFC = 4 * 18 + 5 * 11 + 10 * 1 = 137.

Resurssijärjestelmä on mitä ilmeisemmin tarkoituksella tehty .NET-spesifiseksi. Tekstiresurssit on mahdollista kääntää resgen-työkalulla .NET-resurssitiedostosta .txt-muotoon, josta on yksinkertaista muuntaa tekstiresurssi muihin viestiluettelojärjestelmiin (esim. Gettext-tyyppisiksi .po-viestiluetteloiksi).

5.5. Java

Java SE sisältää omanlaisensa oliopohjaisen paikannejärjestelmän. Paikannejärjestelmä perustuu tyypilliseen RFC 3066 –tyyliseen paikanteiden määrittämiseen kielen, maan ja variantin perusteella. Java sisältää .NET-kehyksen tapaan mahdollisuuden luoda neutraaleja paikanteita, jolloin määrittelemätöntä maatunnusta edustaa tyhjä merkkijono. Variantteja ei ole määritelty Java-ympäristössä, joten ne ovat täysin mielivaltaisia ja niiden toteutus on täysin ohjelmoijan omalla vastuulla. Eräs Java Tutorialissa esitetty käyttötapaus paikannevarianteille on käyttää eri paikanteita Windows- ja UNIX-ympäristöissä. Muutamalla suurimmalle maalle ja niiden kielille on määritelty Locale-tyyppisiä vakioita (esim. java.util.Locale.GERMANY).

Javan viestiluettelo- ja resurssijärjestelmä perustuu ResourceBundle-luokkiin. Käytännössä ResourceBundlet ovat java.util.ResourceBundle-luokan aliluokkia ja niitä vastaavia tekstitiedostoja tai luokkatiedostoja, jotka nouttavat tiettyä nimeämiskäytäntöä. Nimeämiskäytäntö on mallia <nimipohja>_<kieli>-_<maa>_<variantti>. ResourceBundlen lataaja yrittää ladata käytössä olevaa paikannetta mahdollisimman hyvin vastaavan ResourceBundlen edeten yksityiskohtaisesta yleiseen, jolloin viimeisenä vaihtoehtona on pelkästä nimipohjasta koostuva ResourceBundlen nimi eli oletusresurssitiedosto. Varsinaisena resurssitiedostona toimii tiedostojärjestelmässä sijaitseva tiedosto, jolla on sama nimi kuin vastaavalla ResourceBundle-oliolla sekä .properties- (Property-ResourceBundle) tai .class- (ListResourceBundle) –pääte. Properties-tiedosto on yksinkertainen tekstitiedosto, joka voi sisältää #-alkuisia kommenttirivejä sekä avain-arvo pareja muodossa <avain> = <arvo>.[8] ListResourceBundlen perustana olevat luokkatiedostot ovat kuten mitkä tahansa Java-luokkatiedostoja, joten ne luodaan normaaliin tapaan kääntämällä lähdekoodista tavukoodiksi. (Sun, 2007). ListResourceBundlen käyttötapa sotii hieman sitä internationalisoinnin perusajatusta vastaan, että lokalisoitavan sisällön ei pitäisi sisältää koodia, jota ”maallikkolokalisoijan” voi olla vaikeaa ymmärtää ja muokata. ListResource-Bundlen käyttö tulisikin rajoittaa sellaisiin paikannekohtaisiin algoritmeihin, joissa sen käyttö on perusteltua. Mikäli mahdollista, tulisi mieluummin käyttää internationalisointitapaa, jossa toiminnallisuus voidaan parametrisoida teksti-

---

[8] Tämä formaatti on identtinen .NET-resurssien tekstitiedostojen kanssa.

muotoisesti properties-tiedostossa ja luoda tarvittava olio dynaamisesti PropertyResourceBundlesta luettuihin paikannekohtaisiin parametreihin perustuen.

ResourceBundle-luokkien lisäksi lokalisoitujen viestien käyttämiseen tarvittavia luokkia löytyy paketista (Java-nimiavaruudesta) java.text. java.text.MessageFormat-luokka huolehtii parametrien sijoittamisesta viestiin samoin kuin C:n printf-perheen funktiot. Tämän lisäksi monikkomuotojen hallintaan on oma luokkansa java.text.ChoiceFormat (vrt. C:n ngettext()-funktio ja wxWidgetsin wxPLURAL-makro). Numeroiden muotoilusta vastaa java.text.NumberFormat-luokka. Luokka on abstrakti, joten siitä ei voi luoda instansseja suoraan new-operaattorilla. Sen sijaan luokkaa käytetään kutsumalla sen staattisia metodeja, jotka palauttavat kuhunkin käyttötarkoitukseen erikoistuneen instanssin huomioiden käytössä olevan paikanteen: getNumberInstance, getCurrencyInstance, getPercentInstance jne. Näiden metodien palauttamille instansseille voidaan kutsua format-metodeja, jotka palauttavat oikealla tavalla paikannekohtaisesti muotoiltuja lukuja. java.text.DateFormat-luokka toimii samalla tavalla päivämäärille. Itse asiassa edellä mainittu paikannekohtaisen instanssin luontitapa on yhteinen kaikille Java-vakiokirjaston paikanteen huomioiville luokille.

Javan kalenteriluokka (java.util.Calendar) on rakennettu siten, että uusien kalenterijärjestelmien lisääminen on mahdollista. Esimerkiksi japanilaisen kalenterin tarvitsema aikakausikenttä (ERA) on määriteltynä luokalle. Ainoa vakiokirjaston virallisesti toteuttama kalenterijärjestelmä on gregoriaanis-juliaaninen hybridikalenteri (java.util.GregorianCalendar). Juliaanisesta gregoriaaniseen järjestelmään siirtymispäivän pystyy asettamaan vastaamaan käytömaan kalenterijärjestelmää, mutta toisin kuin wxWidgets, Java-vakiokirjasto ei sisällä valmiiksi tietoa eri maiden siirtymäpäivästä. Edellisen lisäksi Java 6:n SDKssa on dokumentoimaton japanilainen kalenteri (JapaneseImperialCalendar-luokka), joka palautuu esimerkiksi seuraavasta lausekkeesta:

```
Calendar japanese = Calendar.getInstance(new Loca-
le("ja", "JP", "JP"));

System.out.println(japanese.getClass());

/* tämä tulostaa: class ja-
va.util.JapaneseImperialCalendar */
```

Merkkijonojen paikanneherkästä vertailusta vastaa luokka java.text.Collator. Luokka mahdollistaa myös lajittelun tarkkuuden määrittämisen neljällä eri asteella. Aste määrittää sen, mitkä merkkien ominaisuudet ovat merkityksellisiä vertailuissa. Astevaihteluiden määritelmät ovat paikannekoh-

taisia. Yksi esimerkki asteista voisi olla, että eri perusmerkki on ensimmäisen asteen eroavuus, saman perusmerkin eri aksenttiversiot on toisen asteen eroavuus ja ison ja pienen kirjaimen ero on kolmannen asteen eroavuus. Myös Collator-luokassa on huomioitu toistuviin vertailuihin (esim. lajittelualgoritmissä) liittyvä tehokkuusongelma. Toistuvia vertailuja varten verrattavasta merkkijonosta voidaan muodostaa vertailuavain, jota voidaan käyttää bittitason vertailuihin. Tämä tapahtuu getCollationKey-metodin avulla. Vertailuavainten käyttö muistuttaa siis C-kirjaston strxfrm-funktion käyttöä.

Kuten aikaisemmin todettiin, Java on natiivisti Unicode-valmis. Java-merkkijonoja käsitellään sisäisesti UTF-16-koodausta noudattavina kaksitavuisina yksiköinä. Merkkimuunnokset Unicode- ja ei-Unicode-merkistöjen välisen muunnokset on toteutettu luontevasti java.lang.String-luokan metodeina. Yksinkertaisimmillaan muunnos yksi- tai monitavuisesta merkistökoodauksesta Unicode-merkkijonoksi tapahtuu String-luokan rakentajalla, joka saa parametriksi tavutaulukon ja tiedon käytetystä merkistöstä joko merkkijonomuodosta tai java.nio.charset.Charset-instanssina. Päinvastaiseen suuntaan muunnos tapahtuu vastaavanlaisella String.getBytes-metodilla. String-rakennin olettaa, että lähteenä toimiva tavutaulukko sisältää koodauksen kannalta oikeaoppista dataa, ja sen toiminta virhetilanteessa on määrittelemätöntä. Kirjastossa on myös luokka, joka antaa ohjelmoijalle samankaltaiset valtuudet virhetilanteiden ratkomiseen kuin C:n iconv-perheen funktiot ja .NET-kehyksen Encoder/Decoder-luokat. Tämä luokka on nimeltään java.nio.charset.CharsetDecoder ja se esitellään tarkemmin alla rajapintakuvauksen yhteydessä.

Java poikkeaa sikäli muista tässä tutkimuksessa käsitellyistä internationalisointikirjastoista, että siinä on valmiina mukana algoritmi kaksisuuntaisen tekstin käsittelyyn. Tästä algoritmista huolehtii luokka java.text.Bidi. Kaksisuuntainen teksti koostuu joukosta yksisuuntaisia tekstipätkistä, jotka limittyvät toistensa sisälle. Bidi-luokka sisältää joukon metodeja, joilla tekstin lukusuunta ja taso mielivaltaisessa kohdassa tekstiä voidaan selvittää. Taso on määritelty Javassa siten, että se edustaa tekstipätkien sisäkkäisyysastetta sekä suuntaa. Parillisen tason pätkät ovat vasemmalta oikealle luettavia ja parittomat oikealta vasemmalle luettavia. Tason 2 tekstinpätkä edustaa oikealta vasemmalle etenevän tekstiin upotettua vasemmalta oikealle-pätkää jne. Lisäksi Bidi-luokan avulla voi järjestää tekstipätkät visuaaliseen järjestykseen normaalin loogisen järjestyksen sijaan.

Java sisältää syöttömetodeja syöttömetodeja tukevan kehyksen, jossa on kahdenlaiset rajapinnat. Ns. ”input method client API” mahdollistaa Java-ohjelmien kommunikoimisen natiivien syöttömetodieditorien kanssa, jolloin

Java-ohjelmien tekstinsyöttö voidaan toteuttaa mahdollisimman luontevalla tavalla. Toinen rajapinta ”input method engine SPI” on tarkoitettu syöttömetodien toteuttamiseen käyttäen Java-kieltä. Input method engine SPI-rajapintaan perustuvien syöttömetodipalvelujen käyttö on mahdollista missä tahansa Java-ajoympäristössä.

Javan internationalisointispesifiset luokat sisältyvät paketteihin, jotka usein ovat muutenkin mukana sovelluksessa (erityisesti java.util sen sisältämien säiliöluokkien vuoksi). Tässä kohdassa mainittujen internationalisointiominaisuuksien koko kirjastossa on noin 7 MB (Java SDK 6, x86-versio), josta eri merkistötukien toteutukset vievät kokonaista 6 MB.

Aikakäyttäytymisen mittaustapa antaa hieman muista poikkeavan tuloksen johtuen siitä, että suurin osa sovelluksen kokonaissuoritusajasta kuluu virtuaalikoneen käynnistämiseen ja purkamiseen. Näin ollen lokalisoimattoman ja lokalisoidun version suoritusajan suhteellinen ero jää huomattavan pieneksi.

Taulukossa 6 on esitetty viiterajapintaa vastaava Java 6 SE:n internationalisointirajapinta. Tässäkin rajapintakuvauksessa avainsana ”public” on jätetty pois lyhyyden vuoksi. Oletuksena on, että koko sovelluksessa käytetään samaa paikannetta, ja että kaikki resurssit ovat toteutettavissa käyttämällä PropertyResourceBundle-luokkia. Oletamme, että PropertyResourceBundle saa tarvitsemansa tiedostot tiedostojärjestelmän levylle tallennetusta tekstitiedostosta. Periaatteessa resurssitietojen lähteenä voi toimia mikä tahansa merkkisyöte. Taulukosta on lyhyyden vuoksi jätetty pois CharsetEncoder-luokka, joka toimii kuten CharsetDecoder, mutta päinvastaiseen suuntaan.

Metodi	**Tehtävä**
Java.util.Locale(String language, String country)	Rakennin, joka luo kieltä language ja maata country vastaavan paikanteen.
String java.util.Locale.getCountry()	Palauttaa paikanteen maan nimen.
String java.util.Locale.getLanguage()	Palauttaa paikanteen kielen nimen.
static void java.util.Locale.setDefault (Locale newLocale)	Asettaa parametrissa newLocale osoitetun paikanteen Java-virtuaalikoneen oletuspaikanteeksi.
Static ResourceBundle java.util.PropertyResourceBundle.getBundle (String basename)	Palauttaa uuden resurssiolion, joka perustuu käytössä olevaan oletuspaikanteeseen ja jonka resurssitiedostojen perusosa on määritetty merkkijonossa basename. Heittää poikkeuksen MissingResourceException, jos nimettyjä resurssitiedostoja ei ole olemassa.
final String java.util.PropertyResourceBundle.getString	Palauttaa avainta key vastaavan

(String key)	merkkijonoresurssin resurssitiedostosta. Heittää poikkeuksen MissingResourceException, jos nimettyä avainta ei ole olemassa.
static Charset java.nio.charset.Charset.forName (String charsetName)	Palauttaa merkkijonossa charsetName nimettyä merkistöä vastaavan olion, joka edustaa vastaavaa Charset-luokan aliluokkaa. Heittää poikkeuksen IllegalCharsetNameException, jos charsetName ei vastaa mitään tunnettua merkistöä, ja UnsupportedCharsetException, jos käytössä oleva Java-virtuaalikone ei tue pyydettyä merkistöä.
abstract CharsetDecoder java.nio.charset.Charset.newDecoder()	Palauttaa muunninolion, joka muuntaa Charset-aliluokan mukaisesta merkistöstä Unicode-merkeiksi.
Final CoderResult java.nio.charset.CharsetDecoder.decode (ByteBuffer in, CharBuffer out, boolean endOfInput)	Muuntaa merkkejä syötepuskurista in Unicode-merkeiksi puskuriin out. endOfInput-lippu kertoo muuntimelle, että syöte on muunnosoperaation viimeinen.[9] Palauttaa CoderResult-olion, jonka alla esiteltyjä metodeja kutsumalla voidaan selvittää muunnoksen tila.
Boolean java.nio.charset.CoderResult.isUnderflow()	Palauttaa arvon tosi, jos edellinen muunnos päättyi alivuotoon (lähdepuskurista loppui data)
boolean java.nio.charset.CoderResult.isOverflow()	Palauttaa arvon tosi, jos edellinen muunnos päättyi alivuotoon (data ei mahtunut kohdepuskuriin)
Boolean java.nio.charset.CoderResult.isMalformed()	Palauttaa arvon tosi, jos muunnos päättyi virhetilaan vääränmuotoisen syötedatan vuoksi.

[9] Vaikka tämä lippu muistuttaa aiemmin esitellyissä rajapinnoissa käytettävää vastaavaa lopetuslippua, ei tällä lipulla ole tekemistä muuntimen tilan palauttamisen kanssa. Lippu auttaa tunnistamaan vääränmuotoinen syöte. Tilan palauttamiseksi pitää vielä kutsua erikseen reset-metodia.

boolean java.nio.charset.CoderResult.isUnmappable()	Palauttaa arvon tosi, jos muunnos päättyi virhetilaan, koska lähdedatan merkki on validi, mutta ei kuvattavissa kohdemerkistössä.
Final CoderResult java.nio.charset.CharsetDecoder.flush (CharBuffer out)	Kirjoittaa muunnoksen lopuksi vielä kirjoittamatta jääneet merkit kohdepuskuriin out. Normaalitilanteessa palauttaa alivuotoa kuvaavan olion, jos puskuri täyttyi, palautetaan ylivuotoa kuvaava olio.
final charsetDecoder java.nio.charset.CharsetDecoder .reset()	Palauttaa muuntimen alkutilaansa ja palauttaa viitteen muunninolioon.

Taulukko 6. Java 6 SE:n internationalisointirajapinta

Javassa on pyritty edeltäjiään parempaan luettavuuteen ja kirjoitettavuuteen, mikä näkyy osittain myös internationalisointirajapinnassa. getInstance-tyyppinen instantiointi saattaa vaikuttaa aluksi oudolta, mutta sitä noudatetaan johdonmukaisesti eri yhteyksissä, mikä nopeuttaa oppimista.

UFC-laskelmassa tulosteeksi on laskettu tulosteeksi palautusarvojen lisäksi poikkeukset. Ulkoisia tiedostoja on yksi (PropertyResourceBundlen tekstitiedosto). Täten saadaan UFC-arvoksi UFC = 4 * 10 + 5 * 14 + 10 * 1 = 120.

Java-ympäristö on monessa mielessä omaperäinen, joten internationalisointikoodi ei monessa mielessä ole helposti siirrettävissä muihin ympäristöihin. Kielessä ei ole esikäsittelijämakroja, mikä vaikeuttaa siirrettävän koodin kirjoittamista. Itse resurssitiedostot ovat tekstimuotoisia ja yksinkertaisia, joten ne on helppo muuntaa muihin resurssiformaatteihin (erityisesti .NET-resursseiksi, koska resgen-sovellus muuntaa properties-tiedostoja suoraan). Lisäksi voidaan todeta, että .NET-sovellukseen siirryttäessä voidaan käyttää Visual J# -ympäristöä, jolloin ohjelmiin tarvittavat muutokset ovat pienempiä.

6. Yhteenveto

Kävimme läpi viiden eri aikakautena kehitetyn ohjelmointikirjaston internationalisointiominaisuuksia käyttäen kiinnekohtana todellista viitesovellusta ja sen internationalisointiin liittyvistä operaatioista koostuvaa rajapintaa. Käytimme arvioinnissa ISO 9126 –standardiin ja kirjallisuudesta saatuihin internationalisointisuosituksiin pohjautuvaa arviointikehystä näiden ominaisuuksien arvioimiseen.

6.1. Tuloksista

Tutkimuksessa käsiteltyjen ympäristöjen toiminnallisuuksien vertailu vahvistaa sen ennakkokäsityksen, että myöhemmin kehitetyt ympäristöt ja kielet vastaavat monipuolisemmin internationalisointitarpeeseen. Tulos on ymmärrettävä, sillä C:n ja C++:n pääasiallinen painopiste on systeemiohjelmoinnissa, kun taas muut tutkimuksen kirjastot ja kielet ovat luonteeltaan enemmän yleiskehitystyökaluja. Toisaalta nämä ympäristöt on kehitetty aikana, jolloin internationalisointi on jo ehtinyt muodostua tärkeäksi osaksi ohjelmistokehitystä. Yhden tärkeimmistä internationalisointiominaisuuksista, paikannejärjestelmän suhteen ei pystytä osoittamaan merkittäviä eroja eri ympäristöjä verrattaessa. .NET-kehys mahdollista hieman muita tarkemman kontrollin paikanteiden suhteen kolmitasoisine paikanteineen. .NET- ja Java-ympäristöt ovat selkeästi parempia siinä tapauksessa, että on tarpeellista tehdä monisäikeisiä multilingualisoituja sovelluksia. .NET nousee ainoaksi vaihtoehdoksi siinä tapauksessa, että ohjelmoija kaipaa vakiokirjastosta valmista tukea muille kalenterijärjestelmille kuin länsimaisille gregoriaaniselle kalenterille. Tosin sekä wxWidgetsin että Javan kalenterituki näyttäisi olevan laajenemassa lähiaikoina. Java puolestaan antaa ohjelmoijalle eniten mahdollisuuksia bidi-tukea tarvittaessa.

UFC-metriikat antavat viitettä siitä, että uudemmissa ympäristöissä olisi päästy yksinkertaisempaan rajapintaan verrattuna aikaisempiin. Tosin on todettava, että arvot eivät ole täysin vertailtavissa, sillä huolimatta pyrkimyksestä määrittää mahdollisimman hyvin viitesovellusta vastaava osajoukko internationalisointikirjastosta, ne sisältävät varsin erilaisia operaatioita. Koska C-rajapintaan ei sisällä resurssikäsittelyfunktiota, sitä ei voida edes suoraan verrata muihin. Voidaan kuitenkin esimerkiksi todeta, että UFC-arvo on samaa luokkaa kaiken viiterajapinnan toiminnallisuuden sisältävän .NET-kehyksen arvon kanssa, josta voidaan todeta C-rajapinnan olevan kompleksisempi. Toisaalta ei ole itsestäänselvää, että pienempi kompleksisuus on yhtäpitävä helppokäyttöisyyden kanssa. Joissakin tapauksissa voi olla selkeämpää suorittaa tietty tehtävä useammassa osassa käyttäen useampaa operaatiota.

Kaikkien tutkimuksessa käsiteltyjen ohjelmointiympäristöjen internationalisointiominaisuudet vaikuttaisivat aiheuttavan suhteellisen pienen resurssikuorman verrattuna lokalisoimattomiin sovelluksiin. Tutkimuksessa keskityttiin näyttötekstilokalisoinnin tuomaan resurssikuormaan, ja tulee huomata, että vaikutus on kertaluontoinen, koska kertaalleen ladattuja resursseja voidaan jatkossa käsitellä yhtä kevyesti kuin ”kovakoodattuja” vastineita. Nämä huomiot puoltavat sitä, että internationalisointia ei tarvitse ajatella ylimääräisenä harkittavana lisävaatimuksena, vaan sitä on syytä noudattaa järkevänä ohjelmistokehitysperiaatteena, vaikka tähtäimessä ei heti olisi ohjelmiston markkinointi globaalisti. Muiden kuin näyttötekstilokalisointien osalta osoittautui vaikeudeksi määrittää, mikä olisi minimipari internationalisoidun ja perinteisen sovelluksen välillä. Siksi muiden internationalisointitoimien vaikutus jää tämän tutkimusten tuloksista puuttumaan.

Tutkimus osoitti, että ISO 9126 -standardi on ainakin jossakin määrin sovellettavissa ohjelmistokehitysympäristön ja sen internationalisointiominaisuuksien arviointiin. Toisaalta monet asiat jäivät varsin ambivalenteiksi johtuen siitä, että niitä on vaikea siirtää yleisestä ohjelmistokontekstista tämän tutkimuksen kohdetta vastaavaan tarkoitukseen, tai koska arviointikriteerit kohdistuivat toteutustason asioihin.

Voidaan myös todeta, että viitesovellus ei sisältänyt kaikkia mahdollisia internationalisoinnissa huomioitavia näkökohtia, mistä johtuen tarkastelu jää puutteelliseksi. Toisaalta voidaan huomioida, että avoimen lähdekoodin sovelluksessa internationalisointiin ei välttämättä ole kiinnitetty sellaista huomiota, jota se kaupallisessa sovelluksessa saa, koska internationalisoinnilla on pitkälti taloudellisia vaikutuksia. Lisäksi monet arvioinnit jäivät yksiselitteisen objektiivisen mittarin puutteessa kirjoittajan subjektiivisen arvion varaan. Ohjelmointikielten valintaan vaikutti niiden tuttuus kirjoittajalle. Mikäli kielivalikoima olisi sisältänyt laajemman otoksen eri ohjelmointiympäristöjä, olisi ehkä saatu toisenlaisia tuloksia.

6.2. Aiheita jatkotutkimukseen

Koska monet tutkimuksessa arvioiduista asioista ovat lopulta subjektiivisia ja vaikeasti kvantitatiivisesti määritettäviä asioita, olisi mielekästä selvittää, miten esimerkiksi internationalisointirajapintojen kompleksisuus vaikuttaa niiden käyttökokemukseen. Tätä varten pitäisi tehdä kyselytutkimus haastattelemalla ohjelmistokehittäjiä, jotka todella joutuvat käyttämään ohjelmistoympäristöjen internationalisointiominaisuuksia työssään.

Koska ISO 9126 ei suoraan vastaa ohjelmointiympäristön arvioimisessa tarvittaviin asioihin, olisi kenties tarvetta vastaavalle kriteeristölle, jota voitaisiin suoraan käyttää tämän tyyppiseen arviointiin.

Viiteluettelo

[Bos and Dürst, 1998] Bert Bos, Martin Dürst, Internationalization, In: *Seventh International World Wide Web Conference (WWW7)*, Brisbane, 1998.

[Dumbill, 2001] Edd Dumbill, Practical Internationalization, In: O'Reilly xml.com, http://www.xml.com/pub/a/2001/04/18/i18n.html

[Edwards *et al*., 2004] Leigh Edwards, Richard Barker and the Staff of EMCC Software Ltd., *Developing Series 60 Applications: A Guide for Symbian OS C++ Developers*, Addison-Wesley, 2004.

[Esselink, 2000] Bert Esselink, *A Practical Guide to Localization*, John Benjamins Publishing Co., 2000.

[Fenton and Pfleeger, 1997] Norman E. Fenton, Shari Lawrence Pfleeger, *Software Metrics, A Rigorous and Practical Approach*, PWS Publishing Company, 1997.

[Filezilla, 2007] *FileZilla – The free FTP solution*, http://filezilla-project.org/ .

[Garland, 2005] Jeff Garland, *N1900=05-0160 Proposal to Add Date-Time to the C++ Standard Library 0.75*, CrystalClear Software, Inc, 2005. Available on the Internet: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1900.pdf .

[GNU, 2001] *The GNU C Library Reference Manual*, http://www.gnu.org/software/libc/manual/html_node/index.html

[GNU2] *GNU 'gettext' utilities*, http://www.gnu.org/software/gettext/manual/html_mono/gettext.html.

[Gross, 2006] Steffen Gross, *Internationalization and Localization of Software*. Master's thesis, Department of Computer Science, Eastern Michigan University, 2006.

[He *et al*., 2002] Z. He, D. W. Bustard, X. Liu, Software internationalisation and localisation: practice and evolution. In: *Proceedings of the Inaugural Conference on the Principles and Practice of programming, 2002 and Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines, 2002*. National University of Ireland, Dublin 2002.

[Hogan *et al*., 2002] James M. Hogan, Chris Ho-Stuart, Binh Pham, *Current Issues in Software Internationalisation*, available on the Internet: http://sky.fit.qut.edu.au/~hogan/pub/inter.pdf

[HP, 2001] Hewlett-Packard Company, *HP-UX 9.x – 11i Intenationalization Features White Paper, Edition 1*. Palo Alto 2001.

[Immonen ja Sajaniemi, 2003] Jarkko Immonen, Jorma Sajaniemi, *Software Globalization in Finland: A State-of-the-practice Survey*. Report A-2003-1, University of Joensuu, Department of Computer Science.

[ISO 639] International Standard ISO 639. *Codes for the representation of names of languages,* International Organization for Standardization, Geneva 2002.

[ISO 3166]International Standard ISO 3166. *Codes for the representation of countries and their subdivisions*, International Organization for Standardization, Geneva 2006.

[ISO 8859]International Standard ISO/IEC 8869. *Information technology – 8-bit single byte coded graphic character sets*, International Organization for Standardization, Geneva 2003.

[ISO 9126]International Standard ISO/IEC 9126. *Information technology -- Software product evaluation -- Quality characteristics and guidelines for their use,* International Organization for Standardization, International Electrotechnical Commission, Geneva 1991.

[ISO 9899]International Standard ISO/IEC 9899. *Programming Languages – C,* International Organization for Standardization, International Electrotechnical Commission, Geneva 1999.

[Kano, 1995] Nadine Kano, *Developing International Software for Windows 95 and Windows NT*, Microsoft Press, 1995.

[Kaplan, 2007] Michael S. Kaplan, *Mixing it up with bidirectional text*, In: Sorting It All Out, MSDN Blogs, 6.1.2007, available on the Internet: http://blogs.msdn.com/michkap/archive/2007/01/06/1421178.aspx

[Kernighan and Ritchie, 1988] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, New Jersey, 1988.

[Kokkotos and Spyropoulos, 1997] Stavros Kokkotos, C. D. Spyropoulos, An architecture for designing internationalized software. In: *Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering*]. London 1997.

[Kosnik, 2001] Benjamin Kosnik, *Notes on the messages implementation*, http://gcc.gnu.org/onlinedocs/libstdc++/22_locale/messages.html

[Käpyaho, 2001] Jere Käpyaho, *Internationalisation in Operating Systems for Handheld Devices*, Master's thesis, Department of Computer Sciences, University of Tampere, 2001.

[LISA, 2007] The Localization Industry Standards Association, *LISA Globalization Industry Primer,* 2007.

[McKenna, 2002] Michael G. McKenna, Tutorial: Unicode in Distributed Systems. In: *21st International Unicode Conference,* Dublin, 2002.

[Nokia, 2006] Nokia Corporation, *S60 Platform: Internationalization and Localization Guide,* Version 1.0, October 17, 2006. Available on the Internet: http://forum.nokia.com

[RFC 959] J. Postel, J. Reynolds, *RFC 959: File Transfer Protocol (FTP).* October 1985. http://rfc.net/rfc0959.html

[RFC 3066] H. Alvestrand, *RFC 3066: Tags for the Identification of Languages.* January 2001. http://rfc.net/rfc3066.html

[RWS, 1996] Rogue Wave Software, Inc., *Standard C++ Library User Guide, Volume 2: Iostreams and Locale.* Saatavilla Internetissä: http://www.tacc.utexas.edu/resources/user_guides/pgi/pgC++_lib/stdlibug/ug2.htm

[Sebesta, 1999] Robert W. Sebesta, *Concepts of Programming Languages, Fourth Edition.* Addison-Wesley, 1999.

[Singh, 2003] Prakash Kumar Singh, *.NET – Localization using Resource file.* http://www.codeproject.com/dotnet/Localization.asp

[Smart *et al.*, 2007] Julian Smart, Robert Roebling, Vadim Zeitlin, Robin Dunn, et al., *wxWidgets 2.8.5: A Portable C++ and Python GUI Toolkit*. Available on the Internet: http://wxwidgets.org/manuals/stable/wx_contents.html .

[Stroustrup, 1997] Bjarne Stroustrup, *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[Stroustrup, 2003] Bjarne Stroustrup, *Industrial frameworks for distributed systems,* Texas A&M University, 2003. Available on the Internet: http://students.cs.tamu.edu/jchen/cpsc689-606/2003-689-606.htm

[Stroustrup, 2004] Bjarne Stroustrup, *The C++ Programming Language (Third Edition and Special Edition) web page, Appendix D: Locales,* available on the Internet: http://www.research.att.com/~bs/3rd_loc0.html

[Sun, 2005] Sun Microsystems, Inc., *SunOS Reference Manual, Section 3. Library Routines (A-M)*, Mountain View, California, 1995.

[Sun, 2007] Sun Microsystems, Inc., *The Java Tutorials, Trail: Internationalization,* http://java.sun.com/docs/books/tutorial/i18n/

[Symbian, 2007] Symbian Software Ltd, *Symbian OS Guide, Symbian OS v9.2.* Available on the Internet: http://www.symbian.com/developer/techlib/v9.2docs/doc_source/guide/

[Taylor, 1992] Dave Taylor, *Global Software : developing applications for the international market*, Springer-Verlag, New York, 1992.

[uSTL, 2007] *uSTL - the small STL library.* http://ustl.sourceforge.net .

[XPG4] X/Open Company Limited, *X/Open Portability Guide, Volume 4: Programming languages,* Prentice-Hall, 1988.

[Zhao, 2003] Yan Zhao, *Localization requirements and challenges on development of a Chinese Symbian OS,* Master's Thesis, Department of Computer Sciences, University of Tampere, 2003.

Liite 1. Yhteenveto tutkimustuloksista

Kieli / kirjasto	Julkaisuvuosi	Paikannejärjestelmä	Paikanteen tarkkuus	Paikanteen laajuus	RFC 3066:n mukainen	Viestiluettelo	Gettext	Catgets	Java properties
ISO C99	1999	locale.h	variantti	globaali	kyllä	ei	ei	ei	ei
Solaris 9 / SunOS 5.9 C	2001	locale.h	variantti	globaali	kyllä	catgets, gettext	kyllä	kyllä	ei
ANSI / ISO C++ (GNU GCC)	1998	locale, facetit	variantti	olio / säie	kyllä	catgets, gettext	kyllä	kyllä	ei
wxWidgets	2007	wxLocale	variantti	globaali	(kyllä)	gettext	kyllä	ei	ei
C# / .NET Framework 2.0	2005	Kulttuurit	variantti	säie	kyllä	Microsoft .RES	ei	ei	resgen-muunnoksella
Java SE 6	2006	java.util.Locale	maa	säie	kyllä	PropertyBundle	ei	ei	kyllä

Kieli / kirjasto	Multibyte-tuki	Unicode-tuki	Kalenterituki	Bidi-tuki	Viiterajapinnan UFC	Hello World-suoritusajat (s)					
						Ei lokalisointia	%	C / POSIX	%	Muu paikanne	%
ISO C99	kyllä	ei	gregoriaaninen	ei							
Solaris 9 / SunOS 5.9 C	kyllä	ei sisäisesti, käyttöjärjestelmä-API ja UTF-8-paikanne	gregoriaaninen	ei	145 (gettext)	1,16	100	1,49	128,14 %	1,87	160,59 %
ANSI / ISO C++ (GNU GCC)	kyllä	ei	gregoriaaninen	ei	265	2,85	100	2,90	101,75 %	3,54	124,20 %
wxWidgets	kyllä	Unicode-buildissa	gregoriaaninen, juliaaninen rajoitetusti	rajoitettu	247						
C# / .NET Framework 2.0	kyllä	kyllä	Gregoriaaninen, juliaaninen, lunisolar, heprealainen, hijri, japanilainen, korealainen, persialainen, thaibuddhist, Umm-Al Qurah	Rajoitettu / ei-julkinen	137	339,21	100	357,52	105,40 %	357,68	105,44 %
Java SE 6	kyllä (muunnosfunk	kyllä, sisäisesti UTF-16	Gregoriaanis-juliaaninen hybridi, JapaneseImperialCalendar (ei-dokumentoitu)	Kyllä	120	107,87	100	109,47	101,49 %	110,57	102,50 %

Liite 2. Mittauskoodit

C:

```c
#include <libintl.h>

#include <locale.h>

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

  setlocale( LC_ALL, "" );

  bindtextdomain( "hello", "/usr/share/locale" );

  textdomain( "hello" );

  printf( gettext( "Hello, world!\n" ) );

  exit(0);

}
```

C++:

```cpp
#include <iostream>
#include <locale>
using namespace std;

int main(void)
{
  typedef messages<char>::catalog catalog;
  const char *dir = "/usr/share/locale";
  const locale loc_fi("fi_FI.utf8");
  const messages<char>& msg_fi =
use_facet<messages<char> >(loc_fi);

  catalog cat_fi = msg_fi.open("hello", loc_fi, dir);
  string hello = msg_fi.get(cat_fi, 0, 0,
                 "Hello, world!\n");

  cout << hello << endl;

  msg_fi.close(cat_fi);
  exit(0);
}
```

C#:

```
using System;
using System.Reflection;
using System.Resources;
using System.Globalization;

namespace HelloL10N
{
    class Hello
    {
        static void Main(string[] args)
        {
            CultureInfo ci = new CultureInfo
                            ("fi-FI");
            ResourceManager resmgr =
new ResourceManager("HelloL10N.MyResource",
Assembly.GetExecutingAssembly());
            string hello =
resmgr.GetString("hello", ci);
            System.Console.WriteLine(hello);
        }
    }
}
```

Java:

```
import java.util.*;


public class HelloL10N {
    static public void main (String[] args) {
    Locale currentLocale = Locale.getDefault();
    ResourceBundle helloBundle =
ResourceBundle.getBundle("HelloBundle",
                                 currentLocale);
    String hello = helloBundle.getString("hello");
    System.out.println(hello);
    }
}
```