VTT TIEDOTTEITA – MEDDELANDEN – RESEARCH NOTES 1985

Javan luokkakirjasto testitapauseditorin toteutuksessa

Tapani Rauhala
VTT Elektroniikka

VALTION TEKNILLINEN TUTKIMUSKESKUS
ESPOO 1999

ISBN 951–38–5489–2 (nid.)
ISSN 1235–0605 (nid.)

ISBN 951–38–5490–6 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Toimitus Maini Manninen

Libella Painopalvelu Oy, Espoo 1999

Rauhala, Tapani. Javan luokkakirjasto testitapauseditorin toteutuksessa [Implementation of a test case editor with Java's new class library]. Espoo 1999. Valtion teknillinen tutkimuskeskus, VTT Tiedotteita – Meddelanden – Research Notes 1985. 68 s.

Avainsanat JAVA, JFC class library, portable applications, performance

Tiivistelmä

Tässä työssä tutkittiin Javan uuden luokkakirjaston, JFC:n, soveltuvuutta itsenäisten sovellusten toteutukseen. Esimerkkisovelluksena käytettiin MOSIM-testausympäristöön kehitettyä testitapauseditoria. JFC on laiteympäristöstä ja käyttöjärjestelmästä riippumaton luokkakirjasto, jolla toteutetut sovellukset toimivat sellaisenaan kaikissa Javaa tukevissa ympäristöissä. Työn tavoitteena oli selvittää, miten hyvin JFC:n avulla voidaan toteuttaa siirrettäviä sovelluksia. Erityisen huomion kohteena oli suorituskyky. Tarkoituksena oli tutkia, millaisten sovellusten toteutukseen JFC:n suorituskyky riittää tällä hetkellä.

Ohjelmistojen koko on kasvanut räjähdysmäisesti viime vuosina. Samaan aikaan on markkinoille tullut lukuisia uusia prosessorityyppejä ja käyttöjärjestelmiä. Tämä on johtanut tilanteeseen, jossa monessa ympäristössä toimivien ohjelmistojen kehityksestä on tullut hyvin vaikeaa.

Java ja sen luokkakirjastot tarjoavat ratkaisun ohjelmistojen siirrettävyyteen. Javalla toteutettuja sovelluksia voidaan käyttää ilman muutoksia lähes kaikissa käyttöjärjestelmissä. Ohjelmiston tulevia käyttöympäristöjä ei ole tarpeen kiinnittää suunnittelu- ja kehitysvaiheessa.

Java on ajon aikana tulkattava kieli. Tämä aiheuttaa ongelmia silloin, kun sovellukselta vaaditaan erityistä suorituskykyä. Saadun kokemuksen perusteella Java ja JFC-kirjasto eivät vielä sovellu suurta nopeutta vaativien ohjelmistojen toteutukseen. Ongelmia voi tulla myös silloin, kun ohjelmiston koko kasvaa suureksi. Tulevaisuudessa tilanne voi muuttua, kun tietokoneista tulee nykyistä nopeampia. Uudet Java-kääntäjät ja virtuaalikoneet voivat myös ratkaista Javan suorituskykyongelmat.

Rauhala, Tapani. Javan luokkakirjasto testitapauseditorin toteutuksessa [Implementation of a test case editor with Java's new class library]. Espoo 1999. Technical Research Centre of Finland, VTT Tiedotteita – Meddelanden – Research Notes 1985. 68 p.

Keywords JAVA, JFC class library, portable applications, performance

Abstract

The aim of this work was to examine Java's JFC class library, and evaluate its suitability for application development. A test case editor, which was developed for the MOSIM testing environment, was used as an example. JFC is device and operation system independent class library, and applications developed by using JFC are fully portable across all environments having Java support. The purpose of this work was to study how suitable JFC is for portable applications. Special attention was paid to the performance of JFC and Java.

The size of software applications has grown considerably during the past years. At the same time, new processors and operating systems have become available. Because of this it is very difficult to develop software that would be independent from the target environment.

Java and its class libraries offer a solution to the portability of software. It is possible to run applications developed in Java in almost any operating system without changes. It is not necessary to take into account the target environments in software design and implementation.

Java code is interpreted during its execution. This can be a problem, if performance is critical for the application. According to the experience gained in this work, the performance of Java and JFC is not sufficient yet for time-critical applications. The size of software may also cause problems. In the future, however, computers will probably be faster and new Java compilers and virtual machines will improve the performance of Java.

Alkusanat

Tämä työ tehtiin VTT Elektroniikassa Oulussa osana VERSO-tutkimusprojektia. Työssä tutkittiin Javan ja JFC-luokkakirjaston soveltuvuutta itsenäisten, siirrettävien sovellusten toteutukseen. Esimerkkisovelluksena käytettiin MOSIM-testausympäristöön toteutettua testitapauseditoria.

Haluan esittää kiitokseni työn valvojana toimineelle professori Jaakko Sauvolalle sekä toisena tarkastajana toimineelle tutkimusprofessori Veikko Seppäselle.

Kiitokset myös fil. maist. Juha Lehtikankaalle, tekn. lis. Harri Perungalle ja tekn. lis. Hannu Hongalle työn alkuvaiheessa saadusta avusta. Lisäksi haluan kiittää kanssani samassa huoneessa työskennelleitä Jaria, Jukkaa ja Mikaa, joilta sain monia arvokkaita neuvoja ja kommentteja.

Viimeiset kiitokset menevät vaimolleni Merjalle sekä kaikille tutuille ja sukulaisille, jotka tukivat minua kirjoittamisen aikana.

Oulussa 12.4.1999

Tapani Rauhala

Sisällysluettelo

Nimien, lyhenteiden ja merkkien selitykset

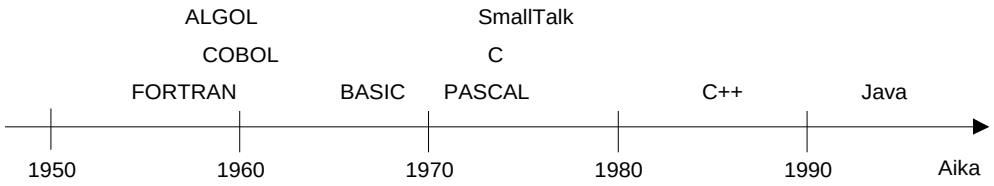
API	Application Programming Interface. Sovellusrajapinta.
AWT	Abstract Window Toolkit.
DLL	Dynamically Linked Library. Dynaamisesti linkitettävä kirjasto.
GUI	Graphical User Interface. Graafinen käyttöliittymä.
HotSpot	Sun Microsystemsin kehittämä uuden sukupolven optimoiva Java-kääntäjä.
HTML	HyperText Markup Language.
HTTP	HyperText Transfer Protocol.
Java	Sun Microsystemsin kehittämä oliopohjainen ohjelmointikieli.
JDK	Java Developer Kit.
JFC	Java Foundation Classes.
ILS	Instruction Level Simulator.
JINI	Sun MicroSystemsin kehittämä tekniikka, joka mahdollistaa sulautettujen laitteiden liittämisen verkkoon.
JIT	Just In Time. Javan tavukoodia juuri ennen ajoa optimoiva kääntäjä.
JRE	Java Runtime Environment.
JVM	Java Virtual Machine. Javan tavukoodia suorittava virtuaalikone.
MDI	Multiple Document Interface.
MFC	Microsoft Foundation Classes. Microsoftin kehittämä luokkakirjasto.
MOSIM	VTT Elektroniikan kehittämä testaustyökalu tietoliikenneohjelmistojen testaukseen.
MVC	Model-View-Controller, arkkitehtuuri, jossa sovelluksen tietoa käsittelevät osat jaetaan malleihin, näkymiin ja ohjausosiin.
OWL	Object Windows Library. Borlandin C++-luokkakirjasto.
PCO	Point of Control and Observation. Tarkkailupiste.

RMI	Remote Method Invocation.
SART	Structured Analysis and Design for Real-Time Systems.
SUT	Software Under Test. Testattava sovellus.
Swing	JFC:n sisältämä graafinen kirjasto.
TCE	Test Case Editor.
TCU	Test Controller Unit. MOSIMin testausyksikkö.
WWW	World Wide Web.

1. Johdanto

Ensimmäiset ohjelmointikielet kehitettiin 1950-luvulla. Niiden avulla oli mahdollista ohjata varhaisimpia tietokoneita. Nämä kielet olivat täysin riippuvaisia käytetystä tietokoneesta. FORTRAN oli ensimmäinen ns. korkean tason ohjelmointikieli (high level programming language). FORTRAN-kääntäjällä voitiin tuottaa matemaattisesta esitysmuodosta ajettavaa konekielistä koodia. Hieman myöhemmin kehitettiin ALGOL-kieli, jonka käsitteet ovat vielä nykyäänkin käytössä.

Tietokonetekniikan kehittyessä syntyi tarve uusille ohjelmointikielille. 1960- ja 1970-luvuilla kehitettiinkin useita uusia kieliä. Osa näistä kielistä on käytössä vielä tänäkin päivänä. Tärkeimpinä voidaan mainita Pascal, C-kieli sekä SmallTalk. Kaikki nykyaikaiset oliopohjaiset ohjelmointikielet ovat saaneet vaikutteita SmallTalkista. Kuvassa 1 esitetään tärkeimmät vaiheet ohjelmointikielten kehityksessä.



Kuva 1. Ohjelmointikielten historia.

Vuonna 1991 Sun Microsystems alkoi kehittää uusia järjestelmiä kulutuselektroniikkaan. Ohjelmistot oli tarkoitus kehittää C++:lla, mutta pian huomattiin ohjelmista tulevan liian raskaita ja vaikeasti siirrettäviä. Tämän takia päätettiin kehittää uusi ohjelmointikieli, jolle annettiin aluksi nimeksi Oak [1].

Kieli perustui C++:aan, mutta siitä oli karsittu pois ne C++:n piirteet, jotka oli käytännössä koettu vaikeiksi ja virheitä tuottaviksi. Poistettavien ominaisuuksien joukossa oli muun muassa C++:n osoittimet. Oak ei kuitenkaan menestynyt kovin hyvin ja päätyi melkein roskakoriin.

Oak nousi uudelleen esille vuonna 1993 World Wide Webin myötä [2]. Tämän jälkeen Oakia kehitettiin kohti nykyistä muotoaan ja vuonna 1995 kielen uudeksi nimeksi annettiin Java. World Wide Web on tämän jälkeen tuonut Javaa tunnetuksi, ja WWW-selaimissa toimivat Java-appletit ovat yleistyneet räjähdysmäisesti. Java-appletit ovat Javalla tehtyjä ohjelmia, joita voidaan suorittaa esimerkiksi WWW-selainten avulla.

WWW:n ansiosta Java mielletäänkin usein vain applettien tekoon soveltuvaksi ohjelmointikieleksi. Java soveltuu kuitenkin myös itsenäisten sovellusten tekoon. Javan käyttöä itsenäisten sovellusten toteutuskielenä ovat rajoittaneet muun muassa tulkattavan tavukoodin suorituksen hitaus ja kielen nopea kehitys. Javalla on kuitenkin monia

ominaisuuksia, jotka puoltavat sen käyttöä myös itsenäisten sovellusten ohjelmointiin, kunhan kielen pahimmat puutteet on korjattu.

Aikaisemmat oliopohjaiset ohjelmointikielet eivät ole sisältäneet standardoitua, kieleen sisältyvää luokkakirjastoa. Javaan kiinteästi kuuluvaa luokkakirjastoakaan ei ole vielä virallisesti standardoitu, mutta sitä on käytetty lähes kaikissa Javalla tehdyissä appleteissa ja sovelluksissa. Java Foundation Classes (JFC) on merkittävä laajennus tähän luokkakirjastoon. Tämän työn kirjoittamisen aikana JFC-luokkakirjastosta julkaistiin lopullinen versio yhdessä seuraavan Java-version (1.2) kanssa [3].

C++:aan on ollut jo pitkään saatavana ohjelmistotyökalujen mukana tulleita luokkakirjastoja. Tunnetuimpia ja käytetyimpiä ovat Microsoftin MFC- [4] ja Borlandin OWL-kirjastot [5]. MFC:stä on tullut suositumpi johtuen Microsoftin hallitsevasta asemasta ohjelmistomarkkinoilla. Näiden luokkakirjastojen suurin puute on ollut niiden sitoutuminen yhteen käyttöjärjestelmään. Sekä MFC- että OWL-kirjastot on suunniteltu toimimaan pelkästään Microsoftin Windowsissa. Muillekin käyttöjärjestelmille on saatavissa luokkakirjastoja, mutta niidenkin ongelmana on riippuvuus käyttöjärjestelmästä.

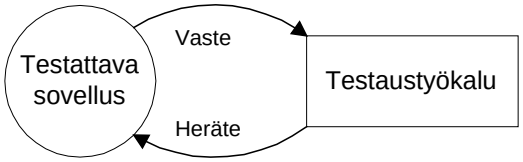
JFC poikkeaa hyvin paljon C++:n kirjastoista, kuten Java-kielikin C++:sta. Tärkein ero on JFC:n siirrettävyys eri käyttöjärjestelmä- ja laiteympäristöjen välillä. Sama kirjasto käy kaikkiin ympäristöihin, joihin on olemassa Java Virtual Machine (JVM). JVM:n tehtävänä on kääntää Javan tavukoodi prosessorin ymmärtämään muotoon ja ohjata ohjelman suoritusta. Javan tavukoodi voidaan siirtää suoraan ympäristöstä toiseen tarvitsematta kääntää tehtyä sovellusta uudelleen. Tulevaisuudessa Java ja siihen sisältyvät kirjastot standardoidaan. Tällöin ohjelmoijilla on käytössään todella tehokas ohjelmointikieli, jolla tehdyt sovellukset siirtyvät helposti ympäristöstä toiseen.

JFC sisältää graafisten komponenttien luontiin tarkoitetun osan, jota kutsutaan tällä hetkellä nimellä Swing. Se on vaihtoehto Javan nykyiselle graafiselle luokkakirjastolle AWT:lle. Swingin komponentit ovat kevyempiä ja monipuolisempia ja korvaavat AWT-komponentit monessa tapauksessa. Swingin beta-versioita on ollut mahdollista käyttää ohjelmistokehitykseen keväästä 1998 lähtien.

1.1 Toteutettava sovellus

Tässä työssä kehitettiin graafinen käyttöliittymä MOSIM-testaustyökalun [6] testitapausten luontiin ja hallintaan. Simulointiin perustuvissa sulautettujen järjestelmien testaamiseen kehitetyissä työkaluissa kommunikaatio testattavan sovelluksen ja testityökalun välillä perustuu viesteihin, kuva 2. Viestien rakenteista tulee helposti monimutkai-

sia, erityisesti tietoliikennejärjestelmissä. Viestit ovat yleensä heksadesimaalisessa muodossa, joten niiden lukeminen on vaikeaa.



Kuva 2. Testaustyökalun ja testattavan sovelluksen väliset viestit.

Testitapahtuma on testauksen yhteydessä käytettävä käsite, joka voi olla esimerkiksi yksi viesti. Testitapaus on useasta testitapahtumasta muodostuva kokonaisuus. MOSIMissa käytettävät testitapaukset on tähän asti tehty käsin, editoimalla ns. INREC-tiedostoja. Näitä INREC-tiedostoja on sitten voitu ajaa MOSIMin Test Controller Unit (TCU) -yksikön avulla. Toteutetun testitapauseditorin tarkoituksena on pienentää testitapausten luontiprosessiin kuluvaa aikaa.

MOSIM-testaustyökalusta on tällä hetkellä olemassa versiot VMS-, Unix- ja Windows NT-ympäristöihin. Tämä oli yksi peruste ohjelmointikielen valinnalle. Javalla toteutettuna työkalu saatiin toimimaan lähes sellaisenaan Windows NT:ssä ja Motifissa. Javan tavukoodin suorituksen hitaus esimerkiksi C++-kääntäjän tuottamaan konekieliseen koodiin verrattuna tiedettiin, mutta koska toteutettava sovellus oli suhteellisen yksinkertainen, nopeusvaatimukset eivät olleet suuria.

Ohjelmointikielenä käytettiin siis Javaa, jolla työkalusta saatiin helposti siirrettävä. Sovellus toteutettiin JFC-luokkakirjastoa käyttäen. Käyttöympäristössä täytyy olla asennettuna Java Virtual Machine (JVM). Version tulee olla 1.1.6 tai uudempi. JVM saadaan joko JDK:n tai JRE:n mukana, ja nykyään se on saatavilla kaikkiin yleisimpiin käyttöjärjestelmä- ja laiteympäristöihin.

1.2 Työn tavoite

Työssä toteutettiin testitapauseditori MOSIM-testausympäristöön. Toteutetun editorin tärkein vaatimus oli helpottaa ja nopeuttaa testiaineistojen luontia MOSIMissa. Koska Javan ja erityisesti JFC-luokkakirjaston käytöstä ei ole paljon kokemusta todellisissa projekteissa, työssä tutkittiin Javan soveltuvuutta itsenäisten sovellusten ohjelmointikieleksi nykyisellä versiolla ja uudella JFC-luokkakirjastolla. Esimerkkisovelluksena käytettiin testitapauseditoria. Yhtenä tavoitteena oli tutkia, miten hyvin tulkattavan ohjelmakoodin suoritusnopeus riittää sovelluksen sujuvaan käyttöön. Toisena tavoitteena oli kokeilla tehdyn sovelluksen toimintaa eri ympäristöissä, jolloin saataisiin käytännön kokemusta Javan luokkakirjaston luvatusta ympäristöriippumattomuudesta.

2. Graafisen käyttöliittymän suunnittelu

Tässä luvussa käsitellään graafisen käyttöliittymän suunnittelussa käytettäviä periaatteita, ja esitellään komponentit, joista graafinen käyttöliittymä muodostuu. Lopuksi tarkastellaan, mitä oliopohjaisuus merkitsee käyttöliittymissä.

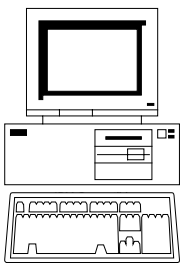
2.1 Kehitys merkkipohjaisista graafisiin käyttöliittymiin

Ensimmäisten tietokoneiden käyttöliittymät olivat hyvin alkeellisia. Käyttäjä ohjasi konetta mekaanisilla kytkimillä ja tietokone antoi vasteita vilkkuvien valojen avulla. Tietokoneen käyttäminen vaati perusteellista koulutusta, ja siihen pystyivät vain korkeasti koulutetut ammattilaiset, kuva 3.

Kuva 3. ENIAC-tietokone.

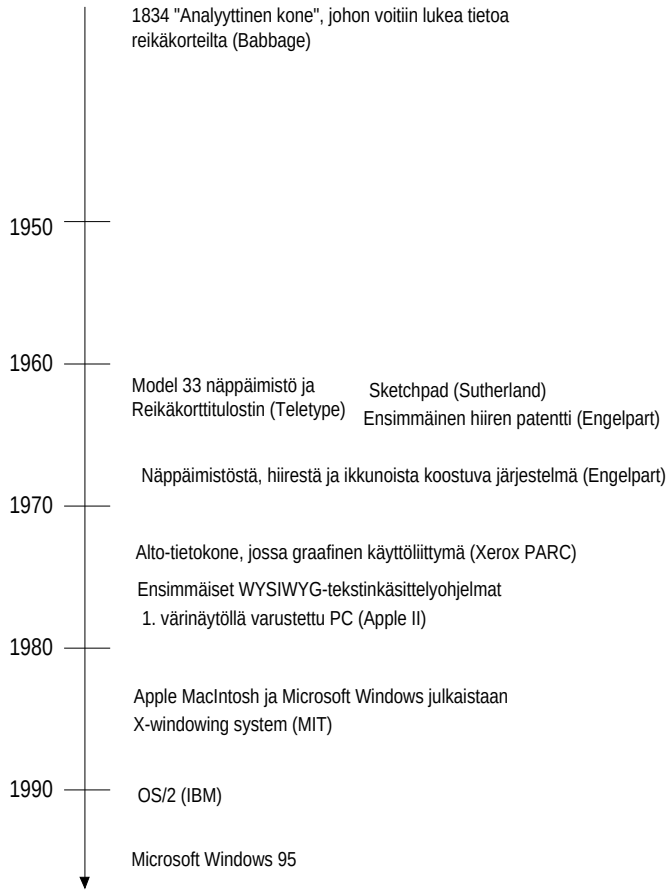
Seuraava vaihe käyttöliittymien kehityksessä saavutettiin, kun tietokoneita alettiin ohjata reikäkorttien avulla. Tietokoneen antamat vasteet saatiin alkeellisten tulostimien avulla. Edelleenkin tietokonetta pystyi käyttämään vain koulutuksen saanut ammattilainen.

Seuraavaksi kehitettiin näyttö, jonka avulla tietokoneen vasteet voitiin lukea helposti. Kun tietokoneen ohjaukseen kehitettiin vielä nykyisen kaltainen näppäimistö, tuli tietokoneen käyttö mahdolliseksi myös tavallisille ihmisille, kuva 4. Käyttäjät joutuivat kuitenkin opettelemaan monimutkaisia näppäinkomentoja eri toimintojen suoritukseen, koska käyttöliittymät olivat vielä merkkipohjaisia.



Kuva 4. Tietokone näytöllä ja näppäimistöllä varustettuna.

Varsinainen mullistus tapahtui, kun graafinen käyttöliittymä (Graphical User Interface) kehitettiin 1960- ja 1970-luvuilla. Kehitystyötä tehtiin useassa tutkimuslaitoksessa ympäri maailmaa; Ensimmäiset siirrettävät, toistensa päällä olevat ikkunat kehitettiin kuitenkin Xerox-yhtiön Palo Alton tutkimuskeskuksessa. Graafisen käyttöliittymän avulla voitiin näytöllä esittää selvästi enemmän informaatiota kerralla, ja päästiin eroon kryptisistä näppäinyhdistelmistä, joita käyttäjien oli pitänyt opetella aikaisemmin. Graafisen käyttöliittymän yhteydessä esiteltiin myös hiiri, joka soveltui graafisen käyttöliittymän hallintaan näppäimistöä paremmin. Kuvassa 5 esitetään tärkeimpiä vaiheita käyttöliittymien kehityksessä. [7]



Kuva 5. Käyttöliittymien kehityksen vaiheita.

Nykyään lähes kaikki uudet sovellukset toimivat graafisessa käyttöjärjestelmässä. Tämä asettaa uusia vaatimuksia ohjelmistokehitystyökaluille ja itse sovelluksille. Hyvän graafisen käyttöliittymän tekeminen on usein vaikeaa.

Ohjelmistokehitystyökalut ovat parantuneet paljon viime vuosina, mutta vieläkin suuri osa graafisen käyttöliittymän suunnittelusta ja toteutuksesta on tehtävä käsin. Microsoftin Visual Basic ja Borlandin Delphi ovat ehkä pisimmälle vietyjä suunnittelun auto-

maation kannalta. Ne ovat niin sanottuja RAD-työkaluja (Rapid Application Development) [8]. RAD-työkalujen visuaalinen suunnittelu rajoittuu kuitenkin sovelluksen alkuvaiheisiin, ja loppuosa joudutaan yhä koodaamaan käsin. Työkalujen käyttö ei ole aina edes mahdollista lopullisessa ohjelmistoversiossa, vaikka ensimmäiset prototyypit saadaankin kehitettyä helposti niiden avulla.

Yksi rajoittava tekijä ovat ajettavan sovelluksen nopeusvaatimukset, jotka eivät välttämättä toteudu kaikilla kehitystyökaluilla ja ohjelmointikielillä. Joissain tapauksissa myös olemassa oleva koodi rajoittaa ohjelmointikielen valintaa: Tämä tarkoittaa sitä, että jos johonkin sovellukseen on jo tehty osia esimerkiksi C-kielellä, on helpointa toteuttaa myös lisäosat samalla kielellä kuin alkuperäinen ohjelmisto. Näin menettelemällä voidaan olemassa olevaa ohjelmistoa päivittää tiettyyn pisteeseen asti. Jossain vaiheessa päivittäminen alkaa kuitenkin vaikeutua ja käy lopulta mahdottomaksi lukuisten aikaisempien päivitysten takia.

2.2 Suunnittelun perusperiaatteita

Ben Shneiderman painottaa kirjassaan [7] inhimillisten tekijöiden merkitystä käyttöliittymien suunnittelussa. Hänen mukaansa suunnittelussa on otettava huomioon sovelluksen käyttäjäryhmän erityistarpeet.

Shneidermanin mukaan mitattavia inhimillisiä tekijöitä ovat

- tyypilliseen käyttäjäryhmään kuuluvan käyttäjän tarvitsema aika sovelluksen tärkeimpien ominaisuuksien oppimiseen
- sovelluksen käytön nopeus niissä tehtävissä, joita varten se on suunniteltu
- käyttäjien tekemien virheiden määrä
- opittujen asioiden muistettavuus
- käyttäjien omakohtainen tyytyväisyys.

Sovellusta suunniteltaessa on mietittävä tulevan käyttäjäryhmän tai käyttäjäryhmien kannalta mahdollisimman hyvä ratkaisu. Tässä työssä toteutettavan sovelluksen käyttäjäryhmänä on pääasiassa testauksen parissa työskentelevät ammattilaiset. Heidän vaatimuksinaan ovat ennen kaikkea sovelluksen monipuolisuus ja tehokkuus. Suunnittelussa on kuitenkin otettava huomioon myös muut käyttäjät, kuten uudet työntekijät, joille on tärkeää päästä nopeasti alkuun työkalun käytössä.

Edellä esitettyjä mitattavia inhimillisiä tekijöitä huomioon otettaessa on tehtävä kompromisseja, koska kaikkien tekijöiden yhtäaikainen optimointi on lähes mahdotonta. Esimerkiksi ohjelman käytön nopeus voi kärsiä, kun käyttäjien tekemien virheiden määrää pyritään vähentämään.

Sovelluksen ollessa kansainvälinen on otettava huomioon eri maiden ja maanosien kulttuurien erot. Eri kulttuureissa on käytössä erilaisia esitysmuotoja mm. ajalle, numeroille ja mittayksiköille.

2.2.1 Värien käyttö

Värien valinnalla voidaan vaikuttaa hyvinkin paljon sovelluksen käytettävyyteen. Liian värikäs käyttöliittymä voi ärsyttää käyttäjää, ja huonolla värien valinnalla käyttäjän huomio voidaan kohdistaa epäoleellisiin asioihin.

Värien käytöstä ei ole olemassa mitään yksiselitteisiä sääntöjä. Värien määrä kannattaa kuitenkin rajoittaa esimerkiksi neljästä seitsemään [7]. Kun värit vielä ovat hillittyjä ja kokonaisuus näyttää tasapainoiselta, ollaan menossa oikeaan suuntaan.

Värien valinnassa kannattaa pyrkiä johdonmukaisuuteen sovelluksen eri osien välillä. Samat komponentit ja toiminnot merkitään siis samoilla väreillä kaikissa sovelluksen osissa. Värejä voidaan käyttää myös erilaisten viestien korostamiseen; käyttäjän huomio voidaan kiinnittää esimerkiksi varoitukseen käyttämällä tiettyä väriä. Kun värit valitaan johdonmukaisesti esittämään erilaisia asioita, puhutaan värien muodostamasta koodikielestä. [7]

Parasta on, jos käyttäjälle annetaan mahdollisuus muokata sovelluksen värejä itselleen sopiviksi. Ominaisuuden toteuttaminen vaatii jonkin verran ylimääräistä suunnittelua ja ohjelmointia, mutta käyttäjät ovat varmasti tyytyväisempiä voidessaan muuttaa värit itselleen sopivimmiksi.

2.3 Graafisen käyttöliittymän elementit

Graafiset käyttöliittymät koostuvat erilaisista komponenteista. Ikkuna on yleensä sovelluksen peruskomponentti. Ikkunat voivat sisältää muita komponentteja, joista yleisimpiä ovat otsikot, tekstialueet ja erilaiset painikkeet. Käyttöjärjestelmät esittävät graafiset komponentit jokainen omalla tavallaan, ja niissä ei välttämättä ole toteutettu kaikkia graafisia komponentteja. Seuraavassa esitellään lyhyesti sellaiset komponentit, jotka löytyvät lähes jokaisesta graafisesta käyttöjärjestelmästä.

2.3.1 Ikkunat

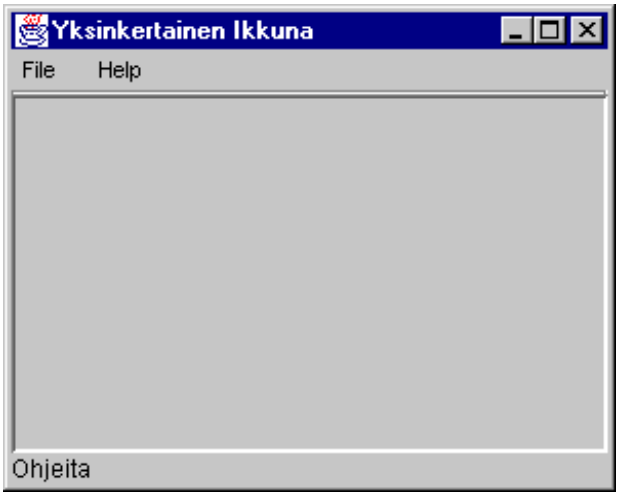
Ikkunat ovat graafisen käyttöliittymän peruskomponentteja, kuva 6. Ikkunoita voidaan yleensä siirtää, pienentää ikoniksi ja suurentaa koko ruudun kokoisiksi. Yleisimpiä ikkunatyyppejä ovat:

- **Kehykset**. Kehykset (frame) ovat pohjana lähes kaikille sovelluksille. Kehys on tavallinen ikkuna, jollainen aukeaa esimerkiksi silloin, kun sovellus käynnistetään.

- **Dialogi-ikkunat**. Dialogi-ikkunoita eli dialogeja käytetään informaation ja palautteen välittämiseen käyttäjälle, samoin kuin sovelluksen ja käyttäjän väliseen vuoropuheluun. Esimerkiksi tiedostoa tuhottaessa voidaan varmistaa, että käyttäjä todella haluaa tuhota tiedoston. Dialogit voivat olla modal- tai modeless-tyyppisiä. Modal-tyypin dialogi estää muun sovelluksen käytön, kunnes dialogi on suljettu. Useita modeless-tyyppisiä dialogi-ikkunoita voi olla avattuna yhtä aikaa eivätkä ne estä muun sovelluksen käyttöä. Dialogit voivat sisältää muita komponentteja. Erilaisia dialogeja on olemassa suuri määrä.

- **Kelluvat ikkunat**. Kelluvat ikkunat (floating window) ovat yleistyneet vasta viime vuosina. Niiden avulla on mahdollista toteuttaa esimerkiksi työkaluikkunoita, joita voidaan siirrellä kehysikkunan sisällä. Tällaisten kelluvien työkaluikkunoiden avulla käyttäjä voi helposti muokata sovelluksen ulkoasun itselleen mieluisaksi. Monesti kelluvat ikkunat voidaan kiinnittää myös ns. työkalupalkkiin.

- **Lapsi-ikkunat**. Esimerkiksi tekstinkäsittelyohjelmissa, kuten Microsoft Wordissa, käytetään pääikkunana niin sanottua MDI-ikkunaa (Multiple Document Interface). Tällainen MDI-ikkuna voi sisältää keskenään samanarvoisia lapsi-ikkunoita, joita voidaan siirrellä vapaasti MDI-ikkunan sisällä.



Kuva 6. Esimerkki yksinkertaisesta ikkunasta.

2.3.2 Valikot

Valikoissa esitetään kaikki ohjelman toiminnot jaoteltuna toiminnallisiin kokonaisuuksiin. Valikot on järkevää toteuttaa kohdeympäristön käyttöjärjestelmän standardin tai yleisen käytännön mukaisiksi, jolloin sovelluksen käytön oppiminen on helpompaa, kuva 7.

Kuva 7. Valikko.

2.3.3 Työkalupalkit

Työkalupalkeissa esitetään yleisimmin käytetyt sovelluksen toiminnot, kuva 8. Työkalupalkki on usein sovellusikkunan yläosassa, heti valikon alapuolella. Myös ikkunan reunassa sijaitseva, koko ikkunan korkuinen työkalupalkki on yleinen. Nykyään työkalupalkeista voidaan tehdä myös siirrettäviä ja kelluvia.

Kuva 8. Työkalupalkki.

2.3.4 Tilapalkit

Tilapalkeissa näytetään sovelluksen käyttäjälle antamia informatiivisia tietoja. Tiedot eivät ole sovelluksen kannalta kriittisiä, vaan tilapalkissa voidaan näyttää esimerkiksi ohjeita silloin kun käyttäjä siirtää hiiren osoitinta eri komponenttien päälle.

2.3.5 Muut graafisen käyttöliittymän komponentit

Muita tärkeitä graafisen käyttöliittymän komponentteja ovat

- **Painonapit**. Painonappeja (button) käytetään toimintojen suoritukseen. Painonapit ovat perinteisesti sisältäneet toimintaa kuvaavan tekstin, mutta nykyään on yleistä, että ne sisältävät myös pienen kuvan eli ikonin.

- **Otsikot**. Otsikot (label) ovat tekstin esittämiseen tarkoitettuja alueita, joiden sisältämää tekstiä voi muokata vain ohjelman koodista eikä käyttäjällä ole siihen mahdollisuutta.

- **Tekstialueet**. Tekstialueet ovat nimensä mukaisesti tekstin esittämistä ja kirjoittamista varten. Tekstialueet voidaan jakaa yhden ja useamman rivin kokoisiin komponentteihin.

- **Valintalistat**. Valintalistoja käytetään yleensä tekstimuotoisen tiedon esittämiseen ja halutun tiedon valitsemiseen esimerkiksi muokkaamista tai poistamista varten.

- **Puut**. Puut ovat monimutkaisempia rakenteita, jotka ovat tulleet useimmille tutuiksi mm. Microsoft Windowsin tiedostonhallintasovelluksesta.

- **Taulukot**. Taulukkoja käytetään taulukkomuotoisen tiedon esittämiseen ja muokkaamiseen.

2.4 Oliopohjaiset käyttöliittymät

Joissain lähteissä kaikkia graafisia käyttöliittymiä pidetään oliopohjaisina. Toisissa lähteissä taas oliopohjaisuus rajataan sellaisiin käyttöliittymiin, jotka käyttävät ikoneita ja muita graafisia tekniikoita reaalimaailman objektien, kuten dokumenttien, esittämiseen. Toisaalta ohjelmoijat, jotka käyttävät oliopohjaisia kieliä käyttöliittymän toteuttamiseen, pitävät toteuttamiaan käyttöliittymiä oliopohjaisina. [9]

Tässä työssä käsitellään oliopohjaisina sellaisia käyttöliittymiä, joissa reaalimaailman käsitteet esitetään tietokonemaailman olioina ja jotka on toteutettu oliopohjaista ohjelmointikieltä, kuten SmallTalkia, C++:aa tai Javaa, käyttäen.

Larry Teslerin [10] mukaan oliopohjaisen käyttöliittymän ominaisuuksia ovat

- Käyttäjät näkevät käyttöliittymän objektit ja valinnat graafisina komponentteina.

- Komennot toteutetaan objektien avulla.

- Käyttäjä saa välittömän palautteen suorittamistaan toiminnoista.

- Käyttöliittymä on mooditon eli siinä ei ole erillisiä käyttötiloja.

- Objektit näytetään WYSIWYG-muodossa.

- Objektit ja toiminnot ovat yhdenmukaisia saman sovelluksen sisällä ja eri sovellusten välillä.

2.4.1 Oliopohjaisen käyttöliittymän edut

Oliopohjainen käyttöliittymä perustuu reaalimaailman käsitteisiin. Esimerkiksi tietokoneen työpöydällä on roskakori, johon tarpeettomat tiedostot voi siirtää, kuva 9. Työpöydän roskakorin tehtävänä on säilyttää roskia. Työpöydällä voi olla myös silppuri, johon käyttäjä voi viedä sellaiset tiedostot, jotka hän haluaa tuhota. Tuhoaminen on tällöin lopullista toisin kuin roskakorissa olevilla tiedostoilla, jotka käyttäjä voi halutessaan ottaa takaisin käyttöön. Roskakori voidaan myös tyhjentää, jolloin siinä olevat tiedostot tuhotaan lopullisesti.

Kuva 9. Tiedoston siirtäminen Windows NT 4.0:n roskakoriin.

Roskakorin ja silppurin toiminta on samankaltaista sekä tietokone- että reaalimaailmassa. Tämä madaltaa oppimiskynnystä ja helpottaa toimintojen muistamista. Jos vastaavat roskakori- ja silppuritoiminnot olisi toteutettu perinteisessä merkkipohjaisessa käyttöliittymässä, olisi niille pitänyt toteuttaa omat komentonsa.

Monia graafisen käyttöliittymän ominaisuuksia voidaan pitää suoravaikutteisina. Objektien käsittely on suoraa ja vaste saadaan välittömästi. [7] Tällöin käyttäjä kokee hallitsevansa sovellusta käyttöliittymän kautta.

2.4.2 Oliopohjaisen käyttöliittymän suunnittelu

Hyvän oliopohjaisen käyttöliittymän suunnittelu on vaikeaa. Työssä voidaan epäonnistua, jos suunnittelija ei hallitse suunnittelumenetelmiä tai suunnitteluun ei ole käytetty riittävästi aikaa. Toisaalta käyttöliittymästä voi saada todella helppokäyttöisen ja saumattoman, jos suunnitteluun panostetaan riittävästi.

Oliopohjaisen käyttöliittymän suunnittelu voidaan Collinsin mukaan [9] jakaa kolmeen osa-alueeseen, jotka ovat järjestelmän käsitteellinen malli, käyttöliittymän tarjoamien objektien esittäminen ja objektien toiminnallisuuden mahdollistavat järjestelmän rakenteet.

Ensimmäinen osa-alue eli järjestelmän käsitteellisen mallin suunnittelu vastaa lähinnä ohjelmistokehityksen analyysivaihetta. Toisen osa-alueen suunnittelussa keskitytään käyttöliittymän objektien esittämiseen. Lopuksi suunnitellaan objektien toiminnallisuus.

3. Graafisen käyttöliittymän toteutus Javalla

Tämän luvun tarkoituksena on tarkastella graafisen käyttöliittymän toteutusta Java-kielellä. Alussa käydään läpi yleisiä käyttöliittymän toteutuksessa käytettäviä prosessimalleja ja loppupuolella tarkastellaan Javan erityispiirteitä käyttöliittymien toteutuksessa.

3.1 Oliopohjainen ohjelmointikieli käyttöliittymän toteutuksessa

Oliopohjaisen kielen käytöstä ohjelmistojen toteutukseen on erilaisia mielipiteitä. Perinteisen modulaarisen ohjelmointityylin kannattajat ovat esittäneet eräänä perusteena oliokieliä vastaan niiden huonomman suorituskyvyn verrattuna esimerkiksi C-kieleen. Nykyisin suorituskykyerot ovat suhteellisen pieniä, ja oliopohjaisten kielten käyttö ohjelmistokehityksessä on yleistynyt nopeasti.

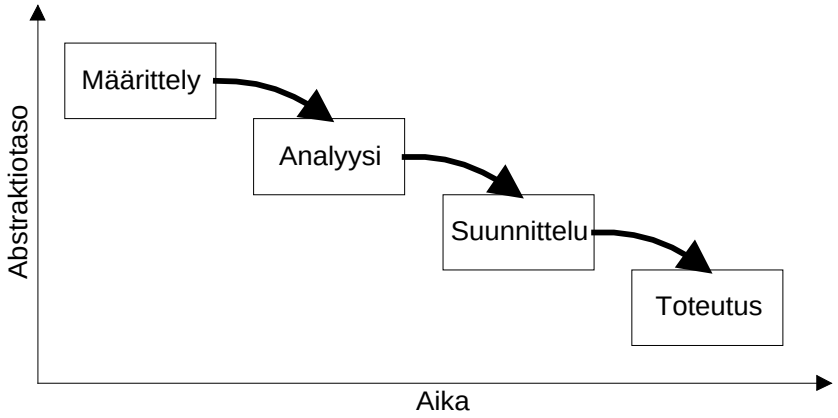
Graafisten käyttöliittymien kehitykseen oliopohjaiset kielet soveltuvat erityisen hyvin. Koska nykyaikaiset graafiset käyttöliittymät pyritään tekemään mahdollisimman oliopohjaisiksi, on luonnollista valita myös ohjelmointikieleksi oliopohjainen vaihtoehto.

3.1.1 Ohjelmistokehityksen vaiheet

Ohjelmistokehitys on yleensä välttämätöntä jakaa vaiheisiin, jotka voidaan esittää ns. prosessimallien avulla. Seuraavana on esitetty yleisesti ohjelmistokehityksessä käytetyt prosessimallit, joita voidaan soveltaa myös oliopohjaisen graafisen käyttöliittymän kehitykseen.

Vesiputousmalli

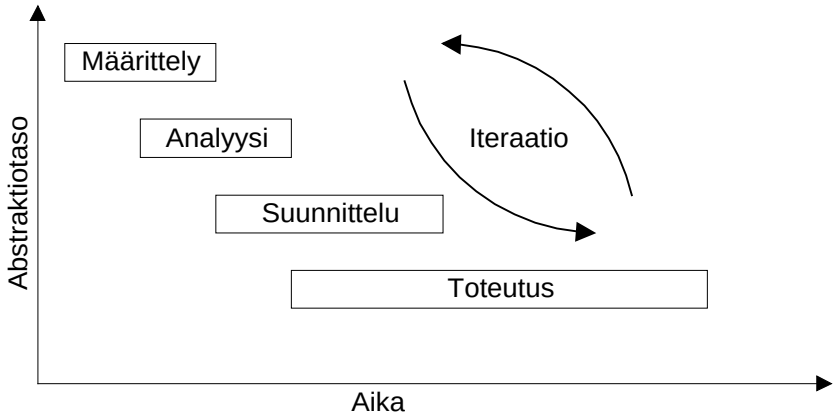
Vesiputousmalli on perinteinen malli, jota on käytetty usein ohjelmistokehityksessä. Siinä ohjelmistokehityksen ajatellaan etenevän vaiheesta toiseen suoraviivaisesti, kuva 10. Ongelmana on kuitenkin ohjelmistoprojektin kuluessa esiintulevat muutokset, jotka vaativat paluuta edelliseen vaiheeseen. Perinteinen vesiputousmalli ei tue tällaista iteraatiota.



Kuva 10. Vesiputousmalli.

Rinnakkainen, iteratiivinen malli

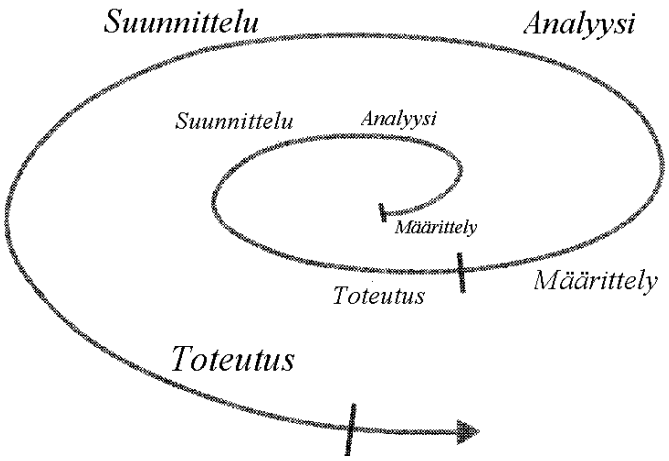
Vesiputousmallia voidaan laajentaa ottamaan huomioon todellisen ohjelmistoprojektin kehityksessä esiintulevat vaiheet. Eri vaiheet menevät osittain päällekkäin ja aikaisempiin vaiheisiin voidaan palata aina tarpeen vaatiessa, kuva 11.



Kuva 11. Rinnakkainen, iteratiivinen malli.

Spiraalimalli

Barry Boehm esitteli spiraalimallin ratkaisuna vesiputousmallin ongelmiin [11]. Spiraalimallissa vesiputousmallin vaiheet käydään läpi useaan kertaan, jolloin mahdollisten muutosten vaikutukset voidaan ottaa huomioon paremmin. Spiraalimalli esitetään kuvassa 12. [9]



Kuva 12. Spiraalimalli.

Prototypointi

Prototypoinnin pohjana voidaan käyttää spiraalimallia, jossa kunkin iteraatiokierroksen lopussa saadaan käyttöliittymän prototyyppi. Tätä prototyyppiä kehitetään iteraatiokierrosten aikana. Prototypointia voidaan käyttää erityisesti graafisten käyttöliittymien kehityksessä, koska käyttöliittymän ensimmäisten prototyyppien ei tarvitse sisältää kuin tärkeimmät toiminnot. Lisätoimintoja voidaan lisätä myöhemmillä iteraatiokierroksilla.

Prototypoinnin etuina voidaan mainita [9]

- Käyttäjiltä voidaan saada palautetta helpommin, koska he voivat prototyypin avulla saada paremman kuvan siitä, millainen lopullinen sovellus tulee olemaan.

- Virheet vaatimusmäärittelyssä ja suunnittelussa paljastuvat aikaisessa vaiheessa.

- Prototyyppejä voidaan käyttää suunnittelun dokumentteina paperilla olevia määrittelyjä paremmin.

Prototypoinnissa on myös ongelmia [9]

- Prototyyppien rakentaminen voi olla hyvin työlästä.

- Ohjelmiston analyysi ja suunnittelu voivat kärsiä.

- Ohjelmiston kannalta välttämättömiä ja kehitystyökalun lisäämiä ylimääräisiä ominaisuuksia ei voida erotella toisistaan.

- Prototyyppi voi antaa väärän kuvan ohjelmiston todellisesta valmiusasteesta. Ohjelmiston hyödyntäjä voi prototyypin perusteella saada liian optimistisen kuvan ohjelmiston valmistumisaikataulusta.

Yhteenvetona voidaan todeta, että iteraatio ja prototypointi soveltuvat oliopohjaisen käyttöliittymän kehitykseen hyvin, mutta niiden hallintaan täytyy kiinnittää erityistä huomiota.

3.1.2 Uudelleenkäytettävyys

Ohjelmiston uudelleenkäytettävyydestä on puhuttu jo 1960-luvulta lähtien. Tästä huolimatta vielä nykyäänkään ei osata hyödyntää tarpeeksi hyvin jo olemassa olevia ratkaisuja uuden ohjelmiston toteutuksessa. Jos ohjelmistoprojektissa päästään 50 prosentin uudelleenkäyttöön, voidaan kehityskustannukset periaatteessa puolittaa. Todellisuudessa näin suurta säästöä ei saavuteta, koska uudelleenkäytettävien ohjelmistokomponenttien kehitystyö on vaikeampaa ja kalliimpaa.

Toteutettaessa uudelleenkäytettäviä ohjelmiston osia on hyvä etukäteissuunnittelu tärkeää. Osista on tehtävä sellaisia, että niitä voidaan käyttää mahdollisimman useasti samankaltaisissa sovelluksissa. Laatuun on myös kiinnitettävä entistä enemmän huomiota. Toteutetut komponentit on tuotava ohjelmistokehittäjien saataville ja ylläpitoa varten on järjestettävä menettely.

Vaikka uudelleenkäytettävien ohjelman osien kehitystyö on työläämpää kuin yhtä tarkoitusta varten tehtävien osien kehitys, saadaan niistä usein parempilaatuisia. Koodissa olevat virheet karsiutuvat pois, kun samoja komponentteja käytetään useamman kerran.

Ohjelmiston uudelleenkäyttöä rajoittavat Coadin ja Yourdonin [12] mukaan seuraavat seikat:

- Ohjelmointia opettavat kirjat eivät painota uudelleenkäytettävyyttä.

- Ohjelmiston kehittäjien halu "tehdä kaikki itse".

- Epäonnistuneet kokemukset uudelleenkäytöstä.

- Yritykset eivät palkitse uudelleenkäytettäviä osia tekevää ohjelmoijaa, vaan usein otetaan huomioon vain koodirivien määrä tuottavuutta arvioitaessa.

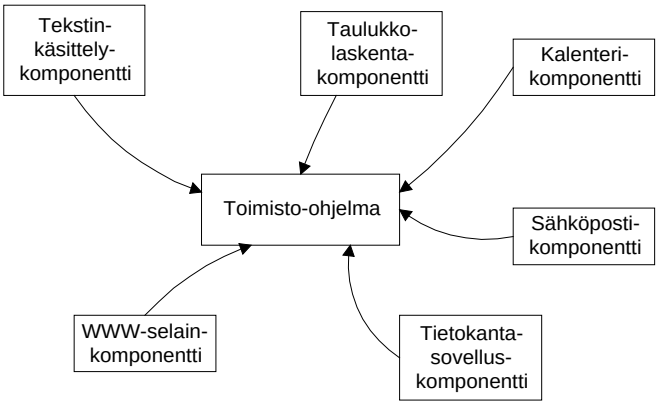
Ensimmäinen askel uudelleenkäytössä on aikaisemmin tehdyn koodin hyödyntäminen. Siihen on olemassa eriasteisia keinoja. Yksinkertaisin tapa on ns. "leikkaa ja liimaa"-tekniikka, jossa olemassa olevasta koodista kopioidaan osia. Kopioidut osat otetaan käyttöön pienen muokkaamisen jälkeen toisessa osassa koodia tai kokonaan eri ohjelmistossa.

Toinen tapa on valmiin koodin ottaminen käyttöön muokkaamatta sitä (source-level include). Tämä on tuttua mm. C-kielestä, jossa standardikirjaston osia tai itse tehdyn koodin osia voi sisällyttää include-lauseella uusiin toteutettaviin osiin. Olioperustaiset kielet tarjoavat kooditason uudelleenkäyttöön tehokkaan periyttämismenetelmän. Luokkia voidaan toteuttaa helposti periyttämällä niitä jo olemassa olevista luokista. Muita mahdollisuuksia ovat binäärien linkitys ja ajonaikaiset kutsut. Ajonaikainen funktioiden lataaminen ja kutsuminen on yleistä tulkkaavissa kielissä.

Koodin uudelleenkäytön lisäksi voidaan esimerkiksi onnistunutta luokkamallia käyttää hyväksi suunniteltaessa toisia samantyyppisiä sovelluksia. Tällöin suunnittelijan ei tarvitse aloittaa puhtaalta pöydältä, vaan hänen tukenaan on olemassa oleva, toimiva luokkamalli. Jos suunnittelussa esille tulevat ongelmat poikkeavat paljon toisistaan, ei suunnittelutulosten uudelleenkäyttö ole välttämättä järkevää.

3.1.3 Uudelleenkäytettävät komponentit

1990-luvun puolenvälin jälkeen on kokonaisten komponenttien uudelleenkäytöstä tullut tärkeä tutkimusalue. Tavoitteena on päästä ohjelmistokehityksessä tilanteeseen, joka vallitsee esimerkiksi rakennusalalla. Rakennusalalla voidaan valita talon rakennuksessa käytettävät komponentit valmiiden elementtien joukosta. Ovet, ikkunat, ja muut rakentamisessa tarvittavat osat ovat tiettyjen standardien mukaisia, ja niitä yhdistelemällä saadaan kasattua kokonainen talo. Ohjelmistokehityksessä tämä voisi merkitä esimerkiksi ohjelmistoa, joka on koottu standardoituja komponentteja yhdistelemällä, kuva 13.



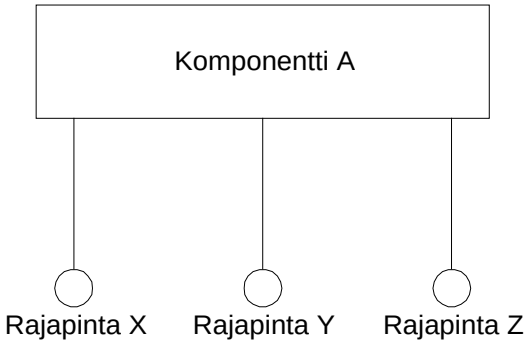
Kuva 13. Komponenteista koottu toimisto-ohjelma.

Oliotekniikoiden tullessa laajemmin julkisuuteen 1980-luvulla ennustettiin, että ns. ohjelmistokriisi (software crisis) [13] olisi kokonaan ratkaistavissa oliotekniikoiden avulla.

Käytäntö on kuitenkin osoittanut, että oliotekniikat eivät yksin pysty ratkaisemaan kaikkia nykyaikaisen ohjelmistokehityksen ongelmia. Vastauksena oliotekniikoiden tuottamaan pettymykseen alettiin 1990-luvun alussa puhua komponenttien uudelleenkäytöstä oliotekniikoiden vaihtoehtona. Innokkaimmat komponentoinnin kannattajat olivat jopa sitä mieltä, että se korvaisi kokonaan oliot.

Oliotekniikat ja komponentointi on kuitenkin mahdollista yhdistää. Itse asiassa komponentteja voidaan kehittää olioita yhdistelemällä. Tällöin olion voidaan ajatella olevan suhteellisen pieni osa, ja komponentin olevan olioista koostuva laajempi kokonaisuus.

Ohjelmistokomponentit liitetään toisiinsa rajapintojen avulla. Kaikki komponenttien välinen kommunikointi toteutetaan näitä rajapintoja hyväksikäyttämällä. Komponentin rajapinnat muodostavat kyseisen komponentin palvelut, kuva 14.



Kuva 14. Komponentti ja sen toteuttamat rajapinnat.

Nykyiset ohjelmointikielet tukevat komponenttien käyttöä jossain määrin. Microsoftin kehittämä Visual Basic on ollut edelläkävijä valmiiden komponenttien hyödyntämisessä. Nykyään Visual Basic ja muut Microsoftin kehitystyökalut käyttävät pääasiassa Microsoftin kehittämää COM-komponenttimallia. Se on perustana kaikille ActiveX-teknologioille. [14]

ActiveX-komponenteista on tullut melko suosittuja, ja esimerkiksi vuonna 1996 niiden kaupan arvo oli 240 miljoonaa dollaria. Microsoftin mukaan kaupallisten ActiveX-kontrollien määrä on tällä hetkellä yli 2 000. Kyseisen tekniikan kanssa kilpaillut OpenDoc on menettänyt asemiaan ja tällä hetkellä sen kehitys on lähinnä IBM:n varassa [15].

Javan vastaus Microsoftin ActiveX-komponenteille on nimeltään JavaBeans. JavaBeans-tekniikan avulla on mahdollista kehittää ympäristöstä riippumattomia valmiskomponentteja. JavaBeans ei ole kuitenkaan saavuttanut merkittävää asemaa, koska

siinä käytetty teknologia on vielä uutta. JavaBeans-komponentit [16] voidaan yhdistää ActiveX-komponentteihin ns. JavaBeans-sillan (JavaBeans Bridge) avulla.

3.2 Java-kielen tarjoamat uudet mahdollisuudet

Tämän luvun tarkoituksena on tarkastella Javan vaikutusta graafisten käyttöliittymien suunnitteluun ja toteutukseen. Yksi Javan suurimmista lupauksista koskee graafisten käyttöliittymien siirrettävyyttä eri ympäristöjen välillä. Komponentoinnin avulla uskotaan lisäksi olevan mahdollista nopeuttaa käyttöliittymien toteutusta.

3.2.1 Siirrettävyys eri ympäristöjen välillä

Lähes kaikissa ohjelmointikielissä on ollut erilaisia standardikirjastoja jo pitkään. Tällaisten standardikirjastojen tarkoituksena on helpottaa ohjelmoijien työtä tarjoamalla valmiita alemman tason funktioita. Näin ohjelmoija voi toteuttaa sovelluksia ylemmällä tasolla ja hänen tuottavuutensa kasvaa. Standardikirjastot on kuitenkin suunniteltu yleiskäyttöisiksi, eivätkä ne ole tukeneet esimerkiksi graafisten käyttöliittymien toteutusta.

Oliopohjaisissa kielissä on ollut mahdollista käyttää valmiita luokkakirjastoja, joita on ollut saatavilla mm. ohjelmistokehitystyökalujen mukana. Nämä luokkakirjastot ovat tarjonneet luokkia myös graafisten käyttöliittymien tekoon.

Oliopohjaisten kielten luokkakirjastoissa on ollut monia puutteita. Esimerkiksi C++:ssa on käytetty Borlandin kehittämää OWL-luokkakirjastoa ja Microsoftin kehittämää MFC-luokkakirjastoa. Kumpaakaan ei ole standardoitu C++:n viralliseksi luokkakirjastoksi, eivätkä ne ole olleet keskenään yhteensopivia.

Toinen suuri puute eri valmistajien tekemissä luokkakirjastoissa on ollut heikko siirrettävyys eri laite- ja käyttöjärjestelmien välillä. Sekä OWL- että MFC-luokkakirjastot on tehty pelkästään Microsoft Windows -käyttöön, ja tämä on suuresti rajoittanut niiden käyttöä. Tehtäessä sovellusta useaan ympäristöön jokaiselle ympäristölle on pitänyt koodata erilliset käyttöliittymän toteuttavat osat.

Java tuo muutoksen tarjoamalla kieleen sisältyvät luokkakirjastot, jotka toimivat kaikissa ympäristöissä. Ainoa vaatimus on, että ympäristölle on saatavilla Java Virtual Machine (JVM). Siirrettävyys koskee kaikkia luokkakirjaston osia, myös graafisten käyttöliittymien tekoon tarkoitettuja luokkia. Javan standardia luokkakirjastoa käyttäen saadaan kehitetystä sovelluksesta täysin siirrettävä. Tämä tarjoaa todellisen haasteen kaikille muille ohjelmointikielille. Javaa käytettäessä riittää, että kehitetään vain yksi versio,

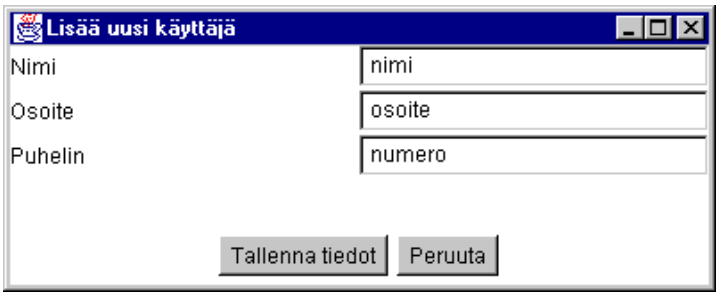
joka toimii joka paikassa. Sun MicroSystems onkin käyttänyt Javaa markkinoidessaan iskulausetta "Write-once-run-everywhere" [17].

3.3 Abstract Window Toolkit

Abstract Window Toolkit eli AWT on ollut jo Javan ensimmäisistä versioista lähtien osa kieleen sisältyvää luokkakirjastoa. AWT sisältää luokkia, joiden avulla voidaan helposti toteuttaa graafisia komponentteja eli ikkunoita, nappuloita, tekstialueita, kuvia ja muita graafisen käyttöliittymän peruskomponentteja.

AWT-kirjaston avulla voidaan toteuttaa graafisia käyttöliittymiä, jotka ovat joustavasti siirrettävissä eri laitteisto- ja käyttöjärjestelmäympäristöjen välillä. Sovellukset, jotka on tehty käyttäen AWT-luokkia, näyttävät lähes samanlaisilta kaikissa ympäristöissä.

AWT-kirjaston avulla toteutettavat käyttöliittymät ovat kuitenkin rajoittuneita. Niistä ei saada yhtä joustavia kuin käyttöjärjestelmän tarjoamia ohjelmointirajapintoja suoraan käyttämällä. AWT-kirjaston avulla toteutettu käyttöliittymä toimii kaikissa ympäristöissä kohtuullisen hyvin, mutta ei ole yhtä viimeistelty kuin perinteisillä ohjelmointikielillä tehty käyttöliittymä, kuva 15.



Kuva 15. Esimerkki yksinkertaisesta AWT:lla tehdystä ikkunasta.

Käyttöliittymiä kehitettäessä komponentit on yleensä sijoitettu tarkasti tiettyihin paikkoihin näytöllä, niin että on määrätty vasemman yläreunan koordinaatit sekä komponentin pituus ja korkeus.

Koska Java on suunniteltu toimimaan monessa eri ympäristössä, ei tällainen tarkka komponenttien sijoittaminen ole mahdollista. Eri ympäristöissä on käytössä erilaiset koordinaattisysteemit, ja erilaiset näytöt asettavat rajoituksia. Javalla tehdyn sovelluksen tulee ainakin periaatteessa toimia yksinkertaisimmasta kämmenmikrosta monipuolisiin työasemiin. Tällöin on selvää, ettei komponenttien paikkaa näytöllä voi määrätä tarkasti.

AWT:ssa käytetäänkin Layout Manager -järjestelmää, jossa ohjelmoija ei määrittele graafisille komponenteille tarkkoja koordinaatteja. Vain komponenttien keskinäiset suhteet määritellään, ja Javan Layout Manager laskee sovelluksen suorituksen aikana kullekin komponentille sopivimman paikan ja koon. Näin sama sovellus saadaan skaalattua toimimaan kaikissa ympäristöissä.

AWT-kirjasto sisältää useita erilaisia Layout Manager -toteutuksia. Kokoojiksi (container) kutsutaan sellaisia AWT-komponentteja, jotka voivat sisältää muita komponentteja. Niille voidaan määritellä haluttu ulkoasu setLayout-metodilla. AWT-kirjastossa on toteutettu seuraavat Layout Manager -luokat: BorderLayout, FlowLayout, GridLayout, CardLayout ja GridBagLayout.

3.4 Java Foundation Classes -luokkakirjasto

Java Foundation Classes on uusi Java-ytimeen sisältyvä luokkakirjasto. JFC-luokkia ei tarvitse toimittaa sovelluksen mukana, vaan ne löytyvät jokaisesta Java-ympäristön sisältävästä koneesta. JFC laajentaa alkuperäistä AWT-kirjastoa uusilla graafisten käyttöliittymien tekoon tarkoitetuilla luokkakirjastoilla. Se on siirrettävä eri ympäristöjen välillä kuten AWT:kin. JFC-kirjastoa käyttäen sovelluksista saadaan kuitenkin laadukkaampia ja suorituskykyisempiä kuin perinteisen AWT:n avulla. [18]

3.4.1 JFC:n ominaisuudet

JFC-kirjasto tuo mukanaan useita uusia ominaisuuksia, jotka esitellään lyhyesti seuraavassa.

"Swing" sai nimensä JavaOne-kokouksessa San Franciscossa vuonna 1997, kun insinöörit, jotka olivat olleet kehittämässä uutta komponenttijoukkoa, suunnittelivat musiikkia sisältävää demoa. Swing-tiimin jäsen Georges Saab mainitsi, että swing-musiikki oli tulossa uudelleen muotiin ja ehdotti, että Swing voisi olla hyvä nimi uudelle projektille. Kaikki hyväksyivät ehdotuksen, ja Swing-nimestä tuli hyvä iskulause mainonnassa. [19]

Swing on JFC:hen sisältyvä kirjasto, joka sisältää graafisen käyttöliittymän kehityksessä tarvittavia visuaalisia komponentteja. Niitä ovat muun muassa valikot, työkalupalkit ja dialogit. [19]

Käyttäen Swingiä on mahdollista kehittää käyttöliittymä, joka voidaan siirtää lähes kaikkiin laitteisto- ja käyttöliittymäympäristöihin sellaisenaan. Swing-kirjaston avulla

saadaan kehitetty käyttöliittymä näyttämään siltä kuin se olisi kehitetty ympäristöön, missä sitä ajetaan.

Swingin yhteydessä käytetään usein englanninkielistä termiä “look and feel”, jolle ei ole vakiintunutta suomenkielistä ilmaisua. Termi tarkoittaa, että ajettaessa Java-sovellusta esimerkiksi Windowsissa, sovelluksen ulkonäkö ja käyttäytyminen ovat samanlaisia kuin muidenkin Windows-sovellusten. Ajettaessa samaa Java-sovellusta UNIX-koneessa se näyttää ja käyttäytyy samalla tavalla kuin muutkin UNIX-sovellukset. Swing-kirjastoa käsitellään jäljempänä vielä erikseen, koska se on tärkein osa JFC-luokkakirjastoa.

"Pluggable look and feel" liittyy läheisesti Swing-komponentteihin. Se mahdollistaa sovelluksen ulkonäön ja käyttäytymisen mukauttamisen käyttöjärjestelmälle sopivaksi. Sovelluksen ulkonäkö ja käyttäytyminen voidaan määrätä jo kehitysvaiheessa ja sovellus voidaan asettaa kiinteästi esimerkiksi Motif-tyyppiseksi. Tällöin käyttöliittymä on aina Motifin näköinen ja tuntuinen riippumatta siitä, missä ympäristössä sitä ajetaan.

Swing sisältää myös kokonaan uudennäköisen käyttöliittymän, jonka nimi on Java look and feel. Sen avulla on mahdollista muodostaa Java-sovelluksille täysin käyttöjärjestelmästä riippumaton ulkonäkö ja käyttäytyminen. Tässä yhteydessä voidaan mainita, että Microsoft on kieltänyt käyttämästä Windowsin näköistä käyttöliittymää muissa käyttöjärjestelmissä kuin Windows.

"Accessibility" tarjoaa rajapinnan JFC- ja AWT-komponenttien käyttöön henkilöille, joilla on jokin vamma tai vajavuus, joka estää tai rajoittaa heidän kykyään käyttää tietokonetta normaaliin tapaan. [20]

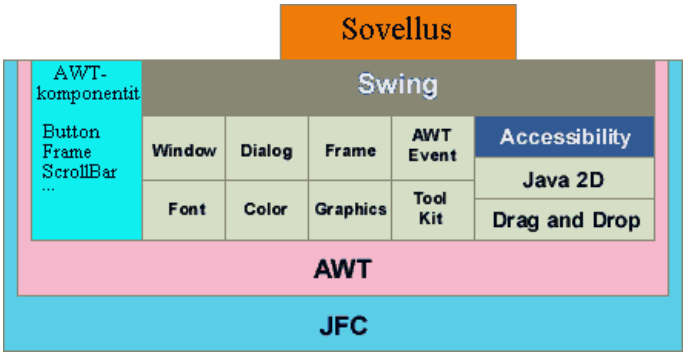
"Java 2D API" koostuu puolestaan luokista, jotka on tarkoitettu helpottamaan ja monipuolistamaan kaksiulotteisen grafiikan tekemistä Javalla. [21]

"Drag and drop" parantaa huomattavasti Javalla tehtyjen ja muiden sovellusten välistä vuorovaikutusta. Se mahdollistaa toiminnot, joiden toteuttaminen on tähän asti ollut erittäin vaikeaa. Ne ovat tärkeä osa nykyaikaisia graafisia käyttöliittymiä, joten JFC:n mukana tuleva tuki on erittäin tarpeellinen.

3.4.2 JFC Swing – siirrettävyyttä ja monipuolisuutta

Swing on graafisia komponentteja sisältävä JFC:n osa, joka on kehitetty alkuperäisen AWT:n rinnalle. Swing ei siis korvaa kokonaan AWT:ta. Monessa tapauksessa Swing-komponenttien käytöstä on kuitenkin niin paljon etuja vastaaviin AWT-komponenttei-

hin verrattuna, että Swingin käyttö on paljon järkevämpää. Kuvassa 16 [19] on esitetty, miten Swing sijoittuu suhteessa AWT:hen. AWT- ja Swing-komponentteja voi käyttää rinnakkain, jolloin sovellus hyödyntää komponentteja molemmista kirjastoista.



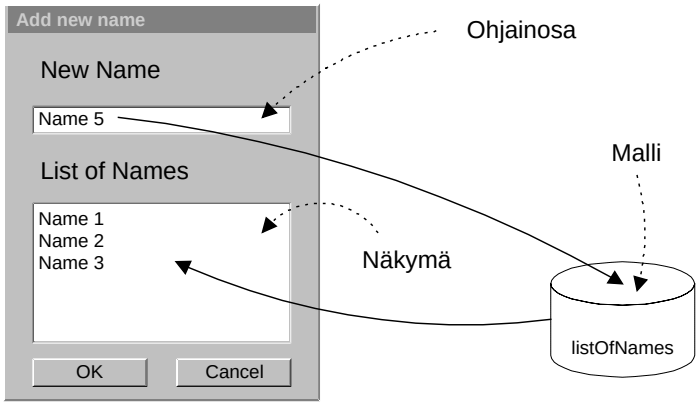
Kuva 16. Swingin suhde AWT:hen ja JFC:hen.

Swing on ensimmäinen graafisten käyttöliittymien kehittämiseen suunniteltu luokkakirjasto, joka mahdollistaa ns. keveiden komponenttien luonnin, joihin on mahdollista liittää kytkettävä ulkonäkö ja toiminta. Jo JDK:n versio 1.1 toi keveät komponentit AWT:hen, ja Swing laajentaa AWT-komponentteja mahdollistamalla tietyn ulkonäön ja toiminnan liittämisen niihin. [19]

Kevyt komponentti ei ole riippuvainen sen järjestelmän käyttöliittymän koodista, jossa sovellusta ajetaan. Tällaisten komponenttien toteutukseen tarvitaan vain puolet aikaisempien komponenttien toteutukseen tarvittavista luokista. [19]

3.4.3 Model View Controller -arkkitehtuuri

Swing pohjautuu MVC-arkkitehtuuriin (model-view-controller), joka tuotettiin jo SmallTalk-kieltä [22] kehitettäessä. MVC-arkkitehtuurissa komponentit jaetaan kolmeen tyyppiin. Malli (model) esittää sovelluksessa käsiteltävää tietoa. Näkymä (view) on tiedon visuaalinen esitys, ja ohjaimen tehtävänä on muokata mallin sisältämää tietoa. Ensimmäiset Swingin prototyypit toteuttivat tämän perinteisen MVC-mallin täydellisesti.



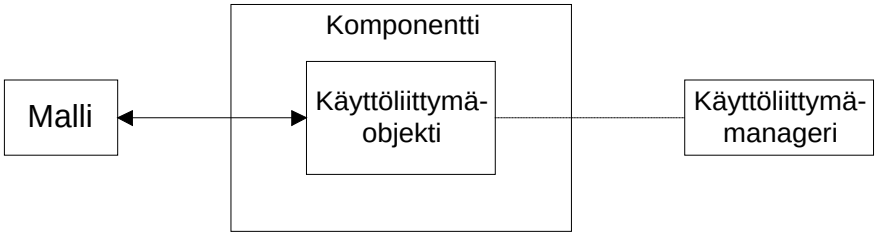
Kuva 17. Yksinkertainen model-view-controller-arkkitehtuurin toteutus.

Kuvassa 17 on esitetty yksinkertainen ikkuna, jonka toiminta pohjautuu MVC-malliin. Ikkunassa on mahdollista lisätä uusi nimi listaan. Lisäys tehdään antamalla nimi tekstin “New Name” alla olevaan tekstikenttään ja painamalla sen jälkeen rivinvaihtonäppäintä. Perinteinen tapa olisi ollut, että rivinvaihdon huomaamisen jälkeen olisi uusi nimi luettu suoraan tekstikentästä ja lisätty se “List of Names” -tunnisteen alla olevaan listaan.

MVC-mallin mukaisesti toimittaessa uusi nimi luetaan rivinvaihdon painamisen jälkeen tekstikentästä, jonka jälkeen päivitetään mallina toimivaa ei-graafista listaa listOfNames. Ikkunassa näkyvää listaa päivitetään vasta mallin muuttuessa.

Swingin ensimmäisten prototyyppien jälkeen huomattiin, että perinteinen MVC-jako ei toiminut kovin hyvin. Tämä johtui siitä, että geneeristä ohjainosaa oli erittäin vaikea toteuttaa ja näkymä- ja ohjainosien piti olla läheisesti sidoksissa toisiinsa. Näkymä- ja ohjainosat päätettiin yhdistää, jolloin syntyi käsite edustaja (UI delegate).

Swing-kirjastossa näkymä ja ohjainosat yhdistetään yhdeksi komponentin sisäiseksi käyttöliittymäobjektiksi. Malli on edelleen erillinen, kuten alkuperäisessä MVC-mallissa. Kuva 18 selventää Swing-komponentin rakennetta [19].



Kuva 18. Swing-komponentin arkkitehtuuri.

3.5 Ohjelmistotyökalujen arviointia

Java-ohjelmistokehitykseen riittävät yksinkertainen tekstieditori ja JDK-kehityspaketin mukana tuleva javac-kääntäjä. Projektien hallinta on kuitenkin helpompaa erityisillä kehitystyökaluilla, joita on saatavilla useita. Nämä työkalut ovat kehittyneet selvästi ensimmäisistä versioista, mutta siltikään ne eivät ole vielä esimerkiksi C++-työkalujen tasoisia. Ohjelmistotyökalujen kehittelijöillä on ollut kiire kehittää Javan kehitystyökaluja, ja se näkyy niiden laadussa.

Markkinoilla on tällä hetkellä ainakin seuraavat työkalut:

- Borland JBuilder/JBuilder2
- Cosmo Code
- IBM VisualAge for Java™
- Metrowerks CodeWarrior
- Microsoft VisualJ++
- Sun Java™ WorkShop™
- SuperCede
- Sybase PowerJ
- Symantec Visual Cafe
- Together J
- Visix Vibe.

Näistä testattiin työn aikana Borlandin JBuilder 1- ja 2-versioita, Microsoft VisualJ++:aa sekä Symantecin Visual Cafe-työkalua. Ohjelmistokehitys tehtiin Windows NT 4.0 -ympäristössä.

Microsoftin VisualJ++ jäi pois sen takia, että sillä käännetty tavukoodi ei ollut sataprosenttisesti Java-yhteensopivaa. Borlandin JBuilder ja Symantecin Visual Cafe olivat molemmat käyttökelpoisia, mutta suurin osa kehityksestä tehtiin JBuilderilla.

Sekä JBuilderin että Visual Cafen avulla on mahdollista kehittää käyttöliittymiä graafisen työkalun avulla. Tässä on kuitenkin ongelmana se, että molempien työkalujen graafiset editorit käyttävät koodia tuottaessaan omia luokkiaan. Tämä aiheuttaa yhteensopivuusongelmia, jos työkalua on jostain syystä vaihdettava kesken projektin. Tämän takia kaikki koodi tuotettiin käsin, ja näin saatiin sataprosenttisesti yhteensopivaa Javakoodia.

JBuilderista jäi kaiken kaikkiaan hieman keskeneräinen vaikutelma. Ohjelma tuntui todella raskaalta, kun koneessa oli 64 megatavun keskusmuisti. Kaikista eniten haittasivat toistuvat ohjelman kaatumiset. Kun keskusmuistin määrää lisättiin 192 megatavuun, JBuilder tuntui vakaammalta. Sekään ei kuitenkaan auttanut täysin, ja ohjelman kaatumiseen oli vain totuttava. Kun tehdyt muutokset tallennettiin tarpeeksi usein, JBuilder oli kuitenkin aivan kelvollinen väline koodin tuottamiseen.

4. Testitapauseditorin suunnittelu ja toteutus

Tässä luvussa kuvataan testitapauseditorin suunnittelun ja toteutuksen eri vaiheet ja tekniset ratkaisut. Alussa käydään myös läpi ympäristöä, johon testitapauseditori pääasiassa suunniteltiin.

4.1 MOSIM-testausympäristö

Tässä kappaleessa esitetään aluksi yleisiä periaatteita simulointipohjaisesta testauksesta, ja tarkastellaan hieman MOSIM-testaustyökalun arkkitehtuuria [6]. Lopuksi esitetään MOSIMissa käytettävän testiaineiston rakenne. Kappaleessa ei ole tarkoitus käsitellä kovin syvällisesti simulointipohjaisen testauksen periaatteita ja MOSIMin arkkitehtuuria, vaan keskittyä työssä tarvittavaan tietoon testitapauseditorin kohdeympäristöstä.

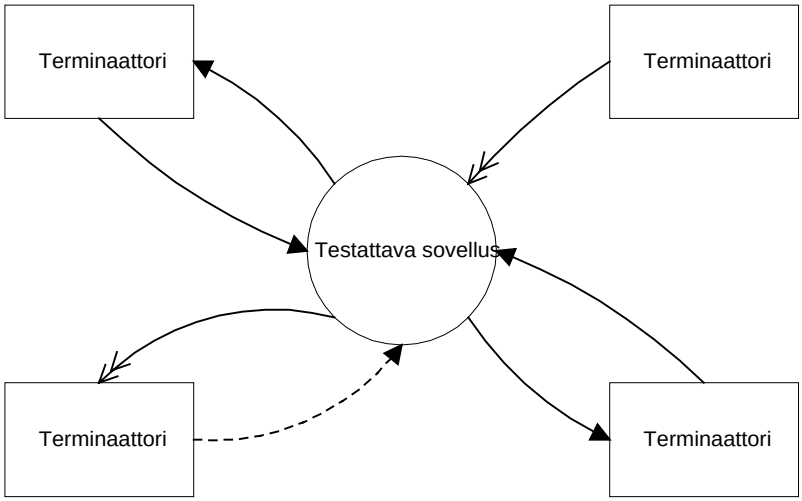
4.1.1 Simulointipohjainen testaus

Sulautettujen järjestelmien kehityksessä on yleistä, että ohjelmiston ja kohdelaitteen kehitysaikataulut ovat erilaisia. Tuote on kuitenkin saatava markkinoille mahdollisimman nopeasti, jolloin ajaudutaan helposti tilanteeseen, jossa kehitettyä ohjelmistoa ei ehditä testata tarpeeksi lopullisessa kohdelaitteessa.

Tämän takia on yleisesti otettu käyttöön erilaisia simulointiin pohjautuvia testausympäristöjä. Näille kaikille on yhteistä se, että niiden avulla voidaan kohdeympäristön toimintaa simuloida ja saada suuri osa tarvittavasta ohjelmistotestauksesta tehtyä jo ennen kehitettävän laitteen valmistumista.

Sulautetuista järjestelmistä voidaan erottaa kolme osaa testauksen näkökulmasta: testattava ohjelmisto (SUT), ympäristö ja rajapinta niiden välillä. Kommunikointi testattavan ohjelmiston ja ympäristön välillä tapahtuu testitapahtumien kautta. Ne ovat viestejä, jotka voidaan erottaa ympäristön tuottamiin herätteisiin ja testattavan ohjelmiston tuottamiin vasteisiin.

Ympäristö voi muodostua erilaisista toiminnallisista kokonaisuuksista, riippuen sovellusalueesta ja kehitettävästä järjestelmästä. Koko ympäristön esittäminen yhdessä osassa ei siis yleensä ole järkevää, eli ympäristö on jaettava toiminnallisiin kokonaisuuksiin. Jako voidaan tehdä esimerkiksi käyttämällä SART-menetelmää [23].



Kuva 19. SART-kaavio.

SART-kaaviossa ympäristö on jaettu komponentteihin, joita kutsutaan terminaattoreiksi. Testattava sovellus on esitetty kaavion keskellä, ja se on vuorovaikutuksessa terminaattoreiden kanssa. Terminaattorit eivät voi kommunikoida suoraan keskenään. Vuorovaikutusta terminaattoreiden ja testattavan sovelluksen välillä kuvataan vuolla. Ohjausvuot merkitään katkoviivalla ja tietovuot kiinteällä viivalla, kuva 19.

Sulautettujen järjestelmien testattava ohjelmisto voi koostua esimerkiksi sovelluksen reaaliaikatehtävistä, keskeytyskäsittelijöistä, ja käyttöjärjestelmän muodostamasta paketista. Ympäristö riippuu hyvin paljon sovellusalueesta, mutta joitain yhtäläisyyksiä on kuitenkin nähtävissä.

Nykyään on yleistä, että sulautettu ohjelmisto kehitetään ensin esimerkiksi työasemassa ja siirretään sitten siihen laitteeseen, jossa sen on tarkoitus toimia. Tässä työssä näitä kutsutaan kehitysympäristöksi ja kohdelaitteeksi.

Kohdelaitteen simulointi voidaan jakaa osiin, jolloin kohdelaitteen käyttöjärjestelmää, I/O-voita, terminaattoreita ja ajoitusta simuloidaan erikseen.

4.1.2 Kohdelaitteen käyttöjärjestelmän simulointi

Kohdelaitteen käyttöjärjestelmää voidaan ajaa kehitysympäristössä käyttämällä käskytason simulaattoreita (ILS) ja ns. ristikääntäjiä (cross-compilers). Molemmista on saatavilla kaupallisia toteutuksia.

Käskytason simulaattorissa simulointi toteutetaan ajamalla kohdelaitteen konekielistä koodia kehitysympäristössä. Näin voidaan kohdelaitteen käyttöjärjestelmää jäljitellä helposti, tekemättä käyttöjärjestelmään mitään muutoksia. Käyttäytyminen on hyvin samanlaista kuin lopullisessa kohteessa. Käskytason simulaattori on käytännöllinen yksikkötestauksessa, mutta hitaus haittaa sen käyttöä integrointi- ja järjestelmätestauksessa [24].

Kun käskytason simulaattori ei sovellu käytettäväksi, voidaan käyttöjärjestelmä muuntaa toimimaan suoraan kehitysympäristössä. Jos suuri osa käyttöjärjestelmästä on koodattu ANSI-C:llä, sen muuntaminen voidaan suorittaa suhteellisen helposti.

4.1.3 I/O-voiden simulointi

Testattava sovellus ja terminaattorit ovat keskenään vuorovaikutuksessa I/O-voiden välityksellä. Kohdelaitteessa I/O-voiden käsittely hoidetaan erityisten operaatioiden ja keskeytysten avulla, jotka on koodattu laiteajureiksi ja keskeytyskäsittelijöiksi. Koska kehitysympäristössä ei ole näitä komponentteja, niiden toimintaa täytyy jäljitellä.

I/O-simuloinnin toteuttamisessa voidaan käyttää kolmea tapaa. On mahdollista rakentaa kohdelaitteelle spesifinen I/O-simulointiyksikkö, jolloin simuloinnista saadaan mahdollisimman samanlainen kohdelaitteen kanssa. Haittana on I/O-simulointiyksikön rakentamiseen kuluva aika, joka on usein liian pitkä ohjelmistokehityksen viemään aikaan verrattuna.

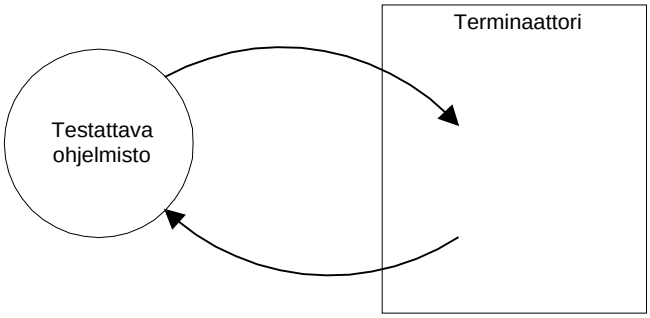
Toinen mahdollisuus on yleiskäyttöisen kaupallisen lisälaitteen käyttö. Yleiskäyttöisyyden takia simulointiyksikkö ei välttämättä sisällä kaikkia tarpeellisia toimintoja I/O-operaatioiden jäljittelyyn. Puuttuvien toimintojen simulointi joudutaan tässä tapauksessa tekemään ohjelmallisesti.

Kolmas mahdollisuus on simuloida I/O:ta ohjelmallisesti. Tavallinen kehitysympäristönä toimiva työasema riittää simuloinnin tekemiseen ja mitään erityisiä lisälaitteita ei tarvita.

4.1.4 Terminaattoreiden simulointi

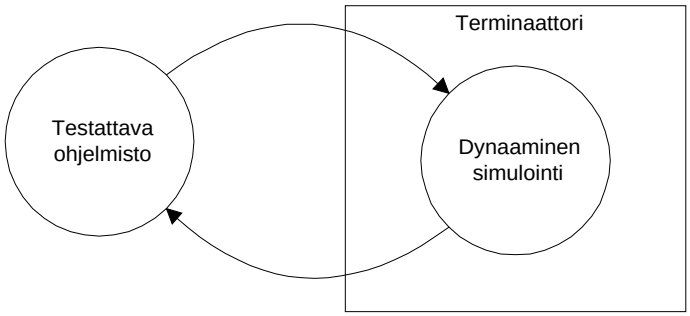
Ympäristö esitetään testauksen yhteydessä terminaattoreina. Terminaattoreiden lukumäärä ja tyyppi riippuvat täysin sovelluksesta. Koska terminaattoreiden rakenteesta tulee helposti liian monimutkaisia, joudutaan usein karsimaan pois epäoleelliset osat ja ottamaan mukaan simulointiin vain testauksen kannalta tärkeimmät toiminnot.

Ympäristön eli terminaattoreiden jäljittelyssä voidaan käyttää lähestymistapaa, joka perustuu herätteisiin ja vasteisiin. Se voidaan jakaa kahteen tyyppiin: avoimen silmukan ja suljetun silmukan simulointiin.



Kuva 20. Avoimen silmukan simulointi.

Avoimen silmukan simuloinnissa (kuva 20) heräte voidaan luoda interaktiivisesti käyttäjän toimesta esimerkiksi työaseman näppäimistöltä ja vaste lukea työaseman näytöltä. Herätteen luomisessa ei välttämättä tarvita interaktiivista vuorovaikutusta käyttäjän kanssa, vaan heräte voidaan lähettää testattavalle ohjelmistolle ennalta määrättyinä ajanhetkinä. Heräte luetaan esimerkiksi tiedostosta. Myös testattavan ohjelmiston tuottama vaste voidaan tallentaa tiedostoon myöhempää käyttöä varten.



Kuva 21. Suljetun silmukan simulointi.

Suljetun silmukan simuloinnissa (kuva 21) testattavan sovelluksen vasteet otetaan vastaan terminaattorin mallissa, joka tuottaa saamiensa vasteiden perusteella uudet herätteet testattavalle sovellukselle. Näin saadaan syntymään suljettu silmukka. Simulointifunktio tarkoittaa yhtä tällaista heräte-vaste-paria.

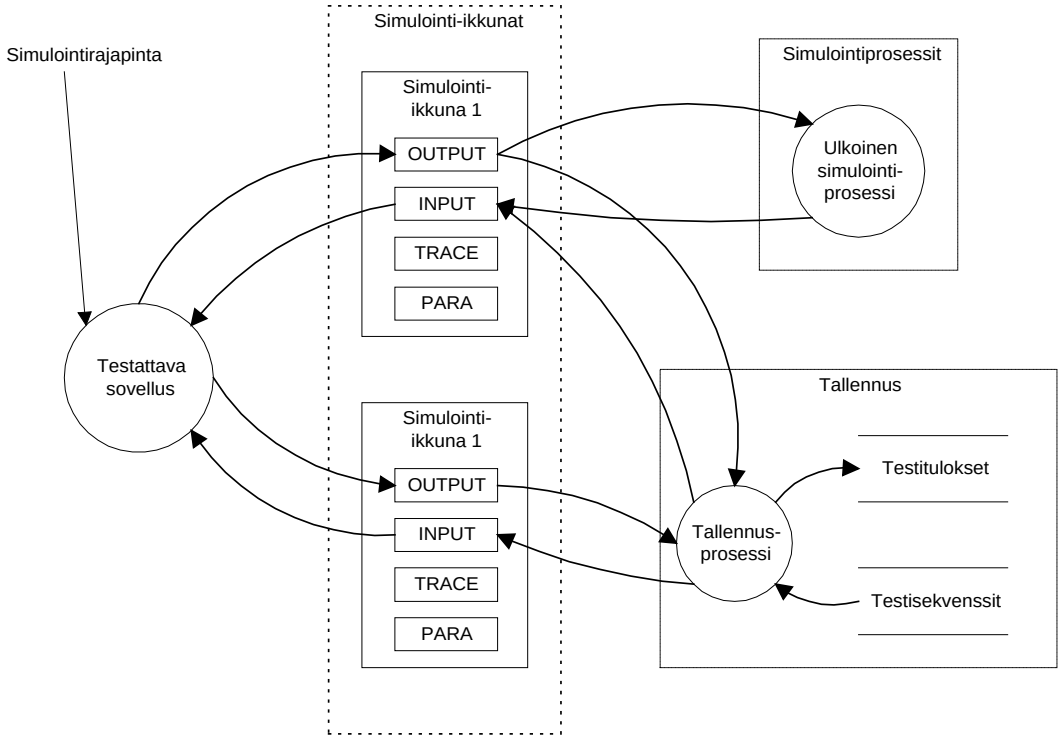
4.1.5 Ajoituksen simulointi

Tärkeimpiä ajoitukseen liittyviä käsitteitä ovat testattavan ohjelmiston sisäinen ajoitus, herätteiden ajoitus ja vasteiden ajoitus. Näistä herätteiden ajoitus on tärkeä osa testitapausten luontia ja vasteiden ajoitus on kriittistä testitulosten tarkastelun kannalta.

4.1.6 MOSIMin arkkitehtuuri

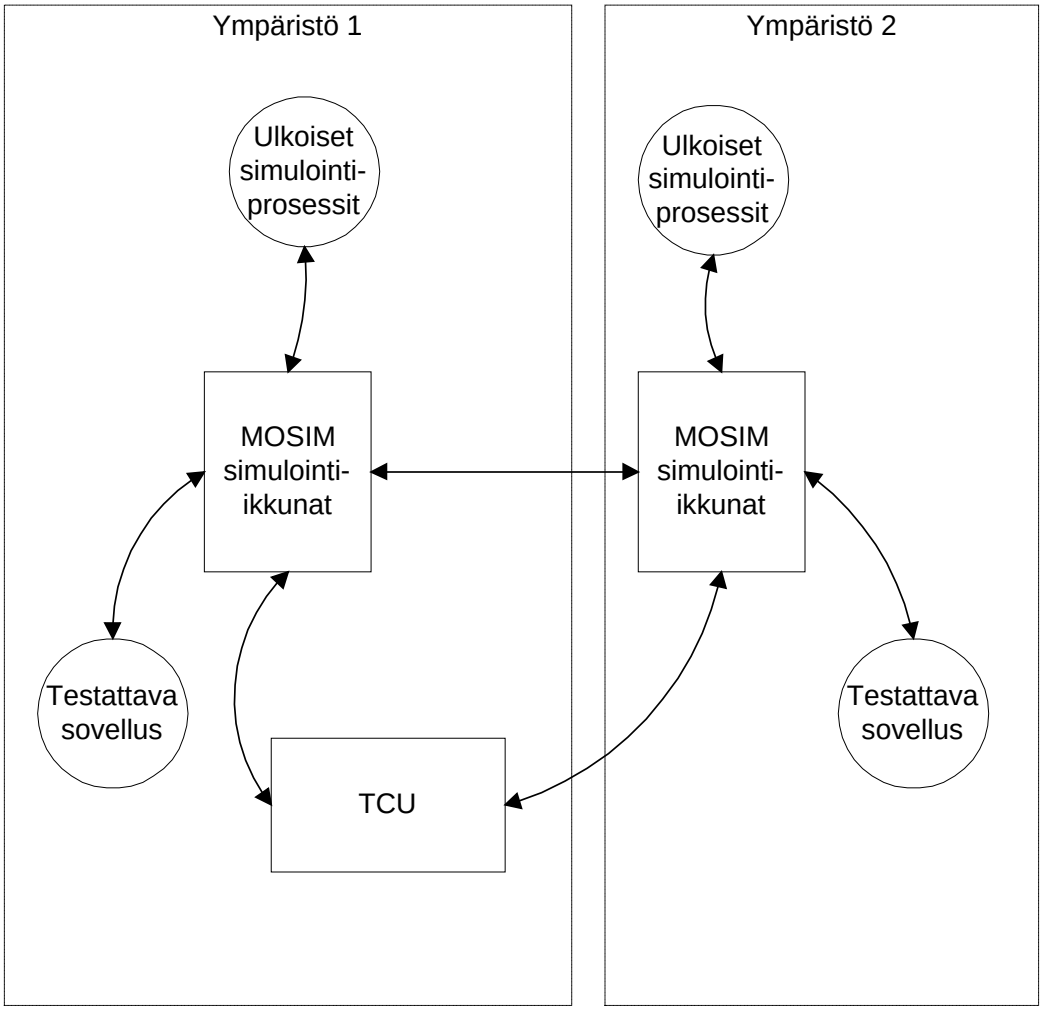
MOSIM-työkalu on kehitetty toteuttamaan edellisessä kappaleessa käsitellyn simulointipohjaisen testauksen periaatteita. Arkkitehtuurissa on käytetty SART-menetelmän käsitteitä.

Testausympäristö muodostuu yhdestä tai useammasta simulointi-ikkunasta, simulointiprosesseista, simulointirajapinnasta ja testisekvenssien tallennukseen ja toistamiseen tarkoitetusta tallennusosasta. Kuvassa 22 [6] on esitetty MOSIM-testausympäristön arkkitehtuuri.



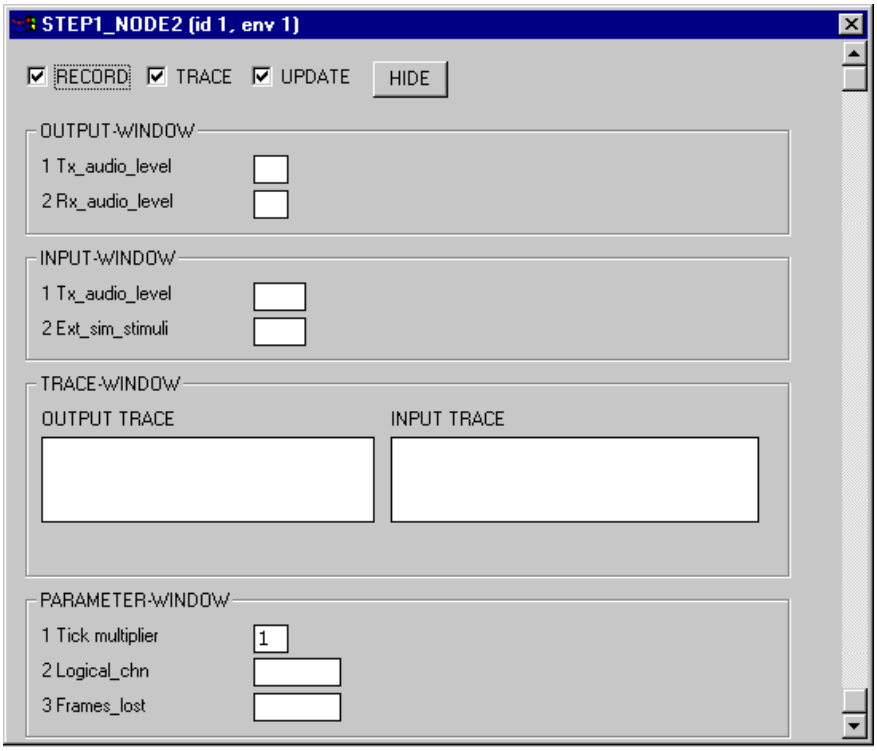
Kuva 22. MOSIM-arkkitehtuuri.

Testausympäristöt ovat usein erittäin monimutkaisia ja ne joudutaan rakentamaan uudelleen jokaiselle testattavalle ohjelmistolle. MOSIM soveltuu myös hajautettujen järjestelmien testaukseen, jolloin kaksi tai useampi MOSIM-testausympäristöä voidaan liittää yhteen ja ajaa niitä erillisissä työasemissa. Kuva 23 [25] esittää kahden erillisen MOSIM-ympäristön yhteenliittämistä.



Kuva 23. MOSIM-ympäristöjen liittäminen yhteen.

Simulointirajapinnan tehtävänä on jäljitellä kohdelaitteen keskeytyksiä ja I/O-operaatioita kehitysympäristössä. Rajapinta kytkee testattavan ohjelmiston yhteen ympäristöä mallintavien simulointi-ikkunoiden kanssa. Käytännössä rajapinta koostuu kahdesta erillisestä moduulista, MOSIM I/O-funktiokirjastosta ja konfiguraatiomoduulista.

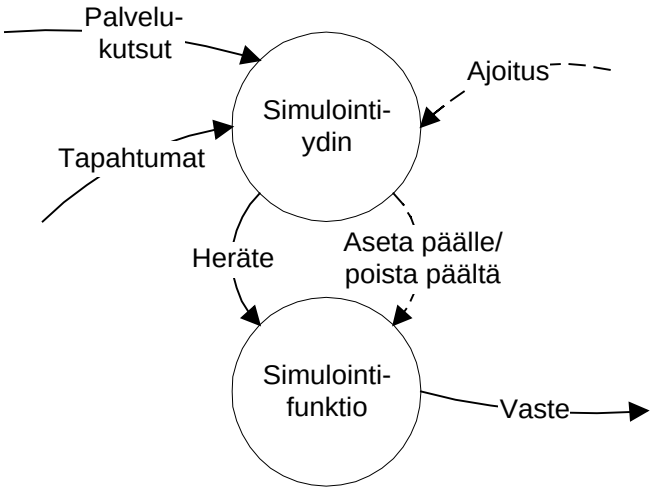


Kuva 24. Simulointi-ikkuna.

Simulointi-ikkunoissa näkyvät sekä testattavan sovelluksen tuottamat vasteet että testattavalle sovellukselle lähetetyt herätteet, kuva 24. Ikkuna on jaettu neljään osaan: output, input, trace ja parameter. Jokainen osio on jaettu kenttiin ja kenttä on yksittäisen tapahtuman esityksen perusosa. Testaajalla on mahdollisuus syöttää herätteitä testattavalle sovellukselle input-osion kautta, ja testattavan sovelluksen lähettämät vasteet näkyvät output-osassa. Trace-osio sisältää historiatietoja ja parametri-osassa voidaan määritellä erilaisia parametrien arvoja.

Tallennusosan tehtävänä on tallentaa suoritettujen testien tulokset ja mahdollistaa testisekvenssien automaattinen ajaminen. Pitkiä testisekvenssejä ei tarvitse ajaa aina uudelleen käsin, vaan kerran ajettu sekvenssi voidaan ajaa uudelleen automaattisesti.

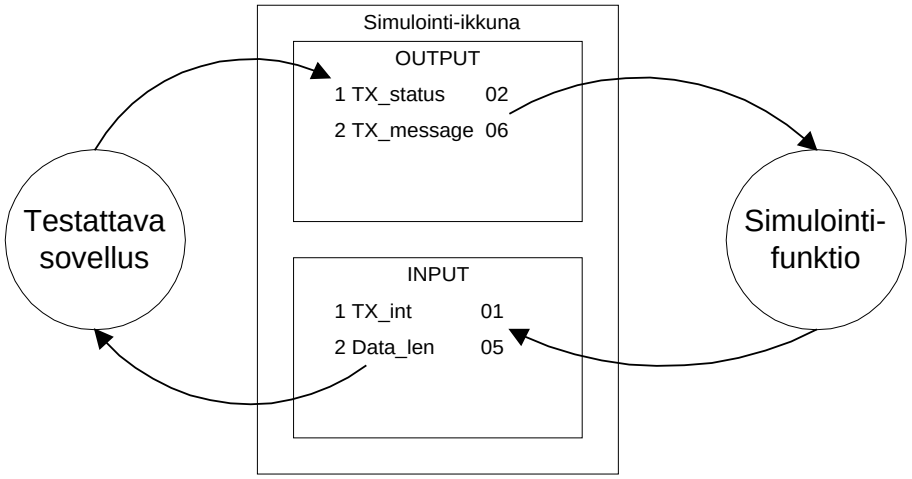
Testitulokset tallennetaan yhteen tiedostoon. Vasteiden lisäksi on mahdollista tallentaa myös herätteet ja simulointiparametrien muutokset. Kaikkiin tapahtumiin liitetään aikaleima ja juokseva numero, jotta tapahtumat voitaisiin pitää aikajärjestyksessä. Käyttäjä voi myös valita vain kiinnostavimpien asioiden tallennuksen.



Kuva 25. Simulointifunktio.

Simulointifunktion tehtävänä on muodostaa uusi heräte testattavalta sovellukselta saamansa vasteen pohjalta, kuva 25. Funktiot ovat yleensä pieniä, testaajan kirjoittamia C-kielisiä funktioita. Simulointiydin lähettää funktioille herätteitä, joihin ne tuottavat vasteen.

Kuvassa 26 on esitetty testattavan ohjelmiston, simulointi-ikkunan ja simulointifunktioiden suhde toisiinsa. [6]



Kuva 26. Simulointifunktion toiminta.

Simulointiprosessit ottavat vastaan vasteita ja tuottavat niiden perusteella uusia herätteitä. Prosessit ovat kuitenkin suurempia kokonaisuuksia kuin funktiot. Esimerkkinä simulointiprosessista on graafinen käyttöliittymä, joka on liitetty testausympäristöön.

4.2 MOSIM-testausympäristössä käytettävän testiaineiston rakenne

4.2.1 Yksinkertainen testitapaus

MOSIM-testausympäristöön on kehitetty erityinen kuvauskieli, jonka avulla on mahdollista suunnitella ja tallentaa tiedostoon ajettava testitapaus. Tämä testitapaus voidaan ajaa Test Controller -moduulin avulla. Testitapauksen ajo voidaan tallentaa myös tiedostoon samassa muodossa.

Tiedoston nimi on vapaasti määriteltävissä. Tiedosto on ASCII-tyyppinen tekstitiedosto, jonka rakenne on määritelty tietyntyyppiseksi. Yhdellä rivillä on esitetty korkeintaan yksi testiheräte, kommenttirivien käyttö on mahdollista. [25]

Normaali testiheräte on seuraavanlainen:

>ENV 1 WIN 1 ID 213 TIME 1 chan 1

Yllä oleva rivi tarkoittaa, että tieto (1) lähetetään MOSIM-ympäristössä 1 olevan ikkunan numero 1 kenttään numero 213. Lähetys tapahtuu 1 sekunnin kuluttua, koska TIME parametrilla on arvo 1.

Rivi koostuu seuraavista osista:

ENV 1	MOSIM-testausympäristön numero
WIN 1	Ikkunan numero
ID 213	Kentän tunniste
TIME 1	Ajoitus sekunteina
chan	Kentän nimi
1	Lähetettävä tieto.

Toinen esimerkki on hieman erilainen. Uutena kenttänä on TICKS 10, joka määrää, että ennen tiedon (010005) lähetystä odotetaan 10 kellojaksoa. TICKS-parametria käytetään pääasiassa synkronisessa ajoitusmoodissa.

>ENV 1 WIN 1 ID 214 TICKS 10 M_Chan 010005

Kuvauskieli mahdollistaa myös ns. tulkattujen herätteiden käytön:

>ENV 1 WIN 1 ID 214 TIME 0 M_Chan M_CH_MSG:SCCH TRUE ON 30 24

4.2.2 Muuttujat ja ehdollinen haarautuminen

MOSIM-kuvauskieli tukee muuttujien ja ehdollisen haarautumisen käyttöä (if-then-else). [26]

Muuttujat tulee määritellä ennen käyttöä. Muuttujat ovat voimassa vain siinä INREC-tiedostossa, jossa ne on määritelty. Seuraavassa on esimerkki muuttujan määrittelystä:

< VARIABLE <variable_name> [= <value>]

Muuttuja voidaan alustaa määrittelyn yhteydessä halutulla arvolla. Muuttujan arvo voidaan asettaa myös myöhemmin SET-käskyllä.

Ehdollinen haarautuminen perustuu IF-THEN-ELSE -rakenteiden käyttöön. Rakenne alkaa aina IF-lauseella ja viimeisenä tulee olla ENDIF, joka päättää rakenteen:

< IF <if_expr>

  ...INREC statements

[< ELSE IF <if_expr>]

  [...INREC statements]

[< ELSE]

  [...INREC statements]

< ENDIF .

Ehdollinen haarautuminen on siis toteutettu samoin kuin useimmissa ohjelmointikielissä. ELSE-IF-ehtoja voi olla useita ja ennen rakenteen loppua voi olla yksi ELSE-ehto. Rakenteessa <if_expr> on boolean-tyyppinen lauseke.

4.3 Testitapauseditorin tarkoitus

MOSIM-testausympäristössä suurin osa prosessien välisestä kommunikaatiosta perustuu sanomien käyttöön. Testauksessa lähetetään erityyppisiä viestejä testattavalle sovellukselle ja kuunnellaan sovelluksen antamia vasteita.

Sanomarakenteet voivat olla monimutkaisia erityisesti tietoliikennejärjestelmiä testattaessa. Sanomien luonti on vaikeaa, koska ne ovat tavallisesti heksadesimaalisia numeroita ja siten ihmiselle erittäin vaikeasti ymmärrettäviä. Pitkien heksadesimaalisten viestien kirjoittamisessa tulee myös helposti virheitä.

Työn tavoitteena oli kehittää testitapauseditori, jonka avulla sanomien luonti olisi helpompaa ja veisi vähemmän aikaa. Koko testiaineiston tuli olla helposti hallittavissa ja työkalussa piti olla mahdollisuus luoda valmispohjia, joita voitaisiin myöhemmin käyttää uusien viestien pohjana.

Käyttöliittymästä pyrittiin tekemään Windows-tyyppinen, koska pääasiallinen käyttöympäristö tulee olemaan Windows NT. Käytännössä tämä tarkoittaa sitä, että testitapauseditorin valikot, työkalupalkki, ikkunat, ja muut komponentit näyttävät ja käyttäytyvät samalla tavoin kuin muissakin Windows-sovelluksissa.

Seuraavissa kappaleissa käsitellään tarkemmin testitapauseditorille asetettuja vaatimuksia ja sitä, miten ne toteutettiin.

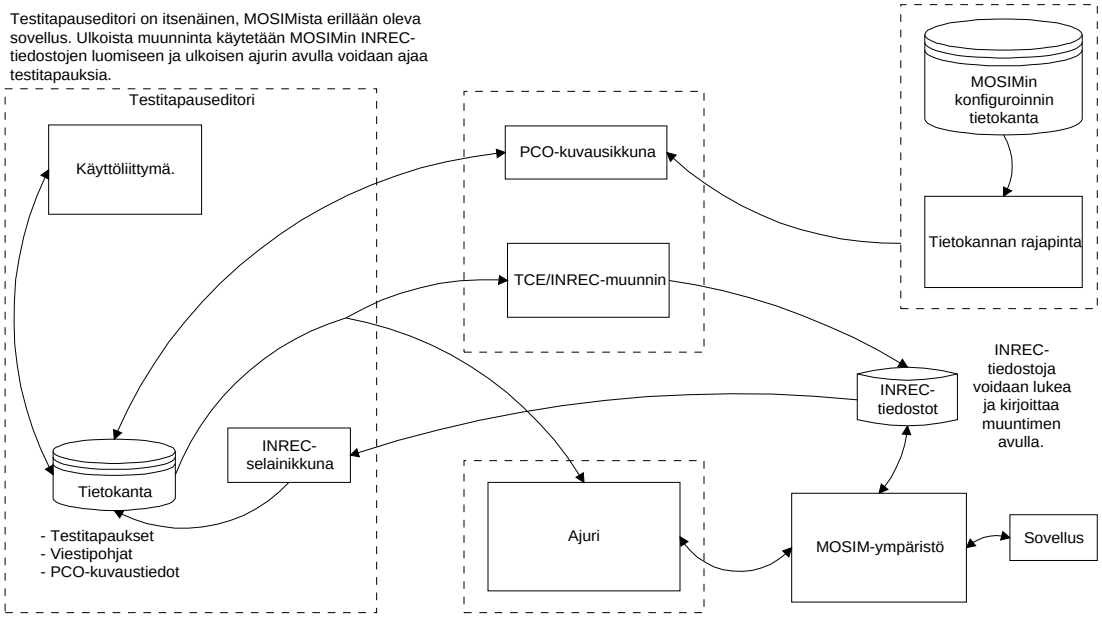
4.4 Vaatimukset

Editoriosan piti sisältää seuraavat ominaisuudet, kuva 27:

- Puumainen esitystapa testisekvenssille pääikkunan vasemmassa reunassa
- Puusta valitun testitapauksen näyttäminen ja editoiminen pääikkunan oikeassa puoliskossa
- Sanomapohjien editointi-ikkuna
- Sanomien editointi-ikkuna
- Yksinkertaisten viestien editointi-ikkuna
- PCO-tietojen kuvaaminen MOSIMin ympäristöihin, ikkunoihin ja kenttiin.

Tietokanta tallennettiin yhteen tiedostoon, jonka tuli sisältää seuraavat tiedot:

- Luodut testitapaukset
- Sanomien pohjat
- MOSIMin PCO-kuvaustiedot.



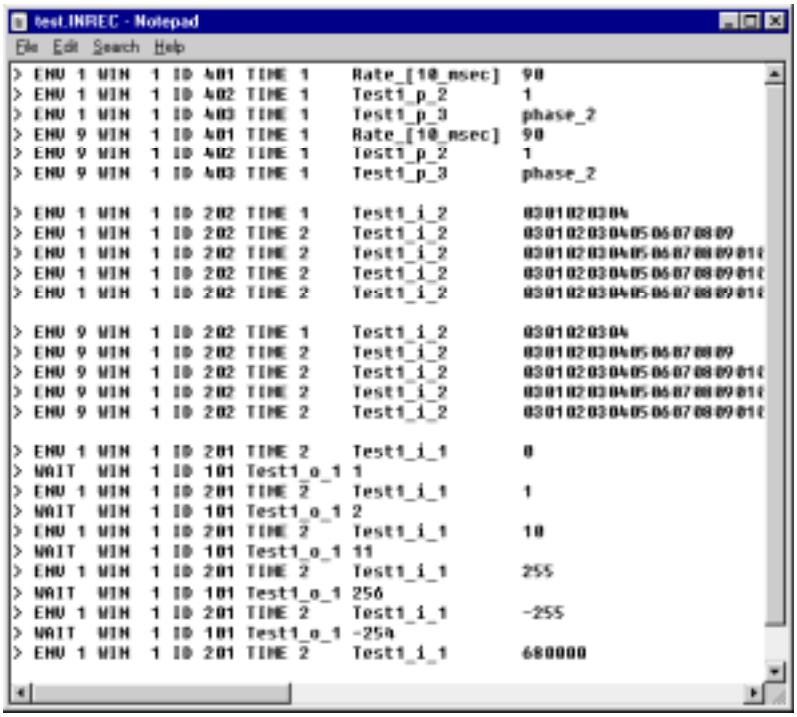
Kuva 27. Testitapauseditorin arkkitehtuuri.

4.5 Testitapauseditorin käyttöliittymän suunnittelu ja toteutus

Käyttöliittymän suunnittelussa käytettiin hyväksi työn alussa esitettyjä periaatteita. Prosessimallina käytettiin prototypointia. Tässä kohdassa käydään läpi testitapauseditorin suunnittelu ja toteutus.

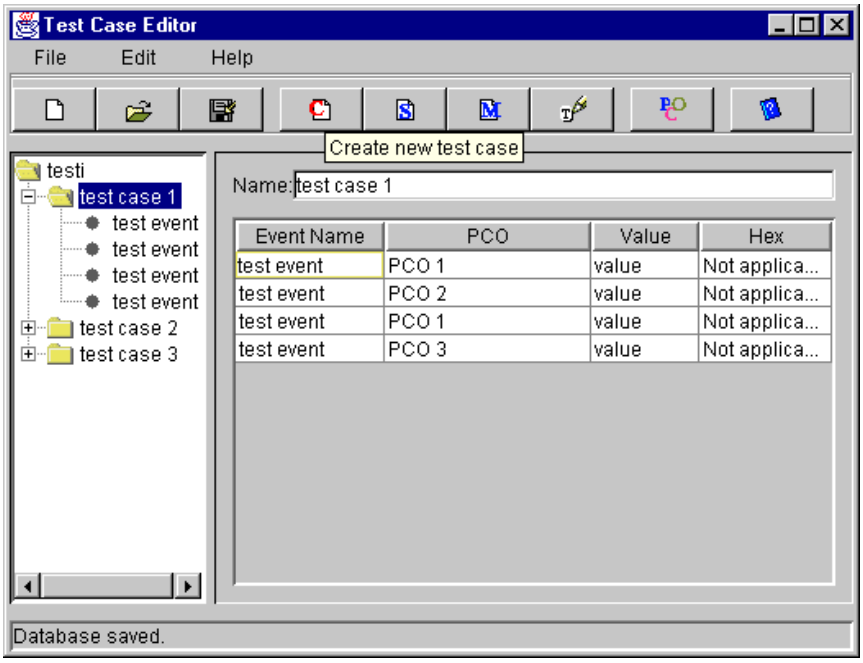
4.5.1 Käyttöliittymän toteutus JFC:n avulla

Aikaisemmin MOSIMin testitapauksia tehtiin editoimalla niitä suoraan tekstitiedostoon, kuva 28. Uusi käyttöliittymä (kuva 29) tarjoaa testitapausten luontiin selkeämmän graafisen lähestymistavan.



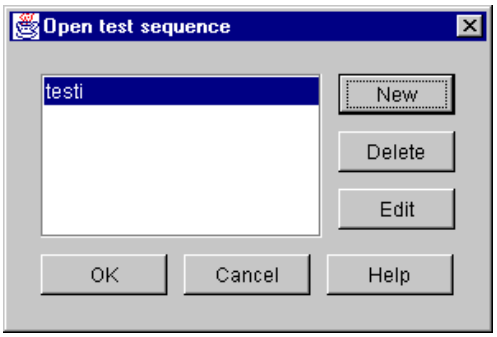
Kuva 28. MOSIMin testitapauksen editointi.

Testitapauseditorin käyttöliittymän toteutukseen käytettiin JFC-luokkakirjastoa, ja siitä erityisesti Swing-komponentteja. Layout Managerina käytettiin BorderLayoutia, jonka avulla sijoitettiin työkalupalkki ylös ja tilapalkki alas. Keskelle sijoitettiin luokasta JSplitPane periytetty splitPanel, joka jaettiin horisontaalisesti kahtia puuta ja taulukkoa varten.



Kuva 29. Testitapauseditorin pääikkuna.

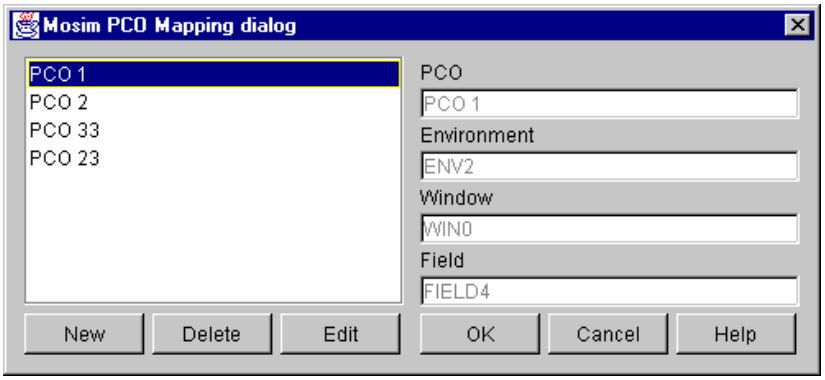
Testiaineisto esitetään puuna, joka muodostuu testitapauksista ja testitapahtumista. Testiaineiston esittävä puu toteutettiin periyttämällä oma luokka luokasta JTree. Kerralla on näkyvissä vain yksi testiaineisto. Valikosta kohdasta File|Open test sequence saadaan esille kuvassa 30 näkyvä dialogi, josta voidaan valita muokattava testiaineisto. Myös uuden testiaineiston luonti on mahdollista.



Kuva 30. Muokattavan testiaineiston valinta.

Valittaessa testiaineistoa esittävästä puusta testitapaus näytetään sen sisältö taulukkona pääikkunan oikeassa reunassa. Taulukon yksi rivi vastaa yhtä testitapahtumaa.

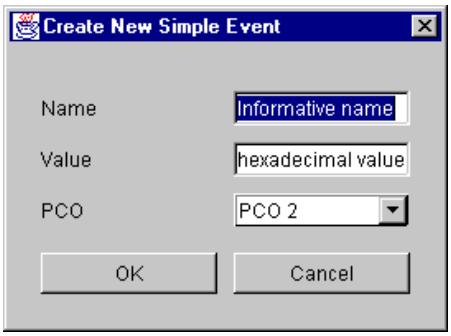
Jokainen testitapahtuma liitetään johonkin PCO:hon (Point of Control and Observation). MOSIM ei tunne käsitettä PCO, joten jokainen PCO pitää liittää MOSIMin ymmärtämään muotoon, kuva 31. PCO:lle annetaan nimi ja MOSIMissa käytettävät ympäristö, ikkuna ja kenttä. Tiedot tallennetaan testitapauseditorin tietokantaan.



Kuva 31. PCO:n liittäminen MOSIM-ympäristöön.

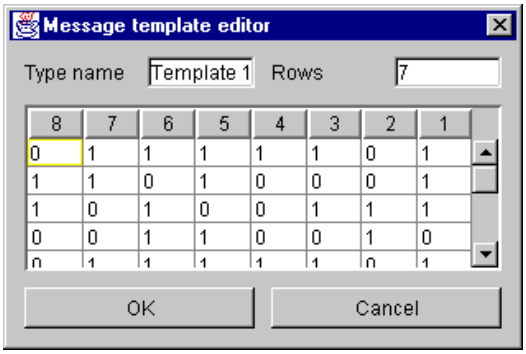
Uusi testitapaus lisätään testiaineistoon valitsemalla valikosta File|New test sequence tai painamalla työkalupalkissa olevaa painonappia. Olemassa oleva testitapaus voidaan kopioida ja liittää testiaineistoon. Luotaessa testitapaukselle annetaan informatiivinen nimi, jonka jälkeen testitapausta voidaan editoida.

On olemassa useita erityyppisiä testitapahtumia. Uusi testitapahtuma lisätään valittuun testitapaukseen valikosta tai painamalla työkalupalkissa olevaa vastaavaa painonappia. Tällöin aukeaa valitun tyyppisen testitapahtuman lisäykseen tarkoitettu dialogi, kuva 32.



Kuva 32. Uuden testitapahtuman luonti-ikkuna.

Sanomatyyppisiä testitapahtumia varten voidaan tehdä valmiita pohjia, joita voidaan käyttää uusien sanomatyyppisten testitapausten pohjana, kuva 33.



Kuva 33. Viestipohjan luonti.

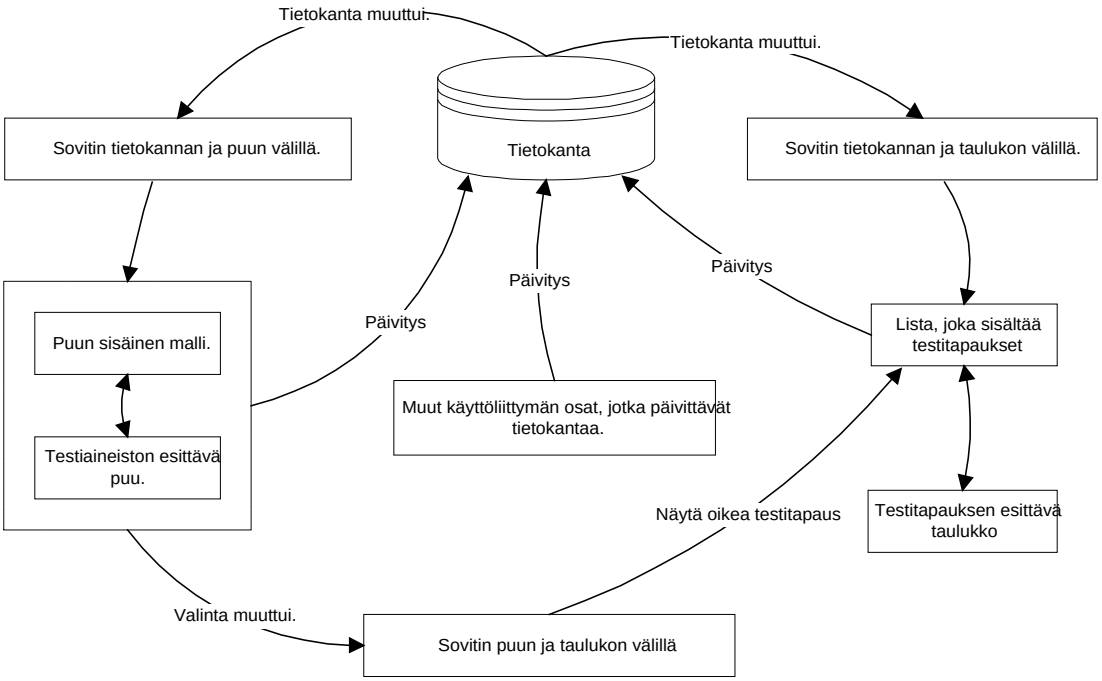
Yksinkertaisia testitapahtumia voidaan muokata suoraan testitapauksen esittävässä taulukossa. Muuntyyppisille testitapahtumille aukeaa kullekin oma editointi-ikkunansa, kun käyttäjä valitsee testitapahtuman muokkauksen.

4.5.2 Tiedonsiirto käyttöliittymän komponenttien välillä

Koska tietokannassa olevia tietoja voidaan muokata sovelluksessa useassa paikassa, tietojen yhtenäisyydestä täytyy huolehtia. Parhaiten tämä onnistuu tekemällä sovelluksesta Swingin modifioidun MVC-arkkitehtuurin mukainen.

Tietojen yhtenäisyys on varmistettu käyttämällä model-view-controller-mallia, jonka pohjalle myös JFC:n komponentit on rakennettu. Erityisen selvästi MVC-mallin käyttö näkyy JFC:n puu- ja taulukkokomponenteissa.

Kuvassa 34 on esitetty testitapauspuun, testitapauksen näyttävän taulukon ja tietokannan välinen yhteys.



Kuva 34. MVC-arkkitehtuuriin pohjautuva tietojen päivitys.

Kuten kuvasta nähdään, arkkitehtuurista muodostuu helposti erittäin monimutkainen MVC-mallia käytettäessä. Testitapauseditorissa rakennetta vaikeuttaa entisestään se, että tietokannan tietojen päivitys on mahdollista sekä puusta että taulukosta käsin. Swingin modifioidun MVC-mallin hyödyt perinteiseen MVC-malliin tulevat hyvin esille tässä tapauksessa.

Kun tietoa muokataan jossain osassa sovellusta, se päivitetään ensin tietokantaan. Tietokantaa kuuntelevat sovittimet saavat tämän jälkeen tiedon muutoksesta ja välittävät sen eteenpäin niille komponenteille, joihin ne on liitetty. Tällä on varmistettu se, että kaikissa sovelluksen osissa on aina käytössä sama tieto.

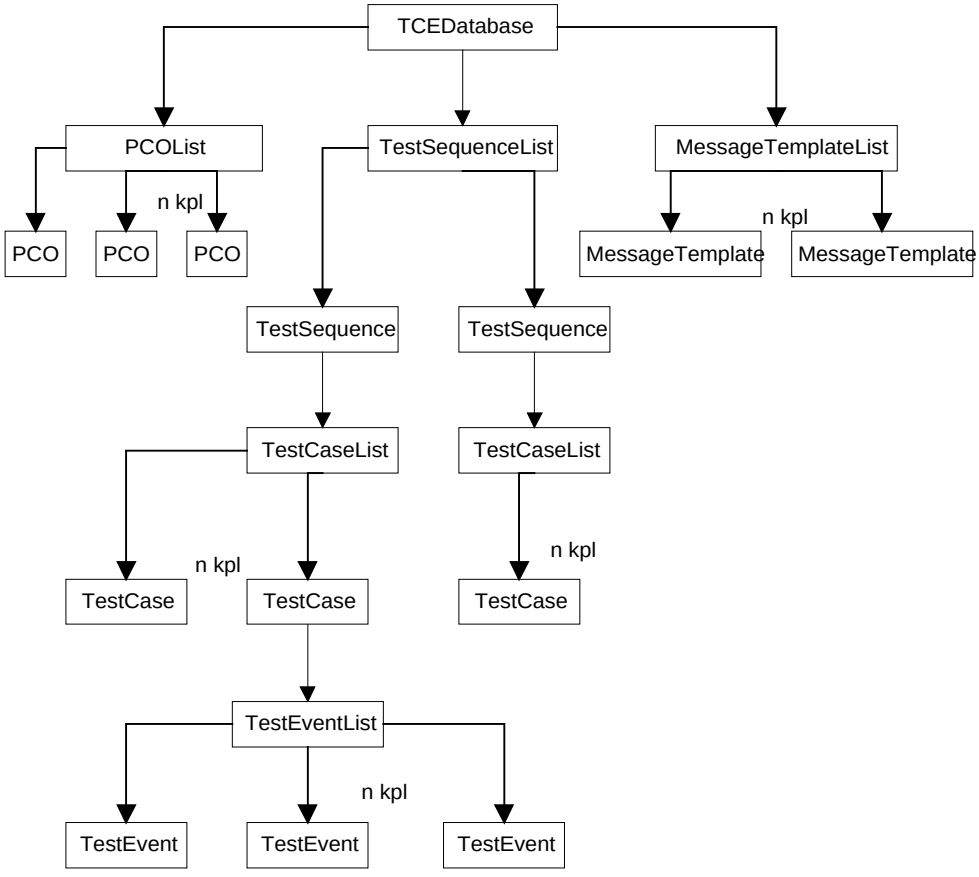
Swing-komponentit voivat toimia sekä ohjain- että näkymäobjekteina. Perinteisessä MVC-mallissa jokainen objekti toimii vain yhdessä roolissa. Edellisessä kuvassa näkyvä SequenceTreeModel on puun sisäinen malli, joten se ei toimi ylimmällä tasolla mallina, kuten tietokantaolio.

4.6 Tietokannan toteutus

Testitapauseditorin tietokanta toteutettiin Javalla. Tietokanta tallennetaan yhteen tiedostoon, joka luetaan muistiin. Tietojen käsittely tehdään muistissa olevaan tietokantaan, ja tallennusvaiheessa koko tietokanta tallennetaan kerralla takaisin tiedostoon.

Tietokanta on rakenteeltaan hierarkkinen, kuva 35. Kuvan luokkien nimet vastaavat toteutuksen luokkien nimiä. Tietokannan sisältämä tieto muodostaa puun, jonka juurisolmuna on luokan TCEDatabase instanssi. Juurisolmu sisältää lapsinaan kolme luokasta Vector periytettyä listaa: PCOList, TestSequenceList ja MessageTemplateList.

Nämä listat voivat sisältää yhden tai useamman esiintymän luokista PCO, TestSequence ja MessageTemplate. Näistä PCO ja MessageTemplate ovat samalla lehtisolmuja. TestSequence haarautuu edelleen pienempiin osiin ja sen lehtisolmuina voivat toimia luokan TestEvent esiintymät.



Kuva 35. Tietokannan rakenne.

4.7 Jatkokehitysmahdollisuudet

Testitapauseditori on tällä hetkellä prototyyppi, eikä se ole vielä tuotantokäytössä. Perustoiminnot ovat kuitenkin valmiina, ja niiden päälle voidaan kehittää uusia ominaisuuksia suhteellisen helposti. Jatkokehityskohteina voisivat olla esimerkiksi seuraavat ominaisuudet:

1. Kuvassa 27 esitetyn ajurin toteuttaminen, joka mahdollistaisi MOSIMin testitapausten suorittamisen suoraan editorista.

2. Testitapausten generointi sanomakaavioiden pohjalta.

3. Automaattinen testitapausten generointi testattavan sovelluksen koodia tutkimalla.

Nykyisin MOSIMin testitapaukset suoritetaan TCU-yksikön avulla. Testitapausten ajamista helpottaisi, jos ne voitaisiin suorittaa suoraan testitapauseditorista. Ominaisuus voitaisiin toteuttaa editorin ja MOSIM-ympäristön välille toteutettavalla ajurilla.

Testitapausten tuottaminen sanomakaavioiden perusteella olisi erittäin hyödyllistä tietoliikenneohjelmistojen testauksessa. Ominaisuuden toteuttaminen vaatisi sanomakaavioeditorin toteuttamisen testitapauseditorin yhteyteen.

Testitapausten automaattinen generointi testattavan sovelluksen koodia tutkimalla voitaisiin toteuttaa lisäämällä testattavan sovelluksen lähdekoodiin testitapauseditorin ymmärtämiä tunnisteita. Se vaikeuttaisi kuitenkin ohjelmistojen toteutusta, ja tulisi kyseeseen vain uusien ohjelmistojen kohdalla. Olisikin parempi, jos ominaisuus voitaisiin toteuttaa ilman mitään erityistunnisteita.

5. Arviointi Javan soveltuvuudesta

Luvun tarkoituksena on arvioida, miten hyvin Java ja JFC-luokkakirjasto soveltuivat testitapauseditorin toteutukseen. Tarkasteltavia asioita ovat muun muassa toteutetun sovelluksen suorituskyky sekä siirrettävyys eri laite- ja käyttöliittymäympäristöjen välillä.

5.1 Siirrettävyys eri ympäristöjen välillä

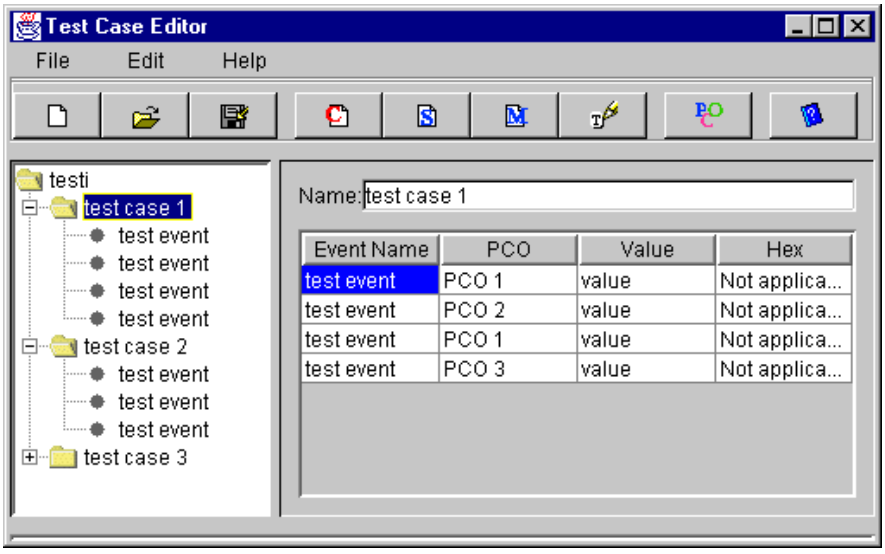
Siirrettävyys eri ympäristöjen välillä on Javan ja sen luokkakirjastojen ehkä tärkein ominaisuus. Javalla tehdyt sovellukset siirtyvät sujuvasti kaikkiin ympäristöihin, joihin on saatavilla virtuaalikone. Testitapauseditorista saadun kokemuksen mukaan siirrettävyys toteutuu myös käytännössä. Suuria ongelmia ei esiinny, kun otetaan huomioon seuraavat seikat:

- Käytetään vain JDK-pakettiin sisältyviä luokkia. Ohjelmointityökalujen mukana tulee usein valmistajien omia luokkakirjastoja, jotka on toteutettu Javan yleisten luokkakirjastojen päälle. Niitä ei kannata käyttää, jos sovellus on tarkoitus siirtää useisiin erilaisiin ympäristöhin.

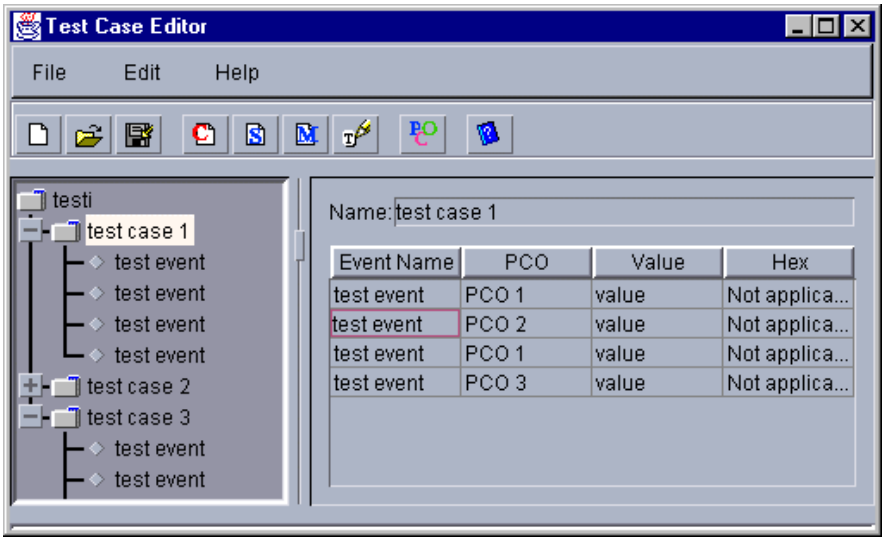
- Ei käytetä komponenttien suoraa sijoitusta ikkunaan. Tämä on mahdollista Javassakin, mutta samalla menetetään osa siirrettävyydestä. Suoran sijoituksen sijaan kannattaa aina käyttää valmiita tai itse tehtyjä Layout Manager -luokkia.

- Siirrettävän sovelluksen ulkonäkö on lähes aina jonkinasteinen kompromissi. On melkein mahdotonta saada sovellus näyttämään täsmälleen samalta jokaisessa ympäristössä. Käyttöliittymästä kannattaa tehdä "väljä", eli komponenttien väliin jättää tyhjää tilaa. Tällä varmistetaan, että kaikki komponentit saadaan näkyviin pienemmälläkin näytöllä tai suurilla järjestelmän fonteilla.

Kuvissa 36 ja 37 esitetään testitapauseditori Windows- ja Motif-tyylisinä. Sunin Solaris-käyttöjärjestelmässä ulkoasu ja toiminta ovat samantyyppiset kuin kuvassa 37. Molemmat kuvat on otettu Windows NT 4.0 -ympäristössä käyttäen JFC:n kytkettävää ulkonäköä ja tuntumaa (Pluggable Look and Feel). Ulkoasut poikkeavat hieman toisistaan. Selkeimmät erot löytyvät värityksestä ja testitapaukset näyttävästä puusta. Myös työkalupalkkien koossa on eroa. Toiminta on kuitenkin samanlainen molemmissa tapauksissa.



Kuva 36. Testitapauseditori Windows-tyylisenä.



Kuva 37. Testitapauseditori Motif-tyylisenä.

5.2 Suorituskykytestit

Suorituskyvyn arvioinnissa ei tehty tarkkoja vertailuja esimerkiksi C++-kieleen verrattuna. Tällaisten vertailujen tekoa varten olisi ollut tarpeen toteuttaa lähes vastaava sovellus myös C++:lla. Vertailua varten toteutettiin erilliset pienet testisovellukset C++:lla ja Javalla. Niistä pyrittiin tekemään mahdollisimman samanlaiset, jotta vertailutuloksista saataisiin luotettavia. Kaikki mittaukset tehtiin Windows NT 4.0 -ympäristössä, kun muita sovelluksia ei ollut käynnissä. Testit suoritettiin 266 MHz:n Pentium II -koneessa, jossa oli 192 megatavua muistia.

Javalla toteutetun graafisen sovelluksen suorituskyky koostuu kahdesta osa-alueesta, kielen yleisestä ja käytetyn luokkakirjaston suorituskyvystä. Tässä yhteydessä edellistä kutsutaan sisäiseksi ja jälkimmäistä käyttöliittymän suorituskyvyksi.

Sisäistä suorituskykyä mitattiin seuraavilla mittareilla:

- Kokonaislukujen yhteen-, vähennys-, kerto- ja jakolaskuun kuluva aika
- Liukulukujen yhteen-, vähennys-, kerto- ja jakolaskuun kuluva aika
- Tiedoston lukuun ja kirjoitukseen kuluva aika.

Sisäisen suorituskyvyn mittauksissa vertailtiin Microsoftin Visual C++ 6.0:lla tehdyn sovelluksen nopeutta JDK 1.1.6:lla tehtyyn vastaavaan sovellukseen. C++-sovellus käännettiin aluksi ilman optimointeja ja sen jälkeen optimoituna. Java-tavukoodi ajettiin JDK 1.1.6-paketin mukana tulleella java.exe-ohjelmalla JIT-kääntäjän (Just In Time Compiler) kanssa ja ilman.

Kokonais- ja liukulukujen laskutoimitusten mittaamista varten tehtiin ohjelma, joka laski silmukassa 10 miljoonaa kertaa jokaisen laskutoimituksen. Laskutoimitukset suoritettiin kahdella operandilla, jotka saatiin taulukoista. Tiedoston lukuun ja kirjoitukseen kuluvaa aikaa mitattiin lukemalla tiedostoa rivi kerrallaan ja kirjoittamalla se toiseen tiedostoon. Luettavan tiedoston koko oli noin 351 kilotavua. Iteraatioiden lukumäärä oli 100. Jokaisessa tapauksessa mitattiin operaatioihin kulunut aika sekunteina. Taulukossa 1 esitetyt mittaustulokset ovat 10 mittauskerran keskiarvoja.

Taulukko 1. Sisäisen suorituskyvyn mittaukset.

Mitattava asia		**C++ (ei optimointeja)**	**C++ (optimoitu)**	**Java (JIT)**	**Java (ei JIT-kääntäjää)**
Kokonaisluvut (32-bittinen **int**)	yhteenlasku	0.691	0.160	0.561	7.481
	vähennyslasku	0.671	0.160	0.581	7.405
	kertolasku	0.685	0.181	0.571	7.406
	jakolasku	1.382	1.462	1.382	8.733
Liukuluvut (32-bittinen **float**)	yhteenlasku	0.671	0.230	0.561	7.711
	vähennyslasku	0.681	0.230	0.591	7.857
	kertolasku	0.671	0.241	0.570	7.781
	jakolasku	1.201	1.185	1.212	8.552
Tiedostojen luku ja kirjoitus		10.976	10.876	36.482	246.541

Käyttäjä kokee sovelluksen suorituskyvyn sen perusteella, miten nopeasti hän saa vasteen suorittamiinsa toimintoihin. Monesti käyttäjä arvioi sovelluksen laatua juuri käyttöliittymän sujuvan toiminnan perusteella. Vaikka viiveet käyttöliittymän toiminnassa eivät olisikaan suuria, ne voivat tehdä sovelluksen käytöstä epämiellyttävää.

Käyttöliittymän suorituskykyyn vaikuttavat mm. ikkunoiden päivitysnopeus, ikkunoiden avaamiseen ja sulkemiseen kuluva aika sekä viive toimintojen suorituksessa. Käyttöliittymän suorituskykyä mitattiin seuraavilla mittareilla:

1) Käyttöliittymäolioiden luontiin ja tuhoamiseen kuluva aika. Mitattiin aikaa, joka kuluu, kun luodaan, näytetään ja tuhotaan tyhjä dialogi 100 kertaa peräkkäin.

2) Päivitysnopeus. Luotiin 300 x 300 pikselin kokoinen dialogi vasempaan ylänurkkaan. Sen jälkeen dialogia siirrettiin pikseli kerrallaan reunoja myöten koko ruudun ympäri.

3) Tekstialueen täyttö tekstillä. Täytettiin tekstialue lisäämällä siihen silmukassa yhteensä 300 sadan merkin pituista riviä.

4) Pienennettiin testattavan ohjelman pääikkuna ikoniksi ja palautettiin se takaisin normaalikokoiseksi 50 kertaa peräkkäin.

Vertailussa käytettiin Visual C++ 6.0:lla tehtyä MFC-luokkakirjastoa hyödyntävää sovellusta, joka optimoitiin nopeuden suhteen. Javan mittauksissa käytettiin perinteisellä AWT:lla ja uudella JFC-kirjastolla toteutettuja sovelluksia. Molemmat ajettiin tavukoodina JDK 1.2 -paketin mukana tulleella virtuaalikoneella käyttäen JIT-kääntäjää. Mittausympäristö oli sama kuin aikaisemmin. Tulokset esitetään taulukossa 2. Kuluneet ajat ovat 10 mittauskerran keskiarvoja ja ne esitetään sekunteina. Selvästi poikkeavat tulokset on jätetty pois keskiarvoa laskettaessa.

Taulukko 2. Käyttöliittymän suorituskyvyn mittaukset.

Mitattava asia	**C++ (s)**	**Java JFC (s)**	**Java AWT (s)**
Dialogien luominen ja tuhoaminen	4.687	5.548	5.308
Dialogin siirto	4.827	7.040	8.141
Tekstialueen täyttö	1.082	0.510	3.525
Pienentäminen ikoniksi ja takaisin	26.177	26.078	26.258

5.3 Testitulosten arviointi

Tehdyt mittaukset tukevat hyvin testitapauseditorin pohjalta saatuja kokemuksia. Javan kokonais- ja liukulukujen laskutoimituksissa ei ole huomattavaa eroa C++:aan verrattuna käytettäessä virtuaalikonetta, jossa on JIT-kääntäjä. JITin avulla näiden laskutoimitusten suoritus on jopa nopeampaa kuin optimoimattomalla C++:lla. Optimoitu C++ on kuitenkin selvästi nopeampaa kuin Java.

Tiedostonkäsittelyssä C++ jättää Javan selvemmin jälkeensä ja on lähes neljä kertaa nopeampi kuin Java JIT. Ilman JIT-kääntäjää ajettu tavukoodi on tässä tapauksessa yli 20 kertaa hitaampaa kuin C++.

Käyttöliittymävertailujen teko oli vaikeaa, koska luokkakirjastojen komponenteissa on eroja. Vertailuissa pyrittiin mahdollisimman samanlaisiin sovellustoteutuksiin kaikilla luokkakirjastoilla. Testeistä saadut tulokset ovat samansuuntaisia kuin testitapauseditorin käytöstä saadut.

Käyttöliittymäolioiden luonti on C++:ssa hieman nopeampaa. Laajemmilla sovelluksilla ero kasvaa Javan virtuaalikoneiden roskienkeruun takia. Ikkunoiden siirtäminen ruudulla on myös hitaampaa Javassa. Kolmas testi eli tekstialueen täyttäminen tekstillä ei ole täysin vertailukelpoinen luokkakirjastojen erojen takia. Pienentäminen ikoniksi ja palauttaminen takaisin on kaikissa ympäristöissä yhtä nopeaa, koska se on käyttöjärjestelmän hoitama rutiini.

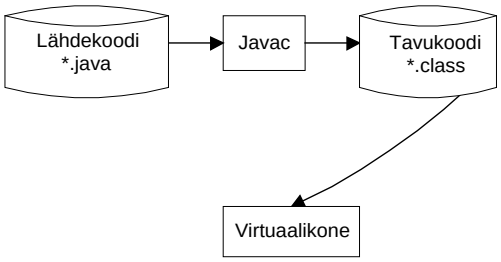
Javan AWT- ja JFC-luokkakirjastojen välillä ei havaittu merkittävää eroa. Joissain tapauksissa on AWT nopeampi ja toisissa JFC. Valinnan niiden välillä ratkaisee lähinnä käyttötarve.

5.4 Vaihtoehtoisia tapoja Java-sovelluksen kääntämiseen ja suorittamiseen

Javalla tehdyn sovelluksen kääntämiseen ja ajamiseen on useita vaihtoehtoja, joita käydään läpi seuraavissa kappaleissa.

5.4.1 Normaali tavukoodi

Lähdekoodi käännetään kehitysvaiheessa tavukoodiksi, jota voidaan ajaa Java-virtuaalikoneen avulla, kuva 38. Ajettava tavukoodi on riippumaton käyttöympäristöstä ja virtuaalikoneen tehtävänä on tulkata tavukoodia.



Kuva 38. Tavukoodin suoritus.

Työstä saadun kokemuksen perusteella Javan tavukoodin suorituskyky riittää tällä hetkellä pienten applettien ja itsenäisten sovellusten toteuttamiseen. Tällöinkin on käytössä oltava sellainen virtuaalikone, joka osaa tavukoodin ajonaikaisen optimoinnin (Just In Time Compiler). Ilman JIT-kääntäjää testitapauseditorin käyttäminen oli tuskallisen hidasta. Tällainen kokeilu tehtiin Sunin JDK-paketin mukana tulevalla java.exe-nimisellä ohjelmalla antamalla parametriksi -nojit.

Käytettäessä JIT-kääntäjää ohjelman suoritus nopeutui ratkaisevasti, mutta oli edelleenkin varsin hidasta erityisesti graafisten toimintojen suorituksessa. Kehitystyön alkuvaiheissa osasyynä hitauteen oli käytetyn Swing-kirjaston keskeneräisyys.

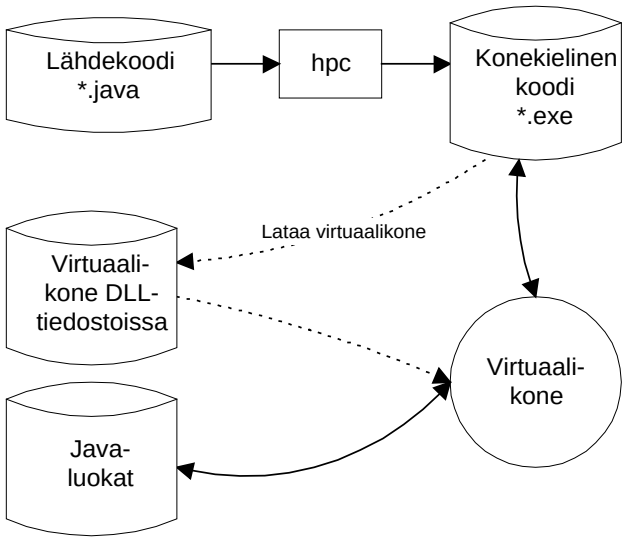
5.4.2 Konekielinen koodi

Työn kuluessa tutkittiin mahdollisuutta kääntää Javalla tehty sovellus suoraan Windowsille soveltuvaksi ajettavaksi exe-tiedostoksi. Tällaisia konekielistä koodia tekeviä kääntäjiä oli tarjolla useita, joista päätettiin kokeilla joko Symantecin tai IBM:n kehittämiä versioita. Symantecin kääntäjästä jouduttiin kuitenkin luopumaan, koska senhetkinen versio ei vielä tukenut JFC-komponentteja.

Näin jäljelle jäi IBM:n kehittämä HPC (High Performance Compiler) for Java, joka sekin oli vasta alpha-vaiheessa. HPC osoittautui suhteellisen toimivaksi, ja sillä käännetyt sovellukset toimivat hieman nopeammin kuin JIT-kääntäjän avulla ajettavat tavukoodia sisältävät luokat. HPC toimi kuitenkin virheellisesti joissain tilanteissa, joten kokeilu jätettiin kesken.

HPC:n toimintaperiaate on melko yksinkertainen, kuva 39. Java-virtuaalikone ja kaikki kieleen sisältyvät luokat on pakattu DLL-tiedostoihin (Dynamic Link Library). Toteutettu sovellus käännetään HPC-kääntäjällä, joka tuottaa konekielistä koodia.

Kääntäjän tuottama exe-tiedosto lataa käynnistyessään DLL-kirjastossa sijaitsevan Javan virtuaalikoneen, jonka jälkeen virtuaalikone huolehtii sovelluksen toiminnasta kuten normaalistikin Java-sovelluksia ajettaessa.



Kuva 39. IBM HPC:n toiminta.

5.4.3 Tavukoodin ja konekielisen koodin yhdistäminen

Javan native-rajapinnan avulla voidaan yhdistää Javalla ja C-kielellä toteutetut osat. Osa suoritettavasta koodista voidaan tehdä C-kielisenä. Tätä konekieliseksi koodiksi käännettyä moduulia voidaan käyttää Java-koodista käsin.

Native-rajapinta toimii myös toisin päin eli C-kielellä tehdyistä moduuleista voidaan tarvittaessa kutsua Java-tavukoodia. Tällä tavoin saadaan Javalla tehty sovellus näyttämään konekieliseltä koodilta. C-kielellä voidaan tehdä esimerkiksi pieni Windows-sovellus, jonka ainoana tehtävänä on käynnistää Javalla tehty sovellus. Käyttäjälle tällainen järjestely näyttää aivan normaalilta Windows-sovellukselta, vaikka tosiasiassa käytetäänkin Java-virtuaalikonetta.

Työssä kokeiltiin tätäkin mahdollisuutta, mutta sovellusta ei saatu toimimaan oikein. Sovellus käynnistyi ja näytti normaalilta, mutta ei ottanut vastaan mitään viestejä näppäimistöltä eikä hiireltä. Luultavasti syynä oli jälleen sovelluksen käyttämä Swing-luokkakirjasto, joka on vielä niin uusi, ettei muu Java-tekniikka ole täysin valmis käyttämään sitä.

5.4.4 Hot Spot

Hot Spot on uusi, Sunin kehittämä ajonaikaista optimointia tekevä kääntäjä. Sen kehitystyö on vielä kesken, joten tässä työssä ei päästy vertailemaan Hot Spotin nopeutta muihin ratkaisuvaihtoehtoihin verrattuna. Sunin mukaan Hot Spotin tuoma nopeuslisäys on todella merkittävä. Sunin mukaan Hot Spotin avulla päästään lähes C++:lla tehtyjen sovellusten nopeuksiin.

Javan virtuaalikoneen tehtävänä on taata sovelluksen nopeus, turvallisuus ja oikeellinen suoritus mitä erilaisimmissa laite- ja käyttöjärjestelmäympäristöissä. Tämä asettaa virtuaalikoneen toteutukselle suuria vaatimuksia, ja virtuaalikoneen toiminta on kriittistä kaikille suunnittelijoille ja käyttäjille. [27]

Javan virtuaalikoneissa on jo tällä hetkellä käytössä tekniikka, joka tunnetaan nimellä JIT. Siinä ajettavaa tavukoodia optimoidaan ajon aikana. Tällainen ajonaikainen optimointi on erittäin tärkeää Javan tapauksessa, ja Hot Spot menee tässä optimoinnissa vielä paljon pidemmälle kuin nykyisin käytössä olevat optimoivat virtuaalikoneet.

Virtuaalikone voi hyödyntää kaikkia tietojaan ajoympäristöstä ajaessaan sovellusta. Voidaan esimerkiksi suunnitella virtuaalikone, joka generoi tietylle prosessorityypille optimaalista konekielistä koodia ajonaikaisesti. Vaikka esimerkiksi Intelin valmistamat Pentium ja Pentium II osaavat molemmat suorittaa samaa konekielistä koodia, molemmille prosessoreille voidaan löytää optimaalinen koodi, joka ei toimi toisessa prosessorissa optimaalisesti.

Seuraavat kappaleet perustuvat lähteeseen [27]. Tietoa HotSpotista on saatavilla melko vähän, ja siksi ohessa esitetyt ominaisuudet voivat vielä muuttua, kun HotSpot-tekniikka tulee markkinoille.

5.4.5 HotSpot-arkkitehtuuri

Java HotSpot toteuttaa samat Sunin virtuaalikoneille asettamat vaatimukset kuin nykyisetkin virtuaalikoneet. Monen vuoden perustutkimuksen ansiosta HotSpotin virtuaalikoneen sisäinen arkkitehtuuri on kuitenkin muuttunut täysin aikaisempiin virtuaalikonetoteutuksiin verrattuna. Java HotSpotin virtuaalikoneessa käytetään monia suoritusta nopeuttavia tekniikoita. Alla on lueteltu tärkeimmät uudet ominaisuudet:

- Ajonaikainen adaptiivinen optimointitekniikka, joka keskittyy nimenomaan koodin kriittisimpien osien optimointiin. Tästä tuleekin nimi HotSpot.

- Nopea säikeiden synkronointi.
- Uusittu roskien keruu (Garbage Collector, GC).
- Virtuaalikoneen toteutuksessa on käytetty korkean tason oliopohjaista suunnittelutyyliä, jonka ansiosta ylläpidettävyys ja laajennettavuus ovat hyviä.

Uusi muistimalli

Java HotSpotin muistimalli on uudistunut, ja se sisältää muun muassa seuraavat ominaisuudet:

- Suorat osoittimet objekteihin
- Kahden sanan pituiset olioiden otsikot (two-word object headers)
- Reflektiivisen tiedon esitys olioina
- Tuki luonnollisen kielen säikeille.

Perinteisissä virtuaalikonetoteutuksissa käytetään epäsuoria kahvoja (handles) olioviitteiden esittämiseen. Tämä helpottaa olioiden uudelleenjärjestelyä roskien keruun yhteydessä, mutta on tehotonta. Java HotSpotin virtuaalikone käyttää epäsuorien kahvojen sijasta suoria osoittimia. Suorat osoittimet toimivat samoin kuin C-kielessä, ja niiden avulla päästäänkin C-kielen nopeuteen olioviitteiden käsittelyssä.

Nykyisissä virtuaalikoneissa käytetään kolmen sanan pituisia otsikkokenttiä kaikille olioille. HotSpotissa otsikkokentän pituus on lyhennetty kahteen. Ainoina poikkeuksina ovat taulukot, joille käytetään edelleenkin kolmen sanan pituisia otsikkokenttiä. Otsikkokentän lyhentämisellä saavutetaan noin 8 prosentin säästö pinon koossa. [27]

Luokat, metodit ja muu sisäinen reflektiivinen tieto esitetään suoraan olioina pinossa, vaikka niitä ei voida suoraan käsitellä sovelluksessa. Menettely yksinkertaistaa muistimallia, ja mahdollistaa myös näiden objektien keräämisen samalla roskien kerääjällä.

HotSpot tukee luonnollisen kielen säikeitä, ja se sisältää tuen myös ns. pre-emptiivisille säikeille. Pre-emptiivisten säikeiden tuki on toteutettu käyttöjärjestelmätason skedulointimekanismin avulla. Luonnollisen kielen säikeet ja skedulointi mahdollistavat käyttöjärjestelmätason tukeman multiprosessoinnin.

Uusittu roskien keruu

Nykyisissä virtuaalikonetoteutuksissa käytetyt roskien kerääjät ovat vielä melko alkeellisia eikä niiden toiminta ole mitenkään reaaliaikaista. Toisin sanoen pieniä sovelluksia ajettaessa voi käydä usein niin, että roskien keruu käynnistyy vasta juuri ennen virtuaalikoneen pysähtymistä. On myös tavallista, että suurempia sovelluksia ajettaessa roskien keruu käynnistyy jossakin vaiheessa ohjelman suoritusta, ja tämä on havaittavissa ohjelman selvänä hidastumisena. Roskien keruumekanismia käytetään yleensä vasta silloin, kun on pakko eli vasta silloin, kun resurssit ovat lopussa.

HotSpotin roskienkeruumekanismi on erilainen; roskien keruuta tapahtuu koko ajan, jolloin käyttäjä ei huomaa sitä mitenkään. Olioiden varaaminen on entistä nopeampaa ja roskien keruu tarkempaa. Vanhojen olioiden poistaminen tehdään mark-compact-tekniikalla, jolla eliminoidaan muistin pirstoutuminen ja lisätään muistin paikallisuutta. HotSpotin roskien keruu on täysin tarkkaa (fully accurate), kun tällä hetkellä käytössä olevien virtuaalikoneiden roskien keruu on joko konservatiivista (conservative) tai osittain tarkkaa (partially accurate).

Java HotSpot kääntäjä

Suorituskyvyn kannalta ehkä ratkaisevimmat muutokset tulevat HotSpotin kääntäjään. Käytössä olevat JIT-kääntäjät kääntävät Java-tavukoodia konekieliseen muotoon ohjelman suorituksen avulla. Tämä hidastaa luonnollisesti ohjelman ajoa valmiiksi konekieliseksi käännettyyn ohjelmaan verrattuna.

JIT-kääntäjien käyttämät optimointimenetelmät perustuvat perinteisten ohjelmointikielien optimointitekniikoihin. Koska Java poikkeaa paljon muista ohjelmointikielistä, perinteiset tekniikat eivät toimi Javassa yhtä hyvin. Perinteiset optimointimenetelmät eivät toimi Javassa mm. seuraavista syistä:

- Javan virtuaalikoneen täytyy tarkistaa, etteivät ohjelmat riko kielen semantiikkaa eivätkä osoita suoraan vääriin muistialueisiin.

- Olioiden varaus on Javassa erilaista kuin esimerkiksi C++:ssa.

- Javassa metodien kutsut ovat virtuaalisia.

- Javassa on mahdollisuus ladata luokkia dynaamisesti.

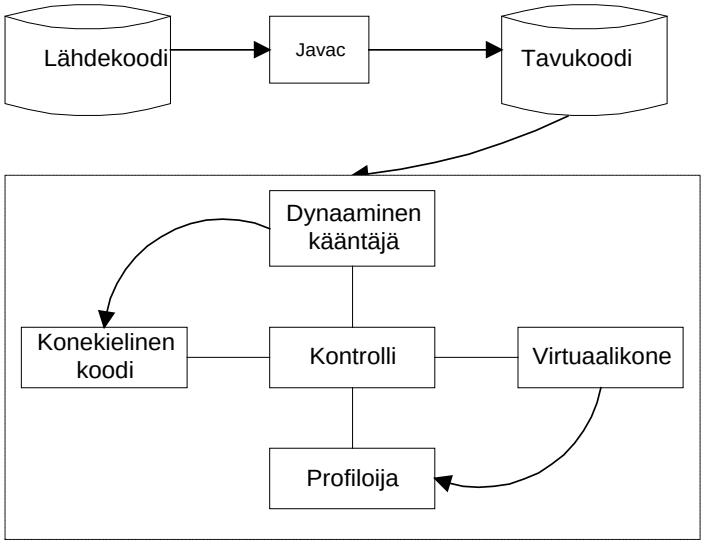
Edellä esitettyjen syiden takia on HotSpotin kääntäjässä otettu käyttöön uusi adaptiivinen optimointitekniikka eli Hot Spot. Optimoinnissa keskitytään sellaisten koodin osien

optimointiin, joissa optimoinnilla saavutetaan suurin hyöty ohjelman suoritusnopeudelle. Java HotSpot virtuaalikone ei käännä koko tavukoodia konekieliseen muotoon ohjelmaa käynnistettäessä kuten JIT-kääntäjät tekevät. Sen sijaan Java HotSpot virtuaalikone ajaa koodin kerran läpi ja etsii kriittisimmät osat, joihin optimointi kannattaa kohdistaa. Optimoitavien osien etsintä jatkuu koko ohjelman suorituksen ajan.

Optimoitava koodi käännetään tämän jälkeen konekieliseen muotoon. Sen lisäksi kääntäjä sijoittaa osan metodeista suoraan koodiin (method inlining). Näin voidaan pienentää metodien kutsumiseen kuluvaa aikaa, joka on nykyisissä virtuaalikonetoteutuksissa pullonkaula. Menetelmällä syntyy suurempia koodilohkoja, joiden optimointi voidaan toteuttaa paljon tehokkaammin.

Method inlining -tekniikan käyttö onnistuu Hot Spotissa paremmin kuin missään aikaisemmassa optimoivassa kääntäjässä. Hot Spotin optimointivaiheessa tiedetään paljon enemmän suoritettavasta ohjelmakoodista kuin perinteisten staattisten kääntäjien tapauksessa. Muun muassa metodien koot ovat tiedossa, joten liian suuria metodeja ei sisennetä.

Koska Javassa on mahdollista ladata ajon aikana uutta Java-koodia ajettavaan ohjelmaan, HotSpot-tekniikkaan perustuvan kääntäjän on voitava optimoida uudelleen muuttuneet koodin osat. Tätä tekniikkaa kutsutaan nimellä dynaaminen uudelleenoptimointi (dynamic deoptimization). Kuvassa 40 [27] on esitetty Java HotSpot -arkkitehtuurin perusosat.



Kuva 40. HotSpotin arkkitehtuuri.

5.4.6 Java chip

Eräänä ratkaisuna Javan suorituskykyongelmiin on esitetty sellaisia prosessoreja, jotka osaavat suorittaa suoraan Javan tavukoodia tarvitsematta erillistä tulkkausta. Sun MicroSystems on tehnyt muutaman tällaisen prosessorin, mutta ne eivät ole saavuttaneet suurta suosiota. Tällä hetkellä Java Chip -prosessorien tulevaisuus ei näytä kovin hyvältä.

5.5 Tulevaisuuden näkymät ja jatkotutkimus

Java ja siihen liittyvät tekniikat ovat tällä hetkellä jatkuvan kehityksen kohteena. Tämän takia tässäkin työssä esitetyt asiat vanhenevat muutamassa vuodessa, joten jatkotutkimukset ja kehityksen seuraaminen ovat tarpeellisia.

Seuraavat vuodet ovat ratkaisevia Javan tulevaisuuden kannalta. Suurimpana uhkana on, ettei Java kykene täyttämään sille asetettuja odotuksia. Vaikka Java on saanut suurta suosiota ohjelmistokehittäjien parissa, heidän mielenkiintonsa voi laantua alkuinnostuksen jälkeen ellei toiveita pystytä täyttämään.

Suurimpana esteenä Javan kehitykselle on tulkattavan tavukoodin hitaus. Jatkotutkimuksessa voidaankin keskittyä erityisesti uusien kääntäjien ja virtuaalikoneiden suorituskyvyn arviointiin. Tässä työssä arvioitiin lyhyesti HotSpotia, joka voi siirtää Javan hitauden historiaan. HotSpot ei ole kuitenkaan vielä markkinoilla eikä sen suorituskykyä päästy vertailemaan.

Mielenkiintoinen tutkimuskohde on myös Javan käyttö sulautettujen järjestelmien ohjelmistokehitykseen. Java suunniteltiin alun perin juuri sulautettujen laitteiden ohjelmointiin soveltuvaksi, mutta ainakin tällä hetkellä Javan ongelmana on kovien reaaliaikavaatimusten saavuttaminen. Sulautettuja järjestelmiä varten kehitellään erilaisia ratkaisuja. Kaikille niistä on yhteistä se, että sulautetuissa laitteissa tarpeettomia kielen ominaisuuksia karsitaan pois.

Sun MicroSystems esitteli tammikuussa 1999 JINI-tekniikan [28], joka on tarkoitettu sulautettujen laitteiden liittämiseen yhteen. Yhteenliittäminen toteutetaan Javan ja RMI:n (Remote Method Invocation) [29] avulla. Onnistuessaan JINI toisi Javan kodin elektroniikkaan, ja vahvistaisi merkittävästi kielen asemaa.

6. Yhteenveto

Työssä tutkittiin esimerkin avulla Javan, ja erityisesti JFC-luokkakirjaston, soveltuvuutta itsenäisen sovelluksen toteutuksessa. Esimerkkisovelluksena käytettiin MOSIM-testausympäristöön kehitettyä testitapauseditoria. Se toteutettiin kokonaan Javalla ja käyttöliittymän rakentamiseen käytettiin JFC-luokkia.

Javan siirrettävyys ympäristöstä toiseen toteutui odotetusti. Suuria ongelmia ei esiintynyt kokeilluissa ympäristöissä. Testitapauseditori kehitettiin Windows NT 4.0:ssa. Sovelluksen toiminta oli lähes samanlaista Sun MicroSystemsin Solaris-käyttöjärjestelmässä. JFC-komponenttien toteutuksessa oli pieniä eroja näiden kahden ympäristön välillä. Erot olivat kuitenkin lähinnä kosmeettisia eivätkä haitanneet sovelluksen käyttöä millään tavalla.

Javan suorituskyky on yhä suuri ongelma. Ajonaikaisen tulkkauksen takia Java-koodista on lähes mahdotonta saada yhtä nopeaa kuin konekieliseksi käännetystä koodista. Tämän takia Javaa ei voi vielä tällä hetkellä suositella laajoihin sovelluksiin. Java ei ole paras ratkaisu myöskään sellaisiin sovelluksiin, joissa hyvä suorituskyky on tärkeintä.

Siirrettävien luokkakirjastojensa takia Java on hyvä vaihtoehto, kun kehitetään ohjelmistoja useisiin käyttöympäristöihin. Laajoja ohjelmistoja toteutettaessa on mietittävä tarkkaan ohjelmiston suorituskykyvaatimukset. Yksinkertaisissa sovelluksissa Javan käyttö on usein paras vaihtoehto. Parhaimmillaan Java on kehitettäessä Internet- tai muita verkkosovelluksia, koska kieli tarjoaa kehittyneet verkko-ominaisuudet.

Työn aikana tuli esille ongelmana Javan ja siihen sisältyvien luokkakirjastojen nopea kehitys. Vaikka Javan uudet versiot ovat alaspäin yhteensopivia, tehtyä sovellusta joudutaan silti usein muokkaamaan uuden version mukaiseksi uusien versioiden tarjoamien lisäominaisuuksien ja paremman laadun takia. Tämä johtaa helposti jatkuvaan päivityskierteeseen.

Lähdeluettelo

[1] http://www.sunworld.com/swol-07-1995/swol-07-java.html.

[2] http://www.w3.org/.

[3] http://www.javasoft.com/products/jdk/1.2/.

[4] Microsoft Corporation (1997) Microsoft Visual C++ MFC Library Reference, Part 1 and Part 2. Microsoft Press, 1456 s., 1392 s.

[5] Borland Press (1996) Borland C++ Resource Kit, ObjectWindows Programmer's Guide. Sams/Borland Press.

[6] Honka, H. (1992) A simulation-based approach to testing embedded software. Technical Research Centre of Finland, VTT Publications 124, 118 s.

[7] Shneiderman, B. (1992) Designing the User Interface: Strategies for Effective Human-Computer Interaction. Addison-Wesley Publishing Company, 573 s.

[8] Chasan, G. (1997) Trends in Rapid Application Development. http://www.wtgi.com/article_trends.htm.

[9] Collins, D. (1995) Designing Object-Oriented User Interfaces. The Benjamin/Cummings Publishing Company, 590 s.

[10] Tesler, L. (1983) Object Oriented User Interfaces and Object Oriented Languages. ACM Conference on Personal and Small Computers, s. 3 - 5.

[11] Boehm, B. (1988) A spiral model of software development and enhancement. IEEE Computer, 21, 5, s. 61 - 72.

[12] Coad, P. & Yourdon, E. (1991) Object-Oriented Design. Prentice Hall, 197 s.

[13] Brooks, Frederick P. Jr. (1987) No Silver Bullet. Essence and Accidents of Software Engineering. Computer Magazine, April 1987. First published in Information Processing 1986, ISBN No. 0444-7077-3, H. J. Kugler, ed., Elsevia Science Publishers B.V. (North-Holland) IFIP 1986.

[14] Jaaksi, A. (toim.) (1998) Oliot, komponentit ja ohjelmistoarkkitehtuurit. Tampereen yliopisto, tietojenkäsittelyopin laitos.

[15] http://www.software.ibm.com/ad/opendoc/.

[16] http://java.sun.com/beans/.

[17] Curtin, M. (1998) WRITE ONCE, RUN ANYWHERE™: WHY IT MATTERS. http://java.sun.com/features/1998/01/wora.html.

[18] http://java.sun.com/marketing/collateral/foundation_classes.html

[19] http://java.sun.com/products/jfc/tsc/index.html.

[20] http://java.sun.com/products/jfc/accessibility.html.

[21] http://java.sun.com/products/java-media/2D/index.html.

[22] Lewis, S. (1995) The Art and Science of SmallTalk. Hewlett-Packard professional books. Prentice Hall International (UK) Limited, 212 s.

[23] Ward, P. T. & Mellor, S. J. (1985) Structured development for real-time systems. Volumes 1-3. New York, Yourdon Press, 509 s.

[24] Gafni, V., Shelef, R. & Tibber, H. (1989) A real-time simulation environment for embedded computer systems software testing. Proceedins of the Fourth Israel Conference on Computer Systems and Software Engineering. Herzlia, Israel, 5-6 June 1989. IEEE Computer Society Press, s. 344 - 352.

[25] Mosim 7.0 User's Guide (1995). VTT Elektroniikka, 49 s.

[26] User’s Guide for Variables and If-Then-Else in MOSIM Test Controller (1997). VTT Elektroniikka, 19 s.

[27] http://java.sun.com/products/hotspot/whitepaper.html.

[28] http://www.sun.com/jini/.

[29] http://www.javasoft.com/products/jdk/rmi/index.html.