Siirrettävän ja uudelleenkäytettävän tutkimuskäyttöliittymän suunnittelu ja toteuttaminen

Markku Mielityinen

Tietotekniikan pro gradu –tutkielma
8. helmikuuta 2002

Jyväskylän yliopisto
tietotekniikan laitos

Tekijä: Markku Mielityinen.
Yhteystiedot: mmmm@st.jyu.fi.
Työn nimi: Siirrettävän ja uudelleenkäytettävän tutkimuskäyttöliittymän suunnittelu ja toteuttaminen.
Title in English: Design and implementation of portable and reusable scientific user interface.
Sivumäärä: 180.
Linja: Ohjelmistotekniikka.
Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos.
Avainsanat: Käyttöliittymät, siirrettävyys, laajennettavuus, uudelleenkäytettävyys, suunnittelumallit, komponenttiteknologiat, kääntäjät, tulkit, virtuaalikoneet, tekoäly, konenäkö.
Keywords: User interfaces, portability, extendability, software re-use, design patterns, component technologies, compilers, interpreters, virtual machines, artificial intelligence, machine vision.

Esipuhe

Tämä pro gradu -tutkielma kirjoitettiin Jyväskylän yliopiston tietotekniikan laitoksella vuosina 2001-2002. Työn ohjaajina toimivat Pasi Koikkalainen ja Jouni Raitamäki.

Haluan kiittää työni ohjaajia heidän työpanoksestaan tähän opinnäytteeseen liittyen. Lisäksi haluan kiittää Anssi Lensua häneltä saamastani ohjauksesta työn pohjan muodostavan NDA-kehitysympäristön käyttöön ja koko työryhmää, joka on vuosien aikana osallistunut kyseisen ohjelmiston kehitykseen.

Lopuksi haluan kiittää perhettäni opiskelulle otollisen ilmapiirin luomisesta.

Jyväskylässä 8. helmikuuta 2002
Markku Mielityinen

Tiivistelmä

Viimevuosina niinkutsutun laskennallisen älykkyyden käyttö on yleistynyt nopeasti. Kyseisiä menetelmiä käytetään niin tieteelliseen tutkimukseen kuin teollisuusprosessien ohjaukseenkin. Viimepäiviin asti laskennallista älykkyyttä ovat käyttäneet lähinnä tutkimuslaboratoriot ja suuret yritykset johtuen menetelmien soveltamiseen tarvittavasta asiantuntemuksesta. Tulevaisuudessa tarvitaan kuitenkin yhä enemmän myös tavallisten ihmisten käytettävissä olevia sovelluksia.

Tämän työn tavoitteena on kehittää siirrettävä, laajennettava ja uudelleenkäytettävä sovelluskehitysympäristö laskennallisesti älykkäisiin analyysiketjuihin pohjautuvien sovellusten toteuttamiseen. Sovelluksen toiminnallisen pohjan muodostukseen käytetään jo olemassaolevaa NDA-ympäristöä, jota on kehitetty ja käytetty Jyväskylän yliopistolla muutamien vuosien ajan.

Paremman ymmärtämyksen saamiseksi ongelmasta ja sen mahdollisista ratkaisuista, suoritetaan ohjelmistotuotannon peruskirjallisuuteen sijoittuva kirjallisuuskatsaus, jota täydennetään monilla asiaan liittyvillä julkaisuilla. Tavoitteena on löytää valmiita ratkaisuja tai ainakin vihjeitä siihen, kuinka määritellyn kaltainen ohjelmistoprojekti tulisi muodostaa. Tarkoitusta varten syvennytään tarkemmin suunnittelumalleihin, komponenttiteknologioihin ja kääntäjä-, tulkki- ja virtuaalikonepohjaiseen ohjelmistojen tuotantoon.

Ohjelmiston kehitys aloitettiin sopivan kehitysympäristön valinnalla. Tämän jälkeen arvioitiin aiemmin esiteltyjen teknologioiden soveltamismahdollisuuksia. Ohjelmointi aloitettiin komponenttikirjaston toteutuksella. Myöhemmin komponettien pohjalta toteutettiin graafinen käyttöpaneeli, joka sisältää kaikki alla olevasta NDA-ympäristöstä tarvittavat toiminnat. Työssä esitettyjen ajatusten oikeellisuuden koettelemiseksi päätettiin toteuttaa teollisuussovellus paperipinnan kuitujen tutkimukseen. Sovellusta voidaan käyttää esimerkiksi paperitehtaissa suoritettavaan paperilaatujen luokitteluun.

Abstract

In recent years the use of computationally intelligent systems has increased enormously. Use of these methods ranges from scientific research to industrial process control. Until now computationally intelligent methods have mostly been used by research laboratories and big companies due to advanced knowledge requirements in applications. In the future there is an increasing need for easy-to-use software suitable for consumer use.

The goal of this thesis is to develop a portable, extendable and re-usable software development environment for applications based on analysis chains, which contain various methods of computational intelligence. Software functionality is to be based on an already existing NDA environment which has been developed and is beeing used in the University of Jyväskylä for a few years now.

In order to get a better understanding of the problem and it's possible solutions we review the mainstream software engineering literature and many related publications. Our aim was to find either ready solutions or at least clues to how the software project should be formed. For these purposes we took a closer look at design patterns, component technologies and compiler, interpreter and virtual machine based software development.

Implementation of the software began with selection of proper software development tools. After that we evaluated the previously introduced technologies. Software programming began on component library implementation which was later applied to a graphical software panel which wraps all necessary functionality of the underlying NDA environment. To verify the usability of our ideas we also implemented an industrial application for fibre image analysis which is to be used for paper quality classification in papermills.

Sisältö

Kuvat

Taulukot

Luku 1

Johdanto

Tieteellisessä tutkimustyössä ja yritysten tuotekehityksessä tietokoneavusteiset analyysit ja simulaatiot ovat saavuttaneet yhä keskeisemmän roolin uuden tiedon luomisprosessissa. Useille tieteenaloille on kehittynyt juuri näihin erityisvaatimuksiin mukautettuja tutkimusohjelmistoja. Yleensä tutkimusohjelmiston tavoitteena on mallintaa joku realimaailman tutkittava ilmiö. Vaikka ilmiö voi olla luonnontieteen tutkimuskentän ohella esimerkiksi liiketaloudellinen tai hallinnollinen ongelma, pätevät niiden mallintamiseen periaatteiltaan oleellisesti samanlaiset menetelmät. Oleellista ilmiössä on, että sen olemusta voidaan kuvata joukolla sääntöjä, joita tietokonesimulaatiolla voidaan yrittää jäljitellä.

Tässä opinnäytteessä suunnitellaan ja toteutetaan uudelleenkäytettävä käyttöliittymä laskennallisesti älykkäisiin menetelmiin perustuvan tutkimusohjelmiston tarpeisiin. Laaditulla käyttöliittymällä toteutetaan lisäksi esimerkkisovellus paperin kuituanalyysiä varten.

Työn sisältö:

Toisessa luvussa luodaan vaatimusmääritys siirrettävän ja uudelleenkäytettävän tutkimuskäyttöliittymän toteuttamista varten.

Kolmannessa luvussa luodaan katsaus tieteelliseen kirjallisuuteen tavoitteena kartoittaa vaatimusmäärittelyn toteutukseen tarvittavia teknologioita.

Neljännessä luvussa tutustutaan käyttöliittymän alla toimivaan tutkimusohjelmistoon. Luvussa kartoitetaan ohjelmiston ominaisuudet edellisessä luvussa määritellyillä osa-alueilla.

Viidennessä luvussa hahmotellaan tulevaa ohjelmistoprojektia valitsemalla ohjelmistonkehitystyökalut. Valittuun Delphi/Kylix -ympäristöön tutustumisen jälkeen päätetään kirjallisuuskatsauksen pohjalta tärkeimmistä ohjelmistoteknisistä linjanvedoista.

Kuudennessa luvussa perehdytään ensimmäisenä toteutettavan kompo-

nenttikirjaston kehitystyöhön. Luvun loppuosassa lukijalle opetetaan komponenttikirjastolla suoritettavaa sovellusten ohjelmointia.

Seitsemännessä luvussa tutustutaan seuraavana luotavan graafisen käyttöpaneelin kehitysvaiheisiin. Luvun loppuosassa lukijalle opetetaan käyttöpaneeliin pohjautuvaa sovellustuotantoa.

Kahdeksannessa luvussa kehitetään komponenttikirjaston ja käyttöpaneelin ohjelmointia esittelevä esimerkkisovellus. Vaatimusmäärittelyn toteutumisen mittauksen ohella pyritään opettamaan lukijaa analyysiketjuihin pohjautuvien sovellusohjelmien tuotannosta.

Yhdeksännessä luvussa arvioidaan ohjelmistoa vaatimusmäärityksen ja käytettävyyden näkökulmasta.

Kymmenennessä luvussa suoritetaan lyhyt yhteenveto koko työn suorituksesta niin lähdemateriaalien käytön, ohjelmointityön kuin kirjallisen toteutuksen näkökulmasta.

Liitteessä A esitetään suunnitelma komponenttikirjaston luokkarakenteesta UML-kuvauskielellä.

Liitteessä B esitetään suunnitelma käyttöpaneelin luokkarakenteesta UML-kuvauskielellä.

Liitteessä C listataan esimerkkisovelluksen kehityksessä laaditut käyttöliittymän kuvaustiedostot.

Liitteessä D listataan esimerkkisovelluksen kehityksessä laaditut sovelluslogiikkatiedostot.

Luku 2

Vaatimusten määritys

Tämän opinnäytteeksi laadittavan työn tavoitteena on tutkia tutkimuskäyttöön soveltuvan käyttöliittymän toteutusmahdollisuuksia. Käyttöliittymän on tarkoitus tukea olemassa olevaa ja useissa projekteissa käytettyä ohjelmistoalustaa. Ohjelmistoja varten on aiemmin suunniteltu ja toteutettu useita erilaisia sovellusspesifisiä käyttöliittymiä, joita on vaikea siirtää sovelluksesta toiseen. Tässä työssä pääpainopisteinä pidetään luotavan ohjelmiston siirrettävyyttä, laajennettavuutta ja uudelleenkäytettävyyttä. Vaatimusmäärittelyn tueksi tutkitaan ensin John M. Chambersin esittämiä ajatuksia käyttöliittymien suunnittelun ja toteutuksen painopisteistä [1], [2] ja [3]. Osa ajatuksista on esitetty hieman alkuperäisestä poikkeavalla tavalla, jolloin on pyritty laajentamaan ja sopeuttamaan alkuperäistä ideaa paremmin sovelluskohteeseen sopivaksi.

2.1 Johdatus tutkimusohjelmistojen avulla suoritettavaan tutkimustyöhön

Luonto monine ilmiöineen muodostaa hyvin kirjavan tutkimusalueen ja siten monien luonnonilmiöiden mallinus tietokoneella ei ole täysin triviaali tehtävä. Tietokoneilla luotava diskreetti ja äärellinen maailmankuva voi parhaimmillaankin vain lähestyä realimaailman ilmiöitä ja siten mallinnuksen toteutusta edeltävään suunnitteluun on syytä kiinnittää erityistä huomiota. Tietokoneille kehitetyt laskennallisesti älykkäät menetelmät ovat käytännössä varsin tyhmiä algoritmeja. Varsin monet nykyisistä prosessiketjuista ovat vielä täysin riippuvia käyttäjältä saatavasta ohjausinformaatiosta ja siten käyttäjän tekemät virheet johtavat lähes aina analyysista saavutettujen tulosten vääristymiseen.

Tutkimustyö, suoritettiinpa se millä tieteenalalla hyvänsä sisältää yleensä

seuraavat kolme työvaihetta: organisointi, visualisointi ja analysointi.

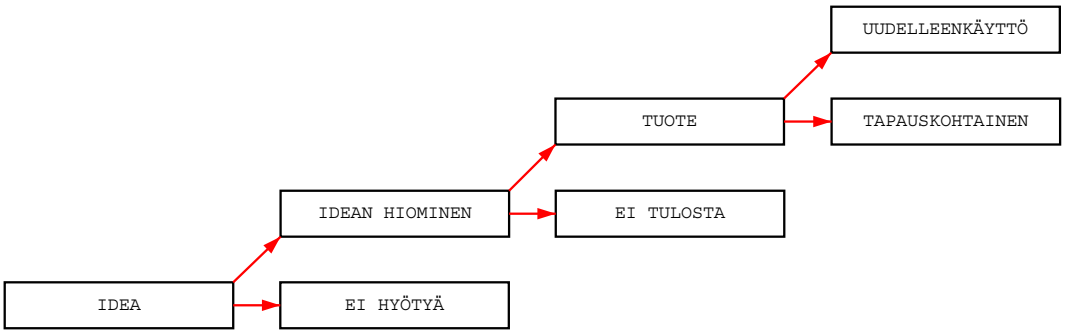
Kuva 2.1: Tutkimustyön vaiheet.

Organisointivaihe sisältää tiedon keräämiseen ja järjestelyyn liittyvän työn. Jotta tietokoneella voidaan suorittaa minkäänlaista mallinnusta, on mallinnuksen kohteesta ensin kerättävä havaintoja, jotka järjestellään ja tallennetaan tietokoneen ymmärtämään muotoon. Yleisimmäksi tiedonhallintaalkioksi on muodostunut yksinkertainen tietomatriisi monine variaatioineen. Tutkimusalueesta riippuen voi esiintyä tarvetta myös monipuolisemmille tietorakenteille. Toteutukseltaan yksinkertaisena, mutta ilmaisuvoimaltaan kuitenkin yleensä riittävänä, tietomatriisit muodostavat lähes kaikkien tietojenkäsittelyohjelmistojen selkärangan taulukkolaskentasovelluksista aina tietokantaohjelmistoihin. Koska tietotauluilla luotava relaatiomalli ei ole parhaimmillaan rakenteisen tiedon esittämisessä, on sitä korvaamaan kehitetty useita oliomalleja, joilla tietoa voidaan kuvata relaatiomallia monipuolisemmin. Relaatiomallia on kuitenkin täydennetty ja hyvällä suunnittelulla monet ongelmakohdista voidaan kiertää. Erilaiset oliomallit ovatkin jääneet vähemmistöön, eikä niiden käytön enää uskotakaan yleistyvän.

Visualisointiin kuuluu kerätyn ja käsitellyn data-aineiston muuntaminen ihmissilmälle havainnolliseen muotoon. Tutkimuksissa on saatu selville, että käyttäjälle laadittavat grafiikkaan perustuvat visualisoinnit, kuten histogrammit ja pinnat antavat käyttäjälle huomattavasti paremman käsityksen kerätyn data-aineiston sisällöstä, kuin mitä pelkkiä tietomatriiseja katselemalla voidaan saavuttaa. Amatöörikäyttäjille, joille ei varsinaisen tutkimustyön ohessa ole kehittynyt silmää data-aineiston numeeriseen havainnoimiseen, graafiset esitykset muodostavat helpoimman ja useinmiten ainoan tavan ymmärtää hankitun tiedon sisältöä. Ohjelmiston käyttäjän työskentelyä voidaan helpottaa mahdollistamalla interaktiivinen työskentely, jossa käyttäjä voi helposti suorittaa tarvittavia muutoksia ja nähdä välittömästi muutosten vaikutukset. Visualisointityökaluilla voidaan pelkän data-aineiston havainnollistamisen ohella suorittaa myös datan käsittelyä, sillä käyttäjän on usein helpompi suorittaa muutokset suoraan visuaalisiin grafiikoihin kuin raakaan data-aineistoon.

Analysointivaihe sisältää erilaisia data-aineiston käsittelyä suorittavia prosesseja, joilla pyritään löytämään aineistosta jokin etsitty ominaisuus. Analyysiprosessit ovat yleensä laskennallisesti raskaita ja siten oikealla prosessin ohjauksella voidaan säästää huomattavasti aikaa. Ohjelmiston käyttäjien kannalta onkin tärkeää, että käytettävissä olevia analyysiprosesseja voidaan jatkuvasti ohjata saavutettujen tulosten perusteella.

Tutkimustyön tulokset syntyvät usein iteratiivisen kehitysprosessin seurauksena. Käytettävän tutkimusohjelmiston tulisikin mahdollistaa progressiivinen ideoiden kehitys, jossa erilaisia idean protoversioita voidaan toteuttaa mahdollisimman nopeasti ja vähällä vaivalla.



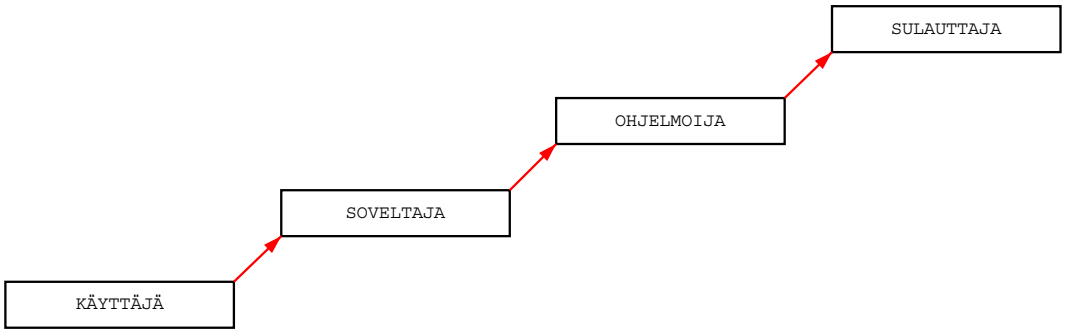
Kuva 2.2: Idean kehitys.

Tutkimustyössä syntyy usein 'villejä' ideoita, joiden toimivuudesta ei voida olla täysin varmoja ja siten idean toteutus malliksi, jolla toimivuus voidaan selvittää, tulisi olla helppoa. Suurin osa ideoista osoittautuu yleensä epäkäytännöllisiksi. Osa ideoista on kuitenkin toteuttamiskelpoisia ja ne voidaan siirtää seuraavalle kehitystasolle.

Toimivaksi havaitun idean kehittyessä yhä pidemmälle tarvitaan sen toteuttamiseen yhä tehokkaampia ja vapaampia ilmaisukeinoja. Viimeisimmät versiot toteutetaan yleensä jollakin ohjelmointikielellä ja siten sopivasti skaalautuvan laajennettavuuden olemassaolo voidaan katsoa oleelliseksi.

Idean saavuttaessa riittävän kypsyyden siitä tuotetaan yleensä tuotantoversio, jota voidaan soveltaa ongelmien ratkaisuun. Toteutuksesta ja sovellusalasta riippuen luodusta mallista voidaan laatia uudelleenkäytettävä komponentti, jolloin ideaa voidaan hyödyntää tulevien tutkimusprojektien rakennuspalikkana.

Tieteellisten ohjelmistojen käyttäjiä voidaan profiloida esimerkiksi jakamalla heidät seuraavasti neljään kategoriaan: käyttäjä, soveltaja, ohjelmoija ja sulauttaja.



Kuva 2.3: Käyttäjän kehitys.

Käyttäjät muodostavat käyttäjäkunnan ensimmäisen tason. Uudet käyttäjät harjaantuvat ensisijassa ohjelmiston perusominaisuuksien käyttöön ja vasta pidempiaikaiset käyttäjät saavuttavat riittävän kokemuspohjan ylemmille tasoille.

Soveltajat muodostuvat kokeneista käyttäjistä, jotka osaavat jo hyödyntää ohjelmiston tarjoamia palveluita ja koostaa niistä tarvitsemiaan kokonaisuuksia. Ohjelmiston soveltava käyttäminen sisältää peruskäytön lisäksi lähinnä makrokieliin pohjautuvaa ohjelmointia.

Ohjelmoijat hallitsevat käyttämänsä ohjelmiston ja sen tarjoamat palvelut. Ohjelmiston tarjoamien palveluiden lisäksi ohjelmoijat kykenevät tuottamaan ohjelmistoon tarvitsemiaan laajennuksia.

Sulauttajia ovat ohjelmoijien ohella käyttäjäkunnan pisimmälle edistyneet henkilöt. Sulauttajat kykenevät laajennusten tuottamisen ohella liittämään käyttämänsä ohjelmiston osaksi monien ohjelmistojen muodostamaa kokonaisuutta ja täten luomaan suurempia kuin yhden ohjelmiston käsittäviä järjestelmiä.

2.2 Toiminnalliset vaatimukset

Toiminnallisiin vaatimuksiin on kerätty tärkeimmät tämän työn ohjelmiston suunnitteluun vaikuttavat tavoitteet. Vaatimukset pyritään esittämään siten, että ne rajaisivat mahdollisimman vähän toteutusvaiheessa käytettävissä olevia ratkaisumalleja.

1. Suoritusalustariippumattomuus:

   (a) Käyttöliittymäkonseptin tulisi olla riippumaton toteutusteknologiasta. Suunnitelman mukainen käyttöliittymä tulisi voida toteut-

taa graafisen työpöydän lisäksi ainakin tekstitilassa toimivana konsolisovelluksena.

(b) Käyttöliittymän tulisi olla siirrettävissä erilaisille käyttöjärjestelmille.

(c) Käyttöliittymän suunnittelussa tulisi noudattaa käyttöliittymätyypille vakiintuneita suunnittelusääntöjä.

2. Laajennettavuus ja uudelleenkäytettävyys:

(a) Käyttöliittymän suunnitelman tulisi olla mahdollisimman yleiskäyttöinen, jotta sillä voitaisiin toteuttaa monenlaisia käyttöliittymiä.

(b) Toteutettavan sovellusohjelmiston käyttöliittymän toteutuksen tulisi olla riippumaton käyttöliittymäohjelmiston kehitysympäristöstä.

(c) Toteutettuja sovellusohjelmistoja tulisi voida laajentaa ilman ohjelmointityökalujen käyttöä.

(d) Toteutettuja sovellusohjelmistoja tulisi voida uudelleenkäyttää tulevissa tutkimusprojekteissa.

(e) Sovellusohjelmiston tulisi tarjota riittävät työkalut kaikille tutkimustyön työvaiheille.

3. Käyttäjän huomioiminen:

(a) Uuden sovellusohjelmiston prototyypin valmistukseen kuluvan ajan tulee olla huomattavasti varsinaisen sovelluksen toteutukseen kuluvaa aikaa lyhyempi. Tällöin epävarmoja ideoita voidaan varmentaa koeperäisesti.

(b) Sovellusohjelmiston tulisi mahdollistaa käyttäjän kehitys aloittelevasta käyttäjästä kohti ohjelmoijaa, joka kykenee työstämään työkalusta mieleisensä.

2.3 Ei-toiminnalliset vaatimukset

Ei-toiminnallisilla vaatimuksilla täydennetään edellistä luetteloa vaatimuksilla, jotka lähinnä parantavat ohjelmiston käytettävyyttä.

1. Käyttöliittymästä tulee toteuttaa ainakin graafinen versio.

2. Käyttöliittymän toteutus tulee suorittaa sellaisilla komponenteilla, että toteutettavien sovellusohjelmistojen suorituskyky ei vaarannu tavallisilla mikrotietokoneilla.

3. Käyttöliittymän tulee olla riittävän vakaa, jotta sillä voidaan suorittaa sovellusohjelmien kehitystä ilman työtä hidastavia ohjelmiston kaatumisia ja lukkotilanteita.

2.4 Rajoitteet

Rajoitteilla määritellään ne toimintarajat, joiden puitteissa ohjelmiston kehitys tulee toteuttaa.

1. Ohjelmiston kehityksen tulee työmäärältään soveltua opinnäytteeksi.

2. Ohjelmiston toteutus tulee suorittaa sellaisilla kehitystyökaluilla, jotka voidaan hankkia pienin kustannuksin.

3. Ohjelmistoa on tarkoitus jakaa ilmaisena esimerkkisovelluksena, jolloin kaupalisen ja salaisen materiaalin käyttö ei käytännössä ole mahdollista.

Luku 3

Kirjallisuuskatsaus

Edellisessä luvussa suoritettu vaatimusmääritys luo motivaation tutustua ohjelmistoalan kirjallisuudessa esitettyihin ratkaisumalleihin. Tavoitteena on löytää valmiita ja hyväksi havaittuja ratkaisumalleja, joista suunnitteluvaiheessa valitaan työn toteutukseen soveltuvimmat. Valmiita ratkaisuja hyödyntämällä pyritään vähentämään sitä riskiä ja työmäärää, joka sisältyy täysin omaan toteutukseen perustuviin ohjelmistoihin.

3.1 Vaatimusmäärittelyn käsitteitä

Tässä alaluvussa määritellään tarkemmin vaatimusmäärittelyssä esiinnostettuja ohjelmistotuotannollisia käsitteitä. Määrittelyssä noudatetaan ohjelmistotuotannon peruskirjallisuuteen kuuluvien I. Sommerville:n [4], R. S. Pressman:in [5] ja H. Vlient:in [6]) viitoittamaa tietä. Peruskirjallisuudesta saatuja tietoja täydennettiin paikallisesta kirjastosta löytyneillä P. J. Brown:in [7], Peter J. L. Wallis:in [8] ja Pamela Gray:n [9] aihetta tarkemmin käsittelevillä teoksilla.

3.1.1 Siirrettävyys

> *Portability - the ease with which a system or component can be transferred from one hardware or software environment to another.*
>
> *The Institute of Electrical and Electronics Engineers, Inc. [10]*

Siirrettävyydellä tarkoitetaan sitä helppouden astetta, jolla järjestelmä tai komponentti voidaan siirtää uuteen laitteisto- tai ohjelmistoalustaan. Tässä

työssä siirrettäväksi katsotaan sellainen ohjelmisto, jonka siirrosta aiheutuvat kustannukset ovat uuden vastaavan ohjelmiston luontia pienemmät.

Ohjelmiston siirrettävyyden vertailemiseksi täytyy ensin määritellä siirrettävyyttä mittaava indikaattori. Koska ohjelmistoalalle ei ole vakiintunut standardia tapaa vertailuun, käytetään tässä työssä J. D. Mooneyn [12] määrittelemää laskukaavaa.

$$DP = 1 - (\textit{'cost to port'}/\textit{'cost to redevelop'}) \tag{3.1}$$

Kaavalla laskettava siirtyvyyden indikaattori liikkuu välillä [0,1], jossa lukuarvon nolla voidaan katsoa edustavan täydellistä siirtymättömyyttä ja lukuarvon yksi täydellistä siirtyvyyttä. Käytännössä ohjelmistojen tuotannossa saavutettavat arvot sijoittuvat jonnekin näiden lukuarvojen väliin ääripäiden jäädessä lähinnä teoreettisiksi vaihtoehdoiksi.

Siirrettävyyden merkitys ohjelmistojen tuotannossa on kasvussa. Viimeisten viidentoista vuoden aikana markkinoita on hallinnut Intellin-Microsoftin luoma x86-DOS-Windows arkkitehtuuri, joka loi riittävät toimintapuitteet ohjelmistojen tuottajille. Vaikka kyseinen arkkitehtuuri ei ollutkaan avoin, oli se silti riittävän yhtenäinen ja stabiili ohjelmistojen tuottajien ja käyttäjien tarpeisiin. Arkkitehtuuri on saavuttanut suuren suosion ja siitä on muodostunut toimialan kipeästi kaipaama standardi. Menestyksen pohjan muodosti arkkitehtuurin luoma siirrettävyys, joka ei tosin perustunut ohjelmistojen siirrettävyyteen, vaan laitteistoalustan homogeeniseen toteutukseen.

Mahdollisuus käyttää samoja sovelluksia lähes kaikkien valmistajien laitteistoissa on saanut suurimman osan sekä laitteistojen, että ohjelmistojen tuottajista hylkäämään vaihtoehtoiset teknologiat ja keskittämään kaikki voimansa tälle päämarkkina-alueelle. Laitteistojen monimuotoisuus on kuitenkin taas kasvussa. Käyttäjille on tarjolla monenlaisia kannettavia PDA-laitteita ja älypuhelimia ja näiden tarjonnan uskotaan vielä kasvavan. Kannettavissa laitteissa käytettävät resurssit ovat aina niukemmat kuin työasemissa ja tämä on huomioitava myös ohjelmistojen suunnittelussa. Palvelinpuolella ollaan viimeinkin siirtymässä lopullisesti 64-bittiseen aikakauteen ja sitä myötä uudentyyppisiin teknisiin ratkaisuihin. Tulevaisuudessa markkinoilla onkin yhden yhtenäisen alustan tilalla vähintään kolme hieman erityyppistä alustaa. Nykyinen markkinajohtaja Microsoft on jo esitellyt tuotteita kaikille kyseisille alustoille, joissa on hyödynnetty kunkin alustan erityisluonnetta. Näin on samalla saavuttu tilanteeseen, jossa ohjelmistoista on pian taas tuotettava useita versioita riippumatta Intel-Microsoft -liittouman menestyksestä ohjelmistomarkkinoilla.

Binäärisessä siirrettävyydessä ohjelmisto käännetään sellaiseen konekieliseen muotoon, jota voidaan ajaa sellaisenaan kaikissa kohderyhmän alus-

toissa. Jos kaikissa kohderyhmän alustoissa ei ole käytössä samanlaista laitteistoa, ei menetelmällä toteutettuja ohjelmistoja yleensä voida toteuttaa suoraan minkään alustan konekielellä. Sen sijaan voidaan muodostaa matalan tason binääriesitys, joka voidaan formaalisti muuntaa pienellä vaivalla konekieliseen esitykseen tai vaihtoehtoisesti suorittaa tulkkaamalla. Menetelmässä ohjelmistojen siirretävyys toteutuu siis jo valmistusvaiheessa ja siten käyttäjän kannalta ohjelmiston asennus ja käyttö on kaikilla alustoilla yhtä helppoa. Virtuaalikoneisiin pohjautuvat ohjelmistot ovat luonnollisesti korkeitaan yhtä vakaita kuin suoritusalustana toimiva virtuaalikone. Kaikkein suurinta vakautta vaativiin järjestelmiin nykyiset virtuaalikoneet eivät siten välttämätä sovellu.

Lähdekoodeihin perustuvassa siirrettävyydessä ohjelmiston lähdekoodit laaditaan sellaisiksi, että ne voidaan joko suoraan tai pienin muutoksin kääntää kyseiselle alustalle sopivaan muotoon. Noudattamalla kurinalaisuutta lähdekoodeja toteutettaessa saadaan nykyisillä korkean tason kielillä tuotettua jo varsin hyvin järjestelmästä toiseen siirtyviä ohjelmistoja. Varsinainen siirto suoritetaan kääntämällä käyttäjän toimesta ohjelmistosta uusi versio käytetylle alustalle. Menetelmän hyvänä puolena on laaditun ohjelmiston hyvä suorituskyky johtuen konekielisestä toteutuksesta. Koska menetelmä vaatii ohjelmiston lähdekoodien luovuttamisen käyttäjille, on se yleistynyt lähinnä vapaasti levitettävien ohjelmistojen keskuudessa. Täydellisen siirrettävyyden saavuttaminen lähdekoodeilla on yleensä mahdotonta, jolloin siirrosvaiheessa joudutaan suorittamaan sopeuttamistoimenpiteitä. Tehtävää voidaan helpottaa rajaamalla jo toteutusvaiheessa alustakohtaiset toteutukset omiin moduleihinsa, joissa niiden sopeuttaminen voidaan suorittaa keskitetysti. Oikein toteutettuina tällaiset modulit muodostavat yleiskäyttösiä ohjelmistokomponentteja, joita voidaan uudelleenkäyttää myös muissa vastaavan toimialan ohjelmistoprojekteissa. Jos ohjelmistossa käytetään sellaisia alustan tuottamia palveluita, joita ei löydy kaikista kohderyhmän laitteistoista, on modulijakoon kiinnitettävä erityistä huomiota, sillä ohjelmiston on mahdollisesti kyettävä tuottamaan osa toiminnoistaan ilman kyseisiä palveluita.

Ohjelmiston siirrettävyyteen vaikuttavat ratkaisevasti ohjelmistoprosessin alkuvaiheessa suoritetut valinnat. Jos työn alkuvaihe on suoritettu väärin, ei ohjelmistolla aina saavuteta kaikkia sille asetettuja tavoitteita. Tällöin on suoritettava valinta ohjelmiston ominaisuuksien ja siirrettävyyden välillä. Valintaa ei kuitenkaan tarvitse suorittaa täysin joko-tai -pohjalta, vaan yleensä parhaaseen lopputulokseen päästään miettimällä kompromissi, jossa kummankin osa-alueen painopisteitä on toteutettu parhaan kyvyn mukaan.

Määrittelyvaiheessa on syytä ottaa heti kantaa siihen, kuinka kattavaa siirrettävyyttä tavoitellaan. Käytännön toteutuksissa päädytään yleensä

vain osittaiseen siirrettävyyteen. Ylimalkaiset vaatimukset ohjelmiston helposta siirrettävyydestä voidaan suosiolla jättää kirjaamatta vaatimusmäärittelyyn. Sen sijaan tulisi määritellä tarkemmin minkälaisilla alustoilla ohjelmistoa on tarkoitus käyttää. Tällöin voidaan määritellä vaadittavaa suorituskykyä keskeisimmille resursseille, kuten muistille, IO- ja aritmeettiselle yksikölle. Määrittelyssä tulisi välttää nimeämästä nykyisiä alustoja, sillä markkinoille voi tulla uusia samat suoritusarvot sisältäviä alustoja, joiden ilmestymistä ei voida ennakoida. Sen sijaan alustat kannattaa jakaa suurpiirteisesti alustatyyppeihin, jota sovelletaan määrittelyyn.

Määrittelyvaiheessa tulee lisäksi pyrkiä välttämään siirrettävyyden kanssa suoranaisessa ristiriidassa olevia tavoitteita. Tällaisia ovat esimerkiksi liian tarkat määritelmät siitä, miten joku tietty toimenpide tulee suorittaa. Käyttäjän kannalta lopputulokset ovat usein itse työskentelyä tärkeämpiä ja siten ohjelmiston toteutukseen voidaan jättää liikkumavaraa. Nykyisissä laitteistoissa on olemassa monenlaisia ohjaimia ja liian tarkka käyttöliittymän kuvaus voi estää ohjelmiston siirron sellaisiin alustoihin, joihin kyseisen kaltaista ohjausjärjestelmää ei ole saatavilla. Konkreettisen esimerkin muodostavat Macintosh-järjestelmässä käytettävä yksinkertainen hiiri verrattuna Microsoftin-älyhiiriin, joissa on huomattavasti enemmän toiminnallisuutta. Esimerkin tapauksessa rullahiirelle suoritetut määritelmät eivät ole siirtyviä. Ohjelmiston määritelmässä tulisi myös välttää liian tarkkaa laitteiston toiminnan kuvausta. Useimmissa matemaattisissa ongelmissa on tarkoituksenmukaisempaa ilmoittaa tarvittava lukualue sen sijaan että ilmoitettaisiin suoraan käytettävä bittimäärä, jolloin osa laitteistoista ei välttämättä pysty toteuttamaan määritelmää.

Suunnitteluvaiheessa on kiinnitettävä huomiota määriteltyjen tavoitteiden toteutumiseen. Modulijaossa on huomioitava tavoite eristää alustakohtaiset palvelut omiksi moduleikseen. Toteutustekniikoita valittaessa on syytä suosia standardeja toteutusmenetelmiä, ellei määrittelyvaiheessa ole erityisvaatimuksia, jotka pakottavat käyttämään jotain tiettyä erikoispalvelua. Yleisimmistä standardeista voidaan mainita ASCII-merkistö, POSIX-järjestelmäpalvelut ja IEEE-standardin mukainen liukulukuaritmetiikka.

Toteutusvaiheessa muunnetaan laadittu suunnitelma ohjelmiston lähdekoodeiksi. Hyvä suunnitelma on helposti siirrettävissä erilaisille alustoille, vaikka siinä ei olisikaan huomioitu siirrettävyyden erityisteemoja. Tuotettu lähdekoodi sen sijaan ei välttämättä ole siirtyvää ja siten suunnitelma on toteutettava kullakin alustalla erikseen. Tuotantovaiheessa ohjelmistoa kehitetään yleensä vain yhdellä alustalla ja vasta tietyn valmiusasteen saavutettuaan sitä ryhdytään siirtämään muille kohdealustoille. Toteutuksessa käytettävä ohjelmointikieli on syytä valita huolella, sillä väärin suoritettu valinta voi johtaa ohjelmiston siirrettävyyden ja jatkokehityksen ajautumiseen um-

pikujaan. Kielen valintaan vaikuttavat sen saatavuus kaikille kohdealustoille ja sen soveltuvuus määriteltyjen ominaisuuksien toteuttamiseen. Varmimpina vaihtoehtoina voidaan pitää pitkään käytössä olleita standardoituja kieliä, kuten Ada, C, C++ ja Java. Varsinaisen ohjelmointikielen lisäksi tarvitaan monissa ohjelmistotyypeissä myös muita kieliä, joilla toteutetaan esimerkiksi komponenttien hallintaa ja tietokantojen ylläpitoa. Kehitystyökalua valittaessa tulee huomioida, että suurin osa kehitystyökalujen tuottajista sisällyttää niihin epästandardeja lisäpalveluita, joita tulisi työkalun vaihdettavuuden takia tietysti välttää.

Testausvaiheeseen sisältyy tavallisten toimenpiteiden lisäksi erityinen siirtyvyyden testaus, jolla varmistetaan ohjelmiston toiminnan oikeellisuus kaikilla kohdealustoilla. Tärkeintä on ohjelmiston helppo ja yhtenäinen siirrettävyys kaikille käyttöalustoille, jolloin siirtämisiin ja erillisten versioiden ylläpidollisiin tehtäviin kuluu huomattavasti vähemmän aikaa. Siirrettävyyden varmistamista varten voidaan laatia uudelleenkäytettäviä testisuunnitelmia. Ylläpitovaiheessa voidaankin sitten nauttia siirrettävyyden hedelmistä. Siirrettävyyden aikaansaamiseksi tehty huolellisempi suunnitelu auttaa yleensä myös tavallisissa ohjelmiston ylläpitoon liittyvissä ongelmissa.

3.1.2 Yhteensopivuus

> *Compatibility - the ability of two or more systems or components to perform their required functions while sharing the same hardware or software environment.*
>
> *The Institute of Electrical and Electronics Engineers, Inc. [10]*

Yhteensopivuudella tarkoitetaan järjestelmien ja ohjelmistojen kykyä toimia yhteistyössä yhteisen päämäärän saavuttamiseksi. Yhteensopivuus on aina ollut käyttäjien suurimpia toiveita ohjelmistojen osalta, mutta toistaiseksi tässä on onnistuttu vain välttävästi.

Yhteensopivuus on aina ollut kaksiteräinen miekka kaupallisille ohjelmistojen tuottajille. Toisaalta valmistajat haluavat tehdä ohjelmistoistaan yhteensopivia kilpailevien tuotteiden kanssa, jotta näiden käyttäjät voisivat siirtyä heidän asiakkaikseen. Uuden asiakkaan saatuaan yrittää ohjelmiston tuottaja luonnollisesti pitää hänet asiakkaana mahdollisimman kauan. Varmin tapa säilyttää asiakas on tarjota hänelle sellaisia lisäominaisuuksia, joita hän työssään tarvitsee, mutta joita ei löydy kilpailijoiden tuotteista. Tilanne on johtanut eräänlaiseen kilpavarustelun kierteeseen, jossa kaikki ohjelmistojen tuottajat pyrkivät yhtäaikaa tarjoamaan yhteensopivuutta kilpailevien

tuotteiden kanssa ja samalla tarjoamaan kilpailijoitaan runsaampia ominaisuuksia.

Käyttäjät ottavat uudet ominaisuudet yleensä innolla vastaan, merkitseväthän parannukset heille usein huomattavaa vaivojen säästöä. Varomaton käyttäjä tekee kuitenkin näin itsensä riippuvaiseksi kyseisestä tuotteesta, kunnes kilpailijat taas lisäävät vastaavat ominaisuudet omiin ohjelmistoihinsa. Vuosien kuluessa kertyvä työmäärä muodostaa yleensä ylipääsemättömän esteen käyttäjän halutessa siirtyä käyttämään korvaavaa ohjelmistoa.

Nopeasti kehittyvillä markkinoilla on usein ajauduttu tilanteisiin, joissa edes samaa ohjelmistoperhettä olevat eri ohjelmistoversiot eivät ole toistensa kanssa yhteensopivia. Tästä aiheutuu ongelmia erityisesti yrityksille, joiden on käytännön syistä pyrittävä pitämään ohjelmistokantansa mahdollisimman yhtenäisinä. Ongelman ratkaisemiseksi on laadittu lukuisia muuntimia, joilla laadittuja tiedostoja voidaan muuntaa toisille ohjelmistoille soveltuviin muotoihin. Muunnosten teko on useinmiten kuitenkin ongelmallista ja yleensä aina menetetään pienimuotoisia, lähinnä tiedon esitysasuun vaikuttavia, yksityiskohtia. Ongelman kiertämiseksi ovat suurimmat kaupallisten ohjelmistojen tuottajat niputtaneet ohjelmia yhdeksi paketiksi, joka muodostaa tavallaan yhtenäisen ohjelmiston ja on siten sisäisesti yhteensopiva.

Ohjelmistojen yhteensopivuutta voidaan parantaa lähinnä uusilla standardeilla, joiden käyttöön mahdollisimman moni ohjelmistojen tuottajista sitoutuu. XML-standardin kehitys antaa hyvän kuvan niistä tuloksista, joita yhteisellä ohjelmistorintamalla voidaan saavuttaa. Vielä suurelta osaltaan suunnitteluvaiheessa oleva standardi luo yhteiset pelisäännöt rakenteellisen informaation kuvaamiseen. Yleistyessään se parantaa oleellisesti ohjelmien kykyä käsitellä muiden ohjelmistojen tiedostomuotoja.

3.1.3 Käytettävyys

> *Usability - the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component.*
>
> *The Institute of Electrical and Electronics Engineers, Inc. [10]*

Käytettävyydellä mitataan kuinka helppoa käyttäjien on oppia käyttämään uusia ohjelmistoja ja hyödyntämään niitä työssään. Ohjelmiston käyttöä varten täytyy käyttäjän osata muotoilla ohjelmiston ymmärtämä kuvaus työtehtävästään. Syötteen perusteella ohjelmisto antaa yleensä palautetta. Käyttäjän on ymmärrettävä saamansa palaute, jotta hän voi saavuttaa hyötyä ohjelmiston käytöstä. Vaikka tavoitteena yleensä pidetäänkin käyttäjän ehdoilla

toimivia ohjelmistoja, mittaa nykyinen määritelmä käyttäjän kykyä omaksua ohjelmiston taustalla oleva toimintamalli mahdollisimman tehokkaasti. Ideaalisessa tilanteessa varsinaista oppimista ei tarvittaisi ollenkaan eri ohjelmistojen muistuttaessa niin paljon toisiaan, että yhdestä ohjelmistosta saatu kokemus riittäisi myös muiden sovellusten käyttöön.

Käytettävyyttä pidetään usein aloittelevien käyttäjien ongelmana, mutta yhtälailla huono käytettävyys vaivaa myös edistyneempiä käyttäjiä. Siinä missä aloittelijat tuskailevat tavallisten dokumenttien vaikeaa työstettävyyttä, on toisessa ääripäässä tiedemiehillä vaikeuksia pukea ajatuksiaan ohjelmistojen ymmärtämään muotoon. Molemmissa esimerkeissä on ongelmilla sama perimmäinen syy. Ihmisten ymmärtämä maailmankuva on tyystin erilainen, kuin mitä ohjelmistoilla voidaan toteuttaa. Ohjelmistoista voidaan toki tehdä yhä paremmin reaalimaailmaa jäljitteleviä, mutta tällöin on vaarana käyttöliittymän muuttuminen yhä tehottomammaksi, jolloin käyttö on toki helpompaa, mutta samojen toimenpiteiden aikaansaamiseen voi kulua myös pidempi aika.

Informaatioteknologian alalla koulutusmenot ovat aina muodostaneet huomattavan osan kokonaiskustannuksista. Varsinkin ohjelmistoteollisuuden ulkopuolella työskentelevien käyttäjien kynnys jatkuvaan kouluttautumiseen on korkea ja siten ohjelmistojen kirjavien ja jatkuvasti muuttuvien käyttöliittymien katsotaan vaikeuttavan tarpeettomasti ohjelmistojen käyttöä. Ongelmaa voidaan helpottaa vain luomalla entistä yhtenäisempään sääntöpohjaan perustuvia käyttöliittymiä. Jossain määrin tässä on jopa onnistuttu, sillä Microsoft Windows -ohjelmistot muistuttavat yhä enemmän toisiaan. Erillisten käyttöjärjestelmien käyttöliittymien toimintaperiaatteet poikkeavat kuitenkin niin paljon toisistaan, ettei samaa yhtenäisyyden tasoa voida saavuttaa siirrettävien ohjelmistojen tapauksissa.

Suurimmat ohjelmistojen tuottajat ovat jo vuosien ajan suorittaneet erityisiä käytettävyystestejä, joilla pyritään mittaamaan käyttäjien tyytyväisyyttä heidän tuottamiensa ohjelmistojen käyttöliittymiin. Tutkimuksista on saavutettu arvokasta tietoa ihmisen tavasta oppia ja käyttää tietokoneita. Ihmiselle ominaisen kuvamuistin johdosta ovat graafiset käyttöliittymät osoittautuneet huomattavasti komentopohjaisia liittymiä intuitiivisemmiksi. Lisäksi ihmisten muiden aistien, kuten kuulon hyväksikäytöstä on saatu hyviä kokemuksia. Tutkimustuloksilla on ajankuluessa onnistuttu parantamaan yleisimpien ohjelmistojen käytettävyyttä. Valitettavasti yksityisellä rahalla tuotetujen tutkimusten tulokset on yleensä pidetty salassa, eikä niillä siten ole ollut koko alaa edistäviä vaikutuksia.

3.1.4 Laajennettavuus

> *Extendability - the ease with which a system or component can be modified to increase its storage or functional capacity.*
>
> *The Institute of Electrical and Electronics Engineers, Inc. [10]*

Laajennettavuudella tarkoitetaan sitä helppouden astetta, jolla ohjelmistoa voidaan laajentaa suorittamaan sellaisia toimenpiteitä, joita alkuperäisellä versiolla ei voida suorittaa. Laajennettavuutta hyödyntävät lähinnä pidemmälle edistyneet käyttäjät, joiden tarpeet ylittävät markkinoilta löytyvien valmiiden ohjelmistopakettien tarjonnan.

Yleisin tapa ohjelmistojen laajentamiseen on käyttää niiden tarjoamaa makrokieltä. Vaikka makrolaajennusten teko on yleensä mahdollista vain suurimpien ohjelmistojen kohdalla, muodostaa se näissä varsin käyttökelpoisen menetelmän. Koska makrokielet ovat luonteeltaan systeemiohjelmointikieliä yksinkertaisempia, voivat tavalliset käyttäjätkin tuottaa pienellä vaivalla yksinkertaisia, mutta yleensä silti varsin hyödyllisiä laajennuksia. Yleisin tapa tuottaa makrotiedostoja nykyisillä graafisilla sovelluksilla on nauhoittaa käyttäjän syötteitä. Uusi lähestymistapa mahdollistaa ohjelmoinnin ilman ohjelmointikielten tuntemusta ja siten laajentaa potentiaalisen ohjelmoijakunnan kattamaan lähes kaikki edistyneemmät käyttäjät.

Monista eduistaan huolimatta makrolaajennukset eivät ole yleistyneet toivotulla tavalla. Makroteknologiaa vaivaa paha standardien puute. Lähes kaikki ohjelmistot toteuttavat vain niille tarkoitettuja makrotiedostoja ja siten makrolaajennusten siirtäminen ohjelmistojen välillä on vielä dokumenttien siirrettävyyttäkin vaikeampi ongelma. Lisäksi nykyiset makrovirukset ovat saaneet käyttäjät varuilleen ja makrojen suoritus onkin perusasetuksilla estetty lähes kaikissa ohjelmistoissa.

Suurempien laajennusten toteuttamiseen makrokielet ovat yleensä riittämättömiä ja tällöin joudutaan turvautumaan perinteisiin ohjelmointikieliin. Makrokielet soveltuvat olemassa olevien palveluiden yhdistelemiseen uusilla tavoilla, mutta niiden kuvaus- ja suorituskyky eivät riitä esimerkiksi algoritmien toteuttamiseen. Systeemiohjelmointikielillä toteutettavaa ohjelmistojen laajennusta rajoittaa varsinkin kaupallisten ohjelmistojen osalta lähdekoodien heikko saatavuus. Vapaasti kehitettävien ohjelmistojen osalta tilanne on siinä mielessä parempi, että lähdekoodit ovat yleensä kaikkien halukkaiden saatavilla ja ohjelmistojen jatkokehitykseen suorastaan kannustetaan. Vapaasta kehitysilmapiiristä johtuen lähdekoodien dokumentointi on kuitenkin usein olematonta ja siten laajennusten toteuttamiseen tarvittavan tietotaidon hankkiminen on vaikeaa.

Suurinta osaa ohjelmistoista voidaan laajentaa ominaisuuksien lisäämisen ohella tuottamalla ohjelmiston käyttämää oheismateriaalia, kuten kuvakirjastoja ja mallipohjia. Ohjelmistoalalla olisikin runsaasti tilaa perinteisistä ohjelmistontuottajista riippumattomille sisällön tuottajille. Vapaan kilpailuasetelman saavuttamiseksi alalle olisikin saatava pikimmiten yhtenäinen standardi.

3.1.5 Uudelleenkäytettävyys

> *Reusability - the degree to which a software module or other work product can be used in more than one computing program or software system.*
>
> *The Institute of Electrical and Electronics Engineers, Inc. [10]*

Uudelleenkäytöllä tarkoitetaan ohjelmistojen kehitykseen käytetyn työpanoksen hyödyntämistä jollakin tavalla myös tulevissa projekteissa.

Perinteisimmillään uudelleenkäyttö on monien ohjelmointityökalujen mukana toimitettavien aliohjelmakirjastojen hyödyntämistä omissa ohjelmistoprojekteissa. Kirjastoihin on koottu runsaasti sellaisia aliohjelmia, joita ohjelmoijat tarvitsevat säännöllisesti työssään riippumatta millaista ohjelmistoa he ovat tuottamassa. Kirjastot ja myöhemmin niiden lisäksi ilmestyneet luokkakirjastot ovat lisänneet ohjelmoijien tuottavuutta ja parantaneet samalla ohjelmistojen luotettavuutta. Valitettavasti kirjastojen tuottaminen ei ole yleistynyt odotetulla tavalla ja siten ohjelmoijien käytettävissä ovat lähinnä kehitysympäristön mukana toimitetut kirjastot.

Olio-ohjelmoinnin yleistymisestä lähtien on uudenlaisiin yleiskäyttöisiin komponentteihin perustuvan ohjelmistotuotannon uskottu ennenpitkään korvaavan nykyisen tee-se-itse mallin. Koska tarvittavat työkalut ovat olleet saatavilla jo yli kymmenen vuotta, eikä standardeihin pohjautuvan komponenttimallin yleistyminen ole vieläkään tapahtunut, on olio-ohjelmointiin ratkaisevana tekijänä alettu suhtautua yhä skeptisemmin. Esimerkkinä tällaisesta ajattelusta voidaan pitää BYTE-lehdessä [13] julkaistua artikkelia, jossa väitetään olio-ohjelmoinnin epäonnistuneen muuttamaan ohjelmistojen tuottajien tapaa tuottaa uusia ohjelmistoja. Vaikka alalle ei olekaan muodostunut yleistä standardia, antavat nykyiset nopean kehityksen työkalut, eli niin sanotut RAD-työkalut, esimakua siitä, millaista ohjelmistojen tuottaminen tulevaisuudessa voi olla. Parhaimmilla tarjolla olevilla työkaluilla voidaan saavuttaa jopa kymmenkertainen tuottavuuden parannus verrattuna tee-se-itse malliin.

RAD-työkalut pohjautuvat pitkälle uudelleenkäytettävien komponenttien luomiseen ja käyttöön. Perinteisesti komponentteja on käytetty käyttöliittymien toteuttamiseen, mutta yhtä lailla niitä voidaan käyttää minkä tahansa ohjelmiston osan tuotantoon. Malliesimerkiksi RAD-työkalun toteutuksesta voidaan ottaa Borland Corp.:in valmistama Delphi, josta löytyy valmiit komponentit käyttöliittymän, tietokantapalveluiden ja uusimpana ominaisuutena verkkopalveluiden toteuttamiseen. Lisäksi uusien komponenttien laatiminen on toteutettu mallikkaasti, jolloin ohjelmoija tuottaa lähes huomaamattaan uudelleenkäytettäviä ohjelmistoja. Työkalun kruunaa pitkälle viety graafinen työympäristö, jossa suuri osa ohjelmointityöstä voidaan toteuttaa ilman varsinaisen lähdekoodin kirjoittamista.

Kaiken tämän ohella RAD-työkalujen käytöllä on myös huomattavasti huonoja puolia. Johtuen RAD-työkalujen helposti etenevästä ohjelmiston toteutusvaiheesta on vaarana suunnittelun jääminen liian vähäiseksi, jolloin ohjelmiston kasvaessa törmätään erilaisiin suunnitteluvirheisiin. Lisäksi useimmat RAD-työkalut ovat rajoittuneet vain tietylle toiminta-alueelle ja tiettyihin käyttöalustoihin. Ohjelmiston laajentaminen työkalun tukemien alustojen ulkopuolelle on osoittautunut käytännössä lähes mahdottomaksi.

Tulevaisuudessa ohjelmistojen rakenne saattaa muodostua nykyistä modulaarisemmaksi, mahdollisesti jopa komponenteista koostuvaksi. Tällöin olisi periaatteessa mahdollista käyttää ohjelmistonkehitysympäristön mukana toimitettujen komponenttien lisäksi myös suoranaisia paloja muista ohjelmistoista. Toteutuakseen tämä vaatisi nykyisten asenteiden ja lisenssointikäytännön muuttamista. Kaupalliset ohjelmistojen tuottajat ovat valmiita avaamaan tuotteensa uudelleenkäyttöön vain, jos heille voidaan taata oikeudenmukaiset korvaukset ohjelmisto-osuuksien käytöstä. Lisäksi hyödyntävien ohjelmistojen tulisi tarjota uusia ominaisuuksia, jolloin uudet ohjelmistot eivät olisi vain pelkkiä alkuperäisten kopioita. Vaikka lähdekoodien avoimeen jakamiseen tuskin päästään koskaan, olisi standardinmukaisiin komponentteihin pohjautuvien ohjelmistojen uudelleenkäyttö silti teknisesti mahdollista.

3.1.6 Elinkaari

> *Software life cycle - the period of time that begins when a software product is conceived and ends when the software is no longer available for use. The life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. These phases may overlap or be performed iteratively, depending on the*

> *software development approach used.*
>
> *The Institute of Electrical and Electronics Engineers, Inc. [10]*

Ohjelmiston elinkaari muodostuu sen toteuttamisesta ja käyttämisestä. Uusi elinkaari alkaa ohjelmiston suunnittelusta ja päättyy sen poistamiseen käytöstä. Usein elinkaari koostuu iteratiivisista vaiheista, joissa uutta ohjelmistoversiota toteutetaan vanhemman ollessa vielä käytössä.

Elinkaaret voidaan jakaa ohjelmiston kehityksen mukaan jatkuvasti eteneviin ja hypäyksittäin eteneviin elinkaariin. Näissä kummassakin ohjelmistoa kehitetään jatkuvasti, mutta ne eroavat toisistaan siinä miten kehitystyön tuloksia jaetaan loppukäyttäjille.

Monista vapaasti kehitettävistä ohjelmistoista on saatavilla lähes päivittäin uusia versioita. Koska kehitystyössä tapahtuu väistämättä virheitä, ei ohjelmien toiminnan oikeellisuudesta voidakaan olla varmoja, vaan jokaisen käyttäjän on oman harkintansa mukaisesti tehtävä valinta vähemmän ominaisuuksia sisältävien mutta koeteltujen ja uusien mutta epävarmenpien versioiden välillä. Päätöksen helpottamiseksi esitetään käyttäjille usein viimeisimmät versiot sekä vakaista, uusimmista ja kehitystyön alla olevista versioista. Malli tarjoaa edistynemmille käyttäjille paremmat mahdollisuudet ohjelmistojen hallintaan, mutta on toisaalta aivan liian monimutkainen ja työläs peruskäyttäjien hallittavaksi.

Kaupallisten ohjelmistotuotteiden levityksessä käytetään lähes poikkeuksetta hypäyksittäin etenevää mallia, jossa ohjelmistojen tuottajat kehittävät ohjelmistoa, kunnes se saavuttaa tietyn valmiusasteen, joka vastaa jatkuvan mallin vakaata versiota. Näin ollen ohjelmistoista ilmestyy uusia versioita huomattavasti harvemmin kuin jatkuvassa mallissa. Käyttäjien kannalta tämä on tietysti positiivinen asia, sillä jo nykyisellä noin kahden vuoden päivitysvauhdilla useimmat käyttäjät kokevat päivityksistä lähes yhtä paljon haittaa kuin hyötyä. Malli onnistuu siis jossain määrin luomaan alan tarvitsemaa vakautta. Toisaalta ohjelmistoissa mahdollisesti esiintyvät virheet vaivaavat käyttäjiä pidempään ja uusien ominaisuuksien käyttöönotto voi viivästyä vuosilla.

Elinkaaret voidaan jakaa myös ohjelmiston suunnitellun eliniän mukaisesti lyhyt- ja pitkäaikaisiin elinkaariin. Lyhytaikaisen elinkaaren mukaisia ohjelmistoja on tarkoitus käyttää vain määrätty aika, joka on yleensä enimmilläänkin vain muutamia vuosia. Pitkäaikaisen elinkaaren ohjelmistoissa ei käytöstäpoiston ajankohta ole välttämättä tiedossa.

Lyhytaikaisen elinkaaren ohjelmistoissa kehityskustannukset muodostuvat yleensä varsinaisia käyttökustannuksia suuremmiksi. Ongelmaa voidaan

lievittää vähentämällä ohjelmiston suunnittelua, sillä lyhyeksi jäävän käyttöajan seurauksena ohjelmistossa esiintyvistä epäkäytännöllisyyksistä ei aiheudu suurta vaivaa. Varsinaisia ongelmia onkin tiedossa vasta siinä vaiheessa, kun ohjelmiston käyttö osoittautuu odotettua suuremmaksi ja valmiiksi heikon rakennelman päälle tulisi rakentaa uusia ominaisuuksia.

Pitkäaikaisen elinkaaren ohjelmistot suunnitellaankin jo huolellisemmin ja niiden toteutukseen kulutetaan muutenkin huomattavsti enemmän resursseja. Ohjelmiston suunnitelmissa varaudutaan jo ennalta sellaisiin ominaisuuksiin, joita ohjelmistossa voidaan tarvita. Ohjelmistosta kehitetään hitaasti mutta varmasti uusia versioita, joissa korjataan edellisissä olleita virheitä ja lisätään uusia ominaisuuksia. Tavoitteena on rauhallinen kehitys kohti yhä parempaa ohjelmistotuotetta.

3.2 Vaatimusmäärittelyn huomioiminen ohjelmistoarkkitehtuurissa

Tässä luvussa tarkastellaan tarkemmin vaatimusmäärittelyn toteuttamiseen tarvittavia toimintamalleja.

3.2.1 Ohjelmistotekniset ratkaisut

Sopivilla ohjelmistoteknisillä ratkaisuilla voidaan kiertää jo ennalta monia ohjelmistotuotannossa esiintyviä karikoita. Tässä työssä esitellään muutamia yleisimmin käytetyistä ratkaisumalleista, joita on kuvattu I. Sommerville:n [4], R. S. Pressman:in [5] ja H. Vlient:in [6] kirjoittamissa ohjelmistotuotannon perusteoksissa.

Abstrahoiva ohjelmistorakenne

Abstrahoinnilla nostetaan ohjelmistojen tuotannossa käytössä olevien palveluiden abstraktiotasoa korkeammalle. Käytännössä tällä tarkoitetaan matalampien tasojen toteutuksen standardoivaa määrittelyä ja eristämistä, jolloin varsinaisia ohjelmistoja voidaan luoda puuttumatta jokaisessa ohjelmistossa samoihin matalan tason ongelmiin. Jos matalan ja korkean tason välinen liityntärajapinta määritellään hyvin, voidaan sekä alemman että ylemmän tason rakenteita muuttaa ilman, että toisella abstraktiotasolla oleviin ohjelmistojen palasiin on tarpeellista tehdä muutoksia.

Proseduraalisella toiminnalla tarkoitetaan hyvin määriteltyä keskitettyä toimenpidettä, jota edustaa toimenpiteen nimi. Esimerkiksi lauseessa "sulje

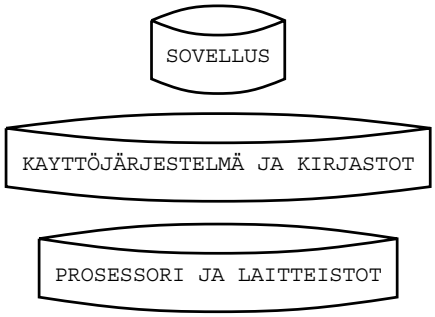
ikkuna", sana sulje muodostaa lauseen toiminnallisen osan. Ikkunan sulkeminen voi koostua erilaisista alitoimenpiteistä, mutta näiden toimenpiteiden eritteleminen ei ole lauseen ymmärtämisen ja toimeenpanon kannalta tarpeen.

Datan abstrahoinnilla tarkoitetaan käsitellyn data-aineiston määrittelyä ylemmän tason konsepteilla. Edellisessä esimerkissä ikkuna muodostaa lauseen toiminnan kohteen. Ikkuna voi koostua erilaisista alemman tason osista, kuten lasiruudusta, kehyksistä ja kahvasta. Lauseen ymmärtämiseen ja toimeenpanoon ei kuitenkaan ole tarpeellista ryhtyä erottelemaan näitä toisistaan, vaan lause on helpompi ymmärtää käsittämään toimintaa, joka kohdistuu kaikkiin aliosiin yhtenäisenä objektina.

Kontrollin abstrahoinnilla tarkoitetaan toiminnan ohjausta tuntematta ohjauksen toimintaperiaatteita. Esimerkiksi voidaan antaa lause: "kellon ollessa viisi illalla, sulje ikkuna". Tällöinkään ei ole lauseen ymmärtämisen kannalta oleellista tuntea kellon sisäistä toimintaa. Toimenpiteen oikeanlaiseen ajastamiseen riittää osata lukea kellosta nykyinen aika ja verrata sitä lauseessa mainittuun tapahtuma-aikaan.

Abstrahoimalla tuotettava ohjelmisto sopivasti, voidaan suuri osa ohjelmiston lähdekoodeista siirtää sellaisenaan uuteen ympäristöön. Sellaiset matalan tason toteutukset, joita ei voida implementoida standardeilla menetelmillä mukautetaan järjestelmäkohtaisiin vaatimuksiin ilman, että on olemassa tarvetta muuttaa korkeammalla tasolla sijaitsevaa sovelluslogiikkaa.

Esimerkkinä abstraktion käytöstä voidaan esittää malli, johon lähes kaikki nykyisistä tietokonejärjestelmistä pohjautuvat. Malli jakautuu kolmeen abstraktiotasoon kuvan 3.1 mukaisesti. Alimmalla tasolla sijaitsevat suoritusyksikkö ja muut laitteistotason ratkaisut. Keskimmäisen tason muodostavat käytetty käyttöjärjestelmä ja käytettävissä olevat aliohjelmakirjastot. Ylimmän tason muodostavat käytettävissä olevat ohjelmistot.



Kuva 3.1: Järjestelmän jako rakenteellisiin osiin.

Ohjelmistojen toteutuksessa ei mallin mukaisesti oteta kantaa laitteiston toimintaan, sillä erilaiset laitealustat muodostavat käyttöjärjestelmän abstrahoimana samanlaisen suoritusympäristön.

Modulaarinen ohjelmistorakenne

Ohjelmistotuotannossa esiintyvien laajojen ja vaikeiden ongelmien ratkaiseminen onnistuu usein helpoimmin jakamalla tehtävä osaongelmiin, sillä osaratkaisuiden vaativuus on käytännössä aina lähtökohtaa yksinkertaisempi. Sovellusohjelmoinnissa tällainen hajota ja hallitse- tekniikka toteutetaan jakamalla ohjelmiston toteutus moduleihin siten, että kuhunkin moduliin tulee sopiva määrä toiminnallisuutta. Ohjelmiston jakoa suoritettaessa on kiinnitettävä huomiota käytännön toteutettavuuteen. Pienten modulien toteutus voi olla helppoa mutta samalla niiden määrä kasvaa pahimmillaan eksponentiaalisesti ja siten säästetty aika kuluu modulien väliseen integrointiin. Liian suuret modulit eivät vastaavasti kykene hajottamaan ongelmaa riittävästi, jotta ongelman ratkaisu helpottuisi merkittävästi. Näin parhaaseen tulokseen päästäänkin ääripäiden puolivälissä, jossa saavutetut edut korostuvat edellä mainittujen haittojen pysyessä pieninä.

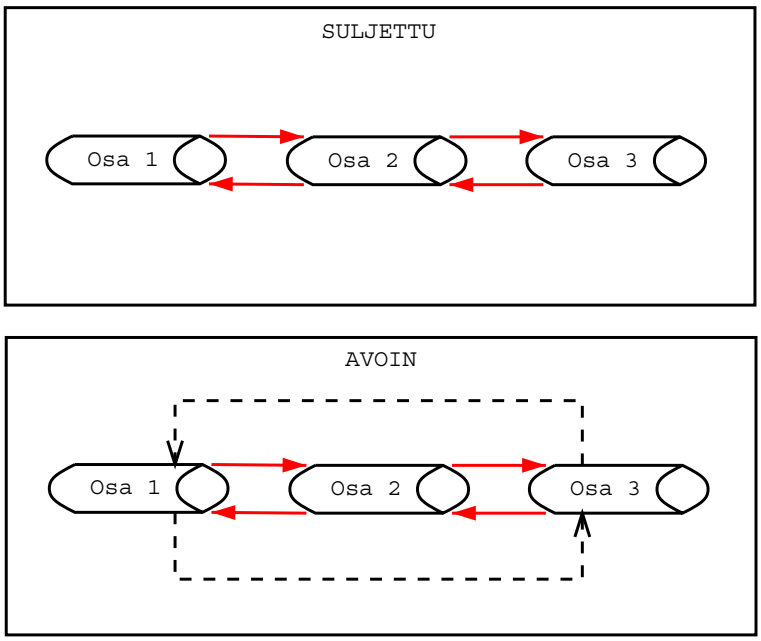
Partitioiva ohjelmistorakenne

Partitioinnilla pyritään jakamaan ohjelmiston toiminta osakokonaisuuksiin, joilla on tarkasti määritellyt rajapinnat toistensa suhteen. Partitiointi muodostaa modulijakoa huomattavasti karkeamman jakoperiaatteen ja sillä pyritään toisiin tavoitteisiin. Partitioinnilla pyritään siihen, että järjestelmän osakokonaisuuksia voidaan toteuttaa itsenäisesti ja vaihtaa vaivattomasti ilman, että koko järjestelmän toiminta vaarantuu. Partitioinnilla saavutetaan siis luonteva työnjako mahdollisesti erillisten työryhmien välille ja samalla mahdollistetaan myös ulkopuolisten toimittajien osajärjestelmien integroiminen ohjelmistoon.

Jakoa kutsutaan suljetuksi, jos jako-osat voivat keskustella vain niitä juuri edeltävän ja seuraavan osan kanssa. Avoimessa jaossa jako-osat voivat keskustella suoraan myös muiden osien kanssa. Suljettu jako on huomattavasti avoimempaa vakaampi, helpommin siirrettävissä, laajennettavissa ja ylläpidettävissä.

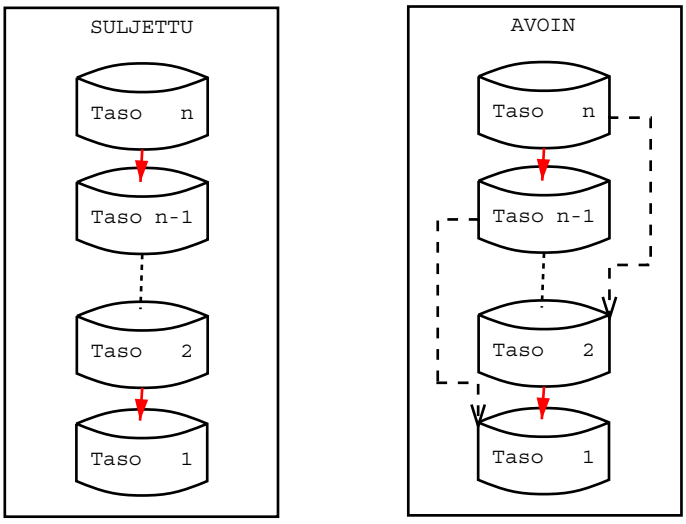
Horisontaalisessa partitioinnissa käytetään yleensä seuraavia jakoperusteita:

- Järjestelmä jaetaan toiminnallisiin kokonaisuuksiin siten, että kukin toiminto voidaan toteuttaa itsenäisesti, kunhan noudatetaan laadittua liityntärajapintaa.



Kuva 3.2: Horisontaalinen partitiointi.

- Esimerkiksi käy dataa käsittelevä analyysiohjelmisto, jossa tiedon keräys, käsittely ja tulostus muodostavat kukin oman kokonaisuutensa.



Kuva 3.3: Vertikaalinen partitionti.

Vertikaalisessa partitioinnissa käytetään yleensä seuraavia jakoperusteita:

- Järjestelmä ositetaan kerroksittain. Kukin osajärjestelmä käsittää koko järjestelmän toiminta-alan, mutta vain rajatulta osa-alueelta.
- Kerrokset muodostavat hierarkian, jossa ylemmät kerrokset perustuvat alempien kerrosten palveluista jalostamalla. Muutokset alempiin kerroksiin vaikuttavat siten myös ylempien kerrosten toimintaan.
- Alin kerros muodostuu yleensä laitteistoläheisistä ohjelmistoista, joita joudutaan muuttamaan siirryttäessä järjestelmästä toiseen. Välikerrokset järjestelevät ja kokoavat toimintoja siten, että niitä olisi helpompi käyttää ylemmillä tasoilla. Ylin kerros muodostaa varsinaisen sovelluksen toiminnan. Kerrosten määrä vaikuttaa ohjelmiston siirrettävyyteen. Mitä useammalle tasolle järjestelmä on jaettu, sitä tarkemmin siirtotyö voidaan kohdentaa.
- Hyvänä esimerkkinä vertikaalisesta partitionnista ovat protokollapinot, jossa ylemmän tason palvelut pohjautuvat alemman tason protokolliin. Tunnetuin protokollapinomalli lienee ISO:n määrittelemä OSI-malli, joka on samalla hyvä esimerkki suljetusta järjestelmäjaosta.

Käytännössä ohjelmistojen tuotannossa päädytään yleensä sekamuotoon, jossa järjestelmä jaetaan ensin horisontaalisesti itsenäisiksi osajärjestelmiksi, jonka jälkeen nämä osajärjestelmät jaetaan vielä vertikaalisesti toteutuksen yksinkertaistamiseksi.

3.2.2 Komponenttiteknologiat

Ohjelmistotuotantoa lukuunottamatta lähes kaikille muille insinööritieteiden aloille on muodostunut hierarkinen toimintamalli, jossa aikaisempien töiden tulokset muodostavat pohjan uusille yhä korkeamman tason omaaville töille. Aikaisempien töiden uudelleenkäyttö on mahdollistanut viime vuosikymmenten aikana nähdyn ennennäkemättömän nopean kehityksen. Ohjelmistojen tuotannossa uudelleenkäyttö on sen sijaan ollut vähäistä ja uuden ohjelmiston toteutus on perinteisesti aloitettu lähes puhtaalta pöydältä. Ohjelmistojen paisuttua valtaviksi järkäleiksi, on viimeisten kymmenen vuoden aikana työskennelty yhä kiivaammin käyttökelpoisen komponenttiarkkitehtuurin luomiseksi ohjelmistotuotantoon [14], [15], [16], [17], [18], [19]. Toistaiseksi mikään luoduista komponenttimalleista ei ole saavuttanut riittävää suosiota ja siten ongelma on vailla lopullista ratkaisua. Tuottavuuden merkittävän parantumisen ohella avoin komponenttimalli tarjoaa mahdollisuuden ohjelmistojen helpompaan siirrettävyyteen ja laajennettavuuteen.

Komponenttien pitkän historian aikana on komponenttien kuvaamiseksi ehditty esittää useita vaihtoehtoisia määritelmiä. Suurin osa ehdotuksista muistuttaa enemmän tai vähemmän Jun Han:in esittämää [20] komponentin määritelmää, jossa komponentti koostuu attribuuteista, operaatioista, rakenteellisista rajoitteista, operaatioiden rajoitteista, tapahtumista, rajapinnoista ja ei-toiminnallisista ominaisuuksista. Komponentti voidaan myös määritellä ohjelmiston rakenneosana, joka voidaan toteuttaa, myydä ja käyttää yhtenä kokonaisuutena. Kaikkia määritelmiä yhdistää niiden tavoite korvata perinteinen API-ohjelmistorajapinta [21] abstraktimmalla ja siten helppokäyttöisemmällä toimintamallilla.

Ohjelmistokomponenttien suunnittelussa voidaan käyttää apuna Markku Nykyn [22] I-Kineticsin määritelmästä suomentamia komponenttien perusominaisuuksia:

1. *Ympäristöriippumattomuus* - Komponentit voivat toimia keskenään ohjelmointikielistä, käyttöjärjestelmistä, verkoista ja kehitysympäristöistä riippumata.

2. *Sijainnin läpinäkyvyys* - Komponenttia voidaan käyttää läpinäkyvästi yhdestä prosessista, useasta samalla koneella ajettavasta prosessista tai useammasta eri puolilla verkkoa ja eri käyttöjärjestelmäalustoilla ajettavista prosesseista.

3. *Rajapinnan erottaminen toteutuksesta* - Rajapinta määrittelee komponentin toiminnallisuuden ottamatta kantaa siihen, kuinka tämä toiminnallisuus varsinaisesti toteutetaan.

4. *Rajapinnan kuvaus* - Komponentin tulisi kyetä kuvaamaan käyttämänsä rajapinta. Ominaisuutta tarvitaan lähinnä komponentin ajonaikaisessa sidonnassa.

5. *Alustariippumattomuus* - Komponentti voidaan asentaa ja sitä voidaan käyttää eri käyttöjärjestelmäalustoilla.

6. *Yleinen ohjelmistokomponenttikehys* - Komponentit voivat kommunikoida keskenään yhteisen ohjelmistokehyksen kautta.

Komponenttiteknologioita vertailtaessa on siirrettäviä ohjelmistoja laadittaessa tietysti kiinnitettävä erityistä huomiota käytettävien komponenttien siirrettävyyteen. Pearl Brereton ja David Budgen [23] ovat esittäneet vertailumenetelmää, jossa komponenttimalleja vertaillaan niiden modulaarisen rakenteen, toteutuskieliriippuvuuden ja suoritusalustariippuvuuden suhteen. Yleisimmät RAD-työkalut, kuten Delphi ja Visual Basic, saavat tässä vertailussa kehnon tuloksen. Niissä käytettävät komponenttijärjestelmät

ovat usein työkalukohtaisia, jolloin niitä voidaan käyttää vain kyseisen työkalun tukemilla kielillä ja niillä alustoilla, joille kyseinen työkalu on saatavilla. Vastoin yleistä käsitystä JavaBean:it ja CORBA-komponentit eivät täysin toteuta siirrettävyyden ehtoja. JavaBean:it ovat riippuvaisia Java-kielestä. CORBA-komponenteihin perustuvia ohjelmistoja voidaan tuottaa monilla kielillä, mutta valmiit CORBA-komponentit rajoittuvat toiminnalliselta toteutukseltaan usein yhden alustan sisäiseen käyttöön. Vasta yhdistämällä molemman edellä mainitut tekniikat luomalla CORBA-standardin mukaisia komponentteja Java-kielellä voidaan saavuttaa tavoitteeksi asetettu siirrettävyys.

Komponenttipohjainen sovellusohjelmointi sopii hyvin yhteen ohjelmistotuotannossa nykyisin käytettävien suunnittelumenetelmien kanssa. Komponenttipohjainen sovellus voidaan suunnitella käyttäen UML-pohjaista kuvauskieltä, jolloin sovelluksen toteutus tapahtuu suunnitelman lähes formaalilla muunnoksella. Ohjelmiston suunnitelussa voidaan käyttää esimerkiksi Philippe Kruchten [24] esittelemää mallia komponenttien rajapintojen, toteutuksen ja riippuvuussuhteiden kuvaamiseen. Sovelluksen laajempaan kuvaukseen voidaan tarvittaessa käyttää arkkitehtuurin kuvauskieliä [25] ja moduulien välisten liitäntöjen suunnittelukieliä [26].

Omien ohjelmistokomponenttien kehittäminen ei yleensä ole tarpeen sillä markkinoilta on saatavissa lukuisa joukko valmiita komponenttikirjastoja. Komponenttimallilla pyritään nimenomaan vähentämään oman ohjelmistotyön osuutta sovelluksien tuotannossa. Parhaimmassa tapauksessa tarvittavat komponentit voidaan saada jopa ilmaiseksi sillä tietoverkkojen kautta voidaan hankkia monia harrastelijoiden toteuttamia komponenttikirjastoja. Näiden lisäksi on olemassa yrityksiä, jotka ovat erikoistuneet yleiskäyttöisten komponenttien valmistukseen. Kaupallisten komponenttien hankintahinta on yleensä selvästi vastaavien komponenttien toteutukseen kuluvia kustannuksia edullisempi. Ohjelmistoprojektissa tarvittavien komponenttien etsimiseen on olemassa useita hakukoneita, jolla muutoin vaikea tehtävä voidaan osittain automatisoida koneen suoritettavaksi.

Erilaisia komponenttien toimitusmalleja:

1. Valkoisen laatikon *(engl. white box)* mallissa komponentin käyttäjällä on käytössään komponentin lähdekoodit, joiden avulla hän voi halutessaan muuttaa ja parantaa komponentin toimintaa.

2. Harmaan laatikon *(engl. grey box)* mallissa käyttäjälle ei anneta komponentin lähdekoodeja. Sen sijaan komponenttiin sisällytetään laajennuksia varten esimerkiksi tulkattava laajennuskieli.

3. Mustan laatikon *(engl. black box)* mallissa käyttäjä saa komponentista vain sen binäärisen version, eikä hänellä näin ollen ole mahdollisuutta jatkokehittää komponenttia.

Periaatteessa sovellusohjelmien toteutusvaihe koostuu valmiiden komponenttien yhteenliittämisestä. Käytännössä jokaiselle sovellukselle ominainen sovelluslogiikka joudutaan kuitenkin kirjoittamaan alusta lähtien uudella lähdekoodilla. Sovelluksen elinkaaren aikana esiintyy usein tarvetta korjata sovelluslogiikkakerrosta. Sen sijaan alustan muodostavat komponentit säilyvät lähes muuttumattomina. Jakamalla ohjelmiston rakenne Sanya Ueharan [27] kuvaamalla tavalla sovelluslogiikkaan ja toteutuslogiikkaan, voidaan ohjelmiston usein muuttuvia osia muokata puuttumatta käytettyjen komponenttien toimintaan.

Komponenttipohjaisessa ohjelmistotuotannossa joudutaan käytännössä aina turvautumaan johonkin olemassa olevaan komponenttikehykseen, joka määrittelee komponenttien yhteistoiminnan puitteet. Useiden komponenttikehysten käyttö samassa sovelluksessa johtaa yleensä ongelmiin ja siten kehyksen valintaan on kiinnitettävä erityistä huomiota. Yleiskäyttöisyydessään OMG-ryhmän määrittelemä CORBA [28] muodostaa hyvän lähtökohdan ohjelmistojen kehitykselle. Vaativuudesta ja teknologian kalleudesta johtuen pienimuotoisemmissa projekteissa voidaan käyttää työkalukohtaisia komponenttikehyksiä.

Komponenttien avulla ohjelmistot on mahdollista rakentaa siten, että sovellusta voidaan tarvittaessa laajentaa sitomalla siihen ajon aikana uusia komponentteja. Tällöin ohjelmiston tuottajasta riippumattomatkin tahot voivat laatia tarvitsemiaan lisäpalveluita noudattamalla laajennusten toteutukseen ennalta määriteltyä rajapintaa. Laajennukset jäävät suoritettavan tiedoston ulkopuolelle, mutta voivat silti sulautua osaksi sovelluksen käyttöliittymää, eikä niiden välttämätä tarvitse edes erottua sovelluksen perustoiminnoista.

Käyttöliittymien rakentamiseen on 1970-luvulta lähtien kehitetty erilaisia työkaluja. Kehitys on edennyt suurin harppauksin ja nykyisillä sovellustyökaluilla voidaan pelkän käyttöliittymän kuvauksen ohella tehdä myös varsinaista ohjelmointityötä. Monet nykyisistä graafisten käyttöliittymien kehitysympäristöistä pohjautuvat johonkin komponenttiteknologiaan. Jos komponentteihin toteutetaan tarvittavat rajapinnat, voidaan osa varsinaisesta lähdekoodista muodostaa piirto-ohjelmien tapaan valmiita palasia yhdistelemällä. Software Engineering Institute:n tekemän teknologiakatsauksen [29] mukaan sopivilla työkaluilla voidaan käyttöliittymien kehitykseen kuluva aika lähes aina puolittaa perinteisiin toteutusmalleihin verrattuna. Toteutusvaiheen nopeutuminen ei kuitenkaan tarkoita sitä, että sovellustuotannon mui-

hin vaiheisiin voitaisiin käyttää vähemmän aikaa. Käyttöliittymätyökalujen suurimpina ongelmina voidaan pitää niiden riippuvuutta käytetyistä ikkunanhallintajärjestelmistä ja varsinkin laadukkaimpien työkalujen korkeahkoa hankintahintaa.

Ennen komponenttiteknologioihin siirtymistä tulee ohjelmistonkehitysryhmän kuitenkin pohtia seuraavia Software Engineering Institute:n teknologiaraportissa esiinnostamia kysymyksiä [30], [16].

Lyhytaikaiset vaikutukset:

- Projektiorganisaatio voi joutua muuttamaan toimintatapojaan ja tottumuksiaan vastaamaan uuden menetelmän tarpeita.
- Komponenteista rakennettavan ohjelmistoprosessin suunnittelu vaikeutuu sillä saatavilla olevien komponenttien ja oman työn suhdetta on vaikea arvioida.
- Olemassa olevien komponenttien vaatimusmäärittelyt voivat olla täysin ristiriidassa toteutettavan sovellusohjelmiston vaatimusmäärittelyn kanssa. Esimerkiksi turvallisuuteen voi kohdistua eriäviä vaatimuksia.
- Komponenttien toteutusarkkitehtuurista tulee käytännössä myös sovellusohjelman käyttämä arkkitehtuuri, joka rajaa ohjelmiston suunnitteluvapautta.
- Komponenttien toteutus vaikuttaa ohjelmiston siirrettävyyteen erilaisten suoritusalustojen välillä.
- Komponenttien tarjoamasta uudelleenkäytettävyydestä huolimatta komponentteja on usein sovitettava kulloisenkin sovelluksen tarpeisiin. Tähän kuluva aika on yleensä pienempi kuin komponentin luontiin kulunut aika mutta voi silti olla huomattava.
- Komponenttien laadukkuuden vertailu on vaikeaa ja siten sopivan komponentin valinta ei välttämätä ole helppoa.

Pitkäaikaiset vaikutukset:

- Riippuvuus ulkopuolisista toimittajista ja tästä aiheutuva epävarmuus. Sovelluksen valmistajat ovat riippuvaisia komponenttien toimittajien menestyksestä. Jos komponenttitoimittaja joutuu syystä tai toisesta luopumaan toiminnastaan, ajautuvat kyseisten komponenttien käyttäjät ennenpitkää vaikeuksiin.

- Uuden teknologian käyttöönotto. Komponenttien toimittaja voi ottaa käyttöön uutta teknologiaa, jolloin uudet komponentit eivät välttämätä sellaisenaan sovellu vanhojen sovellusohjelmien tarpeisiin. Komponenttien toimittajat ovat usein haluttomia jatkamaan vanhojen komponenttikirjastojen tukemista.

3.3 Standardit ja avoimet järjestelmät

Standardeilla tarkoitetaan sellaisia sääntöjoukkoja, joiden käytöllä pyritään yhtenäistämään valmistettuja tuotteita. Standardeja käytetään lähinnä käyttäjien vaatimuksesta sillä heterogeenisesta infrastruktuurista käyttäjille aiheutuvat kustannukset voivat pahimmillaan hidastaa siirtymistä uuteen teknologiaan.

Standardilla tarkoitetaan sellaista sääntöjoukkoa, jonka tavoitteena on luoda yhteiset puitteet ongelmien ratkaisuista. Standardeilla pyritään yhtenäistämään standardoinnin kohteena olevaa asiaa siten, että tuottajasta riippumatta kaikki tuotteet toimisivat standardin määräämissä asioissa samalla tavalla. Standardin oleellisin tarkoitus käyttäjän kannalta on tehdä standardoitavasta kohteesta helposti vaihdettavissa oleva, jolloin käyttäjä ei ole sidottu toimissaan vain yhteen tuotteeseen.

De jure -standardit vahvistetaan laeilla ja niiden noudattamista valvotaan tarkasti. Standardin määrittely ja kehitys on yleensä jonkun julkishallinnon alaisen toimikunnan vastuulla ja on siten ainakin periaatteessa riippumaton markkinavoimista. De jure -standardeita käytettiin aiemmin kaikkilla teollisuuden aloilla rautateistä puhelinyhtioihin. De jure -standardeilla saavutetaan nopeasti homogeeninen laitteistoympäristö, joka nopeuttaa teknologian yleistymistä. Haittapuoleksi muodostuu yleensä byrokraattisen hitaasti etenevä kehitys, joka jää teknologisten innovaatioiden jalkoihin.

De facto -standardit muodostuvat markkinataloudellisen kilpailun seurauksena. Yritykset kehittävät kilpaa uusia teknologioita, sillä markkinoilla innovatiivisuus muodostaa parhaan kilpailuvaltin. Uudet keksinnöt yleistyvätkin nopeasti, mutta usein varsin heterogeenisellä laitteistopohjalla. Kilpailullisista syistä vain muutamat toteutukset saavuttavat markkinoiden suosion. Tällöin niistä muodostuu käytännössä standardeja, joiden noudattamiseen koko toimiala ennenpitkää sitoutuu. Nykyään de facto -standardit muodostavat koko IT-toimialan pohjan.

Avoimilla standardeilla tarkoitetaan sellaisia määritelmiä, joiden kehitykseen ja hyötykäyttöön kaikki halukkaat voivat osallistua. Standardia kehitetään yhteisen edun nimissä ja päätökset tehdään enemmän tai vähemmän demokraattisesti. Useimmat de jure -standardit muodostuvat avoimiksi.

Suljetut standardit muodostavat nimensä mukaisesti avoimien vastakohdan. Standardin omistaja suorittaa jatkokehitystä lähinnä omista lähtökohdistaan ja muiden osaksi jää lähinnä valmiin standardin hyödyntäminen. Kilpailun tuloksena syntyvät de facto -standardit sisältävät yleensä omistajiensa patentoimia teknologioita ja siten niistä muodostuu käytännössä aina suljettuja, joiden käyttöoikeus määräytyy tapauskohtaisilla lisensseillä.

Ohjelmistotuotannon kannalta kattavat standardit ovat yhtä oleellisia kuin muillakin toimialoilla. Valitettavasti nopeasti kehittyvällä ohjelmistoalalla ei yleensä olla ehditty riittävään suunnitteluun ja siten heterogeeniseksi muodostuneelle markkinalle on syntynyt lähinnä de facto -standardeja. Alan suurimmat tuottajat ovat hyötyneet tilanteesta eniten ja maksumiehiksi ovat käytännössä jääneet loppukäyttäjät.

Ohjelmistojen toteutusta varten ei tähän päivään mennessä ole vakiintunut yhtä kattavaa standardin mukaista alustaa. Markkinajohtan aseman saavuttanut Microsoft on koko toimintansa ajan pyrkinyt luomaan käyttöjärjestelmästään kaikkien käyttämää de facto -standardia. Käytännössä tässä onkin onnistuttu varsin hyvin saavutetulla yli 90% markkinaosuudella. Suljetusta arkkitehtuurista johtuen muut ohjelmistoyritykset ovat pyrkineet luomaan avointa toteutusalustaa, joka asettaisi kaikki ohjelmistojen tuottajat tasavertaisiin asemiin. 1990-luvun alussa kehitetty POSIX määrittelee standardin tavan käyttöjärjestelmäpalveluiden toteuttamiseen. Standardin käyttö on yleistynyt lähinnä UNIX-maailmassa, eikä sillä ole ollut toivottua vaikutusta koko toimialan kehitykseen. Lisäksi vanhojen suoritusalustojen korvaaminen uusilla epäyhteensopivilla vaihtoehdoilla on osoittautunut käytännössä vaikeaksi.

3.4 Kääntäjäteknologiat ohjelmistotuotannossa

Lähes kaikki nykyisin tehtävä ohjelmistonkehitys tapahtuu kääntäjäpohjaisilla teknologioilla. 1990-luvun alun jälkeen yleistyneet kääntäjät ovat korvanneet lähes täysin aiemmin käytetyt symbolisiin konekieliin pohjautuvat teknologiat. Ensimmäiset kääntäjät sisälsivät vain yksinkertaisimpia ohjelmistorakenteita. 1990-luvun puolivälissä peruskääntäjistä siirryttiin niin sanottuihin olioteknologioihin, jotka paransivat kielten ilmaisuvoimaa huomattavasti. Uudet ominaisuudet auttoivat erityisesti suurten ohjelmistojen suunnittelussa ja ne käytännössä mahdollistivat nykyisten ohjelmistojärkäleiden kaltaiset sovellukset. Kääntäjillä suoritettavaa, yleensä laitteisto- tai käyttöjärjestelmäläheistä, ohjelmointia kutsutaan yleisemmin systeemiohjelmoinniksi.

Ohjelmointikielet voidaan jakaa erillisiin luokkiin sen perusteella, miten ne abstrahoivat käytettyä alustaa. Ylempien tasojen kielille ei välttämättä kannata toteuttaa omaa koodigeneraattoria, vaan ylemmän tason kieli voidaan muuntaa formaalisti jollekin alemman tason kielelle. Tällöin voidaan hyödyntää aiemmin laadittuja työkaluja lopullisen ohjelmiston tuottamisessa. Tarvittaessa työkaluja voidaan ketjuttaa edellisen esimerkin kuvaamalla tavalla jonoiksi, joissa ylemmän tason työkalun tulokset ohjataan seuraavan tason työkalun käsiteltäväksi. Ohjelmiston ajettava versio saadaan suorittamalla koko ketju.

1. Ensimmäisen sukupolven kieliä varten ei ole olemassa varsinaisia kehitysympäristöjä, vaan ohjelmistojen toteutus tehdään kirjoittamalla valmista konekielistä koodia. Ohjelmistojen tuottaminen tällä menetelmällä on äärimmäisen työlästä ja ohjelmistovirheiden esiintyminen yleistä. Ensimmäisen tason ohjelmointityötä onkin suoritettu lähinnä parempien ohjelmointityökalujen tuottamiseen.

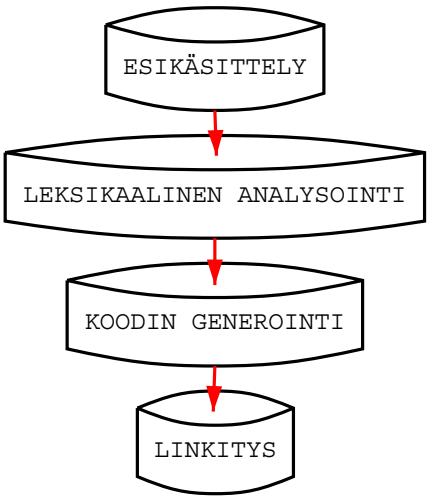
2. Toisen sukupolven ohjelmointikielet muodostuvat binäärisen konekielen symbolisoivista makroassemblerkääntäjistä. Symbolisella ohjelmointikielellä ohjelmisto voidaan rakentaa ihmissilmälle ystävällisemmässä muodossa käskyjen ja muistin osoitusten rakentuessa kuvaavista komentosanoista. Assemblerkieliset ohjelmistot ovat yleensä varsin alustasidonnaisia, vaikka makroprosessoreilla voidaankin toteuttaa alkeellinen käskykannan abstrahointi.

3. Kolmannen sukupolven kielet kattavat nykyiset korkean tason kieliksi kutsutut ohjelmistonkehitysympäristöt ja ne perustuvat yleensä kääntäjäteknologioihin. Kielien päätavoitteena on abstrahoida yleisimmin käytetyt ohjelmistorakenteet siten, ettei käyttäjän tarvitse ottaa kantaa niiden toteutukseen. Näin saavutetaan huomattavasti parempi siirrettävyys ja samalla tuotetun ohjelmiston parempi luotettavuus.

4. Neljännen sukupolven kielien määritelmä ei ole yhtä hyvin vakiintunut kuin edellisten tasojen. Nykyisin käytössä olevia skriptikieliä kutsutaan usein neljännen sukupolven kieliksi niiden korkean abstraktiotason perusteella. Toisaalta monia integroituja sovelluskehittimiä saatetaan kutsua neljännen sukupolven työkaluiksi, vaikka ne pohjautuisivat alemman tason kieliin. Varsin monia nopean kehityksen työkaluja *(engl. Rapid Application Development)*, jotka sisältävät valmiit mallit graafisen käyttöliittymän ja tietokantayhteyden toteuttamiseen, kutsutaan lähes poikkeuksetta neljännen sukupolven työkaluiksi.

5. Viidennen sukupolven kielien määritelmä on vielä edellistäkin epämääräisempi. Pelkän abstraktiotason perusteella vertailtaessa minkään nykyisen kielen ei voida katsoa saavuttaneen uutta tasoa. Toisaalta monet asiantuntijajärjestelmät nimittävät jo nyt itseään viidennen sukupolven työkaluiksi.

Lähes kaikki olemassa olevat kääntäjät sisältävät kuvan 3.4 mukaisesti neljä erillistä työvaihetta. Ensimmäisessä vaiheessa suoritetaan lähdekoodien esikäsittely makroprosessorilla. Esikäsittelijää voidaan käyttää esimerkiksi lähdekoodeissa esiintyvien vakioiden korvaamiseen niiden numeerisilla arvoilla, käännettäväksi tulevan lähdekoodin valintaan ja muihin yksinkertaisiin tehtäviin. Esikäsittelyvaihe ei ole välttämätöntä mutta sillä aikaansaatu tuottavuuden parantuminen on usein varsin huomattava. Toisessa vaiheessa suoritettava leksikaalinen analysointi tarkistaa lähdekoodien syntaksin ja jäsentää tämän pohjalta ohjelmiston rakenteen. Lähdekoodeissa esiintyvät yksinkertaisimmat virheet havaitaan yleensä tässä vaiheessa. Kolmannessa vaiheessa tuotetaan laaditun rakenteen pohjalta konekielinen esitys. Ohjelmarakenteet muunnetaan binäärisiksi moduleiksi, joita käytetty laitteistoalusta voi suorittaa. Viimeisessä vaiheessa syntyneet ohjelmistomodulit ja tarvittavat kirjastot liitetään yhteen käyttöjärjestelmän suoritettaville tiedostoille määräämään muotoon.



Kuva 3.4: Kääntäjän toimintaperiaate.

Oman kääntäjän toteuttaminen vaatii runsaasti erityistietoja. Onneksi kääntäjistä on olemassa runsaasti hyvälaatuista kirjallisuutta. Kääntäjän

suunnitteluun annetaan vinkkejä kirjoissa Compilers : Principles, Techniques, and Tools [31] ja Advanced Compiler Design and Implementation [32]. Varsinaista toteutusta varten löytyy ohjeita kirjoista Writing Compilers and Interpreters [33] ja A Retargetable C Compiler : Design and Implementation [34]. Oman kääntäjän toteuttaminen ei siten ole ylipääsemättömän vaikea tehtävä. Käyttökelpoisen kääntäjän kehitykseen kuluu kuitenkin runsaasti aikaa ja siten valmiiden kääntäjien käyttö on yleensä järkevin vaihtoehto.

UNIX-käyttöjärjestelmien kehityskielenä yleistynyt C-kieli suunniteltiin jo alunperin helposti siirrettäväksi. Koska C-kielisten ohjelmointityökalujen toteutusta ei valvota mitenkään, joutuu käyttäjä käytännössä luottamaan työkalun tuottajan lupauksiin standardien noudattamisesta. Useimmat työkalujen tuottajat ovat parannelleet tuotteitaan epästandardeilla laajennuksilla. Epästandardien toteutusten aiheuttaessa merkittäviä ongelmia on kielen kehitystä ryhdytty ohjaamaan American National Standards Institute -organisaation toimesta. Laadittu ANSI C-standardi ei kuitenkaan kiellä epästandardeja lisäominaisuuksia. Työkalujen käyttämien kielisyntaksien saavuttaessa riittävän hyvän standardointiasteen on työkalujen mukana toimitetuista aliohjelmakirjastoista noussut uusi siirrettävyytä rajoittava tekijä. Lähes kaikki työkaluvalmistajat ovat lisänneet kirjastoihinsa standardien vaatimien palveluiden lisäksi työkalukohtaisia laajennuksia. Tällaisten laajennusten käyttö johtaa nopeasti ohjelmiston lähdekoodien siirtymättömyyteen.

Olio-ohjelmoinnin yleistyttyä on ANSI C++ -standardi alkanut syrjäyttää ANSI C -standardia ohjelmien tuotantoalustana. C++ tarjoaa monia huomattavia parannuksia C-kieleen verrattuna. Siirrettävyyttä rasittavat kuitenkin jo C-kielen peruja olevat ongelmat. Lähes jokaisella valmistajalla on tuotteissaan mukana omia laajennuksia kielen syntaksiin. Standardien mukaiset kirjastot ovat nykyään kattavampia, mutta puutteita esiintyy silti usein. Pahimman ongelman muodostavat valmistajakohtaiset kirjastot, jotka ovat paisuneet valtaviksi. Varsinkin graafisen käyttöliittymän luomiseen on kehitetty lähes lukematon määrä erilaisia kirjastoja, jotka ovat kaikki keskenään yhteensopimattomia. Näiden kirjastojen käyttöä ei kuitenkaan käytännössä voida välttää sillä vastaavien ominaisuuksien toteuttaminen käsityönä olisi aivan liian työlästä.

Perinteisten kääntäjien lisäksi on olemassa monia neljännen sukupolven käännöstyökaluja. Monet niistä ovat alakohtaisia ja siten niitä yhdistää lähinnä kyky kuvata määriteltävää ongelmakenttää muita korkeammalla tasolla. Käyttäjän tavoitteena on kuvata tarkasti rajattu ongelma hyvin korkean tason kielellä, jonka perusteella työkalu generoi kuvatun sovelluksen toteuttavan koodin. Yleisimmät käyttökohteet tällähetkellä ovat erilaiset tietokantasovellukset ja niihin liittyvä raporttien tuotanto. 4GL-ohjelmistoprosessin alkuvaihe muodostuu lähes aina iteratiiviseksi johtuen käyttäjän tavoittees-

ta minimoida oma määrittelytyönsä mahdollisimman vähäiseksi. Näin ollen käyttäjä luo ensin pelkistetyn kuvauksen ohjelmiston toiminnasta ja tarkistaa sen pohjalta luodun ohjelmiston vastaavuutta tavoitteisiinsa. Niiden ominaisuuksien osalta, jotka eivät vastaa käyttäjän toiveita, on kuvausta tarkennettava. Tällainen tarkennusketju lähestyy lopulta niin lähelle tavoitetasoa, että kehitystyö voidaan keskeyttää ja sovelluksesta on saavutettu toimiva versio.

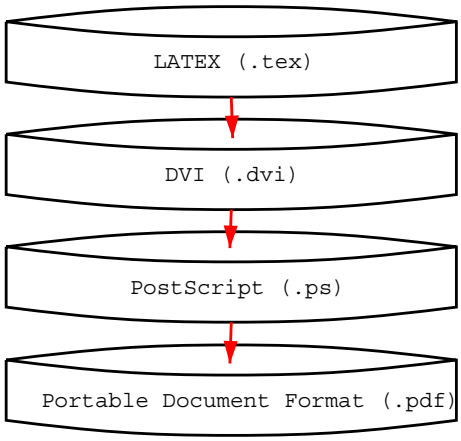
Ennen sovelluksen tuotteistamista on myös 4GL-ohjelmoinnissa suoritettava ohjelmiston testaus ja kunnollinen dokumentaatio. Vaikka 4GL-työkalut vähentävätkin toteutusvaiheeseen kuluvaa aikaa, on niillä päinvastainen vaikutus testausvaiheeseen, joka on suoritettava erityisen tarkasti sillä 4GL-työkalun generoima lähdekoodi ei välttämätä vastaakaan käyttäjän käsitystä ongelman ratkaisusta.

Kääntäjillä aikaansaadut sovellukset ovat perinteisesti olleet vaikeita laajentaa. Alkuaikojen kääntäjät eivät tunteneet ohjelmistomodulien dynaamista sidontaa, eikä menetelmän käyttö ole yleistynyt vielä nykyisillä työkaluillakaan. Kääntäjillä voidaan toki tuottaa tehokkaita laajennusmoduleja mutta niiden sitominen osaksi valmista sovellusta ei ole helppoa. Systeemiohjelmoinnilla toteutettujen sovellusten laajentaminen tehdäänkin yleensä seuraavassa alaluvussa esiteltävillä makrotulkeilla.

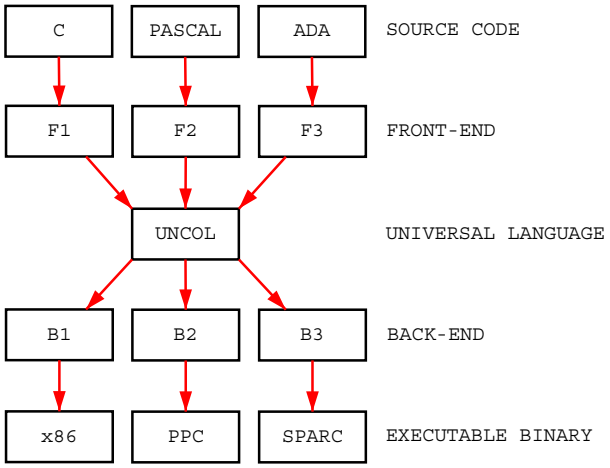
Ajettavan binäärikoodin ohella kääntäjillä voidaan tuottaa myös ohjelmointikieltä vastaavan tason muilla kielillä olevia esityksiä. Tällöin lähdekoodit muunnetaan formaalisti jollekin toiselle ohjelmointikielelle. Esimerkiksi GNU-työkalupaketin mukana toimitettava Pascal-kääntäjä toimii tällä periaatteella. Menetelmä ei takaa parasta luodun binäärikoodin tehokkuutta mutta on silti varsin käyttökelpoinen haluttaessa vaihtaa sovelluksen kehityksessä käytettävää ohjelmointikieltä. Tarkempia tietoja formaalien muunnoskoneiden toteutukseen ja käyttöön liittyvistä seikoista on saatavilla D. Jones:in kirjoittamasta C-kielen ja Ada:n välistä muunnosta käsittelevästä artikkelista [35]. Python:illa toteutettujen ohjelmistojen nopeuttamista muuntamalla ne C-kielelle on käsitelty J. Aycock:in artikkelissa Converting Python Virtual Machine Code to C [36].

Kuvassa 3.5 on esitetty käytännön esimerkki dokumentaation tuottamisesta formaalilla käännöksellä. Esimerkissä käyttäjä luo Latex-taittoohjelmaa varten lähdekoodiesityksen dokumentistaan. Latex suorittaa dokumentin koostamisen ja ladonnan, ja kääntää tuloksen UNIX-maailmassa käytettyyn .dvi-kuvausformaattiin. Tulos käsitellään toisella työkalulla, jolloin se saadaan tulostimien ymmärtämään PostScript-muotoon. Lopuksi suoritetaan vielä muunnos internetissä paljon käytettyyn Portable Document Format -muotoon.

Jo tietokoneiden aikakauden alkuvaiheessa ymmärrettiin siirrettävyydestä aiheutuvan ennenpitkään vaikeasti hallittava ongelma [37], [38] ja [39].



Kuva 3.5: Esimerkki ristiinkäännöksestä.



Kuva 3.6: UNCOL toimintaperiaate.

Koska kääntäjät muodostuvat edellä kuvatulla tavalla laitteistoriippumattomasta ja kieliriippuvasta alkupäästä *(engl. front-end)*, sekä laitteistoriippuvasta ja kieliriippumattomasta loppupäästä *(engl. back-end)*, tutkittiin mahdollisuutta luoda uudentyyppinen ohjelmointityökalu, joka voisi kääntää käytössä olevilla kielillä tuotettuja ohjelmia eräänlaiselle välikielelle, joka voitaisiin jokaisessa laitteistossa kääntää tämän käyttämälle konekielelle. Menetelmän toimintaa on havainnollistettu kuvassa 3.6. UNCOL-kieli muodostaa rajapinnan, jolla eri kielille sovitetut leksikaaliset analysaattorit ja erityyppisille laitealustoille sovitetut koodigeneraattorit saadaan toimimaan yhtenäisenä ketjuna.

Periaatteessa minkä tahansa laitteiston konekieli voisi toimia välittävänä kielenä. Käytännössä konekieli osoittautui kuitenkin liian matalatasoiseksi, jotta sen pohjalta olisi voitu muodostaa riittävän korkeatasoisia käännöksiä muihin ympäristöihin. Toteutuessaan UNCOL olisi poistanut monta nykypäivänä ohjelmistoja vaivaavaa ongelmaa.

Siirrettäviä ohjelmistoja tuotettaessa voidaan korkean tason kieliä käyttää abstrahoimaan laitteistoläheiset ohjelmiston osat. Käytettävää kieltä valittaessa on syytä kiinnittää huomiota työkalun yhteensopivuuteen alalla vallitsevien standardien kanssa. Lisäksi on syytä huomata, että useimpien standardien työkalujen mukana toimitetaan sellaisia aliohjelmakirjastoja, jotka eivät toimi yleisten standardien mukaisesti. Varsinkin pienempien työkalujen mukana toimitetaan aliohjelmakirjastot, voivat olla puutteellisia ja voivat toimia standardista poikkeavalla tavalla.

Kirjastostandardit määrittelevät eräitä yleisimmin tarvittavia perusominaisuuksia, joiden tulisi löytyä jokaisesta työkalusta. Standardeiksi kutsuttuja kirjastojakin käytettäessä on syytä säilyä valppaana, sillä useimmat niistä sisältävät myös epästandardeja laajennuksia, joita ei löydy muista vastaavista kirjastoista. Lisäksi osa standardeista kirjastoista toimii vain niissä ympäristöissä, joihin ne on suunniteltu. Esimerkkinä tällaisista voidaan antaa UNIX-standardin mukainen tapa hallita prosesseja ja säikeitä, joka ei toimi UNIX-maailman ulkopuolella.

Standardin mukaisia työkaluja käytettäessä ohjelmiston siirtäminen uusille alustoille perustuu käytettävän työkalun siirrettävyyteen. Jos korkean tason kielen kääntäjä ja yleisimmät kirjastot saadaan siirrettyä uudelle alustalle, ei ohjelmistojen siirrosta yleensä enää aiheudu suurtakaan vaivaa. Näin olisi periaatteessa mahdollista saavuttaa siirrettävyys ilman muutoksia ohjelmiston lähdekoodeihin. Koska työkalun tuottaminen ja siirtäminen on varsin työläs operaatio, joudutaan usein turvautumaan valmiisiin työkaluihin. Tällöin työkalujen välillä voi olla eroja ja siten lähdekoodeja on sopeutettava käytetyn työkalun erityispiirteisiin. Huolellisesti suunnitellussa ohjelmistossa nämä muutokset on kuitenkin useinmiten helppo tehdä.

Tee-se-itse ratkaisuilla tarkoitetaan sellaista lähestymistapaa, jossa ohjelmistosta pyritään toteuttamaan mahdollisimman suuri osuus itsenäisesti hyödyntämättä työkalujen mukana toimitettuja kirjastoja. Laatimalla ohjelmiston lähdekoodit standardilla kielellä ja rajoittamalla käytettyjen aliohjelmakirjastojen käyttö minimiin, saavutetaan erittäin hyvä siirrettävyys alustalta toiseen. Menetelmän huonona puolena voidaan pitää sen vaatimaa valtavaa työpanosta, joka kuluu suurimmaksi osaksi epäoleellisten yksityiskohtien ratkaisemiseen. Tuloksena saatu ohjelmisto on kuitenkin yleensä erittäin sopeutumiskykyinen, joten erikoistapauksissa menetelmän käyttö voi olla perusteltua. Suuria ongelmia voi aiheutua silloin, kun tarvittavaa ominaisuut-

ta ei voida toteuttaa itse ja sopivaa standardia ei ole käytettävissä. Nykyisellään tällaisen ongelman aiheuttaa esimerkiksi käyttöliittymän ikkunoiden hallinta. Menetelmä soveltuu parhaiten erilaisten tieteellisiä analyyseja suorittavien ohjelmistojen ja matemaattisten kirjastojen toteuttamiseen, joiden pääpaino on algoritmisissa rakenteissa, eikä käyttöalustasta riippuvissa palveluissa, kuten käyttöliittymissä.

3.4.1 Tulkkaavat teknologiat ohjelmistotuotannossa

Tulkattavat kielet pyrkivät parantamaan ohjelmistotyön tuottavuutta nostamalla ohjelmoinnin abstraktiotasoa vielä edellisessä alaluvussa esiteltyjä systeemiohjelmointikieliäkin korkeammalle tasolle [40], [41]. Skriptiohjelmointi pohjautuu pitkälti edellisissä alaluvuissa esiteltyyn komponenttimalliin. Periaatteena on, että skriptikielellä ei edes yritetä toteuttaa kaikkea toiminnallisuutta vaan kieltä käytetään lähinnä valmiiden ohjelmistopalasten yhteenliimaamiseen.

Skriptikielillä on saavutettu vielä systeemiohjelmointikieliäkin korkeampia tuottavuuden tasoja (taulu 3.1 sivulla 38). Siinä missä systeemiohjelmointikielillä saavutettiin 5:1 muunnossuhteita saavutetaan skriptikielillä käytetyistä komponenttikirjastoista riippuen jopa 1000:1 muunnoksia, joissa yksi lähdekoodirivi suorittaa keskimäärin tuhat konekielistä käskyä. Keskimäärin saavutettu suhde muodostuu tapauskohtaisesti lähelle 30:1 suhdelukua, joka on silti huomattava parannus systeemiohjelmointikielillä saavutettuun tasoon. Korkeasta abstraktiotasosta johtuen skriptikielet eivät enää sovellu systeemiohjelmointikielten tapaan laitteistoläheiseen ja algoritmien ohjelmointiin, vaan nämä on syytä edelleen toteuttaa muilla menetelmillä skriptikielistä käytettäviksi komponenteiksi.

Ohjelmoinnin yksinkertaistamiseksi useimmat skriptikielet ovat tyypittömiä. Käytännössä tämä tarkoittaa, että samaan ohjelmistomuuttujaan voidaan tallentaa vuoroin erillaisia muuttujia, kuten numeroita ja merkkijonoja. Muuttujien lisäksi tyypittömyyttä käytetään komponenttien välisissä rajapinnoissa, jolloin useista eri lähteistä hankittujen komponenttien rajapinnat saadaan muistuttamaan toisiaan. Koska suoritettavan lähdekoodin ja muuttujien välinen erotus ei ole yhtä selvä kuin systeemiohjelmoinnissa, voidaan skriptikielisellä sovelluksella tuottaa uutta skriptilähdekoodia, joka voidaan ajaa osana sovellusta. Ominaisuus on erittäin käyttökelpoinen esimerkiksi käyttöliittymästä nauhoitettavien makrojen toteutukseen. Kaikki edellämainitut vapaudet eivät kuitenkaan pelkästään edistä ohjelmointityötä. Tyypittömyydestä johtuen muuttujien tyypit on tarkistettava ohjelman suoritusvaiheessa, jolloin menetetään suoritustehoa. Mahdollisten virheiden esiintyminen johtaa sovelluksen suorituksen päättymiseen ja siten valmistu-

Application	Comparison	Code Ratio	Effort Ratio	Comments
Database application (Ken Corey)	C++ version: 2 months. Tcl version: 1 day.		60	C++ version implemented first; Tcl version had more functionality.
Computer system test and installation (Andy Belsey)	C test application: 272 Klines, 120 months. C FIS application: 90 Klines, 60 months. Tcl/Perl version: 7.7 Klines, 8 months.	47	22	C version implemented first. Tcl/Perl version replaced both C applications.
Database library (Ken Corey)	C++ version: 2-3 months. Tcl version: 1 week.		8-12	C++ version implemented first.
Security scanner (Jim Graham)	C version: 3000 lines. Tcl version: 300 lines.	10		C version implemented first. Tcl version had more functionality.
Display oilwell production curves (Dan Schenck)	C version: 3 months. Tcl version: 2 weeks.		5	Tcl version implemented first.
Query dispatcher (Paul Healy)	C version: 1200 lines, 4-8 weeks. Tcl version: 500 lines, 1 week.	4		Tcl version implemented first.
Spreadsheet tool	C version: 1460 lines. Tcl version: 380 lines.	4		Tcl version implemented first.
Simulator and GUI (Randy Wang)	Java version: 3400 lines, 3-4 weeks. Tcl version: 1600 lines, $<$ 1 week.	2	3-4	Tcl version had 10-20% more functionality, was implemented first.

Taulukko 3.1: Ohjelmistojen tuottaminen systeemiohjelmoinnilla ja skriptikielillä.

neen työn menetykseen. Lisäksi lähdekoodien tyypitön rakenne rapauttaa laadittuja ohjelmistorakenteita vaikeuttaen näin ohjelmiston ylläpitoa.

Tulkattavien ohjelmistojen suoritus eroaa merkittävästi edellisessä alaluvussa esitellyistä käännetyistä ohjelmistoista. Binäärisen esityksen sijaan

tulkattavista ohjelmistoista toimitetaan niiden lähdekoodit, joita parsitaan ajonaikaisesti ohjelmiston suorituksen edetessä. Ohjelmiston käynnistys tapahtuu näin huomattavasti nopeammin ja kehitysvaiheessa vältytään runsaasti aikaa kuluttavilta käännösoperaatioilta. Viimeaikoina skritikielien puhtaasta tulkkaamisesta ollaan osittain siirrytty jäljempänä esiteltävään virtuaalikonemalliin, jossa skritikielinen lähdekoodi käännetään yksinkertaiseen binääriseen esitykseen, joka suoritetaan virtuaalikoneella ajamalla [42].

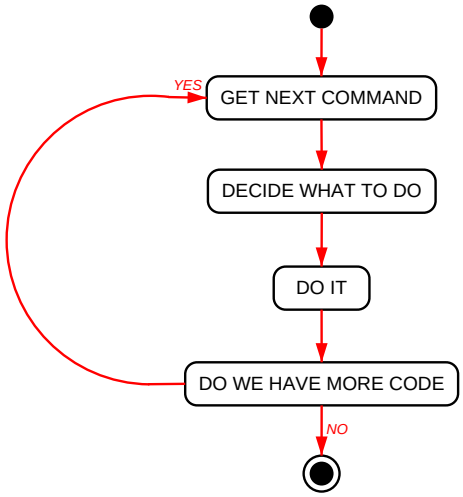
Tulkattavien kielien suurimpana heikkoutena verrattuna käännettäviin kieliin voidaan pitää niiden osittain puutteellista suorituskykyä. Suurin syy suorituskyvyn laskuun löytyy tulkaavan ajon huomattavasti laitteistoajoa suuremmasta vaativuudesta. Lisäksi monissa skriptityökaluissa on panostettu helppoon ohjelmoitavuuteen suorituskyvyn kustannuksella. Vapaamuotoiset komponenttien rajapinnat ja tyypittömät muuttujat ovat helppoja käyttää mutta aiheuttavat suoritusvaiheessa ylimääräisiä toimenpiteitä, joihin käännetyillä ohjelmistoilla ei ole tarvetta. Keskimäärin skriptikielisen lähdekoodin suoritusnopeus on noin 1/20-osa vastaavasta konekielisestä toteutuksesta.

Saavutetusta heikommasta suorituskyvystä ei suurimmassa osassa ohjelmistotyyppejä ole varsinaista haittaa. Nykyisissä ohjelmistoissa suurin osa tietokoneen ajasta kuluu käyttäjän syötettä odotellessa ja siten tavalliset skriptikieliin pohjautuvat käyttöliittymäpainotteiset sovellukset eivät juuri eroa muista vastaavista ohjelmista. Skriptikielen suorituskykyä parantaa myös se, että niillä toteutetuissa ohjelmistoissa on yleensä huomattavasti vähemmän lähdekoodia kuin käännetyissä ohjelmissa. Näin suurin osa ohjelmiston suorituksesta tapahtuu ohjelmiston toteutukseen käytetyissä komponenteissa. Helpoin tapa parantaa skriptikielen suorituskykyä onkin parantaa käytettyjen komponenttien toiminnallisuutta.

Oman skriptikielisen tulkin rakentaminen on huomattavasti vastaavaa kääntäjää helpompi operaatio. Yksinkertainen tulkki koostuu kuvan 3.7 mukaisesti silmukasta, joka suorittaa yhden skriptikielisen rivin kerrallaan.

Oman skriptikielen toteuttaminen alkaa laadittavan kielen syntaksin määrittelyllä. Yksinkertaisimmillaan skriptikieli voi koostua vain muutamista avainsanoista.

```
<stored_program> ::= <program_line>{<program_line>}
<program_line> ::= <number><statement>
<statement> ::= <print_stmt>|<input_stmt>|
                END|<assign_stmt>|<goto_stmt>
<print_stmt> ::= PRINT ["prompt" ]<expression>
<input_stmt> ::= INPUT ["prompt" ]<variable>
<assign_stmt> ::= LET <variable>=<expression>
<goto_stmt> ::= GOTO line-number
```



Kuva 3.7: Tulkin toimintaperiaate.

```
<expression> ::= <factor>{ <op> <factor>}
<factor> ::= <variable>|<number>
<op> ::= +|-|*|/
...
```

Esimerkissä [43] on aloitettu uuden BASIC:in kaltaisen kieliopin määrittely. Kielen määrittely aloitetaan yleensä ylemmän tason avainsanoista, joista jatketaan kohti yhä yksinkertaisempia kokonaisuuksia. Tämän jälkeen ohjelmoidaan kielelle varsinainen parseri, jolla määriteltyä kieltä käyttäviä lähdekoodeja voidaan suorittaa.

Oman skriptitulkin toteuttaminen ei yleensä ole tarpeellista. Nykyisten tietoverkkojen kautta on mahdollista hankkia ilmaiseksi useita tulkkityökaluja, kuten Tcl/Tk [44], Perl [45] ja Python [46]. Näistä Tcl/Tk:ta on käytetty paljon erilaisten UNIX-ympäristössä toimivien graafisten käyttöpaneelien toteutukseen. Suhteellisen uusi tulokas Python yhdistää skriptikielten hyvät puolet olio-ohjelmoinnin paradigmaan. Edellä mainittuja työkaluja on kehitetty jo vuosia vapaaehtoisten ohjelmoijien toimesta ja siten ne muodostavat jo varsin luotettavan toteutusalustan sovellusohjelmointiin. Osa saatavilla olevista skriptitulkeista on tehty siten, että ne voidaan liittää systeemiohjelmointikielellä toteutettavaan sovellukseen, jolloin sovellusta voidaan laajentaa käyttämällä skriptikielen tarjoamia palveluita.

Sovellusohjelmien kehitykseen kannattaa käyttää valmiita skriptityökaluja, joihin on saatavilla riittävän laajat komponenttikirjastot. Eri skriptityökalut soveltuvat hieman erilaisiin ongelmiin. Monet UNIX:eissa käytettävät

komentotulkit *(engl. shell)* luetaan myös skriptikieliksi. Komentotulkit soveltuvat yleensä huonosti graafisten käyttöliittymien toteutukseen ja niiden pääkäyttökohteena onkin erilaisten käsittelyketjujen kuvaaminen.

Systeemiohjelmointikielillä tuotettuja sovelluksia voidaan laajentaa, mikäli niihin on tarkoitusta varten sisäänrakennettu makrokielen tulkki. Laajennukset voidaan tehdä joko kirjoittamalla makrolähdekoodit sovelluksen ulkopuolisella editorilla, sovelluksen tarjoamalla työkalulla tai nauhoittamalla käyttäjän toimia käyttöliittymässä. Makrolaajennuksilla on yleensä suora pääsy lähes kaikkiin sovelluksen tarjoamiin palveluihin. Siten makrokieliset laajennukset eivät ole yhtä riippuvaisia ulkopuolisista komponenteista, kuin varsinaiset skriptikieliset sovellukset.

Rajoituksistaan johtuen skriptikielet eivät voi yksinään korvata perinteisiä ohjelmointikieliä. Skriptikielten käytöstä päättäminen onkin suoritettava aina tapauskohtaiset ohjelmiston erityisvaatimukset huomioiden. Skriptikielten käyttö on perusteltua silloin kun suorituskyky ei muodostu rajoittavaksi tekijäksi ja tarvittavat komponentit ovat saatavilla. Jos komponentteja ei ole saatavilla, joudutaan ne joka tapauksessa toteuttamaan jollakin systeemiohjelmointityökalulla. Tällöin on todennäköisesti järkevämpää kehittää koko ohjelmisto samalla työkalulla, kuin jakaa sitä erillisillä työkaluilla toteutettaksi. Skriptikielillä on keskimäärin saavutettavissa 5-10 kertainen tuottavuus tavallisia toimisto-ohjelmistoja toteutettaessa. Toisaalta systeemiohjelmointikielillä toteutetut ohjelmistot suoriutuvat vastaavista tehtävistä yleensä 10-20 kertaa nopeammin.

Ohjelmistotuotannon alalla on muutamia tahoja, jotka uskovat skriptikielien käytön yleistyvän tulevaisuudessa myös tavallisten ohjelmistojen toteutusympäristönä. Alla on lueteltu muutamia tärkeimmistä perusteista, joiden valossa skriptikielillä voitaisiin saavuttaa parempia tuloksia, kuin perinteisillä systeemiohjelmointikielillä.

- Ohjelmistojen tuotanto siirtyy vääjäämättä kohti suurempaa komponenttien käyttöä. Yhä monimutkaisemmiksi muuttuvat ohjelmistot vaativat yhä enemmän ja enemmän lähdekoodia. Lopulta saavutaan siihen tilanteeseen, jossa tehokas uudelleenkäyttö osoittautuu välttämättömäksi. Uusi elektroninen maailma on muovautumassa modulaariseksi. Skriptikielet soveltuvat modulien yhteenliimaamiseen paljon systeemiohjelmointikieliä paremmin.

- Skriptikielet ovat viimeisen vuosikymmenen aikana kehittyneet huomattavasti ja tulevaisuudessa niiden uskotaan nousevan yhä korkeammalle abstraktion tasolle. Tällöin voidaan saavuttaa vielä tässä työssä esitettyjä esimerkkejäkin parempia tuottavuuslukuja.

- Suoritustehon jatkuva kasvu mahdollistaa yhä useamman ohjelmiston toteuttamisen skriptikielellä. Mooren ilmiönä tunnetun prosessoritehoa mittaavan lain uskotaan pätevän vielä ainakin kymmenen vuotta. Esimerkiksi nykyiset tietokoneet ovat jo noin 500-1000 kertaa nopeampia, kuin ensimmäiset koneet.

- Ohjelmistojen kehittäjien määrä kasvaa jatkuvasti, eikä kaikilla halukkailla ole mahdollisuutta kuluttaa aikaa monimutkaisiksi osoittautuneiden systeemiohjelmointikielten opiskeluun. Sen sijaan jo muutaman päivän opiskelulla saatetaan oppia riittävästi skriptikieltä, jotta voidaan rakentaa yksinkertaisia ohjelmistoja tai laajennuksia jo olemassa oleviin ohjelmistoihin.

3.4.2 Virtuaalikoneteknologiat ohjelmistotuotannossa

Virtuaalikoneet ovat viimeisten viiden vuoden aikana saavuttaneet yhä suuremman suosion siirrettävien ohjelmistojen tuotantoalustana. Kirjavassa laitteisto- ja käyttöjärjestelmätarjonnassa virtuaalikoneet tarjoavat mahdollisuuden perinteiseen yhden arkkitehtuurin malliin, jossa ohjelmistosta laaditaan versio vain yhdelle alustalle, ilman siitä siirrettävyydelle aiheutuvia ongelmia.

Virtuaalikoneen toimintaperiaate on yhdistelmä edellisissä alaluvuissa esitellyistä kääntäjä- ja tulkkausteknologioista. Ohjelmistot kirjoitetaan ohjelmointikielellä ja muunnetaan kääntäjällä binääriseen esitykseen. Binäärejä ei kuitenkaan toteuteta suoraan laitteistoalustalla ajettavaan muotoon vaan tarkoitusta varten luodulle virtuaalikäskykannalle. Sovelluksen suoritusta varten tarvitaan virtuaalikone, joka joko tulkkaa binäärisen koodin tai kääntää sen uudelleen nyt alustan käyttämälle konekielelle. Tulkattaessa virtuaalikoneen suorituskyky on noin 1/10 - 1/20 osa vastaavan konekielisen sovelluksen suorituskyvystä. Käyttämällä uusia JIT-kääntäjiä *(engl. Just in Time)* päästään lähes 80%:iin vastaavan konekielisen toteutuksen suorituskyvystä.

Uuden virtuaalikoneen ohjelmointia varten on yleensä kehitetty uusi kieli, joka kykenee hyödyntämään tarjolla olevia erityispalveluita. Suurin osa kielistä sijoittuu syntaksiltaan ja tuottavuudeltaan jonnekin systeemiohjelmointija skriptikielten väliin. Kielten abstraktiotaso voi nousta lähelle skriptikieliä, vaikka kieli muutoin muistuttaisikin systeemiohjelmointikieliä.

Joihinkin virtuaalikoneisiin on sovellusten toimintavarmuuden parantamiseksi toteutettu automaattinen roskien keruu. Nykyisissä ohjelmistoissa suurin yksittäinen ohjelmointivirheiden aiheuttama seuraus on viallinen muistinkäsittely. Roskien keruun ajatuksena on siirtää suurin osa muistin hal-

lintaan liittyvistä operaatioista ohjelmoijalta virtuaalikoneen vastuulle. Lähtokohtaisesti ohjelmoijan ei tarvitse huolehtia varaamiensa muistialueiden vapauttamisesta. Tiukimmin rajoitetuissa kielissä, kuten Java, ohjelmoijalla ei ole edes käytettävissään suoria muitiosoitteita. Automaattisella muistin hallinnalla on saavutettu hyviä käytännön tuloksia ohjelmointivirheiden karsinnassa. Kaikissa tapauksissa automatiikka ei kuitenkaan toimi kunnolla ja siten virtuaalikoneiden tulisikin tarjota ohjelmoijalle mahdollisuus vaikuttaa osaan muistioperaatioista. Jatkuva muuttujien varaus ja vapauttaminen johtaa nopeasti tehottomaan lopputulokseen.

Nykyaikainen verkottunut tietojenkäsittely on luonut uudenlaisia haasteita ohjelmistojen turvallisuuteen. Aiemmin ohjelmat ostettiin myymäläpaketoituina, jolloin käyttäjä saattoi luottaa siihen, että ohjelmistoon ei ollut piilotettu hänen kannaltaan eitoivottuja ominaisuuksia. Nykyisin suuri osa ohjelmistoista hankitaan tietoverkkojen kautta, jolloin käyttäjä ei voi olla varma ohjelmiston luotettavuudesta. Ongelmaa on yritetty helpottaa erilaisilla ohjelmistojen allekirjoitusmenetelmillä, mutta ohjelmistotuottajien hajanaisuuden takia mikään menetelmistä ei ole yleistynyt. Useimpiin virtuaalikoneisiin on toteutettu tarkoitusta varten itsenäisiä toimialueita, eli hiekkalaatikoita *(engl. sandbox)*, joissa epäilyttäviä ohjelmistoja voidaan ajaa ilman, että näillä on mahdollisuus vaikeuttaa käyttäjän työskentelyä. Toimialue koostuu tarkasti rajatusta sääntöjoukosta, jolla kuvataan millaisia operaatioita sovelluksella on oikeus suorittaa. Esimerkiksi verkkosivujen elävöittämiseen käytetyille sovelmille ei yleensä anneta pääsyä käyttäjän tiedostoihin.

Yksi suurimmista nykyisiin virtuaalikoneisiin liittyvistä ongelmista on niiden käytännön toteutuksiin liittyvät heikkoudet [49]. Seuraavassa on lueteltu muutamia näistä puutteista.

- Vaikka olio- ja komponenttiteknologiat ovat olleet käytettävissä jo lähes vuosikymmenen, on suurin osa virtuaalikoneista toteutettu perinteisen monoliittisesti. Tästä aiheutuu väistämätä ongelmia virtuaalikoneen skaalautuvuuteen. Alunperin samojen virtuaalikoneiden tuli toimia aina kämmenmikroista palvelimiin. Nykyiset, lähinnä työasemille optimoidut, virtuaalikoneet ovat lähes poikkeuksetta liian raskaita kämmenmikrojen vaatimattomille resursseille. Ongelmaa olisi voitu helpottaa siten, että virtuaalikoneiden rakenteista olisi suunniteltu modulaarisempia, jolloin pienimmissä ympäristöissä ei olisi tarvinnut toteuttaa kaikkia suurissa järjestelmissä tarvittavia palveluita.

- Luotettavan toimialueen aikaansaaminen on osoittautunut varsin vaikeaksi tehtäväksi. Vaikka virtuaalikoneet suunnitellaan niin, että niillä voidaan suorittaa myös epäluotettavia ohjelmistoja, on lähes kaikista käytännön toteutuksista löytynyt pahoja turvallisuuden vaarantavia

virheitä. Puutteellisesti toteutetut virtuaalikoneet tuudittavat käyttäjät väärään turvallisuuden tunteeseen. Jos virtuaalikoneiden yhteydessä törmätään samanlaisiin ongelmiin kuin makroja sisältävien tiedostojen yhteydessä, voi virtuaalikoneteknologian yleistyminen pysähtyä.

- Lähes kaikki virtuaalikonetoteutukset muodostavat oman toiminnallisen kokonaisuutensa. Käytettävyyden ja ylläpidettävyyden parantamiseksi olisi suoritusympäristöihin voitu lisätä nykyisistä laskentaympäristöistä tuttuja palveluita, joilla laajojen verkkojen, kuten esimerkiksi yrityksen sisäisen verkon, käyttöä voitaisiin hallita keskitetymmin. Keskitetty ohjelmistojen, tietokantojen ja dokumenttien hallita helpottaisi huomattavasti teknisten tukihenkilöiden työtä.

Oman virtuaalikoneen toteuttaminen on työläs ja erityisosaamista vaativa tehtävä. Virtuaalikoneen perustoiminta-ajatus ei sinänsä ole erityisen vaikea ja siten yksinkertaisia suoritusalustoja voidaan tehdä myös pienin resurssein. Nykyisten virtuaalikoneiden tarjoamat suorituspohjan ylittävät palvelut ovatkin sitten jo vaikeampia toteuttaa. Esimerkiksi kattavan ja turvallisen hiekkalaatikkomallin tekeminen on osoittautunut vaikeaksi tehtäväksi. Onneksi oman virtuaalikoneen suunnittelun pohjaksi on olemassa runsaasti julkista materiaalia, jota voidaan käyttää apuna suunnittelu- ja toteutusvaiheessa [50], [51], [52], [53], [54], [55]. Täysin omaa tuotantoa olevan virtuaalikoneen toteuttamista helpompi vaihtoehto on muokata jokin tarjolla olevista virtuaalikoneen rungoista vastaamaan omia tarpeita. Tällöin on tosin huomioitava alkuperäisen ohjelmiston lisenssointiehdoissa määrätyt rajoitteet.

Sun Microsystems julkaisi vuonna 1995 ensimmäisen kaupallisesti merkittävän virtuaalikonealustan [56]. Virtuaalikoneen liittäminen silloin suosituimpaan Netscape Navigator -selaimeen johti teknologian nopeaan yleistymiseen varsinkin verkkopohjaisissa sovelluksissa. Java-teknologia soveltuu kaikenlaisten ohjelmistojen kehittämiseen aina verkossa ajettavista sovelmista työpöytäsovelluksiin ja palvelinten laajennuksiin. Java:lla saavutettiin ensi kertaa käyttökelpoinen siirrettävyys, joka ei vaatinut ohjelmiston uudelleenkäännöstä, ja se saavutti nopeasti suuren suosion ohjelmoijien keskuudessa. Javan lopullista voittokulkua on hidastanut käyttäjien mielestä suurissa sovelluksissa puutteelliseksi jäävä suorituskyky. Julkistamisensa jälkeen monet ohjelmistoyritykset ilmoittivat siirtävänsä ohjelmistotuotantonsa uudelle alustalle. Käytännössä yksikään yrityksistä ei johtanut toimiviin tuotteisiin ja siten siirtyminen täysin virtuaalikoneisiin pohjautuvaan teknologiaan koki ainakin toistaiseksi haaksirikon. Osasyynä ohjelmistoyritysten epäonnistumiseen saattoi olla niiden vanhanaikaisissa toimintamalleissa, jotka eivät enää soveltuneet uudenlaiseen ohjelmistoarkkitehtuuriin. Java suunniteltiin

ennenkaikkea verkoissa toimivien sovelmien toteutusalustaksi. Keveät verkkosovelmat ovatkin osoittautuneet riittävän nopeiksi, eikä rajallinen suorituskyky ole muodostunut niitä rajoittavaksi tekijäksi.

Ohjelmistomarkkinoita suvereenisti hallitseva Microsoft on aina suhtautunut epäillen kilpailevien ohjelmistoyritysten tuotteisiin. Koska virtuaalikoneiden ymmärrettiin ajan kuluessa muodostavan merkittävän sovellusalustan, ei sitä voitu jättää huomiotta. Microsoftin yritettyä epäonnistuneesti tehdä Java:an epästandardeja laajennuksia, se päätti luoda itse vastaavan kilpailevan teknologian [57]. Tekniikkaa on hiottu jo vuosia ja määritelmien mukaan CLR tuleekin olemaan huomattavasti nykyistä Java-ympäristöä monipuolisempi. JVM:stä poiketen voidaan CLR:ään tuottaa binäärikoodia monilla erilaisilla ohjelmointikielillä. Koska CLR-teknologia julkaistaan vasta myöhemmin, ei sen ja Javan välillä voida vielä suorittaa varsinaisia vertailuja. Johtuen Microsoftin merkittävästä markkina-asemasta voidaan CLR:lle kuitenkin povata valtavaa menestystä, jos teknologia osoittautuu yhtä suorituskykyiseksi, kuin nykyinen Java. CLR:n voidaankin olettaa siirtävän ohjelmistotuotannon viimeinkin virtuaalikoneaikaan.

Virtuaalikoneilla toteutettavien sovellusohjelmien pohjaksi kannattaa ehdottomasti valita jokin vakiintuneista alustoista. Viimevuosina virtuaalikoneiden tarjonta on kasvanut nopeasti. Suurin osa tarjolla olevista alustoista on keskittynyt tietyn ongelman ratkaisemiseen ja vain muutama tarjolla olevista vaihtoehdoista soveltuu kaikentyyppisten sovellusten toteuttamiseen. Varsinaisen sovellusohjelmoinnin vaiheet ovat samat kuin systeemiohjelmoinnissakin. Ohjelmiston kehitys aloitetaan huolellisella suunnittelulla ja päätetään toiminnallisuuden tarkistuksella. Jo suunnitteluvaiheessa on syytä ottaa huomioon suorituskyvyn aiheuttamat rajoitteet sovelluksen toimintoihin. Paljon algoritmeja sisältävät ja laskennallisesti raskaat sovellukset voi edelleen olla järkevämpää toteuttaa systeemiohjelmoinnin menetelmin. Virtuaalikoneet soveltuvat parhaimmin keveiden käyttöliittymien, tietokantojen ja tietoverkkoja hyödyntävien sovellusten toteutukseen. Tietoverkoissa käytettävien sovelmien alustana virtuaalikoneet ovat vertaansa vailla.

Virtuaalikoneilla tehtyihin sovelluksiin voidaan yleensä helposti toteuttaa dynaamiseen komponenttien sidontaan perustuva laajennusväylä. Uudet ominaisuudet laaditaan komponenteiksi, joilla on ennalta sovittu rajapinta. Sovelluksen käynnistyessä se lataa määritelmän mukaisesti sovelluksen ulkopuolisia komponentteja ja liittää ne osaksi sovelluksen rakennetta. Parhaimmillaan laajennukset liittyvät saumattomasti osaksi sovelluksen käyttöliittymää, jolloin loppukäyttäjä ei välttämätä edes tiedä mitkä toiminnot on toteutettu sovelluksessa ja mitkä laajennuksilla. Laajennusten toteutukseen tarvitaan varsinaisen sovelluksen ohella luonnollisesti myös virtuaalikoneen ohjelmointiin käytettävä kehitysympäristö.

Systeemiohjelmointikielillä tehtyihin sovelluksiin voidaan liittää osaksi pienimuotoinen virtuaalikone, jolloin sovellusta voidaan räätälöidä toteuttamalla laajennukset käytetyn virtuaalikoneen sovelluksina. Menetelmä ei ole yhtä elegantti kuin täysin virtuaalikoneisiin pohjautuva, mutta tarjoaa silti monia hyviä puolia. Sopivilla valinnoilla voidaan aikaansaada lähes systeemiohjelmointikielten tarjoama suorituskyky säilyttäen silti virtuaalikoneiden tarjoama siirrettävyys ja laajennettavuus.

Luku 4

Neural Data Analysis -ohjelmisto

Ohjelmistotekniikan ja laskennallisesti älykkäiden järjestelmien tutkimusryhmässä on useiden vuosien ajan kehitetty ohjelmistoteknisiä sovelluksia, jotka perustuvat tutkimuslähtöisiin, laskennallisesti älykkäisiin järjestelmiin. Alunperin tarvittavat ohjelmistot toteutettiin lähes puhtaalta pöydältä, mistä aiheutui turhaa ohjelmointityötä, joka olisi voitu vältää luomalla uudelleenkäytettäviä ohjelmistoja. Erillisten ohjelmistojen ylläpito kulutti lisäksi tarpeettomasti muihin tehtäviin tarvittavia resursseja. Ongelman ratkaisemiseksi tutkimusryhmässä laadittiin 1990-luvun puolivälissä suunnitelma yleiskäyttöisen ohjelmistoalustan kehittämiseksi, jonka avulla tutkimuskäyttöön tarvittavia ohjelmistoja voitaisiin luoda nopeammin mukauttamalla alustan palvelut käyttökohteen tarpeisiin. NDA-kehitysympäristöksi nimettyä ohjelmistoa on sittemmin käytetty noin tusinan ohjelmiston toteuttamiseen ja uusia ohjelmistoja toteutetaan edelleen. Seuraavissa alaluvuissa käydään läpi NDA:n toteutusta edellisessä luvussa määriteltyjen ohjelmistotuotannollisten käsitteiden puitteissa. Ohjelmiston kuvaus perustuu lähes vuoden pituisen käyttökokemuksen ohella ohjelmiston rakennetta käsittelevään Erkki Häkkisen väitöskirjaan [58].

4.1 Siirrettävyys

NDA-ohjelmiston suunnittelun tavoitteena oli kehittää ohjelmisto, joka voitaisiin helposti siirtää useimmille laitteistoalustoille. Tavoitteen saavuttamiseksi valittiin ohjelmiston toteutusmenetelmäksi tee-se-itse -malli, joka soveltuu parhaiten juuri algoritmipainotteisten ohjelmistojen toteuttamiseen. Ohjelmiston toteutuskielenä on ANSI C, jonka katsottiin tarjoavan parhaan siirtyvyyden. Vaikka kielellisesti edistyneempiä ANSI C++-kielisiä ohjelmointityökaluja olikin jo saatavilla, katsottiin näiden vielä rajoittavan ohjelmiston

siirrettävyyttä.

Ohjelmiston toteutusmallina on kerrosarkkitehtuuri. Ohjelmiston lähdekoodit on jaettu sopivan kokoisiin moduleihin, jolloin ohjelmiston käännösvaiheessa voidaan valita tapauskohtaisesti tarvittavat palvelut. Ohjelmisto käännetään yhdeksi konekieliseksi kirjastoksi, joka voidaan liittää osaksi varsinaista sovellusohjelmaa.

Paremman siirrettävyyden aikaansaamiseksi rakenteellisen tiedon järjestelyyn tarvittavan ohjelmistologiikan hallinta on sisällytetty kirjaston palveluihin. Vaihtoehtoisen tiedonhallintajärjestelmän *(engl. DBMS)* katsottiin aiheuttavan ohjelmiston siirrettävyyden ohella taloudellisia ongelmia nykyisten laadukkaiden tietokantaohjelmistojen ollessa varsin kalliita.

Toimivan ja monipuolisen käyttöliittymän toteutus on osoittautunut siirrettävyyden kannalta kaikkein vaikeimmaksi toteuttaa. Koska graafisten käyttöliittymien toteuttamiseksi ei toistaiseksi ole tarjolla yhtään standardia, joka olisi toteutettu sekä Unix- että Windows-ympäristöissä, on ongelman ratkaisu siirretty toistaiseksi kirjaston ulkopuolelle, jolloin jokaisessa sovellusohjelmistossa on jouduttu toteuttamaan oma käyttöliittymä.

Ensimmäinen yritys yhtenäisen graafisen käyttöliittymän luomiseksi toteutettiin Sun Microsystems:in Java-kehitystyökalulla [61]. Laadittu ohjelmisto osoittautui kuitenkin epäkäytännöllisen hitaaksi eikä se yleistynytkään toivotulla tavalla. Riittävän vapaamuotoisen, siirrettävän ja laajennettavan käyttöliittymän luomiseksi on viime aikoina virinnyt ajatus täysin tee-se-itse -menetelmään pohjautuvasta toteutuksesta.

4.2 Yhteensopivuus

Kehitysympäristön yhteensopivuus muiden ohjelmistojen kanssa on toteutettu matalan tason palveluilla. Tietojenkäsittelyn parissa data-aineistot esitetään yleensä ASCII-standardin mukaisesti koodatuissa tekstitiedostoissa. Koska lähes kaikki ohjelmistot kykenevät sekä avaamaan että tallentamaan tietoa tässä muodossa, ei ohjelmistojen yhteistoiminnasa yleensä ole ongelmia. Jos käytettävyyttä haluttaisiin nostaa tätä korkeammalle tasolle, voitaisiin ohjelmistoon tuottaa valmiit tiedostonkäsittelijät useimmin käytetyille tiedostomuodoille.

Tietojenkäsittelyn alalla on muutamia vakiintuneita ohjelmistoja, joita lähes kaikki alalla työskentelevät voivat halutessaan käyttää. Matemaattisten ongelmien ratkaisuun erikoistunut ja tässä työssä jo aiemminkin esitelty MATLAB on yksi merkittävimmistä alalla käytettävistä työkaluista. Yhteistoimintaa varten on NDA-kirjastosta käännetty erityinen MATLAB:in tunnistama MEX-versio, jolloin kirjaston tuottamia palveluita voidaan hyödyn-

tää myös MATLAB:illa toteutetuissa ohjelmistoissa. Yhteistoimintaa tarvitaan lähinnä silloin kun sovelluskohteessa tarvitaan palveluita, joita ei ole vielä saatavilla NDA-kirjastosta.

4.3 Käytettävyys

Yksi NDA:n kehittämisen päätavoitteista oli tehdä taloudellisesti käytettävissä oleva työkalu, jolla voidaan tarvittaessa korvata kalliita kaupallisia tutkimusohjelmistoja, kuten SAS, S-plus ja MATLAB. Kehitysympäristön suunnittelussa on panostettu lähinnä tehokkaan asiantuntijakäyttöliittymän luontiin. Laadittu kirjoitettaviin komentoihin pohjautuva ydin muodostaakin tehokkaan ympäristön kokeneelle NDA-käyttäjälle. Koska tulevaisuudessa tekoälyn uskotaan leviävän myös kuluttajille tarkoitettuihin ohjelmistoihin, ei kaikilta käyttäjiltä voida vaatia tarkkaa komentokielen tuntemusta. Tarkoitusta varten ollaankin laatimassa erilaisia graafisia käyttöpaneeleja, joilla monimutkainen komentokieli voidaan piilottaa loppukäyttäjältä.

Tiedon louhinta muodostuu yleensä iteratiivisista toimenpiteistä, joissa käyttäjän antamat ohjeet ja tietokoneen laskemat tulokset vuorottelevat. Erityisesti useissa sovelluksissa käytettävä SOM-algoritmi vaatii esityskelpoisten tulosten saavuttamiseksi käyttäjän palautetta. Ohjelmiston tarkoituksena on laatia valmiiksi joukko vaihtoehtoja, joista käyttäjä muokkaa lopullisen esityksen. Louhittaessa tuntematonta materiaalia, jolloin käytäjällä ei ole tietoa etsimästään informaatiosta, on edellisiä työvaiheita suoritettava kunnes data-aineistosta löydetään käyttäjää kiinnostavaa informaatiota.

Tutkimuksessa tuotettavia dokumentteja varten voidaan ohjelmistolla tuottaa laaditusta grafiikasta PostScript-muotoisia tiedostoja, jotka voidaan lisätä useimmilla tekstinkäsittelyohjelmistoilla suoraan tekstin osaksi, jolloin lukija pystyy seuraamaan kuvista saavutettuja tuloksia. Ominaisuus on erityisen hyödyllinen kirjoitettaessa esimerkiksi tieteellisiä julkaisuja Latex-ohjelmistolla. Kuvatiedostoihin sisällytetään myös ylimääräistä informaatiota, jolloin vektorigrafiikasta koostuvia kuvatiedostoja voidaan tarvittaessa käsitellä esimerkiksi ivtools-työkalupakettiin [62] kuuluvalla idraw-sovelluksella.

4.4 Laajennettavuus

Eräs NDA-ympäristön suunnittelun tavoitteista on ollut helppo laajennettavuus. Tämän perustana toimii muistinhallintajärjestelmä. jonka päälle varsinaiset palvelut on toteutettu.

Uusien algoritmien lisäys onnistuu helposti. Lisättävät palvelut toteutetaan ANSI C-kielellä ja sijoitetaan projektihakemistossa käyttäjälle tarkoitusta varten luotuun hakemistoon. Ohjelmiston valmiiseen lähdekoodiin tarvitaan vain pieniä muutoksia ja nekin voidaan suorittaa keskitetysti yhteen paikkaan. Tämän jälkeen uudet lähdekoodimodulit lisätään projektitiedostoon ja kirjastosta käännetään uusi versio.

Ohjelmistoa on mahdollista laajentaa myös makrokielisillä kooditiedostoilla, jotka sijoitetaan ohjelmiston hakupolkuun. Makrotiedostot soveltuvat lähinnä pitkien analyysiketjujen toteuttamiseen, joissa käytetään yleensä useita eri algoritmeja. Makrojen toteutus ei eroa varsinaisen komentokielen käytöstä ja siten makrojen tuottamiselle ei muodostu samanlaista kynnystä, kuten monissa muissa ohjelmistoissa.

Laadittaessa sovelluksia, joissa tarvitaan kirjaston palvelut ylittäviä matemaattisia ominaisuuksia, voidaan kirjasto saattaa yhteistoimintaan MATLAB-ympäristön kanssa. Tällöin voidaan käyttää kummankin ohjelmiston palveluita keskitetysti MATLAB:ista käsin.

4.5 Uudelleenkäytettävyys

Koska NDA:n kehityksen lähtökohtana oli ohjelmien uudelleenkäytön vaikeus, on tämän edistämiseen kiinnitetty huomiota. Edellisessä alaluvussa mainittu muistinhallinta, johon liittyy nimiavaruus eli nimettyjen tietorakenteiden käyttö, mahdollistaa algoritmien välisen kommunikoinnin ilman eksplisiittistä tiedosto- tai muistiosoitteita, jolloin saavutetaan hyvä suorituskyky sekä geneerinen toiminnallisuus.

Useiden erilaisten tietostorakenteiden tilalle on luotu keskitetty tiedonhallintajärjestelmä, jolloin algoritmit voivat kommunikoida keskenään ilman tuotettujen data-aineistojen kirjoittamista tiedostoiksi ja samalla saavutetaan huomattavasti parempi suorituskyky.

ANSI C-kielellä toteutetut algoritmit ovat yleensä sellaisenaan uudelleenkäytettäviä muissa ohjelmistoissa. Koska ohjelmistoa käytetään lähinnä työasemissa, ei algoritmien erottamiseen varsinaisesta ohjelmistosta ole tarvetta, vaan ohjelmistosta voidaan tuottaa tarkoitukseen sopiva versio.

Makrokielisiä ohjelmistolaajennuksia voidaan uudelleenkäyttää niiden soveltuvuuden rajoissa. Tämän työn tuottamisen aikana on meneillään kehitysprojekti puutteellisen datan täydentämisessä eli imputoinnissa esiintyvien ongelmien ratkaisemiseksi automaattisella prosessilla. Esimerkin makrokielisiä laajennuksia voidaan uudelleenkäyttää tulevien sovellusohjelmien toteutuksessa.

4.6 Elinkaari

NDA-kehitysympäristö kuuluu jatkuvasti kehittyvien pitkäaikaisten elinkaarten joukkoon. Työn toteutushetkellä ohjelmistosta on olemassa lähinnä kehitysolosuhteisiin soveltuva versio, joka sisältää vielä virheitä, jotka saattavat häiritä käyttäjää. Ohjelmistoa kehitetään kuitenkin jatkuvasti korjaamalla virheitä, lisäämällä uusia ominaisuuksia ja parantamalla sen käytettävyyttä erilaisilla graafisilla käyttöpaneeleilla.

Luku 5

Arkkitehtuurisia valintoja

Tässä luvussa tehdään ohjelmistoprojektiin liittyviä alustavia suunnitelmia menemättä kuitenkaan toteutukseen liittyviin yksityiskohtiin.

5.1 Kehitysympäristön valinta

Sopivan kehitysympäristön valitsemiseksi käytiin läpi muutamia varteenotettavia vaihtoehtoja. Varsinainen NDA-kirjasto on jo ennalta toteutettu ANSI C-kielellä, joten valittavan kehitysympäristön tulee voida käyttää hyväkseen standardeilla työkaluilla laadittuja, käyttöjärjestelmästä riippuvia, dynaamisesti linkitettäviä kirjastoja. Työn ollessa selvästi käyttöliittymän kehitykseen painottuvaa, suositaan työkaluvalinnassa tähän tarkoitukseen tehtyjä nopean kehityksen *(engl. Rapid Application Development)* -kehitysympäristöjä. Yksityiskohtaisemmat vaatimukset ovat:

1. Käytettävän toteutusteknologian tulee olla mahdollisimman edullisesti ja nopeasti hankittavissa. Lisäksi annetaan arvoa sille, että teknologia olisi yleisesti tunnettu ja siten nopeasti työskentelevän henkilöstön käytettävissä.

2. Käytettävän toteutusteknologian tulee olla mahdollisimman hyvin siirrettävissä toisille alustoille. Kehitystyökalun on tuettava vähintään Windows ja Linux -ympäristöjä. Tuki Solaris-käyttöjärjestelmälle katsotaan työkalulle eduksi, mutta ei kuitenkaan välttämättömäksi.

3. Käytettävän toteutusteknologian tulee olla hyvin tuettu ja sen jatkokehityksen tulisi olla turvattu. Työkalun valinnassa suositaan jo vuosia käytössä olleita ja käytön aikana monipuolisiksi ja vakaiksi havaittuja työkaluja.

4. Käytettävän toteutusteknologian tulee mahdollistaa modulaarinen ohjelmistontuotanto ja edistää laadittujen ohjelmistokomponenttien uudelleenkäyttöä. Uudelleenkäytön integrointi kehitysympäristön käyttöliittymään katsotaan työkalulle eduksi.

Käytettävissä olevista työkaluista Borland Corp.:in Delphi/Kylix -ympäristö antaa parhaan ensivaikutelman. Tarkemmin valintaan vaikuttaneita seikkoja on käyty läpi seuraavissa luetteloissa.

Hyvät puolet:

1. Selvästi käytettävissä olevista tekniikoista helpoin ja nopein laatia.
2. Tukee vahvasti toteutettujen ohjelmistokomponenttien uudelleenkäyttöä.
3. Lähdekoodit toimivat suoraan tai pienin muutoksin sekä Microsoft Windowsissa että Linuxissa.
4. Kehitysympäristö on hyvin vakiintunut ja sen jatkokehitys on turvattu lähitulevaisuudessa.
5. Kehitysympäristön hyvä tuntemus Jyväskylän yliopiston opiskelijoiden keskuudessa johtuen Vesa Lappalaisen viimevuosina pitämästä graafisten käyttöliittymien kurssista.
6. Käyttöliittymä voidaan piirtää Delphin monipuolisella ja helppokäyttöisellä kehitysympäristöllä.
7. Prosessoriläheinen binäärikoodi toimii nopeammin kuin C# tai Java -pohjainen virtuaalikoneeseen pohjautuva binäärikoodi.

Huonot puolet:

1. Arkkitehtuuri on lujasti sidottu Borland Corp.:in Delphi/Kylix -konseptiin. Siirtyminen korvaavaan järjestelmään on liki mahdotonta.
2. Ohjelmointi on suoritettava Object Pascal -syntaksilla, eikä suositummilla, paremmin tuetuilla ja osatuilla kuten C++, C# tai Java.
3. Prosessoriläheinen binäärikoodi toimii vain siinä ympäristössä, jossa se on käännetty.
4. Tutkimuskäytössä yleiselle Solaris-käyttöjärjestelmälle ei ole tarjolla minkäänlaista tukea.

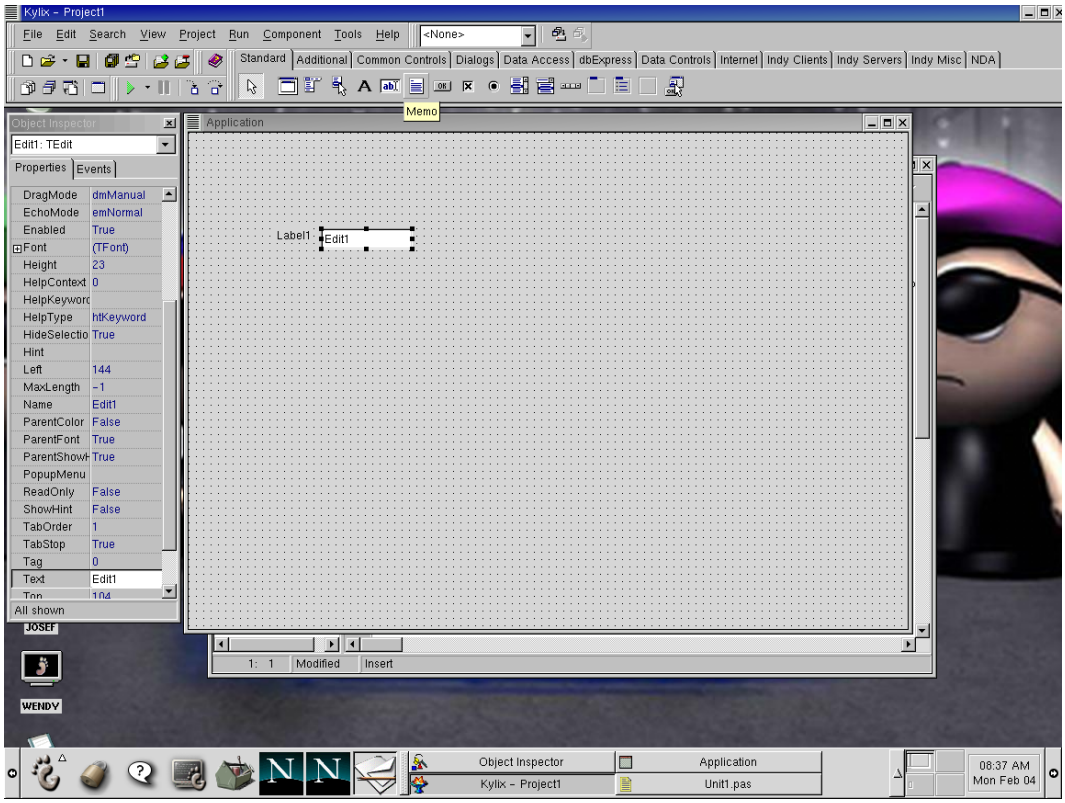
Perinteisten systeemiohjelmointityökalujen, kuten GCC, Visual C++ ja Borland C++, katsottiin soveltuvan huonosti käyttöliittymien toteutukseen. Lisäksi näillä tuotettu lähdekoodi ei olisi muodostunut siirrettäväksi eri kohdeympäristöjen välillä. Skriptikielten katsottiin mahdollistavan tarvittavan siirrettävyyden, mutta toisaalta rajoittavan ohjelmiston toteutusta liiaksi. Lisäksi puutteellisen suorituskyvyn uskottiin muodostuvan ennenpitkää ongelmalliseksi. Viimeaikoina paljon esillä olleen Java-kehitysympäristön ei katsottu aikaisemmista ohjelmistoprojekteista saatujen kokemusten perusteella vielä vastaavan kaikkia käytettävään työkaluun kohdistuvia odotuksia. Varsinkin puutteelliseksi jäänyt suorituskyky on vaikeuttanut sovellusten käyttöä. Kehitysympäristön valinnassa päädyttiin siten kompromissiin, jossa yhdistyvät toisaalta systeemiohjelmoinnin vapaa ja tehokas toteutus, ja skriptikielten parempi tuottavuus.

5.2 Kehitysympäristön esittely

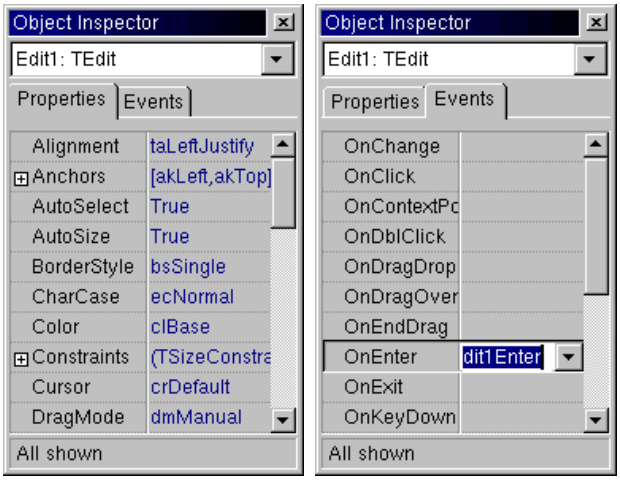
Kehitysympäristön käyttöliittymä noudattaa nopean kehityksen työkaluista tuttua suunnittelumallia. Sovellusohjelmat kootaan pudottamalla lomakkeelle valmiita komponentteja. Komponenttien asetukset suoritetaan attribuuttitaululla ja tapahtuman käsittelijät määritellään tapahtumataululla. Varsinaista ohjelmointikielellä tapahtuvaa ohjelmointia tarvitaan vain ennalta esiteltyjen tapahtumankäsittelijöiden toiminnan kuvaamiseen. Kehitysympäristöllä suoritettavaa sovelluskehitystä on havainnollistettu kuvissa 5.1, 5.2 ja 5.3.

Ohjelmointikielenä käytetään Borland Corp.:in työkalua varten kehittämää Object Pascal -syntaksia. Object Pascal perustuu standardiin Pascal-syntaksiin, jota on laajennettu monilla rakenteellisilla ominaisuuksilla. Object Pascal on täysiverinen olio-ohjelmointikieli, jolla voidaan kuvata moniperintää lukuunottamatta kaikki olio-ohjelmoinnissa tarvittavat rakenteet. Vahvasti komponenttien käyttöön perustuvana Object Pascal sisältää muutamia valmiita komponenttirakenteita. Esimerkiksi komponenttien attribuuttien saantimetodit voidaan integroida suoraan muuttujan nimeen, jolloin ohjelmoija voi viitata suoraan kyseiseen muuttujaan ja kääntäjä valitsee sopivan metodin käytettävissä olevien metodien joukosta. Monet työkaluun tutustuneet pitävät Object Pascal:ia yhtenä parhaimmista komponenttiohjelmointikielistä.

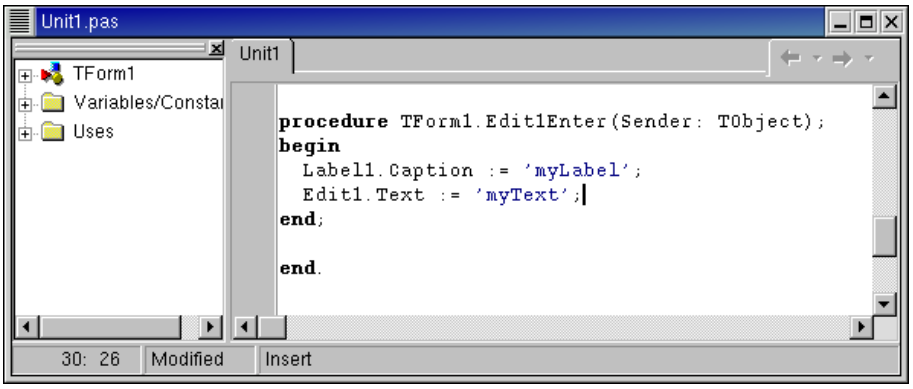
Kehitysympäristön yksi suurimmista valteista on sen kattava, laajennettava ja helppokäyttöinen komponenttikehys. Kehys sisältää valmiit työkalut yleisimmille käyttöjärjestelmän palveluille, graafiselle käyttöliittymälle, relaatiotietokantayhteydelle ja tietoverkkojen hallinnalle. Kehyksen tarjoamat



Kuva 5.1: Kylix sovelluskehitysympäristö.



Kuva 5.2: Komponentin attribuuttien ja tapahtumien hallinta.



Kuva 5.3: Tapahtumien kuvaaminen Object Pascalilla.

palvelut riittävät lähes kaikenlaisten sovellusten tuotantoon ja siten omien komponenttien kehitystä ei yleensä tarvita. Työkalun peruspalveluvalikoimaa voidaan parantaa myös tietoverkoista saatavilla komponenteilla, joita on saatavilla tuhansittain.

Uusien komponenttien rakentaminen on tehty erittäin helpoksi. Komponenttivelho luo ohjelmoijalle valmiin rungon, johon tämä lisää haluamansa attribuutit, metodit ja tapahtumankäsittelijät. Ohessa on esimerkki yksinkertaisen komponentin rajapinnan esittelyosaan kuuluvasta lähdekoodista.

```
TMyComponent = class(TComponent)
private
protected
  myName: String;
  procedure myEvent(Sender: TObject);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure SetName(Name: String);
published
  property Name: String read myName write SetName;
end;
```

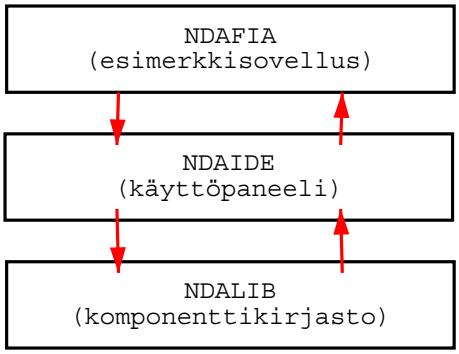
Komponenttien hallinnan helpottamiseksi komponentit kerätään paketeiksi eli komponenttikirjastoiksi. Paketit ovat käytön helpottamiseksi funktiokirjastoja pidemmälle tuotteistettuja. Jokainen paketti osaa asentaa itsensä osaksi kehitysympäristön käyttöliittymää, eikä työkalun mukana toimitettujen ja omatekoisten komponenttien käyttö eroakaan toisistaan. Jokainen

komponentti sisältää kuvauksen rajapinnastaan. Lisäksi mukana on pieni bittikartta komponenttipalettiin sijoitettavaksi. Paketeissa olevat komponentit ovat valmiiksi ajettavassa muodossa ja sovelluksen kehityksen aikana komponentteja ajetaan aivan samaan tapaan kuin lopullisessa sovelluksessa. Näin sovelluksen toimintaa voidaan tarkastella jo sen kehitysvaiheessa.

Kehitysympäristö osaa liittää projekteihin niin stattisesti kuin dynaamisesti sidottavia kirjastoja. Ulkoisten palveluiden käyttöönotto on helppoa ja niiden esittely ei juurikaan poikkea tavallisten funktioiden esittelystä. Dynaamisesti sidottavien kirjastojen yhteydessä on kirjasto luonnollisesti esiteltävä käyttöjärjestelmälle, ennenkuin sen palveluita voidaan hyödyntää.

5.3 Ohjelmiston arkkitehtuuri

Vaatimusmäärittelyn tavoitteiden saavuttamiseksi ohjelmiston toteutus päätettiin jakaa kuvan 5.4 mukaisesti kolmeen loogisesti erilliseen kokonaisuuteen.



Kuva 5.4: Ohjelmistotyön vaiheet.

Ensimmäisen työvaiheen muodostaa Delphi/Kylix -ympäristöön toteutettava komponenttikirjasto (NDALIB). Komponenttien suunnittelussa noudatetaan työkalun oman komponenttikirjaston toteutuksessa käytettyjä periaatteita. Näin laadittavista komponenteista muodostuu automaattisesti helposti uudelleenkäytettäviä niiden integroituessa suoraan osaksi kehitysympäristöä.

Toinen työvaihe muodostuu graafisen käyttöpaneelin (NDAIDE) luomisesta. Ohjelmisto muodostaa muista ohjelmistonkehitysympäristöistä riippumattoman työkalun analyysiketjuihin pohjautuvien sovellusten kehitykseen. Ohjelmisto rakennetaan käyttäen Delphin/Kylixin mukana toimitetta-

vaa CLX-komponenttikirjastoa sekä ensimmäisessä työvaiheessa kehitettyjä omia komponentteja.

Kolmannessa työvaiheessa luodaan edellisen työvaiheen käyttöpaneelin ohjelmointia havainnollistava esimerkkisovellus (NDAFIA). Esimerkkisovellukseksi päätettiin kehittää uusi versio jo aiemmin toteutetusta kuituanalyysiohjelmistosta.

Ohjelmisto ositettiin jakamalla ohjelmistorakenteet sopivan kokoisiin moduleihin. Komponenttien jako tapahtuu lähes automaattisesti kunkin komponentin käsittäessä tarkasti rajatun käyttöalueen. Käyttöpaneelin osalta jako suoritetaan toteuttamalla ohjelma useista ikkunoista, joissa kussakin on sopivaksi katsottava määrä toiminnallisuutta. Esimerkkisovelluksen osalta ohjelmistorakenne muodostuu käytännössä automaattisesti modulaariseksi jokaisen sovelluslogiikkapalasen muodostaessa oman hyvin määritellyn kokonaisuuden.

NDA-kirjaston liityntärajapinta eristetään tarkasti muusta ohjelmistosta. Käytännössä tämä tehdään keskittämällä kirjastoläheinen ohjelmointi yhteen komponenttiin, jonka läpi kaikki NDA-ytimen ja sovellusohjelman välinen tietoliikenne suoritetaan.

Valitun ohjelmistonkehitysympäristön tarjoama komponenttimalli soveltuu erinomaisesti ohjelmistonkehityksen tarpeisiin. Kehitysympäristön mukana toimitetaan riittävä määrä komponentteja vaativienkin käyttöliittymien toteuttamiseen. Peruskomponenttien lisäksi laaditaan tarvittava määrä uusia NDA-kirjaston käyttöön liittyviä komponentteja, jolloin varsinainen sovellusohjelmointi voidaan toteuttaa lähes täysin valmiita komponentteja yhdistelemällä.

Komponenttikirjaston ja käyttöpaneelin tarpeisiin päätettiin määritellä kaksi uutta makrokieltä. Vaihtoehtoisista malleista kielen tulkkaamisen katsottiin skaalautuvan parhaiten yksinkertaisten kielten tarpeisiin. Vaihtoehtoina olleet kääntäjä- ja virtuaalikoneteknologiat olisivat olleet liian työläitä toteuttaa opinnäytteenä, eikä niiden tarjoamalle suuremmalle ilmaisuvoimalle ollut käytännössä tarvetta. Tulkattavan esityksen myötä saavutetaan myös hyvä siirrettävyys, sillä ASCII-muotoinen rakenne ei ole binääriesityksen tapaan herkkä laitteistojen välisille eroille.

Luku 6

Komponenttikirjasto

Tässä luvussa kuvataan ensimmäisenä toteutettava ja kaiken jäjempänä esitellyn ohjelmistotyön pohjan muodostava Delphi/Kylix -komponenttikirjasto.

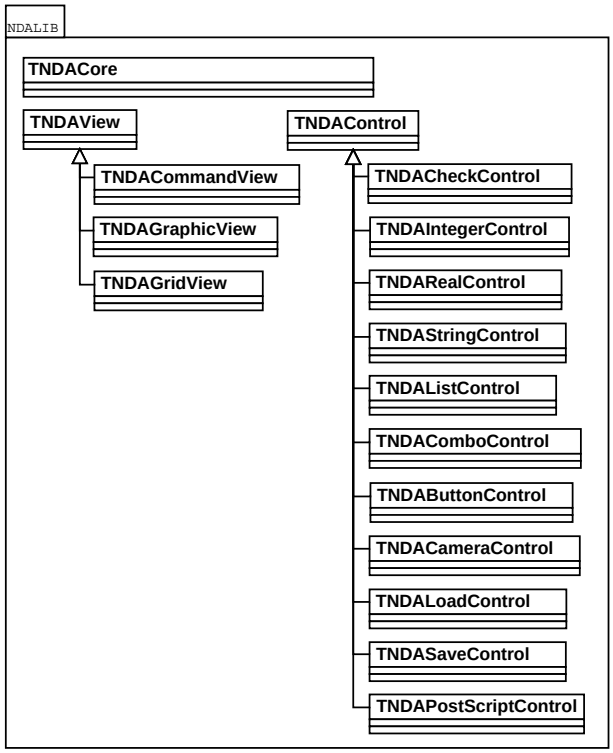
6.1 Kuvaus

NDA-sovelluksien ohjelmoiminen on aiemmin aloitettu puhtaalta pöydältä, jolloin NDA-kirjaston liittäminen sovellukseen jouduttiin suorittamaan käsin. Lisäksi graafisten esitysten visualisointiin tarvittavan komentotulkin laatiminen oli varsin työläs operaatio. Komponenttikirjasto luo mahdollisuuden toteuttaa Delhi/Kylix -sovelluksia ilman kirjastoläheistä ohjelmointia, jolloin liityntärajapinnan tarpeeton uudelleenkirjoitus voidaan välttää. Pitkälle automatisoiduista palveluista huolimatta kirjaston käyttö ei millään tavalla rajoita NDA-kirjaston käyttömahdollisuuksia, vaan ohjelmoijalla on tarvittaessa koko ajan käytettävissään suora yhteys NDA-ytimeen.

Tehtävään valittu Delphi/Kylix -ympäristö sisältää valmiin rakenteen uusien komponenttien tuottamiseen. Laadittavat komponentit jaetaan kolmeen ryhmään niiden käyttötarkoituksen perusteella. Komponenttikirjaston rakennetta on esitelty kuvassa 6.1. Kirjaston rakennetta tarkemmin esittävä UML-kielinen kuvaus löytyy liitteestä A.

6.1.1 NDA-rajapintakomponentti

Kommunikoidakseen NDA-kirjaston kanssa on sovelluksen ennen tätä suoritettava monia alustavia toimenpiteitä. Käyttöönoton nopeuttamiseksi ja samalla luotettavuuden parantamiseksi on koko kommunikaatioliikenne NDA-kirjaston ja komponenttikirjaston välillä automatisoitu mahdollisimman pit-



Kuva 6.1: Komponenttikirjaston luokat.

källe.

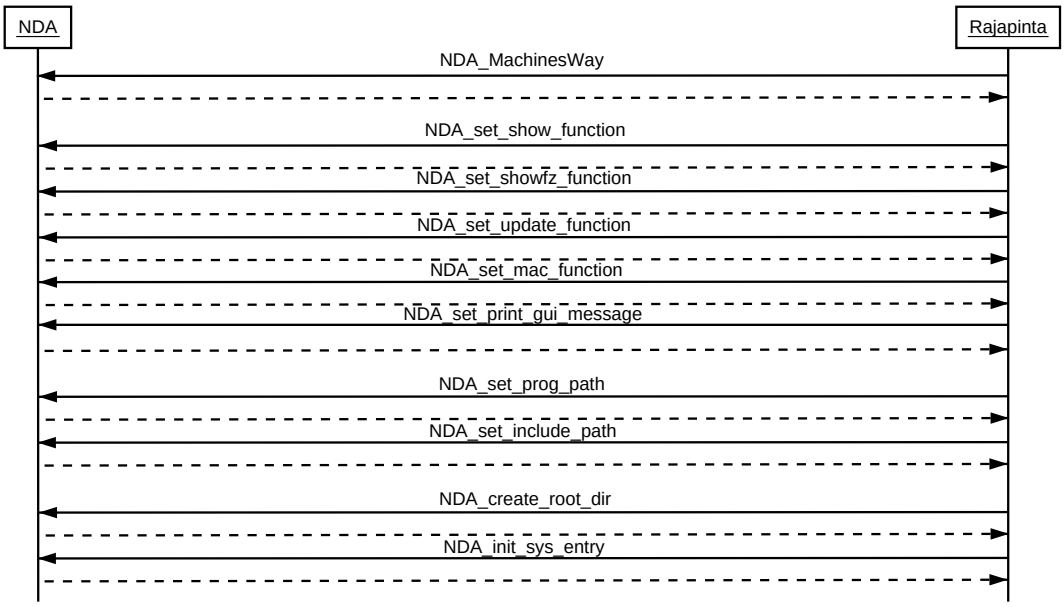
TNDACore

```
+Create(AOwner:TComponent): constructor; override;
+Destroy(): destructor; override;
+Init(Path:String,ProgPath:String,IncludePath:String): procedure;
+RunCommand(Command:String): procedure; overload;
+RunCommand(Command:String,var Result:TStringList): procedure; overload;
```

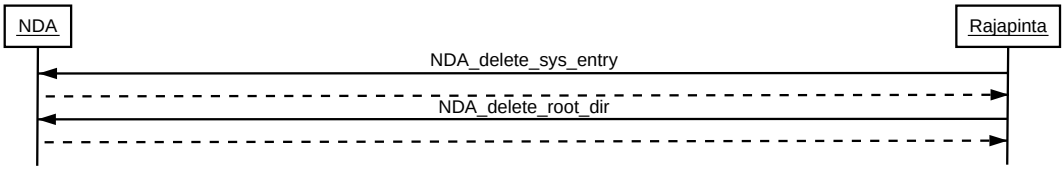
Kuva 6.2: NDA-rajapinnan muodostava komponentti.

Rajapinnan avaamiseen, sulkemiseen ja ajonaikaiseen tietojen vaihtoon tarvittavat palvelut kerätään yhteen kuvan 6.2 mukaiseksi `TNDACore`-komponentiksi. Komponentin konstruktori avaa ja destruktori sulkee kirjastoyhteyden. Komponentin `RunCommand`-metodit muodostavat NDA:n ohjaamiseen tarvittavan komentotien.

NDA-rajapinta avataan sovelluksen alustusvaiheessa kutsumalla avausfunktioita kuvan 6.3 mukaisessa järjestyksessä. Sovelluksen käytön päättyes-



Kuva 6.3: NDA-rajapinnan avaaminen.

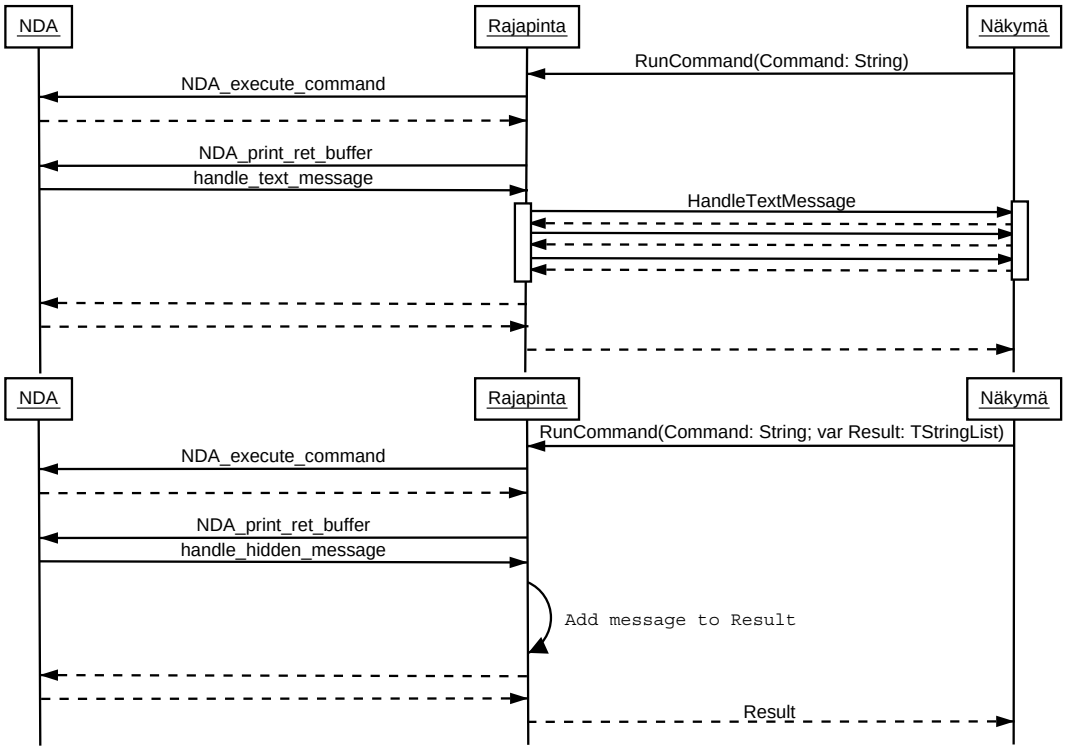


Kuva 6.4: NDA-rajapinnan sulkeminen.

sä on NDA-rajapinta suljettava. Sulkeminen suoritetaan kutsumalla sulkufunktioita kuvan 6.4 esittämällä tavalla.

Tietoliikenne NDA-ytimen ja komponenttikirjaston välillä voi tapahtua kahdella tavalla. Ensimmäisessä vaihtoehdossa suoritetaan merkkijonoon asetettu NDA-kielinen komento välittämättä mahdollisista vasteista. Vasteetonta vaihtoehtoa käytetään lähinnä näkymissä, joissa vaste saavutetaan yleensä tapahtumankäsittelyn kautta. Toisessa vaihtoehdossa suoritetaan merkkijonoon asetettu NDA-kielinen komento ja ytimeltä saatu vaste kerätään talteen. Vasteita parsimalla voidaan siirtää informaatiota ytimeltä käyttäjälle. Tietoliikennettä on havainnollistettu kuvassa 6.5.

NDA-ydin voi lähettää sovelluksen suoritettavaksi tapahtumia riippumatta sille lähetetyistä komennoista. Tapahtumien käsittelymekanismit on esitetty kuvassa 6.6. NDA:lta tuleva viesti kerätään ensin rajapinnan muistiin,

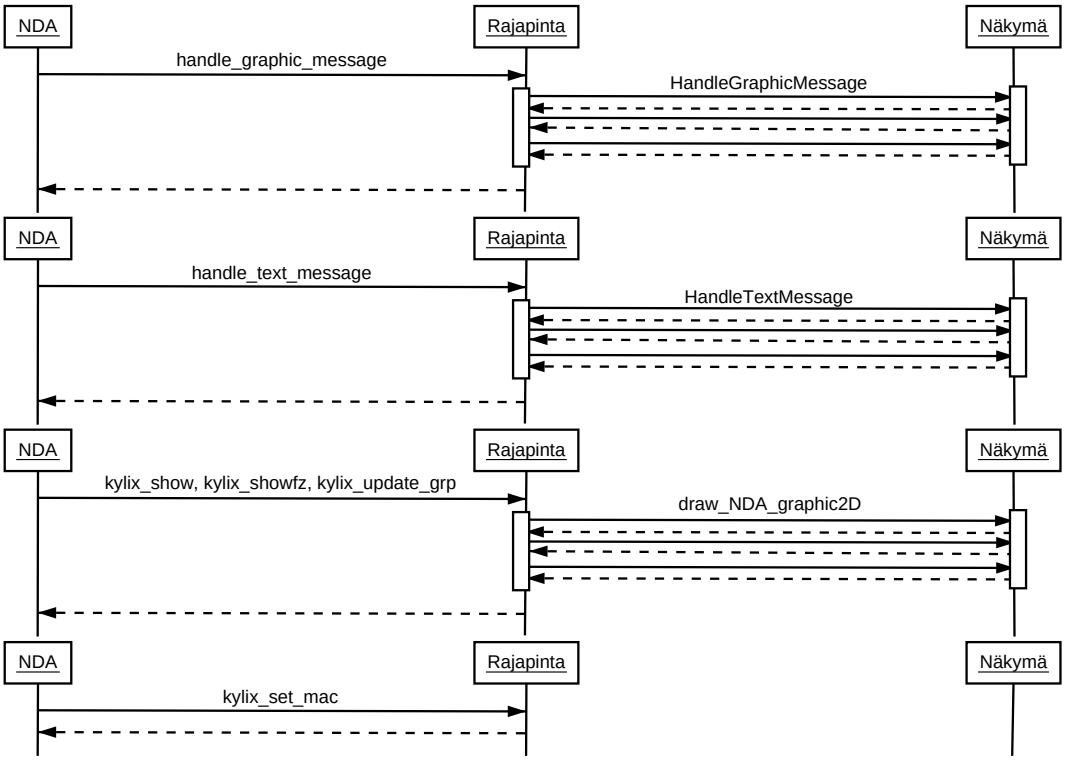


Kuva 6.5: Tiedon siirto NDA-ytimen ja sovelluksen välillä.

josta se sitten lähetetään kaikille asiaan kuuluville näkymille. Jokaisella näkymällä on oma tapahtumankäsittelijänsä viestin lopullista käsittelyä varten. Grafiikan piirtoon liittyvät viestit lähetetään muista poiketen vain asianosaisille näkymille.

Näkymät toimivat lähettämällä palvelupyyntöjä rajapinnalle, joka siirtää ne edelleen NDA-ytimen suoritettavaksi. Palveluista saatava vaste jätetään yleensä käsittelemättä ja varsinainen paluuinformaatio saadaan tapahtumien muodossa. Työkalut tarvitsevat toimiakseen yleensä kaksisuuntaisen yhteyden NDA-ytimeen. Lähetettävällä komennolla tehdään muutoksia muuttujiin ja asetuksiin, ja palautteena saatuja vasteita käytetään vallitsevan tilanteen havainnoimiseen. Sovellusohjelmoija voi halutessaan viestiä suoraan NDA-ytimen kanssa kutsumalla itse `RunCommand`-metodia. Menetelmä on käyttökelpoinen silloin kun valmiista komponenteista ei löydy tarvittavia palveluita. Erillaisia rajapinnan käyttömuotoja on havainnollistettu kuvassa 6.7.

Kuvassa 6.8 esitetään `TNDACore`-komponentin käyttöä sovellusohjelmoinnissa.



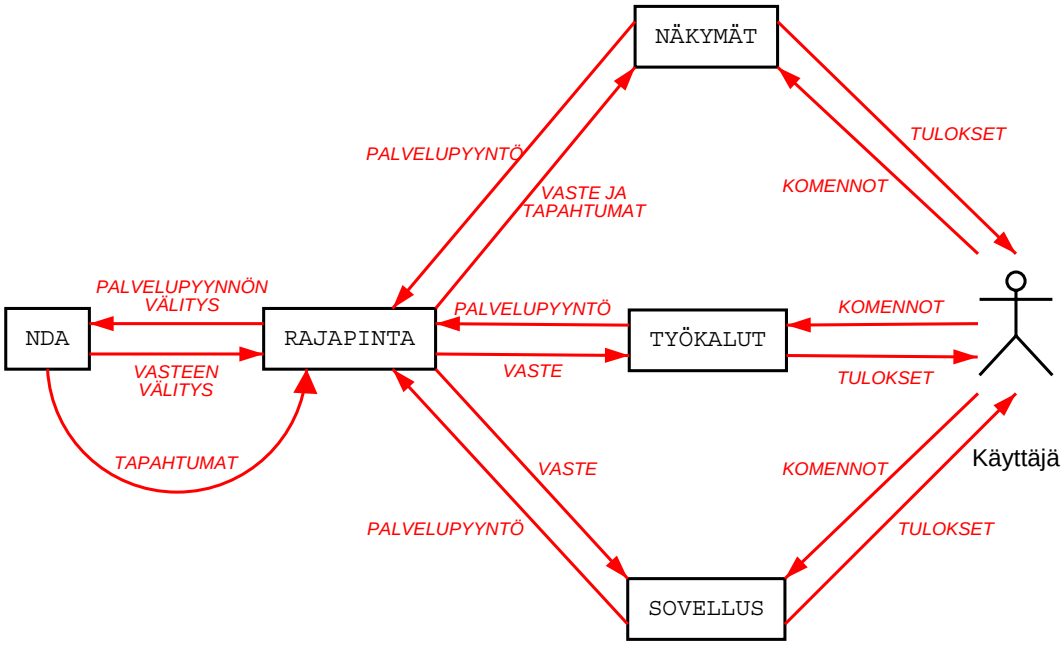
Kuva 6.6: NDA-rajapinnan tapahtumankäsittely.

6.1.2 Näkymäkomponentit

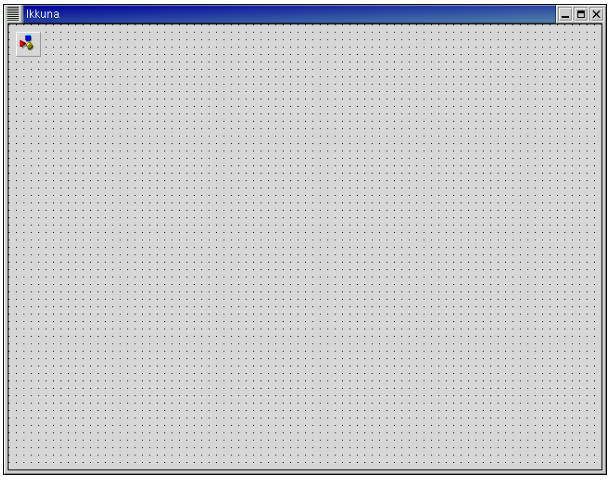
Sovellusohjelmien käyttöliittymät tarvitsevat usein samanlaisia palveluita. Sovelluskehityksen nopeuttamiseksi on muutamista yleisimmistä käyttöliittymistä laadittu valmis komponentti, jolloin kyseinen liittymä voidaan pudottaa valmiina lomakkeelle. Komponenttikirjaston mukana toimitetaan neljä näkymäkomponenttia, joiden tarjoamia palveluita tarvitaan lähes kaikenlaisissa sovellusohjelmissa.

Kuvassa 6.9 esiintyvä `TNDACommandView`-komponentti luo yksinkertaisen komentokäyttöliittymän NDA-ytimen ohjaamiseen. Suoritettavat komennot kirjoitetaan alalaidassa sijaitsevalle komentoriville ja suoritetaan painamalla rivinvaihtopainiketta. Aiemmin syötettyjä komentoja voidaan noutaa komentoriville nuolinäppäimien avulla. Toiminto muistuttaa UNIX-ympäristöjen komentoliittymää. Komentorivillä esiintyvää komentoa koskeva opaste saadaan näkyviin painamalla `[F1]`-painiketta. Sovelluksessa suoritetut komennot ja niiden vasteet kirjautuvat ylälaidassa näkyvään luettelokomponenttiin.

Kuvan 6.10 `TNDAGraphicView`-komponenttia käytetään NDA:lla raken-

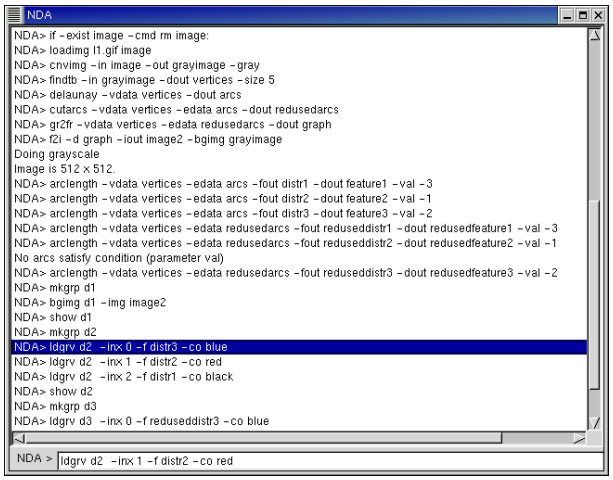


Kuva 6.7: Tiedon siirto NDA-ytimen ja sovelluksen välillä.

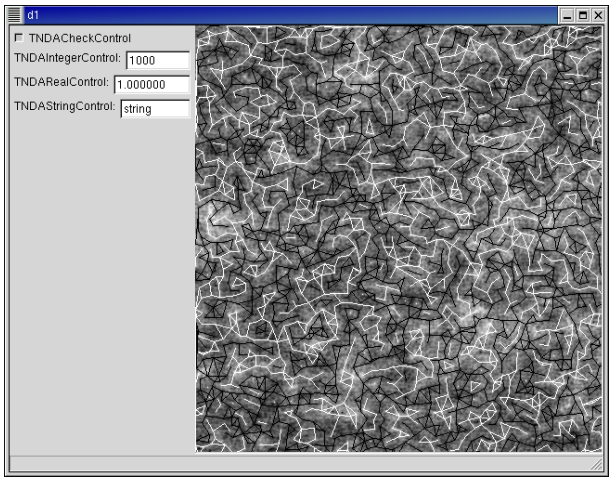


Kuva 6.8: TNDACore-komponentti sovellusohjelmassa.

netuista malleista saatujen graafien piirtämiseen näytölle. Graafit piirretään ensin bittikartalle, joka voidaan siirtää nopeasti näytölle ja näin vältytään samalla kuvan päivitystä häiritsevältä vilkkumiselta. Grafiikkakomponentin koon muuttaminen johtaa grafiikan automaattiseen päivittymiseen. Näin ko-



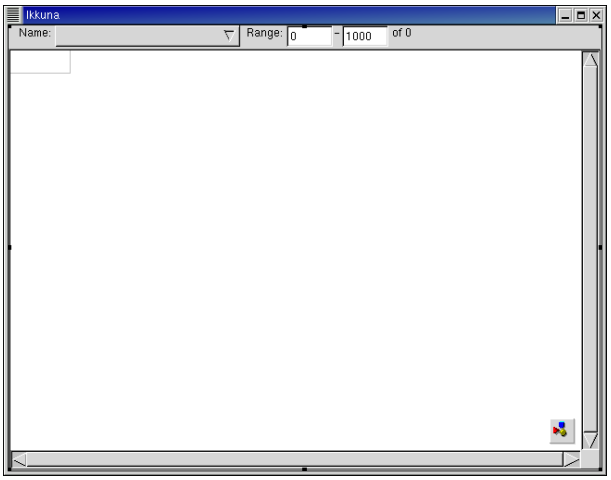
Kuva 6.9: TNDACommandView-komponentti sovellusohjelmassa.



Kuva 6.10: TNDAGraphicView-komponentti sovellusohjelmassa.

ko työtila saadaan käytettyä tehokkaasti. Komponentissa esiintyvän grafiikan käsittelyä varten voidaan komponentin vasempaan laitaan luoda työkalupalkki, johon sijoitettavilla työkaluilla käyttäjä voi muokata NDA:n muuttujia ilman komentoliittymässä tarvittavaa NDA-kielen tuntemusta. Työkalupalkin näkyvyyttä voidaan vaihtaa painamalla kuva-alueella osoittimen oikeaa painiketta.

Tietotaulukoiden esitystä varten laaditaan kuvan 6.11 mukainen taulukkokomponentti. Komponentin ylälaitaan sijoitetaan työkalurivi, jolla varsi-



Kuva 6.11: TNDAGridView-komponentti sovellusohjelmassa.

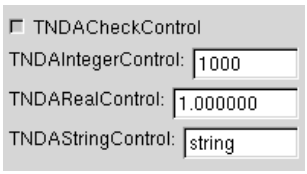
naista taulukkoaluetta ohjataan. Ensimmäisellä pudotusvalikolla valitaan esitettävä tietotaulukko. Pudotusvalikko sisältää automaattisesti listan kaikista esitettävissä olevista tietotaulukoista. Seuraavilla kahdella arvolla voidaan rajata näkyviin piirrettävää taulukon osaa. Suuret tietotaulukot ovat usein liian raskaita näytettäväksi kerrallaan ja silloin sovelluksen käytettävyys paranee sopivalla rajauksella huomattavasti.

Näiden kolmen peruskomponentin lisäksi voidaan luoda uusia räätälöityjä näkymiä erillisten sovellusten tarpeisiin. Laadittavat näkymät nimetään TNDA*View-nimisiksi, jossa tähden tilalle sijoitetaan näkymän toimintaa kuvaava englanninkielinen termi. Uudet näkymät toteutetaan perimällä tuleva komponentti TNDAView-komponentista. Komponentin tulee sisäisesti huolehtia ikkunan statukseen ja grafiikan päivitykseen liittyvistä tapahtumista. Varsinainen toiminnallisuus toteutetaan komponenttiin dynaamisesti luotavilla kontrolleilla ja tapahtumilla. Tapahtumien hallinta tulee suorittaa sisäisesti ilman käyttäjän kirjoittamaa lähdekoodia.

6.1.3 Työkalukomponentit

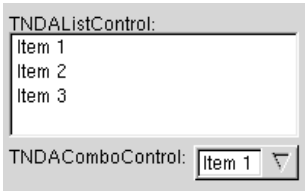
TNDAGraphicView-komponentti voi edellä kuvatulla tavalla sisältää työkaluja NDA-sovelluksen ohjaamiseen. Työkalut muodostetaan niin sanotuista työkalukomponenteista NDALIB-kielisen työkalupalkin kuvauksen perusteella.

`TNDACheckControl`:ia käytetään totuusarvoisen NDA-muuttujan asettamiseen. Muuttuja saa arvon 0 (ei-valittu) tai 1 (valittu).



Kuva 6.12: Muuttujatyökalut sovellusohjelmassa.

`TNDAIntegerControl`:ia ja `TNDARealControl`:ia käytetään numeroarvoisten NDA-muuttujien asetuksiin. Työkalujen arvoa voidaan vähentää ja lisätä yhdellä käyttäen nuolinäppäimiä. `TNDAStringControl` toimii vastaavalla periaatteella mutta sillä voidaan asettaa merkkijonoja. Työkalujen käyttöä sovellusohjelmassa on havainnollistettu kuvassa 6.12.



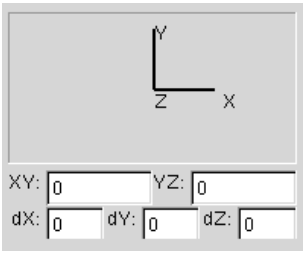
Kuva 6.13: Luettelotyökalut sovellusohjelmassa.

Luettelokomponentit (kuva 6.13) `TNDAListControl` ja `TNDAComboControl` esittävät käyttäjälle luettelon vaihtoehtoisista muuttujan arvoista. Muuttujan arvoksi voi siis tulla vain jokin ennaltamainituista vaihtoehdoista. Menetelmä soveltuu erityisen hyvin esimerkiksi käytetyn värin valitsemiseen.

Kuva 6.14: Painiketyökalu sovellusohjelmassa.

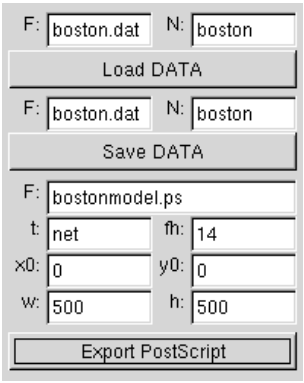
`TNDAButtonControl`:ia (kuva 6.14) käytetään lähinnä NDA-kielisten lauseiden suoritukseen. Yleensä suoritettavaa komentoa ei kirjoiteta suoraan työkaluun. Sen sijaan komentojono kirjoitetaan tiedostoon, johin työkalussa viitataan.

Grafiikassa käytettyä kuvakulmaa voidaan vaihtaa `TNDACameraControl`:lilla (kuva 6.15). Kuvakulmassa annetaan katsojan sijainti X,Y,Z -koordinaatistossa ja kiertymät XY- ja YZ-akselien



Kuva 6.15: Kameratyökalu sovellusohjelmassa.

suhteen. Kiertymiä voidaan vaihtaa myös havainnollisella pienoiskoordinaatistolla, jolla käytettyjä kulmia voidaan muuttaa vetämällä koordinaatistoa osoittimen vasen painike alaspainettuna.



Kuva 6.16: Tiedostotyökalut sovellusohjelmassa.

Tiedostotyökaluja (kuva 6.16) `TNDALoadControl`, `TNDASaveControl` ja `TNDAPostScriptControl` käytetään NDA-ytimen ja tiedostojärjestelmän välisen tiedonvaihdon ohjaukseen. `TNDALoadControl`:ia käytetään tiedostoihin tallennettujen aineistojen lataamiseen NDA:n nimiavaruuteen. Ladattavan tiedoston tulee noudattaa jotain NDA:n tukemista tiedostoformaateista. Vastaavasti `TNDASaveControl`:ia käytetään NDA:n nimiavaruudessa esiintyvän tietotaulun tallentamiseen. Taulu tallennetaan ASCII-formaatissa, joten sitä voidaan jatkokäsitellä useimmilla taulukkolaskentasovelluksilla. `TNDAPostScriptControl`:ia käytetään NDA:lla laadittujen grafiikoiden tallennukseen. Tallennusformaattina käytetään yleisesti tunnettua PostScript-formaattia, joten kuvia voidaan helposti liittää osaksi tekstinkäsittelysovelluksilla tuotettavia raportteja.

Näiden perustyökalujen lisäksi on mahdollista laatia räätälöityjä työka-

luja. Laadittavat näkymät nimetään TNDA*Control-nimisiksi, jossa tähden tilalle kirjoitetaan kontrollin toimintaa kuvaava englanninkielinen termi. Työkalut laaditaan perimällä ne TNDAControl-komponentista. Tapahtumien hallinta tulee suorittaa sisäisesti ilman käyttäjän kirjoittamaa lähdekoodia.

6.2 Kehitys

Ohjelmiston kehitys aloitettiin komponenttikirjastosta. Koska kehitysympäristöksi valittua Delphi 6/Kylix 1 -ohjelmistonkehitysympäristöä ei vielä tuolloin ollut saatavilla, toteutettiin kirjaston ensimmäinen versio Delphi 5:lla. Kirjaston ohjelmointi edistyi hyvin ja toimivia koeohjelmia voitiin tuottaa jo muutaman viikon sisällä aloituspäivästä. Kirjaston laajennusvaiheessa esiintyi muutamia käytetystä työkalusta aiheutuvia ongelmia, jotka johtivat alkuperäisen suunnitelman hienoiseen muutokseen. Kirjaston ensimmäinen toimiva versio saatiin aikaan ensimmäisen kuukauden aikana.

Kirjaston kehityksen toisen vaiheen muodosti laaditun ohjelmistotyön siirtäminen Delphi 6/Kylix 1 -ympäristöön ja ennenkaikkea niiden mukana toimitetulle CLX-kirjastolle, jolla saavutettaisiin suunnitelman mukainen siirrettävyys Windows ja Linux -käyttöjärjestelmien välille. Uuden ohjelmistoalustan käyttöönotto ei kuitenkaan onnistunut täysin kivuttomasti ja osa toteutetuista lähdekoodeista oli kirjoitettava uudelleen. Siirrettävyyden kannalta vaikeimmaksi ongelmaksi muodostuivat Delphi 6:n ja Kylix 1:n välillä havaitut kielen syntaktiset erot. Ongelman ratkaisemiseksi ei onnistuttu löytämään hyvää ratkaisua. Ongelmakohdat jouduttiin toteuttamaan kummallekin työkalulle erikseen siten, että lähdekoodin alussa ilmoitetaan käytetty työkalu, jonka perusteella päätetään kummalla syntaksilla toteutetut lähdekoodit käännetään.

6.3 Esittely

Tässä alaluvussa esitellään kehitystyön tuloksena syntynyttä komponenttikirjastoa ja sen käyttöön liittyviä seikkoja. Ennen tarkempaa tutustumista on lukijalla hyvä olla perustiedot Delphi/Kylix -ympäristöstä ja sillä suoritettavasta ohjelmistojen tuotannosta. Tarvittava tietotaito voidaan hankkia esimerkiksi työkalun mukana toimitettavasta kirjallisuudesta.

Kehitysympäristöön integroimista varten komponenttikirjastoon toteutettiin palvelun rekisteröivä metodi, jonka suoritus lisää kehitysympäristön työkalupalettiin uuden `NDA`-nimisen välilehden. Välilehdelle sijoitetaan käytettävissä olevista komponenteista `TNDACore`, `TNDACommandView`,

`TNDAGraphicView` ja `TNDAGridView`. Komponenttikirjaston käyttöönotto tapahtuu valikosta `Component > Install Packages...`, josta avautuvasta dialogista painetaan `Add`-painiketta. Pakettitiedoston etsimisen ja kirjastoa koskevien perustietojen syötön jälkeen kehitysympäristö liittää tarjotut palvelut itseensä ja varsinainen sovellusohjelmointi voi alkaa.

6.3.1 NDALIB-kieli

Komponenttikirjaston `TNDAGraphicView`-komponentin vasempaan laitaan on edellä esitetyllä tavalla mahdollista toteuttaa työkalupalkki. Työkalupalkki koostuu sivuista ja niihin sijoitettavista työkaluista. Erillisiä sivuja käytetään samaa käyttötarkoitusta varten olevien työkalujen ryhmittelyyn lähinnä silloin kuin kaikki työkalut eivät sovi näkymään samalla sivulla. Näkyvillä olevaa sivua voidaan vaihtaa painamalla työkalupalkin yllä osoittimen oikeaa painiketta ja valitsemalla näkyviin tuleva sivu esiinnousevasta valikosta. Työkalupalkin kuvaukseen määritetään tämän työn puitteissa uusi NDALIB-kieli. Kielen syntaksi on esitetty seuraavassa käyttäen yksinkertaista BNF (Bacus-Naur Format) kuvauskieltä, jonka jälkeen käydään tarkemmin läpi kielen semantiikkaa.

```
AKSIOMAATTISET MÄÄRITELMÄT:
<boolean> = "totuusarvo"
<string> = "merkkijono"
<EOL> = "rivin loppumerkki"
<EOF> = "tiedoston loppumerkki"

KIELIOPPI:
<root> ::= <head> <pages>
<head> ::= <barvisible> <visiblepage>
<barvisible> ::= 'BARVISIBLE=' <boolean> <EOL>
<visiblepage> ::= 'VISIBLEPAGE=' <string> <EOL>
<pages> ::= <page> <pages>
        |'!' <string> <EOL> <pages>|<EOF>
<page> ::= 'PAGE=' <string> <EOL> <controls>
<controls> ::= <control> <controls>
        |'!' <string> <EOL> <controls>|<EOF>
<control> ::= 'CONTROL=' <ctrltype> <EOL> <attributes>
<ctrltype> ::= 'TNDACheckControl'|'TNDAIntegerControl'
        |'TNDARealControl'|'TNDAStringControl'|'TNDAListControl'
        |'TNDAComboControl'|'TNDAButtonControl'
        |'TNDACameraControl'|'TNDALoadControl'
        |'TNDASaveControl'|'TNDAPostScriptControl'
```

```
<attributes> ::= <attribute> <attributes>
        |'!' <string> <EOL> <attributes>|<EOF>
<attribute>::= <caption>|<value>|<hint>|<name>|<command>
<caption> ::= 'CAPTION=' <string> <EOL>
<value> ::= 'VALUE=' <string> <EOL>
<hint> ::= 'HINT=' <string> <EOL>
<name> ::= 'NAME=' <string> <EOL>
<command> ::= 'COMMAND=' <string> <EOL>
<boolean> ::= 'True'|'False'
```

Tiedosto koostuu otsikko- ja kuvausosasta. Jokainen komento muodostuu yhdestä rivistä, joka päättyy rivinvaihtoon. Kommenttirivien alkuun sijoitetaan '!'-merkki, jonka jälkeen voidaan kirjoittaa vapaamuotoinen viesti.

Otsikossa <head> määritetään, onko työkalupalkki oletusarvoisesti näkyvissä <barvisible> ja mikä sivuista on oletusarvoisesti näkyvillä <visiblepage>. Näkyväksi määrätty sivu tulee kuvata saman tiedoston sisällä.

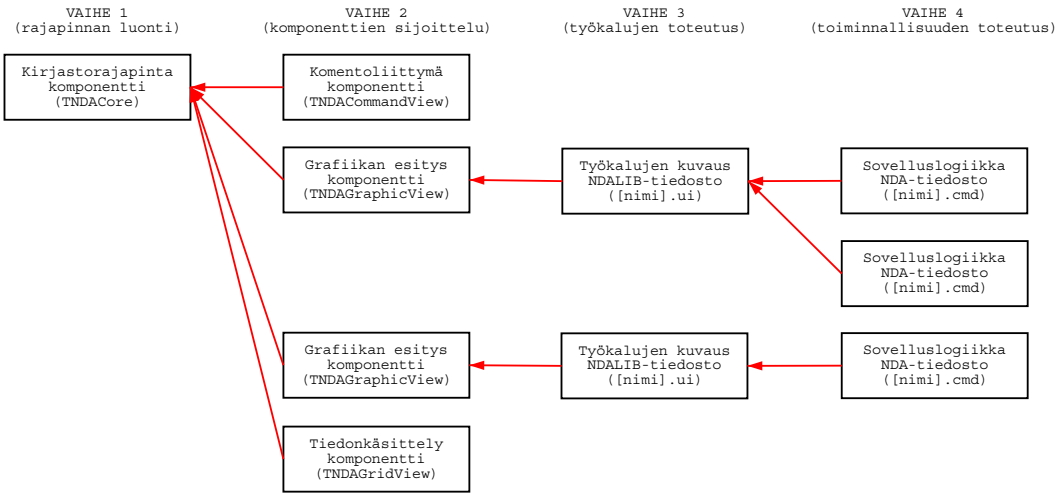
Tiedoston toisessa <pages> osassa määritellään kaikki työkalupalkkiin sijoitettavat sivut työkaluineen. Uuden sivun kuvaus aloitetaan <page>-avainsanalla. Samalla annetaan parametrina sivun nimeävä merkkijono. Sivulle asetettavan työkalun määrittely aloitetaan <control>-avainsanalla, jossa samalla nimetään merkkijonolla käytettävä työkalu. Uuden työkalun esittelyn jälkeen määritellään sen attribuutit. Kaikki attribuutit syötetään merkkijonoina. Otsikko <caption> kuvaa työkalun käyttötarkoitusta. Arvo <value> määrää työkalun oletusarvon. Arvon kuvaavan merkkijonon muodostuksessa on noudatettava työkalukohtaisia sääntöjä. Yleensä sääntö voidaan päätellä suoraan työkalun tyypistä. Numeerisiin työkaluihin käyvät numeroin kirjoitetut arvot. Luetteloiden yhteydessä useat merkkijonot erotetaan pilkuilla toisistaan. Opaste <hint> kuvaa sanallisesti työkalun toimintaa. Opasteeseen sijoitettu merkkijono näkyy ikkunoihin sijoitettavassa tilapalkissa. Työkalun nimi <name> sitoo sen toiminnan johonkin tiettyyn NDA-grafiikkaan. Yleensä nimen määritys tapahtuu automaattisesti ja siten <name>-avainsanan käyttöä tulee harkita. Komento <command> määrittää työkalukohtaisen tapahtumankäsittelijän. Jokaisella työkalutyypillä tapahtuman suorituksen laukaisee jokin tietty käyttäjän suorittama toimenpide. Useimmiten kyseinen toimenpide muuttaa työkalun arvoa. Samaa attribuuttia voidaan asettaa useita kertoja, jolloin viimeisin asetus jää voimaan. Uuden työkalun ja sivun aloitus tapahtuu yksinkertaisesti kirjoittamalla uusi <control> tai <page>-avainsana.

Esimerkki NDALIB-kielen käytöstä:

```
! Työkalupalkki näkyvissä?
BARVISIBLE=True
! Oletuksena näkyvä sivu
VISIBLEPAGE=Värien hallinta
! Esimerkkisivun aloitus
PAGE=Esimerkki
! Komponentin aloitus
CONTROL=TNDAIntegerControl
! Attribuuttien asetus
! Työkalun otsikko
CAPTION=Kokonaisluku
! Työkalun arvo
VALUE=100
! Sijoitetaan työkalun arvo (100) NDA-muuttujaan (arvo1)
! %d ~ C-kielinen esitysmuoto kokonaisluvuille
COMMAND=expr -fout arvo1 -expr %d;
! Uuden työkalun esittely
CONTROL=TNDARealControl
CAPTION=Liukuluku
VALUE=1.000
! %f ~ C-kielinen esitysmuoto liukuluvuille
COMMAND=expr -fout arvo2 -expr %f;
! Uuden työkalun esittely
CONTROL=TNDAStringControl
CAPTION=Merkkijono
VALUE=datat.dat
! %s ~ C-kielinen esitysmuoto merkkijonoille
COMMAND=expr -fout tiedosto -expr %s;
! Uuden sivun esittely
PAGE=Värien hallinta
CONTROL=TNDAListControl
CAPTION=Käytetty väri:
! Vaihtoehdot erotellaan ','-merkillä
VALUE=Punainen,Vihreä,Sininen
HINT=Käytettävissä olevat värit
COMMAND=expr -fout color -expr %s;
CONTROL=TNDAButton
CAPTION=Aseta väri
HINT=Valitun värin asetus
COMMAND=setcolor.cmd
```

6.3.2 Sovellusohjelmointi komponenttikirjastolla

Uuden NDA-sovelluksen toteutus aloitetaan luomalla siirrettävä CLX-projekti. Projektivelho luo valmiin ohjelmistopohjan sekä tyhjän aloitussivun, josta ohjelmistonkehitys voidaan aloittaa. Varsinainen sovellusohjelmointi tapahtuu neljässä työvaiheessa. Työvaiheita on havainnollistettu kuvassa 6.17.



Kuva 6.17: Sovellusohjelmointi komponenttikirjastolla.

Ensimmäisessä työvaiheessa luodaan liityntärajapinta sovelluksen ja NDA-kirjaston välille. Käytännössä tämä tehdään pudottamalla lomakkeelle komponenttipaletin `NDA`-sivulta `TNDACore`-komponentti. Jos komponentin käynnistyksessä esiintyy ongelmia, voi tämä johtua siitä, että NDA-kirjastoa ei ole asetettu oikealla tavalla käyttöjärjestelmän palveluihin. Muita toimenpiteitä kirjastoon liittymiseksi ei tarvita ja sovelluksen kehityksessä voidaan siirtyä varsinaiseen toteutukseen.

Toisessa työvaiheessa luodaan sovelluksen käyttöliittymän runko käyttäen `TNDACommandView`, `TNDAGraphicView` ja `TNDAGridView` komponentteja. Yksinkertainen käyttöliittymä voidaan muodostaa pelkällä `TNDACommandView`-komponentilla, sillä järjestelmä avaa automaattisesti uudet ikkunat sellaisille grafiikoille, joille ei ole osoitettu valmiiksi omaa paikkaa. Yleensä luodut grafiikat halutaan kuitenkin esittää samassa ikkunassa, jolloin jokaista NDA-grafiikkaa kohti pudotetaan oma `TNDAGraphicView`-komponentti. Joissain tapauksissa halutaan graafisten esitysten lisäksi nähdä NDA:n tietotauluja, jota varten on lomakkeelle pudotettava `TNDAGridView`-komponentti. Käytetyt komponentit sidotaan NDA:n tietoalkioihin asettamalla niiden `NDAName`-

attribuutti vastaamaan kyseisen tietoalkion nimeä.

Kolmannessa työvaiheessa luodaan käytetyille graafisille komponenteille tarkoituksenmukaiset työkalupalkit. Jokaista grafiikkakomponenttia kohden voi olla oma NDALIB-makrokielinen tiedosto, joka kuvaa palkkiin sijoitettavat sivut ja niiden työkalut. Makrotiedoston nimi on muotoa "[grafiikan nimi].ui"ja sen tulee sijaita sovelluksen kanssa samassa hakupolussa. Jos kyseistä tiedostoa ei löydy, yritetään seuraavaksi käyttää "default.ui-nimistä tiedostoa ja jos tätäkään ei ole saatavilla, jää työkalupalkki tyhjäksi. Varsinaiset työkalut toteutetaan komponenttikirjaston tarjoamilla työkalukomponenteilla, joiden toiminta tarkennetaan makrokielisesti käyttötapaukseen sopivaksi.

Edellisillä työvaiheilla on määritelty sovelluksen käyttörajapinta ottamatta kantaa varsinaisen sovelluslogiikan toteutukseen. Viimeisessä työvaiheessa laaditaan NDA-makrokieltä käyttäen rajapinnan toteuttava logiikkakerros, joka luo käyttöliittymän ohjaamana lähes kaiken sovelluksen toiminnallisuuden. Jokaista työkalupalkeissa sijaitsevaa komponenttia kohti voi olla olemassa oma makrokielinen tapahtumankäsittelijä. Käytännössä sovelluksen toteutus kannattaa rakentaa siten, että muut komponentit asettelevat parametreja NDA:n nimiavaruuteen ja vain painikkeena toimiva `TNDAButtonControl` aiheuttaa käyttöliittymästä erillisen makrotiedoston suorituksen.

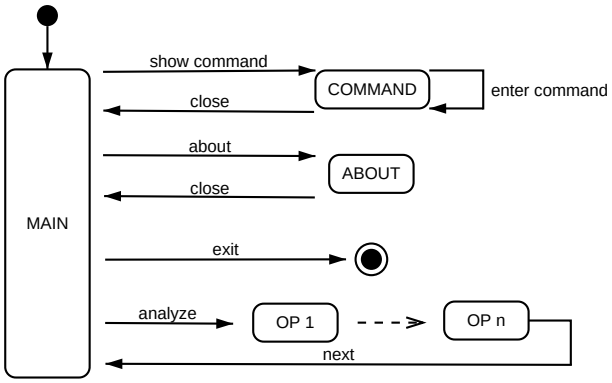
Luku 7

Käyttöpaneeli

Tässä luvussa kuvataan komponenttikirjaston päälle rakennettava käyttöpaneelisovellus.

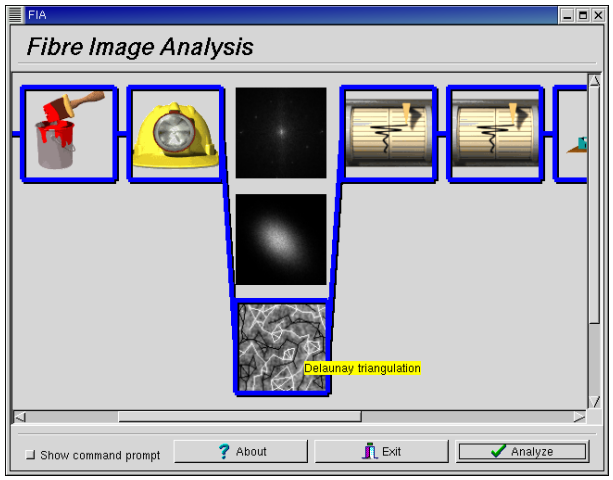
7.1 Kuvaus

Käyttöpaneelin tavoitteena on luoda sovelluskehitysympäristö analyysiketjupohjaisten sovellusten tuottamiseen, jotka toimivat ilman kolmannen osapuolen valmistamia ohjelmistonkehitystyökaluja. Uuden sovellusohjelman toteuttamiseen tarvitaan käyttöpaneelin lisäksi vain käytettävien makrokielten tuntemusta.



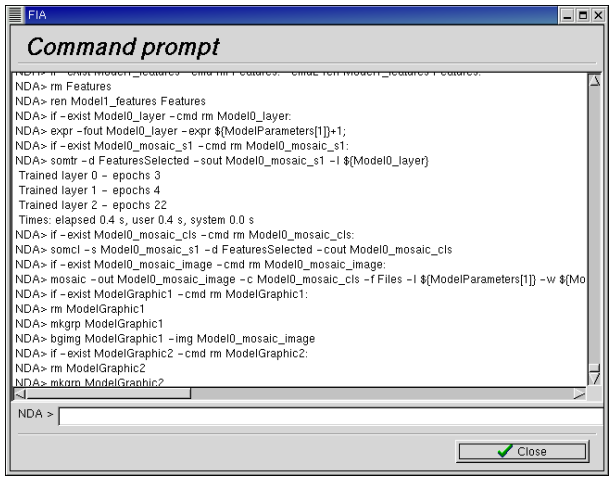
Kuva 7.1: Käyttöpaneelin toimintaperiaate.

Käyttöpaneelin toiminta koostuu kuvan 7.1 mukaisesti viidestä erillisestä toimintatilasta. Käynnistymisensä jälkeen sovellus on perustilassa (tila `MAIN`), josta siirrytään käyttäjän valintojen mukaisesti muihin tiloihin.



Kuva 7.2: Pääikkuna.

Käyttöpaneelin pääikkuna (kuva 7.2) toimii sovelluksen keskipisteenä, josta tarjolla olevia palveluita käytetään (tila `MAIN`). Pitkät analyysiketjut muodostuvat usein käyttäjän kannalta sokkeloisiksi. Analyysiketjussa liikkumista varten pääikkuna piirtää analyysiketjusta verkkomaisen kartan, johon käyttäjän edistyessä lisätään käyttäjän kulkema reitti.

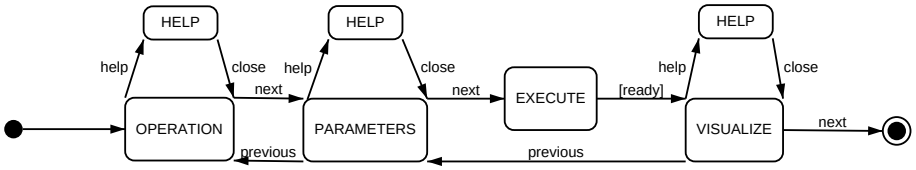


Kuva 7.3: Komentoikkuna.

Komentoikkuna (kuva 7.3, tila `COMMAND`) on tarkoitettu lähinnä sovellusohjelmien kehitysvaiheessa esiintyvien virheiden jäljittämiseen, eikä sitä voida käyttää analysointiprosessin ollessa käynnissä. Komentoikkunalla voidaan

tarvittaessa syöttää NDA-makrokielisiä komentoja ytimen suoritettavaksi. Kuvassa 7.3 näkyy esimerkki komentoikkunan käytöstä.

Käyttöpaneelin kehityksestä ja kehittäjästä saadaan tietoja painamalla pääikkunan `[About]`-painiketta. Käyttöpaneelin suorituksen päättävään tilaan siirrytään painamalla pääikkunan `[Exit]`-painiketta. Sovellusohjelman analyysiketjun suoritus aloitetaan `[Analyze]`-painikkeella. Sovelluksen muodostavan analyysiketjun päättyessä suoritus siirtyy takaisin pääikkunaan (`MAIN`-tilaan).



Kuva 7.4: Operaatioketju.

Tietojen analysoinnissa käytetään kuvan 7.4 mukaista perusrakennetta.

1. Ensimmäisessä vaiheessa valitaan käytettävissä olevien operaatioiden joukosta yksi suoritettavaksi. Vaiheen käyttöliittymä toteutetaan käyttäen operaatioikkunaa.

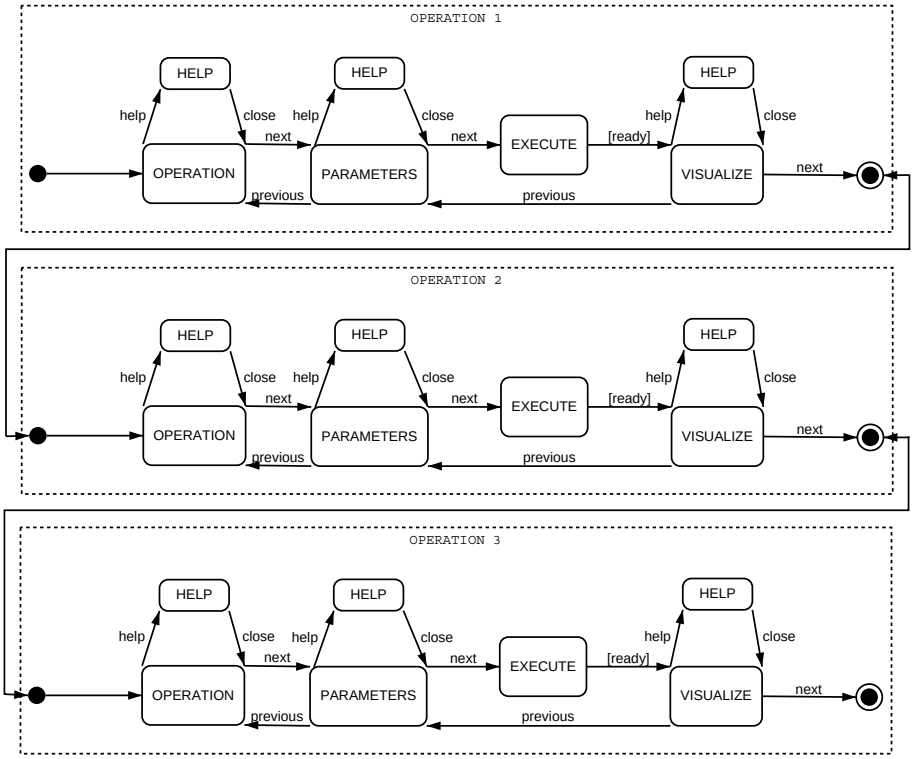
2. Toisessa vaiheessa syötetään valitun operaation tarvitsemat parametrit. Parametrit muodostavat käyttäjän syötteen, jolla operaatiota ohjataan. Käyttöliittymän muodostukseen käytetään parametri-ikkunaa.

3. Kolmannessa vaiheessa suoritetaan varsinainen operaation toimeenpano.

4. Neljännessä vaiheessa havainnollistetaan edellisten vaiheiden perusteella saavutettuja tuloksia. Havainnollisin tapa esitysten tekoon on laatia niistä malli, joka havainnollistetaan tietokonegrafiikkaa käyttäen. Käyttöliitymän tekemiseen käytetään grafiikkaikkunaa.

Toinen ja neljäs vaihe voivat tarpeen mukaan sisältää yhden sijaan useita ikkunoita. Varsinaista analyysiä suorittavan operaatiojonon sijaan jono voi koostua myös koko prosessia ohjaavista ikkunoista.

Analyysiketju muodostuu joukosta operaatioketjuja. Analyysiketjun suoritus alkaa ensimmäisestä operaatioketjusta ja päättyy viimeisen operaatioketjun valmistuttua, jolloin käyttäjä palaa takaisin ohjelmiston pääikkunaan. Analyysiketjun rakenne kuvataan ohjelmiston hakupolkuun kirjoitettavalla



Kuva 7.5: Analyysiketju.

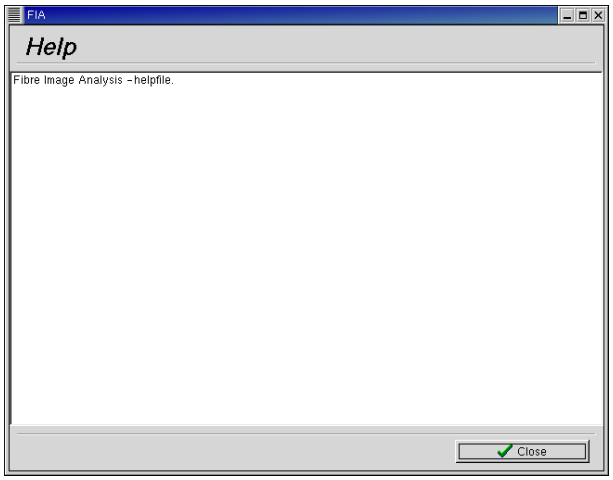
NDAIDE.code-tiedostolla käyttäen tarkoitusta varten määriteltyä makrokieltä. Kuvassa 7.5 annetaan esimerkki analyysiketjun kuvaamisesta tilakonetta käyttäen.

Lähes kaikissa ikkunoissa on opastepainike, joka on aktiivisena riippuen siitä onko sovellusohjelmiston toteuttaja kirjoittanut opasteen kyseistä ikkunaa varten. Painikkeesta avautuu erillinen opasteikkuna, joka tulee sulkea ennen varsinaisen prosessin jatkamista. Kuvassa 7.6 näkyy esimerkki opasteikkunan käytöstä.

7.2 Kehitys

Käyttöpaneelin kehityksessä noudatetaan kuvan 6.17 mukaisesti neljää työvaihetta. Ennen käyttöpaneelin kehitykseen perehtymistä tulee lukijan tutustua huolellisesti edellisessä luvussa oleviin komponenttikirjaston ohjelmointiin käsitteleviin alalukuihin.

Käyttöpaneelin suunnittelu aloitettiin tutustumalla erilaisiin käyttöliitty-



Kuva 7.6: Opasteikkuna.

mämalleihin. Toteutuksessa pyrittiin hakemaan sellaista käyttöliittymämallia, joka tukisi käyttäjäänsä siten, että sovelluksen toimintaan syventymätönkin henkilö voisi suorittaa käyttöliittymän ohjaamana monimutkaisia analyysiketjuja oikeassa toimintajärjestyksessä. Pyrkimyksenä oli myös estää liian monesta yhtäaikaisesta ikkunasta aiheutuva käytettävyyden heikentyminen.

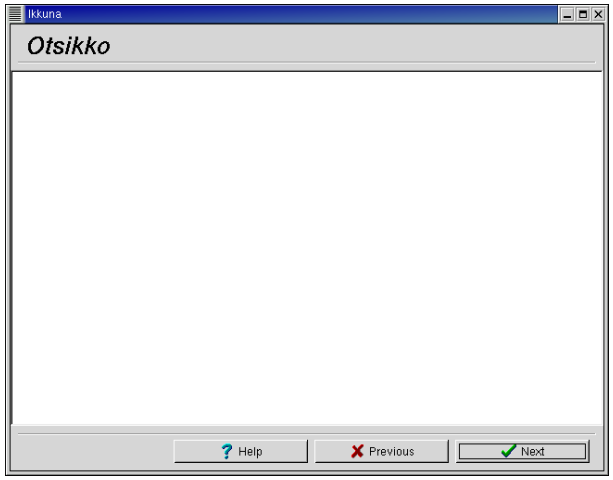
Koeohjelmista saatujen kokemusten perusteella päätettiin alkuperäistä vaatimusmäärittelyä laajentaa käyttöpaneelin osalta siten, että ohjelmiston yleiskäyttöisyys ja valmiiden sovellusohjelmien laajennettavuus paranisi niin, että käyttöliittymää voitaisiin soveltaa myös muiden ongelmien ratkaisuun.

Varsinainen työ käynnistettiin luomalla uusi CLX-projekti ja suorittamalla kirjaston käyttöönottoon liittyvät toimenpiteet. Valmiiden komponenttien avulla sovelluksen ohjelmointi etenikin rivakkaasti, eikä ohjelmiston toteutusvaiheessa esiintynyt ongelmia. Ohjelmiston uudelleenkäytettävyyden ja laajennettavuuden aikaansaavan makrokielen tulkitsemista varten kehitettiin määritellyn kielen parsiva ja suorittava algoritmi. Makrokielisten merkkijonojen parsintaan käytettiin apuna työkalun mukana toimitettuja työkaluja.

7.2.1 Ensimmäinen ja toinen työvaihe

Ensimmäisessä työvaiheessa luotiin rajapinta NDA-kirjaston ja käyttöpaneelisovelluksen välille. Koska kaikki tarvittava työ oli jo toteutettu `TNDACore` ja `TNDAGraphicView`-komponenttien muodossa, ei ensimmäisen vaiheen suoritukseen kulunut juurikaan aikaa.

Toisessa työvaiheessa luotiin käyttöpaneelisovelluksessa esiintyvät ikku-



Kuva 7.7: Käyttöpaneelin ikkunoiden rakenne.

nat. Ohjelmiston käyttöliittymän omaksumisen nopeuttamiseksi kaikki ikkunat päätettiin tuottaa samanlaisella rakenteella. Ikkunoiden yläosaan luotiin `TLabel`-komponentilla suurin kirjaimin sovelluksen tilaa kuvaava otsikkorivi. Ikkunoiden alalaitaan pudotettiin joukko `TBitBtn`-komponentteja painikkeiksi, joilla sovelluksen etenemistä ohjataan. Edellisten väliin jäävä tila muodostaa ikkunan varsinaisen työtilan, johon ikkunakohtaiset palvelut toteutettiin. Tarkempia tietoja sovelluksen rakenteesta on saatavilla liitteessä B, jossa esitetään UML-kuvauskielellä ohjelmiston luokkarakenne.

7.2.2 Kolmas ja neljäs työvaihe

Luotava käyttöpaneeliohjelmisto ei sovellu suoraan minkään ongelman analysointiin vaan se muodostaa sovelluskehitysympäristön, jolla varsinaisia ongelmia ratkovia sovelluksia voidaan tuottaa. Komponenttikirjaston sovellusohjelmoinnissa suoritettavat 3. ja 4. työvaihe liittyvät kiinteästi lopullisen sovelluksen toteutukseen. Näin ollen kyseiset työvaiheet siirtyvät luonnollisesti osaksi käyttöpaneelin sovellusohjelmointia.

7.3 Esittely

Tässä alaluvussa esitellään käyttöpaneelisovellus opettamalla lukijalle käyttöpaneelin sovellusohjelmointia. Ohjelmoinnista saatavaa teoreettista näkökulmaa täydennetään seuraavassa luvussa esiteltävän sovellusohjelman avulla.

7.3.1 NDAIDE-kieli

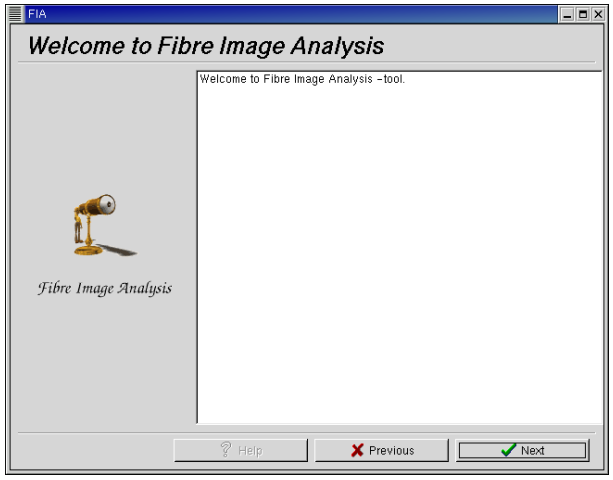
Käyttöpaneelilla rakennettavat sovellusohjelmat koostuvat yksinkertaisista elementeistä, joita sopivasti ketjuttamalla tarjottavat palvelut muodostetaan. Analyysiketjun tavoitteena on yleensä rakentaa näytteistetystä informaatiosta selittävä malli, jonka avulla alkuperäistä informaatiota voidaan ymmärtää paremmin. Analyysiketjun kuvaukseen tarvitaan uusi NDAIDE-kieli. Kielen syntaksi on esitetty jäljempänä käyttäen BNF (Bacus-Naur Format) -kuvauskieltä, jonka jälkeen lukija perehdytetään kielen semantiikkaan.

```
AKSIOMAATTISET MÄÄRITELMÄT:
<boolean> = "totuusarvo"
<string> = "merkkijono"
<EOL> = "rivin loppumerkki"
<EOF> = "tiedoston loppumerkki"

KIELIOPPI:
<root> ::= <lines>
<lines> ::= <line> <lines>|<EOF>
<line> ::= '#' <string>|<briefing>|<filecase>
        |<operation>|<parameter>|<data>|<datacase>
        |<graphic>|<question>|<execute>|<run>
<briefing> ::= 'Briefing,'<title>','<header>','<file>','<nda><EOL>
<filecase> ::= 'FileCase,'<title>','<header>','<file>','<nda><EOL>
<operation> ::= 'Operation,'<title>','<header>','<file>','<nda><EOL>
<parameter> ::= 'Parameter,'<title>','<header>','<file>','<nda><EOL>
<data> ::= 'Data,'<title>','<header>','<file>','<nda><EOL>
<datacase> ::= 'DataCase,'<title>','<header>','<file>','<nda><EOL>
<graphic> ::= 'Graphic,'<title>','<header>','<file>','<nda><EOL>
<question> ::= 'Question,'<title>','<header>','<file>','<nda><EOL>
<title> := '"' <string> '"'
<header> := '"' <string> '"'
<file> := <string>
<nda> := <string>
<execute> ::= 'Execute,' <string>
<run> ::= 'Run,"' <string> '"'
```

Tiedosto kuvaa lineaarisen analyysiketjun, jossa voi olla sekä käyttöliittymää ohjaavia että komentojonoja suorittavia komentoja. Jokainen komento sijoitetaan omalle rivilleen. Komentorivi koostuu varsinaisesta komennosta ja sitä seuraavista parametreista, jotka erotellaan pilkuilla. Kommenttirivit aloitetaan '#'-merkillä, jonka jälkeen voidaan kirjoittaa vapaamuotoinen viesti.

Käyttöliittymän komennot käyttävät kaikki samaa syntaksia. Varsinaisen komennon jälkeen annetaan käytettävän ikkunan otsikko <`title`>. Tämän jälkeen syötetään varsinainen otsikko <`header`>, joka näkyy suurilla kirjaimilla ikkunan sisällä. Molemmat edellä mainituista parametreista rajataan lainausmerkein. Näiden jälkeen määrätään komentoon kuuluva tiedostoviite <`file`>, joka määrää kaikki komennon suorituksessa tarvittavat tiedostot. Erillisissä tiedostoissa on kuvattu komennon tuottaman ikkunan rakennetta (.ui-päätteinen tiedosto) ja ikkunaan kuuluva opaste (.help-päätteinen tiedosto). Lopuksi nimetään komennon NDA-viite <`nda`>, joka sitoo komennon toiminnan kyseiseen NDA-muuttujaan. Tiedostoviitteen ja NDA-viitteen käyttö riippuu täysin käytettävästä komennosta. Suorittavat komennot sisältävät vain yhden merkkijonomuotoisen parametrin. `Run`-komennon yhteydessä parametri kirjoitetaan lainausmerkkeihin.

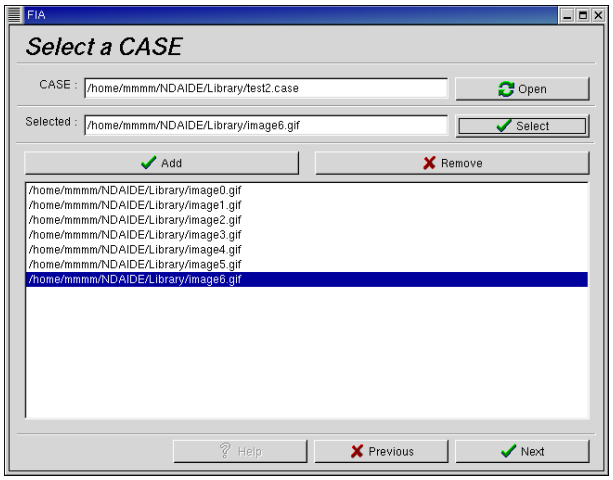


Kuva 7.8: Johdatusikkuna.

Esimerkki: `Briefing,"Title","Header",Briefing,Briefing`

Johdatusikkunalla <`briefing`> voidaan käyttäjälle välittää tietoja niin sovellusohjelmiston laatijasta, kuin toteutetusta analyysiprosessiketjustakin. Kuvan 7.8 esimerkki näyttää käyttäjälle tiedostoon Briefing.ui kirjoitetun tekstin. Tiedoston ensimmäinen rivi voi olla myös viite kuvatiedostoon, jolloin kyseinen kuva esitetään ikkunan vasemmassa laidassa.

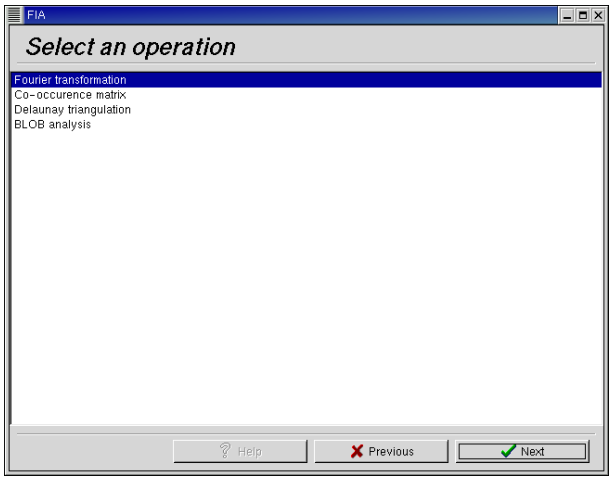
Koeasetelmaikkunalla <`filecase`> valitaan ne näytetiedostot, jotka halutaan sisällyttää suoritettavaan analyysiin. Näytetiedostoista yksi valitaan tarkastelun kohteeksi ja muita käytetään referenssiaineistona piirteiden vertailussa. Kuvan 7.9 esimerkki tallentaa käyttäjän valitsemien tiedostojen nimet polkuineen muuttujaan Files. Tarkastelun kohteeksi valitun tiedoston



Kuva 7.9: Tiedostoikkuna.

Esimerkki: `FileCase,"Title","Header",Files,Files`

nimi tallennetaan polkuineen muuttujaan FilesSelected. Kuvaustiedosto Files.ui tallentaa aktiivisen koeasetelmatiedoston nimen polkuineen. Koeasetelmatiedostot (*.case) sisältävät tekstiriveinä viittaukset niissä mukana oleviin näytetiedostoihin.

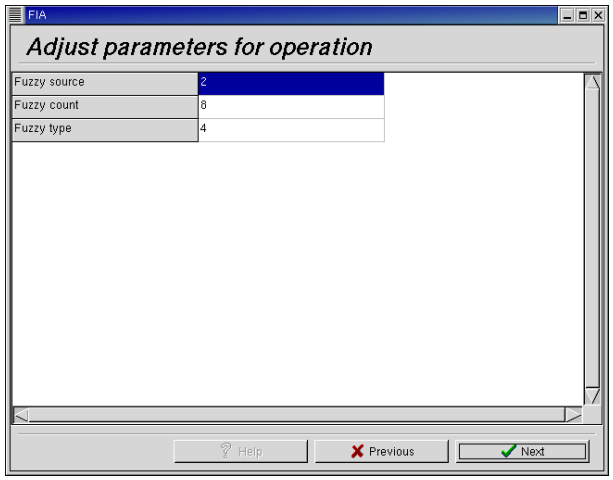


Kuva 7.10: Operaatioikkuna.

Esimerkki: `Operation,"Title","Header",Model,Model`

Operaatioikkuna <`operation`> aloittaa uuden operaatioketjun esittelemällä käyttäjälle sovellusohjelman tarjoamia suoritusvaihtoehtoja. Tarjolla

olevat vaihtoehdot esitellään kuvaustiedoston tekstiriveinä. Käyttäjän tekemä valinta vaikuttaa seuraavien ikkunoiden ja koko operaatioketjun suoritukseen. Kuvan 7.10 esimerkki antaa käyttäjän valita suoritettavan operaation tiedostossa Model.ui määritellyistä vaihtoehdoista.



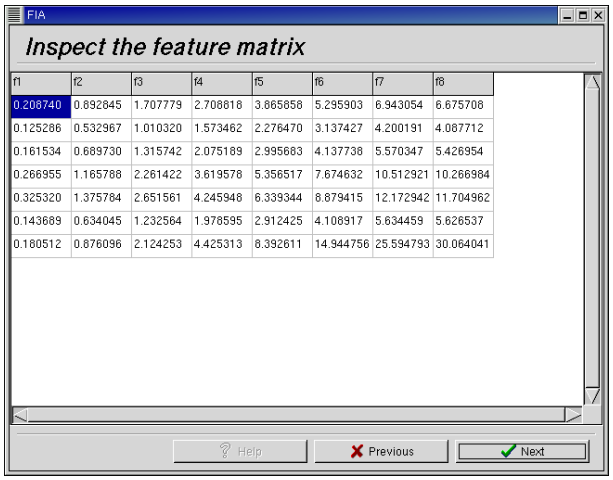
Kuva 7.11: Parametri-ikkuna.

Esimerkki: `Parameter,"Title","Header",Model,Model`

Parametri-ikkuna <`parameter`> esiintyy yleensä operaatioikkunan jäljessä keräämässä käyttäjältä valitun operaation suoritukseen vaikuttavia syötteitä. Kaikille parametreille on yleensä annettu oletusarvo, jolloin operaatio voidaan toteuttaa sellaisenaan. Kaikki parametrit esitetään merkkijonoina ja siten on käyttäjän vastuulla syöttää parametrit sovellusohjelmiston pyytämässä muodossa. Ikkunan sulkeutuessa parametrit tallennetaan NDA-ytimen muuttujaan <`nda`>. Kuvan 7.11 esimerkki antaa käyttäjän muuttaa valitun Model-nimisen operaation parametreja.

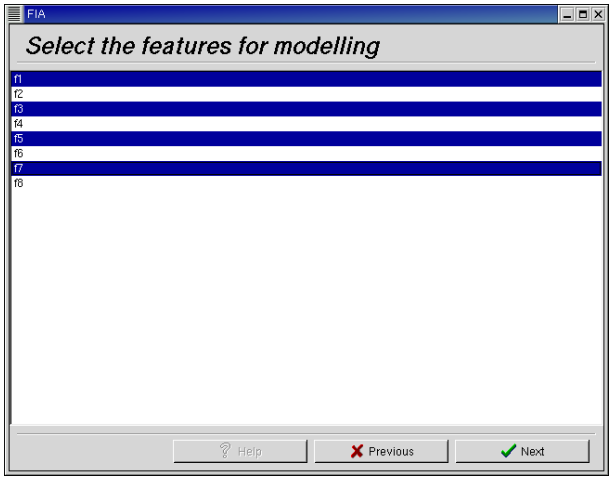
Taulukkoikkunalla <`data`> voidaan tuoda käyttäjän nähtäville NDA:n nimiavaruudessa sijaitseva tietotaulukko <`nda`>. Käyttäjällä on mahdollisuus selata taulukon arvoja mutta hänellä ei ole oikeutta suorittaa muutoksia taulukon sisältöön. Kuvan 7.12 esimerkki näyttää käyttäjälle Features-nimisen tietotaulukon sisällön.

Piirreikkunaa <`datacase`> käytetään analyysiprosessissa valitsemaan ne piirretaulussa esiintyvät piirteet (tietotaulun sarakkeet), jotka halutaan ottaa mukaan jatkotutkimuksiin. Piirteiden valinta tapahtuu merkitsemällä luettelosta mukaan otettavat piirteet. Useiden piirteiden valinnassa on monivalintaluetteloiden tapaan käytettävä `Shift` tai `Ctrl` -painikkeita. Ikkunan sulkeutuessa valittujen taulukon sarakkeiden nimet tallennetaan



Kuva 7.12: Taulukkoikkuna.

Esimerkki: `Data,"Title","Header",Model,Features`

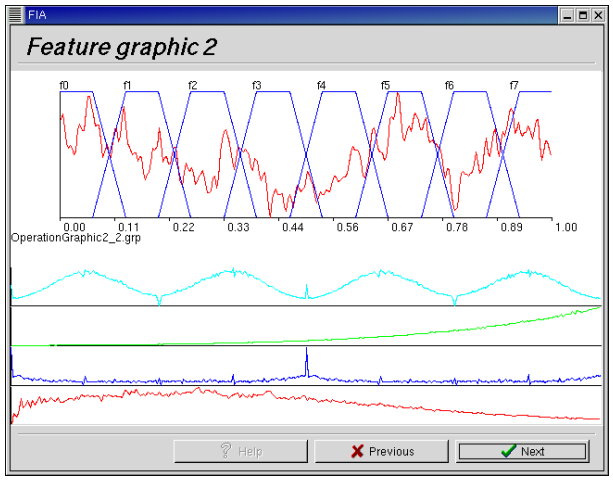


Kuva 7.13: Piirreikkuna.

Esimerkki: `DataCase,"Title","Header",Stage1,Features`

NDA-ytimen muuttujaan <`nda`>. Kuvan 7.13 esimerkissä käyttäjä valitsee Features-nimisestä tietotaulukosta piirteiden nimiä, jotka tallennetaan FeaturesSelected-nimiseen muuttujaan.

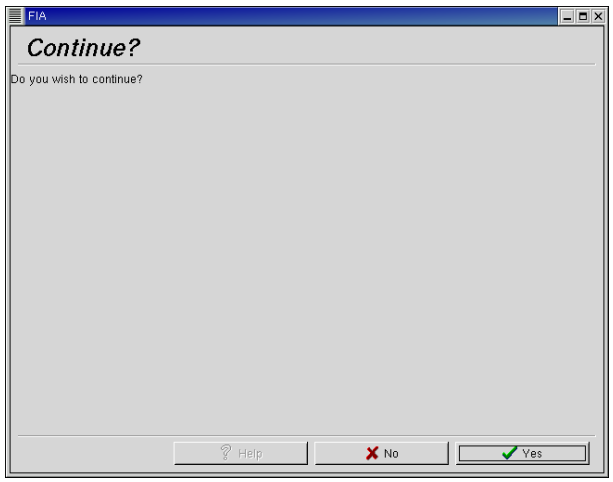
Grafiikkaikkuna <`graphic`> on tarkoitettu analyysien tuloksista laadittavien graafien esitykseen. Jokaiselle grafiikkaikkunalle ilmoitetaan sen NDA-grafiikan nimi <`nda`>, joka ikkunaan halutaan piirtää. Ikkunassa voi olla



Kuva 7.14: Graafiikkaikkuna.

Esimerkki: `Graphic,"Title","Header",Graph,Histogram`

kerrallaan näkyvillä vain yksi NDA-grafiikka ja useiden grafiikka-alkioiden piirtäminen tulee suorittaa sijoittamalla päägrafiikkaan useita aligrafiikka-alkioita. Kuvan 7.14 esimerkki piirtää ikkunaan Histogram-nimisen grafiikan. Kuvaustiedostossa Graph.ui kuvataan NDALIB-kielellä ikkunan vasempaan laitaan näkyviin saatava työkalupalkki.



Kuva 7.15: Kysymysikkuna.

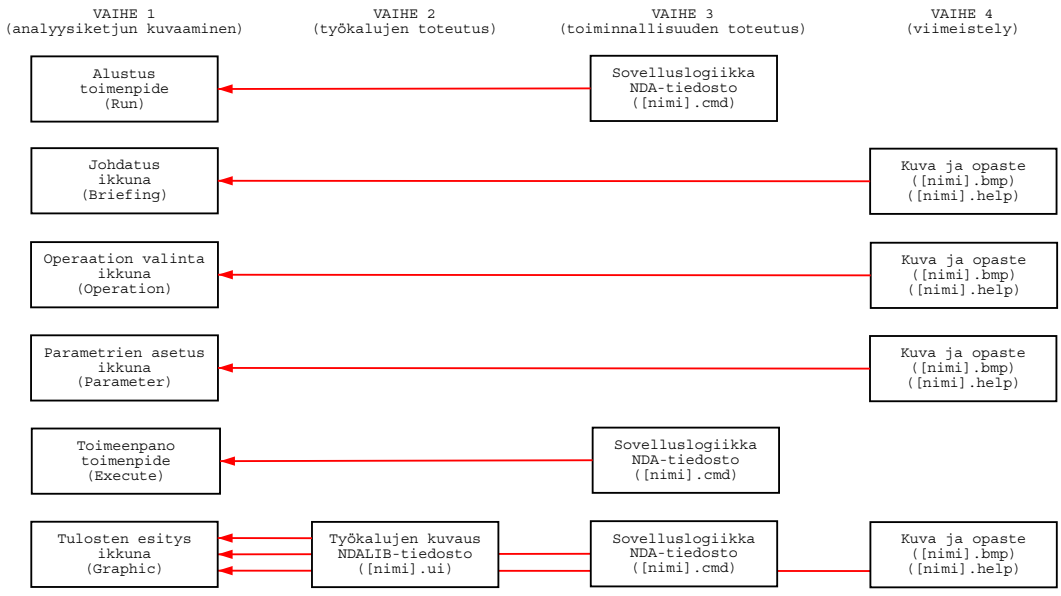
Esimerkki: `Question,"Title","Header",Question,Answer`

Kysymysikkunalla <question> voidaan käyttäjälle esittää kysymys, johon tämän tulee antaa joko Kyllä tai Ei -vastaus. Kuvan 7.15 esimerkki esittää käyttäjälle tiedostossa Question.ui olevan kysymyksen ja tallentaa Answer-nimiseen NDA-muuttujaan joko "Yes"tai "No"riippuen käyttäjän palautteesta.

Suorituskomento `Execute` ajaa parametrilla nimetyn operaatiojonon NDA-kielisen sovelluslogiikan. Jonosta suoritetaan aiemmin operaatioikkunalla valuttu vaihtoehto. Suorituskomento `Run` ajaa parametrina toimitetun NDA-komennon. Suoritettava komentoparametri esitetään lainausmerkeillä rajattuna.

7.3.2 Sovellusohjelmointi käyttöpaneelilla

Uuden käyttöpaneelisovelluksen kehitys aloitetaan kopioimalla käyttöpaneelin suoritustiedosto (NDAIDE.exe/NDAIDE) tyhjään hakemistoon. Ohjelmiston käynnistystä varten voidaan hakemistoon luoda sovelluksen toimintaa paremmin kuvaava symbolinen linkki, joka käynnistää ohjelmiston. Tämän jälkeen voidaan ryhtyä varsinaisen sovelluksen kehitykseen. Uuden sovelluksen toteutus kannattaa jakaa neljään työvaiheeseen. Työvaiheiden suoritusta on havainnollistettu kuvassa 7.16 sivulla 89.



Kuva 7.16: Sovellusohjelmointi käyttöpaneelilla.

Ensimmäisessä työvaiheessa kuvataan analyysiketjun rakenne

NDAIDE.code-tiedostoon. Sovelluksen suunnittelussa voidaan käyttää apuna tilakoneita tämän työn suunnitteluvaiheessa esitetyn esimerkin mukaisesti. Esimerkin mukaisesti laaditut tilakoneet voidaan muuntaa suoraan NDAIDE-makrokielelle, jolloin suunnitelma samalla dokumentoi osan sovelluksen toteutuksesta.

Esimerkki tiedostosta NDAIDE.code

`Operation,"Ikkunan otsikko","Otsikko",operaatio,Operaatio`
`Parameter,"Ikkunan otsikko","Otsikko",operaatio,Parametrit`
`Execute,operaatio`

NDAIDE.code tiedosto kuvaa sovelluksen rungon, jota täydennetään ikkunoiden kuvaustiedostoilla. Operaatiojonon tiedostojen nimeämisessä käytetään seuraavaa periaatetta:

Käytettävissä olevat operaatiot => operaatio.ui
Ensimmäisen operaation parametrit => operaatio0.ui Toisen operaation parametrit => operaatio1.ui ...
Ensimmäisen operaation toimeenpano => operaatio0.cmd Toisen operaation toimeenpano => operaatio1.cmd ...

Operaatioikkunan kuvaustiedoston jokainen rivi määrittelee uuden operaatiovaihtoehdon. Ensimmäisen rivin valinta johtaa ensimmäisen operaatiojonon suoritukseen, toinen rivi toisen ja niin edelleen.

Esimerkki tiedostosta operaatio.ui

`Operaatio 1`
`Operaatio 2`
`Operaatio 3`

Kuhunkin operaatioon liittyvät parametrit kuvataan valittuun operaatioon liittyvänä kuvaustiedostona. Kukin tiedoston rivi määrää uuden parametrin nimen ja oletusarvon.

Esimerkki tiedostosta operaatio0.ui

`''Parametri1'',''Arvo1''`
`''Parametri2'',''Arvo2''`
`''Parametri3'',''Arvo3''`

Toisessa työvaiheessa kirjoitetaan ikkunoihin liittyvät kuvaustiedostot (*.ui). Grafiikkaikkunoihin tulevat työkalupalkit kuvataan NDALIB-kielellä, jonka käyttö on opetettu edellisessä luvussa. Muiden ikkunoiden kuvaukseen

käytetään ikkunakohtaisia tiedostoformaatteja.

Kolmannessa työvaiheessa ohjelmoidaan käyttöliittymän toteuttava sovelluslogiikka käyttäen NDA-makrokieltä. Makrokoodissa noudatetaan NDAIDE.code tiedostossa määriteltyä nimeämiskäytäntöä NDA-muuttujien hallintaan.

Esimerkki tiedostosta operaatio0.cmd

```
echo Ensimmäisen parametrin arvo on ${Parametrit[0]}
echo Toisen parametrin arvo on ${Parametrit[1]}
echo Kolmannen parametrin arvo on ${Parametrit[2]}
```

Viimeisessä työvaiheessa suoritetaan sovelluksen viimeistely, joka ei enää lisää sovelluksen toiminnallisuutta mutta parantaa sen käytettävyyttä. Käyttöpaneelin pääikkunassa esitettävää analyysiketjun havainnollistavaa verkkoa varten on ohjelmiston hakupolussa oltava jokaista ikkunaa esittävä kuvatiedosto. Kuvat tallennetaan nimellä "[ikkunan nimi].bmp"käyttäen BMP (Windows or OS/2 Bitmap) -formaattia. Kaikkien käytettävien kuvien tulee olla saman kokoisia. Kuvien vakiokoko on 100x100 pikseliä mutta tarvittaessa sitä voidaan vaihtaa luomalla NDAIDE.net-tiedosto, jossa kolmella rivillä ilmaistaan käytetty leveys, korkeus ja kuvien välillä oleva tyhjä tila kuvapisteinä.

Esimerkki tiedostosta NDAIDE.net

```
100
100
10
```

Sovelluksen käytön helpottamiseksi voidaan tehdä opastejärjestelmä, jossa jokaiseen ikkunaan liittyy oma tekstimuotoinen käyttöohje. Opasteet toteutetaan kirjoittamalla ikkunan opasteen teksti "[ikkunan nimi].help"nimiseen tiedostoon.

Esimerkki tiedostosta operaatio0.help

```
Ohje ensimmäisen operaation suorittamiseksi:
...
```

Luku 8

Esimerkkisovellus

Tässä luvussa esitellään työssä aiemmin toteutetun käyttöpaneelin ohjelmointia käytännön sovellustuotannossa. Esimerkkisovelluksella pyritään käyttäjien opetuksen ohella määrittämään, kuinka hyvin vaatimusmäärittelyssä esitellyt tavoitteet on saavutettu.

8.1 Kuvaus

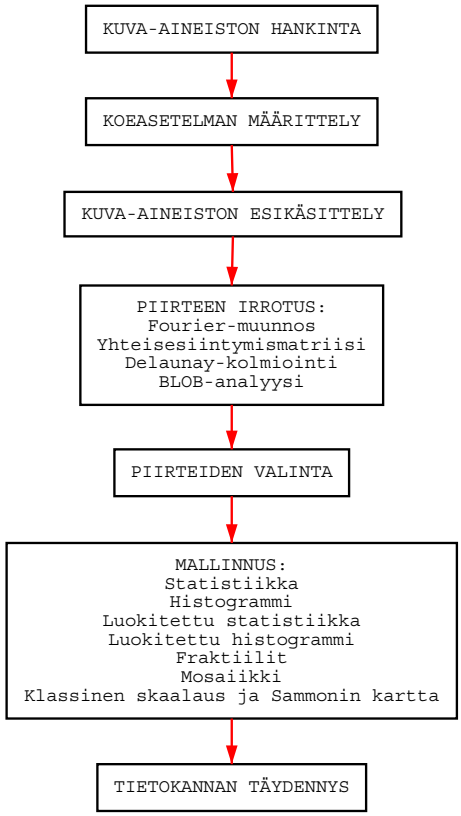
Esimerkkisovelluksen kehitys aloitetaan tutustumalla aiemmin laadittuihin analyysiohjelmistoihin ja -makrotiedostoihin. Kuituanalyysien rakennusmallina käytetään Jukka Perkiön vuonna 1999 toteuttamaa ohjelmistoa [63], jota on jatkokehitty uusilla analyyseilla. Luotava esimerkkisovellus jakautuu kuvan 8.1 mukaisesti seitsemään työvaiheeseen.

Kuituanalyysin suoritus alkaa kuva-aineiston hankinnalla. Koska nykyiset kuvankäsittelyohjelmistot sisältävät riittävät ominaisuudet sekä kuvan kaappaukseen että kuvan esikäsittelyyn, päätettiin kuvien näytteistysvaiheen suoritukseen käyttää valmiita työkaluja.

Varsinainen analyysiprosessi käynnistyy koeasetelman määrittelyllä. Koeasetelmassa valitaan analyysissä käytettävät näytteet. Tutkimusta varten yksi näytteistä valitaan tutkinnan kohteeksi ja muita, mielellään jo ennalta tutkittuja ja siten tunnettuja, näytteitä käytetään tulosten vertailuun.

Ennen varsinaista näytteen analysointia saattaa esiintyä tarvetta tutkittavan näytteen esikäsittelyyn. Esikäsittelyllä pyritään korostamaan näytteen oleellisia ominaisuuksia epäoleellisten kustannuksella ja näin tehostamaan käytettyjen algoritmien toimintaa.

Piirteiden irrotuksella tiivistetään näytteen informaatio muutamaksi näytettä kuvaavaksi tunnusluvuksi. Näytteinä toimivat kuva-aineistot muodostavat liian monimutkaisen esityksen, jotta niiden pohjalta voitaisiin suoraan



Kuva 8.1: Kuituanalyysiohjelmiston toimintaperiaate.

tehdä pidemmälle meneviä johtopäätöksiä. Samalla tavalla esikäsitellyistä näytteistä laskettuja piirteitä voidaan sensijaan vertailla helposti keskenään ja niiden pohjalta voidaan jatkossa luoda tilannetta kuvaavia malleja. Toteutettaviin piirteiden irrotusmenetelmiin tutustutaan tarkemmin sovellusta esittelevässä luvussa.

Piirteiden irrotusvaiheen tuloksena saadaan käytetystä menetelmästä riippuva määrä näytteitä kuvaavia tunnuslukuja. Aina ei olla yhtä kiinnostuneita kaikista piirteitä, vaan jostakin tietystä piirteiden osajoukosta, jolla oletetaan kuvaavan kokonaisuutta mielenkiintoisella tavalla. Piirteiden valintavaiheessa käyttäjälle annetaan mahdollisuus tutustua laskettuihin piirteisiin, jonka jälkeen tämä voi valita niistä mielenkiintoisimmat.

Numeeriset piirteet eivät välttämättä muodosta riittävän selvää kuvaa eri näytteiden välisistä eroista. Tilannetta voidaan helpottaa luomalla piirteiden pohjalta malleja, jotka voidaan havainnollistaa tietokonegrafiikan avulla. Esimerkkisovelluksen ensimmäisen version mukana toimitettavat mallit rakenne-

taan käyttäen Kohosen itseorganisoituvaa piirrekarttaa *(engl. Kohonen's Self Organizing Map)* [64]. Laskennallisen suorituskyvyn parantamiseksi itseorganisoituvien piirrekarttojen toteutukseen käytetään algoritmin puumaista variaatiota *(engl. Tree Structured Self Organizing Map)* [65]. Mallin rakennukseen käytettäviin menetelmiin tutustutaan tarkemmin sovelluksen toimintaa esittelevässä luvussa.

Näytteistä parhaimmat ja mielenkiintoisimmat liitetään osaksi näytetietokantaa, jolloin niiden tuloksia voidaan käyttää vertailuaineistona uusia näytteitä tutkittaessa. Tietokanta päätettiin toteuttaa yksinkertaisesti luomalla tyhjä hakemisto, johon kyseiseen tietokantaan liitettävät näytetiedostot kopioidaan. Koska käyttöjärjestelmä tarjoaa riittävät työkalut tiedostojen hallintaan, ei sovellukseen ryhdytty lisäämään tietokantojen ylläpitoon tarvittavia työkaluja.

8.2 Kehitys

Esimerkkisovellus kehitetään neljässä työvaiheessa kuvan 7.16 mukaisesti. Ennen tarkempaa esimerkkisovellukseen perehtymistä, tulee lukijan ensin huolellisesti tutustua edellisissä luvuissa esitettyihin sovellusohjelmointia käsitteleviin alalukuihin.

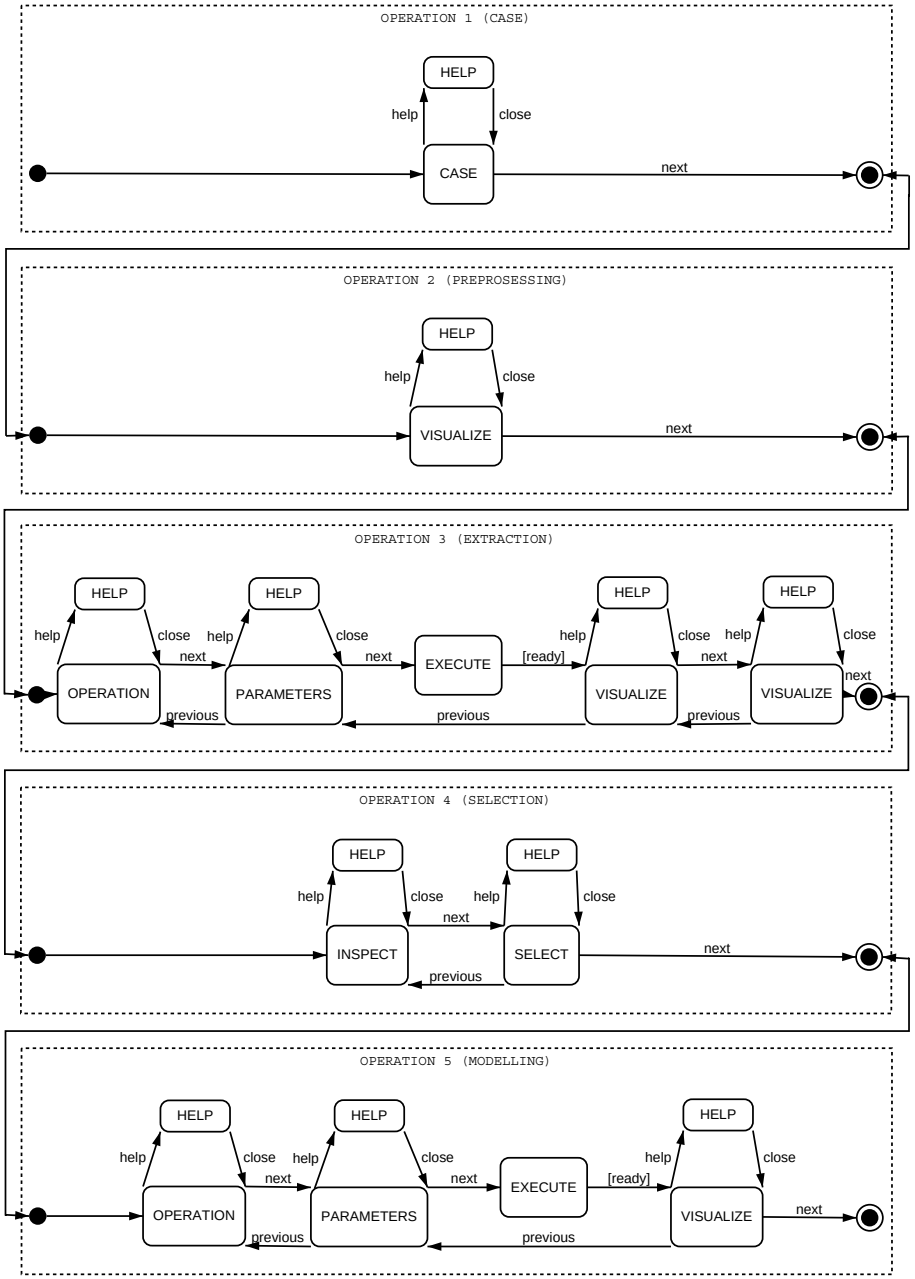
Edellä kuvattu sovelluksen toiminta voidaan piirtää tilakoneeksi kuvan 8.2 sivulla 96 mukaisesti. Samaistamalla tilakoneen tilat ikkunoihin ja toimeenpanovaiheisiin, voidaan kaavio muuttaa suoraan NDAIDE-makrokielelle.

8.2.1 Ensimmäinen vaihe

Ensimmäisessä vaiheessa kuvattiin sovelluksen analyysiketju NDAIDE.code-tiedostoon (taulu 8.1) muuntamalla kuvan 8.2 tilakone NDAIDE-kieliseksi esitykseksi.

Analyysin suoritus alkaa NDA-nimiavaruuden tyhjennyksellä, jotta uuden analyysikierroksen tulokset eivät häiriintyisi muistissa olevista vanhoista tuloksista. Tämän jäkeen esitellään analyysissä käytetyt grafiikkamuuttujat. Toimenpiteellä estetään virheellisten graafiikoiden esitys siinä tapauksessa, että makrojen toimeenpano keskeytyy virheeseen ennenkuin näytölle piirrettävät grafiikka-alkiot on määritelty. Alustustoimenpiteiden jälkeen sovellus kertoo käyttäjälle tietoja sovelluksen toteuttajasta ja käyttötarkoituksesta.

Varsinainen analyysiosa muodostuu kahdesta operaatioketjusta - piirteiden irrotuksesta ja piirteiden mallinnuksesta. Näiden lisäksi sovellukseen tarvitaan lisätyökaluja, joilla näytteiden valinta, esikäsittely ja piirteiden valinta suoritetaan.



Kuva 8.2: Kuituanalyysisovelluksen rakenne.

Näytteiden valinta suoritetaan koeasetelmaikkunalla, joka muodostaa NDA-nimiavaruuteen kaksi merkkijonolistaa, joissa luetellaan mukaan otettavat näytteet ja tutkittava näyte. Valitun kuvan esikäsittelyä varten laadi-

NDAIDE.code-tiedoston lähdekoodi:

```
# Clear namespace
Run,''if -exist Files -cmd rm -r *:''
Run,''mkgrp FilterGraphic1''
Run,''mkgrp FilterGraphic2''
Run,''mkgrp OperationGraphic1''
Run,''mkgrp OperationGraphic2''
Run,''mkgrp ModelGraphic1''
Run,''mkgrp ModelGraphic2''
# Brief the user
Briefing,''FIA'',''Welcome to Fibre Image Analysis'',Briefing,Briefing
# Select files
FileCase,''FIA'',''Select a CASE'',Files,Files
# Process selected image
Run,''PreFilter.cmd''
Graphic,''FIA'',''Filter selected image'',FilterGraphic1,FilterGraphic1
Run,''PostFilter.cmd''
# Extract features
Operation,''FIA'',''Select an operation'',Operation,Operation
Parameter,''FIA'',''Adjust parameters for operation'',Operation,OperationParameters
Execute,Operation
Graphic,''FIA'',''Feature extraction 1'',OperationGraphic1,OperationGraphic1
Graphic,''FIA'',''Feature extraction 2'',OperationGraphic2,OperationGraphic2
# Select features
Data,''FIA'',''Inspect the feature matrix'',Features,Features
DataCase,''FIA'',''Select the features for modelling'',FeaturesSelected,Features
# Do a model
Operation,''FIA'',''Select a model'',Model,Model
Parameter,''FIA'',''Adjust parameters for model'',Model,ModelParameters
Execute,Model
Graphic,''FIA'',''Model'',ModelGraphic1,ModelGraphic1
```

Taulukko 8.1: Esimerkkisovelluksen NDAIDE-kielinen lähdekoodi.

taan grafiikkaikkunalla ja NDALIB-kielellä yleisimpiä esikäsittelytyökaluja. Piirteiden irrotusmenetelmän valintaan käytetään operaatioikkunaa ja tämän parametrien asetukseen luonnollisesti parametri-ikkunaa. Piirteiden irrotuksen jälkeisiä tuloksia havainnollistetaan kahdella grafiikkaikkunalla. Varsinaista piirteisiin tutustumista varten käytetään taulukkoikkunaa piirrematriisin esitykseen, jonka jälkeen käyttäjä valitsee haluamansa piirteet piirreikkunasta. Mallin toteutus aloitetaan operaatioikkunalla suoritettavalla mallin

valinnalla ja tätä seuraavalla parametri-ikkunalla. Mallin rakennuksen jälkeen saavutettuja tuloksia esitellään grafiikkaikkunassa.

8.2.2 Toinen työvaihe

Toisessa työvaiheessa kuvattiin tarvittavat työkalut käyttäen NDALIB-makrokieltä sekä ikkunakohtaisia kuvauksia. Kuvaustiedostoja syntyi 18 ja niissä oli yhteensä noin 120 koodiriviä. Kaikki kuvaustiedostot on esitetty liitteessä C, jonka seuraaminen on olennaista sovelluksen ymmärtämiselle.

Piirteiden irrotuksessa ja mallinnuksessa esitettävät grafiikat poikkeavat niin paljon toisistaan, että niihin päätettiin sijoittaa vain PostScript-tiedostojen tallennustyökalu. Yhteinen työkalupalkki kuvattiin sääntöjen mukaisesti tiedostoon `default.ui`. Valitun kuvan esikäsittelyä varten kehitettiin työkalupalkki, joka sisältää viisi perustyökalua näytteen käsittelyyn. Esikäsittelyikkunan työkalupalkki kuvattiin tiedostoon `FilterGraphic1.ui`.

Loput tiedostot sisältävät ikkunakohtaisia kuvauksia, joiden syntaksi ja semantiikka on selitetty käyttöpaneelin ohjelmointia opettavassa alaluvussa.

8.2.3 Kolmas työvaihe

Kolmannessa työvaiheessa kirjoitettiin esimerkkisovelluksen sovelluslogiikka NDA-makrokielellä. Sovelluskäyttöliittymän toiminnallisuuden toteutukseen tarvittiin 18 NDA-makrokielistä tiedostoa, joissa oli yhteensä noin 1300 riviä. Kyseiset tiedostot on esitetty liitteessä D.

Aiemmista kuituanalyysiprojekteista oli jo ennalta olemassa joukko makrokielisiä esimerkkejä, joista otettiin oppia esimerkkisovelluksen rakennukseen. Varsinaisten analyysimenetelmien kehitys ei kuulunut tämän työn piiriin. Käyttöpaneelin toimintaperiaatteen mukaisesti suurin osa sovelluksen kehitykseen kuluneesta ajasta kului juuri NDA-makrokielisen sovelluslogiikan toteutukseen. Esimerkkisovelluksen tapauksessa ensimmäisen ja toisen työvaiheen suorittamiseen kului vain noin yksi kymmenesosa varsinaisen NDA-makrokielisen sovelluslogiikan ohjelmointiin kuluneesta ajasta. Jokaisen tiedoston alkuun kirjoitettiin lyhyt yhteenveto tiedoston kehityksestä ja käyttötarkoituksesta. Varsinaiset lähdekoodit kommentoitiin huolellisesti mahdollisen laajennettavuuden ja uudelleenkäytön helpottamiseksi. Lähdekoodien suuresta määrästä ja työn aihealueen määrittelystä johtuen varsinaisia lähdekoodilistauksia ei esitellä tarkemmin tämän työn puitteissa.

Analyysiohjelmiston valmistuttua parannettiin sovelluslogiikan suorituskykyä optimoimalla laskennallisesti raskaiden kokonaisuuksien toimintaa. Sovelluksen käytössä tutkitaan usein samojen näytteiden välisiä tuloksia. Sovelluksen toiminta nopeutui huomattavasti, kun kerran irrotetut piirteet tallen-

nettiin tiedostoihin, joista ne seuraavilla analysointikerroilla ladataan. Poikkeukseksi muodostuu analyysin kohteena oleva näyte, sillä siitä saataviin piirteisiin voidaan vaikuttaa esikäsittelyllä. Näin ollen tutkittavan näytteen piirteet irrotetaan aina näytteestä laskemalla. Ennen optimoinnin suoritusta sovellusta voitiin käyttää lähinnä alle kymmenen näytteen koeasetelmiin. Optimoinnin jälkeen sovellusta voidaan käyttää jopa sadan erillisen näytteen koeasetelmiin.

8.2.4 Neljäs työvaihe

Viimeisessä työvaiheessa suoritetaan sovelluksen viimeistely. Analyysiketjusta piirrettävää verkkoa varten hankittiin lähes kolmekymmentä eri tilannetta esittävää kuvatiedostoa. Kuvien kooksi valittiin 100x100 kuvapistettä ja kuvien välisiksi etäisyyksiksi 10 kuvapistettä.

Ajanpuutteen ja työn painopisteen vuoksi sovellukselle ei ryhdytty kirjoittamaan ikkunakohtaisia opasteita. Tarpeen vaatiessa opasteet on kuitenkin helppo lisätä olemassa olevaan sovellusrunkoon.

8.3 Esittely

Tässä alaluvussa esitellään tarkemmin valmistunut kuituanalyysisovellus. Kuvauksen sekaan on sijoitettu toteutukseen viittaavia tauluja, joiden avulla nähdään suoraan, mitkä tiedostot kuuluvat mihinkin analyysin osaan. Viittauksissa esitetyt lähdekoodit on listattu NDAIDE.code-tiedostoa lukuunottamatta liitteissä C ja D.

8.3.1 Kuva-aineiston hankinta

Kuva-aineiston näytteistys alkaa näytepaperin sijoittamisella kuvanlukijaan, jonka jälkeen kuvankäsittelyohjelmistoa hyväksikäyttäen suoritetaan pinnan näytteistys. Saatu kuva tarkistetaan silmämääräisesti ja epäonnistuneet kuvankaappausoperaatiot uusitaan säätämällä näytteistysparametreja. Kuvien näytteistyksessä on syytä käyttää samaa pistetarkkuutta, kuin näytekirjaston kuvissa, sillä muutoin eri skaaloihin kuvautuvat pinnanmuodot voivat häiritä analyysiprosessia. Jos näyte on liian tumma tai vaalea, eli näytepisteiden jakauman massa ei jakaudu riittävän tasaisesti, voidaan kuvaa käsitellä kuvankäsittelyohjelmistosta löytyvin menetelmin. Hyvän lopputuloksen jälkeen kuvasta rajataan koko pintaa edustava alue ja näin saatava näyte tallennetaan tiedostoon GIF (Graphics Interchange Format) -formaattiin.

Harmaasävykuvat muodostavat topografisen kartan näytepaperin pinnanmuodoista. Tummat näytepisteet edustavat paperipinnan matalia kohtia ja vaaleat vastaavasti hieman koholla olevia pisteitä. Näytteistystarkkuutena käytetään kahdeksaa bittiä näytepistettä kohti, jolloin saavutetaan varsin riittävä erottelukyky matalien ja korkeiden pisteiden välille. Koska paperipinnan tekstuuri on yleensä kauttaaltaan lähes samanlaista, voidaan sovelluksen suorituskykyä parantaa näytteistämällä vain pieni alue, jolloin saavutetaan huomattavaa etua laskennallisesti raskaita analyysimenetelmiä käytettäessä. Analyysejä suoritettaessa on huomioitava, että kaikkien analyysiin osallistuvien näytteiden on oltava samankokoisia. Käytännössä 512x512 näytepistettä muodostaa riittävän laajan otannan, joskin analyysit voidaan suorittaa millä tahansa symmetrisellä kakkosen potenssin kokoisella näytteellä.

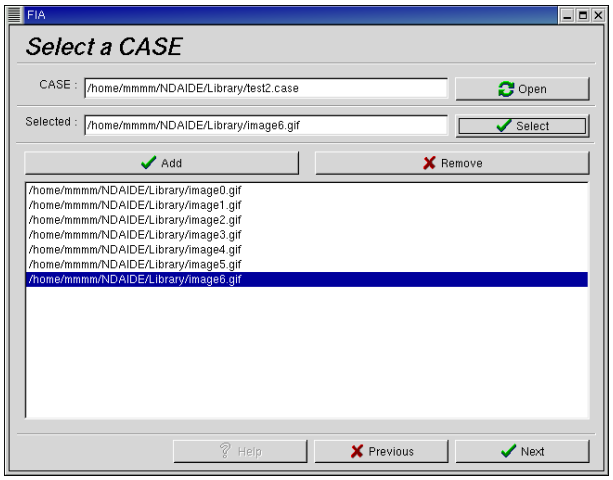
Sovellus ei tarjoa valmiita työkaluja kuva-aineistojen näytteistykseen, vaan näytteet on kerättävä kolmansien osapuolten tuottamilla työkaluilla. Kuituanalyysiprojektin tavoitteena oli luoda helposti hankittava ja edullinen järjestelmä. Niinpä paperipinnan näytteistys voidaan suorittaa sopivin asetuksin tavallisella tasokuvanlukijalla. Näytteiden esikäsittelyyn voidaan käyttää kaupallisia kuvankäsittelyohjelmistoja, jotka tarjoavat lähes poikkeuksetta riittävät työkalut kaikkiin työvaiheisiin.

8.3.2 Koeasetelman määrittely

Koeasetelman määrittelyssä on kiinnitettävä huomiota siihen, että analyysiin otetaan mukaan näytejoukko, joka edustaa tasapuolisesti koko otosavaruutta. Yksipuolisen ja toistuvan näyteavaruuden käyttö johtaa helposti totuuden vastaisiin malleihin. Kuvassa 8.3 valitaan koeasetelmaan kuuluvia näytteitä.

Kuituanalyysisovelluksessa voidaan valita vapaasti haluttu määrä näytteitä. Suuria näytejoukkoja käsiteltäessä on huomioitava tarvittavan laskenta-ajan kasvu. Kokeiden mukaan sovellusta voidaan käyttää ainakin sadan näytteen kokoisiin asetelmiin asti. Tulosten luotettavuuden parantamiseksi kannattaa analyysit suorittaa ainakin kymmenen näytteen kokoisilla koeasetelmilla.

Uuden näytteen analyysiin liitetään kyseisen näytetiedoston lisäksi tietokantaan ennalta tallennettuja näytetiedostoja, jolloin uuden näytteen tuloksia voidaan verrata tuttuihin vertailuaineistoihin. Jos kaikki analyysiin liitettävät näytteet ovat ennalta käsittelemättömiä, voi mallin antamien tulosten vertailu osoittautua mahdottomaksi. Koeasetelman määrittelyä on havainnollistettu kuvassa 8.3.



Kuva 8.3: Koeasetelman määrittely.

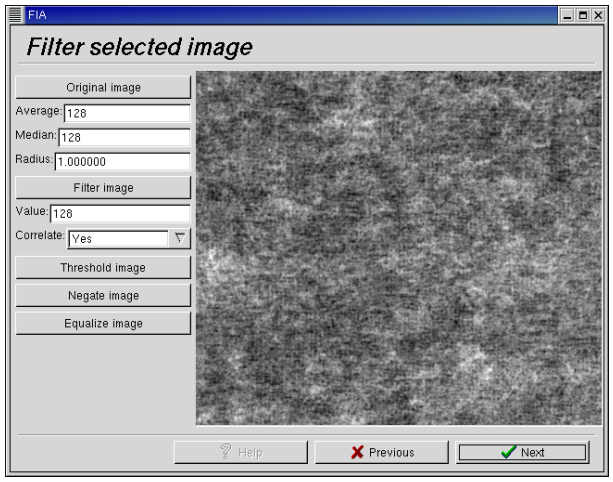
NDAIDE-lähdekoodi:
`FileCase,"FIA","Select a CASE",Files,Files`
Tiedostot:
Vaihe 1: `NDAIDE.code`-tiedoston rivi 18 **Vaihe 2:** `Files.ui` **Vaihe 3:** - **Vaihe 4:** `Files.bmp, Files.help`
NDA-muuttujat:
Sisään: - **Ulos:** `Files` sisältää luettelon kaikista mukana olevista näytetiedostoista polkuineen. `FilesSelected` sisältää luettelon valitusta näytetiedostosta polkuineen.

8.3.3 Kuva-aineiston esikäsittely

Esikäsittelyllä pyritään yhtenäistämään näytteiden esitystä siten, että niistä laskennallisesti saavutettavat tulokset olisivat keskenään vertailukelpoisia. Vaihtelevat kontrastit, skaalat ynnä muut eroavaisuudet johtavat siihen, että jopa samaa pintaa esittävät näytteet voivat johtaa täysin toisistaan eroaviin piirteisiin. Väärin muodostetuista piirteistä laadittavat mallit eivät luonnollisestikkaan edusta realimaailman tilannetta ja siten saavutettuja tuloksia ei voida hyödyntää tavoitteena olevaan paperilaatujen luokitukseen.

Sovelluksen tarjoamilla työkaluilla voidaan tutkimuksen kohteena olevalle näytteelle suorittaa lopullista viimeistelyä. Lähtökohtana voidaan pitää, että

näytteiden esikäsittely suoritetaan kuva-aineiston hankintavaiheessa ja tarjolla olevia työkaluja käytetään lähinnä pienimuotoisten muutosten tekoon. Analyysiä suoritettaessa ei aina voida olla varmoja alkuperäisen esikäsittelyn toimivuudesta. Käyttämällä sovelluksen tarjoamia työkaluja voidaan näytteen esitystä hienosäätää iteratiivisesti kokeilemalla, jolloin saavutettujen analyysitulosten pohjalta voidaan valita lopullinen näytteen esitysmuoto. Esikäsittelyn suoritusta on havainnollistettu kuvassa 8.4.



Kuva 8.4: Kuva-aineiston esikäsittely.

Esikäsittelyikkunan työkalut:

- Painikkeella `[Original image]` ladataan kuvan alkuperäinen esitys.
- Painikkeella `[Filter image]` suoritetaan kuvalle valinnan mukaisesti joko keskiarvo- tai mediaanisuodatus.
- Painikkeella `[Threshold image]` suoritetaan kuvan kynnystys, jossa kynnysarvoa matalammat pisteet korvataan mustalla ja korkeammat valkoisella.
- Painikkeella `[Negate image]` käännetään näytteen korkeuskartta ylösalaisin. Näytteen huippupisteistä tulee pohjapisteitä ja päinvastoin.
- Painikkeella `[Equalize image]` suoritetaan näytteen pisteiden jakauman tasapainotus siten, että käsittelyn jälkeen näyte käyttää esitykseensä koko 8-bitin tarkkuuden.

NDAIDE-lähdekoodi:

`Run,"PreFilter.cmd"`
`Graphic,"FIA","Filter selected image",FilterGraphic1,FilterGraphic1`
`Run,"PostFilter.cmd"`

Tiedostot:
Vaihe 1: `NDAIDE.code`-tiedoston rivit 22-24 **Vaihe 2:** `FilterGraphic1.ui` **Vaihe 3:** `PreFilter.cmd, Filter.cmd, Filter0.cmd, Filter1.cmd, Filter2.cmd, Filter3.cmd, PostFilter.cmd` **Vaihe 4:** `FilterGraphic1.bmp, FilterGraphic1.help`
NDA-muuttujat:
Sisään: `FilterGraphic1` määrää käytetyn grafiikan nimen. **Sisäiset:** `FilesSelected` sisältää luettelon valitusta näytetiedostosta polkuineen. **Ulos:** -

8.3.4 Piirteiden irrotus

Kuituanalyysiin tuotavat paperipinnan näytteet sisältävät runsaasti informaatiota, jonka selitysaste on alhainen. Esimerkiksi 512x512 näytepisteen tarkkuudella yksi näyte kuluttaa esitykseensä 2097152 bittiä. Pidemmälle menevien johtopäätösten tekemiseksi on näytteiden selitysastetta lisättävä ja esityskokoa pienennettävä. Piirteiden irrotuksella kerätään pieni joukko tunnuslukuja, jotka kuvaavat alkuperäisen näytteen oleelliset ominaisuudet. Tavoitteena on luoda kunkin näytteen piirteistä vektorihahmo, jolloin voidaan laskea kullekin näytteelle tiettyyn luokkaan kuulumisen todennäköisyys.

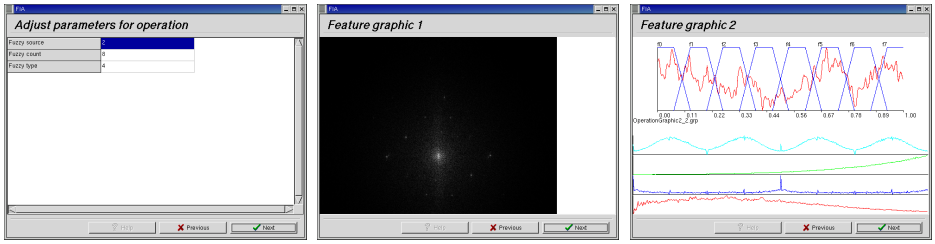
Fourier-muunnos

Fourier-muunnosta käytetään näytteen informaation jaksollisuuden tutkimiseen. Kuvassa 8.5 esitetään esimerkki kolmen tarvittavan työvaiheen suorittamisesta.

Fourier-muunnoksen suoritus aloitetaan tarvittavien parametrien asetuksella. Parametreilla valitaan piirteiden irrotukseen käytettävä jakauma, taajuuskaistojen määrä ja muoto.

Fourier-spektristä voidaan silmämääräisesti havaita näytteessä esiintyviä säännöllisyyksiä. Ensimmäisessä grafiikkaikkunassa näkyvän esimerkkitapauksen säännölliset pistejonot muodostuvat telan paperiin jättämistä säännöllisistä kuvioista.

Piirteiden irrotusta varten lasketaan Fourier-spektrin tehon ja kulman jakaumat sekä karteesisessa että polaarisessa koordinaatistossa. Lasketut ja-



Kuva 8.5: Piirteiden irrotus Fourier-muunnoksella.

kaumat näkyvät operaation toisen grafiikkaikkunan alalaidassa. Varsinaista piirteiden irrotusta varten valitaan yksi edellä mainituista jakaumista. Valittu jakauma jaetaan taajuuskaistoihin jonka jälkeen sumealla jäsenyysfunktiolla määritetään kunkin kaistan käyttöaste. Piirteiden muodostus kaistojen käyttöasteista havainnollistetaan toisen grafiikkaikkunan ylälaitaan piirrettävällä luokituksella.

NDAIDE-lähdekoodi:

```
Operation,"FIA","Select an operation",Operation,Operation
Parameter,"FIA","Adjust parameters for operation",Operation,OperationParameters
Execute,Operation
Graphic,"FIA","Feature extraction 1",OperationGraphic1,OperationGraphic1
Graphic,"FIA","Feature extraction 2",OperationGraphic2,OperationGraphic2
```

Tiedostot:

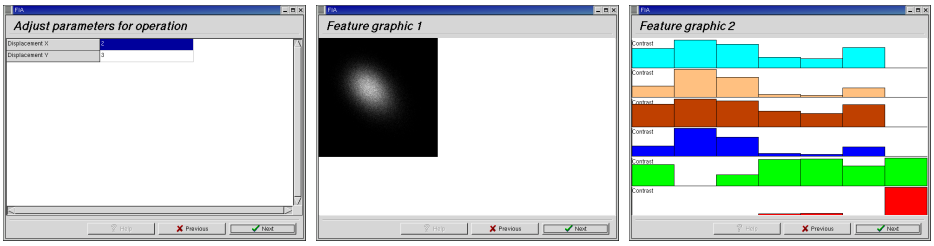
Vaihe 1: `NDAIDE.code`-tiedoston rivit 28-32
Vaihe 2: `Operation.ui, Operation0.ui, default.ui`
Vaihe 3: `Operation0.cmd`
Vaihe 4: `Operation0.bmp, Operation0.help`

NDA-muuttujat:

Sisään: `OperationParameters` tallentaa piirteiden irrotukseen käytettävät parametrit.
`OperationGraphic1` ja `OperationGraphic2` määrävät käytettyjen NDA-grafiikkojen nimet.
Sisäiset: `Files` sisältää luettelon kaikista mukana olevista näytetiedostoista polkuineen.
`FilesSelected` sisältää luettelon valitusta näytetiedostosta polkuineen.
Ulos: `Features` sisältää irrotetut piirteet taulukossa.

Yhteisesiintymismatriisi

Yhteisesiintymismatriisilla voidaan tutkia näytteessä esiintyvien toistuvuuksien todennäköisyyksiä. Kuvassa 8.6 esitetään esimerkki kolmen tarvittavan työvaiheen suorittamisesta.



Kuva 8.6: Piirteiden irrotus yhteisesiintymismatriisilla.

Yhteisesiintymismatriisin luomiseksi asetetaan parametreilla tekstuuria tutkivan sapluunan siirtymät X- ja Y-suunnassa. Sapluunan siirto vaikuttaa työkalun kykyyn tunnistaa tiettyjä muotoja. Esimerkiksi diagonaalinen siirto tunnistaa parhaiten juuri viistottain esiintyvää toistuvuutta.

Yhteisesiintymismatriisista laaditaan ensimmäiseen grafiikkaikkunaan bittikarttaesitys, jonka avulla voidaan päätellä, missä näytteen kohdissa toistuvuuden määrä on suurimmillaan.

Yhteisesiintymismatriisin piirteet hankitaan laskemalla matriisista tilastollisia piirteitä. Piirteiden pohjalta laaditaan toiseen grafiikkaikkunaan kullekin piirteelle histogrammi, jolla voidaan vertailla näytteiden välistä vaihtelua.

NDAIDE-lähdekoodi:

```
Operation,"FIA","Select an operation",Operation,Operation
Parameter,"FIA","Adjust parameters for operation",Operation,OperationParameters
Execute,Operation
Graphic,"FIA","Feature extraction 1",OperationGraphic1,OperationGraphic1
Graphic,"FIA","Feature extraction 2",OperationGraphic2,OperationGraphic2
```

Tiedostot:

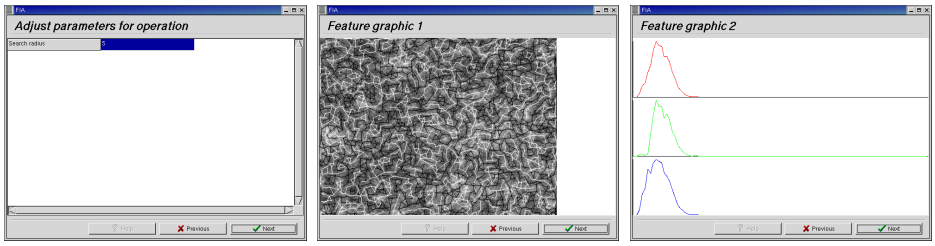
Vaihe 1: `NDAIDE.code`-tiedoston rivit 28-32
Vaihe 2: `Operation.ui, Operation1.ui, default.ui`
Vaihe 3: `Operation1.cmd`
Vaihe 4: `Operation1.bmp, Operation1.help`

NDA-muuttujat:

Sisään: `OperationParameters` tallentaa piirteiden irrotukseen käytettävät parametrit.
`OperationGraphic1` ja `OperationGraphic2` määrävät käytettyjen NDA-grafiikkojen nimet.
Ulos: `Features` sisältää irrotetut piirteet taulukossa.

Delaunay-kolmiointi

Näytteen topografinen kartta sisältää joukon paikallisia huippu- ja kuoppapisteitä. Paperipinnan tapauksessa huiput ja kuopat muodostavat paljon jonoja, joista voidaan päätellä rakenteen vahvuutta. Kuvassa 8.7 esitetään esimerkki kolmen tarvittavan työvaiheen suorittamisesta.



Kuva 8.7: Piirteiden irrotus Delaunay-kolmioinnilla.

Delaunay-kolmioinnin suoritus alkaa huippu- ja kuoppapisteiden pienimmän sallitun välimatkan määrittelyllä. Menettelyllä määrätään, mitkä pisteet edustavat saamaa ja mitä erillisiä ääriarvoja.

Pisteiden määrittelyn jälkeen suoritetaan varsinainen kolmiointi, jossa vierekkäin olevat ääriarvopisteet yhdistetään verkon kaarilla. Tämän jälkeen poistetaan ylimääräiset kaaret siten, että verkot eivät ole missään kohtaa pää-

lekkäin. Muodostunut verkko havainnollistetaan piirtämällä se ensimmäisessä grafiikkaikkunassa näytekuvan päälle.

Verkon piirteiden määrittelemiseksi lasketaan verkon kaikkien kaarien, huippuja yhdistävien kaarien ja kuoppia yhdistävien kaarien jakaumat. Jakaumien havainnollistamiseksi piirretään toiseen grafiikkaikkunaan niiden viivadiagrammit. Jakaumainformaation tiivistämiseksi lopulliset piirteet muodostetaan edellä mainittujen jakaumien tunnusluvuista.

NDAIDE-lähdekoodi:

`Operation,"FIA","Select an operation",Operation,Operation`
`Parameter,"FIA","Adjust parameters for operation",Operation,OperationParameters`
`Execute,Operation`
`Graphic,"FIA","Feature extraction 1",OperationGraphic1,OperationGraphic1`
`Graphic,"FIA","Feature extraction 2",OperationGraphic2,OperationGraphic2`

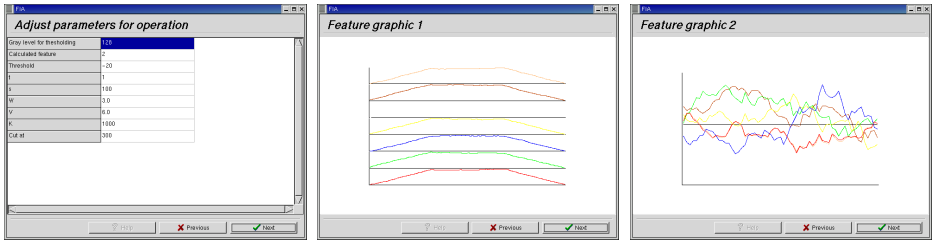
Tiedostot:
Vaihe 1: `NDAIDE.code`-tiedoston rivit 28-32 **Vaihe 2:** `Operation.ui, Operation2.ui, default.ui` **Vaihe 3:** `Operation2.cmd` **Vaihe 4:** `Operation2.bmp, Operation2.help`
NDA-muuttujat:
Sisään: `OperationParameters` tallentaa piirteiden irrotukseen käytettävät parametrit. `OperationGraphic1` ja `OperationGraphic2` määrävät käytettyjen NDA-grafiikkojen nimet. **Ulos:** `Features` sisältää irrotetut piirteet taulukossa.

BLOB-analyysi

Paperipinnan tekstuuri muodostuu pienistä alkeisosasista, kuten kuiduista, liima-aineesta ja roskista. Tarkasti katsottuna nämä alkeisosat muodostavat paperin pintaan pieniä kuvioita, joiden ominaisuuksia voidaan tutkia. Kuvassa 8.8 esitetään esimerkki kolmen tarvittavan työvaiheen suorittamisesta.

Analyysiä varten valitaan kynnystysarvo, jolla näyte käsitellään alkeisosien erottelemiseksi. Lisäksi valitaan laskettava piirre ja jakaumien epäsäännöllisyyksien tasoittamiseen käytettävä suodin parametreineen.

Piirteiden irrotus suoritetaan erottelemalla alkeisosaset toisistaan, jonka jälkeen kunkin osasen piirteet voidaan laskea. Koska näytteessä on aina lukuisia alkeisosasia, muodostuu piirteen alkioista jakauma, jota voidaan vertailla eri näytteiden välillä. Ensimmäinen grafiikkaikkuna piirtää kaikkien näytteiden jakaumat. Jakaumien välisen vertailun helpottamiseksi piirretään toiseen grafiikkaikkunaan jakaumien väliset erot, jolloin nähdään selvästi,



Kuva 8.8: Piirteiden irrotus alkeisosien muodoista.

mitkä jakaumat poikkeavat keskiarvosta milläkin kohtaa. Varsinaiset piirteet muodostetaan laskemalla piirrejakaumien yleisimpiä tunnuslukuja.

NDAIDE-lähdekoodi:

```
Operation,"FIA","Select an operation",Operation,Operation
Parameter,"FIA","Adjust parameters for operation",Operation,OperationParameters
Execute,Operation
Graphic,"FIA","Feature extraction 1",OperationGraphic1,OperationGraphic1
Graphic,"FIA","Feature extraction 2",OperationGraphic2,OperationGraphic2
```

Tiedostot:

Vaihe 1: `NDAIDE.code`-tiedoston rivit 28-32
Vaihe 2: `Operation.ui, Operation3.ui, default.ui`
Vaihe 3: `Operation3.cmd`
Vaihe 4: `Operation3.bmp, Operation3.help`

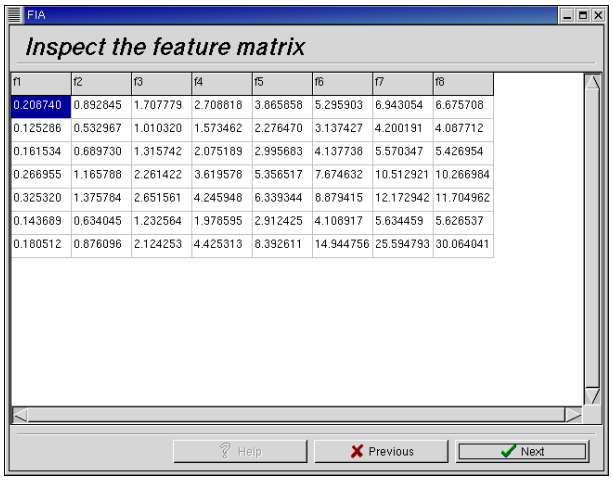
NDA-muuttujat:

Sisään: `OperationParameters` tallentaa piirteiden irrotukseen käytettävät parametrit.
`OperationGraphic1` ja `OperationGraphic2` määrävät käytettyjen NDA-grafiikkojen nimet.
Ulos: `Features` sisältää irrotetut piirteet taulukossa.

8.3.5 Piirteiden valinta

Piirteiden irrotuksen tuloksena syntyy joukko piirrevektoreja, jotka kuvaavat kukin kaikkien näytteiden jotain tiettyä ominaisuutta. Piirteiden määrä vaihtelee käytetystä menetelmästä ja sen parametreista riippuen. Yleensä luotavaa mallia varten käytetään vain osaa piirteistä.

Ennen valintaa käyttäjällä on mahdollisuus tutustua laskettuihin piirrevektoreihin. Piirteet esitetään taulukolla, jossa rivit muodostavat yhdestä näytteesä lasketut erilliset piirteet ja sarakkeet kaikista näytteistä lasketun tietyn piirteen. Taulukon rivit on järjestetty siten, että ensimmäisellä rivillä

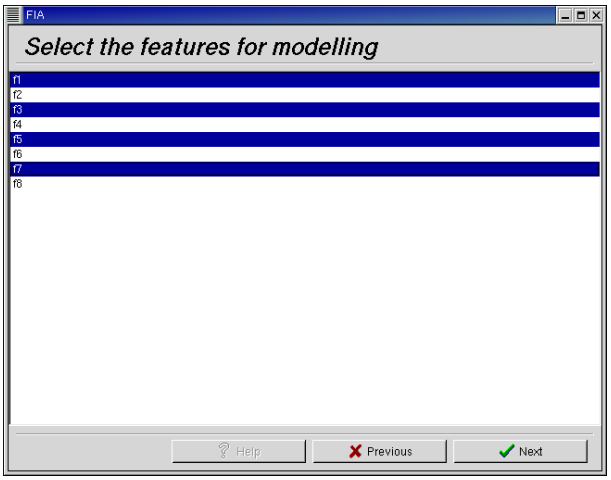


Kuva 8.9: Piirteisiin tutustuminen.

sijaitsee ensimmäisestä näytteestä saadut tulokset toisella toisen näytteen ja niin edelleen. Jos piirteiden arvot eivät sovi kunnolla näkyviin, voidaan niiden näkyvyyttä parantaa leventämällä sarakkeita. Suurten taulukoiden tapauksessa on lisäksi käytettävä taulukon sivuilla olevia vierityspalkkeja. Kuvassa 8.9 tutustutaan esimerkin piirteisiin.

NDAIDE-lähdekoodi:
`Data,"FIA","Inspect the feature matrix",Features,Features`
Tiedostot:
Vaihe 1: `NDAIDE.code`-tiedoston rivi 36 **Vaihe 2:** - **Vaihe 3:** - **Vaihe 4:** `Features.bmp, Features.help`
NDA-muuttujat:
Sisään: `Features` esittää lasketut piirteet taulukkomuodossa. **Ulos:** -

Tutustuttuaan laskettuihin piirteisiin käyttäjä valitsee niistä ne, jotka otetaan mukaan jatkossa laadittavaan malliin. Valinta suoritetaan merkitsemällä piirteiden luettelosta ne piirteet, joista ollaan kiinnostuneita. Monien graafisten käyttöliittymien tapaan monivalinnan suorittamiseksi on käyttäjän painettava `Ctrl` tai `Shift` -näppäimiä. Kuvassa 8.10 suoritetaan esimerkin piirteiden valinta.



Kuva 8.10: Piirteiden valinta.

NDAIDE-lähdekoodi:
`DataCase,"FIA","Select the features for modelling",FeaturesSelected,Features`
Tiedostot:
Vaihe 1: `NDAIDE.code`-tiedoston rivi 37 **Vaihe 2:** - **Vaihe 3:** - **Vaihe 4:** `FeaturesSelected.bmp, FeaturesSelected.help`
NDA-muuttujat:
Sisään: `Features` esittää lasketut piirteet taulukkomuodossa. **Ulos:** `FeaturesSelected` sisältää luettelon valittujen piirteiden nimistä (valituista piirretaulukon sarakkeista).

8.3.6 Piirteiden mallinnus

Näytteistä lasketut piirteet muodostavat moniulotteisen avaruuden, jonka tulkitseminen suoraan on vaikeaa. Havainnollisuuden lisäämiseksi valituista piirteistä voidaan luoda malleja, jotka tiivistävät piirreinformaatiota siten, että oleelliset yksityiskohdat korostuvat epäoleellisten kustannuksella, jolloin ihmissilmän on helpompi omaksua esitetty informaatio.

Kuituanalyysisovellus käyttää itseorganisoituvaa piirrekarttaa piirreavaruuden ryhmittelyyn. Samanlaiset näytteet pyritään keräämään ryhmiin, jolloin näytteen ryhmitystä voidaan käyttää apuna paperilaadun luokittelussa. Mallit laaditaan käyttäen puurakenteista piirrekarttaa, jolloin mallia varten valitaan haluttu tarkkuus. Esimerkkiaineistoilla puun syvyys voidaan huoletta valita aina kahdeksankerroksiseksi, jolloin alin verkko koostuu 16384:stä

neuronista. Tämän jälkeen käytetyn algoritmin suoritusaika kohoaa nopeasti liian suureksi. Käytännössä kahdeksankerroksisen verkon erottelukyky on enemmän kuin riittävä, eikä tarvetta ylempien kerrosten käyttöön juuri ole

Sovelluksen ensimmäinen versio tukee seitsemää mallia. Mallien toimintaa on havainnollistettu kuvilla 8.11 ja 8.11.

1. Statistiikka selvittää piirteiden sisäiset dynamiikat. Jokaiselle piirteelle lasketaan seuraavat tunnusluvut:

   (a) Minimi
   (b) Maksimi
   (c) Summa
   (d) Keskiarvo
   (e) Varianssi
   (f) Keskihajonta

   Esitystä varten tunnusluvut kerätään histogrammeihin, joilla eri piirteiden samoja tunnuslukuja voidaan vertailla.

2. Histogrammilla voidaan vertailla saman näytteen eri piirteiden dynamiikkaa keskenään. Jokaiselle näytteelle piirretään oma histogrammi.

3. Luokitettu statistiikka jakaa piirteet SOM-kartan mukaisiin ryppäisiin ja laskee kunkin ryppään statistiikan.

4. Luokitettu histogrammi jakaa piirteet SOM-kartan mukaisiin ryppäisiin ja laskee kunkin ryppään histogrammin.

5. Fraktiili havainnollistaa piirteiden jakaumien prosentuaalisia koostumuksia. Taustalla oleva palkki ilmoittaa jakauman keskiarvon ja päälle piirrettävä väli pyydetyn fraktiilivälin.

6. Mosaiikki laatii näytteistä mosaiikkikartan, jossa näytteistä koostuvat palat järjestellään SOM-kartan mukaiseksi kuvioksi. Lopullisesta mosaiikkikartasta nähdään havainnollisesti näytteiden piirteiden keskinäisen samanlaisuus.

7. Haluttaessa laskea piirreaineiston dimensiota, voidaan mallintaa klassinen skaalaus ja Sammonin kartta. Klassinen skaalaus käyttää dimension redusointiin pääkomponenttianalyysia ja Sammonin kartta nimensä mukaisesti Sammonin karttaa. Lisäksi SOM-kartasta laaditaan U-matriisiesitys, jolla voidaan vertailla kartan neuronien etäisyyksiä toisistaan.

NDAIDE-lähdekoodi:

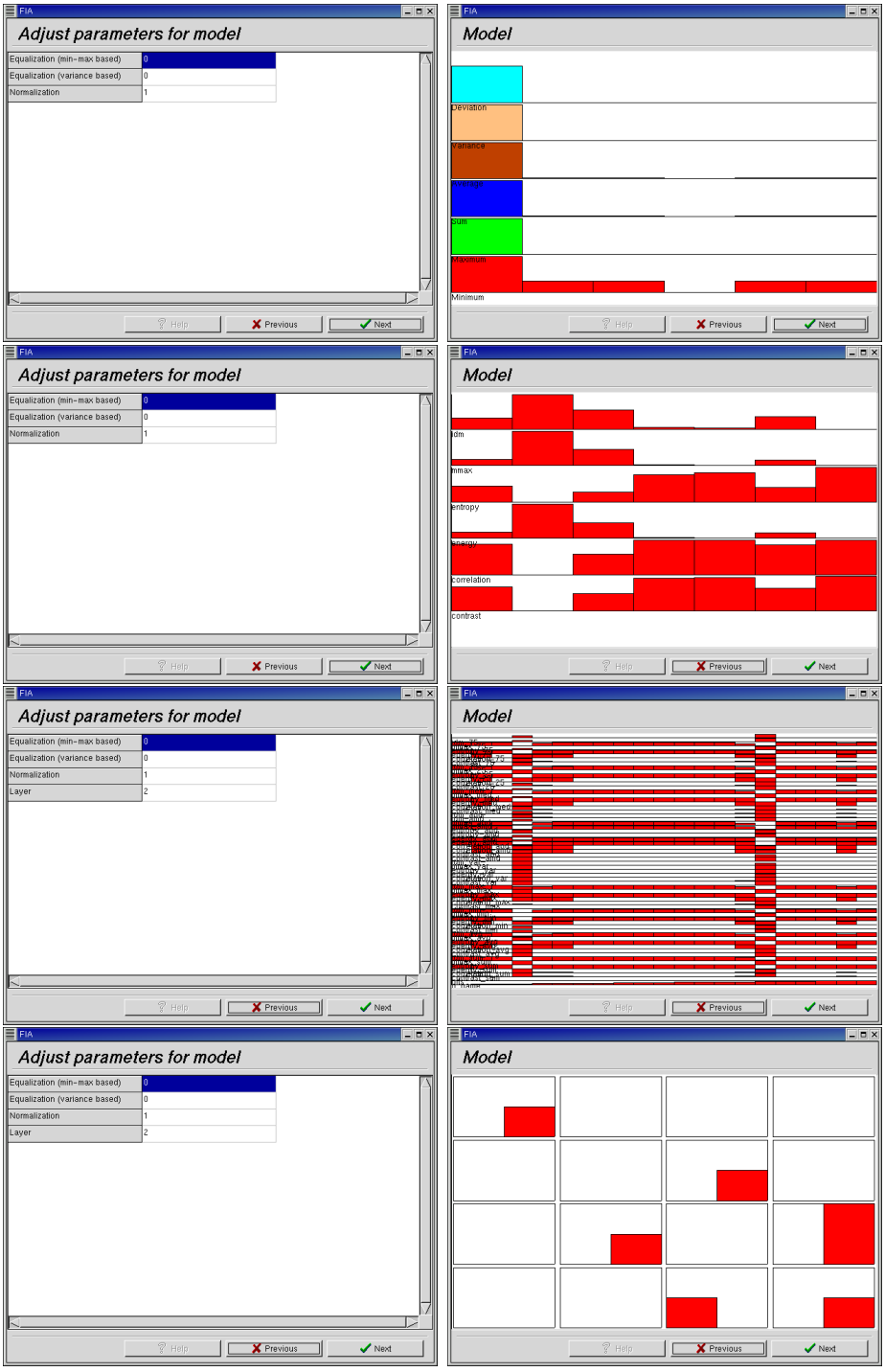
`Operation,"FIA","Select a model",Model,Model`
`Parameter,"FIA","Adjust parameters for model",Model,ModelParameters`
`Execute,Model`
`Graphic,"FIA","Model",ModelGraphic1,ModelGraphic1`

Tiedostot:
Vaihe 1: `NDAIDE.code`-tiedoston rivit 41-44 **Vaihe 2:** `Model.ui, Model?.ui`; ?=valitun mallin numero **Vaihe 3:** `Model?.cmd`; ?=valitun mallin numero **Vaihe 4:** `Model?.bmp, Model?.help`; ?=valitun mallin numero
NDA-muuttujat:
Sisään: `ModelParameters` tallentaa mallin parametrit. `ModelGraphic1` määrää käytettävän NDA-grafiikan nimen. **Sisäiset:** `FeaturesSelected` sisältää luettelon valittujen piirteiden nimistä (valituista piirretaulukon sarakkeista). `Features` sisältää varsinaisen piirreinformaation taulukkomuodossa. **Ulos:** -

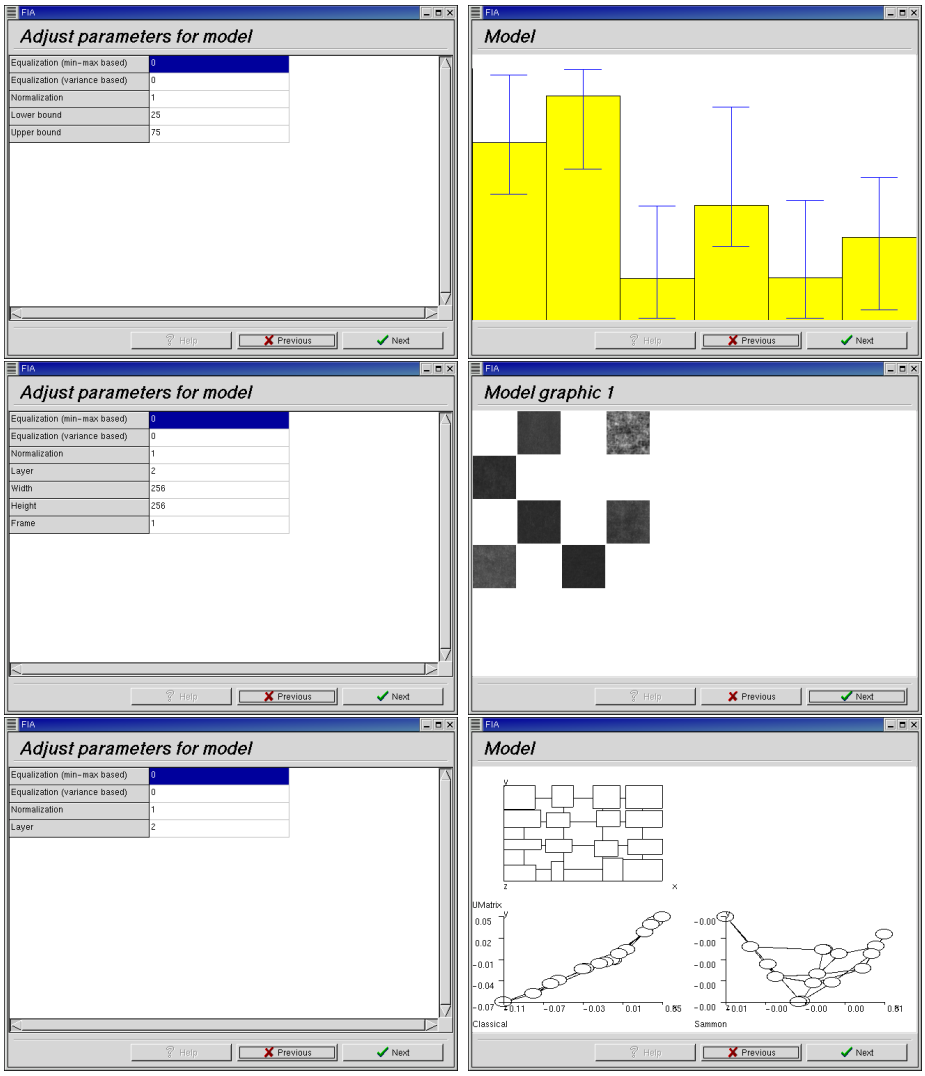
8.3.7 Näytetietokannan täydentäminen

Analyysien suorituksessa tarvitaan uuden näytteen lisäksi kattava tietokanta, johon näytteestä saavutettuja tuloksia voidaan verrata. Kuituanalyysisovelluksen mukana toimitetaan alustava tietokanta, jolla sovelluksen käyttö voidaan aloittaa. Pidemmän päälle ja laajempia analyysejä varten on kirjastoa kuitenkin täydennettävä mitattavan paperikoneen tuottamasta paperista saaduilla näytteillä.

Sovellus ei tarjoa valmiita työkaluja tietokannan hallintaan ja siten työvaihe suoritetaan käyttöjärjestelmän tarjoamilla palveluilla. Juuri analysoitu näyte lisätään tietokantaan kopioimalla sovelluksen hakupolusta Selected.gif -niminen tiedosto tietokannan hakupolkuun näytettä kuvaavalle nimelle. Tietokannan käytettävyyden ylläpitämiseksi kaikki tietokannan näytteet kannattaa tallentaa samalla tavalla esikäsitellyssä muodossa. Tiedoston kopioinnin jälkeen uusi näyte on osa tietokantaa ja sitä voidaan käyttää tulevissa analyyseissä referenssipisteenä.



Kuva 8.11: Piirteiden mallit 1-4.



Kuva 8.12: Piirteiden mallit 5-7.

Luku 9

Työn itsearviointi

Tässä luvussa tarkastellaan kuinka hyvin työssä suunniteltu ja toteutettu järjestelmä vastaa työlle asetettuja vaatimuksia. Yleisellä tasolla voidaan todeta, että ohjelmistokokonaisuus toteuttaa vaatimusmäärittelyn muutamia poikkeuksia lukuunottamata. Hyvään vastaavuuteen vaikuttaa osaltaan kevyt vaatimusten asettelu, joka antaa paljon liikkumatilaa. Ohjelmiston vastaavuutta yksittäisiin vaatimuksiin on käsitelty tarkemmin seuraavalla vaatimuskohtaisella tarkastelulla.

Toiminnallisten vaatimusten toteutuminen:

1. Suoritusalustariippumattomuus:
   (a) Käyttöliittymä on toteutettavissa graafisen työpöydän ohella tekstitilassa, jolloin esitetyt grafiikat tulostetaan luonnollisesti näytön sijasta joko PostScript-tiedostoihin tai paperille. Käyttöliittymän ohjaus voitaisiin tarvittaessa toteuttaa näppäimistön ja hiiren ohella myös muunlaisilla ohjaimilla. Muiden ohjaimien käyttö tulee kuitenkin kysymykseen lähinnä sulautetuissa teollisuusympäristöissä.
   (b) Käyttöliittymän siirrettävyyden osalta tehtiin työn alkuvaiheessa kompromissi, jossa kohdealustojen määrä rajoitettiin Intellin 80x86-teknologiaan pohjautuviin Windows ja Linux -käyttöympäristöihin. Tarvittaessa ohjelmisto voidaan suorittaa etäajona palvelimelta, jolloin ohjelmistoa voidaan hyödyntää esimerkiksi Solaris-työasemilla. Siirrettävyyden osalta pääpaino sijoitettiin lähdekoodien yhteneväisyyteen ja siinä onnistuttiinkin muutamia poikkeuksia lukuunottamata erinomaisesti. Toteutetut lähdekoodit ovat käytännössä täysin siirrettäviä kehitysympäristöinä käytettyjen Delphi 6:n ja Kylix 1:n välillä.

(c) Käyttöliittymän toteutus käytetyllä kehitysympäristöllä johti valmiiden käyttöliittymäkomponenttien osalta automaattisesti ohjesäännöstön noudattamiseen. Käyttöpaneeliin toteutetut ikkunat oli yksinkertaisuudessaan helppo suunnitella säännösten mukaisiksi.

2. Laajennettavuus ja uudelleenkäytettävyys:

   (a) Käyttöliittymän yleiskäyttöisyyttä on vaikea arvioida ennenkuin sillä on toteutettu useampia sovelluksia. Kirjoitushetkellä ohjelmistoa on sovellettu kahden erilaisen sovellusohjelman toteutukseen. Ensimmäisen sovelluksen toteutuksessa ei esiintynyt ongelmia. Toinen sovellus kärsi hieman käytetyn kielen korkeasta abstraktiotasosta, joka ei täysin soveltunut sovelluksen tarpeisiin. Puute olisi voitu korjata joko käyttämällä komponenttikirjastoa tai toteuttamatta jäänyttä suoraa NDA-komentomallia.

   (b) Käyttöpaneelin osalta sovellusohjelmistojen kehitystä varten ei tarvita erillisiä ohjelmistonkehitystyökaluja, vaan koko sovelluksen toteutus suoritetaan käytettävissä olevilla makrokielillä. Toteutettaessa suurempia sovelluksia, joissa tarvitaan enemmän vapauksia, jolloin turvaudutaan komponenttikirjaston käyttöön, joudutaan luonnollisesti käyttämään Delphi/Kylix -kehitysympäristöä.

   (c) Koska sovellusohjelmistojen toteutus perustuu makrokielisiin tiedostoihin, jotka toimitetaan lähdekoodimuodossaan, voidaan sovelluksia muunnella ja laajentaa samoin vapauksin kuin sovelluksen toteutusvaiheessa.

   (d) Sovellusohjelmistojen uudelleenkäytettävyys riippuu täysin sovellusohjelman toteutuksesta ja luotujen sovellusosien toiminnallisesta yhteensopivuudesta. Lähtökohtana voidaan pitää, että yleensä lähdekoodikieliseen ohjelmointiin ei tarvitse turvautua.

   (e) Käyttöpaneelin toteutuksen pääpaino on analyysiketjujen ohjaamisessa ja niiden tulosten visualisoinnissa. Tiedon organisointiin toteutetut työkalut ovat varsin suppeita, eivätkä sellaisenaan riitä laajempaan koeasetelman hallintaan.

3. Käyttäjän huomioiminen:

   (a) Sovellusohjelmistojen toteutuksen työmäärää dominoi lähes täysin tarvittavien NDA-makrotiedostojen toteutus. Periaatteessa kehitysideioita voidaan kokeilla nopeasti mutta väitteen todenperäi-

syyden vahvistamiseen ei vielä tässä vaiheessa ole riittävää näyttöä.

(b) Sovellusohjelmistojen helppo ohjelmoitavuus ja laajennettavuus mahdollistaa käyttäjän siirtymisen käyttäjäkategoriasta soveltajakategoriaan jo varsin pienellä kokemuspohjalla. Varsinaiseksi ohjelmoijaksi kehittyminen riippuu lähinnä NDA-ytimen toimintaperiaatteiden omaksumisesta.

Ei-toiminnallisten vaatimusten toteutuminen:

1. Käyttöliittymästä on toteutettu tavoitteiden mukainen graafinen versio.

2. Käyttöliittymän suorituskyky on riittävä, mahdollistaen sen käytön myös hieman vanhemmilla työasemilla. Sovellusohjelmiston suorituskyky riippuu lähinnä käytettyjen makrotiedostojen laskennallisesta vaativuudesta.

3. Käyttöliittymän vakautta on vaikea arvioida kirjoitusvaiheessa toteutetuilla sovellusohjelmistoilla. Toteutetut sovellukset toimivat kuitenkin ongelmitta.

Rajoitteet:

1. Ohjelmiston suunnittelussa valittujen toteutusteknologioiden valinnassa yhtenä kriteerinä käytettiin niiden soveltuvutta työmäärältään opinnäytteeksi. Toteutettavat ohjelmistot valmistuivat ajallaan ja siten valintojen voidaan katsoa onnistuneen toivotulla tavalla.

2. Ohjelmiston toteutus päätettiin suorittaa Borland Corp.:in toteuttamilla kehitystyökaluilla, joiden käyttöön yliopistolla on olemassa edulliset lisenssit. Näin ollen ohjelmistotyökaluista ei aiheutunut projektille merkittäviä kustannuksia. Mahdollisia tutkimus ja kaupallisia sovelluksia varten joudutaan hankkimaan kaupallinen lisenssi, joka tosin on varsin edullinen.

3. Käytetty ohjelmistonkehitysympäristö sisälsi riittävät työkalut ohjelmiston tuottamiseen, eikä ylimääräisien kaupallisten kirjastojen käyttöön ollut tarvetta. Esimerkkiohjelmiston tarpeisiin oli saatavilla riittävästi ilmaista oheismateriaalia, jonka avulla myös esimerkkiohjelmistoa voidaan levittää ilman kustannuksia.

Ohjelmiston käytettävyydestä on vielä tässä vaiheessa vaikea antaa tarkkoja arvioita. Ohjelmistoissa esiintyneet virheet on korjattu heti löytymisensä jälkeen ja siten toteutetut sovellukset toimivat moitteettomasti.

Komponenttikirjaston integroituminen osaksi ohjelmistonkehitysympäristöä tekee uusien ohjelmistojen laatimisen miellyttävän yksinkertaiseksi ja nopeaksi. Uuden NDA-ohjelmiston toteutus onnistuu lähes täysin RAD-työkaluista tutulla piirtelyllä, eikä varsinaista systeemiohjelmointia juuri tarvita sovelluslogiikan toteutuksen nojatessa NDA-makrotiedostoihin.

Käyttöpaneelin sovellusohjelmointi tarvitsee toimiakseen vain vähän käyttöliittymän kuvaukseen tarvittavaa makrokoodia. Käyttöliittymän toteutuksessa on havaittu muutamia käyttäjän toimintaa vaikeuttavia puutteita. Tehokkaan moniajon puuttuminen johtaa käyttöliittymän lukkiutumiseen NDA-makrojen suoritusajaksi. Lisäksi analyysiprosessin käynnistäminen lukitsee muut näkyvissä olevat ikkunat. Molemmat ongelmat juontavat juurensa samasta lähtokohdasta. Lineaarisesti etenevä analyysiprosessi voi antaa käyttäjälle aikaa vain silloin, kun käyttäjältä tarvitaan syötettä. Koska ongelmaa ei voitu ratkaista pelkästään muodostamalla ohjelmiston rakenne säikeistetyksi, ja koska ongelmasta käyttäjälle aiheutuva haitta on lähinnä esteettinen, päätettiin ongelman ratkaisua siirtää mahdollisesti tuleviin versioihin. Käyttöpaneelin onnistuneimpana piirteenä voidaan pitää sen kykyä muodostaa graafisesti toteutettu mutta silti pelkistetyn yksinkertainen ja kuitenkin toiminnallisesti monipuolinen käyttöliittymä.

Esimerkkisovellukseksi toteutettu paperin kuituanalyysiohjelmisto esittelee havainnollisesti lähes kaikki ohjelmistotyön tulokset. Pelkän toteutettujen teknologioiden havainnollistamisen ja validoinnin lisäksi ohjelmiston on tarkoitus kyetä suorittamaan paperianalyysejä käytännön olosuhteissa. Koska ohjelmistoa ei ole käytetty käytännön olosuhteissa, ei sen soveltuvuudesta käyttötarkoitukseensa voida esittää tarkkoja väitteitä.

Luku 10

Yhteenveto

Vaatimusmäärittelyn pohjalta suoritettu kirjallisuuskatsaus tuotti suunnitteluvaihetta varten laajan toteutusvaihtoehtojen kirjon. Lähdeaineistoina käytettiin paikallisen kirjaston kirjojen lisäksi alan lehtien artikkeleita ja varsinkin internetissä julkaistuja artikkeleita. Usean median käytöstä seurasi myös ongelmia, sillä varsinkin verkosta hakukoneiden avulla löytynyt materiaali oli laadultaan vaihtelevaa. Siten vain muutamat aineistoehdokkaat otettiin mukaan tämän työn toteutukseen. Kirjallisuuden osalta paikallinen tarjonta oli varsin suppeaa ja aiheeseen keskittyneitä kirjoja löytyikin vain muutama, joista suurin osa oli vielä noin kaksikymmentä vuotta vanhoja. Siten osa niissä esitetyistä toimintamalleista on jo korvattu uudemmilla ja tehokkaammilla malleilla. Niukkaa aihealueen kirjallisuutta täydennettiin vielä alan kirjallisuuteen lukeutuvilla ohjelmistotuotannon perusteoksilla. Kirjallisuuden ohella suoritettiin laaja alan lehdistöön kohdistunut seulonta, jonka tuloksena löydettiinkin muutamia käyttökelpoisia artikkeleita. Tärkeimmäksi tiedonlähteeksi muodostuivat kuitenkin internetissä julkaistut tieteelliset artikkelit, opinnäytteet ja väitöskirjat.

Ohjelmistokokonaisuuden tuotanto suoritettiin suunnitelman mukaisesti kolmessa työvaiheessa, joskin löytyneiden virheiden korjausta suoritettiin luonnollisesti kaikissa vaiheissa. Komponenttikirjaston toteutusta vaikeutti alussa se, että käytetystä kehitysympäristöstä ei vielä tuolloin ollut saatavilla riittävän uutta versiota. Versioiden välinen vaihdos ei sujunutkaan täysin ongelmitta ja lähdekoodien uudelleenkirjoittamiseen kului huomattavasti aikaa. Komponenttikirjaston valmistuessa aloitettiin varsinaisen käyttöpaneelin suunnittelu ja koeversioiden hahmottelu. Suunnitelmaa täydennettiin jatkuvasti saatujen kokemusten perusteella ja varsinaisen käyttöpaneeliohjelmiston toteutus sujuikin varsin mutkattomasti. Esimerkkisovellukseksi luotavan kuituanalyysiohjelmiston kehitys aloitettiin käyttöpaneelin ensimmäisillä versioilla. Sovellus kehittyi hiljalleen saavuttaen tavoitteensa noin kaksi

kuukautta ennen työn valmistumista. Lopputulosten valossa ohjelmistotyön voidaan katsoa saavuttaneen sille asetetut tavoitteet.

Opinnäytteen kirjallinen dokumentointi aloitettiin heti työn aloitusvaiheessa ja sitä suoritettiin koko työn toteutusajan lähinnä varsinaisesta ohjelmistonkehityksestä vapaiksi jääneinä ajankohtina. Työn alkuvaiheessa pääpaino oli kattavan kirjallisuuskatsauksen työstämisessä kirjalliseen muotoon. Myöhemmin ohjelmistojen valmistuessa pääpaino siirtyi ohjelmiston dokumentointiin ja esittelyyn.

Opinnäytteeseen kuuluvan ohjelmiston jatkokehityksestä ei vielä tässä vaiheessa ole tietoa. Ajan puutteen vuoksi kaikkia ideoita ei ehditty toteuttaa sovelluksiksi asti. Oheisessa luettelossa on kuvattu muutamia jatkokehitysideoita, jotka lisäisivät ohjelmiston käytettävyyttä entisestään.

1. Käyttöpaneelin dynaamisen muodostuksen rajoja ei vielä tutkittu kattavasti. Toinen sovellusohjelmistoista hyödyntää dynaamista käyttöliittymän muodostusta, mutta vain pieneltä osin käytettävissä oleviin palveluihin verrattuna.

2. Graafinen käyttöliittymä olisi mahdollista integroida osaksi NDA-ydinkirjastoa, jolloin käyttöpaneelista tuttuja ikkunoita voitasiin käyttää suoraan NDA-makrotiedostoista.

3. Graasisesta käyttöpaneelista voitaisiin toteuttaa ANSI C-kielellä tekstitilassa toimiva versio, jonka siirrettävyys järjestelmästä toiseen olisi NDA-ydinkirjaston luokkaa.

Kirjallisuutta

[1] Chambers J. M. (2000): *Users, Programmers, and Statistical Data*; ASA Journal of Comp. and Graph. Stat, pp. 404-422; `http://cm.bell-labs.com/stat/doc/jmcJCGS2000.ps` viitattu viimeksi 25.1.2002.

[2] Chambers J. M. (1999): *Computing with Data: Concepts and Challenges*; The American Statistician, pp. 73-84; `http://cm.bell-labs.com/stat/doc/Neyman98.ps` viitattu viimeksi 25.1.2002.

[3] Chambers J. M. and Lang D. T. (1999): *Omegahat – A Component-based Statistical Computing Environment*; Proceedings of the 52nd Session of the ISI, Helsinki, Finland; `http://cm.bell-labs.com/stat/doc/ISI99Omegahat.pdf` viitattu viimeksi 25.1.2002.

[4] Sommerville I. (1996): *Software Engineering*; Addison-Wesley Publishers Ltd.

[5] Pressman R. S. (1997): *Software Engineering - A Practitioner's Approach*; McGraw-Hill Book Company Ltd.

[6] van Vlient H. (2000): *Software Engineering - Principles and Practice*; John Wiley & Sons.

[7] Brown P. J. (1974): *Macro Processores and Techniques for Portable Software*; John Wiley & Sons.

[8] Wallis P. J. L. (1982): *Portable Programming*; The MacMillan Press Ltd.

[9] Gray P. (1991): *Open Systems - A Business Strategy for the 1990s*; McGraw-Hill Book Company Ltd.

[10] The Software Engineering Institute (2001): *Keyword Index*; Carnegie Mellon University;
`http://www.sei.cmu.edu/str/indexes/keywords/` viitattu viimeksi 25.1.2002.

[11] Luck S. (1995): *Machine Abstractions For Software Distribution*; Thesis Defence Presentation Handout.

[12] Mooney J. D. (1997): *Bringing Portability to the Software Process*; University of West Virginia;
`http://www.csee.wvu.edu/~jdm/research/portability/reports/TR_97-1.pdf` viitattu viimeksi 25.1.2002.

[13] Udell J. (1994): *Componentware*; BYTE Magazine, May 1994, pp. 46-56.

[14] Oberndorf T. (1997): *COTS and Open Systems–An Overview*; The Software Engineering Institute, Carnegie Mellon University;
`http://www.sei.cmu.edu/str/descriptions/cots.html` viitattu viimeksi 25.1.2002.

[15] Bergner K., Rausch A. and Sihling M. (1998): *Componentware–The Big Picture*; A position paper presented at the International Workshop on Component-Based Software Engineering;
`http://www.sei.cmu.edu/cbs/icse98/papers/p6.html` viitattu viimeksi 25.1.2002.

[16] Aoyama M. (1998): *New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development*; A position paper presented at the International Workshop on Component-Based Software Engineering;
`http://www.sei.cmu.edu/cbs/icse98/papers/p14.html` viitattu viimeksi 25.1.2002.

[17] Brown A. (1998): *From Component Infrastructure to Component-Based Development*; A position paper presented at the International Workshop on Component-Based Software Engineering;
`http://www.sei.cmu.edu/cbs/icse98/papers/p21.html` viitattu viimeksi 25.1.2002.

[18] Schmidt R. and Assmann U. (1998): *Concepts for Developing Component-Based Systems*; A position paper presented at the International Workshop on Component-Based Software Engineering;

`http://www.sei.cmu.edu/cbs/icse98/papers/p12.html` viitattu viimeksi 25.1.2002.

[19] Sant'Anna M., do Prado Leite J. C. S. and do Prado A. F. (1998): *A Generative Approach to Componentware*; A position paper presented at the International Workshop on Component-Based Software Engineering; `http://www.sei.cmu.edu/cbs/icse98/papers/p11.html` viitattu viimeksi 25.1.2002.

[20] Han J. (1998): *Characterization of Components*; A position paper presented at the International Workshop on Component-Based Software Engineering; `http://www.sei.cmu.edu/cbs/icse98/papers/p4.html` viitattu viimeksi 25.1.2002.

[21] Clements P. and Kereschen J. (1997): *Application Programming Interface*; The Software Engineering Institute, Carnegie Mellon University; `http://www.sei.cmu.edu/str/descriptions/api.html` viitattu viimeksi 25.1.2002.

[22] Nykky M. (1999): *Komponentit ohjelmistotuotannossa*; Pro gradu -tutkielma, Jyväskylän yliopisto.

[23] Brereton P. and Budgen D. (2000): *Component-Based Systems: A Classification of Issues*; IEEE Computer, November 2000, pp. 54-62.

[24] Kruchten P. (1998): *Modeling Component Systems with the Unified Modeling Language*; A position paper presented at the International Workshop on Component-Based Software Engineering; `http://www.sei.cmu.edu/cbs/icse98/papers/p1.html` viitattu viimeksi 25.1.2002.

[25] Clements P., Kogut P. and Tracz W. (1997): *Architecture Description Languages*; The Software Engineering Institute, Carnegie Mellon University; `http://www.sei.cmu.edu/str/descriptions/adl.html` viitattu viimeksi 25.1.2002.

[26] Gerken M. (1997): *Module Interconnection Languages*; The Software Engineering Institute, Carnegie Mellon University; `http://www.sei.cmu.edu/str/descriptions/mil.html` viitattu viimeksi 25.1.2002.

[27] Uehara S. (1998): *Component Architecture for Business Application Systems*; A position paper presented at the International Workshop on Component-Based Software Engineering; `http://www.sei.cmu.edu/cbs/icse98/papers/p17.html` viitattu viimeksi 25.1.2002.

[28] Wallnau K. (1997): *Common Object Request Broker Architecture*; The Software Engineering Institute, Carnegie Mellon University; `http://www.sei.cmu.edu/str/descriptions/corba.html` viitattu viimeksi 25.1.2002.

[29] Bray M. (1997): *Graphical User Interface Builders*; The Software Engineering Institute, Carnegie Mellon University; `http://www.sei.cmu.edu/str/descriptions/guib.html` viitattu viimeksi 25.1.2002.

[30] Haines G., Carney D. and Foreman J. (1997): *Component-Based Software Development / COTS Integration*; The Software Engineering Institute, Carnegie Mellon University; `http://www.sei.cmu.edu/str/descriptions/cbsd.html` viitattu viimeksi 25.1.2002.

[31] Aho A. V., Sethi R., Ullman J. D. (1985): *Compilers : Principles, Techniques, and Tools*; Addison-Wesley Publishers Co.

[32] Muchnick S. S. (1997): *Advanced Compiler Design and Implementation*; Morgan Kaufmann Publishers.

[33] Mak R. (1996): *Writing Compilers and Interpreters*; John Wiley & Sons.

[34] Fraser C. W., Hanson D. R. and Hansen D. (1995): *A Retargetable C Compiler : Design and Implementation*; Addison-Wesley Publishers Co.

[35] Jones D. (1995): *Translating C to Ada*; Knowledge Software Ltd.; `http://www.knosof.co.uk/ctoa.html` viitattu viimeksi 25.1.2002.

[36] Aycock J. (1998): *Converting Python Virtual Machine Code to C*; The Seventh International Python Conference, Department of Computer Science in University of Victoria; `http://www.foretec.com/python/workshops/1998-11/proceedings/papers/aycock-211/aycock211.html` viitattu viimeksi 25.1.2002.

[37] Website (2002): *Runtime Code Generation*; Compilers at the University of Washington; `http://www.cs.washington.edu/research/compiler/rtcg.html` viitattu viimeksi 25.1.2002.

[38] Website (2002): *compilers & interpreters archive*; Department of Computer Science, QMW, London University; `http://www.dcs.qmul.ac.uk/SEL-HPC/Articles/GeneratedHtml/comp.rcg.html` viitattu viimeksi 25.1.2002.

[39] Franz M. S. O. (1994): *Code-Generation On-the-Fly*; PhD thesis, Swiss Federal Institute of Technology Zurich; `ftp://ftp.inf.ethz.ch/doc/diss/th10497.ps.gz` viitattu viimeksi 25.1.2002.

[40] Ousterhout J. K. (1998): *Scripting: Higher Level Programming for the 21st Century*; Computer, March 1998 (Vol. 31, No. 3), pp. 23-30; `http://home.pacbell.net/ouster/scripting.html` viitattu viimeksi 25.1.2002.

[41] Ousterhout J. K. (1998): *Additional Information For Scripting White Paper*; `http://home.pacbell.net/ouster/scriptextra.html` viitattu viimeksi 25.1.2002.

[42] McGlasham B. and Bower A. (1999): *The Interpreter is Dead (slow). Isn't it?*; `http://www.object-arts.com/Papers/TheInterpreterIsDead.PDF` viitattu viimeksi 25.1.2002.

[43] Moretti G. (2001): *Interpreter Implementation*; `http://www-ist.massey.ac.nz/GMoretti/59704/Lectures/Interpreters-TinyBasic.pdf` viitattu viimeksi 25.1.2002.

[44] Website (2002): *Tcl Developer Xchange*; `http://www.tcl-tk.net/` viitattu viimeksi 25.1.2002.

[45] Website (2002): *Perl Mongers - The Perl advocacy people*; `http://www.perl.org/` viitattu viimeksi 25.1.2002.

[46] Website (2002): *Python Language Website*; `http://www.python.org/` viitattu viimeksi 25.1.2002.

[47] Goldberg A. and Robson D. (1983): *Smalltalk-80: The Language and Its Implementation*; Addison-Wesley Publishers Ltd.;

http://users.ipa.net/ dwighth/smalltalk/bluebook/ bluebook_imp_toc.html viitattu viimeksi 25.1.2002.

[48] Comella-Dorda S. and Tilley S. (2000): *Java(TM)*; The Software Engineering Institute, Carnegie Mellon University; http://www.sei.cmu.edu/str/descriptions/java_body.html viitattu viimeksi 25.1.2002.

[49] Sirer E. G., Grimm R., Gregory A. J. and Bershad B. N. (1999): *Design and implementation of a distributed virtual machine for networked computers*; 17th ACM Symposium on Operating System Principles, published as Operating Systems Review 34(5), pp. 202216; http://www.cs.washington.edu/homes/egs/papers/ kimera-sosp99.pdf viitattu viimeksi 25.1.2002.

[50] Website (2002): *Wisconsin Virtual Machine Co-design Group*; http://www.cae.wisc.edu/∼ecearch/strata/ viitattu viimeksi 25.1.2002.

[51] Folliot B. (2000): *The Virtual Virtual Machine Project*; Laboratoire d'Informatique de Paris 6, University Pierre et Marie Curie; http://www-sor.inria.fr/projects/vvm/doc/00SBAC.pdf viitattu viimeksi 25.1.2002.

[52] Winterbottom P. and Pike R. (2002): *The design of the Inferno virtual machine*; Bell Labs, Lucent Technologies; http://www.cs.bell-labs.com/who/rob/hotchips.html viitattu viimeksi 25.1.2002.

[53] Reuben T. (1996): *Tradeoffs in the implementation of the Beetle virtual machine*; http://www.mupsych.org/rrt/timings.pdf viitattu viimeksi 25.1.2002.

[54] Website (2002): *Parallel Virtual Machine*; http://www.epm.ornl.gov/pvm/pvm_home.html viitattu viimeksi 25.1.2002.

[55] Chien A. A., Reed D. and Padua D. (1999): *High Performance Virtual Machines*; http://www-csag.ucsd.edu/projects/hpvm.html viitattu viimeksi 25.1.2002.

[56] Sun Microsystems, Inc. (2002): *The Source for Java Technology*; http://java.sun.com/ viitattu viimeksi 25.1.2002.

[57] Microsoft Corporation (2002): *Microsoft Developer Network*; `http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000451frame=true` viitattu viimeksi 25.1.2002.

[58] Häkkinen E. (2001): *Design, implementation and evaluation of the Neural Data Analysis Environment*; Doctorate Thesis, University of Jyväskylä.

[59] Research group on Software Engineering and Computational Intelligence (2001): *Neural Data Analysis Environment - User's Guide*; University of Jyväskylä.

[60] Häkkinen E. (1999): *Examples about NDA applications*; University of Jyväskylä.

[61] Kaski I. (1998): *Javalla toteutettu ympäristöriippumaton käyttöliittymä - esimerkkinä neuroverkkopohjainen data-analyysiohjelmisto*; Diplomityö, Lappeenrannan Teknillinen Korkeakoulu.

[62] Vlissides M. and Linton M. A. (2002): *A Framework for Building Domain-Specific Graphical Editors*; `http://www.vectaport.com/ivtools/` viitattu viimeksi 25.1.2002.

[63] Perkiö J. (1999): *Tekstuurianalyysiin ja itseorganisaatiokarttaan perustuva painopaperin laaduntarkkailujärjestelmä*; Pro gradu -tutkielma, Jyväskylän yliopisto.

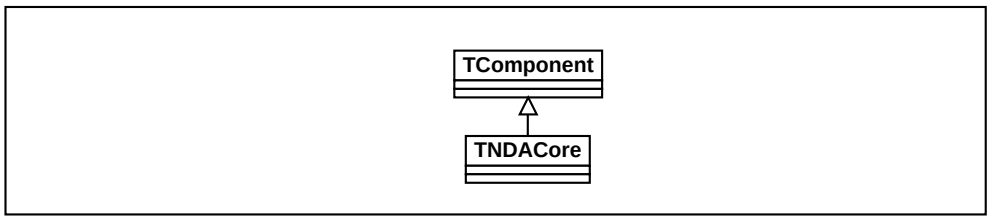
[64] Kohonen T. (2001): *Self-Organizing Maps, Third Edition*; Springer-Verlag.

[65] Koikkalainen P. ja Oja E. (1990): *Self-Organizing Hierarchical Feature Maps*; In Proc. IJCNN-90 1990, IEEE Press, pp. 279-284.

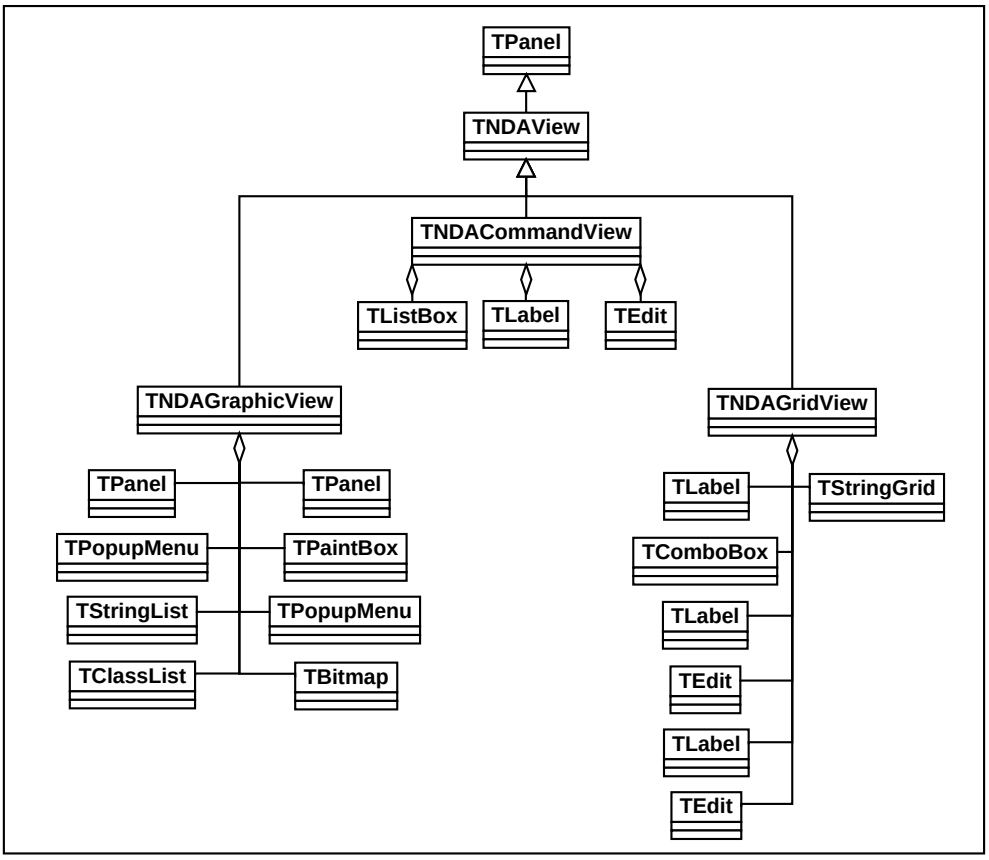
Liite A

Komponenttikirjaston luokkakaaviot

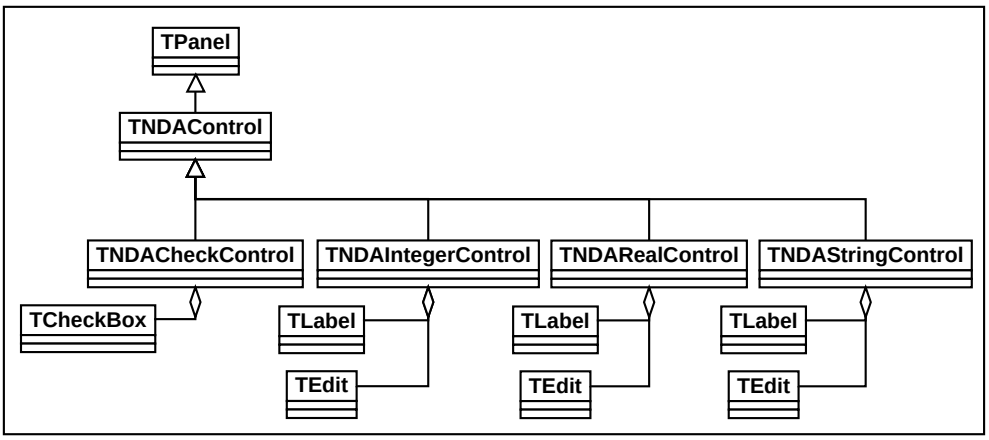
Komponenttikirjaston toteutuksessa käytetyt UML-kuvauskieliset luokkakaaviot.



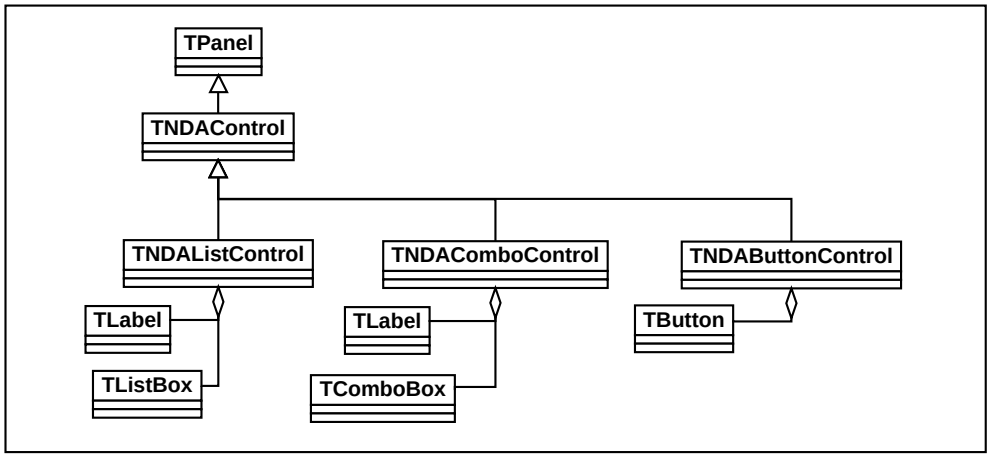
Kuva A.1: UML luokkakaavio - NDA-rajapinta.



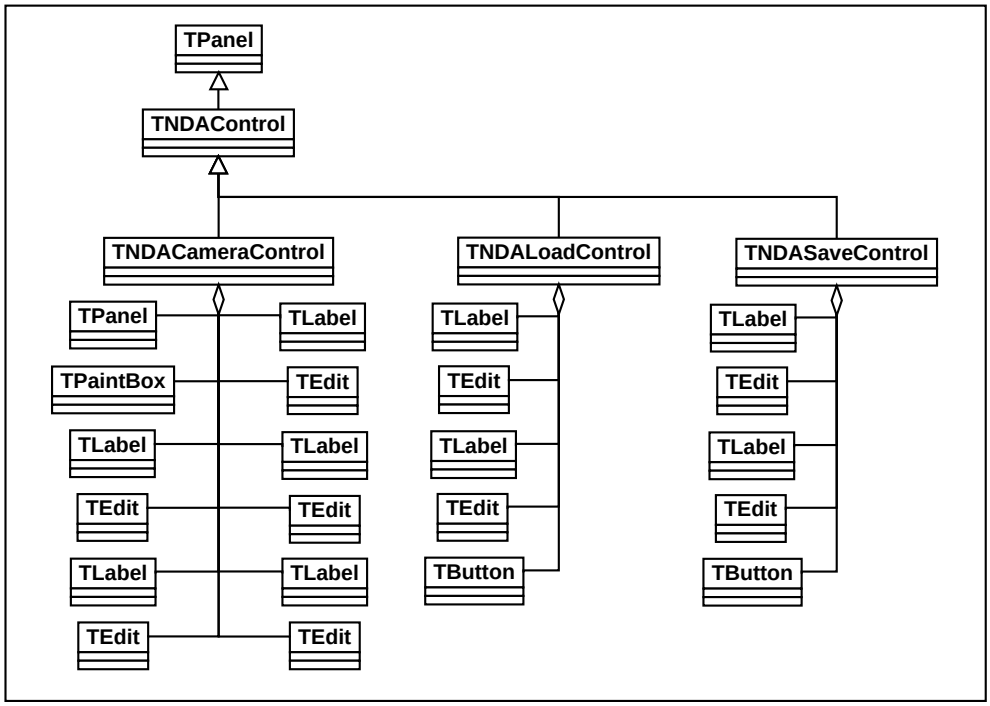
Kuva A.2: UML luokkakaavio - Näkymät.



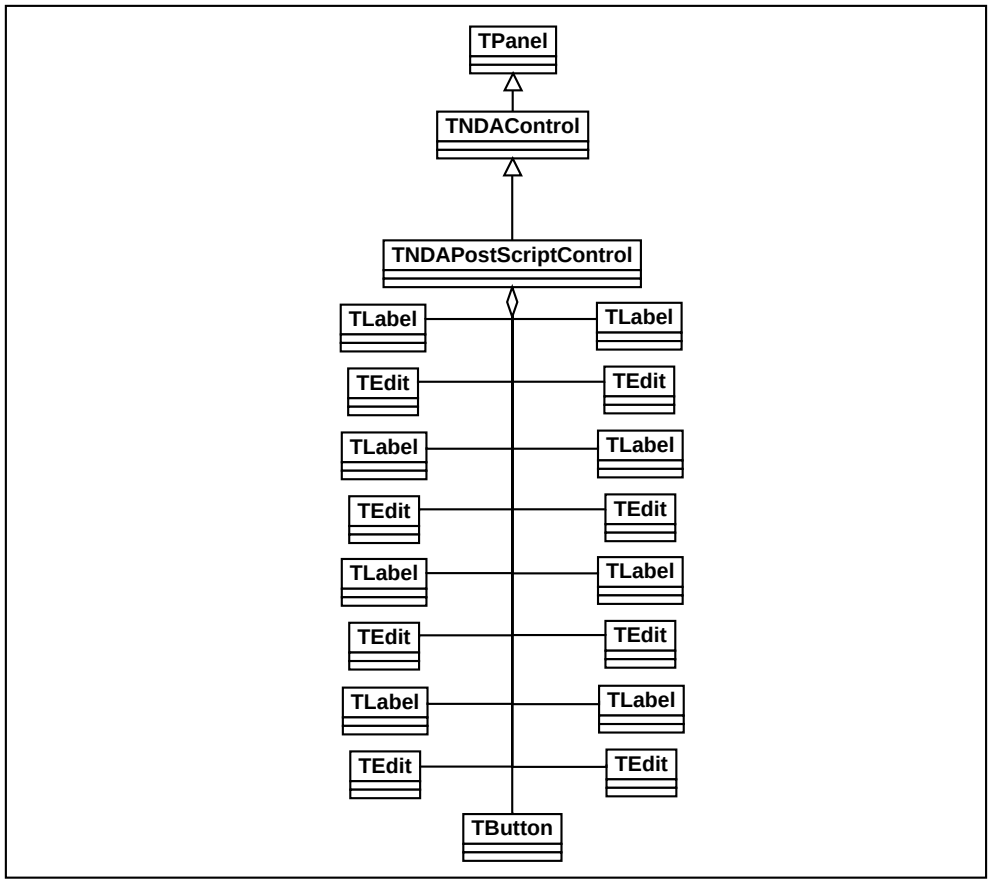
Kuva A.3: UML luokkakaavio - Työkalut 1.



Kuva A.4: UML luokkakaavio - Työkalut 2.



Kuva A.5: UML luokkakaavio - Työkalut 3.

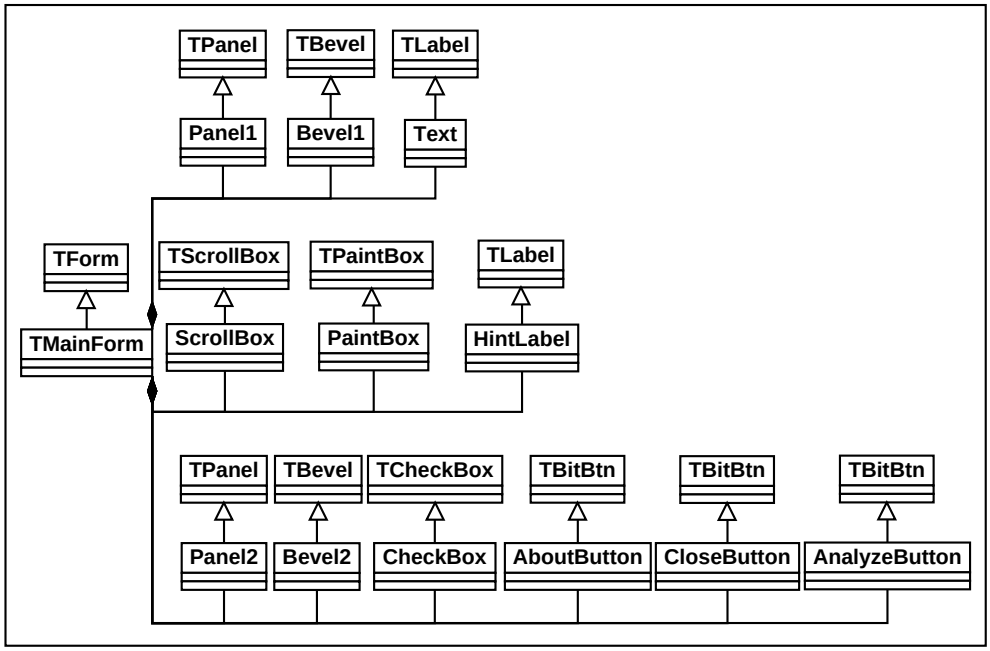


Kuva A.6: UML luokkakaavio - Työkalut 4.

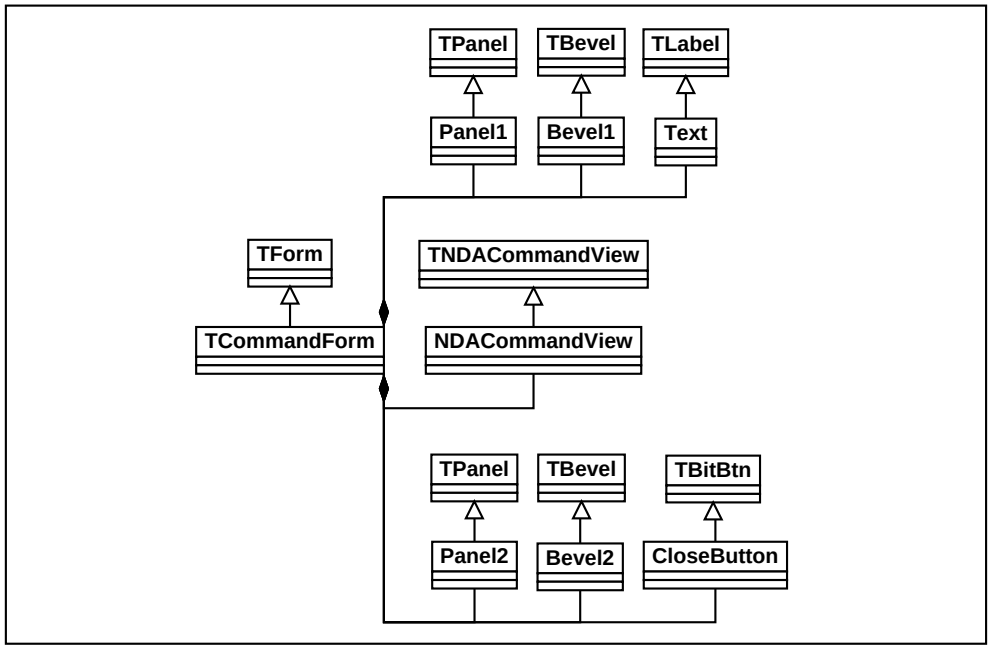
Liite B

Käyttöpaneelin luokkakaaviot

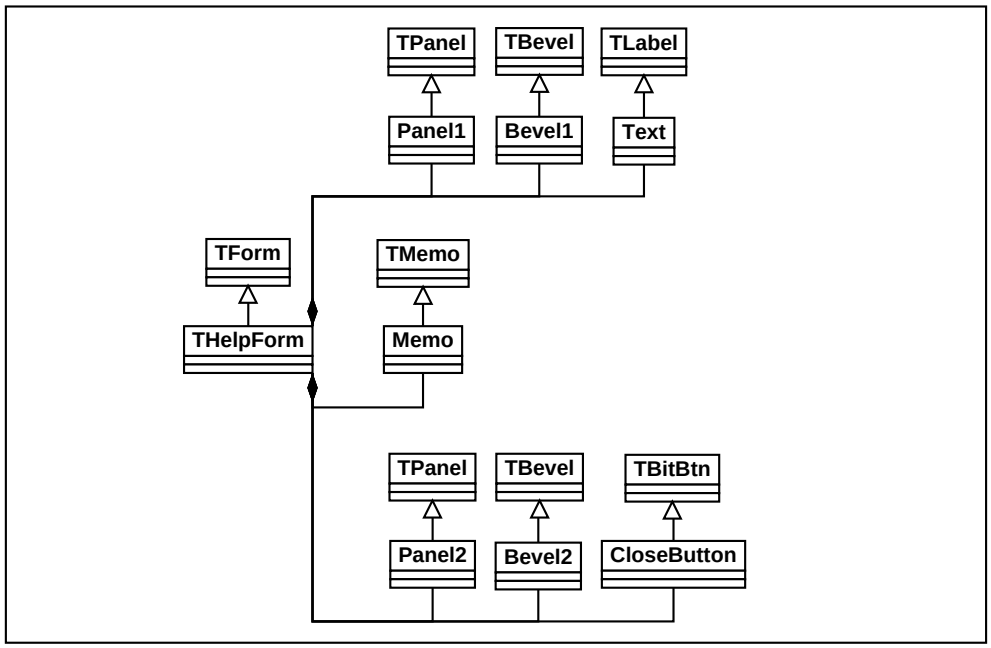
Graafisen käyttöpaneelin toteutuksessa käytetyt UML-kuvauskieliset luokkakaaviot.



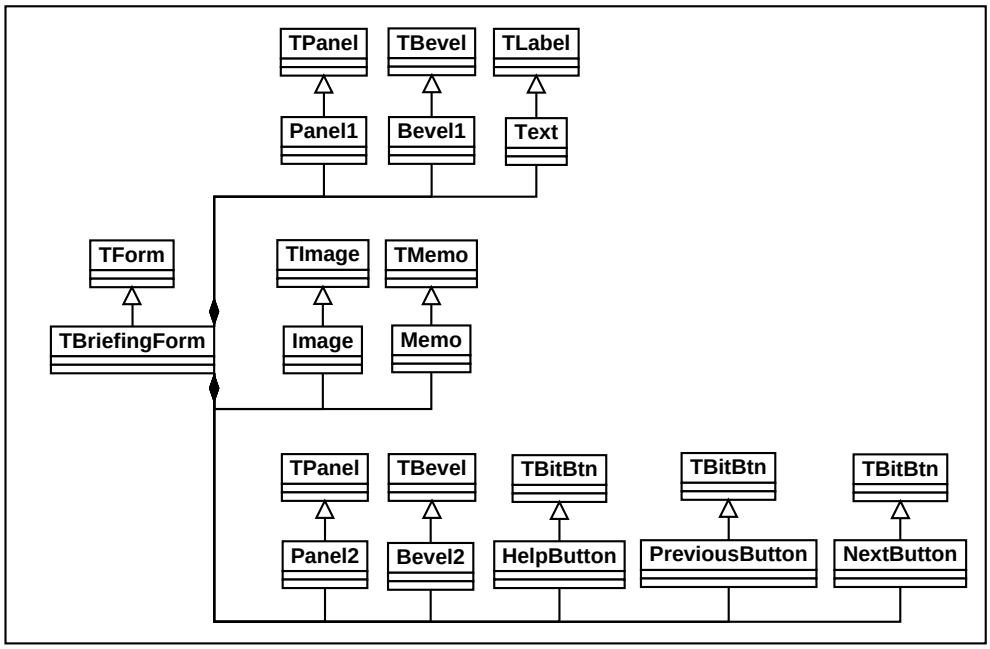
Kuva B.1: UML luokkakaavio - Pääikkuna.



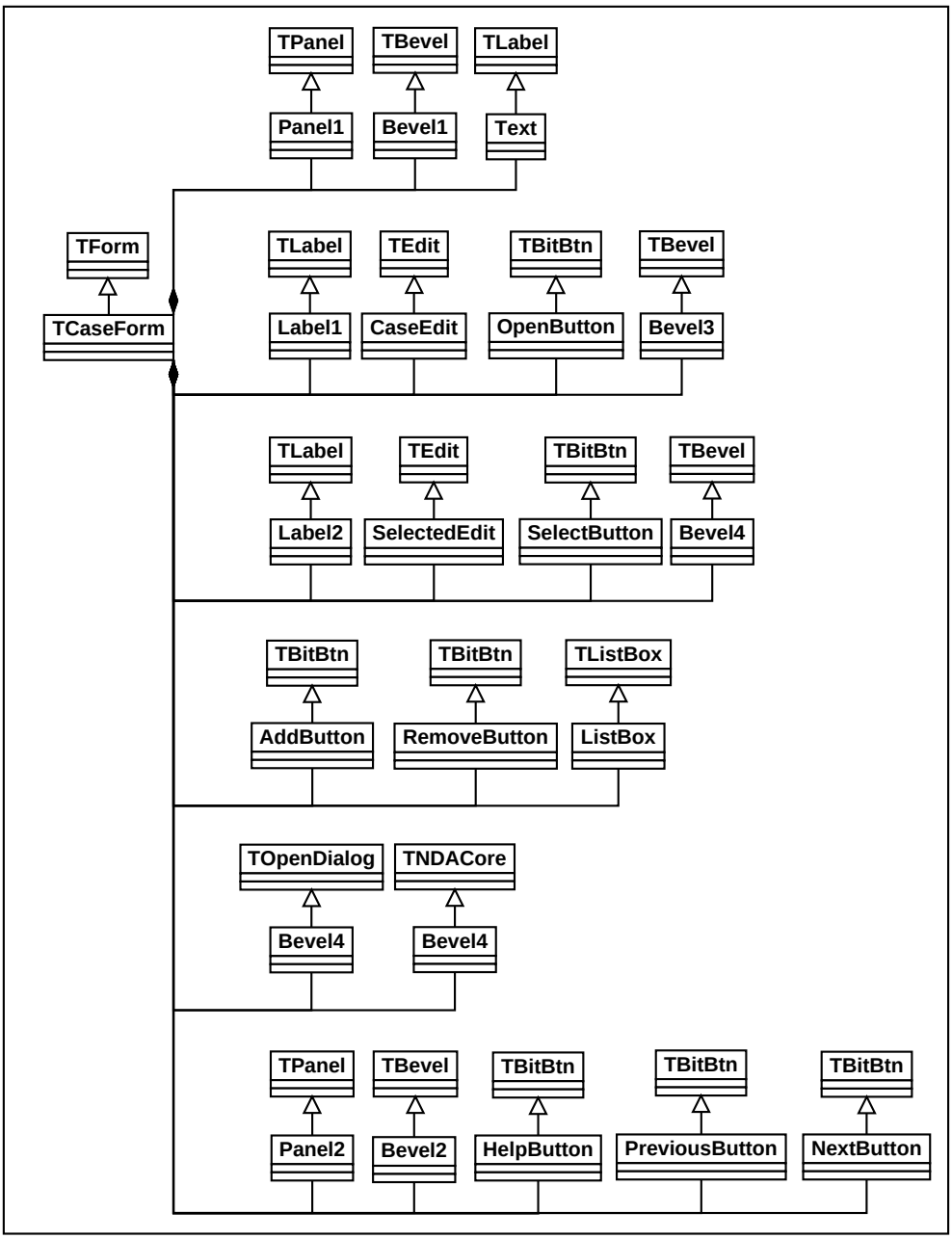
Kuva B.2: UML luokkakaavio - Komentoikkuna.



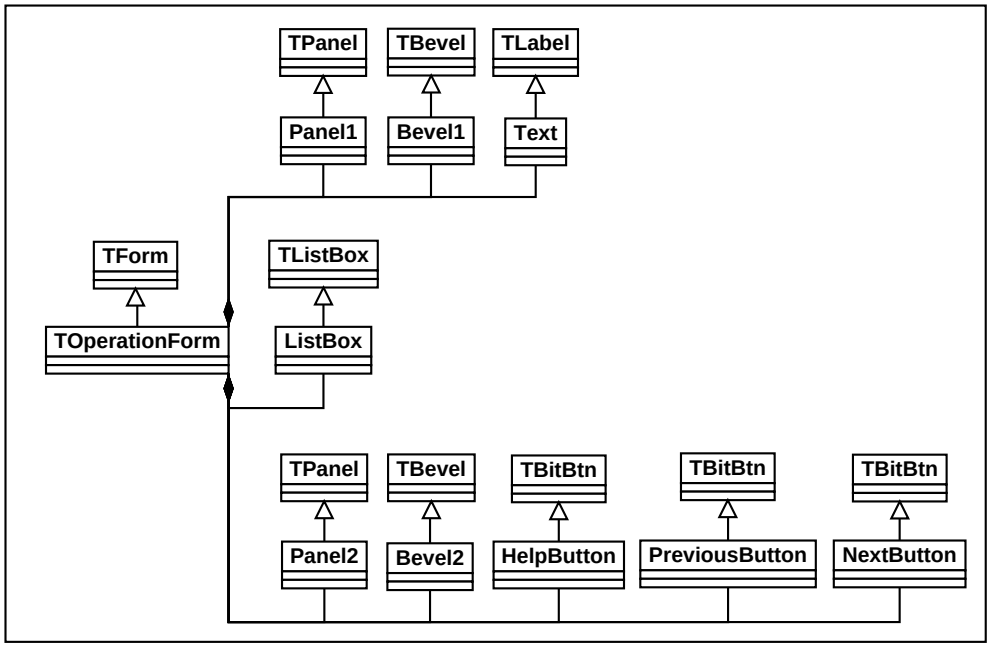
Kuva B.3: UML luokkakaavio - Opasteikkuna.



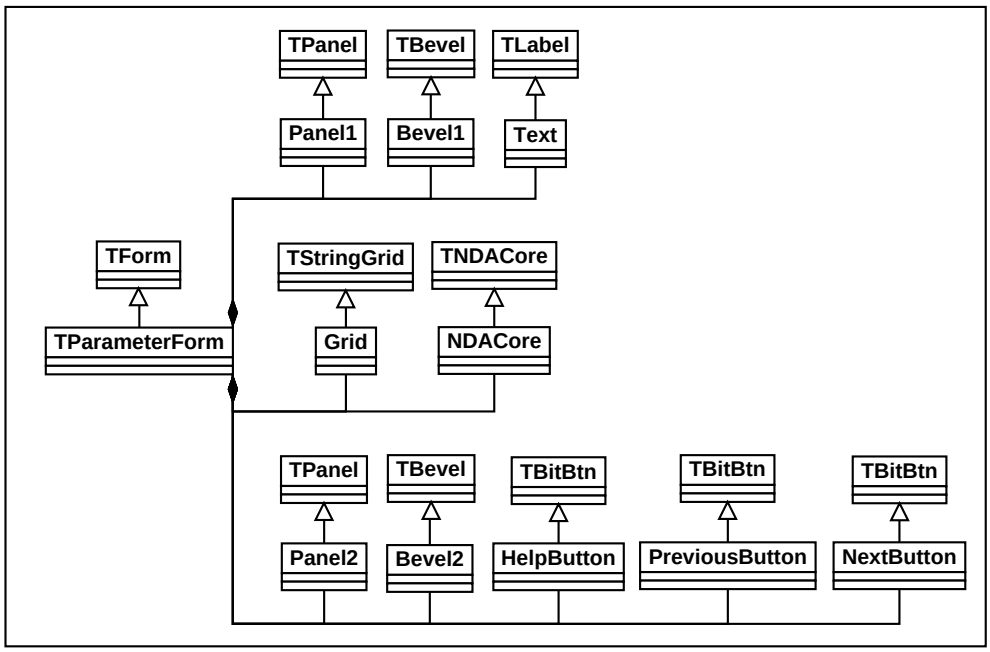
Kuva B.4: UML luokkakaavio - Johdatusikkuna.



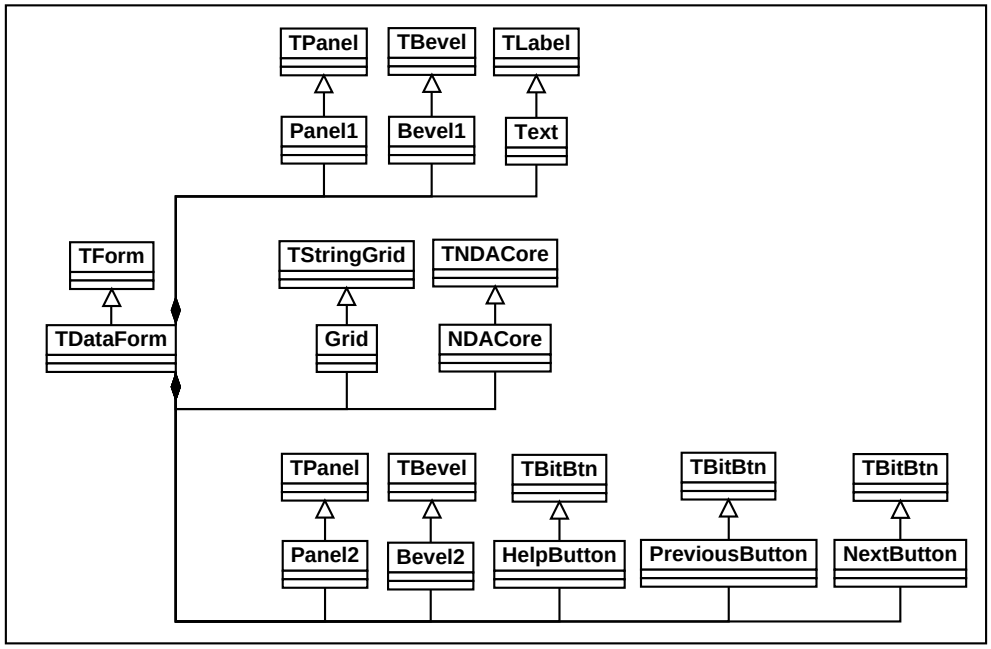
Kuva B.5: UML luokkakaavio - Tiedostoikkuna.



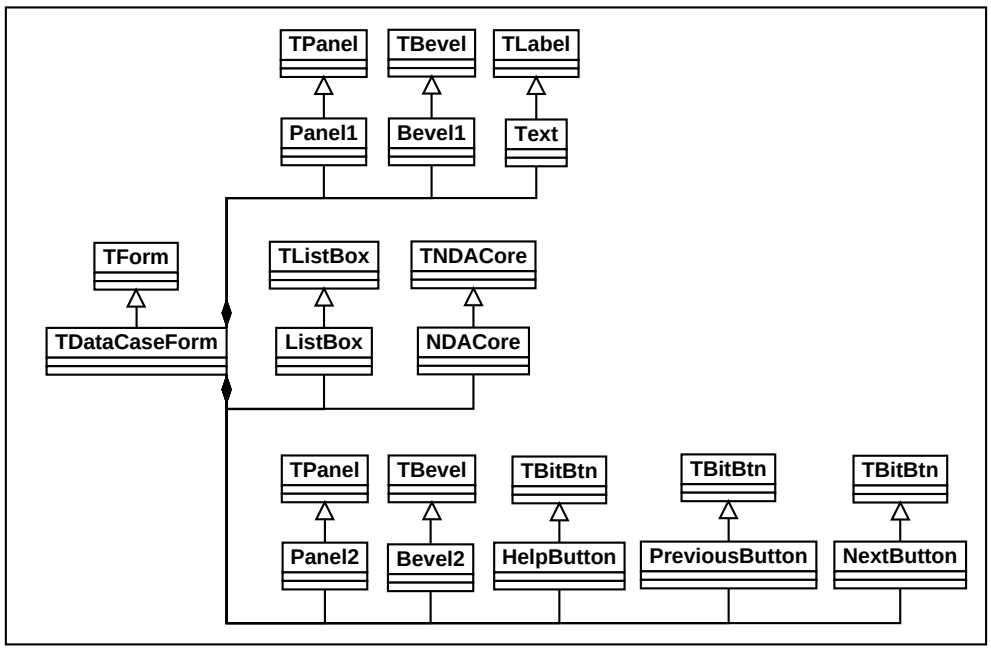
Kuva B.6: UML luokkakaavio - Operaatioikkuna.



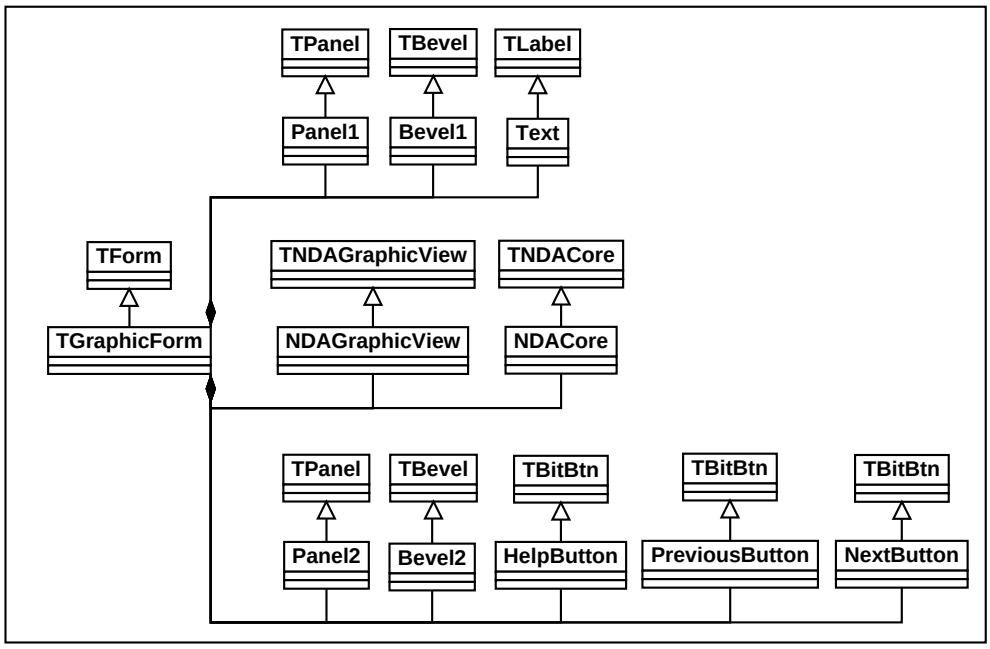
Kuva B.7: UML luokkakaavio - Parametri-ikkuna.



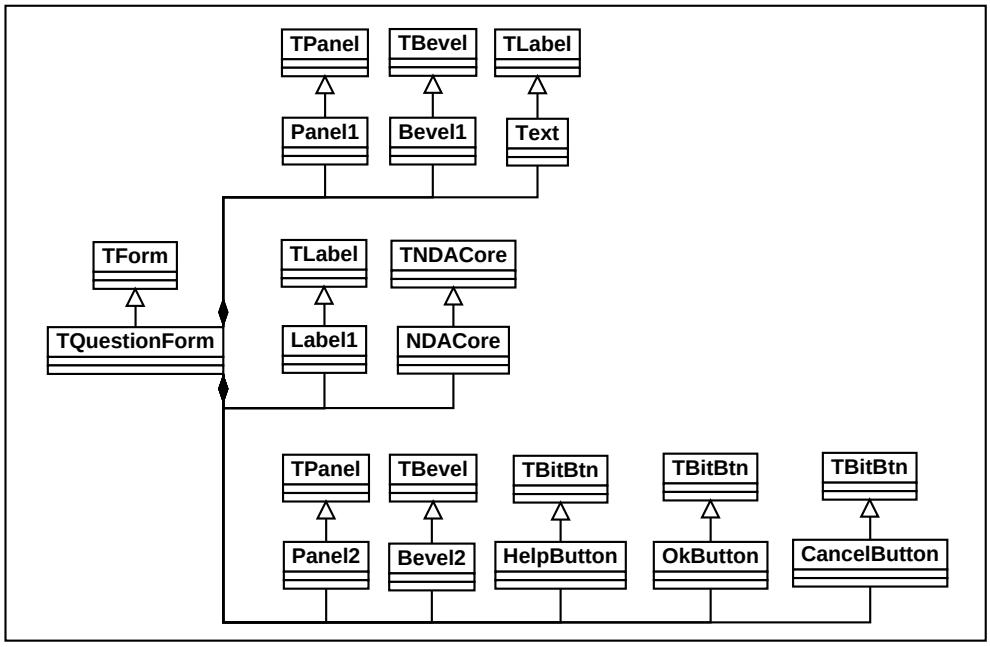
Kuva B.8: UML luokkakaavio - Taulukkoikkuna.



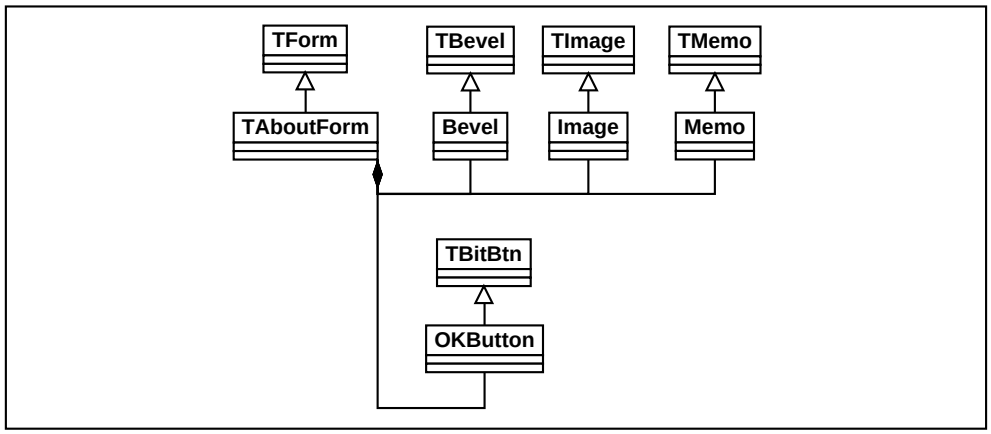
Kuva B.9: UML luokkakaavio - Piirreikkuna.



Kuva B.10: UML luokkakaavio - Grafiikkaikkuna.



Kuva B.11: UML luokkakaavio - Kysymysikkuna.



Kuva B.12: UML luokkakaavio - Ohjelmistontietoikkuna.

Liite C

Esimerkkisovelluksen käyttöliittymän kuvaukset

```
[NDAIDE.ui]
FIA
Fibre Image Analysis


[default.ui]
BARVISIBLE=False
VISIBLEPAGE=Tools
PAGE=Tools
CONTROL=TNDAPostScriptControl
CAPTION=Export PostScript
HINT=Export graphic in PostScript format.


[Briefing.ui]
B.bmp
Welcome to Fibre Image Analysis -tool.


[Files.ui]
C:\NDA\ndaide\Library\test2.case


[FilterGraphic1.ui]
BARVISIBLE=True
VISIBLEPAGE=Filters
PAGE=Filters
CONTROL=TNDAButtonControl
CAPTION=Original image
COMMAND=Filter.cmd
HINT=Restore original image
CONTROL=TNDAComboControl
CAPTION=Method:
VALUE=Average,Median
COMMAND=if -expr "%s"="Average"; -cmd expr -fout Filter0_mode \\
        -expr 1;: -else expr -fout Filter0_mode -expr 0;:
HINT=Select filtering method
CONTROL=TNDAIntegerControl
CAPTION=Radius:
```

```
VALUE=1
COMMAND=expr -fout Filter0_rad -expr %d;
HINT=Filter mask radius in pixels
CONTROL=TNDAButtonControl
CAPTION=Filter image
COMMAND=Filter0.cmd
HINT=Execute filter operation
CONTROL=TNDAIntegerControl
CAPTION=Value:
VALUE=128
COMMAND=expr -fout Filter1_val -expr %d;
HINT=Threshold cutting limit
CONTROL=TNDAComboControl
CAPTION=Correlate:
VALUE=Yes,No
COMMAND=if -expr "%s"="Yes"; -cmd expr -fout Filter1_corr\\
        -expr 1;: -else expr -fout Filter1_corr -expr 0;:
HINT=Use correlation method for thresholding
CONTROL=TNDAButtonControl
CAPTION=Threshold image
COMMAND=Filter1.cmd
HINT=Execute threshold operation
CONTROL=TNDAButtonControl
CAPTION=Negate image
COMMAND=Filter2.cmd
HINT=Execute negation operation
CONTROL=TNDAButtonControl
CAPTION=Equalize image
COMMAND=Filter3.cmd
HINT=Execute histogram equalization operation


[Operation.ui]
Fourier transformation
Co-occurence matrix
Delaunay triangulation
BLOB analysis


[Operation0.ui]
"Fuzzy source","2"
"Fuzzy count","8"
"Fuzzy type","4"


[Operation1.ui]
"Displacement X","2"
"Displacement Y","3"


[Operation2.ui]
"Search radius",5


[Operation3.ui]
"Gray level for thesholding","128"
"Calculated feature","2"
"Threshold","-20"
"t","1"
"s","100"
"W","3.0"
"V","6.0"
```

```
"K","1000"
"Cut at","300"


[Model.ui]
Statistics
Histogram
Class statistics
Class histogram
Fractiles
Mosaic
Classical scaling and Sammon's mapping


[Model0.ui]
"Equalization (min-max based)","0"
"Equalization (variance based)","0"
"Normalization","1"


[Model1.ui]
"Equalization (min-max based)","0"
"Equalization (variance based)","0"
"Normalization","1"


[Model2.ui]
"Equalization (min-max based)","0"
"Equalization (variance based)","0"
"Normalization","1"
"Layer","2"


[Model3.ui]
"Equalization (min-max based)","0"
"Equalization (variance based)","0"
"Normalization","1"
"Layer","2"


[Model4.ui]
"Equalization (min-max based)","0"
"Equalization (variance based)","0"
"Normalization","1"
"Lower bound","25"
"Upper bound","75"


[Model5.ui]
"Equalization (min-max based)","0"
"Equalization (variance based)","0"
"Normalization","1"
"Layer","2"
"Width","256"
"Height","256"
"Frame","1"


[Model6.ui]
"Equalization (min-max based)","0"
"Equalization (variance based)","0"
"Normalization","1"
```

```
"Layer","2"
```

Liite D

Esimerkkisovelluksen sovelluslogiikan kuvaukset

```
##############################################################################
#  Application: Fibre Analysis Tool
#  File: PreFilter.cmd
#  Description: Standard image processing
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
##############################################################################
#
# Load
#
if -exist PreFilter_image -cmd rm PreFilter_image;
loadimg ${FilesSelected[0]} PreFilter_image
if -exist GrayImage -cmd rm GrayImage;
cnvimg -in PreFilter_image -out GrayImage -gray
rm PreFilter_image
#
# Graphics
#
if -exist FilterGraphic1 -cmd rm FilterGraphic1;
mkgrp FilterGraphic1
bgimg FilterGraphic1 -img GrayImage
#
# Create variables
#
expr -fout Filter0_mode -expr 1;
expr -fout Filter0_rad -expr 1;
expr -fout Filter1_val -expr 128;
expr -fout Filter1_corr -expr 1;
##############################################################################
#
# End of file
#
##############################################################################

##############################################################################
#  Application: Fibre Analysis Tool
#  File: Filter.cmd
```

```
#  Description: Standard image processing
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
################################################################################
#  Parameters:
#
################################################################################
#
# Load
#
if -exist Filter_image -cmd rm Filter_image:
loadimg ${FilesSelected[0]} Filter_image
bgimg FilterGraphic1 -rm
rm GrayImage
cnvimg -in Filter_image -out GrayImage -gray
rm Filter_image
bgimg FilterGraphic1 -img GrayImage
show FilterGraphic1
################################################################################
#
# End of file
#
################################################################################


################################################################################
#  Application: Fibre Analysis Tool
#  File: Filter0.cmd
#  Description: Standard image processing
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
################################################################################
#  Parameters:
#
################################################################################
#
# Process
#
if -exist Filter0_image -cmd rm Filter0_image:
if -expr ${Filter0_mode}=1; -cmd filter -in GrayImage -out Filter0_image -avg -rad ${Filter0_rad}: -else filter -in GrayImage \
       -out Filter0_image -med -rad ${Filter0_rad}:
bgimg FilterGraphic1 -rm
```

```
rm GrayImage
ren Filter0_image GrayImage
bgimg FilterGraphic1 -img GrayImage
show FilterGraphic1
##############################################################################
#
# End of file
#
##############################################################################


##############################################################################
#  Application: Fibre Analysis Tool
#  File: Filter1.cmd
#  Description: Standard image processing
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
##############################################################################
#
# Process
#
if -exist Filter1_image -cmd rm Filter1_image:
if -expr ${Filter1_corr}=1; -cmd threshold -in GrayImage -out Filter1_image -val ${Filter1_val} -corr: -else threshold -in GrayImage \
        -out Filter1_image -val ${Filter1_val}:
bgimg FilterGraphic1 -rm
rm GrayImage
ren Filter1_image GrayImage
bgimg FilterGraphic1 -img GrayImage
show FilterGraphic1
##############################################################################
#
# End of file
#
##############################################################################


##############################################################################
#  Application: Fibre Analysis Tool
```

```
#  File: Filter2.cmd
#  Description: Standard image processing
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
################################################################################
#  Parameters:
#
################################################################################
#
# Process
#
if -exist Filter2_image -cmd rm Filter2_image:
negative -in GrayImage -out Filter2_image
bgimg FilterGraphic1 -rm
rm GrayImage
ren Filter2_image GrayImage
bgimg FilterGraphic1 -img GrayImage
show FilterGraphic1
################################################################################
#
# End of file
#
################################################################################


################################################################################
#  Application: Fibre Analysis Tool
#  File: Filter4.cmd
#  Description: Standard image processing
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
################################################################################
#  Parameters:
#
################################################################################
#
# Process
#
if -exist Filter4_image -cmd rm Filter4_image:
histoeq -in GrayImage -out Filter4_image
bgimg FilterGraphic1 -rm
rm GrayImage
```

```
ren Filter4_image GrayImage
bgimg FilterGraphic1 -img GrayImage
show FilterGraphic1
##############################################################################
#
# End of file
#
##############################################################################


##############################################################################
#  Application: Fibre Analysis Tool
#  File: PostFilter.cmd
#  Description: Standard image processing
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
##############################################################################
#
# Save
#
if -exist PostFilter_image -cmd rm PostFilter_image:
cnvimg -in GrayImage -out PostFilter_image -indc
saveimg PostFilter_image Selected.gif
rm PostFilter_image
#
# Destroy variables
#
if -exist Filter0_avg -cmd rm Filter0_avg:
if -exist Filter0_med -cmd rm Filter0_med:
if -exist Filter0_rad -cmd rm Filter0_rad:
if -exist Filter1_val -cmd rm Filter1_val:
if -exist Filter1_corr -cmd rm Filter1_corr:
##############################################################################
#
# End of file
#
##############################################################################
```

```
##############################################################################
#  Application: Fibre Analysis Tool
#  File: Operation0.cmd
#  Description: Fourier transformation operation
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
#  0) Fuzzy data source
#  1) Fuzzy class count
#  2) Fuzzy type
##############################################################################
##############################################################################
#
# File macro (index, count, file, selectedfile, filepath, fuzzysource, fuzzycount, fuzzytype)
#
##############################################################################
BeginMacro Operation0_Foreach
if -exist Operation0_flag1 -cmd rm Operation0_flag1:
if -exist Operation0_flag2 -cmd rm Operation0_flag2:
if -file $5_features0_$6_$7_$8.dat -cmd expr -fout Operation0_flag1 -expr 1;: -else expr -fout Operation0_flag1 -expr 0;:
if -expr "$3"!="$4"; -cmd expr -fout Operation0_flag2 -expr 1;: -else expr -fout Operation0_flag2 -expr 0;:
if -expr ('Operation0_flag1'=1) AND ('Operation0_flag2'=1); -cmd Begin Operation0_Results $1 $2 $3 $4 $5 $6 $7 $8
##############################################################################
# We have existing results
##############################################################################
if -exist Operation0_feature -cmd rm Operation0_feature:
load $5_features0_$6_$7_$8.dat -n Operation0_feature
if -exist Features -cmd attchdat -d Operation0_feature -dout Features: -else copy Operation0_feature Features:
##############################################################################
# End
##############################################################################
EndBegin: -else Begin Operation0_NoResults $1 $2 $3 $4 $5 $6 $7 $8
##############################################################################
# We do not have existing results
##############################################################################
# Load
if -exist Operation0_image -cmd rm Operation0_image:
if -expr "$3"="$4"; -cmd loadimg Selected.gif Operation0_image: -else loadimg $3 Operation0_image:
```

```
if -exist Operation0_grayimage -cmd rm Operation0_grayimage:
cnvimg -in Operation0_image -out Operation0_grayimage -gray
rm Operation0_image
# Process
if -exist  Operation0_idim -cmd rm  Operation0_idim:
idim -in Operation0_grayimage -out Operation0_idim
if -exist Operation0_matrix -cmd rm Operation0_matrix:
convmatrix -in Operation0_grayimage -out Operation0_matrix
if -exist Operation0_balancedmatrix -cmd rm Operation0_balancedmatrix:
balance -in Operation0_matrix.real -out Operation0_balancedmatrix.real
rm Operation0_matrix
if -exist Operation0_fft -cmd rm Operation0_fft:
fft2 -in Operation0_balancedmatrix -out Operation0_fft -all
# Insert features to list
if -exist Operation0_fz -cmd rm Operation0_fz:
mkfz Operation0_fz
setdata -f Operation0_fz.ranges.min -vals 0=0.0;
setdata -f Operation0_fz.ranges.max -vals 0=1.0;
if -exist i -cmd rm i:
expr -expr 0; -fout i
while -expr 'i' < $7; -cmd Begin Operation0_Fuzzy $8
addfzs Operation0_fz -fs f${i} -t $1
EndBegin: -loop expr -fout i -expr 'i'+1;:
rm i
initfz Operation0_fz -lock
if -exist Operation0_shift -cmd rm Operation0_shift:
shiftspec -in Operation0_fft.spec -out Operation0_shift
if -exist Operation0_fftrad -cmd rm Operation0_fftrad:
freqvec -in Operation0_shift -out Operation0_fftrad -meth 0
if -exist Operation0_fftangle -cmd rm Operation0_fftangle:
freqvec -in Operation0_shift -out Operation0_fftangle -meth 1
if -exist Operation0_fftxxr -cmd rm Operation0_fftxxr:
freqvec -in Operation0_fft.spec -out Operation0_fftxxr -meth 0
if -exist Operation0_fftxxa -cmd rm Operation0_fftxxa:
freqvec -in Operation0_fft.spec -out Operation0_fftxxa -meth 1
if -exist Operation0_featurefield -cmd rm Operation0_featurefield:
if -expr $6=0; -cmd feature -fz Operation0_fz -in Operation0_fftrad -out Operation0_featurefield:
if -expr $6=1; -cmd feature -fz Operation0_fz -in Operation0_fftangle -out Operation0_featurefield:
if -expr $6=2; -cmd feature -fz Operation0_fz -in Operation0_fftxxr -out Operation0_featurefield:
if -expr $6=3; -cmd feature -fz Operation0_fz -in Operation0_fftxxa -out Operation0_featurefield:
```

```
if -exist Operation0_feature -cmd rm Operation0_feature:
select Operation0_feature -f Operation0_featurefield
if -exist Operation0_transposedfeature -cmd rm Operation0_transposedfeature:
transpose -d Operation0_feature -dout Operation0_transposedfeature
if -exist Features -cmd attchdat -d Operation0_transposedfeature -dout Features: -else copy Operation0_transposedfeature Features:
save Operation0_transposedfeature -o $5_features0_$6_$7_$8.dat -t tab
# Graphics
if -expr "$3"!="$4"; -cmd return:
if -exist OperationGraphic1 -cmd rm OperationGraphic1:
mkgrp OperationGraphic1
if -exist Operation0_fftimage -cmd rm Operation0_fftimage:
visualfft -in Operation0_fft -out Operation0_fftimage -width ${Operation0_idim.width} -height ${Operation0_idim.height} -scale -shift
bgimg OperationGraphic1 -img Operation0_fftimage
if -exist FFT -cmd rm FFT:
mkgrp FFT
if -exist Operation0_shift -cmd rm Operation0_shift:
shiftspec -in Operation0_fft.spec -out Operation0_shift
if -exist Operation0_fftrad -cmd rm Operation0_fftrad:
freqvec -in Operation0_shift -out Operation0_fftrad -meth 0
if -exist Operation0_fftangle -cmd rm Operation0_fftangle:
freqvec -in Operation0_shift -out Operation0_fftangle -meth 1
if -exist Operation0_fftxxr -cmd rm Operation0_fftxxr:
freqvec -in Operation0_fft.spec -out Operation0_fftxxr -meth 0
if -exist Operation0_fftxxa -cmd rm Operation0_fftxxa:
freqvec -in Operation0_fft.spec -out Operation0_fftxxa -meth 1
dis FFT all
en FFT ldgr saxis
ldgrv FFT -inx 0 -f Operation0_fftrad -co red
ldgrv FFT -inx 1 -f Operation0_fftangle -co blue
ldgrv FFT -inx 2 -f Operation0_fftxxr -co green
ldgrv FFT -inx 3 -f Operation0_fftxxa -co cyan
if -exist Operation0_freqstat -cmd rm Operation0_freqstat:
freqstat -in Operation0_balancedmatrix.real -out Operation0_freqstat
if -exist Fuzzy -cmd rm Fuzzy:
mkfz Fuzzy -f Operation0_freqstat
setdata -f Fuzzy.ranges.min -vals 0=0.0;
setdata -f Fuzzy.ranges.max -vals 0=1.0;
if -exist i -cmd rm i:
expr -expr 0; -fout i
while -expr 'i' < $7; -cmd Begin Operation0_Fuzzy2 $8
```

```
addfzs Fuzzy -fs f${i} -t $1
func1 Fuzzy.grp -f Fuzzy.sets.f${i} -co blue
EndBegin: -loop expr -fout i -expr 'i'+1;:
rm i
initfz Fuzzy -lock
dis Fuzzy.grp all
en Fuzzy.grp ldgr saxis
ldgrv Fuzzy.grp -inx 0 -f Operation0_freqstat -co red
viewp Fuzzy.grp -dz 120
if -exist OperationGraphic2 -cmd rm OperationGraphic2:
mkgf -gf OperationGraphic2
addgf -gf OperationGraphic2 -g FFT
addgf -gf OperationGraphic2 -g Fuzzy.grp
setgfplc -gf OperationGraphic2 -g FFT -x0 0 -y0 0.05 -x1 1 -y1 0.45
setgfplc -gf OperationGraphic2 -g Fuzzy.grp -x0 0 -y0 0.55 -x1 1 -y1 1.0
##############################################################################
# End
##############################################################################
EndBegin:
##############################################################################
# End Macro
##############################################################################
EndMacro
##############################################################################
#
# Foreach in Files
#
##############################################################################
if -exist Features -cmd rm Features:
if -exist Operation0_index -cmd rm Operation0_index:
expr -expr 0; -fout Operation0_index
if -exist Operation0_filecount -cmd rm Operation0_filecount:
len -n Files -fout Operation0_filecount
while -expr 'Operation0_index' < 'Operation0_filecount'; -cmd \Operation0_Foreach ${Operation0_index} ${Operation0_filecount} \
        ${Files[${Operation0_index}]} ${FilesSelected[0]} ${FilesPaths[${Operation0_index}]} ${OperationParameters[0]} \
        ${OperationParameters[1]} ${OperationParameters[2]}\: -loop expr -fout Operation0_index -expr 'Operation0_index'+1;:
##############################################################################
#
# End of file
#
```

```
##############################################################################


##############################################################################
#  Application: Fibre Analysis Tool
#  File: Operation1.cmd
#  Description: Co-occurence matrix operation
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
#  0) Displacement x
#  1) Displacement y
##############################################################################

##############################################################################
#
# File macro (index, count, file, selectedfile, filepath, dx, dy)
#
##############################################################################
BeginMacro Operation1_Foreach
if -exist Operation1_flag1 -cmd rm Operation1_flag1:
if -exist Operation1_flag2 -cmd rm Operation1_flag2:
if -file $5_features1.dat -cmd expr -fout Operation1_flag1 -expr 1;: -else expr -fout Operation1_flag1 -expr 0;:
if -expr "$3"!="$4"; -cmd expr -fout Operation1_flag2 -expr 1;: -else expr -fout Operation1_flag2 -expr 0;:
if -expr ('Operation1_flag1'=1) AND ('Operation1_flag2'=1); -cmd Begin Operation1_Results $1 $2 $3 $4 $5 $6 $7
##############################################################################
# We have existing results
##############################################################################
if -exist Operation1_feature -cmd rm Operation1_feature:
load $5_features1.dat -n Operation1_feature
if -exist Features -cmd attchdat -d Operation1_feature -dout Features: -else copy Operation1_feature Features:
##############################################################################
# End
##############################################################################
EndBegin: -else Begin Operation1_NoResults $1 $2 $3 $4 $5 $6 $7
##############################################################################
# We do not have existing results
##############################################################################
# Load
```

```
if -exist Operation1_image -cmd rm Operation1_image:
if -expr "$3"="$4"; -cmd loadimg Selected.gif Operation1_image: -else loadimg $3 Operation1_image:
if -exist Operation1_grayimage -cmd rm Operation1_grayimage:
cnvimg -in Operation1_image -out Operation1_grayimage -gray
rm Operation1_image
# Process
if -exist Operation1_comatrix -cmd rm Operation1_comatrix:
comatrix -in Operation1_grayimage -out Operation1_comatrix -dx $6 -dy $7
if -exist Operation1_comatriximage -cmd rm Operation1_comatriximage:
visualfft -in Operation1_comatrix -out Operation1_comatriximage -width 256 -height 256 -scale
# Insert features to list
if -exist Operation1_feature -cmd rm Operation1_feature:
if -exist contrast -cmd rm contrast:
contrast -in Operation1_comatrix -out contrast
select Operation1_feature -f contrast
rm contrast
if -exist correlation -cmd rm correlation:
correlation -in Operation1_comatrix -out correlation
select Operation1_feature -f correlation
rm correlation
if -exist energy -cmd rm energy:
energy -in Operation1_comatrix -out energy
select Operation1_feature -f energy
rm energy
if -exist entropy -cmd rm entropy:
entropy -in Operation1_comatrix -out entropy
select Operation1_feature -f entropy
rm entropy
if -exist mmax -cmd rm mmax:
mmax -in Operation1_comatrix -out mmax
select Operation1_feature -f mmax
rm mmax
if -exist idm -cmd rm idm:
idm -in Operation1_comatrix -out idm
select Operation1_feature -f idm
rm idm
if -exist Features -cmd attchdat -d Operation1_feature -dout Features: -else copy Operation1_feature Features:
save Operation1_feature -o $5_features1.dat -t tab
# Graphics
if -expr "$3"!="$4"; -cmd return:
```

```
if -exist OperationGraphic1 -cmd rm OperationGraphic1:
mkgrp OperationGraphic1
bgimg OperationGraphic1 -img Operation1_comatriximage
##############################################################################
# End
##############################################################################
EndBegin:
##############################################################################
# End Macro
##############################################################################
EndMacro
##############################################################################
#
# Foreach in Files
#
##############################################################################
if -exist Features -cmd rm Features:
if -exist Operation1_index -cmd rm Operation1_index:
expr -expr 0; -fout Operation1_index
if -exist Operation1_filecount -cmd rm Operation1_filecount:
len -n Files -fout Operation1_filecount
while -expr 'Operation1_index' < 'Operation1_filecount'; -cmd \Operation1_Foreach ${Operation1_index} ${Operation1_filecount} \
        ${Files[${Operation1_index}]} ${FilesSelected[0]} ${FilesPaths[${Operation1_index}]} ${OperationParameters[0]} \
        ${OperationParameters[1]}\: -loop expr -fout Operation1_index -expr 'Operation1_index'+1;:
##############################################################################
#
# Graphics
#
##############################################################################
if -exist Contrast -cmd rm Contrast:
mkgrp Contrast
addsg Contrast
hst Contrast -d Features.contrast -co red -inx 0
if -exist Correlation -cmd rm Correlation:
mkgrp Correlation
addsg Correlation
hst Correlation -d Features.correlation -co green -inx 0
if -exist Energy -cmd rm Energy:
mkgrp Energy
addsg Energy
```

```
hst Energy -d Features.energy -co blue -inx 0
if -exist Entropy -cmd rm Entropy:
mkgrp Entropy
addsg Entropy
hst Entropy -d Features.entropy -co brown -inx 0
if -exist MMax -cmd rm MMax:
mkgrp MMax
addsg MMax
hst MMax -d Features.mmax -co orange -inx 0
if -exist IDM -cmd rm IDM:
mkgrp IDM
addsg IDM
hst IDM -d Features.idm -co cyan -inx 0
if -exist OperationGraphic2 -cmd rm OperationGraphic2:
mkgf -gf OperationGraphic2
addgf -gf OperationGraphic2 -g Contrast
addgf -gf OperationGraphic2 -g Correlation
addgf -gf OperationGraphic2 -g Energy
addgf -gf OperationGraphic2 -g Entropy
addgf -gf OperationGraphic2 -g MMax
addgf -gf OperationGraphic2 -g IDM
setgfplc -gf OperationGraphic2 -g Contrast -x0 0 -y0 0.05 -x1 1 -y1 0.2
setgfplc -gf OperationGraphic2 -g Correlation -x0 0 -y0 0.2 -x1 1 -y1 0.35
setgfplc -gf OperationGraphic2 -g Energy -x0 0 -y0 0.35 -x1 1 -y1 0.5
setgfplc -gf OperationGraphic2 -g Entropy -x0 0 -y0 0.5 -x1 1 -y1 0.65
setgfplc -gf OperationGraphic2 -g MMax -x0 0 -y0 0.65 -x1 1 -y1 0.8
setgfplc -gf OperationGraphic2 -g IDM -x0 0 -y0 0.8 -x1 1 -y1 0.95
##############################################################################
#
# End of file
#
##############################################################################


##############################################################################
#  Application: Fibre Analysis Tool
#  File: Operation2.cmd
#  Description: Delaunay triangulation operation
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
```

```
#  Parameters:
#
#  0) Radius for triangulation
################################################################################
################################################################################
#
# File macro (index, count, file, selectedfile, filepath, radius)
#
################################################################################
BeginMacro Operation2_Foreach
if -exist Operation2_flag1 -cmd rm Operation2_flag1:
if -exist Operation2_flag2 -cmd rm Operation2_flag2:
if -file $5_features2.dat -cmd expr -fout Operation2_flag1 -expr 1;: -else expr -fout Operation2_flag1 -expr 0;:
if -expr "$3"!="$4"; -cmd expr -fout Operation2_flag2 -expr 1;: -else expr -fout Operation2_flag2 -expr 0;:
if -expr ('Operation2_flag1'=1) AND ('Operation2_flag2'=1); -cmd Begin Operation2_Results $1 $2 $3 $4 $5 $6
################################################################################
# We have existing results
################################################################################
if -exist Operation2_feature -cmd rm Operation2_feature:
load $5_features2.dat -n Operation2_feature
if -exist Features -cmd attchdat -d Operation2_feature -dout Features: -else copy Operation2_feature Features:
################################################################################
# End
################################################################################
EndBegin: -else Begin Operation2_NoResults $1 $2 $3 $4 $5 $6
################################################################################
# We do not have existing results
################################################################################
# Load
if -exist Operation2_image -cmd rm Operation2_image:
if -expr "$3"="$4"; -cmd loadimg Selected.gif Operation2_image: -else loadimg $3 Operation2_image:
if -exist Operation2_grayimage -cmd rm Operation2_grayimage:
cnvimg -in Operation2_image -out Operation2_grayimage -gray
rm Operation2_image
# Process
if -exist Operation2_nodes -cmd rm Operation2_nodes:
findtb -in Operation2_grayimage -dout Operation2_nodes -size $6
if -exist Operation2_arcs -cmd rm Operation2_arcs:
delaunay -vdata Operation2_nodes -dout Operation2_arcs
if -exist Operation2_cutarcs -cmd rm Operation2_cutarcs:
```

```
cutarcs -vdata Operation2_nodes -edata Operation2_arcs -dout Operation2_cutarcs
if -exist Operation2_graph -cmd rm Operation2_graph:
gr2fr -vdata Operation2_nodes -edata Operation2_cutarcs -dout Operation2_graph
if -exist Operation2_image -cmd rm Operation2_image:
f2i -d Operation2_graph -iout Operation2_image -bgimg Operation2_grayimage
if -exist Operation2_distr1 -cmd rm Operation2_distr1:
if -exist Operation2_feature1 -cmd rm Operation2_feature1:
arclength -vdata Operation2_nodes -edata Operation2_arcs -fout Operation2_distr1 -dout Operation2_feature1 -val -3
if -exist Operation2_distr2 -cmd rm Operation2_distr2:
if -exist Operation2_feature2 -cmd rm Operation2_feature2:
arclength -vdata Operation2_nodes -edata Operation2_arcs -fout Operation2_distr2 -dout Operation2_feature2 -val -2
if -exist Operation2_distr3 -cmd rm Operation2_distr3:
if -exist Operation2_feature3 -cmd rm Operation2_feature3:
arclength -vdata Operation2_nodes -edata Operation2_arcs -fout Operation2_distr3 -dout Operation2_feature3 -val -1
# Insert features to list and save it
if -exist Features -cmd attchdat -d Operation2_feature1 -dout Features: -else copy Operation2_feature1 Features:
save Operation2_feature1 -o $5_features2.dat -t tab
# Graphics
if -expr "$3"!="$4"; -cmd return:
if -exist OperationGraphic1 -cmd rm OperationGraphic1:
mkgrp OperationGraphic1
bgimg OperationGraphic1 -img Operation2_image
if -exist OperationGraphic2 -cmd rm OperationGraphic2:
mkgrp OperationGraphic2
ldgrv OperationGraphic2 -inx 0 -f Operation2_distr3 -co blue
ldgrv OperationGraphic2 -inx 1 -f Operation2_distr2 -co green
ldgrv OperationGraphic2 -inx 2 -f Operation2_distr1 -co red
##############################################################################
# End
##############################################################################
EndBegin:
##############################################################################
# End Macro
##############################################################################
EndMacro
##############################################################################
#
# Foreach in Files
#
##############################################################################
```

```
if -exist Features -cmd rm Features:
if -exist Operation2_index -cmd rm Operation2_index:
expr -expr 0; -fout Operation2_index
if -exist Operation2_filecount -cmd rm Operation2_filecount:
len -n Files -fout Operation2_filecount
while -expr 'Operation2_index' < 'Operation2_filecount'; -cmd \Operation2_Foreach ${Operation2_index} ${Operation2_filecount} \
        ${Files[${Operation2_index}]} ${FilesSelected[0]} ${FilesPaths[${Operation2_index}]} ${OperationParameters[0]}\: \
        -loop expr -fout Operation2_index -expr 'Operation2_index'+1;:
###############################################################################
#
# End of file
#
###############################################################################


###############################################################################
#  Application: Fibre Analysis Tool
#  File: Operation3.cmd
#  Description: BLOB analysis operation
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
###############################################################################
#  Parameters:
#
#  0) Graylevel
#  1) Inspected feature
#  2) Thresholding value
###############################################################################
###############################################################################
#
# File macro (index, count, file, selectedfile, filepath, graylevel, feature, threshold)
#
###############################################################################
BeginMacro Operation3_Foreach
if -exist Operation3_flag1 -cmd rm Operation3_flag1:
if -exist Operation3_flag2 -cmd rm Operation3_flag2:
if -file $5_distr3.dat -cmd expr -fout Operation3_flag1 -expr 1;: -else expr -fout Operation3_flag1 -expr 0;:
if -file $5_features3.dat -cmd expr -fout Operation3_flag2 -expr 1;: -else expr -fout Operation3_flag2 -expr 0;:
if -expr "$3"!="$4"; -cmd expr -fout Operation3_flag3 -expr 1;: -else expr -fout Operation3_flag3 -expr 0;:
if -expr ('Operation3_flag1'=1) AND ('Operation3_flag2'=1) AND ('Operation3_flag3'=1); -cmd Begin Operation3_Results \
        $1 $2 $3 $4 $5 $6 $7 $8
```

```
##########################################################################
# We have existing results
##########################################################################
if -exist Operation3_distr$1 -cmd rm Operation3_distr$1:
load $5_distr3.dat -n Operation3_distr$1
if -exist Operation3_feature -cmd rm Operation3_feature:
load $5_features3.dat -n Operation3_feature
if -exist Features -cmd attchdat -d Operation3_feature -dout Features: -else copy Operation3_feature Features:
##########################################################################
# End
##########################################################################
EndBegin: -else Begin Operation3_NoResults $1 $2 $3 $4 $5 $6 $7 $8
##########################################################################
# We do not have existing results
##########################################################################
# Load
if -exist Operation3_image -cmd rm Operation3_image:
if -expr "$3"="$4"; -cmd loadimg Selected.gif Operation3_image: -else loadimg $3 Operation3_image:
if -exist Operation3_grayimage -cmd rm Operation3_grayimage:
cnvimg -in Operation3_image -out Operation3_grayimage -gray
rm Operation3_image
# Process
if -exist Operation3_equalizeimage -cmd rm Operation3_equalizeimage:
histoeq -in Operation3_grayimage -out Operation3_equalizeimage
rm Operation3_grayimage
if -exist Operation3_negativeimage -cmd rm Operation3_negativeimage:
negative -in Operation3_equalizeimage -out Operation3_negativeimage
rm Operation3_equalizeimage
if -exist Operation3_distr$1 -cmd rm Operation3_distr$1:
#uad -in Operation3_negativeimage -out Operation3_distr$1 -val $6 -feat $7
gen -U 0.1 9.9 -dout Operation3_distr$1 -s 2000
save Operation3_distr$1 -o $5_distr3.dat -t tab
if -exist Operation3_feature -cmd rm Operation3_feature:
imgfeat -in Operation3_negativeimage -out Operation3_feature -val $8
rm Operation3_negativeimage
# Insert features to list and save it
if -exist Features -cmd attchdat -d Operation3_feature -dout Features: -else copy Operation3_feature Features:
save Operation3_feature -o $5_features3.dat -t tab
# Graphics
##########################################################################
```

```
# End
##############################################################################
EndBegin:
##############################################################################
# End Macro
##############################################################################
EndMacro
##############################################################################
#
# Foreach in Files
#
##############################################################################
if -exist Features -cmd rm Features:
if -exist Operation3_index -cmd rm Operation3_index:
expr -expr 0; -fout Operation3_index
if -exist Operation3_filecount -cmd rm Operation3_filecount:
len -n Files -fout Operation3_filecount
while -expr 'Operation3_index' < 'Operation3_filecount'; -cmd \Operation3_Foreach ${Operation3_index} ${Operation3_filecount} \
        ${Files[${Operation3_index}]} ${FilesSelected[0]} ${FilesPaths[${Operation3_index}]} ${OperationParameters[0]} \
        ${OperationParameters[1]} ${OperationParameters[2]}\: -loop expr -fout Operation3_index -expr 'Operation3_index'+1;:
##############################################################################
#
# Graphics
#
##############################################################################
##############################################################################
# Make all
##############################################################################
if -exist all -cmd rm all:
if -exist Operation3_index -cmd rm Operation3_index:
expr -expr 0; -fout Operation3_index
if -exist Operation3_filecount -cmd rm Operation3_filecount:
len -n Files -fout Operation3_filecount
while -expr 'Operation3_index' < 'Operation3_filecount'; -cmd Begin Operation3_Loop1
if -exist all -cmd attchdat -d Operation3_distr${Operation3_index} -dout all: -else copy Operation3_distr${Operation3_index} all:
EndBegin: -loop expr -fout Operation3_index -expr 'Operation3_index'+1;:
##############################################################################
# Find min and max
##############################################################################
if -exist st1 -cmd rm st1:
```

```
fldstat -d all -dout st1 -min -max
##########################################################################################
# Smooth
##########################################################################################
if -exist Operation3_index -cmd rm Operation3_index:
expr -expr 0; -fout Operation3_index
if -exist Operation3_filecount -cmd rm Operation3_filecount:
len -n Files -fout Operation3_filecount
while -expr 'Operation3_index' < 'Operation3_filecount'; -cmd Begin Operation3_Loop2
if -exist f_${Operation3_index} -cmd rm f_${Operation3_index}:
smooth -fin Operation3_distr${Operation3_index}.FO -t ${OperationParameters[3]} -s ${OperationParameters[4]} \
       -W ${OperationParameters[5]} -V ${OperationParameters[6]} -K ${OperationParameters[7]} -a ${st1.min} \
       -b ${st1.max} -dout f_${Operation3_index}
    EndBegin: -loop expr -fout Operation3_index -expr 'Operation3_index'+1;:
##########################################################################################
# Calculate f_avg
##########################################################################################
if -exist f_avg -cmd rm f_avg:
expr -fout f_avg -expr ('f_0.fx' - 'f_0.fx');
if -exist Operation3_index -cmd rm Operation3_index:
expr -expr 0; -fout Operation3_index
if -exist Operation3_filecount -cmd rm Operation3_filecount:
len -n Files -fout Operation3_filecount
while -expr 'Operation3_index' < 'Operation3_filecount'; -cmd Begin Operation3_Loop3
expr -fout f_avg -expr ('f_avg' + 'f_${Operation3_index}.fx');
EndBegin: -loop expr -fout Operation3_index -expr 'Operation3_index'+1;:
expr -fout f_avg -expr ('f_avg' / ${Operation3_filecount});
##########################################################################################
# Calculate differentials
##########################################################################################
if -exist D -cmd rm D:
if -exist Operation3_index -cmd rm Operation3_index:
expr -expr 0; -fout Operation3_index
if -exist Operation3_filecount -cmd rm Operation3_filecount:
len -n Files -fout Operation3_filecount
while -expr 'Operation3_index' < 'Operation3_filecount'; -cmd Begin Operation3_Loop4
if -exist df_${Operation3_index} -cmd rm df_${Operation3_index}:
expr -dout D -fout df_${Operation3_index} -expr ('f_${Operation3_index}.fx' - 'f_avg') * 100.0;
EndBegin: -loop expr -fout Operation3_index -expr 'Operation3_index'+1;:
expr -dout D -fout zero -expr ('f_avg' - 'f_avg');
```

```
##############################################################################
# Select data for display (cut!)
##############################################################################
if -exist D1 -cmd rm D1:
if -exist D2 -cmd rm D2:
selrec -d f_0 -expr 'f_0.x' < ${OperationParameters[6]}; -dout D1
selrec -d D -expr 'f_0.x' < ${OperationParameters[6]}; -dout D2
select D2 -f D1.x
##############################################################################
# Calculate scales
##############################################################################
if -exist ABC -cmd rm ABC:
if -exist Operation3_index -cmd rm Operation3_index:
expr -expr 0; -fout Operation3_index
if -exist Operation3_filecount -cmd rm Operation3_filecount:
len -n Files -fout Operation3_filecount
while -expr 'Operation3_index' < 'Operation3_filecount'; -cmd Begin Operation3_Loop5
select ABC -f D2.df_${Operation3_index}
EndBegin: -loop expr -fout Operation3_index -expr 'Operation3_index'+1;:
if -exist st2 -cmd rm st2:
fldstat -d ABC -dout st2 -min -max
if -exist st3 -cmd rm st3:
select st3 -n st2.min
if -exist st4 -cmd rm st4:
select st4 -n st2.max
if -exist st5 -cmd rm st5:
fldstat -d st3 -dout st5 -min
if -exist st6 -cmd rm st6:
fldstat -d st4 -dout st6 -max
if -exist sa -cmd rm sa:
expr -fout sa -expr 'st5.min' * 1.3;
if -exist sb -cmd rm sb:
expr -fout sb -expr 'st6.max' * 1.3;
##############################################################################
# Load colors
##############################################################################
if -exist colors -cmd rm colors:
load colors -n colors
##############################################################################
# Graphics 1
```

```
##############################################################################
if -exist OperationGraphic1 -cmd rm OperationGraphic1:
mkgrp OperationGraphic1
#dis OperationGraphic1 all
#en OperationGraphic1 ldgr saxis
viewp OperationGraphic1 -dz 150 -xy 0 -yz 0
#saxis OperationGraphic1 -ax x -f f_${Operation3_index}.x -inx 0 -step 8 -sca data
if -exist Operation3_index -cmd rm Operation3_index:
expr -expr 0; -fout Operation3_index
if -exist Operation3_filecount -cmd rm Operation3_filecount:
len -n Files -fout Operation3_filecount
while -expr 'Operation3_index' < 'Operation3_filecount'; -cmd Begin Operation3_Loop6
#saxis OperationGraphic1 -ax y -f f_${Operation3_index}.fx -inx ${Operation3_index} -step 8 -sca data
ldgrv OperationGraphic1 -inx ${Operation3_index} -f f_${Operation3_index}.fx -co ${colors.name[${Operation3_index}]}
EndBegin: -loop expr -fout Operation3_index -expr 'Operation3_index'+1;:
#fval OperationGraphic1 -f colors.name -len 8
##############################################################################
# Graphics 2
##############################################################################
if -exist OperationGraphic2 -cmd rm OperationGraphic2:
mkgrp OperationGraphic2
#dis OperationGraphic2 all
#en OperationGraphic2 ldgr saxis
viewp OperationGraphic2 -dz 150 -xy 0 -yz 0
#saxis OperationGraphic2 -ax x -f D2.x -inx 0 -step 8 -sca data
#saxis OperationGraphic2 -ax y -f D2.df_0 -inx 0 -step 8 -sca abs -min ${sa} -max ${sb}
if -exist Operation3_index -cmd rm Operation3_index:
expr -expr 0; -fout Operation3_index
if -exist Operation3_filecount -cmd rm Operation3_filecount:
len -n Files -fout Operation3_filecount
while -expr 'Operation3_index' < 'Operation3_filecount'; -cmd Begin Operation3_Loop7
ldgrv OperationGraphic2 -inx 0 -f D2.df_${Operation3_index} -co ${colors.name[${Operation3_index}]} -sca abs -min ${sa} -max ${sb}
EndBegin: -loop expr -fout Operation3_index -expr 'Operation3_index'+1;:
ldgrv OperationGraphic2 -inx 0 -f D2.zero -co black -sca abs -min ${sa} -max ${sb}
#fval OperationGraphic2 -f colors.name -len 8
##############################################################################
#
# End of file
#
##############################################################################
```

```
##############################################################################
#  Application: Fibre Analysis Tool
#  File: Model0.cmd
#  Description: Statistical analysis
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
#  0) Equalization (min-max based)
#  1) Equalization (variance based)
#  2) Normalization
##############################################################################
##############################################################################
#
# Preprocess
#
##############################################################################
if -exist Model0_features -cmd rm Model0_features:
if -expr ${ModelParameters[0]}!=0; -cmd prepro -d Features -dout Model0_features -e:
if -exist Model0_features -cmd rm Features: -cmd2 ren Model0_features Features:
if -exist Model0_features -cmd rm Model0_features:
if -expr ${ModelParameters[1]}!=0; -cmd prepro -d Features -dout Model0_features -edev:
if -exist Model0_features -cmd rm Features: -cmd2 ren Model0_features Features:
if -exist Model0_features -cmd rm Model0_features:
if -expr ${ModelParameters[2]}!=0; -cmd prepro -d Features -dout Model0_features -n:
if -exist Model0_features -cmd rm Features: -cmd2 ren Model0_features Features:
##############################################################################
#
# Process
#
##############################################################################
##############################################################################
#
# Graphics
#
##############################################################################
fldstat -d FeaturesSelected -dout Model0_stat -min -max -sum -avg -var -dev
if -exist Minimum -cmd rm Minimum:
```

```
mkgrp Minimum
addsg Minimum
hst Minimum -d Model0_stat.min -co red -inx 0
if -exist Maximum -cmd rm Maximum:
mkgrp Maximum
addsg Maximum
hst Maximum -d Model0_stat.max -co green -inx 0
if -exist Sum -cmd rm Sum:
mkgrp Sum
addsg Sum
hst Sum -d Model0_stat.sum -co blue -inx 0
if -exist Average -cmd rm Average:
mkgrp Average
addsg Average
hst Average -d Model0_stat.avg -co brown -inx 0
if -exist Variance -cmd rm Variance:
mkgrp Variance
addsg Variance
hst Variance -d Model0_stat.var -co orange -inx 0
if -exist Deviation -cmd rm Deviation:
mkgrp Deviation
addsg Deviation
hst Deviation -d Model0_stat.dev -co cyan -inx 0
if -exist ModelGraphic1 -cmd rm ModelGraphic1:
mkgf -gf ModelGraphic1
addgf -gf ModelGraphic1 -g Minimum
addgf -gf ModelGraphic1 -g Maximum
addgf -gf ModelGraphic1 -g Sum
addgf -gf ModelGraphic1 -g Average
addgf -gf ModelGraphic1 -g Variance
addgf -gf ModelGraphic1 -g Deviation
setgfplc -gf ModelGraphic1 -g Minimum -x0 0 -y0 0.05 -x1 1 -y1 0.2
setgfplc -gf ModelGraphic1 -g Maximum -x0 0 -y0 0.2 -x1 1 -y1 0.35
setgfplc -gf ModelGraphic1 -g Sum -x0 0 -y0 0.35 -x1 1 -y1 0.5
setgfplc -gf ModelGraphic1 -g Average -x0 0 -y0 0.5 -x1 1 -y1 0.65
setgfplc -gf ModelGraphic1 -g Variance -x0 0 -y0 0.65 -x1 1 -y1 0.8
setgfplc -gf ModelGraphic1 -g Deviation -x0 0 -y0 0.8 -x1 1 -y1 0.95
if -exist ModelGraphic2 -cmd rm ModelGraphic2:
mkgrp ModelGraphic2
##########################################################################################
```

```
#
# End of file
#
################################################################################


################################################################################
#  Application: Fibre Analysis Tool
#  File: Model1.cmd
#  Description: Histogram
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
################################################################################
#  Parameters:
#
#  0) Equalization (min-max based)
#  1) Equalization (variance based)
#  2) Normalization
################################################################################
################################################################################
#
# Preprocess
#
################################################################################
if -exist Model1_features -cmd rm Model1_features:
if -expr ${ModelParameters[0]}!=0; -cmd prepro -d Features -dout Model1_features -e:
if -exist Model1_features -cmd rm Features: -cmd2 ren Model1_features Features:
if -exist Model1_features -cmd rm Model1_features:
if -expr ${ModelParameters[1]}!=0; -cmd prepro -d Features -dout Model1_features -edev:
if -exist Model1_features -cmd rm Features: -cmd2 ren Model1_features Features:
if -exist Model1_features -cmd rm Model1_features:
if -expr ${ModelParameters[2]}!=0; -cmd prepro -d Features -dout Model1_features -n:
if -exist Model1_features -cmd rm Features: -cmd2 ren Model1_features Features:
################################################################################
#
# Process
#
################################################################################
################################################################################
#
# Graphics
```

```
#
##############################################################################
if -exist ModelGraphic1 -cmd rm ModelGraphic1:
mkgf -gf ModelGraphic1
if -exist Model1_index -cmd rm Model1_index:
expr -expr 0; -fout Model1_index
if -exist Model1_count -cmd rm Model1_count:
len -n Features -fout Model1_count
while -expr 'Model1_index' < 'Model1_count'; -cmd Begin Model1_Loop
if -exist ${Features[${Model1_index}]} -cmd rm ${Features[${Model1_index}]}:
mkgrp ${Features[${Model1_index}]}
addsg ${Features[${Model1_index}]}
hst ${Features[${Model1_index}]} -inx 0 -d Features.${Features[${Model1_index}]} -co red
addgf -gf ModelGraphic1 -g ${Features[${Model1_index}]}
if -exist Model1_y1 -cmd rm Model1_y1:
expr -fout Model1_y1 -expr (1.0 / (${Model1_count}.0+1.0)) * (${Model1_index}.0 + 1.0);
if -exist Model1_y2 -cmd rm Model1_y2:
expr -fout Model1_y2 -expr (1.0 / (${Model1_count}.0+1)) * (${Model1_index}.0 + 2.0);
setgfplc -gf ModelGraphic1 -g ${Features[${Model1_index}]} -x0 0 -y0 ${Model1_y1} -x1 1 -y1 ${Model1_y2}
EndBegin: -loop expr -fout Model1_index -expr 'Model1_index'+1;:
if -exist ModelGraphic2 -cmd rm ModelGraphic2:
mkgrp ModelGraphic2
##############################################################################
#
# End of file
#
##############################################################################


##############################################################################
#  Application: Fibre Analysis Tool
#  File: Model2.cmd
#  Description: Class statistics
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
#  0) Equalization (min-max based)
#  1) Equalization (variance based)
#  2) Normalization
```

```
#  3) Layer
##############################################################################
##############################################################################
#
# Preprocess
#
##############################################################################
if -exist Model2_features -cmd rm Model2_features:
if -expr ${ModelParameters[0]}!=0; -cmd prepro -d Features -dout Model2_features -e:
if -exist Model2_features -cmd rm Features: -cmd2 ren Model2_features Features:
if -exist Model2_features -cmd rm Model2_features:
if -expr ${ModelParameters[1]}!=0; -cmd prepro -d Features -dout Model2_features -edev:
if -exist Model2_features -cmd rm Features: -cmd2 ren Model2_features Features:
if -exist Model2_features -cmd rm Model2_features:
if -expr ${ModelParameters[2]}!=0; -cmd prepro -d Features -dout Model2_features -n:
if -exist Model2_features -cmd rm Features: -cmd2 ren Model2_features Features:
##############################################################################
#
# Process
#
##############################################################################
if -exist Model2_layer -cmd rm Model2_layer:
expr -fout Model2_layer -expr ${ModelParameters[3]}+1;
if -exist Model2_som -cmd rm Model2_som:
somtr -d FeaturesSelected -sout Model2_som -l ${Model2_layer}
if -exist Model2_cls -cmd rm Model2_cls:
somcl -s Model2_som -d FeaturesSelected -cout Model2_cls
##############################################################################
#
# Graphics
#
##############################################################################
if -exist Model2_stat -cmd rm Model2_stat:
clstat -c Model2_cls -d Features -dout Model2_stat -all
if -exist ModelGraphic1 -cmd rm ModelGraphic1:
mkgf -gf ModelGraphic1
if -exist Model2_index -cmd rm Model2_index:
expr -expr 0; -fout Model2_index
if -exist Model2_count -cmd rm Model2_count:
len -n Model2_stat -fout Model2_count
```

```
while -expr 'Model2_index' < 'Model2_count'; -cmd Begin Model2_Loop
if -exist ${Model2_stat[${Model2_index}]} -cmd rm ${Model2_stat[${Model2_index}]}:
mkgrp ${Model2_stat[${Model2_index}]}
addsg ${Model2_stat[${Model2_index}]}
if -exist Model2_type -cmd rm Model2_type:
getfieldtype -d Model2_stat -i ${Model2_index} -o Model2_type
if -expr ${Model2_type}!=2; -cmd hst ${Model2_stat[${Model2_index}]} -inx 0 -d Model2_stat.${Model2_stat[${Model2_index}]} -co red:
addgf -gf ModelGraphic1 -g ${Model2_stat[${Model2_index}]}
if -exist Model2_y1 -cmd rm Model2_y1:
expr -fout Model2_y1 -expr (1.0 / (${Model2_count}.0+1.0)) * (${Model2_index}.0 + 1.0);
if -exist Model2_y2 -cmd rm Model2_y2:
expr -fout Model2_y2 -expr (1.0 / (${Model2_count}.0+1)) * (${Model2_index}.0 + 2.0);
setgfplc -gf ModelGraphic1 -g ${Model2_stat[${Model2_index}]} -x0 0 -y0 ${Model2_y1} -x1 1 -y1 ${Model2_y2}
EndBegin: -loop expr -fout Model2_index -expr 'Model2_index'+1;:
if -exist ModelGraphic2 -cmd rm ModelGraphic2:
mkgrp ModelGraphic2
##########################################################################
#
# End of file
#
##########################################################################


##########################################################################
#  Application: Fibre Analysis Tool
#  File: Model3.cmd
#  Description: Class histogram
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##########################################################################
#  Parameters:
#
#  0) Equalization (min-max based)
#  1) Equalization (variance based)
#  2) Normalization
#  3) Layer
##########################################################################
##########################################################################
#
# Preprocess
#
```

```
##############################################################################
if -exist Model3_features -cmd rm Model3_features:
if -expr ${ModelParameters[0]}!=0; -cmd prepro -d Features -dout Model3_features -e:
if -exist Model3_features -cmd rm Features: -cmd2 ren Model3_features Features:
if -exist Model3_features -cmd rm Model3_features:
if -expr ${ModelParameters[1]}!=0; -cmd prepro -d Features -dout Model3_features -edev:
if -exist Model3_features -cmd rm Features: -cmd2 ren Model3_features Features:
if -exist Model3_features -cmd rm Model3_features:
if -expr ${ModelParameters[2]}!=0; -cmd prepro -d Features -dout Model3_features -n:
if -exist Model3_features -cmd rm Features: -cmd2 ren Model3_features Features:
##############################################################################
#
# Process
#
##############################################################################
if -exist Model3_layer -cmd rm Model3_layer:
expr -fout Model3_layer -expr ${ModelParameters[3]}+1;
if -exist Model3_som -cmd rm Model3_som:
somtr -d FeaturesSelected -sout Model3_som -l ${Model3_layer}
if -exist Model3_cls -cmd rm Model3_cls:
somcl -s Model3_som -d FeaturesSelected -cout Model3_cls
##############################################################################
#
# Graphics
#
##############################################################################
if -exist ModelGraphic1 -cmd rm ModelGraphic1:
mkgrp ModelGraphic1 -s Model3_som
layer ModelGraphic1 -l ${ModelParameters[3]}
if -exist Model3_hst -cmd rm Model3_hst:
clhisto -c Model3_cls -f Features.${FeaturesSelected[0]} -dout Model3_hst
hst ModelGraphic1 -d Model3_hst.data -co red -sca som2
if -exist ModelGraphic2 -cmd rm ModelGraphic2:
mkgrp ModelGraphic2
##############################################################################
#
# End of file
#
##############################################################################
```

```
##############################################################################
# Application: Fibre Analysis Tool
# File: Model4.cmd
# Description: Fractiles
# Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
# Parameters:
#
# 0) Equalization (min-max based)
# 1) Equalization (variance based)
# 2) Normalization
# 3) Lower bound
# 4) Upper bound
##############################################################################
##############################################################################
#
# Preprocess
#
##############################################################################
if -exist Model4_features -cmd rm Model4_features:
if -expr ${ModelParameters[0]}!=0; -cmd prepro -d Features -dout Model4_features -e:
if -exist Model4_features -cmd rm Features: -cmd2 ren Model4_features Features:
if -exist Model4_features -cmd rm Model4_features:
if -expr ${ModelParameters[1]}!=0; -cmd prepro -d Features -dout Model4_features -edev:
if -exist Model4_features -cmd rm Features: -cmd2 ren Model4_features Features:
if -exist Model4_features -cmd rm Model4_features:
if -expr ${ModelParameters[2]}!=0; -cmd prepro -d Features -dout Model4_features -n:
if -exist Model4_features -cmd rm Features: -cmd2 ren Model4_features Features:
##############################################################################
#
# Process
#
##############################################################################
##############################################################################
##############################################################################
#
# Graphics
#
##############################################################################
if -exist ModelGraphic1 -cmd rm ModelGraphic1:
```

```
mkgrp ModelGraphic1
addsg ModelGraphic1
setsg ModelGraphic1 -inx 0 -rec 0
en ModelGraphic1 barlab1
if -exist Model4_index -cmd rm Model4_index:
expr -expr 0; -fout Model4_index
if -exist Model4_count -cmd rm Model4_count:
len -n Features -fout Model4_count
while -expr 'Model4_index' < 'Model4_count'; -cmd Begin Model4_Loop
if -exist Model4_data${Model4_index} -cmd rm Model4_data${Model4_index}:
select Model4_data${Model4_index} -f Features.${Features[${Model4_index}]}
if -exist Model4_dataL -cmd rm Model4_dataL:
fldquar -d Model4_data${Model4_index} -fout Model4_dataL -v ${ModelParameters[3]}
if -exist Model4_data50f -cmd rm Model4_data50f:
fldquar -d Model4_data${Model4_index} -fout Model4_data50_${Model4_index} -v 50
if -exist Model4_dataH -cmd rm Model4_dataH:
fldquar -d Model4_data${Model4_index} -fout Model4_dataH -v ${ModelParameters[4]}
bar ModelGraphic1 -inx ${Model4_index} -f Model4_data50_${Model4_index} -sca data -min Features.${Features[${Model4_index}]} \
        -max Features.${Features[${Model4_index}]} -co yellow
rngplt ModelGraphic1 -n Model4_data${Model4_index} -inx ${Model4_index} -f Model4_dataL -f Model4_dataH -sca data \
        -min Features.${Features[${Model4_index}]} -max Features.${Features[${Model4_index}]} -co blue -labs
EndBegin: -loop expr -fout Model4_index -expr 'Model4_index'+1;:
if -exist ModelGraphic2 -cmd rm ModelGraphic2:
mkgrp ModelGraphic2
##############################################################################
#
# End of file
#
##############################################################################


##############################################################################
#  Application: Fibre Analysis Tool
#  File: Model5.cmd
#  Description: Mosaic
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
#  0) Equalization (min-max based)
```

```
# 1) Equalization (variance based)
# 2) Normalization
# 3) Layer
# 4) Width
# 5) Height
# 6) Frame
#############################################################################
#############################################################################
#
# Preprocess
#
#############################################################################
if -exist Model5_features -cmd rm Model5_features:
if -expr ${ModelParameters[0]}!=0; -cmd prepro -d Features -dout Model5_features -e:
if -exist Model5_features -cmd rm Features: -cmd2 ren Model5_features Features:
if -exist Model5_features -cmd rm Model5_features:
if -expr ${ModelParameters[1]}!=0; -cmd prepro -d Features -dout Model5_features -edev:
if -exist Model5_features -cmd rm Features: -cmd2 ren Model5_features Features:
if -exist Model5_features -cmd rm Model5_features:
if -expr ${ModelParameters[2]}!=0; -cmd prepro -d Features -dout Model5_features -n:
if -exist Model5_features -cmd rm Features: -cmd2 ren Model5_features Features:
#############################################################################
#
# Process
#
#############################################################################
if -exist Model5_layer -cmd rm Model5_layer:
expr -fout Model5_layer -expr ${ModelParameters[3]}+1;
if -exist Model5_mosaic_s1 -cmd rm Model5_mosaic_s1:
somtr -d FeaturesSelected -sout Model5_mosaic_s1 -l ${Model5_layer}
if -exist Model5_mosaic_cls -cmd rm Model5_mosaic_cls:
somcl -s Model5_mosaic_s1 -d FeaturesSelected -cout Model5_mosaic_cls
if -exist Model5_mosaic_image -cmd rm Model5_mosaic_image:
mosaic -out Model5_mosaic_image -c Model5_mosaic_cls -f Files -l ${ModelParameters[3]} -w ${ModelParameters[4]} \
       -h ${ModelParameters[5]} -frame ${ModelParameters[6]} -corner
#############################################################################
#
# Graphics
#
#############################################################################
```

```
if -exist ModelGraphic1 -cmd rm ModelGraphic1:
mkgrp ModelGraphic1
bgimg ModelGraphic1 -img Model5_mosaic_image
if -exist ModelGraphic2 -cmd rm ModelGraphic2:
mkgrp ModelGraphic2
##############################################################################
#
# End of file
#
##############################################################################


##############################################################################
#  Application: Fibre Analysis Tool
#  File: Model6.cmd
#  Description: Classical scaling and Sammon's mapping
#  Author: Markku Mielityinen, University of Jyväskylä, 2001
##############################################################################
#  Parameters:
#
#  0) Equalization (min-max based)
#  1) Equalization (variance based)
#  2) Normalization
#  3) Layer
##############################################################################
##############################################################################
#
# Preprocess
#
##############################################################################
if -exist Model6_features -cmd rm Model6_features:
if -expr ${ModelParameters[0]}!=0; -cmd prepro -d Features -dout Model6_features -e:
if -exist Model6_features -cmd rm Features: -cmd2 ren Model6_features Features:
if -exist Model6_features -cmd rm Model6_features:
if -expr ${ModelParameters[1]}!=0; -cmd prepro -d Features -dout Model6_features -edev:
if -exist Model6_features -cmd rm Features: -cmd2 ren Model6_features Features:
if -exist Model6_features -cmd rm Model6_features:
if -expr ${ModelParameters[2]}!=0; -cmd prepro -d Features -dout Model6_features -n:
if -exist Model6_features -cmd rm Features: -cmd2 ren Model6_features Features:
##############################################################################
```

```
#
# Process
#
#########################################################################################
if -exist Model1_layer -cmd rm Model1_layer:
expr -fout Model1_layer -expr ${ModelParameters[3]}+1;
if -exist Model1_som -cmd rm Model1_som:
if -exist Model1_cls -cmd rm Model1_cls:
somtr -d Features -sout Model1_som -cout Model1_cls -l ${Model1_layer}
clstat -c Model1_cls -d Features -dout Model1_stat -all
if -exist Model1_umatrix -cmd rm Model1_umatrix:
#if -exist Model1_som_W -cmd rm Model1_som_W:
calcumat -s Model1_som -d Model1_som_W -umat Model1_umatrix
if -exist Model1_grid3 -cmd rm Model1_grid3:
if -exist Model1_w3 -cmd rm Model1_w3:
psl -s Model1_som -dout Model1_grid3 -l ${ModelParameters[3]} -wout Model1_w3
if -exist Model1_sammon_out -cmd rm Model1_sammon_out:
sammon -dinit Model1_grid3 -d Model1_w3 -dout Model1_sammon_out -imax 300 -eps 0.01 -ef Error
if -exist Model1_sammon_weights -cmd rm Model1_sammon_weights:
ssl -s Model1_som -sset Model1_sammon_out -sout Model1_sammon_weights -l ${ModelParameters[3]}
classical -d Model1_w3 -dout Model1_class_out -dim 2
#if -exist Model1_class_out -cmd rm Model1_class_out:
if -exist Model1_class_weights -cmd rm Model1_class_weights:
ssl -s Model1_som -sset Model1_class_out -sout Model1_class_weights -l ${ModelParameters[3]}
#########################################################################################
#
# Graphics
#
#########################################################################################
if -exist Classical -cmd rm Classical:
mkgrp Classical -s Model1_som
setgdat Classical -d Model1_stat
layer Classical -l ${ModelParameters[3]}
axis Classical -ax x -f Model1_sammon_weights.0 -sca som2 -step 5
nplc Classical -ax x -f Model1_sammon_weights.0
axis Classical -ax y -f Model1_sammon_weights.1 -sca som2 -step 5
nplc Classical -ax y -f Model1_sammon_weights.1
topo Classical -on
nsize Classical -v 0.5
nshape Classical circle
```

```
dis Classical saxis
viewp Classical -dz 140
################################################################################
# Displaing classical
################################################################################
if -exist Sammon -cmd rm Sammon:
mkgrp Sammon -s Model1_som
setgdat Sammon -d Model1_stat
layer Sammon -l ${ModelParameters[3]}
axis Sammon -ax x -f Model1_class_weights.0 -sca som2 -step 5
nplc Sammon -ax x -f Model1_class_weights.0
axis Sammon -ax y -f Model1_class_weights.1 -sca som2 -step 5
nplc Sammon -ax y -f Model1_class_weights.1
topo Sammon -on
nsize Sammon -v 0.5
nshape Sammon circle
dis Sammon saxis
viewp Sammon -dz 140
################################################################################
# Displaing U-matirx
################################################################################
if -exist UMatrix -cmd rm UMatrix:
mkgrp UMatrix -s Model1_som
setgdat UMatrix -d Model1_stat
layer UMatrix -l ${ModelParameters[3]}
umat UMatrix -umat Model1_umatrix
topo UMatrix -on
nsize UMatrix -v 1.0
nshape UMatrix rectangle
dis UMatrix saxis
viewp UMatrix -dz 140
################################################################################
# Compile a frame
################################################################################
if -exist ModelGraphic1 -cmd rm ModelGraphic1:
mkgf -gf ModelGraphic1
addgf -gf ModelGraphic1 -g Classical
addgf -gf ModelGraphic1 -g Sammon
addgf -gf ModelGraphic1 -g UMatrix
setgfplc -gf ModelGraphic1 -g Classical -x0 0 -y0 0.05 -x1 0.5 -y1 0.5
```

```
setgfplc -gf ModelGraphic1 -g Sammon -x0 0.5 -y0 0.05 -x1 1.0 -y1 0.5
setgfplc -gf ModelGraphic1 -g UMatrix -x0 0 -y0 0.5 -x1 0.5 -y1 1.0
if -exist ModelGraphic2 -cmd rm ModelGraphic2:
mkgrp ModelGraphic2
##############################################################################
#
# End of file
#
##############################################################################
```