Metropolia ammattikorkeakoulu
Tietotekniikan koulutusohjelma

Petri Tolonen

Natiivin lähdekoodin uudelleenkäytettävyys Eclipse-ympäristössä JNI-rajapinnan avulla

Insinöörityö 28.7.2008

Ohjaaja: jaospäällikkö Mikko Kamunen
Ohjaava opettaja: yliopettaja Markku Karhu

METROPOLIA AMMATTIKORKEAKOULU INSINÖÖRITYÖN TIIVISTELMÄ

Tekijä Otsikko Sivumäärä Aika	Petri Tolonen Natiivin lähdekoodin uudelleenkäytettävyys Eclipse-ympristössä JNI-rajapinnan avulla. 46 sivua 1.9.2008
Koulutusohjelma	tietotekniikka
Tutkinto	insinööri (AMK)
Ohjaaja Ohjaava opettaja	jaospäällikkö Mikko Kamunen yliopettaja Markku Karhu

Työssä pyrittiin pääasiassa keräämään kokemuksia Javan JNI-rajapinnasta ja selvittämään mahdollisuuksia natiivin ohjelmakoodin uudelleenkäytettävyydestä Eclipse-ympäristöön siirryttäessä. Työn tarkoituksena oli tutkia ongelmaa teknisen toteutuksen näkökulmasta valitun teknologian avulla ilman vertailua muihin mahdollisiin vaihtoehtoihin.

Koska työn tarkoituksen oli kerätä käytännön kokemusta aiheesta, aihetta lähestyttiin toteuttamalla JNI-rajapintaa hyödyntävä ohjelma, jonka avulla voitiin nähdä, mitä rajapinnan käyttöönotto vaatisi ja mitä ongelmia se mahdollisesti toisi mukanaan. Ohjelmaa varten toteutettiin natiivi kirjasto, joka sisälsi erilaisia toimintoja. Osa toiminnoista oli pidempikestoisia, joten näitä varten tulisi tutkia myös natiivien toimintojen suorittamista asynkronisesti. Ohjelman käyttöliittymä toteutettiin Eclipse-ympäristössä käyttäen SWT-käyttöliittymäkirjastoa ja natiivin kirjaston toiminnallisuudet integroitiin käyttöliittymään käyttäen JNI-rajapintaa.

Suoritetussa tutkimuksessa kävi ilmi, että JNI-rajapinta on sen tarjoamien toiminnallisuuksien puolesta riittävän kattava tämän kaltaisten projektien toteuttamiseksi. Kyseessä on kuitenkin melko matalan tason rajapinta, jolloin sen käyttäminen suoraan on melko virhealtista ja tarpeettoman hankalaa. Tämä tulisi huomioida ennen rajapinnan laajamittaista käyttöönottoa.

Tämän projektin tuloksia voidaan hyödyntää siirrettäessä sovellusta natiivista ympäristöstä Eclipse-ympäristöön ja tutustuttaessa integraatiosta aiheutuviin ongelmakohtiin jo projektin alkuvaiheessa, jolloin sovelluksen arkkitehtuuri voidaan suunnitella siten, että rajapinnan aiheuttamat ongelmat saadaan minimoitua.

Hakusanat	JNI-rajapinta, Eclipse, uudelleenkäytettävyys

Helsinki Metropolia
University of Applied Science

ABSTRACT

Author	Petri Tolonen
Title	Reusability of native source code in Eclipse environment with the JNI-interface
Number of Pages Date	46 pages 1 September 2008
Degree Programme	Information technology
Degree	Bachelor of Engineering
Instructor Supervisor	Mikko Kamunen, Section Manager Markku Karhu, Principal Lecturer

The main purpose of this thesis work was to acquire experience of the JNI-interface and investigate possibilities to code reuse when moving from native environment to Eclipse environment. The focus of this work was purely on the technical implementation with a selected technology without making comparisons with any other possible technologies.

Because the purpose of the work was to get practical experience of the JNI-interface, it was accomplished by implementing an application that uses the JNI-interface. With this application it was possible to gather information about the usage of the JNI-interface and possible problems that may be encountered. For the application I implemented a native library that contained several different operations created just for test purposes. Some of the operations were really time-consuming so implementing asynchronous operations would need to be taken care of also. The user interface was implemented with Eclipse using SWT-library and the native library was integrated into the application using the JNI-interface.

The research shows that the JNI-interface provides all required functionalities for this kind of usage. However the interface is quite low-level, so it is unnecessarily difficult and error-prone to use it directly. This would need to be addressed before extensive use.

The results of this work can be used when an application needs to be moved from the native environment to Eclipse and one would like to get familiar with possible problems and architectural challenges when designing the application. With proper design you can minimize the problems that using the JNI-interface may cause.

Keywords	JNI-interface, Eclipse, reusability

Sisällys

Tiivistelmä
Abstract
Lyhenteet

1 Johdanto 7

1.1 Työn tavoite 7
1.2 Työn taustaa 8

2 Eclipse ja SWT lyhyesti 8

2.1 Eclipse 8
2.2 SWT 9

3 JNI-rajapinta 10

3.1 Historia 10
3.2 Toiminnallisuus 11

4 JNI-rajapinnan käsittely natiivista ohjelmakoodista 12

4.1 Perustietotyypit 12
4.2 Referenssitietotyypit 13
4.3 Luokat ja niiden instanssit 15
4.4 Tekstimuotoiset tietotyyppikuvaukset 17
4.5 Luokan kentät 19
4.6 Luokan metodit 21
4.7 Poikkeukset 23
4.8 Paikalliset ja globaalit referenssit 27
4.9 Säikeistys ja synkronointi 28

5 Natiivien kirjastojen käyttäminen Java-ohjelmasta 32

5.1 Natiivien kirjastojen linkitys 32
5.2 Vaikutukset Java-ohjelmassa 35

6 Esimerkkiohjelman toteutus 36

6.1 Tavoitteet 36
6.2 Käyttöliittymä 36
6.3 Toiminnallisuus 37
6.4 Valmis ohjelma 40

7 Käytännön kokemuksia 42

7.1 Yleisiä kokemuksia 42
7.2 Huomioitavia asioita suunnittelussa 43

8 Yhteenveto 45

Lähteet 46

Lyhenteet

API	Application Programming Interface, sovellusohjelmointirajapinta
AWT	Abstract Windowing Toolkit, Javan käyttöliittymäkirjasto
COM	Component Object Model, Windows-käyttöjärjestelmän komponenttimalli
DLL	Dynamic Link Library, Windows-käyttöjärjestelmän jaettu kirjasto
EPL	Eclipse Public License, Eclipsen avoimen lähdekoodin lisenssimalli
JDK	Java Development Kit, Java-kielen kehitysympäristö
JNI	Java Native Interface, Javan rajapinta natiivien kirjastojen käyttämiseksi
MFC	Microsoft Foundation Classes, Microsoftin käyttöliittymäkirjasto
SO	Shared Object, Linux-käyttöjärjestelmän jaettu kirjasto
SWT	Standard Widget Toolkit, Javan käyttöliittymäkirjasto

1 Johdanto

1.1 Työn tavoite

Tietotekniikan kehitys on ollut viime vuosikymmeninä huikeaa. Aikojen saatossa on ollut käytössä useita käyttöjärjestelmiä, kehitysympäristöjä, ohjelmointikieliä ja teknologioita, jotka ovat ajan myötä vaihtuneet uudempiin. Tiettynä aikakautena toteutetut ohjelmistot on toteutettu siihen aikaan parhaiksi nähtyjen työkalujen ja teknologioiden varaan ja ne ovat yleensä näistä riippuvaisia koko niiden elinajan. Samat ohjelmistot ovat usein kuitenkin käytössä useita vuosia, jolloin ohjelmiston suunnitteluvaiheessa valitut työkalut ja teknologiat saattavat olla jo osin vanhentuneita muutaman vuoden kuluttua. Pitkään ohjelmistokehitystä tehneille yrityksille on vuosien varrella kertynyt ohjelmistoja, jotka ovat pitkän kehitystyön tuloksena erittäin vakaita ja toimivia, mutta kenties sisältävät osin vanhentunutta teknologiaa ja ovat mahdollisesti toteutettu vanhentuneella kehitysympäristöllä, jota ei enää tueta, eikä ole vuosiin kehitetty. Toisinaan kuitenkin vanhemmista ohjelmistoista halutaan erinäisistä syistä johtuen toteuttaa uuden sukupolven versioita uusin työkaluin ja teknologioin, ja tällöin haluttaisiin tietysti mahdollisuuksien mukaan hyödyntää pitkän kehitystyön tuloksena syntynyttä jo olemassa olevaa ohjelmistoa.

Tällä hetkellä yksi nykyaikaisista kehitysympäristöjen ja teknologioiden edustajista on Eclipse, joka tarjoaa Java-kehitysympäristön ja lukuisia palveluita tarjoavan ohjelmistoalustan, sekä Java-ohjelmointikieli, joka virtuaalikoneensa ansiosta mahdollistaa käyttöjärjestelmästä riippumattomien ohjelmistojen toteuttamisen. Eclipse-alusta on myös toteutettu usealle eri käyttöjärjestelmälle, joten sitä hyödyntävät ohjelmistot ovat suoraan ajettavissa useissa eri käyttöjärjestelmissä.

Tässä työssä lähdettiin tutkimaan, miten käytännössä voitaisiin siirtää olemassa olevia natiiveja ohjelmia Eclipse-alustalle siten, että myös olemassa olevia testattuja ja toimivia ohjelmiston osia voitaisiin hyödyntää ja näin voitaisiin kenties välttyä merkittäviltä muutoksilta itse ohjelmiston toimivuuteen ja laatuun. Siirrettäviä ohjelmistoja on mahdollisesti kehitetty ja korjailtu jo useita vuosia, ja ne ovat ajan

myötä saavuttaneet tietyn kypsyystason ja vakauden, joten koko ohjelmiston kirjoittaminen täysin uudestaan Eclipse-alustalle tarkoittaisi pahimmillaan sitä, että sama laatutaso saavutettaisiin vasta vuosien kuluttua.

Mahdollisia teknologioita ongelman ratkaisemiseksi on useita, mutta tämän työn aiheeksi valittiin yksi mahdollinen, JNI-rajapinta, lähempään tarkasteluun. Työssä esitellään rajapinnan tarjoamia ominaisuuksia ja lopuksi myös käytännön toteutus aiheesta. Toteutetut esimerkit on tehty C/C++ -kielellä, mutta samaa teknologiaa on mahdollista soveltaa myös muiden ohjelmointikielten osalta.

1.2 Työn taustaa

Tämä insinöörityö on jatkoa keväällä 2005 tehtyihin kolmeen insinöörityöhön, joissa tutkittiin Eclipse-ympäristöä hieman yleisemmällä tasolla. Insinöörityössä ”Eclipsen soveltuvuus ensisijaiseksi kehitysympäristöksi” Sampo Kuisma tutki Eclipse-projektia kokonaisuutena. Insinöörityössä ”Eclipse-alusta ja SWT-kirjasto” Ari Paasonen tutki Eclipse-sovelluskehitysalustaa sekä siihen liittyviä käyttöliittymäkirjastoja ja niiden toimintaa. Työssä ”MFC/Stingray-sovelluksen uudelleentoteutus Eclipse-alustalla” puolestaan Ari Kilponen tutki perinteisen Windows- ja Eclipse-sovelluksen eroja sekä tarkasteli työmääriä siirryttäessä Windows-pohjaisesta sovelluskehitysympäristöstä Eclipse-ympäristöön. Tämä insinöörityö ei ole varsinaisesti jatkoa Ari Kilposen tekemään insinöörityöhön, mutta tarkastelee kuitenkin samaa aihetta teknisen toteutuksen näkökulmasta valitun teknologian avulla.

2 Eclipse ja SWT lyhyesti

2.1 Eclipse

Monesti Eclipse ymmärretään hieman virheellisesti pelkäksi Java-kehitysympäristöksi. Oikeastaan kyseessä on avoimen lähdekoodin yhteisö, jonka projektit ovat keskittyneet tuottamaan toimittajasta riippumatonta avointa kehitysympäristöä ja ohjelmistorunkoja

ohjelmistokehityksen tueksi. Se tarjoaa monipuolisen liitännäisiin perustuvan ohjelmistokehyksen, joka helpottaa ohjelmistojen kehitystä, integroimista ja hyödyntämistä säästäen näin aikaa ja rahaa. Eclipse-alusta on kirjoitettu Java-kielellä ja on saatavilla Linux-, HP-UX-, AIX-, Solaris-, QNX-, Mac OS X- ja Windows-käyttöjärjestelmille.

Alun perin Eclipsen perustivat vuonna 2001 useat alan johtavat yritykset, kuten Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft ja Webgain. Vuoden 2003 loppuun mennessä tähän yhteenliittymään kuului jo yli 80 jäsentä. Vuonna 2004 Eclipse saavutti nykyisen muotonsa, kun siitä tehtiin voittoa tuottamaton avoimen lähdekoodin yhteisö. Kaikki yhteisön tuottama teknologia ja lähdekoodi ovat vapaasti saatavilla EPL-lisenssin alaisuudessa. Nykyään Eclipse-projekti koostuu yhdeksästä suuresta avoimen lähdekoodin projektista, jotka sisältävät yli viisikymmentä pienempää aliprojektia. Näitä kehitetään yli 115 yrityksen avustuksella. [1.]

Eclipse-alustan toiminnallisuutta ja rakennetta on kuvattu hieman tarkemmin Ari Kilposen insinöörityössä ”MFC/Stingray-sovelluksen uudelleentoteutus Eclipse-alustalle” [2, s. 19].

2.2 SWT

SWT on lyhenne sanoista Standard Widget Toolkit, joka on Eclipse-alustan tarjoama käyttöliittymäkirjasto. Perinteisistä Javan AWT- ja Swing -komponenteista poiketen SWT tarjoaa pääsyn käyttöjärjestelmän natiiveihin käyttöliittymäkomponentteihin JNI-rajapinnan avulla. Tällä saavutetaan se, että SWT:n avulla toteutetut käyttöliittymät muistuttavat täysin käytettävän käyttöjärjestelmän natiiveja ohjelmia. Tarkempi kuvaus SWT:n toiminnasta, sen tarjoamista käyttöliittymäkomponenteista ja niiden eroista MFC:n ja Stringray-kirjaston komponentteihin löytyy myös Ari Kilposen insinöörityöstä [1, s. 23].

3 JNI-rajapinta

3.1 Historia

Java on alun perin suunniteltu järjestelmäriippumattomaksi kieleksi. Ohjelma, joka kirjoitetaan kerran ja voidaan käyttää kaikissa eri käyttöjärjestelmissä, on tietysti hieno tavoite, mutta tietyissä tapauksissa täysin mahdoton toteuttaa. Kuten Rob Gordon [3, s. 2] kirjassaan varsin lakonisesti toteaa: ”Niin puoleensavetävä kuin järjestelmäriippumattomuus onkin, siitä tulee todellisuutta vasta kun koko tietokoneteollisuus siirtyy käyttämään yhtä järjestelmää”. Tällä hän tarkoittaa lähinnä sitä, että eri käyttöjärjestelmissä tulee aina olemaan tiettyjä piirteitä joita Java-kieli ei tarjoa jolloin joudutaan turvautumaan toiseen matalamman tason kieleen. Toisaalta myös ohjelmistoja tekevillä yrityksillä saattaa usein olla toisella ohjelmointikielellä ohjelmoituja, jo vuosia käytössä olleita, täysin toimivia ohjelmistokomponentteja, joita ei tietenkään haluta tehdä uudelleen. [3, s. 1–2.]

Huolimatta Java-kielen alkuperäisestä toiminta-ajatuksesta sen suunnittelussa on alusta alkaen nähty tarve myös natiivien kirjastojen hyödyntämiseen ja näin ollen se onkin tehty mahdolliseksi sen ensimmäisestä versiosta lähtien. Jo ensimmäinen julkaisu Java-alustasta, Java Development Kit release 1.0, sisälsi mahdollisuuden natiivien funktioiden kutsumiseen. Tässä kyseisessä versiossa oli kuitenkin kaksi suurta ongelmaa.

- Natiivi ohjelma sai tiedot Java-objektista ja sen sisältämistä kentistä C-kielisenä rakenteena. Java-virtuaalikoneen spesifikaatio ei kuitenkaan määrittele miten rakenteen sisältö tulee järjestellä muistiin, minkä vuoksi eri virtuaalikoneiden välillä kirjasto jouduttiin mahdollisesti kääntämään uudelleen riippuen virtuaalikoneen toteutuksesta.

- Natiivi ohjelma oli riippuvainen virtuaalikoneen ns. konservatiivisesta roskienkeruusta, koska natiivi ohjelma sai osoittimet java-olioihin ja tämän

vuoksi virtuaalikone ei voinut siirtää objekteja toiseen muistiosoitteeseen eikä poistaa niitä muistista kuin vasta ohjelman suorituksen loputtua.

JNI luotiin korjaamaan edellä mainitut ongelmat, ja siitä julkaistiin ensimmäinen versio JDK release 1.1 yhteydessä. Versio käytti kuitenkin vielä sisäisesti aiempaa rajapintaa toteutuksessaan. JNI-rajapinta kirjoitettiin kokonaan uudestaan seuraavaan Java 2 SDK release 1.2 versioon, joka noudattaa täysin JNI-määrityksiä. Tämä versio toi myös mukanaan binääriyhteensopivuuden natiivien kirjastoiden kanssa, mikä tarkoittaa sitä, että sama kirjasto toimii kaikkien eri virtuaalikoneiden kanssa täysin riippumatta virtuaalikoneen sisäisestä toteutuksesta. Seuraavien JNI-versioiden on myös luvattu olevan alaspäin binääriyhteensopivia aiempien versioiden kanssa. [4, s. 7–8.]

3.2 Toiminnallisuus

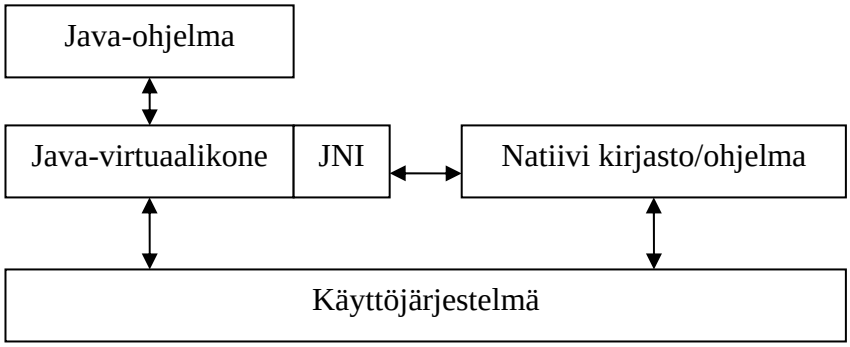
JNI-rajapinta on suunniteltu käsittelemään tilanteita, joissa on tarvetta yhdistää Java-ohjelmia natiivien ohjelmien kanssa. Kaksisuuntaisena rajapintana sitä on myös mahdollista käyttää kahdella eri tavalla.

- Java-kielen metodeille voidaan kirjoittaa toteutukset natiivilla kielellä. Java-ohjelma kutsuu metodeita aivan kuten se kutsuisi normaaleita Java-metodeita, mutta virtuaalikone ohjaa kutsut natiiviin kirjastoon.

- Natiivi ohjelma voi luoda Java-virtuaalikoneen Invocation-rajapinnan avulla. Tämä mahdollistaa Java-virtuaalikoneen luomisen natiivista ohjelmasta käsin, jonka avulla natiivi ohjelma voi ajaa Java-ohjelmia. Tämä on laajalti käytössä esimerkiksi Internet-selaimissa, jotka osaavat ajaa Java-sovelmia.

Yksinkertaistettuna JNI-rajapintaa hyödyntävä ohjelmisto voidaan esittää kuvan 1 mukaisena, jossa Java-ohjelmaa ajetaan käyttöjärjestelmän päällä olevan Java-

virtuaalikoneen avulla, josta on yhteys JNI-rajapinnan kautta käyttöjärjestelmän päällä ajettavaan natiiviin kirjastoon. [4, s. 5].



Kuva 1. JNI-rajapintaa hyödyntävän ohjelmiston rakenne

JNI-rajapinta esiintyy ohjelmoijalle Java-ohjelman sekä natiivin kirjaston perspektiivistä hieman eri tavalla. Tästä syystä rajapinnan esittely on tässä työssä jaettu kahteen lukuun, jotka kuvaavat erikseen nämä tapaukset. Rajapinnan esittelyssä ei myöskään ole pyritty esittelemään kaikkia JNI-rajapinnan tarjoamia ominaisuuksia, vaan keskitytty lähinnä perustoiminnallisuuksiin.

4 JNI-rajapinnan käsittely natiivista ohjelmakoodista

4.1 Perustietotyypit

Kaikissa ohjelmointikielissä on määriteltynä perustietotyypit, jotka yleensä määrittelevät mahdollisen tietosisällön ja kyseiselle tietotyypille varattavan muistin määrän. Java-kielessä määritellyt tietotyypit ovat ympäristöstä riippumatta aina samat. C-kielen standardointi ei sen sijaan määrittele kovinkaan tarkasti tietotyyppien kokoa vaan jättää valinnan kääntäjän vastuulle. Tämä saa aikaan tietysti sen, että vaikka Java- ja C-kielissä on molemmissa samannimisiä tietotyyppejä, niin kyseessä ei välttämättä kuitenkaan ole samankokoinen tietotyyppi. Yhtenäistäkseen tietotyypit natiivien

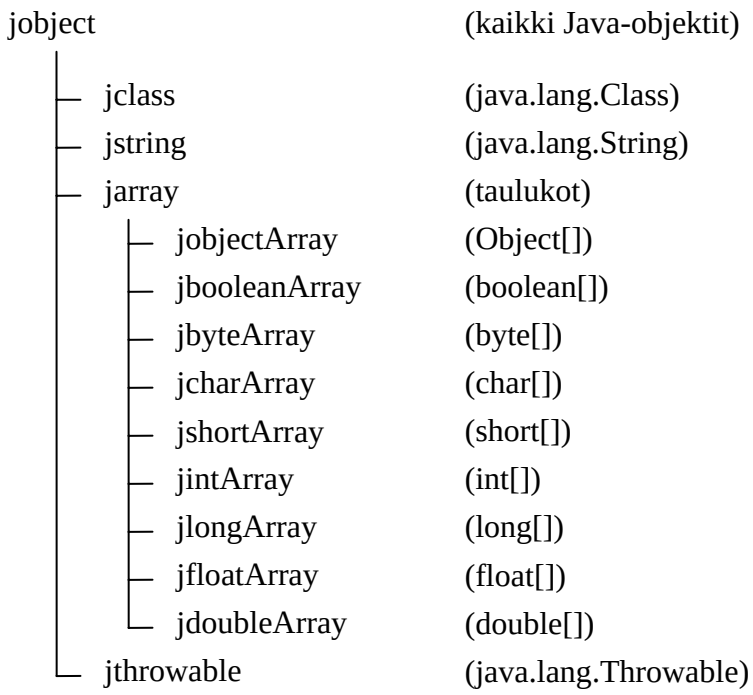
ohjelmien kanssa on JNI-rajapintaa varten määritelty taulukon 1 mukaiset natiivit vastineet Java-kielen tietotyypeille, jotka vastaavat aina Java-kielen tietotyyppejä käyttöjärjestelmästä riippumatta.

Taulukko 1. Java tietotyypit ja niiden natiivit vastineet [4, s. 165]

Java tietotyyppi	Natiivi tietotyyppi	Kuvaus
boolean	jboolean	unsigned 8 bit
byte	jbyte	signed 8 bit
char	jchar	unsigned 16 bit
short	jshort	signed 16 bit
int	jint	signed 32 bit
long	jlong	signed 64 bit
float	jfloat	32 bit (IEEE 754)
double	jdouble	64 bit (IEEE 754)

4.2 Referenssitietotyypit

Perustietotyyppien lisäksi JNI-rajapinta esittelee tietotyyppejä kuvaamaan viittauksia Java-luokkiin ja niiden instansseihin. Kuten kuvasta 2 nähdään, myös natiivin ohjelman puolella määrittely noudattaa täysin samaa luokkahierarkiaa kuin Javassa, eli kaikkien luokkien kantaluokkana toimii jobject-luokka. Kaikki käsiteltävät objektit ovat periaatteessa jobject-tyyppisiä, mutta niiden lisäksi on natiivin ohjelman puolelle määritelty erikseen jstring-tyyppi helpottamaan merkkijonojen käsittelyä, jarray-luokka helpottamaan taulukoiden käsittelyä ja jthrowable-luokka erittelemään poikkeukset muista objekteista.



Kuva 2. Javan referenssityyppien natiivit vastineet ja niiden hierarkia [4, s. 166]

Riippuen siitä käytetäänkö C-kielen kääntäjää vai C++ -kääntäjää, JNI määrittelee referenssityypit hieman eri tavalla, mutta periaate on molemmissa sama. Kuten kuvasta 3 nähdään, ovat kaikki C-kielen tyyppimääritykset käytännössä osoittimia tyhjään tietorakenteeseen. C++ -kielen tapauksessa, kuten kuvasta 4 näemme, määritys sisältää luokkien hierarkian, mutta kantaluokka jobject on määritelty luokaksi, jolla ei ole jäsenmuuttujia eikä -funktioita. [4, s. 166–167.]

```
struct _jobject;
typedef struct _jobject* jobject;

typedef jobject jclass;
typedef jobject jthrowable;
```

Kuva 3. C-kielisiä määrityksiä Java-referenssityypeistä

```
class _jobject {};
class _jclass : public _jobject {};
class _jthrowable : public _jobject {};

typedef _jobject* jobject;
typedef _jclass* jclass;
typedef _jthrowable* jthrowable;
```

Kuva 4. C++ -kielisiä määrityksiä Java-referenssityypeistä

Natiivin ohjelman kannalta JNI-rajapinnan referenssityyppejä ei voi suoraan käyttää mihinkään, koska vaikka ne on osoittimiksi määritelty, ne eivät osoita varsinaisesti mihinkään luokkaan vaan sisältävät virtuaalikoneen määrittelemän informaation tietystä luokasta tai sen instanssista. Kaikki Java-metodien natiivit toteutukset saavat yhden jobject-tyyppisen parametrin, joka on referenssi kutsun tehneeseen objektiin. Tämä kyseinen parametri vastaa siis Java-kielen this-avainsanaa, jonka avulla voidaan käsitellä esimerkiksi kyseisen instanssin jäsenmuuttujia myös natiivin ohjelman puolelta. Tämän lisäksi JNI-rajapinta sisältää lukuisia palvelufunktioita, joiden avulla voidaan pyytää virtuaalikoneelta referenssejä haluttuihin luokkiin.

4.3 Luokat ja niiden instanssit

Natiivi ohjelma käsittelee kaikkia Java-objekteja referenssien avulla. Referenssejä luokkiin voidaan hakea JNI-rajapinnan funktioiden avulla joko pyytämällä jonkin uuden luokan referenssi luokan nimen avulla kuvan 5 mukaisesti tai pyytämällä referenssi jonkin olemassa olevan objektin luokkaan kuvan 6 mukaisesti. Huomion arvoista kuvassa 5 on se, että erotinmerkkinä luokan nimessä käytetään natiivin ohjelman puolella vinoviivaa pisteen sijasta.

```
jclass cls = pJNIEnv->FindClass(”java/lang/Integer”);
```

Kuva 5. Luokka-referenssin hakeminen luokan nimen avulla

```
jclass cls = pJNIEnv->GetObjectClass( obj );
```

Kuva 6. Luokka-referenssin hakeminen instanssi-referenssin avulla

Luokan referenssin avulla voidaan luoda luokasta uusia instansseja muutamin eri tavoin. Yksinkertaisin tapa on luoda instanssi kuvan 7 mukaisesti. Tässä tapauksessa on kuitenkin syytä huomioida, että luokan muodostajaa ei automaattisesti kutsuta, vaan ohjelmoijan tulee pitää itse huoli siitä, että luokan instanssi tulee oikein alustettua.

```
jobject obj = pJNIEnv->AllocObject( cls );
```

Kuva 7. Uuden instanssin luominen luokasta

Toinen tapa on luoda instanssi kuvan 8 mukaisesti komennolla, jolle annetaan myös tunniste luokan muodostajaan, joka virtuaalikoneen tulee suorittaa instanssin luomisen yhteydessä. Luokan muodostajan kohdalla on myös hyvä huomioida, että muodostajia voi olla useita kappaleita eri parametreilla, jolloin ne myös erotetaan toisistaan GetMethodID-funktiolle parametrina annettavalla tietotyyppikuvaajalla, joita käsitellään tarkemmin luvussa 4.4.

```
jmethodID mid;
mid = pJNIEnv->GetMethodID( cls, “<init>”, “()V”);
jobject obj = pJNIEnv->NewObject( cls, mid );
```

Kuva 8. Uuden instanssin luominen luokan muodostajan avulla

Luokan instanssien luomisen lisäksi JNI-rajapintaan on toteutettu taulukon 2 mukaiset apufunktiot, joiden avulla voidaan kysyä, onko jokin objekti tietyn luokan instanssi, viittaako kaksi referenssiä samaan objektiin ja voidaanko objekti sijoittaa toiseen objektiin.

Taulukko 2. Java-referenssityyppien vertailuun käytettäviä funktioita

JNI-funktion prototyyppi	Kuvaus
jboolean IsSameObject(jobject obj1, jobject obj2)	Funktio palauttaa totuusarvon siitä, viittaavatko parametrina annetut referenssit samaan objektiin.
jboolean IsInstanceOf(jobject obj, jclass clazz)	Funktio palauttaa totuusarvon siitä, onko ensimmäisenä parametrina annettu objekti-referenssi toisena parametrina annetun luokan instanssi.
jboolean IsAssignableFrom(jclass sub, jclass sup)	Funktio palauttaa totuusarvon siitä, voidaanko ensimmäisenä parametrina annettu luokka sijoittaa toisena parametrina annettuun luokkaan.

4.4 Tekstimuotoiset tietotyyppikuvaukset

Kutsuttaessa natiivin ohjelman puolelta Java-metodia tai käsiteltäessä jonkin luokan jäsenmuuttujia, on ohjelmoijan annettava JNI-spesifikaation määrittelemä tekstimuotoinen kuvaus Java-ohjelman puolella määritellyistä tietotyypeistä. Määritellyt perustietotyyppien kuvaukset on kuvattuna taulukossa 3. Taulukkoon on lisättynä myös void-tietotyyppi, jota käytetään Java-metodeita käsiteltäessä kuvaamaan sitä, että kyseisellä metodilla ei ole paluuarvoa.

Taulukko 3. Tekstimuotoiset tietotyyppikuvaukset perustietotyypeille [4, s. 169]

Kentän kuvaaja	Java-tietotyyppi
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
V	void

Perustietotyyppien lisäksi JNI-spesifikaatio määrittelee myös, miten referenssityypit tulee kuvata. Referenssityypin kuvaus alkaa ”L”-merkillä, jonka jälkeen kirjoitetaan ilman välimerkkejä Java-luokan nimi vinoviiva erotinmerkkinä. Määritys lopetetaan puolipisteellä. Tämän lisäksi voidaan myös määritellä taulukoita lisäämällä ”[”-merkki kuvauksen alkuun, jolloin määritys tarkoittaa taulukkoa tämän merkin jälkeen määritellystä tietotyypistä. Taulukossa 4 on kuvattuna muutamia tyypillisiä esimerkkejä referenssityyppien ja taulukoiden määrityksistä.

Taulukko 4. Tekstimuotoiset tietotyyppikuvaukset referenssityypeille [4, s. 170]

Kentän kuvaaja	Java-tietotyyppi
“Ljava/lang/String;”	String
“[I”	int[]
“[Ljava/lang/Object;”	Object[]

4.5 Luokan kentät

Java-ohjelmointikieli tukee kahdentyyppisiä luokkamuuttujia. Kuten monissa muissakin olio-oriointuneissa kielissä, muuttujat voivat olla instanssikohtaisia jäsenmuuttujia tai staattisia muuttujia, jotka sidottu itse luokkaan luokan instanssin sijasta. JNI-rajapinta tarjoaa mahdollisuuden näiden molempien muuttujatyypin lukemiseen ja kirjoittamiseen. Luokkien muuttujia käsitellään id-numeron perusteella, jota varten myös JNI-määrityksiin on lisätty oma jfieldID-tietotyyppi. Kentän id-numero voidaan pyytää JNI-rajapinnan GetFieldID-funtion avulla ja haluttaessa lukea tai kirjoittaa kyseiseen kenttään saatu id-numero annetaan parametrina kyseiselle funktiolle. [4, s. 41–42.]

Kuvassa 9 on kuvattuna instanssikohtaisten kenttien käsittelemiseen tarvittavat toimenpiteet. Asian yksinkertaistamiseksi esimerkistä on kuitenkin jätetty pois virheiden tarkistukset, jotka käytännössä tietenkin tulee tehdä. Esimerkissä on jobject-tyyppinen muuttuja, jolle haetaan aluksi referenssi luokkaan jota se edustaa. Seuraavaksi haetaan luokka-referenssin avulla nimi-kentän id-numero. Haettava kenttä on esimerkissä String-tyyppinen muuttuja. Tämän jälkeen kentän arvo voidaan lukea käyttäen GetObjectField-funktiota ja siihen voidaan myös tallentaa uusi arvo käyttäen SetObjectField-funktiota. Staattisten kenttien käsittely on muutoin aivan vastaava, mutta tällöin tulee kenttää luettaessa käyttää GetStaticObjectField- ja vastaavasti kirjoitettaessa SetStaticObjectField-funktiota kuvan 10 mukaisesti. Käytettäessä staattisia muuttujia on myös syytä huomioida, että parametriksi annetaan itse luokan referenssi luokan instanssin referenssin sijasta.

```
jfieldID fid;
jstring jstr;

jclass cls = pEnv->GetObjectClass( obj );
fid = pEnv->GetFieldID(cls,”nimi”,”Ljava/lang/String;”);

/* Kentän lukeminen */
jstr = pEnv->GetObjectField(obj,fid);
/* Kentän kirjoitus */
pEnv->SetObjectField(obj,fid,jstr);
```

Kuva 9. Luokan kenttien lukeminen ja kirjoittaminen

```
/* Kentän lukeminen */
jstr = pEnv->GetStaticObjectField(cls,fid);
/* Kentän kirjoitus */
pEnv->SetStaticObjectField(cls,fid,jstr);
```

Kuva 10. Luokan staattisien kenttien lukeminen ja kirjoittaminen

Natiivin ohjelman puolelta Java-luokkien kenttien lukemiseen ja kirjoittamiseen käytetään JNI-rajapintaan määriteltyjä luku- ja kirjoitusfunktioita. Rajapinta määrittelee erilliset funktiot riippuen käsiteltävästä tietotyypistä taulukon 5 mukaisesti. Kuten taulukosta nähdään, kaikille perustietotyypeille on omat funktiot, ja sen lisäksi funktiot object-tyyppiselle muuttujalle, jonka avulla voidaan käsitellä kaikkia referenssityyppisiä kenttiä, kuten esimerkiksi kuvan 9 esimerkissä käytettävää String-tietotyyppiä. Kaikista taulukon 1 funktioista on olemassa erilliset versiot staattisien kenttien lukemiseen ja kirjoittamiseen. Kuten mainittua, katsomalla kuvan 9 ja 10 esimerkkejä voidaan havaita, että kentän lukemisessa käytettävä funktio GetObjectField muuttuu staattista kenttää

luettaessa muotoon GetStaticObjectField. Kaikki staattisten kenttien käsittelyyn käytettävät funktiot noudattavat täysin samaa nimeämiskäytäntöä, ja näin ollen kaikista taulukon 5 funktioista on olemassa vastaavalla tavalla nimetty versio staattisille kentille.

Taulukko 5. Kenttien käsittelyyn käytettävät funktiot tietotyypeittäin [4, s. 223]

Lukufunktio	Kirjoitusfunktio	Natiivi tietotyyppi
GetObjectField	SetObjectField	jobject
GetBooleanField	SetBooleanField	jboolean
GetByteField	SetByteField	jbyte
GetCharField	SetCharField	jchar
GetShortField	SetShortField	jshort
GetIntField	SetIntField	jint
GetLongField	SetLongField	jlong
GetFloatField	SetFloatField	jfloat
GetDoubleField	SetDoubleField	jdouble

4.6 Luokan metodit

Java-metodien käsittely JNI-rajapinnan avulla noudattaa hyvin pitkälti samaa ideologiaa kuin luvussa 4.5 esitetty luokan kenttien käsittely. Metodeista voi olla olemassa instanssikohtaisia metodeja, samoin kuin staattisia metodeja, jotka on sidottu itse luokkaan luokan instanssin sijasta. JNI-rajapinnan avulla on mahdollista kutsua molemman tyyppisiä metodeja. Metodeja käsitellään id-numeroiden avulla, jota varten JNI-määrityksiin on lisätty kenttien käsittelyssä käytettävää jfieldID-tietotyyppiä vastaava jmethodID-tietotyyppi. Metodin id-numero voidaan hakea käyttäen JNI-rajapinnan GetMethodId-funktiota ja saatu id-numero annetaan parametrina metodia kutsuttaessa.

Metodien kutsuminen voidaan yksinkertaistettuna esittää kuvan 11 mukaisesti, jossa aluksi pyydetään luokan instanssista referenssi itse luokkaan ja luokasta haetaan id-

numero haluttuun funktioon. Lopuksi funktiota kutsutaan käyttäen parametreina referenssiä luokan instanssiin, johon kutsu kohdistuu ja id-numeroa kutsuttavaan metodiin. Esimerkissä kutsuttava Java-metodi ei ota mitään parametreja eikä myöskään palauta mitään.

```
jmethodID mid;

jclass cls = pEnv->GetObjectClass( obj );
mid = pEnv->GetMethodID(cls,”metodi”,”()V”);

pEnv->CallVoidMethod(obj,mid);
```

Kuva 11. Java-metodin kutsuminen JNI-rajapinnan avulla

Aivan kuten luokan kenttiä käsiteltäessä, myös metodeille tulee määritellä sen parametrien ja paluuarvon tietotyypit luvussa 4.4 kuvatuilla tekstimuotoisilla tietotyyppikuvaajilla. Luvussa 4.5 kenttiä käsiteltäessä riitti, kun määritteli käsiteltävän muuttujan tietotyypin. Metodilla taas voi olla useita erityyppisiä parametreja, ja se voi palauttaa joko jonkin perustietotyypin tai luokan referenssin. JNI-spesifikaation määrittelee metodien kuvaajan muodostuvan sulkujen sisällä määritellyistä parametrien tietotyypeistä, joiden jälkeen sulkujen ulkopuolelle merkitään metodin paluuarvon tietotyyppi. Taulukossa 6 on esitettynä muutamia esimerkkejä erilaisista metodin kuvaajista ja niitä vastaavista Java-metodeista. Kuten taulukon 6 ensimmäiseltä riviltä nähdään, on metodin kuvaaja yksinkertaimmillaan tyhjät sulut sekä V-kirjain merkitsemässä sitä, että metodilla ei ole paluuarvoa. [4, s. 48.]

Taulukko 6. Esimerkkejä metodien kuvaajista

Natiivi metodin kuvaaja	Java-metodin prototyyppi
”()V”	void metodi();
”(I)V”	void metodi(int parametri);
”(D)J”	long metodi(double parametri);
”(Ljava/lang/String;)V”	void metodi(String parametri);
“([Ljava/lang/String;)Ljava/lang/String;”	String metodi(String[] parametri);

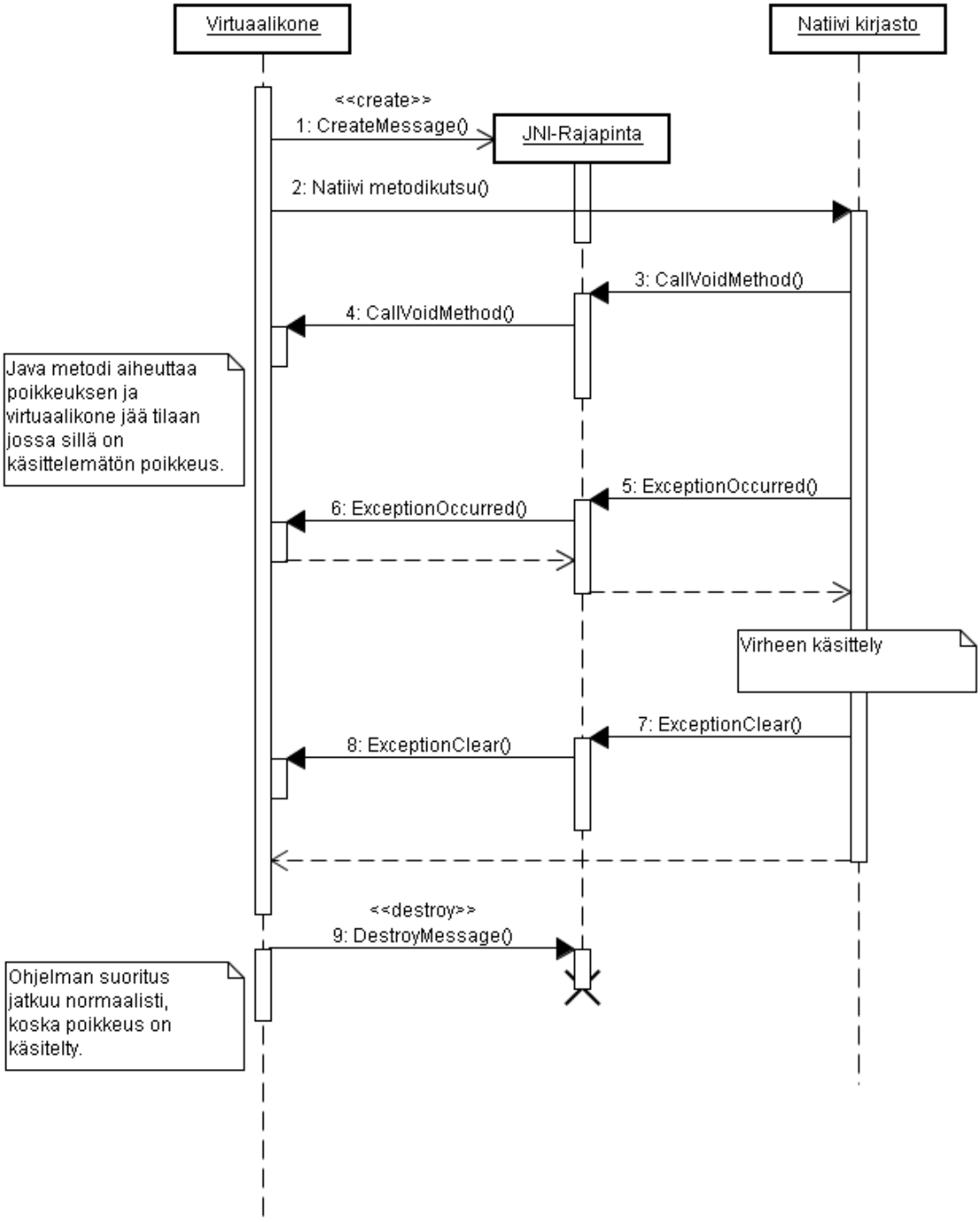
4.7 Poikkeukset

Java-kielessä käsitellään virhetilanteet tyypillisesti poikkeuksien avulla. Mikäli esimerkiksi tiedoston avaaminen syystä tai toisesta epäonnistuu, virtuaalikone heittää ajettavalle ohjelmalle ilmoituksen tapahtuneesta virheestä poikkeuksena, joka voidaan ohjelmassa käsitellä halutulla tavalla. Natiivien ohjelmien kohdalla asia ei olekaan aivan näin yksinkertainen, koska kielestä ja ympäristöstä riippuen virhetilanteita saatetaan käsitellä usein eri tavoin. Natiivia ohjelmaa ajettaessa ei virtuaalikoneella myöskään ole kontrollia ohjelman suorituksesta, joten natiivin ohjelman puolella virhetilanteiden käsittely on jätettyä täysin ohjelmoijan vastuulle. Kuten taulukosta 7 näemme, JNI-rajapinta tarjoaa kuitenkin ohjelmoijalle tarvittavat metodit, joiden avulla virhetilanteet voidaan ilmoittaa virtuaalikoneelle poikkeuksina, samoin metodit, joilla voidaan kysyä virtuaalikoneelta, onko poikkeuksia tapahtunut.

Taulukko 7. Poikkeusten käsittelyyn käytettävät JNI-funktiot

JNI-funktio	Funktion kuvaus
Throw	Heittää parametrina annetun jthrowable tyyppisen objektin poikkeuksena
ThrowNew	Luo parametrina annetusta luokasta uuden instanssin ja heittää sen poikkeuksena. Annetun luokan tulee periytyä jthrowable kantaluokasta.
ExceptionOccurred	Tarkistaa virtuaalikoneelta onko poikkeus tapahtunut ja palauttaa referenssin poikkeukseen.
ExceptionCheck	Tarkistaa virtuaalikoneelta onko poikkeus tapahtunut ja palauttaa tilan totuusarvona.
ExceptionDescribe	Tulostaa ajettavassa säikeessä tapahtuneen poikkeuksen ja pyyhkii poikkeuksen.
ExceptionClear	Pyyhkii poikkeuksen ajettavasta säikeestä.
FatalError	Heittää poikkeuksen josta virtuaalikoneen ei odoteta toipuvan. Virhe tulostetaan ja virtuaalikone lopetetaan.

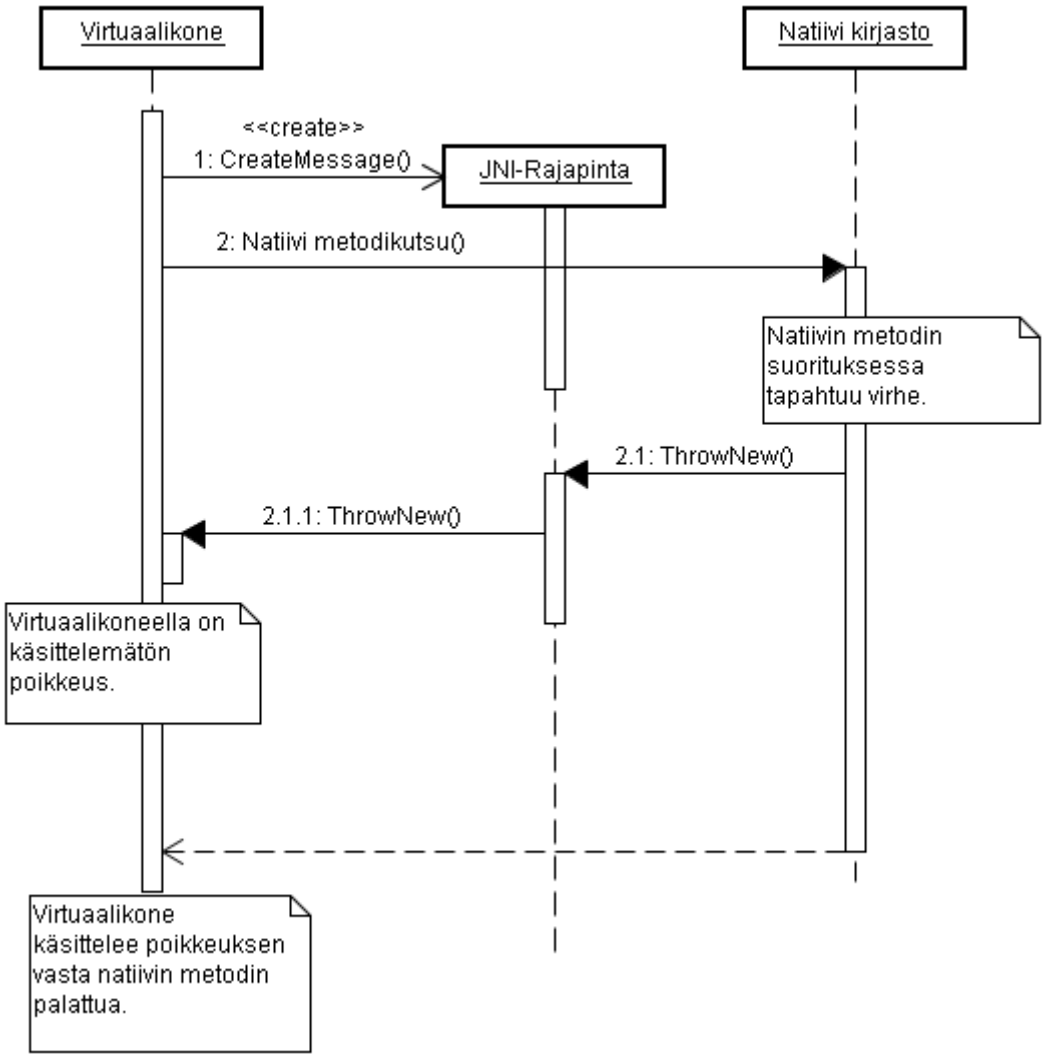
Kutsuttaessa Java-metodia natiivin ohjelman puolelta virtuaalikoneelle voi syntyä virheen seurauksena poikkeus. Perinteisesti Java-ohjelmassa ohjelman suoritus siirtyy tämänkaltaisessa tilanteessa automaattisesti poikkeuksienkäsittelyrutiinille, joka ohjelmasta riippuen käsittelee virhetilanteen halutulla tavalla. JNI-rajapinnan tapauksessa virtuaalikone ei kuitenkaan voi kontrolloida natiivin ohjelman suoritusta, joten virtuaalikone vain merkitsee sisäisesti poikkeuksen tapahtuneeksi ja palaa metodista normaalisti. Virhetilanne voidaan tämän jälkeen käsitellä natiivin ohjelman puolella kuvan 12 mukaisesti ja pyyhkiä poikkeustilanne, jolloin suorituksen palattua Java-ohjelman puolelle ohjelman suoritus jatkuu aivan normaalisti.



Kuva 12. Poikkeusten käsittely natiivin ohjelman puolelta

Kuten mainittua, JNI-rajapinta mahdollistaa myös poikkeuksien käyttämisen virheiden käsittelyyn myös natiivin ohjelman puolelta. Kuten kuvasta 13 nähdään, tässäkään tapauksessa Java-virtuaalikone ei kykene kontrolloimaan ohjelman suoritusta silloin

kun suoritetaan metodia natiivin ohjelman puolella. Ilmoitus poikkeuksesta natiivin ohjelman puolelta aiheuttaa sen, että virtuaalikone merkitsee sisäisesti poikkeuksen tapahtuneeksi ja käsittelee sen vasta natiivin metodin palattua.



Kuva 13. Poikkeuksesta ilmoittaminen natiivin ohjelman puolelta

4.8 Paikalliset ja globaalit referenssit

JNI-rajapinnan avulla käsiteltävät referenssityypit voidaan jakaa kolmeen eri kategoriaan, jotka poikkeavat hieman toisistaan niiden näkyvyyden, käsittelyn ja roskienkeruun kannalta. Kaikille ohjelmointia harrastaneille paikalliset ja globaalit muuttujat lienee tuttuja käsitteitä. Paikalliset ja globaalit referenssit ovat niin kutsuttuja vahvoja referenssejä, joka tarkoittaa sitä, että mikäli muistissa olevaan objektiin on yksikin vahva referenssi, niin roskienkerääjän ei ole sallittua käydä sitä tuhoamassa. Näiden kahden referenssityypin lisäksi JNI-rajapintaan on määritelty heikko globaali referenssityyppi, joka vahvoista referensseistä poiketen sallii roskienkeruun, vaikka objektiin olisi referenssejä vielä jäljellä.

Useimmat JNI-funktiot palauttavat paikallisen referenssin objekteihin, ja ne ovat käytettävissä suoritettavan natiivin funktion ajan. Natiivin funktion päätyttyä Java-virtuaalikoneen roskienkerääjä voi hetkenä minä hyvänsä käydä poistamassa objektin muistista. Tästä syystä paikallisten referenssien tallentaminen esimerkiksi globaaliin muuttujaan natiivin ohjelman puolella on ehdottomasti kielletty. JNI-rajapinta tarjoaa kuitenkin mahdollisuuden tämänkaltaisten tilanteiden hoitamiseen taulukossa 8 kuvattujen funktioiden avulla. Näiden funktioiden avulla voidaan esimerkiksi edellä mainitussa tapauksessa luoda paikallisesta referenssistä globaali referenssi ja tallentaa se globaaliin muuttujaan natiivin ohjelman puolella. Kun asia hoidetaan tällä tavoin, virtuaalikone pysyy tietoisena siitä, että natiivin ohjelman puolella tiettyyn objektiin on tehty globaali referenssi eikä näin ollen yritä sitä tuhota ennen kuin tämä referenssi on vapautettu. [4, s. 61–65.]

Taulukko 8. JNI-rajapinnan tarjoamat funktiot referenssien käsittelyyn

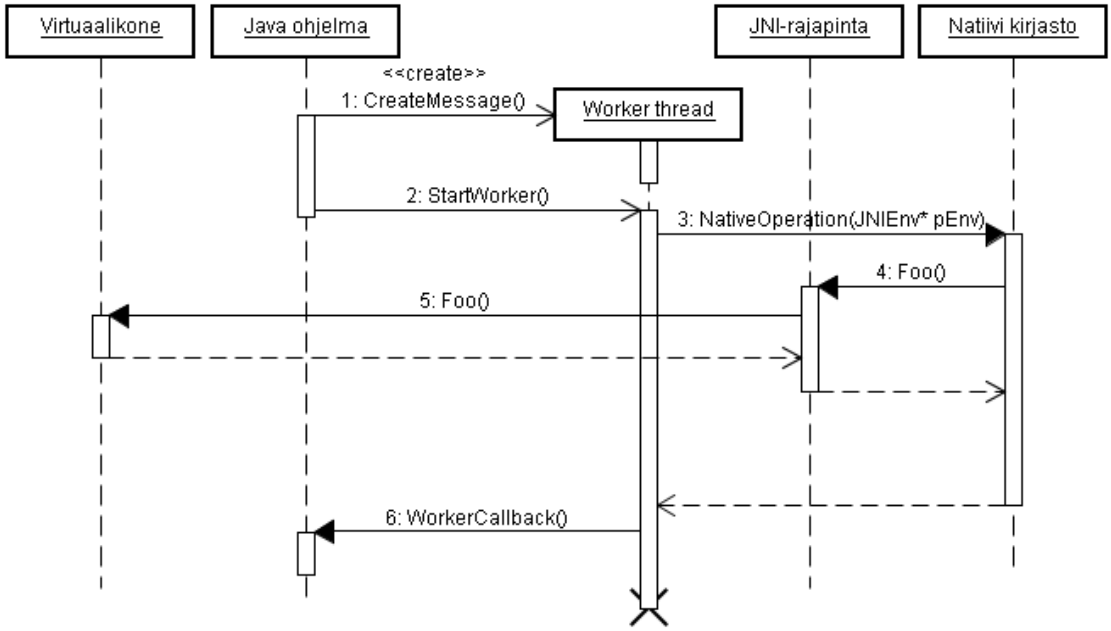
JNI-funktio	Funktion kuvaus
NewGlobalRef	Luo globaalin referenssin parametrina annetusta referenssistä
DeleteGlobalRef	Tuhoaa globaalin referenssin
NewWeakGlobalRef	Luo heikon referenssin parametrina annetusta referenssistä
DeleteWeakGlobalRef	Tuhoaa heikon referenssin
NewLocalRef	Luo paikallisen referenssin parametrina annetusta referenssistä
DeleteLocalRef	Tuhoaa paikallisen referenssin

4.9 Säikeistys ja synkronointi

Nykyisin moniydinprosessoreiden yleistyessä ohjelmistokehityksessä käytetään yhä suuremmassa määrin asynkronisia operaatioita hyväksi. Tämä tarkoittaa sitä, että ajettaessa yhtä ohjelmistoa saattaa se sisäisesti tehdä lukuisia asioita samanaikaisesti eri säikeissä. Monisäikeisen ohjelman toiminnan kannalta mahdollisuus säikeiden synkronointiin on yksi perusedellytyksistä. JNI-rajapintaa hyödyntävässä ohjelmassa, jossa on yhdistettynä Java-kieltä ja natiivia kieltä, on myös säikeiden luominen ja niiden synkronointi mahdollista niin Java-ohjelmasta kuin myös natiivista ohjelmasta käsin.

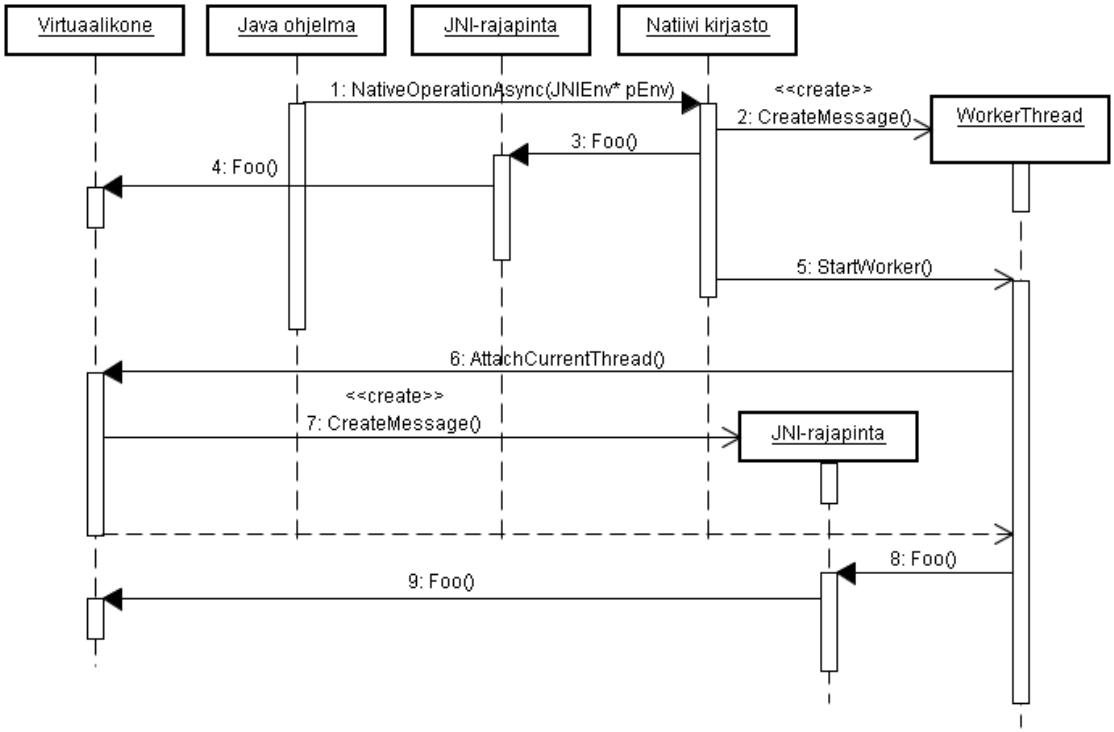
Säikeistykseen ei JNI-rajapinta sinällään tarjoa mitään apua, mutta ei myöskään pakota säikeiden käyttöä mihinkään tiettyyn muottiin. Näin ollen säikeitä voidaan luoda Java-ohjelman puolella kuin myös natiivin ohjelman puolella. Tämän lisäksi natiivista ohjelmasta käsin voidaan yhtä hyvin käyttää Javan tarjoamia luokkia säikeiden luomiseen kuin käyttöjärjestelmän tarjoamia natiiveja säikeitä.

Java-ohjelman puolelta voidaan asynkronisia operaatioita toteuttaa esimerkiksi kuvan 14 mukaisesti, jossa säie luodaan Java-ohjelman puolella ja se tekee synkronisen kutsun natiiviin metodiin. Tässä tapauksessa ei natiivin ohjelman puolella tarvitse säikeistä ja synkronoinneista huolehtia ollenkaan, koska kaikki niihin liittyvä toteutetaan Javan puolella.



Kuva 14. Asynkroninen operaatio Javalla toteutettuna

Natiivit metodit voivat myös käyttää sisäisesti natiiveja säikeitä asynkronisten operaatioiden toteuttamiseen. Kuten kuvan 15 esimerkissä nähdään, Java-ohjelmasta kutsutaan natiivia metodia, joka luo natiivin säikeen. Säikeen käynnistämisen jälkeen suoritus palaa normaalisti Java-ohjelman puolelle. Kuvan 14 esimerkkiin verrattuna huomion arvoista tässä tapauksessa on se, että natiivin ohjelman puolelle välitettävä osoitin JNI-rajapinnan palveluihin on säiekohtainen eikä sitä näin ollen ole sallittua välittää suoraan uudelle säikeelle vaan uuden säikeen tulee käydä liittämässä itsensä virtuaalikoneeseen, jolloin se saa käyttöönsä oman osoittimen JNI-rajapintaan.



Kuva 15. Asynkroninen operaatio natiivilla kielellä toteutettuna

Yksinkertaisin Java-kielen tarjoama väline säikeiden synkronointiin lienee synchronized-avainsana, jonka avulla voidaan määritellä tietty metodi synkronoiduksi [5] kuvan 16 esimerkin mukaisesti. Tämä tarkoittaa sitä, että vain yksi säie pääsee suorittamaan kyseistä metodia kerrallaan. Mikäli toinen säie yrittää lähteä suorittamaan metodia samanaikaisesti, virtuaalikone asettaa säikeen odottamaan, että edellinen säie saa kyseisen metodin suoritettua ensin loppuun. Tämän avainsanan käyttö on sallittua myös natiivien metodien kohdalla, mikä tekee kokonaisten natiivien metodien synkronoimisen hyvin helpoksi.

```
synchronized void testMethod()
{
    // Synkronoitu ohjelmalohko
}
```

Kuva 16. Synkronoitu Java-metodi

Synchronized-avainsana tarjoaa yksinkertaiset mahdollisuuden tehdä tietty metodi synkroniseksi, mutta toisinaan on tilanteita, joissa vain pienempi ohjelmalohko tulee synkronoida. Kuten kuvasta 17 nähdään, on Java-kielessä mahdollista käyttää synchronized-avainsanaa myös pienemmän ohjelmalohkon synkronoimiseen [6]. Saman toiminnallisuuden mahdollistavat natiivin ohjelman puolelta JNI-rajapinnan tarjoamat MonitorEnter- ja MonitorExit-metodit kuvan 18 esimerkin mukaisesti. Käytännössä kuvan 18 esimerkki tekee siis käytännössä saman asian natiivin ohjelman puolelta kuin kuvan 17 Java-kielinen esimerkki. [4, s. 94.]

```
void testMethod()
{
    synchronized(this) {
        // Synkronoitu ohjelmalohko
    }
}
```

Kuva 17. Synkronoitu ohjelmalohko Java-metodin sisällä

```
JNIEXPORT void JNICALL
Java_TestClass_testMethod(JNIEnv* pEnv, jobject thisRef)
{
    pEnv->MonitorEnter(thisRef);
    // Synkronoitu ohjelmalohko
    pEnv->MonitorExit(thisRef);
}
```

Kuva 18. Synkronoitu ohjelmalohko natiivin ohjelman puolella

5 Natiivien kirjastojen käyttäminen Java-ohjelmasta

5.1 Natiivien kirjastojen linkitys

Käytettäessä natiiveja toteutuksia Java-luokan metodeille on virtuaalikoneen tiedettävä, miten linkittää Java-luokan metodit juuri oikeaan natiiviin funktioon. Virtuaalikone pyrkii linkittämään jokaisen natiivin toteutuksen silloin kun niitä ensimmäisen kerran kutsutaan. Virtuaalikone suorittaa seuraavat toimenpiteet linkityksen yhteydessä. [4, s. 151.]

- luokkien lataajan (Class loader) määritteleminen luokalle, joka sisältää kutsuttavan natiivin metodin
- metodin toteutuksen etsiminen kyseisen luokkien lataajan kanssa assosioiduista natiiveista kirjastoista
- sisäisten tietorakenteiden asettaminen siten, että seuraavat kyseisen metodin kutsut ohjautuvat suoraan natiiviin toteutukseen.

Natiivin toteutuksen hakeminen kirjastosta perustuu JNI-spesifikaatiossa määriteltyihin nimeämissääntöihin. Virtuaalikone muodostaa Java-luokan ja sen metodin nimen perusteella määriteltyjen sääntöjen mukaisen funktionimen, jota se yrittää paikantaa natiivista kirjastosta. Funktionimen muodostaminen tapahtuu seuraavien sääntöjen mukaisesti. [4, s. 152.]

- Etuliite ”Java_”
- Luokan nimi
- Alaviiva (”_”) erotinmerkki
- Metodin nimi

Ylikuormitettujen natiivien metodien kohdalla lisätään vielä kaksi alaviivaa (”__”) joiden jälkeen kuvaus argumenteista.

Yksinkertaisimmillaan määriteltyjen nimeämissääntöjen kuvan 19 mukaisesta metodista, jolla ei ole parametreja ja joka ei palauta mitään, syntyy kuvan 20 mukainen C-kielinen funktioprototyyppi. Huomion arvoista esimerkissä on se, että vaikka Java-metodi ei välitä mitään parametreja, niin virtuaalikone välittää natiivin kirjaston funktiolle aina kaksi parametria. Ensimmäisenä parametrina välittyy osoitin, jonka avulla voidaan kutsua JNI-rajapinnan tarjoamia palveluita natiivista kirjastosta käsin. Toisena parametrina välitetään referenssi luokan instanssiin, joka kutsun on suorittanut. Välitettävä referenssi vastaa siis Java-kielen this-avainsanaa, joka viittaa kyseiseen luokan instanssiin.

```
class TestClass
{
    void native dummyMethod();
}
```

Kuva 19. Natiivin metodin sisältävä Java-luokka

```
JNIEXPORT void JNICALL Java_TestClass_dummyMethod(
                            JNIEnv* pEnv,
                            jobject thisRef );
```

Kuva 20. JNI-spesifikaation mukainen natiivi funktio-prototyyppi.

Nimeämissääntöihin perustuvan automaattisen linkityksen sijasta ohjelmoijalla on myös mahdollisuus määritellä, kuinka Java-metodit ja natiivit funktiot tulee linkittää. Linkitys voidaan suorittaa JNI-rajapinnan RegisterNatives-komennon avulla, jolle kuvan 21 mukaisesti annetaan parametriksi tietorakenne, joka sisältää tiedot Java-metodeista sekä funktio-osoittimet niiden natiiveihin toteutuksiin. Kuten kuvasta 22 nähdään, tässä

tapauksessa voidaan funktion nimi määritellä itse eikä myöskään JNIEXPORT makroa tarvitse funktion prototyyppiin määritellä. [4, s. 101–102.]

```
JNINativeMethod nm;
nm.name = “dummyMethod”;
nm.signature = “()V”;
nm.fnPtr = nativeFunction;
pEnv->RegisterNatives( cls, &nm, 1 );
```

Kuva 21. Natiivien funktioiden linkitys RegisterNatives-komennon avulla

```
void JNICALL nativeFunction( JNIEnv* pEnv,
                             jobject thisRef );
```

Kuva 22. RegisterNatives-komennon avulla linkitetyn funktion prototyyppi

RegisterNatives-funktion käyttö on hyödyksi esimerkiksi seuraavissa tapauksissa:

- Toisinaan on mukavampaa ja tehokkaampaa linkittää suuri määrä natiiveja metodeita esimerkiksi kirjastoa ladattaessa verraten siihen, että virtuaalikone linkittää ne vasta niitä käytettäessä.
- RegisterNatives-funktiota voidaan kutsua useita kertoja, joka mahdollistaa linkitysten ajonaikaisen muuttamisen.
- Linkitys RegisterNatives-funktion avulla on myös hyödyllinen ohjelmoitaessa natiivia ohjelmaa, johon sisällytetään Java-virtuaalikone Invocation-API:n avulla, mikäli Java-ohjelma sisältää natiiveja metodeita, joiden toteutus on sisällytetty natiiviin ohjelmaan ulkoisen kirjaston sijasta.

5.2 Vaikutukset Java-ohjelmassa

JNI-rajapinnan käyttö näkyy hyvin vähän itse Java-ohjelman puolella, koska käytännössä virtuaalikone hoitaa natiivien metodien kutsumisen automaattisesti. Java-ohjelmassa natiivit metodit on ainoastaan merkitty native-avainsanalla ja jätetty toteutus kokonaan tekemättä kuvan 23 esimerkin mukaisesti. Tämä ohjaa virtuaalikoneen hakemaan toteutusta linkitetyistä kirjastoista.

```
public class TestClass
{
    static {
        System.LoadLibrary(“Example”);
    }

    public native void testMethod();
}
```

Kuva 23. Kirjaston määritteleminen Java-luokkaan

Huomion arvoista edellä kuvatussa esimerkissä on kirjaston määrittelemiseen käytetty nimi. Koska eri käyttöjärjestelmissä on jaetut kirjastot nimetty hieman eri tavoin, on Javassa jätetty kirjaston lopullisen tiedostonimen määritteleminen virtuaalikoneen vastuulle. Esimerkiksi Windows-maailmassa jaetut kirjastot on tyypillisesti nimetty Dll-päätteellä, kun taas Unix- ja Linux-maailmassa käytetään tyypillisesti So-päätettä. Kuten taulukosta 9 nähdään, käytettäessä kuvan 23 esimerkin mukaista parametria LoadLibrary-kutsussa on ladattavan kirjaston nimi erilainen eri käyttöjärjestelmissä.

Taulukko 9. Kirjaston tiedostonimet eri käyttöjärjestelmissä [3, s. 16]

Käyttöjärjestelmä	LoadLibrary-parametri	Tiedostonimi
UNIX	Example	libExample.so
Windows	Example	Example.dll

6 Esimerkkiohjelman toteutus

6.1 Tavoitteet

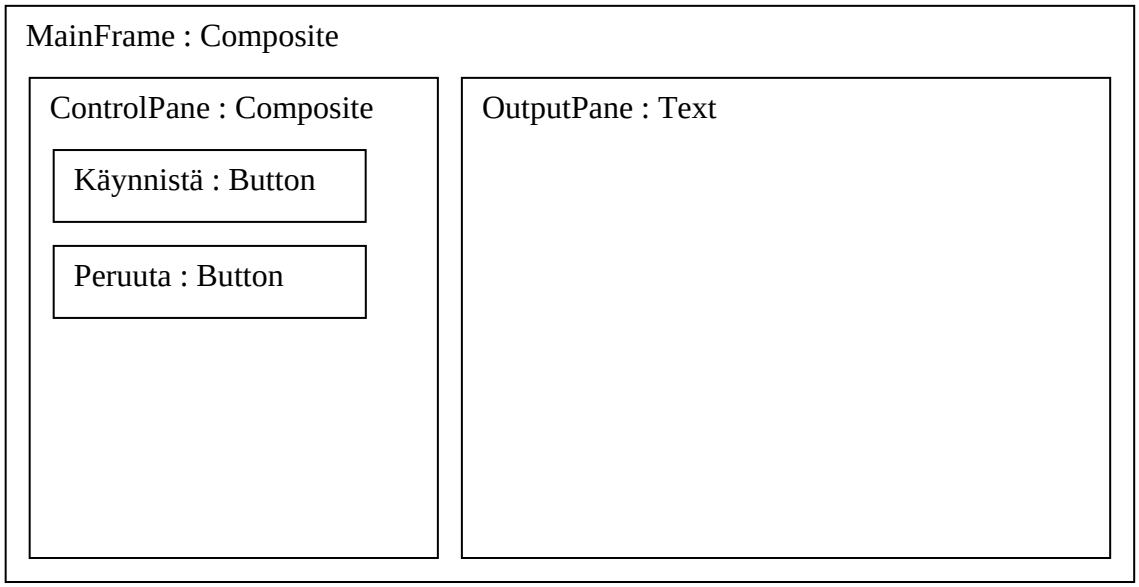
Jotta JNI-rajapinnasta ja sen käytöstä olisi saatu mahdollisimman hyvä kuva, siitä päätettiin tehdä myös esimerkkisovellus, joka sisältää käytännön toteutukset tietyistä ominaisuuksista. Tarkoitus ei ollut käydä läpi jokaista JNI-rajapinnan tarjoamaa toiminnallisuutta vaan lähteä liikkeelle peruskäyttötapauksista ja katsoa, mitä ongelmia toteutuksessa mahdollisesti ilmenee.

Ohjelman toteutuksessa lähdettiin ajatuksesta, että on olemassa natiivi kirjasto, joka sisältää jonkin vanhan ohjelmiston toiminnallisuuksia ja tämän päälle on rakennettu uusi käyttöliittymä Javalla käyttäen Eclipse-kehitysympäristöä. Osa kirjaston toiminnallisuuksista on luonteeltaan sellaisia, että niiden ajaminen kestää pidemmän aikaa ja niitä tulisi pystyä ajamaan asynkronisesti. Asynkronisista toiminnallisuuksista tulisi pystyä saamaan tietoa niiden etenemisestä ja tarvittaessa pystyä keskeyttämään niiden suoritus. Näiden lisäksi kirjasto sisältäisi myös joitain yksinkertaisempia toimintoja, joita suoritettaisiin synkronisesti.

6.2 Käyttöliittymä

Käyttöliittymän tekemiseen käytettiin Eclipse-kehitysympäristöä ja sen tarjoamia SWT-käyttöliittymäkomponentteja. Koska ohjelman tarkoituksena oli vain tutkia JNI-rajapinnan käyttöä, päätettiin käyttöliittymä pitää mahdollisimman yksinkertaisena. Käyttöliittymään päätettiin laittaa napit, joista eri toimintoja voidaan käynnistää ja peruuttaa. Näiden lisäksi käyttöliittymään lisättiin tekstikenttä, johon voidaan tulostaa tietoja toimintojen etenemisestä.

Käytännössä käyttöliittymä jaettiin kuvan 24 mukaisesti kahteen eri paneeliin, joista toinen sisälsi toimintoihin liittyvät kontrollit ja toinen taas lista-tyyppisen tekstikentän, johon voitiin tulostaa tekstimuotoista informaatiota toimintojen etenemisestä.



Kuva 24. Esimerkkiohjelman käyttöliittymä

6.3 Toiminnallisuus

Selkeyttääkseni ohjelman rakennetta ja rajapintaa Java-ohjelman ja natiivin kirjaston välillä päätin sijoittaa kaiken natiivin toiminnallisuuden NativeInterface-luokan alaisuuteen staattisiksi metodeiksi kuvan 25 mukaisesti. Tällöin Java-ohjelman puolelta on helppoa tunnistaa natiivit metodit ja nähdä rajapinta natiivin kirjaston ja Java-ohjelman välillä.

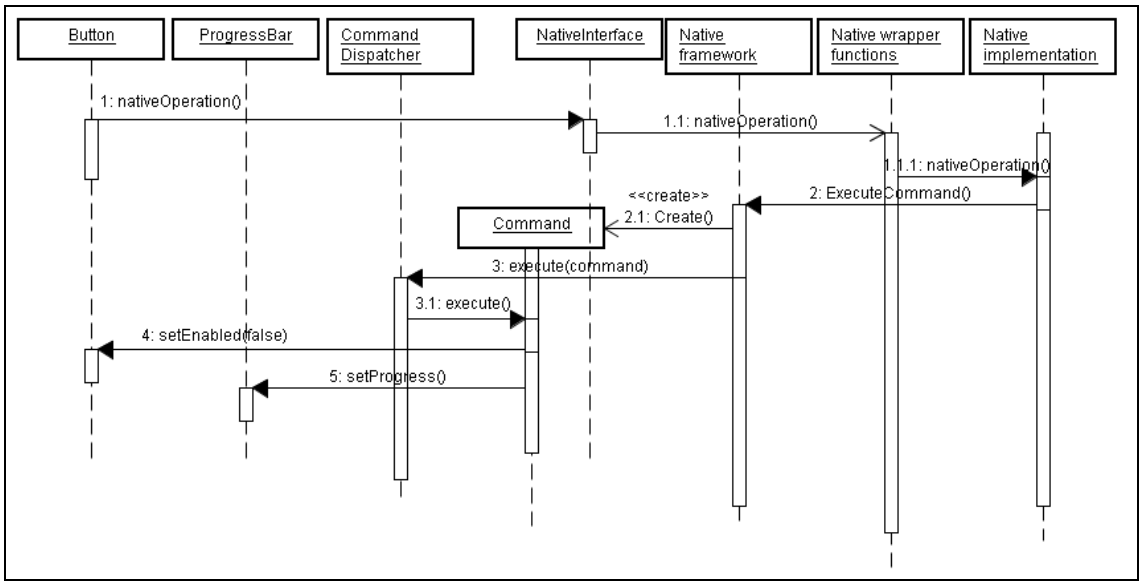
```
public abstract class NativeInterface
{
    public static native void
        simpleNativeOperation() throws Exception;

    public static native
        void findPrimes(int start,int end)
        throws Exception;

    public static native
        void sortRandomNumbers(int numbercount)
        throws Exception;

    public static native
        void trafficGenerator(int stringcount)
        throws Exception;

    public static native
        int[] getIntArray(int size) throws Exception;
}
```

Kuva 25. Natiivin toiminnallisuuden määritykset NativeInterface-luokassa

Natiivista toiminnallisuudesta osa suoritettiin synkronisesti, mutta osa taas oli asynkronisesti suoritettavaa. Käyttöliittymiä ohjelmoitaessa yleensä kaikkia käyttöliittymäkontrolleja tulee käsitellä samasta säikeestä, jossa kontrollit on luotu. Tyypillisesti on siis yksi säie, jossa käyttöliittymää ajetaan ja toiminnallisuuksia ajetaan taustasäikeissä. Taustasäikeistä kuitenkin halutaan yleensä näyttää joitain tietoja käyttöliittymässä ja tällöin tarvittava informaatio tulee toimittaa jollain tavalla käyttöliittymäsäikeelle, joka taas huolehtii käyttöliittymäkontrollien päivittämisestä.

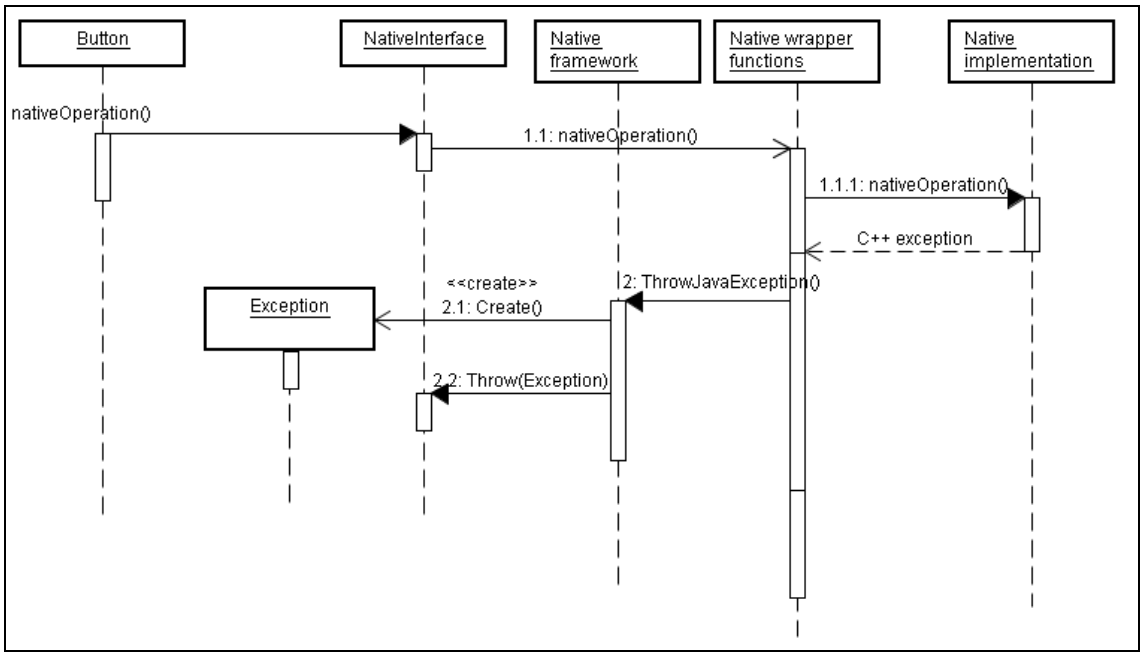
Käytännössä tiedon siirtäminen käyttöliittymäsäikeelle voidaan toteuttaa usein eri tavoin, ja eri ohjelmointikielet ja eri käyttöliittymäteknologiat sisältävät hieman oman tapansa toteuttaa asian.

Omassa ohjelmassani päätin kokeilla command pattern -suunnittelumallin mukaista toteutusta, jossa tietty toiminnallisuus paketoidaan omaan komentoluokkaan ja annetaan toiselle luokalle suoritettavaksi kuvan 26 mukaisesti. Käyttöliittymän ollessa puhtaasti Javalla toteutettu oli myös järkevää toteuttaa kaikki komentoluokat Javalla. Tällä tavoin saatiin myös kaikki käyttöliittymään ja sen päivittämiseen liittyvä koodi Javan puolelle Java-kääntäjän tarkistettavaksi. Komentoja suorittava luokka voitiin tässä tapauksessa myös toteuttaa siten, että sille voitiin lähettää komentoja muista säikeistä ja se osasi siirtää suorituksen käyttöliittymäsäikeelle. Tässä tapauksessa myös komentojen lähettämiseen tarvittava toiminnallisuus voidaan natiivin ohjelman puolella paketoida melko siististi, koska kaikkia komentoja käsitellään samalla tavalla. Tällä tavoin voidaan ehkäistä jossain määrin myös virheiden mahdollisuuksia, koska samoja funktioita voidaan käyttää komentojen lähettämiseen.



Kuva 26. Sekvenssikaavio asynkronisten metodien toteutuksesta

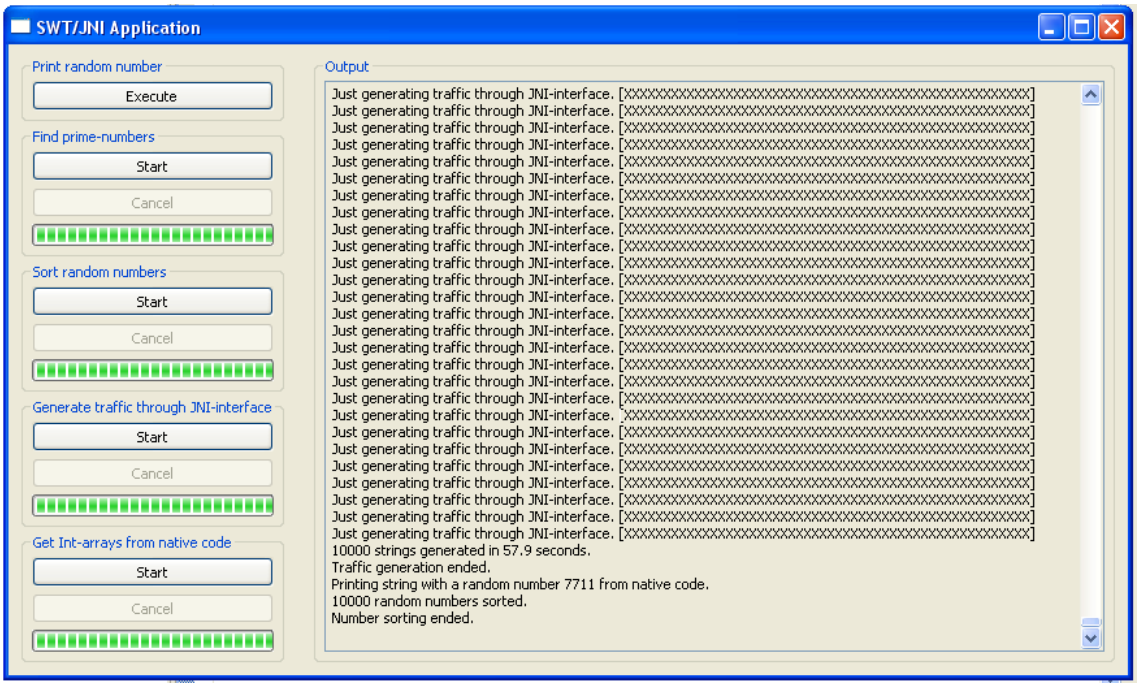
Poikkeuksien käsittelyä varten natiivin ohjelman puolelle lisättiin toiminnallisuus, joka poimii kaikki natiivit poikkeukset ja osaa muuntaa ne Java-poikkeuksiksi. Tämä toteutettiin tekemällä JNI-rajapinnan yli kutsuttavaan välifunktioon toteutus, joka osaa kerätä C++ -poikkeukset, paketoida ne uudelleen Java-poikkeusluokan sisälle ja välittää poikkeuksen JNI-rajapinnan yli kuvan 27 mukaisesti. Tämän ansiosta natiivin ohjelman puolella voidaan virhetilanteita käsitellä aivan normaalisti natiivien poikkeuksien avulla ja ne välittyvät automaattisesti Java-ohjelman puolelle.



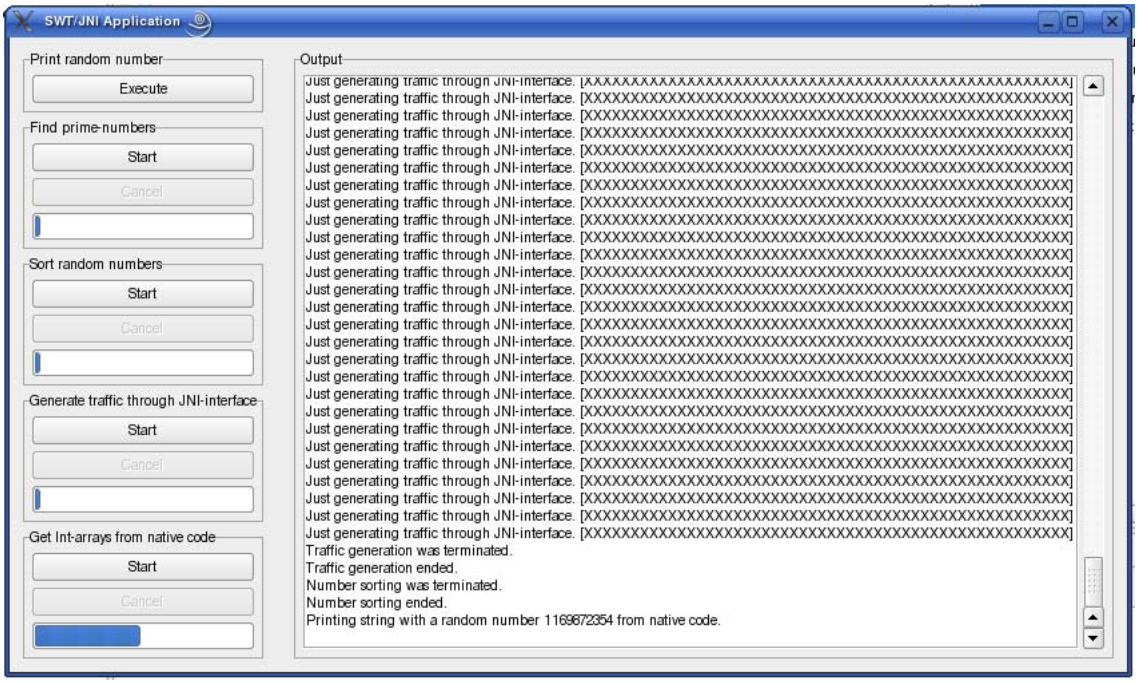
Kuva 27. Sekvenssikaavio poikkeusten käsittelyn toteutuksesta

6.4 Valmis ohjelma

Koska työn tarkoituksena oli tuottaa ympäristöstä toiseen siirrettävä ohjelmisto, sen testaus suoritettiin sekä Windows- että Linux–ympäristössä. Käyttöjärjestelmäversioina käytettiin Windows XP -käyttöjärjestelmää sekä Fedora Linux -jakeluversiota. Kuten kuvista 28 ja 29 voidaan nähdä, on ohjelman käyttöliittymä hieman eri näköinen käyttöjärjestelmästä riippuen, koska SWT-käyttöliittymäkirjasto käyttää käyttöjärjestelmän tarjoamia natiiveja komponentteja. Molemmat versiot ovat kuitenkin toiminnallisuudeltaan täysin identtisiä.



Kuva 28. Esimerkkiohjelma Windows-ympäristössä ajettuna



Kuva 29. Esimerkkiohjelma Linux-ympäristössä ajettuna

7 Käytännön kokemuksia

7.1 Yleisiä kokemuksia

Insinöörityötä aloitettaessa ja materiaalia etsittäessä kävi melko nopeasti selväksi, että JNI-rajapintaan liittyen löytyy materiaalia erittäin vähän. Itse rajapintaan löytyy spesifikaatio, jossa käytännössä määritellään rajapinnan toiminta, mutta sen lisäksi kirjallisuutta ei juurikaan tuntunut olevan. Rajapinta ei kuitenkaan lopulta ole kovin monimutkainen, joten pelkän spesifikaation pohjalta sen käyttäminen luonnistui suhteellisen kivuttomasti.

Vaikka esimerkkiohjelman tekemisessä ei mitään kovin vakavia ongelmia tullut vastaan, niin ohjelmaa toteuttaessa koetin jatkuvasti miettiä asiaa isomman ohjelmistokehitysprojektin näkökulmasta. Tehty esimerkkiohjelma oli pieni, erittäin yksinkertainen ja sitä oli tekemässä vain yksi ihminen. Miten tilanne muuttuisi, jos projektia olisi tekemässä vaikkapa kymmenen ihmistä ja kenties ohjelman eri komponentteja tehtäisiin kokonaan eri osastoilla? Esille tulleet ongelmat eivät ehkä suoraan liity JNI-rajapintaan, mutta ne selkeästi nostavat esiin joitain huomioitavia asioita JNI-rajapintaa hyödyntävän ohjelmiston arkkitehtuuriin, suunnitteluun ja testaukseen liittyen. Mitään yksiselitteistä ratkaisua näihin asioihin ei ole, vaan ratkaisut ovat hyvin tapauskohtaisia.

Ohjelmiston suunnittelun ja arkkitehtuurin näkökulmasta asiaa hieman hämärtää se, että JNI-rajapintaa hyödyntävässä ohjelmassa ei varsinaisesti synny selkeää rajapintaa Java-ohjelman ja natiivin ohjelman välille, koska natiivit funktiot ovat käytännössä Java-luokan metodien natiiveja toteutuksia. Mikäli asiaa verrataan esimerkiksi Windows-ympäristöissä yleisesti käytössä olevaan Microsoftin COM-teknologiaan, on siinä rajapinta tarkasti määritelty ja sekä komponentin että komponenttia käyttävän ohjelman tulee noudattaa määriteltyä rajapintaa. Koska molemmat osapuolet tietävät rajapinnan määrityksen, lähdekoodin oikeellisuus pystytään ainakin jossain määrin tarkistamaan jo käännösvaiheessa. JNI-rajapinnan tapauksessa taas käännösvaiheen tarkistukset jäävät

täysin tekemättä, mikä voi aiheuttaa lukuisia ongelmia laajempia sovelluksia toteutettaessa.

7.2 Huomioitavia asioita suunnittelussa

Hyviin ohjelmistokehityskäytäntöihin tietysti kuuluu, että ohjelmaa kehitettäessä siitä pyritään tekemään mahdollisimman helppo myös ylläpidettävyyden kannalta. Ohjelmistoa ylläpitävät ihmiset saattavat olla täysin eri henkilöitä, jotka alunperin ovat olleet sitä kehittämässä, joten koskaan ei voida olettaa, että ohjelmistoa jollain tavalla muuttava henkilö omaa tietyt pohjatiedot. Hyviä käytäntöjä ylläpidettävyyden kannalta ovat tunnettujen rakenteiden käytön, ohjelman yleisen loogisuuden, loogisten nimeämiskäytäntöjen ja selkeän arkkitehtuurin lisäksi se, että asioita pyritään määrittelemään vain yhteen kertaan, jolloin jonkin asian muuttaminen ei vaadi sitä, että sama muutos pitää tehdä useampaan paikkaan. Virheiden minimoimiseksi on myös tärkeää, että mahdollisimman suuri osa virheistä pyritään havaitsemaan esimerkiksi jo käännösprosessin aikana joko kääntäjien toimesta tai käännösprosessiin liitettyjen yksikkötestien avulla.

Yleensä kääntäjät pyrkivät tarkistamaan kirjoitetun ohjelman mahdollisimman kattavasti mahdollisten virheiden varalta. JNI-rajapintaa käyttävän ohjelman kannalta pelkästään kääntäjien tekemät tarkistukset eivät usein kuitenkaan riitä, koska selkeitä tietotyyppivirheitä voi jäädä kääntäjiltä täysin havaitsematta. Yksinkertaistetun esimerkin avulla kuvattuna: oletetaan, että meillä on kuvan 30 mukainen Java-luokka, jolla on yksi jäsenmuuttuja ja yksi natiivi metodi. Oletetaan myös, että natiivista metodista käsin tarvitsee päivittää jotain jäsenmuuttujaa kuvan 31 esimerkin mukaisesti. Näinkin yksinkertaisessa tapauksessa jo nähdään, että meillä on samoja tietotyyppejä määriteltynä sekä Javan puolella, että myös natiivin ohjelman puolella. Näin ollen mahdolliset tietotyyppien muutokset tulisi myös huomioida sekä Java-ohjelman että natiivin ohjelman puolella.

```
class TestClass
{
    private int m_nCount;

    public native float test(int param);
}
```

Kuva 30. Java-luokka

```
jfloat test(JNIEnv* pEnv,jobject thisRef, jint param)
{
    …
    jfieldID fid =
        pEnv->GetFieldID(cls,”m_nCount”,”I”);
    …
}
```

Kuva 31. Yksinkertaistettu natiivi toteutus

Esimerkin mukaisessa ohjelmassa mahdollinen ristiriita tietotyyppien välillä ei kuitenkaan automaattisesti aiheuta käännösvirhettä. Pienemmissä ohjelmissa nämä virheet on vielä suhteellisen helppo havaita, mutta hieman isommassa projektissa, joka sisältää runsaasti natiivia koodia ja jossa ei välttämättä kovinkaan tarkkaan tiedetä mihin Java-luokkiin ja muuttujiin natiivista ohjelmakoodista käsin viitataan, tämä voi muodostua ongelmaksi. Asian tekee huomattavasti monimutkaisemmaksi ja samalla virheherkemmäksi vielä se, jos natiivin ohjelman puolelle välitetään esimerkiksi omia Java-luokkia parametreina.

8 Yhteenveto

Työn jälkeen JNI-rajapinnasta jäi hieman kaksijakoinen vaikutelma. Natiivin koodin käyttö on rajapinnan avulla erittäin nopeaa, koska natiivi kirjasto ladataan samaan prosessiin, eikä tietoa tarvitse tällöin välittää eri prosessien välillä. Rajapinnan tarjoamat palvelut ovat myös melko selkeitä ja niitä on sen verran rajallinen määrä, että ne on melko helppo oppia. Toisaalta kyseessä on melko matalan tason rajapinta natiivin ohjelman puolelta katsoen ja sen kanssa on erittäin helppo tehdä virheitä.

Käytettäessä JNI-rajapinnan palveluita suoraan natiivista koodista käsin on jatkuvasti tarkkailtava, onko Java-virtuaalikoneen puolella kenties tapahtunut poikkeus jonkin kutsun aikana ja käsiteltävä virhetilanne asiaan kuuluvalla tavalla. Yksinkertainenkin toimenpide saattaa pitää sisällään kymmeniä rajapinnan yli tehtyjä kutsuja, ja kaikkien näiden välissä tulisi käytännössä tehdä samat tarkistukset. Virheiden tarkistus ei sinällään ole ongelma, mutta tekee rajapinnan suoraan käyttämisestä melko hankalaa ja virhealtista. Lisäksi tästä syntyy helposti paljon toisteista ohjelmakoodia, koska useassa tapauksessa tarvittava tarkistus on hyvin saman tyyppinen.

Esimerkiksi C++ -kielellä ohjelmoitaessa tulisikin rajapinnan tarjoama toiminnallisuus paketoida helppokäyttöisempiin luokkiin, jotka sisältävät oikeaoppisen virheiden käsittelyn ja resurssien vapautuksen. Tällöin voitaisiin varmistua, että tarvittavat virheiden tarkistukset suoritetaan joka kerta ja samalla myös tehtäisiin rajapinnan käyttö hyvin helpoksi ohjelmoijille, jotka natiivia toiminnallisuutta kirjoittavat.

Ennen JNI-rajapinnan laajamittaista käyttöönottoa tulisikin ehkä vielä etsiä jotain ratkaisua tähän mainittuun ongelmaan valmiista kirjastoista tai työkaluista tai tuottamalla itse tarvittava kirjasto tai ohjelmistorunko rajapinnan yksinkertaiseen käyttöön.

Lähteet

1 About the Eclipse Foundation. (HTML-dokumentti.) <http://www.eclipse.org/org/>. The Eclipse Foundation 2008. luettu 4.1.2008.

2 Kilponen, Ari. MFC/Stingray-sovelluksen uudelleentoteutus Eclipse-alustalla. Insinöörityö. EVTEK-ammattikorkeakoulu, 2005.

3 Gordon, Rob. Essential JNI. New Jersey: Prentice Hall 1998.

4 Liang, Sheng. The Java Native Interface Programmers Guide and Specification. Massachusetts: Addison Wesley Longman 1999.

5 Synchronized Methods. (HTML-dokumentti.) <http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html>. Sun Microsystems, Inc 2007. luettu 26.1.2008.

6 Intrinsic Locks and Synchronization. (HTML-dokumentti.) <http://java.sun.com/docs/books/tutorial/essential/concurrency/locksync.html>. Sun Microsystems, Inc 2007. luettu 26.1.2008.