Janne Korhonen

UIML-MÄÄRITTELYKIELEN AVULLA ESITETYN KÄYTTÖLIITTYMÄN MUODOSTAMINEN

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniiikan linja

20.12.2002

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Janne Korhonen

Yhteystiedot: Sähköposti jankorho@cc.jyu.fi ja puh. 050-5280928.

Työn nimi: UIML-määrittelykielen avulla esitetyn käyttöliittymän muodostaminen

Title in English: Generating user interfaces specified with UIML

Työ: Pro gradu -tutkielma

Sivumäärä: 63+4

Linja: Ohjelmistotekniikka

Teettäjä: Honeywell Oy ja Jyväskylän yliopiston tietotekniikan laitos

Avainsanat: UIML, rakenteinen, määrittelykieli, dynaaminen käyttöliittymä, MFC, COM, ATL, NPDI, STL.

Keywords: UIML, structured, definition language, dynamic interface, MFC, COM, ATL, NPDI, STL.

Tiivistelmä: Tässä opinnäytteessä tutkitaan UIML-määrittelykielen käyttöä osana dynaamista käyttöliittymän kehitystä. Tutkimuksen tueksi kehitetään sovellus, jonka avulla UIML:n ominaisuuksia arvioidaan. Opinnäytteessä arvioidaan myös sovelluksen kehitykseen käytettävien tekniikoiden soveltuvuutta dynaamiseen käyttöliittymien kehittämiseen ja NPDI-prosessimallia osana pienehköä ohjelmistoprojektia.

Abstract: This thesis considers UIML-definition language as part of creating dynamic user intefaces. UIML's properties are studied and tested by creating the specified UIML-rendering machine. This thesis also examines NPDI-process model and technologies used in UIML-rendering application, especially how they fit into the development of dynamic user interfaces.

Termiluettelo

ActiveX	Microsoftin kehittämien tekniikoiden joukko, joka mahdollistaa ohjelmistokomponenttien vuorovaikutuksen verkkoympäristössä riippumatta siitä, millä kielellä komponentit on luotu.
DHTML	*Dynamic HTML*, tyylisäännöstöjen ja skriptien yhdistelmä, joka mahdollistaa WWW-sivujen dynaamisen muuntamisen.
DOM	*Document Object Model*. Dokumenttioliomalli, joka määrittelee, kuinka dokumentissa olevat elementit välittävät tietoa toisilleen ja kuinka elementteihin voidaan viitata.
Geneerisyys	Geneerisyys tarkoittaa sitä, että yleiskäsitteestä tehdään yksilökäsite. Esimerkiksi geneeristä aliohjelmaa voidaan kutsua useilla eri parametrilistoilla ja jokaista toteutusta ei tarvitse erikseen toteuttaa.
HTML	*Hyper Text Markup Language*, tekstin rakennetta kuvaava määrittelykieli.
Lokalisointi	Ominaisuuksille annetaan erilaiset arvot riippuen käyttöpaikasta. Esimerkiksi käyttöliittymän taustaväri on valkoinen Suomessa mutta punainen Kiinassa.
MSXML	XML-parseri, joka tarkistaa XML-dokumentin oikeellisuuden. Käyttää validointiin DOM-mallia.
OMT	*Object Modeling Technique*. Yksi suosituimmista olio-ohjelmien suunnittelumenetelmistä.

Palm	U.S Roboticsin PalmPilot -taskutietokoneelle kehittämä käyttöjärjestelmä.
Postfix	Jälkiasetusoperaattori, joka esimerkiksi lisää muuttujan arvoa yhdellä muuttujan käytön jälkeen (esim. `i++`).
Prefix	Esiasetusoperaattori, joka esimerkiksi lisää muuttujan arvoa yhdellä ennen muuttujan käyttöä (esim. `++i`).
Renderöidä	Vrt. hahmontaa. Viitataan toimintaan, jonka tuloksena renderöinnin kohteesta luodaan haluttu tulostusasu.
STA	*Single Threaded Apartment*, ohjelmoinnissa käytetty termi säieturvallisuuden takaavalle ominaisuudelle. Esimerkiksi komponentille voidaan STA:n avulla määritellä ominaisuus, jonka mukaan sitä voidaan käyttää vain tietyistä säikeestä.
UIML	*User Interface Markup Language*, rakenteinen käyttöliittymien määrittelykieli.
URL	*Uniform Resource Locator*, osoite, joka viittaa Internetissä olevaan "resurssiin". Tämä resurssi voi olla esimerkiksi tekstidokumentti, HTML-dokumentti, kuva, äänite tai ohjelma.
W3C	*World Wide Web Consortium*, WWW:n ja siihen liittyvien standardien kehittämisorganisaatio.
WAP	*Wireless Application Protocol*, langattoman tiedonsiirron protokolla, jonka avulla voidaan luoda kehittyneitä teleliikennepalveluja sekä käyttää Internet-sivuja matkapuhelimen avulla.
Windows API	Windows-ohjelmointiin tarkoitettu ohjelmistorajapinta.

WML	*Wireless Markup Language*, merkintäkieli, joka pohjautuu XML:ään ja on tarkoitettu sisällön ja käyttöliittymien määrittelyyn kapeakaistaisiin laitteisiin, kuten matkapuhelimiin.
XML	*Extensible Markup Language*, rakenteinen tiedon määrittelykieli.
XSD	*XML Schema Definition* -kielellä määritetään XML-asiakirjojen rakenne.
XUL	*XML-based User Interface Language*, käyttöliittymien kuvauskieli.

Sisältö

1 Johdanto

Käyttöliittymiä käyttävien laitteiden lukumäärän ja käyttöliittymien kehittämiseen tarkoitettujen työkalujen määrän kasvaessa on alkanut ilmetä tarvetta entistä yhtenäisemmälle tavalle luoda monimutkaisempiakin käyttöliittymiä. Tässä opinnäytteessä käytettävä ja tutkittava UIML-määrittelykieli (*User Interface Markup Language*) pyrkii ratkaisemaan nämä molemmat ongelmat. UIML:n avulla luodut käyttöliittymät soveltuvat erilaisiin päätelaitteisiin ja näin myöskään käyttöliittymän luojilla ei ole tarvetta opetella useita eri päätelaitteille tarkoitettuja käyttöliittymän kehitysmenetelmiä. Myös rakenteisten määrittelykielten (esim. HTML, XML, SMIL) käytön yleistyminen vaikutti osaltaan mielenkiintoon tutkia UIML:n tarjoamia mahdollisuuksia käytännössä.

Tärkeimpiä tässä opinnäytteessä käytettyjä termejä ovat rakenteinen määrittelykieli ja käyttöliittymän dynaaminen luonti. *Rakenteisella määrittelykielellä* tarkoitetaan tulkattavaa (*interpreted*) ohjelmointikieltä, jolla luotuun rakenteiseen dokumenttiin liittyy tietokoneen tulkittavissa oleva rakennemäärittely [Salminen, 1992, s. 6]. Esimerkiksi XML-määrittelykielessä rakenteellisuus perustuu kielen kykyyn erottaa rakenne, sisältö ja tyylit toisistaan [Mace, 1998, s. 60]. *Dynaamisella käyttöliittymän luonnilla* tarkoitetaan tämän opinnäytteen osalta rakenteisen UIML-dokumentin avulla määritellyn käyttöliittymän luontia ennalta sovittuja tekniikoita käyttäen.

UIML:n avulla haetaan ratkaisua lukuisiin käyttöliittymien kehittämisen hankaluuksiin. Jo kuvattujen päätelaitteiden ja työkalujen kasvavan määrän lisäksi ongelmina pidetään sovellusten suurta kokoa ja käyttöliittymien toteuttajien taitojen painottumista tekniselle puolelle. Tässä opinnäytteessä kehitettävän UIML-renderöintikoneiston tapauksessa ja yleisemminkin dynaamisessa käyttöliittymän kehittämisessä ongelmia ja valintoja aiheuttavat käyttöliittymäkirjasto ja tekniikoiden yhteensovittaminen. Varsinkin käyttöliittymäkirjaston soveltuminen dynaamiseen käyttöliittymän luontiin vaikuttaa ratkaisevasti lopputulokseen.

Tämän opinnäytteen tavoitteena on tutkia UIML:n soveltuvuutta käyttöliittymien luontiin. Erityisesti tarkastellaan nykyisiä käyttöliittymäkehityksen ongelmia ja kuinka UIML näitä ongelmia ratkaisee. Työn muita tutkimusongelmia ovat kehitettävän UIML-renderöintikoneiston toteuttamisessa käytettävien tekniikoiden yhteistoiminta ja niiden soveltuvuus dynaamiseen käyttöliittymän luontiin sekä NPDI-prosessimallin soveltuvuus pieneen ohjelmistoprojektiin. Opinnäytteen teoriaosuuden sisältö on rajattu valittuihin tekniikoihin, vaihtoehtoisia tekniikoita on käytetty ja kuvattu niukasti. Rajoitteena voidaan pitää myös UIML:n käytön tutkimista tutkimuksen luonteen mukaisesti lyhyellä aikavälillä, sillä kokemuksia sen soveltuvuudesta käyttötarkoitukseensa pidemmällä aikavälillä ei ole haettu.

Koska opinnäytteen tärkeänä osana on kehittää UIML:n ominaisuuksien tutkimiseen erillinen sovellus, on tutkimuksen lähestymistapa käytännönläheinen ja saatuihin kokemuksiin perustuva. Sovelluspohjaisuudesta huolimatta ei teoreettista lähestymistapaakaan ole täysin unohdettu, varsinkin ongelman sisäistäminen abstraktimmalla tasolla vaatii tekniikoiden tuntemusta ja täten myös alan kirjallisuuden tutkimista.

Opinnäyte jakautuu seitsemään lukuun. Ensimmäisessä luvussa on opinnäytteen johdanto. Luvussa 2 kuvataan UIML-määrittelykieltä yleensä ja sen syntaksia tässä opinnäytteessä tarvituilta osilta. Kolmannessa luvussa kerrotaan käytetyistä tekniikoista: NPDI-prosessimallista, C++:n STL-luokkakirjastosta sekä MFC-käyttöliittymäkirjastosta. Luku 4 sisältää kehitetyn UIML-renderöintikoneiston vaatimusmäärittelyn ja kuvauksen sovelluksen ympärillä olevan järjestelmän arkkitehtuurista. Viidennessä luvussa kerrotaan sovelluksen toiminnasta ja kuvataan sen pääluokkien rakenteet ja tehtävät. Luvussa 6 arvioidaan käytettyjä tekniikoita kuten UIML-määrittelykieltä ja MFC:n dynaamisia ominaisuuksia. Lisäksi luvussa arvioidaan kehitetyn sovelluksen lopputulosta sekä kerrotaan NPDI-prosessimallista saaduista kokemuksista. Luvussa 7 on opinnäytteen yhteenveto, ja tutkielman lopussa käytetyt lähteet ja liitteet.

2 User Interface Markup Language (UIML)

Tässä luvussa kuvataan käyttöliittymien kuvaukseen tarkoitettua UIML-määrittelykieltä (*User Interface Markup Language*). Luvussa 2.1 kerrotaan UIML:sta yleisesti, luvussa 2.2 syntaksin perusteet ja luvuissa 2.3-2.6 kuvataan tarkemmin käyttöliittymän kehittämistä.

2.1 Mikä on UIML?

User Interface Markup Language (UIML) on käyttöliittymien kuvaamiseen tarkoitettu rakenteinen määrittelykieli. UIML:n avulla kuvatut käyttöliittymät koostuvat tavallisista käyttöliittymäelementeistä, kuten ikkunoista, dialogeista ja nappuloista. UIML-määrittelykieli tarjoaa työkalut myös eri tapahtumien tunnistamiseen ja niihin reagointiin. Tulevaisuuden tyypillinen skenaario UIML:llä tehdyille käyttöliittymille on, että samaa käyttöliittymää voitaisiin käyttää kaikissa päätelaitteissa, kuten WAP-puhelimessa, WWW-selaimessa ja WebTV:ssä [Abrams, 1999, s. 1965]. Määrittelyltään UIML pohjautuu XML:ään (*Extensible Markup Language*) ja ulkonäöltään se on hyvin samanlainen kuin HTML (*Hyper Text Markup Language*).

2.1.1 UIML:n historia ja eri versiot

1990-luvun puolessa välissä havaittiin tarvetta uudenlaiselle yleiskäyttöiselle käyttöliittymien kuvauskielelle. Tarvetta perusteltiin kahdella vahvalla syyllä. Ensimmäinen peruste uudelle kuvauskielelle oli vanhojen jo käytössä olevien käyttöliittymien kehitysmenetelmien suuri lukumäärä (mm. DHTML, XML-based User Interface Language eli XUL, WML, ja perinteiset ohjelmointikielet kuten Java, Visual Basic and C++). Toisena vahvana perusteena pidettiin lisääntyvää erilaisten päätelaitteiden määrää. Tällaisia uusia laitteita olivat mm. WAP-puhelimet, kannettavat tietokoneet ja WebTV. Laitteiden ja kielien lisääntyessä käyttöliittymien tekemiseen tarvittiin siis useiden eri ohjelmointityökalujen ja -kielien osaamista [Abrams, 1999, s. 1965]. Ongelmaan tartuttiin ja sen tuloksena UIML:n ympärille perustettiin oma yritys nimeltään Harmonia (`www.harmonia.com`), jonka toimesta UIML:n ensimmäinen versio julkaistiin vuoden 1997 joulukuussa. Tämän jälkeen versio 2.0 julkaistiin tammikuussa 2000 ja viimeisin versio 3.0 helmikuussa 2002.

2.1.2 Käyttötilanteet ja hyöty

Tässä kappaleessa kuvataan UIML:n käyttötilanteita ja kerrotaan UIML:n hyödyistä kyseisissä tilanteissa.

Syntaksin helppous: Yksi UIML:n tärkeimmistä ominaisuuksista on kielen syntaksin helppous. UIML:n perustuessa XML:ään ovat ne syntaksiltaankin hyvin samankaltaisia. Perinteisestä WWW-sivujen määrittelykielestä HTML:stä UIML eroaa siten, että sen tunnisteet eivät ole valmiiksi ennalta määrättyjä (*build-in tags*) [Marchal, 2000, s. 18].

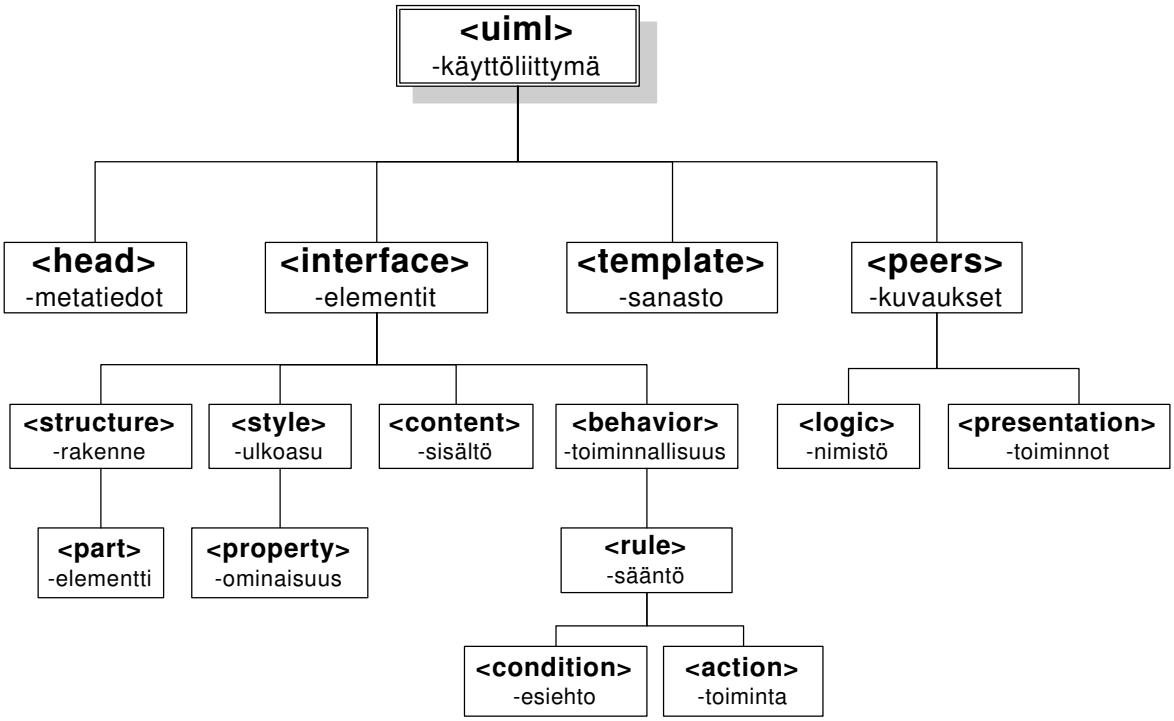
Jos siis XML on käyttäjälle ennestään tuttu, on tällöin myös UIML:n käyttö erittäin helppo ja nopea oppia. Helppoudesta johtuen normaali käyttöliittymäohjelmoinnin osuus toteutuksessa vähenee, ja muutkin kuin ammattitaitoiset ohjelmoijat pystyvät kehittämään käyttöliittymiä.

Käyttöliittymän jakautuminen kahteen osaan: Eräs tärkeimmistä ominaisuuksista on UIML:llä toteutettavan käyttöliittymän selkeä jakautuminen kahteen osaan, itse käyttöliittymän ulkoiseen osaan ja sen sisäiseen toimintalogiikkaan. Jakautuminen selkeyttää huomattavasti käyttöliittymän kehittämisen työnjakoa: sisäisen logiikan ja rajapintojen määritys voidaan perinteisesti jättää ohjelmointiasiantuntijalle, kun taas käyttöliittymän näkyvä ulkoinen osa voidaan suunnitella ja toteuttaa esim. graafisen alan ammattilaisen tai käytettävyyteen perehtyneen kehittäjän toimesta. Selkeä jako kahteen osaan mahdollistaa myös useamman käyttöliittymän yhdistämisen samaan toimintalogiikkaan. Tämä helpottaa esimerkiksi aloittelijoita varten rakennettujen käyttöliittymäratkaisujen toteuttamista (esim. erilaiset avustajat aloitettaessa ohjelman käyttö).

Kielen joustavuus ja siirrettävyys: UIML on käytettävään ulkoiseen rajapintaan sitoutumaton käyttöliittymäkieli, eli kullekin rajapinnalle ominaiset tunnisteet, attribuutit tai avainsanat eivät rajoita UIML:n käyttöä. Tämä mahdollistaa sen, että UIML toimii myös tulevaisuuden sovellusten ja rajapintojen kanssa. Myös käyttöliittymien siirrettävyys eri alustojen (esim. WAP, PC, Palm) välillä helpottuu.

Parhaimmillaan käyttöliittymää ei tarvitse muokata päätelaitetta vaihdettaessa lainkaan, vaan kullekin laitteelle ominaisen sanaston käyttöönotto riittää [Abrams, 1999, s. 1965]. Usein päätelaitetta vaihdettaessa joudutaan kuitenkin esimerkiksi koon asettamien rajoitusten vuoksi muokkaamaan elementtien kokoja tai ulkoasuja.

Sovelluksen pieni koko ja sijainnin vapaus: Verkon käyttönopeuksissa olevien suurten erojen johdosta (modeemi vs. kiinteät yhteydet) on sovelluksen koolla ja siten sen siirrettävyydellä suuri merkitys. Ajettavaan käyttöliittymään verrattuna UIML-dokumentin koko on erittäin pieni, jolloin myös siirrettävyys paranee huomattavasti. Toisaalta HTML:ään verrattuna koko on suurin piirtein sama, mutta HTML ei mahdollista yhtä monipuolisten käyttöliittymien kehitystä kuin UIML. Myös jo XML:stä tuttu ominaisuus, tiedostojen sijainnin vapaus, nopeuttaa verkkokäyttöä. Tiedostojen (mm. käytetty dokumentin tyyppimääritys eli DTD-tiedosto tai käytettävät kuvatiedostot) sijainti pystytään määrittämään URL:n (*Uniform Resource Locator*) avulla. Tämä mahdollistaa sen, että ko. tiedostojen ei tarvitse sijaita samalla koneella kuin käyttöliittymä [Abrams, 1999, s. 1965].



Kuva 1. UIML-dokumentin perusrakenne [Harmonia, 2002].

2.2 UIML-määrittelykielen syntaksin perusrakenne

Tässä kappaleessa kuvataan UIML-dokumentin perusrakenne tässä opinnäytteessä kehitettävässä sovelluksessa tarvittavilta osin. Rakennetta on havainnollistettu kuvassa 1.

2.2.1 Yleistä syntaksista

XML:n mukaisesti UIML perustuu tunnisteiden (*tag*) käyttöön. Tunnisteita on eritasoisia (eli tunniste voi sisältää lisää tunnisteita) ja jokainen niistä sisältää määrätyt osat dokumentin halutusta sisällöstä.

XML:n ominaisuuksien mukaan UIML erottaa keskenään isot ja pienet kirjaimet (*case sensitive*). UIML-dokumentissa tunnisteet tuleekin kirjoittaa pienin kirjaimin. Toimiakseen tunniste vaatii aina myös tunnisteen päättävän parin, esimerkiksi `<tunniste>... </tunniste>` [Harold, 2001, s. 148]. Kommentit UIML-syntaksissa kirjoitetaan esimerkissä 1 kuvatulla tavalla merkkien `<!` ja `-->` väliin.

Aivan kuten XML, UIML ei ole riippuvainen mistään tietyistä tunnisteista, vaan kussakin UIML-dokumentissa hyväksyttävät tunnisteet ja parametrit määritetään joko DTD-tiedostossa (*Document Type Definition*) [Harmonia, 2002, s. 15] tai XML:n kuvauksissa yleisemmin käytetyssä XSD-muotoisessa (*XML Schema Definition*) mallissa. Tässä opinnäytteessä on käytetty kappaleessa 2.2.2. kuvatun Harmonia-yhtiön luoman UIML-spesifikaation 3.0 mukaista standardia (`http://uiml.org/dtds/UIML_3.0a.dtd`) pitkälti muistuttavaa XSD-mallia. UIML:n avulla käyttöliittymiä kehitettäessä tarvitaan DTD-tiedoston lisäksi myös erillinen sanasto (*vocabulary*), joka määrittää käytettävän ohjelmistokirjaston tai määrittelykielen. Sanaston rakenne ja määrittely on kuvattu luvussa 2.5.1.

Jokainen UIML-dokumentti vaatii alkuunsa tietyntyyppisen pakollisen prologin. Tämä sisältää tiedot XML:n versiosta, käytetystä sanastosta ja DTD-tiedostosta tai XSD-mallista [Harmonia, 2002, s.29]. Esimerkissä 1 on kuvattu UIML-dokumentin perusrakenne, sanaston käyttöönotto ja jokaisessa UIML-dokumentissa olevan prologin sisältö.

Esimerkki 1. Tunnisteiden käyttö UIML-dokumentissa ja prologin sisältö.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 3.0 Draft// EN"
  http://uiml.org/dtds/UIML_3.0a. dtd">
<uiml>
   <head>
      <! head-tunnisteen sisään tulevat osat. -->
    </head>
    <template>      ... </template>
    <interface>     ... </interface>
    <peers>         ... </peers>
</uiml>
```

Ylimmän tason tunniste UIML-dokumentissa on siis `<uiml>`, joka jakautuu neljään alaosioon: `<head>`, `<template>` , `<interface>` ja `<peers>` (käyttö kuvattu luvuissa 2.3 –2.6).

UIML-kielessä, kuten jokaisessa XML-pohjaisessa kielessä, itse informaatio sijaitsee tunnisteiden sisällä nk. määreissä (*attribute*) [Harold, 2001, s. 153]. Jokaisella tunnisteella on omat määreensä ja jokaisella määreellä on tarkkaan määritelty tehtävä UIML-käyttöliittymässä. Määreiden käyttöä on kuvattu luvussa 2.3 esimerkissä 2.

2.3 Metadatan määrittely `head`-tunnisteessa

Esimerkissä 1 määriteltiin toiseksi ylimmän tason tunniste `<head>`, jonka avulla UIML-käyttöliittymään liitetään ns. meta- eli tunnistetietoja. Tunnistetietoina voidaan antaa mm. käyttöliittymän tekijä, yhteystietoja ja tekijänoikeustietoja. Esimerkissä 2 määritellään UIML-dokumentille tunnistetiedot.

Esimerkki 2. Määreiden käyttö ja tunnistetietojen määrittäminen.

```
<head>
  <meta name="title" content="Pro-Gradu"/>
  <meta name="author" content="Janne Korhonen"/>
  <meta name="copyright" content="(c)2002 Janne Korhonen"/>
  <meta name="abstract" content="Tutkielma UIML:stä"/>
  <meta name="email" content="jankorho@cc.jyu.fi"/>
</head>
```

2.4 Käyttöliittymän kuvaaminen `interface`-tunnisteessa

Tässä luvussa kuvataan `<interface>`-tunnisteen sisältöä ja määrittelyä käyttöliittymää luotaessa. `<interface>`-tunnisteen avulla käyttöliittymään yhdistetään sen abstraktit osat, tapahtumat ja funktiokutsut. `<interface>`-tunniste jakautuu neljään alatunnisteeseen: `<structure>`, `<style>`, `<content>` ja `<behavior>` [Phanouriou, 2000, s. 69]. Seuraavassa on lyhyesti kuvattuna kunkin tunnisteen tehtävä käyttöliittymää kehitettäessä:

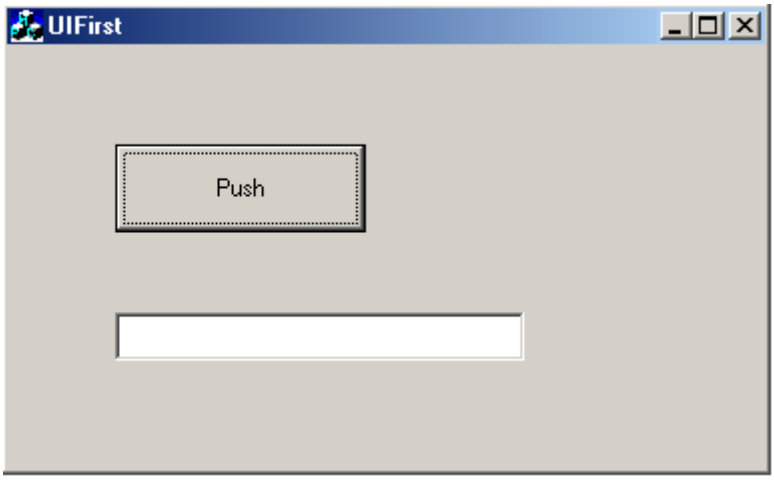
`<structure>`: Luettelee käyttöliittymän eri osat ja määrittää niiden suhteet toisiinsa.

`<style>`: Määrittää arvot ominaisuuksille, joita käyttöliittymää luotaessa tarvitaan.

`<content>`: Määrittää käyttöliittymän vaihtoehtoiset sisällöt.

`<behavior>`: Määrittää käyttöliittymän tapahtumat ja tapahtumien aiheuttamat toiminnot.

Tunnisteiden käyttöä havainnollistetaan kuvassa 2 esitetylle käyttöliittymälle `UIFirst` esimerkeissä 3-6. Tässä opinnäytteessä käyttöliittymäkirjastona käytettiin MFC:tä (*Microsoft Foundation Classes*), joten kaikki luvun 2 esimerkit käyttävät MFC-pohjaista sanastoa (käyttö kuvattu luvussa 2.5.1 ja esimerkki sanastosta liitteessä 1).



Kuva 2. Käyttöliittymä `UIFirst`.

2.4.1 Elementtien määrittely `structure`-tunnisteessa

`<structure>`-tunnisteessa luetellaan käyttöliittymään kuuluvat elementit. `<structure>`-osioita voi yhdessä UIML-dokumentissa olla useampia, tämän takia ne erotetaan toisistaan `id`-parametrillä [Harmonia, 2002, s. 33]. Kussakin `<structure>`-tunnisteessa määritelty käyttöliittymä muodostaa oman ikkunansa. Yhden `<structure>`-tunnisteen sisältämät elementit kuvataan erillisissä `<part>`-tunnisteissa (katso esimerkki 3).

Esimerkki 3. `<structure>`-tunnisteen käyttö.

```
<structure id="UIFirst">
  <part  class="cFrameWnd" id="mainframe">
    <part class="cButton" id="button">
    <part class="cEdit" id="edit">
  </part>
</structure>
<structure id="UISecond">
  <part class="cWnd" id="secmainwnd">
    <part class="cButton" id="secbutton">
  </part>
</structure>
```

Esimerkissä 3 käyttöliittymä `UIFirst` koostuu kolmesta elementistä (kullakin oma `<part>`-tunniste). `<part>`-tunnisteessa jokainen käyttöliittymän elementti yhdistetään haluttuun luokkaan parametrin `class` avulla. `class`-parametrille annetaan arvoksi sanastossa (katso luku 2.5.1) luotavaa elementtiä vastaava `<d-class>`-tunnisteen parametrin `id`-arvo (esimerkissä 3 `cFrameWnd`).

2.4.2 Ominaisuuksien määrittely `style`-tunnisteessa

`<style>`-tunnisteessa määritellään käyttöliittymän elementtien ja luokkien ominaisuudet. Näitä ominaisuuksia ovat mm. elementtien koot, värit ja käytetyt fontit. Ominaisuuksien määrittely tapahtuu `<property>`-tunnisteen avulla. Esimerkissä 4 määritellään ominaisuuksia esimerkin 3 ja kuvan 2 käyttöliittymälle `UIFirst`.

Esimerkki 4. `<property>`-tunnisteen käyttö.

```
<style>
  <property part-class="cFrameWnd" name ="foreground">
  white</property>
  <property part-name="button" name="caption"> Push
  </property>
</style>
```

`<property>`-tunniste yhdistää nimen ja arvoparin joko käyttöliittymän osaan tai toimintoon [Harmonia, 2002, s. 36]. Toimintojen (*event*) käyttö on kuvattu luvussa 2.4.4. Nimiä (attribuutti `name`) ei ole määritelty UIML-syntaksissa, vaan kyseisen käyttöliittymän sanastossa. Näin saadaan UIML-dokumentteihin lisää joustavuutta: ominaisuudet voidaan nimetä aina käyttötarkoituksensa mukaan, jolloin niiden muistaminen ja käyttö helpottuu.

<property>-tunnisteelle määritettävä arvo voidaan antaa usealla tavalla. Esimerkissä 4 arvo annetaan ns. yleisellä tavalla, tekstimuodossa (*string*). Muita vaihtoehtoja antaa arvo ovat mm. toisen <property>-tunnisteen (toinen <property>-tunniste toimii saantimetodina) tai <reference>-tunnisteen avulla (katso esimerkki 5).

2.4.3 Vaihtoehtoisten ominaisuuksien määrittely content-tunnisteessa

<content>-tunnisteessa määritellään käyttöliittymän vaihtoehtoisia ominaisuuksia. Tunnistetta käytetään hyväksi mm. silloin, kun samasta käyttöliittymästä tarvitaan useamman kieliset versiot tai kun käyttöliittymästä tehdään aloittelijalle ja kehittyneemmälle käyttäjälle omat versiot.

Esimerkki 5. <content>-tunnisteen käyttö.

```
<style>
  <property part-name =”mainframe” name =”foreground”>
     <reference constant-name=”backcolor”>
</style>
<content id=”Finland”>
  <constant id=”backcolor”> gray </property>
</content>
<content id=”China”>
  <constant id=”backcolor”> red </property>
</content>
```

Esimerkissä 5 asetetaan mainframe-osion foreground-attribuutin arvo riippuen käyttömaasta. Mikäli käyttömaa on Suomi, tulee taustaväriksi harmaa, Kiinan tapauksessa väriksi tulee punainen. Mikäli <content>-tunnisteen id-attribuuteista ei oikeaa vaihtoehtoa löydy, ei käyttöliittymää voida luoda [Harmonia, 2002, s. 46].

2.4.4 Sisäisten toimintojen määrittely `behavior`-tunnisteessa

<behavior>-tunnisteessa määritellään toiminta loppukäyttäjän toiminnoille käyttöliittymässä, joita ovat mm. nappulan painallus, tekstin syöttö tai listan aktivointi. Toiminnot voivat olla joko UIML-dokumentin sisäisiä (*internal*) toimintoja tai nk. ulkoisia (*external*). Sisäisillä toiminnoilla voidaan esimerkiksi muuttaa jonkun ominaisuuden arvoa, ulkoisella toiminnolla laukaista jokin tietty funktio käyttöliittymää käyttävästä ohjelmasta tai skriptistä.

Kaikkiaan <behavior>-tunnisteessa voidaan määritellä kolmen tyyppisiä sisäisiä toimintoja: ominaisuuden arvon asetus, metodin kutsuminen tai toiminnon (*event*) laukaisu [Harmonia, 2002, s. 50]. Esimerkissä 6 kuvataan kaikkien kolmen toiminnon määrittely. Samalla esimerkissä 6 käydään läpi <behavior>-tunnisteen alitunnisteiden <rule>, <condition>, <event> ja <action> käyttö.

Esimerkki 6. <behavior>-tunnisteen käyttö.

```
<behavior>
  <rule>
   <condition>
    <event class="ButtonClicked" part-name="button">
   </condition>
   <action>
    <property part-name="button" name="color"/> red
      </property>
    <call name="edit.SetText">
      <param> ActionText </param>
    <event class="TextChanged" part-name="edit">
   </action>
  </rule>
 </behavior>
```

<rule>-tunnisteen sisällä määritellään aina condition-action-pari. Kyseisen parin sisältämä <action>-elementti suoritetaan aina, kun <condition>-tunnisteen sisällä oleva esiehto täyttyy. Esimerkissä 6 esiehtona on, että elementtiin nimeltä button on kohdistunut luokan ButtonClicked tapahtuma. Toimintona voi olla esimerkiksi nappulan painallus, joka määritellään <peers>-tunnisteen sisällä (katso luku 2.5). Tässä esimerkissä kyseisen toiminnon tapahtuessa siihen reagoidaan kolmella eri tavalla. Ensimmäisenä suoritetaan **ominaisuuden arvon asetus**, eli asetetaan elementin button väriksi punainen (red). Toisena toimintona on **metodin kutsuminen**.

Esimerkissä 6 suoritettavana metodina on funktio nimeltään SetText, joka asettaa elementtiin edit liittyvän tekstin arvoksi ActionText. Parametrit funktiolle annetaan erillisessä <param>-tunnisteessa. Parametrien määrityksessä voidaan käyttää apuna esimerkin mukaisen syntaksin lisäksi mm. <property>-, <content>- tai <reference>-tunnistetta (katso käyttö esimerkistä 5). Kolmantena esimerkissä 6 kuvataan **toiminnon laukaisu**, johon käytetään <event>-tunnistetta. Nyt <action>-tunnisteessa laukaistaan edit-elementin toiminto nimeltään TextChanged. Laukaistua toimintoa voidaan verrata toiminnan käynnistäneeseen ButtonClicked-toimintoon, ja myös toimintoon reagointi voidaan toteuttaa jälleen esimerkin esittelemällä kolmella eri tavalla.

2.5 Kuvauksien määrittelyt peers-tunnisteessa

<peers>-tunniste jakautuu kahteen osioon, <presentation>- ja <logic>-tunnisteisiin. <peers>-tunnisteen sisällä määritellään ne käytettävän alustan sisältämät elementit, joita käyttöliittymä käyttää. Käyttöliittymän käyttämät metodit tai funktiot, joita esim. olioista tai skripteistä kutsutaan, määritellään <presentation>-tunnisteen sisällä [Phanouriou, 2002, s. 39]. <logic>-tunnisteessa yhdistetään <call>-elementeissä käytetyt nimet ja luokat sovelluksen ulkoiseen toimintalogiikkaan.

2.5.1 Sanaston määrittely `presentation`-tunnisteessa

<presentation>-tunnisteessa määritellyn sanaston avulla yhdistetään UIML-dokumentissa käytetty abstrakti sanasto (*vocabulary*) käytetyn luokkakirjaston (*toolkit*) omaan sanastoon [Phanouriou, 2002, s. 66]. Käyttöliittymän kehittäjä päättää itse, käyttääkö hän valmista standardisoitua sanastoa vai määritteleekö itse haluamansa sanaston sisällön. Valmiita standardoituja sanastoja löytyy verkosta osoitteesta www.uiml.org/toolkits.

Valmiin sanaston käyttöönotto tapahtuu <peers>-tunnisteen alatunnisteessa <presentation> olevien base- ja source-määreiden avulla. Esimerkissä 7 kuvataan valmiin sanaston käyttöönottoa.

Esimerkki 7. Valmiin sanaston käyttöönotto.

```
<presentation
    base="http://www.cc.jyu.fi/~jankorho/UIVoc"
    source="MFC_1.0_JanneKorhonen_1.0.uiml#vocab"/>
```

Mikäli UIML-dokumentti sisältää esimerkkiä 7 vastaavan <presentation>-tunnisteen, niin kyseinen UIML-dokumentti käsitellään UIML-renderöintikoneistolla, joka käyttää sanastoa nimeltään UIVoc. Sanaston sijainti määritetään URL:n avulla. Valinnainen määre source ilmoittaa, että kyseisessä UIML-dokumentissa käytetyt luokkien ja elementtien ilmaisut voivat sijaita myös MFC_1.0_JanneKorhonen_1.0.uiml-nimisessä sanastossa. Liitteessä 1 on kuvattu osa käyttöliittymän UIFirst kehityksessä käytetystä sanastosta.

Uuden sanaston luonti tapahtuu UIML-dokumentissa tunnisteessa <template>. Sanasto siis tallennetaan omaksi tiedostokseen ja otetaan käyttöön varsinaisessa UIML-käyttöliittymädokumentissa. Seuraavassa esimerkissä on kuvattu sanaston käyttöönottoa <peers>-tunnisteessa. Kyseinen sanasto voitaisiin liittää esimerkiksi käyttöliittymään UIFirst yksinkertaisesti lisäämällä esimerkin <peers>-tunniste käyttöliittymän määritelleeseen UIML-dokumenttiin.

Esimerkki 8. Sanaston käyttöönotto <peers>-tunnisteessa.

```
<peers>
  <presentation base="C++_1.0_JK_1.0"/>
</peers>
```

Seuraavassa esimerkissä kuvataan uuden sanaston luontia, ja sen tärkeimmät tunnisteet <d-class>, <d-property> ja <d-param>.

Esimerkki 9. Sanaston sisältö.

```
<uiml>
 <template>
   <presentation>
     <d-class id="cEdit" used-in tag="part" maps-
     type="class" maps-to="CEdit">
        <d-property id="text" maps-type="SetMethod"
        maps-to="setText" return-type="int">
         <d-param type="string"/>
        </d-property>
      </d-class>
   </presentation>
  </template>
</uiml>
```

Sanaston määrittelyssä käytetään siis useita d-alkuisia tunnisteita. Kirjain d kuvaa tunnisteiden määrittelevyyttä (*define*) [Harmonia, 2002, s. 76]. <d-class>-tunnisteen määreessä id määritellään UIML-dokumentin <part>-tunnisteessa elementistä käytetty termi (vertaa esimerkin 3 arvo cButton) ja määreessä used-in-tag määritellään missä tunnisteessa kyseistä elementtiä käytetään. maps-type-määreen arvo kertoo minkä tyyppiseen kieleen sanasto viittaa.

Arvoksi voidaan antaa mm. `tag`, joka viittaa rakenteiseen määrittelykieleen kuten XML ja HTML, ja `class` komentopohjaiseen oliokieleen kuten C++ tai Java. `maps-type`-määreessä viitataan käytetyn ohjelmointikielen haluttuun luokkaan, esimerkissä 9 siis MFC:n `CEdit`.

Luokkien tapaan myös luokkien sisältämät ominaisuudet (*property*) tulee kuvata vastaavalla tavalla. Esimerkissä 9 luokka `CEdit` sisältää yhden ominaisuuden, nimeltään `text`. Ominaisuudet määritellään kuvaavassa `<d-property>`-tunnisteessa. Ominaisuuksille määritellään tarvittavat metodit, joita esimerkissä 9 on yksi, ominaisuuden asetusmetodi `setText`. Metodeille annetaan haluttujen parametrien tyypit `<d-param>`-tunnisteessa ja paluuarvon tyyppi `return-type`-määreessä.

2.5.2 Ulkoisten toimintojen määrittely `logic`-tunnisteessa

`<logic>`-tunnisteessa siis määritellään käyttöliittymän varsinaiselle sovellukselle aiheuttamat toiminnot. Varsinainen sovellus voi olla mm. jokin skriptiohjelmointikielellä ohjelmoitu komponentti tai paketti useita komponentteja, joita käyttöliittymästä kutsutaan [Harmonia, 2002, s. 85].

`<logic>`-tunnisteen tehtävänä on toimia ikään kuin liimana UIML:n ja sovelluksen välillä. Kun edellisessä kappaleessa määritelty *sanasto* yhdisti UIML:ssä käytetyn nimistön varsinaisiin luokkiin ja ominaisuuksiin, yhdistää `<logic>`-tunniste vastaavalla tavalla käytettävät ulkoiset komponentit ja näiden metodit käyttöliittymään. Esimerkissä 10 kuvataan `<logic>`-tunnisteen käyttöä.

Esimerkki 10. <logic>-tunnisteen käyttö.

```
<peers>
 <logic>
 <d-component name="cButton" maps-to="CButton">
     <d-method name="set" return-type="int"
     maps-to="SetWindowText">
       <d-param name="ButtonText" type="LPCTSTR"/>
     </d-method>
   </d-component>
 </logic>
</peers>
```

Esimerkissä 10 yhdistetään UIML-dokumentissa käytetty komponentin nimi cButton ulkoiseen, tässä tapauksessa MFC-luokkakirjaston, komponenttiin luokaltaan CButton. Yhdistämiseen käytetään <d-component>-tunnistetta esimerkissä 10 kuvatulla tavalla. Komponentille on määritelty käyttöön yksi metodi, UIML:ssa käytettävältä nimeltään set. Metodi yhdistetään <d-method>-tunnisteessa CButton-komponentin omaan metodiin SetWindowText.

Myös parametrit tulee määritellä <logic>-tunnisteen sisällä, tunnisteessa <d-param>. Käyttö on kuvattu yllä olevassa esimerkissä, kuten myös aikaisemmin esimerkissä 9.

2.6 Käyttöliittymän ajonaikainen muuntaminen

UIML tarjoaa mahdollisuuden myös käyttöliittymän ajonaikaiseen muuntamiseen. Muuntaminen tapahtuu <restructure>-tunnisteen avulla ja se voidaan asettaa suoritettavaksi tiettyjen ehtojen täyttyessä. Näinä ehtoina voivat olla muun muassa käyttäjän toimintojen seurauksena tapahtuneet muutokset ominaisuuksissa tai muut käyttöliittymässä tapahtuneet muutokset [Harmonia, 2002, s. 64]. Tämä tarkoittaa sitä, että <restructure>-tunniste voi sijaita UIML-dokumentissa ainoastaan <action>-tunnisteen sisällä (katso käyttö luvusta 2.4.4).

Käyttöliittymän ajonaikaiseen muuttumiseen liittyy käsite virtuaalinen käyttöliittymäpuu (*virtual UI tree*). Puun muodostavat aina kaikki UIML-dokumentin sisältämät käyttöliittymäelementit riippumatta siitä, näkyykö elementti käyttäjälle vai ei. Koska elementeillä on oma hierarkiansa ja tämä voi siis ajonaikaisesti muuttua, täytyy ne pitää yllä virtuaalisessa puurakenteessa.

Virtuaalipuun alkuperäinen rakenne muodostetaan UIML-dokumentista ja sen `<structure>`-tunnisteen sisällöstä. Elementtien lisääminen, poistaminen ja muokkaaminen tapahtuvat siis `<restructure>`-tunnisteen avulla, jonka syntaksi on kuvattu esimerkissä 11.

Esimerkki 11. `<restructure>`-tunnisteen syntaksi.

```
<restructure  at-part="[osion_nimi]"
              how="append|cascade|replace|delete"
              where="first|last|before|after"
              where-part="[osion_nimi2]"
              source="[templaten_sijainti]">
```

`<restructure>`-tunniste tarvitsee yleensä sisäänsä rungon (*body*), joka koostuu yhdestä `<template>`-tunnisteesta ja `<template>`-tunnisteen sisältämästä yhdestä `<part>`-tunnisteesta.

2.6.1 Parametrien arvot `restructure`-tunnisteessa

`<restructure>`-tunnisteen parametrien arvot muokkaavat virtuaalipuuta ja käyttöliittymää seuraavanlaisesti:

`how="delete"`: Poistaa parametrissa `at-part` annetun elementin ja kyseisen elementin koko alipuun. Kun arvona on `delete`, ei `<restructure>`-tunniste tarvitse erillistä runkoa ja tällöin attribuutteja `where`, `where-part` ja `source` ei voida käyttää.

how="replace": Korvaa parametrissa `at-part` annetun elementin `<template>`-tunnisteessa annetulla `<part>`-tunnisteen elementillä.

how="append": Liittää parametrissa `at-part` annettuun elementtiin `<template>`-tunnisteen sisältämän elementin. Liittäminen tapahtuu attribuuttien `where` ja `where-part` arvojen edellyttämällä tavalla (katso arvojen merkitykset alta).

how="cascade": Liittää tietyin ehdoin parametrissa `at-part` annettuun elementtiin `<template>`-tunnisteen sisältämän elementin. Liittämisen ehtona on, että parametrissa `at-part` annetulla elementillä ei ole samannimistä lapsielementtiä. Ehdon täyttyessä liittäminen tapahtuu attribuuttien `where` ja `where-part` arvojen edellyttämällä tavalla (katso arvojen merkitykset alta).

`where`-attribuuttia voidaan käyttää vain silloin kun `source`-attribuutti on määritelty ja attribuutin `how` arvona on joko `append` tai `cascade`. `where-part`-attribuuttia voidaan käyttää vain silloin kun attribuutin `where` arvona on joko `before` tai `after`.

where="first": Liittää `<template>`-tunnisteessa annetun elementin `at-part`-attribuutissa määritellyn elementin ensimmäiseksi lapseksi.

where="last": Liittää `<template>`-tunnisteessa annetun elementin `at-part`-attribuutissa määritellyn elementin viimeiseksi lapseksi.

where="before" ja attribuutti where-part=["osion_nimi"] on määritelty: Liittää `<template>`-tunnisteessa annetun elementin `at-part`-attribuutissa määriteltyyn elementtiin ennen `osion_nimi`-nimistä elementtiä.

where="after" ja attribuutti where-part=["osion_nimi"] on määritelty: Liittää `<template>`-tunnisteessa annetun elementin `at-part`-attribuutissa määriteltyyn elementtiin `osion_nimi`-nimisen elementin jälkeen.

3 Prosessimalli ja muut käytetyt tekniikat

Tässä luvussa kuvataan opinnäytetyössä käytettyä NPDI-prosessimallia ja kerrotaan C++-ohjelmointikielen STL-luokkakirjaston käytöstä. Luvussa kuvataan myös kehitetyssä sovelluksessa käytettävää MFC-luokkakirjastoa.

3.1 NPDI-prosessimalli

Seuraavassa kuvataan NPDI- prosessimallia (*New Product Development and Introduction Process*) lähteen [Honeywell, 1998] perusteella.

3.1.1 Prosessimalli

Prosessilla tarkoitetaan toimintojen kokoelmaa, joiden avulla pyritään saavuttamaan jokin tietty päämäärä, tuote tai palvelu [Wilson, 1999, s. 699]. Ongelmia ratkaistakseen yrityksen tulee yhdistää prosessia tukeva kehitysstrategia, käytettävät työkalut, työskentelymenetelmät ja prosessin tuotanto-osuus. Näiden ja varsinaisen prosessin yhdistelmää kutsutaan *prosessimalliksi* [Pressman, 1998, s. 31].

3.1.2 NPDI:n taustaa

NPDI on luotu erillisen prosessinkehitystiimin (*Process improvement team*) kehittämänä. Prosessimallin kehittäminen alkoi vuonna 1997, ja tarkoituksena oli luoda prosessi, joka olisi kenttäväen apuna niin uusia ohjelmistoja rakennettaessa kuin markkinoinnissakin. Työ alkoi vanhan jo olemassa olevan prosessimallin arvioinnilla, ja siitä syntyneen analyysin tuloksena päädyttiin kehittämään prosessimalli NPDI by Honeywell.

NPDI:stä kehittyi laaja-alainen, markkinakeskeinen prosessimalli, jossa integroituvat kaikki yleisimmät prosessimallissa tarvittavat osat. NPDI:ssa kiinnitetään erityisesti huomiota ohjelmistojen (*software*) ja laitteiston (*hardware*) yhteistoimintaan, tuotteistamiseen (*manufacturing*), markkinointiin (*marketing*) ja tuotteen jakeluun (*supply management*).

3.1.3 NPDI:n vaiheet

NPDI-ohjelmistoprosessi jakautuu viiteen eri vaiheeseen. Nämä vaiheet ovat nimeltään alustus- (*concept*), määrittely- (*definition*), kehitys- (*development*), vahvistus- (*validation*) sekä markkinointi- ja jakeluvaihe (*Market introduction & ship stage*). Prosessimalli on pääosin tarkoitettu keskisuurten ja suurten projektien hallintaan, ja näin ollen vaiheet ovatkin melko kaikenkattavia. Tässä opinnäytteessä toteutettava sovellus on NPDI:n mittakaavassa suhteellisen pieni, ja näin ollen NPDI ei kaikilta osiltaan ole soveltuva eikä täydessä laajuudessaan myöskään tarpeellinen projektin läpiviemiseen. Tässä luvussa kuvataankin nk. supistetun, aliprojekteille tarkoitetun NPDI-prosessimallin etenemistä ja vaiheita.

Alustusvaihe: Alustusvaiheessa tuotteen sisältö ja sen tarjoamat palvelut määritellään karkealla tasolla. Vaiheen tärkeimpänä tehtävänä on määrittää tuotteen toteuttamiseen tarvittavien resurssien tarve. Esimerkiksi OMT-prosessimalliin (*Object Modeling Technique*) verrattaessa keskitytään NPDI:ssä tekniikoiden sijasta enemmän muun muassa taloudellisiin ja projektinhallinnallisiin asioihin, kuten resursseihin ja rahoitukseen [Rumbaugh, 1991, s. 16].

Ensimmäisenä tämän vaiheen tehtävistä on alustustiimin (*Concept-team*) perustaminen. Alustustiimi määrittää korkeamman tason vaatimukset ja tuotteen tarpeellisuuden. Määrittelyn tuloksena alustustiimin johtaja (*Concept-team manager*) kirjoittaa alustusvaiheesta vakiomuotoisen dokumentin (*Concept business plan*), jonka tarkoituksena on tarkastella tuotteen tarpeellisuutta ja teknisiä lähtökohtia. Alustustiimin hyväksyttyä erillisessä alustusvaiheen loppupalaverissa (*Concept stage gate review*) dokumentaation, siirrytään prosessissa seuraavaan vaiheeseen.

Määrittelyvaihe: Määrittelyvaiheen tärkeimpänä tavoitteena on selvittää tuotteen sisältö tarkemmin, miettiä kuinka se parhaiten saavutetaan, valita projektin vaatimat resurssit ja luoda projektille aikataulu. Myös asiakkaiden tarpeet tulevan tuotteen tarjoamille palveluille kartoitetaan.

Määrittelyvaiheen aluksi perustetaan ydintiimi (*Core team*), jonka johtajana toimii yleensä alustustiimin johtaja. Ydintiimiin valitaan edustajat seuraavilta prosessin alueilta: markkinointi, kehitys, tuotteistus, jakelu ja testaus. Vaiheen tärkeimpinä dokumentaatioina syntyvät lopullinen vaatimusmäärittely (katso luku 4.1.2) ja suunnitteludokumentti (*Design document*). Vaatimusmäärittelyn ja suunnitteludokumentaation perustana ovat Honeywellissä NPDI-prosessimalliin kehitellyt dokumenttipohjat. Yleensä nämä dokumentit muodostaa kehitystiimi yhdessä asiakkaan kanssa. Määrittelyvaiheen dokumentaatio muodostuu, kun alustusvaiheen dokumentaatiota tarkennetaan ja muokataan yksityiskohtaisemmaksi. NPDI ei vaadi suunnittelijoilta mitään pakollisia määrittely- tai suunnitteluvaiheen kaavioita, kuten esimerkiksi OMT++:ssa määrittelyvaiheen käyttötapausdiagrammit [Haikala, 1998, s. 347], vaan kukin suunnittelija saa toimia omien tottumustensa ja tapojensa mukaisesti. Kaaviot ja dokumentaatio toteutetaan yleensä UML-standardin mukaisesti.

Ydintiimin hyväksyttyä erillisessä määrittelyvaiheen loppupalaverissa (*Definition stage gate review*) dokumentaation, siirrytään prosessissa seuraavaan vaiheeseen.

Kehitysvaihe: Kehitysvaiheen tarkoituksena on tarkentaa edellisessä vaiheessa tehtyä suunnitelmaa, aloittaa sovelluksen toteutus ja sen dokumentointi, kehittää projektin tulokselle tuotteistusprosessi ja luoda markkinointistrategia.

Kehitysvaiheen aluksi voidaan sovelluksesta luoda ensimmäinen prototyyppi. Sen tarkoituksena on luoda sovelluksen kehittäjille kuva sovelluksen lopullisesta rakenteesta ja tuoda esille mahdollisia huomiotta jääneitä, eri tekniikoista johtuvia ongelmia. Mikäli prototyypityksessä suunnitelmaan tehdään muutoksia tai mahdollisia uusia asioita ilmenee, viimeistellään tällöin myös suunnitteludokumentti. Tämän jälkeen sovelluksen kehittäminen jatkuu, ja tuloksena syntyy ensimmäinen, nk. beta-versio tuotteesta. Tässä vaiheessa testiryhmä valitsee joukostaan betatestaajat ja valittu ryhmä koulutetaan testaamaan kyseistä sovellusta.

Vahvistusvaihe: Vahvistusvaiheen tehtävänä on betatestauksen ja suunnitellun testausdokumentin avulla varmistaa, että tuotteen sisältö ja sen tarjoamat palvelut vastaavat alussa asetettuja laatu- sekä asiakasvaatimuksia. Näin pyritään saamaan varmuus siitä, että tuote on valmis asiakkaille jaeltavaksi.

Betatestauksen jälkeen korjataan testauksessa löytyneet epäkohdat. Vahvistusvaiheen tärkein dokumentaatio on sovelluksen dokumentointi. Tämä dokumentti on usein hyvin yrityskohtainen: joissain yrityksissä projekteista tehdään kattava projektidokumentaatio sisältäen myös projektihallinnolliset osat, joissain yrityksissä dokumentaatioksi kelpaa jo valmiina oleva suunnitteluvaiheen dokumentaatio. Toinen tärkeä vahvistusvaiheen dokumentti on tuotteen käyttöohje. Käyttöohjeisiinkin yrityksillä on usein omat valmiit formaattinsa.

Markkinointi- ja jakeluvaihe: Markkinointi- ja jakeluvaiheessa tuote julkaistaan tai annetaan asiakkaiden käyttöön. Vaiheeseen kuuluu myös asiakkaiden kouluttaminen ja tuotteen taloudellisen menestyksen havainnointi ja seuranta.

Markkinointi- ja jakeluvaiheen toteutus riippuu paljolti markkinoinnista vastaavan henkilön aktiivisuudesta. Normaaleissa aliprojekteissa tämä vaihe muodostuu melko kapeaksi, sillä projektin ja sovelluksen markkinoinnin hoitaa siihen tehtävään erikoistunut osasto. Tärkeimpinä tämän vaiheen seikkoina sovelluksen kehittäjien kannalta ovatkin projektin tarkka läpikäynti ja sen selkeä päättäminen. Läpikäynnin avulla pyritään oppimaan tehdyistä virheistä ja näin välttämään niitä tulevaisuuden projekteissa. Tehtyjen virheiden dokumentointi onkin erittäin tärkeä osa tätä vaihetta.

3.2 STL

Tässä luvussa kuvataan UIML-renderöintikoneiston kehityksessä käytettyä C++-ohjelmointikielen STL-kirjastoa (*Standard Template Library*) ja sen käyttöä tämän projektin osalta.

3.2.1 STL:n kuvaus

STL on osa ANSI/ISO:n kehittämää C++:n standardikirjastoa. STL sisältää suuren määrän valmiita malleihin perustuvia algoritmeja ja tietorakenteita. Erona muihin C++:n säilöluokkiin STL:n algoritmit ovat geneerisiä. Geneerisyydellä on pyritty lisäämään algoritmien yleiskäyttöisyyttä ja näin olleen saamaan komponenteista uudelleenkäytettäviä [Musser, 2001, s. 4-5].

UIML-renderöintikoneistoa kehitettäessä käytetään STL:stä hyväksi useita ominaisuuksia. Tallentamiseen ja nopeaan tiedonhakuun käytetään STL:n valmiita tietorakenteita ja tietorakenteiden apuna on STL:n ominaisuuksista käytetty funktio-objekteja, tavallisia operaattoreita ja iteraattoreita.

3.2.2 Käytetyt tietorakenteet

STL:n tärkeimpiä osia ovat sen sisältämät tietorakenteet (*containers*), iteraattorit sekä algoritmit. Lyhyesti näitä osia kuvaten voidaan sanoa, että algoritmit käyttävät iteraattoreita tietorakenteiden läpikäymiseen. Tässä opinnäytteessä STL:n tietorakenteista käytetään kuvauksia (`map` ja `multimap`) ja vektoria (`vector`).

map: tietorakenne `map` on pareista (avain, arvo) muodostuva lista. `map`:n käyttö tarjoaa ohjelmoijalle nopean tavan hakea tietoa avaimen mukaan. `map`:n sisältämät avaimet ovat yksilöllisiä, eli samassa `map`:ssa ei voi olla kahta samaa avainta [Stroustrup, 1997, s. 480]. Arvoksi `map`:iin voidaan tallentaa mitä tahansa C++-ohjelmointikielen tietotyyppiä olevia alkioita. Alkiot voivat olla myös luokkien tai tietueiden (*struct*) esiintymiä. Seuraavassa esimerkissä esitellään `map`:n käyttöä.

Esimerkki 12. map-tietorakenteen käyttö.

```
int main () {
  map<string, int> kengannumerot;
  string name;

  kengannumerot[“jussi”] = 46;
  kengannumerot[“kalle”] = 43;
  kengannumerot[“maija”] = 36;

  while(cin >> name){
     if (kengannumerot.find(name) != kengannumerot.end())
         cout << “Kengännumero henkilölle “ << name << “on“
         << kengannumerot[name] << ”\n”
  }

  map <string, int>::iterator it;
  for(it = kengannumerot.begin(); it != kengannumerot.end();
  ++it)
  {
         cout << it->first <<" : Kengän koko on "<<
         it->second << ”\n”
  }
  for (int i=0; i<kengannumerot.size(); i++){
       kengannumerot.erase(i);
   }
 }
```

multimap: multimap:n erottaa map:sta vain se, että samalla avaimella voi multimap:ssa olla useampi esiintymä [Stroustrup, 1997, s.480]. Tulevassa sovelluksessa tarvitut alustaminen, lisääminen, läpikäynti ja poisto tapahtuvat samojen funktioiden avulla kuin map:ssakin

vector: `vector` on talletusrakenne, johon voidaan (`map`:n tavoin) tallentaa minkä tahansa tietotyypin alkioita. Erona `map`:iin `vector`:ssa ei erikseen määritellä avainta, vaan sitä edustaa vektorin indeksi eli alkion järjestysnumero [Musser, 2001, s. 332]. Rakenteen käyttö on hyvin samankaltaista kuin `map`-tietorakenteella. Seuraavassa taulukossa on esitelty muutama `vector`-talletusrakenteen yleisimmin käytetty jäsenfunktio.

Funktio	Tarkoitus
`size()`	Palauttaa `vector`:ssa olevien elementtien lukumäärän.
`empty()`	Palauttaa `boolean` arvon `true` mikäli kyseisessä `vector`:ssa elementtien lukumäärä on 0, muulloin palauttaa `false`.
`push_back(T& x)`	Lisää `T`-tyyppisen elementin `x` `vector`:n viimeiseksi alkioksi.
`begin()`	Palauttaa iteraattorin (katso käyttö seuraavasta kappaleesta) `vector`:n alkuun.
`end()`	Palauttaa iteraattorin `vector`:n loppuun.
`clear()`	Poistaa `vector`:n kaikki elementit.

Taulukko 1. `Vector`:n yleisimmin käytettyjä jäsenfunktioita [Microsoft, 2002].

3.2.3 Funktio-oliot ja iteraattorit

Tässä luvussa kuvataan funktio-olion käyttöä STL-luokkakirjastossa ja kerrotaan iteraattoreiden käytön perusteet.

Funktio-olion käyttö: Funktio-olioksi (*function object*) eli funktoriksi [Stroustrup, 1997, s. 515] kutsutaan kokonaisuutta, johon voidaan sisällyttää parametreja arvon muuntamiseen ja/tai laskennan tilaan vaikuttamiseen.

C++-ohjelmointikielessä mikä tahansa tavallinenkin funktio täyttää nämä kriteerit, mutta niin tekee myös luokan tai tietueen olio, mikä kuormittaa funktion kutsuoperaattorin `operator()` [Musser, 2001, s. 183]. Tätä ylikuormitusta kutsutaan funktio-olioksi.

Funktio-olion käyttöä tavallisen funktio-osoittimen sijaan puoltaa usea seikka. Ensinnäkin funktio-olio on yleiskäyttöisempi ja näin ollen tarpeen mukaan helpommin muokattavissa. Toiseksi funktio-olio voi välittää ylimääräistä tarvittavaa tietoa mukanaan. Kolmas funktio-oliosta saatava hyöty on sen tehokkuus. Tehokkuus perustuu C++-ohjelmointikielen mahdollisuuteen asettaa haluttuja funktioita funktio-olioon ns. `inline`-funktioiksi [Musser, 2001, s. 187]. Seuraavassa esimerkissä on kuvattu funktio-olion käyttöä.

Esimerkki 13. Funktio-olion käyttö.

```
class Negaatio
{
public:
  int operator() (int luku) { return -luku;}
};

void Vaihto(int n, Negaatio & neg)
{
  int arvo = neg(n);
  cout << arvo;
}
int main()
{
  Vaihto(12, Negaatio() ); //tulostaa -12
}
```

Funktio-olion luonti tapahtuu siis ylikuormittamalla luokan operaattori `()`. Esimerkissä 13 on määritelty operaattori luokasta **`Negaatio`**.

Ylikuormituksessa määritellään kuormitettava operaattori `()` ja halutut parametrit (`int` **`luku`**) sekä runko (*body*) funktiolle: esimerkissä 13 funktio-objekti yksinkertaisesti palauttaa annetun luvun negaation. Nyt funktio-olio on luotu ja sen käyttö tapahtuu aliohjelmassa **`Vaihto`**-määritellyllä tavalla. Kun pääohjelmasta kutsutaan aliohjelmaa **`Vaihto`** saadaan tulosteeksi `-12`.

Iteraattoreiden käyttö: STL:ssä tietorakenteille on määritelty ns. iteraattorityyppi. *Iteraattorin* ideana on tarjota ohjelmoijalle joukko rakennetta läpikäyviä operaatioita. Erilaiset algoritmit vaativat erityyppisen iteraattorin, nyt esiteltävä ja tässä UIML-renderöintikoneistossa käytetty iteraattori on tyypiltään ns. syöttö-iteraattori (*input-iterator*). Muut tyypit ovat englanninkielisiltä nimiltään *output-*, *forward-*, *bidirectional-* ja *random-acces-iterator* [Stroustrup, 1997, s. 551]. Iteraattorit toimivat pitkälti C++-ohjelmointikielen osoittimien tavoin. Yleensä tietorakennetta käydään läpi iteraattorin avulla normaalisti silmukassa inkrementoiden [Musser, 2001, s. 33]. Iteraattorin käyttöä on kuvattu esimerkissä 12 tulostamalla kaikki **`kengannumerot`**-tietorakenteeseen talletetut arvoparit. Nyt iteraattoria `it` käytetään osoittimen tavoin. Iteraattoreilta vaaditaan osoittimen toimintojen lisäksi muunkinlaisia ominaisuuksia. Esimerkissä iteraattorille toteutetaan operaatiot `it++` (*postfix*), joka suuntaa iteraattorin `map`:n seuraavaan elementtiin ja erisuuruusvertailu `!=`. Muita iteraattoreille toimivia operaatioita ovat `*` (palauttaa paikan johon arvo on tallennettu), `++i` (*prefix*) ja muuttujien vertailu `==` [Stroustrup, 1997, s. 551].

3.3 MFC

Tässä luvussa kuvataan UIML-renderöintikoneiston muodostaman käyttöliittymän toteutukseen käytettävää MFC-luokkakirjastoa.

3.3.1 Mikä on MFC?

MFC (*Microsoft Foundation Classes*) on Microsoftin kehittämä C++-ohjelmointikielen luokkakirjasto Windows-ympäristöön. Se tarjoaa ohjelmoijalle monimuotoisen tuen toteuttaa haluamansa sovellus.

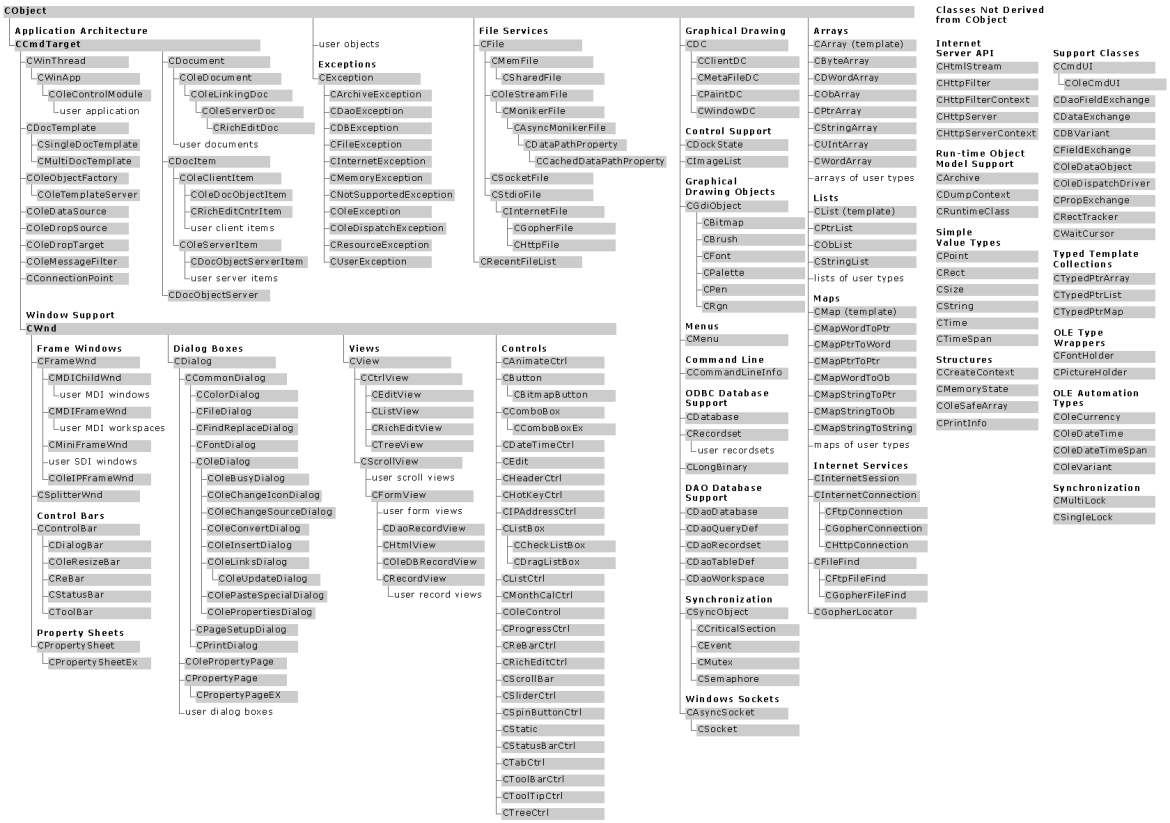
Ensinnäkin luokkakirjasto tarjoaa oliopohjaisen liitännän (*object-oriented wrapper*) Windowsin API:lle [Prosise, 1999, s.12]. Luokkakirjasto koostuu noin 200 luokasta, joita voidaan käyttää joko suoraan, tai niistä voidaan tarvittaessa C++:n keinoin (perintä tai virtuaaliset funktiot) muodostaa omia luokkia. Toisaalta MFC toimii myös sovelluskehyksenä (*Application framework*). MFC sisältää perusrakenteen sovelluksen toteutukselle, ja tarjoaa myös automaattisesti tuen useille Windowsin yleisimmille rutiineille [Prosise, 1999, s. 12]. MFC:n käyttöä puoltavia tärkeimpiä seikkoja ovat muun muassa:

- Windows-ohjelman toteuttamiseen vaadittavan työmäärän huomattava vähentyminen.
- Mitä tahansa Windowsin C-funktiota voidaan kutsua suoraan.
- Suorituskyky verrannollinen vastaavaan C-kieliseen ohjelmaan.
- C-kielisen ohjelman konvertointi MFC:lle on työmäärältään kohtuullinen.

3.3.2 MFC:n luokkien käyttö

MFC-sovelluksen kehitys perustuu siis luokkakirjaston käyttöön ja perinteiseen olio-ohjelmointiin. Ohjelmoija voi käyttää MFC:n tarjoamia luokkia joko suoraan tai luoda haluamansa luokat perintää käyttämällä. MFC tarjoaa luokkiensa perintään erinomaiset mahdollisuudet. Perinnän helppous johtuu välitettävien viestien reitityksestä.

Perinteisesti C-ohjelmointikielellä tehdyssä Windows API-sovelluksessa käytetyn luokan (elementin) synnyttämät viestit kulkevat aina sen luokan vanhemmalle (*parent*), joka käsittelee viestin ja suorittaa määrätyt toiminnot [Petzold, 1990, s. 206]. MFC:ssä luokka itse sisältää omat toimintonsa, jolloin se on myös itse vastuussa omista viesteistään ja niiden käsittelystä [Prosise, 1999, s. 13]. Tämä tekee perinnästä helpompaa, ja näin ollen myös uudelleenkäytettävyys (*reusability*) paranee. MFC:n sisältämistä yli 200 luokasta suurin osa on peritty joko suoraan tai epäsuoraan luokasta `CObject`. MFC:n luokkien väliset suhteet ja perinnät on esitetty kuvassa 3.



Kuva 3. MFC:n luokkahierarkia [Prosise, 1999].

MFC-pohjainen Windows-sovellus sisältää yleensä sovellusluokan, yhden tai useamman kehysikkunan (*Frame Window*) ja kaksi erillistä luokkaa, joita kutsutaan lomake- ja näkymäluokaksi (*document/view architecture*).

Sovellusluokan toiminta perustuu yleensä luokkaan `CWinApp`. Tämä luokka sisältää mm. funktion `WinMain`, jonka tehtävänä on kutsua MFC-olion valmiita jäsenfunktioita (mm. `InitInstance`). Sovelluskehys luo tähän sovellusluokkaan myös eräänlaisen viestisilmukan (*message loop*), joka toimii viestien välittäjänä, eräänlaisena rajapintana, sovelluksen ikkunoille [Microsoft, 2002].

Lomakeluokka peritään yleensä MFC:n luokasta `CDocument`. Lomakeluokka esittelee sovelluksen tietosisällön ja määrittelee tälle operaatiot. Periytyminen luokasta `CDocument` on sinänsä hämäävää, sillä lomakeluokka voi sisältää lähes minkälaista dataa tahansa [Prosise, 1999, s. 498].

Näkymäluokka periytyy luokasta `CView`. Näkymäluokka tarjoaa ikkunan lomakeluokassa sijaitsevaan tietoon. Näkymäluokka määrittelee sen, miten käyttäjä näkee tiedon, ja rajapinnan lomakeluokan tarjoamiin operaatioihin [Prosise, 1999, s.503].

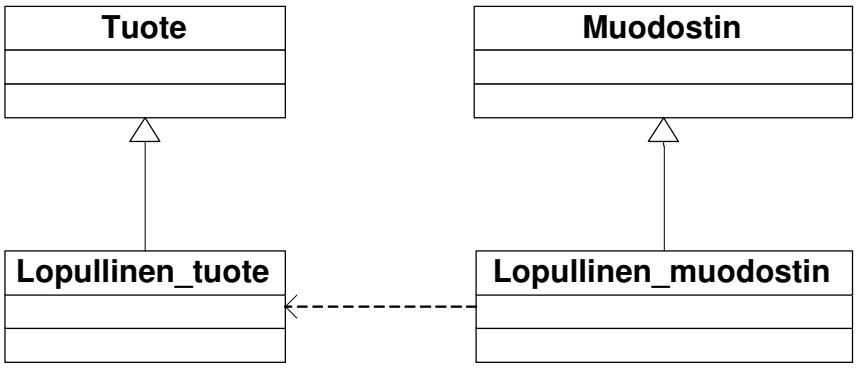
Kehysikkuna toimii sovelluksen pääikkunana ja se sisältää myös edellisessä kohdassa kuvatut näkymäluokan esiintymät. SDI (*Single Document Interface*) sovelluksessa lomakkeen kehysikkuna on samalla sovelluksen pääikkuna. MDI (*Multiple Document Interface*) sovellus voi sisältää pääkehysikkunan sisällä useita ns. lapsi-ikkunoita.

3.4 Muut käytettävät tekniikat

Tässä luvussa kuvataan muita opinnäytteessä kehitettävän sovelluksen luontiin tarvittavia tekniikoita, tehdasfunktiota ja DOM-mallia.

3.4.1 Tehdasfunktio

Tehdasfunktio (*factory method*) on dynaamisessa ohjelmoinnissa käytettävä suunnittelumalli. Tehdasfunktiota käytetään silloin, kun luotavan kohteen (*object*) sisältämät luokat määritellään vasta ajonaikaisesti [Agerbo, 1998, s. 137]. Kuvassa 4 on kuvattu tehdasfunktion rakennetta.



Kuva 4. Tehdasfunktio-suunnittelumallin rakenne[Agerbo, 1998].

Tehdasfunktio-suunnittelumallia käytetään tyypillisimmillään silloin, kun ei etukäteen tiedetä, minkä tyyppisen kohteen kanssa toimitaan. Suunnittelumallia käytettäessä syntyy rinnakkaiset luokkahierarkiat, jossa jokaista luokkaa (kuvassa 4 `Tuote`) kohden on oma tehdas (`Muodostin`). Eräs tyypillisimmistä tehdasfunktion käyttötavoista on ylikuormittaa esimerkiksi käyttöliittymäkirjaston yläluokan funktio `operator*()`, jolloin kaikki määritellyt ajonaikana tapahtuvat alaluokkien alustukset kulkevat tämän yläluokan muodostimen (*constructor*) kautta. Tällä tavoin voidaan ajon aikana luoda uusia elementtejä ilman tarkempaa luontiprosessin tuntemusta (esim. muodostimen parametrit) [Wick, 2001, s. 259]. Toinen saavutettava hyöty on ohjelmakoodin yläpidettävyyden paraneminen, uusia elementtejä lisättäessä tarvitsee vain määritellä lisätyn elementin vastaava funktio (`operator*()`) halutulla tavalla.

3.4.2 DOM

DOM (*Document Object Model*) on W3C:n hyväksymä standardi alusta- ja kieliriippumattomalle rajapinnalle, jonka avulla käsitellään dynaamisesti dokumenttien sisältöä, rakennetta ja tyylejä. DOM perustuu ajattelumalliin, jonka mukaan dokumentit ovat solmuista (*node*) koostuva hierarkkinen rakenne [Nentwitch, 2002, s. 152].

DOM:n käyttö perustuu sen tarjoamaan ohjelmointirajapintaan. DOM:n avulla esimerkiksi XML- tai UIML-dokumentista muodostetaan puurakenne, joka mahdollistaa dokumentin käsittelyn ohjelman sisäisesti. DOM tarjoaa mahdollisuudet myös tämän puurakenteen ohjelmalliseen muokkaamiseen. Dokumentin käsittelyn ja muokkauksen DOM toteuttaa lukuisien metodiensa avulla, esimerkkeinä näistä voidaan mainita muun muassa solmun lapsisolmujen listaus sekä solmujen lisääminen että poistaminen. XML-pohjaisille dokumenteille on kehitetty myös useita kuorikieliä (*core languages*, esim. XPath ja XLink), jotka tarjoavat lisää mahdollisuuksia XML-dokumentin ohjelmalliseen käsittelyyn [Nentwitch, 2002, s. 153]. Tässä opinnäytteessä kehitettävässä sovelluksessa luodaan UIML-dokumentista DOM-mallin mukainen, ja sitä käsitellään DOM:n ja XPath:n tarjoamien keinojen avulla.

4 Vaatimusmäärittely ja järjestelmän arkkitehtuuri

4.1 Sovelluksen vaatimusmäärittely ja käyttötarkoitus

Tässä luvussa kuvataan sovelluksen käyttötarkoitusta ja kerrotaan sovellukselle asetetuista vaatimuksista.

4.1.1 Sovelluksen käyttötarkoitus

UIML-renderöintikoneiston tarkoituksena on toimia apuvälineenä käyttöliittymien kehityksessä. Sovelluksen tärkeimpänä tehtävänä on luoda täsmälleen asiakaskomponentin (*client*) välittämien ominaisuuksien mukainen käyttöliittymä, tarjota luotu käyttöliittymä asiakaskomponentin käytettäväksi ja pitää yllä sen toiminnallisuutta koko elinkaaren ajan. Seuraavassa listassa on lueteltu syyt UIML-renderöintikoneiston kehittämiselle.

- Vähentää perinteisen käyttöliittymäohjelmoinnin osuutta sovelluksia kehitettäessä,
- edistää yrityksen sisällä luotavien käyttöliittymien yhtenäisyyttä,
- luoda käyttöliittymistä helpommin siirrettäviä eri alustojen välillä ja
- helpottaa muutosten tekemistä ja ylläpidettävyyttä.

4.1.2 Vaatimusmäärittely

Sovelluksen tai yleisemmin ohjelmiston vaatimusmäärittely sisältää informaation siitä, mitä kyseisen sovelluksen tulisi tehdä. Vaatimusmäärittelyssä ei oteta kantaa siihen, kuinka tämä toteutetaan [Booch, 1999, s .235]. UIML-renderöintikoneiston vaatimusmäärittely on jaettu kahteen osaan, *toiminnallisiin* ja *ei-toiminnallisiin* vaatimuksiin. **Ei-toiminnalliset** vaatimukset sisältävät informaation käytettävistä tekniikoista ja toiminta- ja kehitysympäristön asettamista rajoitteista. Ei-toiminnallisia vaatimuksia ovat:

- Ohjelmointikielenä käytetään C++:aa,
- käyttöliittymien määrittelykielenä käytetään UIML:ää,
- luotavan käyttöliittymän tulee muodostua MFC-luokkakirjaston tarjoamista elementeistä ja
- komponentin tulee kyetä palvelemaan vähintään kahta asiakasta samanaikaisesti.

Toiminnallisiin vaatimuksiin kuuluvat komponentin toiminnalta vaadittavat ominaisuudet. Toiminnalliset vaatimukset on määritelty seuraavassa taulukossa:

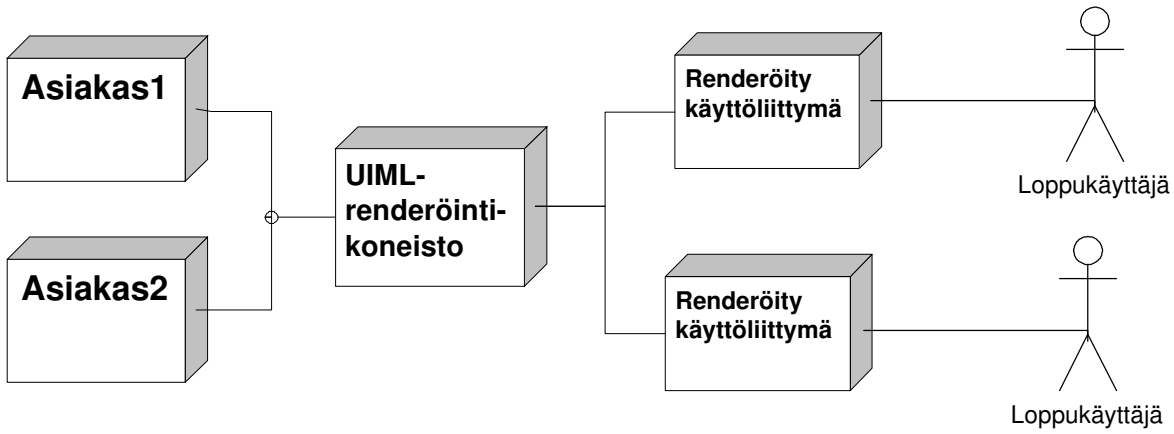
Vaatimus	Kommentti
UIML-renderöintikoneiston tulee olla komponentti, joka tarjoaa asiakkaille (*client*) kaksi rajapintaa: *dispatch*-rajapinta ja tapahtumarajapinta (*event-interface*). Asiakkaalla tarkoitetaan käyttöliittymää pyytävää asiakaskomponenttia.	*Dispatch*-rajapinta tarjoaa palvelut kuten `Renderöi(GUID,UIML-tiedosto)` ja vasteet käyttöliittymän komennoille. Tapahtumarajapinta välittää käyttöliittymän komennot asiakkaalle.
Yhdellä koneella voi olla käynnissä ainoastaan yksi esiintymä UIML-renderöintikoneistoa. Tämän esiintymän tulee palvella koneen kaikkia asiakkaita.	Tällä menettelyllä varmistetaan, että esim. käyttöliittymien toiminnallisuuksiin ei tule päällekkäisyyksiä.
Komponentin tulee kyetä jäsentämään (*parse*) ja validoimaan (*validate*) annettu UIML-dokumentti.	Dokumentin validointi ja syntaksin tarkistus toteutetaan erillisellä MSXML-jäsentäjällä.
UIML-renderöintikoneiston tulee tukea UIML-standardia.	UIML-renderöintikoneiston tulee tukea tiettyä erikseen määriteltyä joukkoa UIML:n toiminnoista.
UIML-renderöintikoneiston tulee tukea MFC-perustaista sanastoa.	Ensimmäinen UIML-renderöintikoneiston versio tarjoaa MFC-elementeistä koostuvan käyttöliittymän.
UIML-renderöintikoneiston tulee tukea erikseen määriteltyjä käyttöliittymäelementtejä.	Jäsentäjän tulee tukea käyttöliittymäelementeistä seuraavia: dialogi, ominaisuussivu (*property sheet*), HTML-näkymä (*HTML view*), nappula (*button*), lista- ja yhdistelmälaatikko (*listbox*), työkalupalkki (*toolbar*), statuspalkki (*statusbar*), tekstiruutu (*edit box*), vakioteksti (*static text*) ja valikot (*menu*).
UIML-renderöintikoneiston ensimmäinen versio käyttää UIML:n versiota 3.0.	Versio 3.0 on tällä hetkellä uusin.
Asiakkaan ja käyttäjän tulee pystyä kommunikoimaan keskenään.	Esimerkiksi käyttöliittymän valmistumisesta ja sulkeutumisesta tulee informoida asiakasta.

Taulukko 2. UIML-renderöintikoneiston toiminnalliset vaatimukset.

4.2 Järjestelmän arkkitehtuuri

Tässä luvussa kuvataan järjestelmän arkkitehtuuria ja tärkeimpiä sen toimintaan vaikuttavia ominaisuuksia.

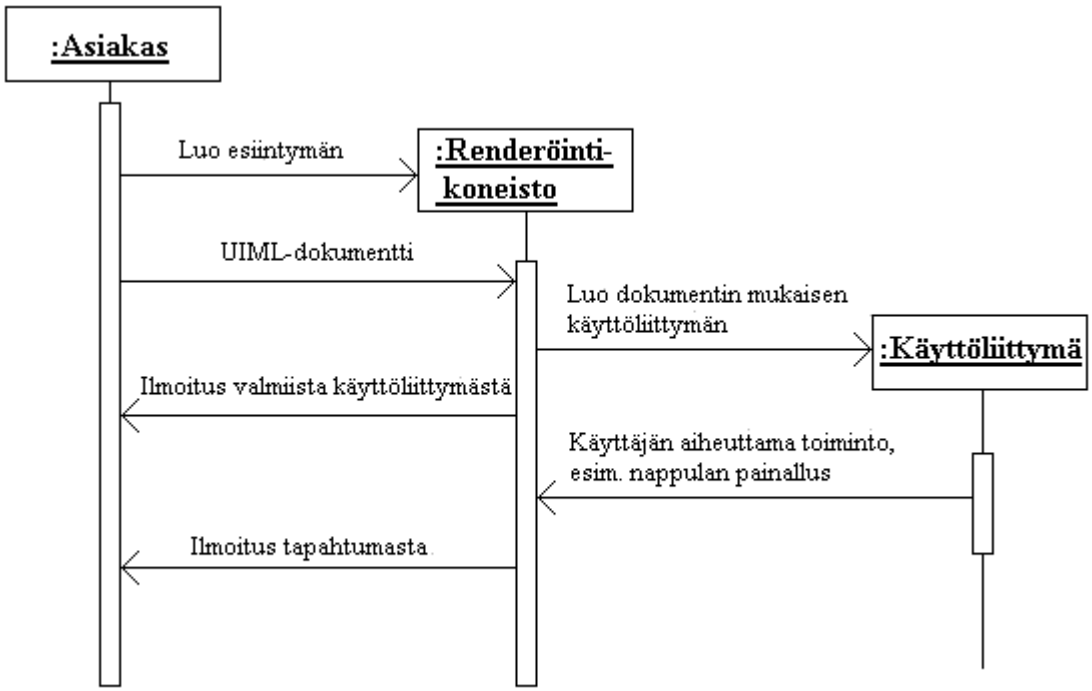
Järjestelmä UIML-renderöintikoneiston ympärillä voidaan korkealla tasolla jakaa neljään pääosaan: asiakaskomponentit (*client*), renderöintikoneisto, renderöintikoneiston luoma valmis käyttöliittymä ja käyttöliittymän loppukäyttäjä. Kuvassa 5 on kuvattu kyseinen järjestelmän arkkitehtuuri komponenttien rakennekaavion (*Deployment Diagram*) [Booch, 1999, s. 408] avulla.



Kuva 5. Järjestelmän arkkitehtuuri.

UIML-renderöintikoneisto toimii kullakin koneella nk. *singleton*-periaatteella, mikä tarkoittaa sitä, että komponentista luodaan vain yksi esiintymä, jota useammat asiakkaat käyttävät [Dung, 1998, s. 336]. UIML-renderöintikoneiston tapauksessa ensimmäinen tarvitseva asiakas luo siitä esiintymän, jota kaikki tarvitsevat asiakkaat tämän jälkeen käyttävät. Tällä menettelyllä varmistetaan luotujen käyttöliittymien identifiointi, eli käyttöliittymien toimintojen päällekkäisyyden vaaraa ei ole. Yksilöllistä käyttöliittymien identifiointia tarvitaan, koska kullakin asiakkaalla on halutessaan mahdollisuus luoda useampi kuin yksi käyttöliittymä.

Seuraavassa kuvassa on sekvenssikaaviona [Booch, 1999, s. 246] korkean tason kuva järjestelmän toiminnasta käyttöliittymää muodostettaessa ja loppukäyttäjän aiheuttamaa käyttöliittymän tapahtumaa välitettäessä.



Kuva 6. Käyttöliittymän luonti ja tapahtuman välitys.

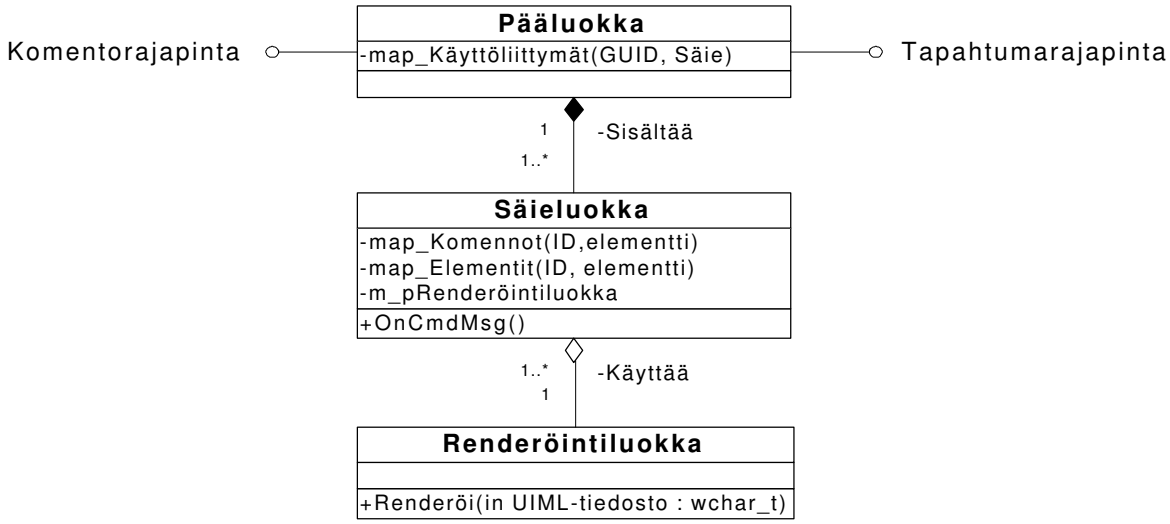
Käyttöliittymän luonti tapahtuu renderöintikomponentissa. Kun asiakkaalla ilmenee tarve käyttöliittymälle, sen ensimmäisenä tehtävänä on tarkistaa, onko renderöintikomponentista esiintymä jo luotuna, ja mikäli ei ole, luoda se. Esiintymän luotuaan asiakas toimittaa haluamansa käyttöliittymän tiedot UIML-dokumentissa UIML-renderöintikoneistolle. Tämä jäsentää dokumentin ja luo sitä vastaavan käyttöliittymän. Asiakkaalle lähetetään ilmoitus tästä ja samalla annetaan luotu käyttöliittymä asiakkaan käyttöön.

5 Sovelluksen toiminta ja pääosiot

Tässä luvussa kuvataan sovelluksen pääosiot ja niiden tarkempi toteutus. Sovelluksen toimintaa kuvataan pääluokittain kuitenkin siten, että kuvaamisen abstraktiotaso on melko korkea. Tämä tarkoittaa käytännössä sitä, että toteutettuun sovellukseen verrattuna oikea nimistö puuttuu, luokkien sisältöä ei käydä yksityiskohtaisesti läpi ja sisältö määritellään yleisluontoisemmin kuin pelkästään tämän sovelluksen osalta.

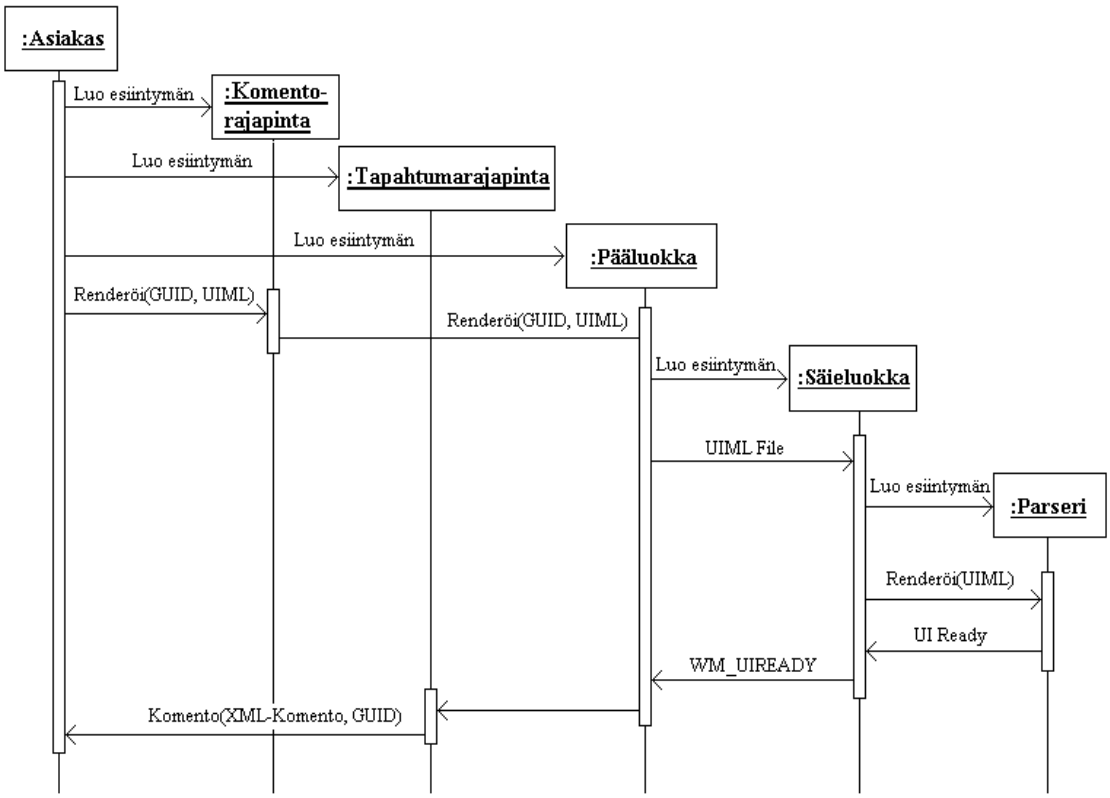
5.1 UIML-renderöintikoneiston pääosiot

UIML-renderöintikoneisto jakautuu useaan pääosioon. Nämä osiot ovat pääluokka, säieluokka, renderöintiluokka eli parseri ja kaksi erillistä COM-rajapintaa (osioiden tehtävät ja tarkempi toiminta luvuissa 5.2-5.5). Luokkien väliset suhteet ja rakenne on kuvattu luokkakaavion [Booch, 1999, s. 105] avulla kuvassa 7. Luokkien ominaisuudet (attribuutit) ja metodit on määritelty kuvassa 7 vain tässä luvussa tarvituilta osin. Arvo `Säie` pääluokan `map_Käyttöliittymät`-tietorakenteessa tarkoittaa säieluokan esiintymää ja säieluokan `map`-tietorakenteissa sijaitsevat `elementti`-arvot puolestaan luotuja MFC-käyttöliittymäelementtejä. Käyttöliittymäkohtaisina tunnisteina käytetään `GUID`-tunnisteita (*Globally Unique Identifier*). `GUID`-tunniste on 16-bittinen arvo, ja sitä voidaan erittäin suurella todennäköisyydellä pitää ainutlaatuisena [IEEE, 2002, s. 8].



Kuva 7. Renderöintikoneiston pääosioiden väliset suhteet ja rakenne kuvatuilta osin.

Renderöintikoneiston tärkein tehtävä on siis annetun UIML-dokumentin mukaisen käyttöliittymän luonti. Käyttöliittymän luontia ja sen aiheuttamaa renderöintikoneiston sisäistä toimintaa on kuvattu sekvenssikaavion avulla kuvassa 8.



Kuva 8. Renderöintikoneiston toiminta käyttöliittymää luotaessa.

5.2 Sovelluksen pääluokka

Tässä luvussa kuvataan sovelluksen pääluokan rakennetta, tehtäviä ja toiminnallisuutta.

5.2.1 Pääluokan rakenne

Sovelluksen pääluokka perustuu tekniikaltaan ja toteutukseltaan MFC-luokkakirjastoon (katso luku 3.3). Pääluokka sisältää oman toiminnallisuutensa lisäksi sovelluksen kaksi COM-rajapintaa, jotka on kuvattu luvussa 5.5 ja COM-tuen toteutuksessa ilmenneitä seikkoja kuvattu luvussa 6.1.

COM-tuesta johtuen MFC-pohjainen pääluokka on peritty myös useasta ATL-luokkakirjaston luokasta. Pääluokan tärkeimpiin ominaisuuksiin kuuluu myös sen ainutlaatuisuus, eli UIML-renderöintikoneiston singleton-ominaisuus on toteutettu tässä luokassa.

5.2.2 Pääluokan tehtävät

COM-rajapintojen toiminnallisuuden lisäksi on pääluokalle useita omiakin tärkeitä tehtäviä. Eräs pääluokan tärkeimmistä tehtävistä on **ottaa vastaan COM-rajapinnan kautta asiakkaan lähettämä UIML -tiedosto** ja **luoda tiedostosta muodostettavalle käyttöliittymälle oma käyttöliittymäsäie** (katso kuvaus luvusta 5.3). Säiettä (*thread*) luotaessa siirretään myös vastaanotettu UIML-tiedosto kyseiselle säikeelle tiedoston jäsentämistä ja käyttöliittymän luontia varten. Toinen pääluokan tärkeistä tehtävistä on **ylläpitää listaa luoduista käyttöliittymäsäikeistä** ja yhdistää ne kullekin käyttöliittymälle kuuluvaan käyttöliittymäkohtaiseen tunnisteeseen. `GUID`-tunnisteiden ja säikeiden yhdistäminen tapahtuu STL:n keinoin, luokan jäsenmuuttujana olevan `map`-tietorakenteen avulla (katso kuva 7, `map_Käyttöliittymät`).

Kolmas tärkeä pääluokan tehtävä on **toimia eräänlaisena ohjaajana** järjestelmän eri osien välillä. Pääluokka ohjaa käyttöliittymältä tulevat ns. ulkoiset viestit ja toiminnot oikealle asiakkaalle. Myös asiakkailta tulevat komennot ohjataan pääluokan kautta. Ohjaus tapahtuu juuri edellisen kohdan tietorakenteen `map_Käyttöliittymät` sisältämien tietojen perusteella.

5.2.3 Pääluokan toiminnallisuus

Tärkein ja suurin osa pääluokan toiminnallisuutta on viestien ohjaaminen niin asiakkaalta käyttöliittymäsäikeelle kuin säikeeltä asiakkaallekin. Myös UIML-renderöintikoneiston esiintymän käynnissäpito on pääluokan vastuulla. Pääluokka lähettää viestejä säikeelle yleensä asiakkaan pyynnöstä. Keskeisin ominaisuus tämän suuntaisessa viestiliikenteessä on kommunikoinnin asynkronisuus. Tällä tarkoitetaan tässä yhteydessä sitä, että uutta viestiä ei lähetetä ennen kuin edellinen viesti tai komento on varmasti mennyt perille ja kuitattu.

Erityisen tärkeää kommunikoinnin asynkronisuus on silloin, kun käyttöliittymää ollaan luomassa. Mikäli käyttöliittymä ei ole valmis ja asiakas lähettää sille viestin, tallettaa pääluokka asiakkaiden sanomat ja purkaa ne yksitellen käyttöliittymälle kun siltä on tullut varmistus valmiudesta ottaa viestejä vastaan. Myös käyttöliittymältä asiakkaalle tapahtuva viestiliikenne ohjautuu pääluokan kautta. Viestejä on kahdentyyppisiä, sekä sisäisiä (*internal*) että ulkoisia (*external*). Näistä vain ulkoiset saapuvat pääluokalle. Ulkoinen viesti voi olla esimerkiksi kutsu toiselle renderöintikomponentille jollain muulla tietokoneella tai viesti käyttöliittymän luoneelle asiakaskomponentille. Ulkoinen viestiliikenne tapahtuu viestin sisältävän erillisen XML-dokumentin avulla (katso liite 2). Asiakkaalta vaaditaan siis kyseisen XML-skeeman syntaksin tuntemus ja kyky osata käsitellä sitä.

Pääluokka huolehtii myös palvelimena toimivan renderöintikomponentin eliniästä. Kun kaikki renderöintikomponentin esiintymältä pyydetyt käyttöliittymät lopetetaan joko käyttäjän tai asiakkaan toimesta, on tärkeää, että myös renderöintikoneisto lopettaa tällöin itse itsensä. Pääluokan tulee huolehtia myös tilanteesta, jossa asiakaskomponentti lopettaa toimintansa. Tällöin kaikki kyseisen asiakkaan pyytämät käyttöliittymät tulee lopettaa, ja niiden varaamat resurssit vapauttaa.

5.3 Säieluokka

Tässä luvussa kuvataan säieluokan tehtäviä ja toteutusta. Säieluokka on, pääluokan tavoin, MFC-pohjainen. Säieluokka on peritty MFC:n säikeiden (*thread*) toteuttamiseen tarkoitetusta `CWinThread`-luokasta (katso kuva 3).

5.3.1 Säieluokan tehtävät

Säieluokan pääasiallisena tehtävän on tarjota infrastruktuuri UIML-renderöintikoneiston luomalle käyttöliittymälle. Jokainen luotava käyttöliittymä vaatii toimiakseen oman esiintymänsä säieluokasta. Toinen tärkeä säieluokan tehtävä on vastata sisältämänsä käyttöliittymän toiminnallisuudesta.

Toiminnallisuuden takaamiseksi sen tehtäviin kuuluvat käyttöliittymässä tapahtuvien toimintojen kiinniotto, niiden tunnistus ja tarvittaessa toiminnosta aiheutuvan tapahtuman laukaisu. Muita säieluokan tehtäviä ovat muun muassa listan pitäminen käyttöliittymässä olevista elementeistä (esim. nappula tai staattinen teksti) ja annetun UIML-dokumentin validointi (*validation*) ja muuntaminen `string`-muodosta DOM-muotoiseksi (DOM-malli kuvattu luvussa 3.4.2).

5.3.2 Säieluokan toteutus

Käyttöliittymän toiminnallisuudesta huolehtiminen on siis säieluokan vastuulla. Säieluokka ottaa kiinni käyttöliittymässä tapahtuneet toiminnot erillisellä `CWinThread`-luokan tarjoamalla, juuri tapahtumien tunnistamiseen tarkoitetulla `OnCmdMsg`-funktiolla (katso kuva 7). Kaikki käyttöliittymässä aiheutuneet toiminnot ohjataan tähän kyseiseen funktioon. Säieluokka sisältää jäsenmuuttujanaan STL:n listatietorakenteen, johon on UIML-dokumenttia läpikäytäessä sijoitettu kaikki kyseiselle käyttöliittymälle mahdolliset toiminnot. Tämä komentolista (`map_Komennot`) pitää sisällään tiedot toiminnon tunnisteesta ja toiminnon kohteena olevasta elementistä. Kun toiminto saapuu `OnCmdMsg`-funktioon, verrataan sen aiheuttajaa ja tunnistetta listan sisältöön. Mikäli kyseiseen toimintoon on tarkoitus reagoida, etsitään UIML-tiedostosta siihen yhdistetty toiminnallisuus (`<action>`-elementti) ja suoritetaan sen edellyttämät tehtävät (sisäiset tai ulkoiset).

Säieluokan muita tärkeitä ominaisuuksia ovat muun muassa lista kaikista käyttöliittymän elementeistä (`map_Elementit`), osoitin renderöintiluokan instanssiin ja saatu UIML-tiedosto DOM-muodossa. Käyttöliittymän elementit on talletettu listarakenteeseen UIML-dokumenttia läpikäytäessä. Tämä tallennus tehdään tehokkuussyistä, elementin haku ja tunnistus on nopeampaa `map`-tietorakenteesta kuin UIML-dokumentista.

5.4 Renderöintiluokka eli parseri

Tässä luvussa kuvataan renderöintiluokan tehtäviä ja toteutusta. Renderöintiluokka on toteutukseltaan pääluokan tavoin singleton-tyyppinen, eli sama instanssi luokasta palvelee kaikkia sitä tarvitsevia säikeitä.

5.4.1 Renderöintiluokan tehtävät

Renderöintiluokan tehtävänä on käydä läpi asiakkaan antama UIML-dokumentti. Renderöinti voidaan jakaa kahteen tärkeään osaan: itse käyttöliittymän luontiin ja toiminnallisuuden takaavan osan läpikäyntiin. Koko sovelluksen osalta tärkeä renderöintiluokan ominaisuus on, että kaikki renderöintikoneiston viittaukset UIML-määrittelykieleen ovat tässä luokassa.

Käyttöliittymää luotaessa renderöintiluokka käy läpi UIML-dokumentin `<interface>`-tunnisteen (katso luku 2.4). Tunnistetta läpikäydessään renderöintiluokka muodostaa halutut elementit ja sijoittaa ne tunnisteineen säieluokan jäsenmuuttujaan `map_Elementit` (katso luku 5.3.2).

Käyttöliittymän elementeille voidaan määritellä UIML-dokumentissa yleensä myös useita ominaisuuksia (mm. koko, väri, teksti). Nämä ominaisuudet asetetaan luotuihin elementteihin myöskin renderöintiluokassa käyttöliittymää luotaessa. Toinen renderöintiluokan tärkeä tehtävä UIML-dokumenttia läpikäytäessä on toiminnallisuuden takaaminen. Käyttöliittymässä tapahtuvat toiminnot, joihin reagoidaan, on määritelty UIML-dokumentissa `<behavior>`-tunnisteessa. Renderöintiluokka käy läpi nämä toiminnot ja asettaa ne kohde-elementteineen säieluokan jäsenmuuttujaan `map_Komennot`, jotta tarvittaviin tapahtumiin osataan säieluokassa reagoida.

5.4.2 Renderöintiluokan toteutus

Renderöintiluokka on siis singleton-tyyppinen luokka. Tämä tarkoittaa käytännössä sitä, että siitä on käynnissä yksi esiintymä kerrallaan, jota kaikki sitä tarvitsevat säieluokan esiintymät käyttävät.

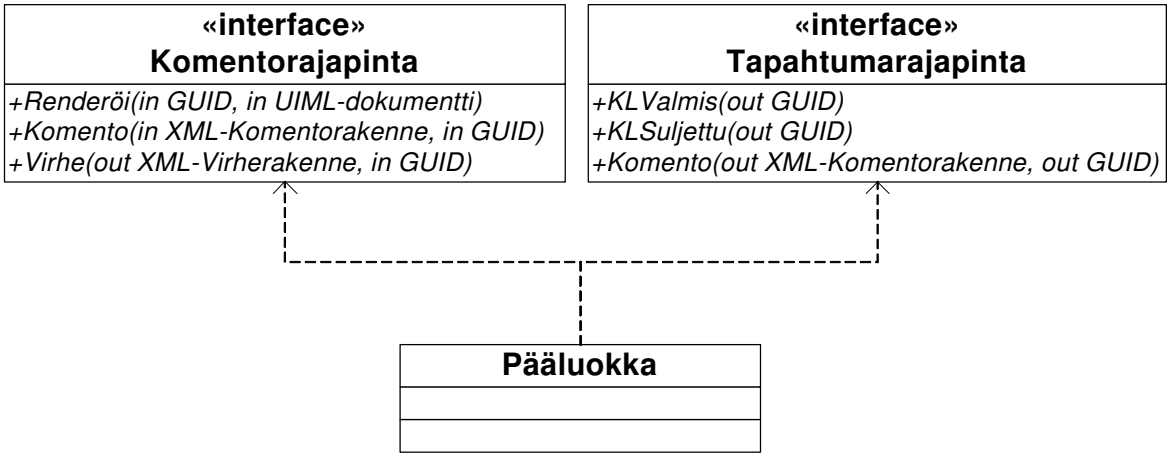
Ominaisuuden toteuttamisessa on käytetty apuna säieturvallista (*thread-safe*) toteutustapaa, jonka avulla taataan luokan käyttö vain yhdeltä säikeeltä kerrallaan. Säieturvallinen toteutustapa edellyttää säieluokan esiintymän aina renderöintiluokkaa kutsuessaan tarkistavan, onko renderöintiluokka vapaa vai käyttääkö joku muu sitä. Mikäli se on jo ennestään käytössä, jää sitä kutsuva esiintymä odottamaan luokan vapautumista. Näin voidaan varmistaa luotavan käyttöliittymän ja sen elementtien tunnisteiden ja toimintojen ainutlaatuisuus.

Renderöintiluokka sisältää useita alaluokkia. Renderöintiluokka on jaettu osiin useastakin syystä ja näistä tärkeimpänä jako mahdollistaa käytettävän suunnittelumallin, tehdasfunktion (*factory method*, kuvattu luvussa 3.4.1), helpomman ja tehokkaamman toteuttamisen. Vaikka renderöintiluokka sisältääkin useamman alaluokan, rajapintana säieluokkaan toimii pelkästään renderöintiluokka.

Toiminnallisuutenaan renderöintiluokka sisältää rekursiivisen silmukan, jonka avulla käydään läpi kaikki vastaanotetun UIML-dokumentin tunnisteet. Alaluokkien tehtäviksi jäävät siten muun muassa käyttöliittymäelementtien luonti, niiden ominaisuuksien asettaminen ja toiminnallisuuden purkaminen `map_Komennot`-tietorakenteeseen. Rekursiivisesta silmukasta kutsutaan kutakin alaluokkaa aina tarpeen mukaan.

5.5 COM-rajapinnat

Tässä luvussa kuvataan sovelluksen kahden ulommaisen rajapinnan tehtävät. Sovelluksen kaksi ulointa rajapintaa ovat tehtäviensä mukaan nimettyinä komentorajapinta (*command interface*) ja tapahtumarajapinta (*event interface*). Rajapinnat ja niiden tärkeimmät metodit on kuvattu rajapintakuvassa 9.



Kuva 9. Rajapinnat ja niiden metodit.

Tekniikaltaan molemmat rajapinnat ovat COM-rajapintoja, ja näiden toteuttamisesta tähän sovellukseen ja yleisemminkin MFC-komponenttiin on kerrottu luvussa 6.1. Myös COM-toteutuksen teknistä puolta on kuvattu kyseisessä luvussa, joten tässä luvussa keskitytään rajapintojen tehtäviin. Rajapinnat on tarkoituksella pyritty tekemään yksinkertaisiksi ja metodimääriltään pieniksi. Näillä ominaisuuksilla on renderöintikomponentista yritetty luoda mahdollisimman helppokäyttöinen.

Komentorajapinnan pääasiallisena tehtävänä on toimia asiakaskomponenttien viestien ja toimintojen välittäjänä renderöintikoneistolle. Komentorajapinnan kautta asiakas luo yhteyden renderöintikoneistoon. Yhteys luodaan yksinkertaisesti kutsumalla rajapinnan metodia `Renderöi` ja antamalla tälle parametreiksi halutun käyttöliittymän sisältävä UIML-dokumentti ja asiakkaan luotavalle käyttöliittymälle luoma `GUID`-muotoinen tunniste. Mikäli renderöintikoneisto ei ollut kyseisellä koneella vielä käynnissä, käynnistyy se nyt myös muiden asiakkaiden käytettäväksi. Tämän yhteyden synnyttävän ja käyttöliittymän synnyttämisen käynnistävän metodin lisäksi kaikki muut välitettävät komennot hoidetaan komentorajapinnassa `Komento`-metodin avulla. Tällä metodilla on parametreinaan XML-dokumentti (liite 2) ja käyttöliittymän `GUID`-muotoinen tunniste. Metodilla välitetään säikeelle esimerkiksi käyttöliittymän uudelleenmuokkauskomento (`<restructure>`-tunniste, katso luku 2.6).

Kolmas komentorajapinnan metodi `Virhe` on tarkoitettu virhetietojen välittämiseen asiakkaalle. Tämän avulla asiakas voi halutessaan pyytää käyttöliittymän luonnissa tapahtuneet virheet XML-muodossa (Katso liite 3).

Tapahtumarajapinnan pääasiallisena tehtävänä on käyttöliittymässä tapahtuvista toiminnoista tiedottaminen asiakkaille. Yleisimmille toiminnoille on kullekin tapahtumarajapintaan toteutettu oma metodinsa. Näitä toimintoja ovat muun muassa ilmoitus siitä, että käyttöliittymä on valmis tai ilmoitus käyttöliittymän sulkeutumisesta. Muista tapahtumista tiedottaminen hoidetaan liitteen 2 mukaisen XML-dokumentin avulla.

6 Tekniikoiden ja sovelluksen arviointia

Tässä luvussa arvioidaan opinnäytteessä kehitetyn sovelluksen lopputulosta sekä arvioidaan tekniikoiden yhteensopivuutta ja prosessimallin toteutumista käytännössä. Luvun alussa on kuvattu vaatimusmäärittelyn mukaisessa COM-tuen toteuttamisessa ilmenneitä ongelmatilanteita ja niiden ratkaisuja.

6.1 COM-tuen toteuttaminen

Sovelluksen vaatimusmäärittelyn mukaan MFC:lla toteutetun pääluokan (kuvattu luvussa 5.2) tulee tarjota asiakkaille COM-rajapinta (*Common Object Model*). Periaatteessa tämän COM-tuen toteuttaminen on mahdollista kahdella eri tavalla: perinteisesti moniperinnän (*multiple inheritance*) avulla tai ns. sisäkkäisten luokkien (*nested classes*) avulla [Li, 1998, s. 91]. Koska pääluokka on toteutettu MFC:llä, oli myös COM-tuki aluksi tarkoitus toteuttaa MFC:n tarjoamalla tavalla (luku 6.1.1). Kuitenkin sen puutteista ja heikkoudesta johtuen jouduttiin sille etsimään vaihtoehtoinen toteutustapa ja COM-tuki päädyttiinkin toteuttamaan ATL-luokkakirjaston avulla (luku 6.1.2). Tässäkin yhteensovittamisessa ilmeni ongelmakohtia, joita on kuvattu luvussa 6.1.3.

6.1.1 COM-tuen lisääminen MFC:n keinoin

MFC käyttää tuen toteuttamiseen sisäkkäisten luokkien tarjoamaa mahdollisuutta [DiLascia, 1999, s. 31]. COM-tuen toteuttamiseen MFC-pohjaiseen ohjelmistokomponenttiin on ohjelmointiympäristöissä olemassa valmis ratkaisu, nk. wizard eli automaattinen koodigeneraattori, jolla kyseinen tuki luokalle tai komponentille luodaan.

MFC:n valmiiksi tarjoamassa ratkaisussa ilmeni tässä opinnäytteessä kehitettävän sovelluksen kannalta puutteita. Ensinnäkin MFC:n käyttämä sisäisiin luokkiin perustuva menetelmä tekee **uusien luokkien lisäämisestä rajapintaan erittäin työlästä**. Tästä johtuen ohjelmakoodin päivittäminen ja myöskin komponentin ylläpito hankaloituu.

Työläästi päivitettävän sisäisiä luokkia käyttävästä sovelluksesta tekee COM-rajapinnan ominaisuus, joka vaatii jokaiselle rajapintaluokalle ainakin luokan `IUnknown`-metodit toteutetuiksi (katso luku 6.1.2). Uusia luokkia lisättäessä nämä joudutaan siten aina toteuttamaan erikseen. Toinen MFC:n valmiiksi tarjoamassa COM-tuessa ilmenevä heikkous on, että sisäkkäisten luokkien käyttämisestä **saadut edut eivät ole tarpeellisia** tässä sovelluksessa.

Sisäkkäisten luokkien käytössä suurimpana etuna verrattuna moniperintään on se, että silloin tuetaan COM-rajapinnassa kahta tai useampaa samannimistä metodia [Lam, 1996b, s. 334]. Moniperinnässähän kahden tai useamman perittävän luokan sisältäessä samannimisen ja parametreiltaan yhtäläisen metodin, syntyy niistä vain yksi rajapinnan metodi [DiLascia, 1999, s. 35]. Tässä sovelluksessa tämä ei kuitenkaan ole ongelma.

Verrattaessa näitä MFC:llä toteutetun COM-rajapinnan ominaisuuksia ja heikkouksia vastaavaan ATL:lla toteutettuun COM-rajapintaan, päädyimme siihen, että tämän sovelluksen osalta on ATL-luokkakirjaston käyttö tarpeellista ja perusteltua. Tähän mielipiteeseen vaikuttivat myös aiemmat kokemukset ATL:n käytöstä ja monipuolisuudesta COM-rajapintoja kehitettäessä. Seuraavassa luvussa on kerrottu COM-tuen toteuttamisesta ATL:n avulla.

6.1.2 COM-tuki ja sen toteuttaminen ATL:n avulla

Tässä luvussa kerrotaan lyhyesti ATL-luokkakirjastosta (*Active Template Library*) ja sen avulla toteutetuista COM-komponenteista (*Component Object Model*). Näistä tekniikoista kuvataan tässä opinnäytteessä kehittävässä sovelluksessa tarvittavat asiat.

Tässä opinnäytteessä kehitetyn sovelluksen kaksi rajapintaa ovat COM-standardin mukaisia. Rajapinnat päädyttiin toteuttamaan COM-standardin mukaisesti, koska se on pitkälti kehitetty toimimaan juuri ajonaikaisessa (*runtime*) ympäristössä. Toinen tärkeä peruste COM-standardin mukaisen rajapinnan käyttöön on sen tarjoama mahdollisuus paikkariippumattomuuteen (*location transparency*) [Grimes, 1998, s. 10].

COM-komponentit toteutetaan nykyään hyvin yleisen tavan mukaisesti ATL-luokkakirjaston avulla. ATL-luokkakirjastoa päädyttiin käyttämään kahdesta syystä: se tarjoaa erinomaiset *valmiit toteutukset tarvituille rajapinnan palveluille* ja ATL on tarkoitettu *palvelintyyppisten sovellusten kehittämistä varten* [Lam, 1996a, s. 233].

COM-rajapinnan tarvitsemista palveluista tärkeimmät tarjoaa ATL:n luokka `IUnknown`. Itse asiassa COM-komponentin ainoa pakollinen ominaisuus on, että `IUnknown`-luokan metodit on toteutettu [Rector, 1999, s. 95]. ATL siis sisältää valmiit käyttökelpoiset toteutukset näille pakollisille metodeille. Muut tässä opinnäytteessä kehitetyssä sovelluksessa COM-rajapinnan luontiin käytetyt ATL:n luokat ovat `CComObjectRootEx`, `CComCoClass`, `ISupportErrorInfo`, `IConnectionPointContainerImpl` ja `IDispatchImpl`. Näiden avulla komponentti muun muassa on säieturvallinen (*thread-safe*) [Rector, 1999, s. 98] ja siinä tuetaan erilaisia virheviestien muotoja (mm. *rich error messages*) [Rector, 1999, s. 487].

ATL tarjoaa myös valmiit työkalut palvelintyyppisen komponentin toteuttamiseen. COM-komponentin on siis tarkoitus toimia ns. COM-palvelimena. COM-palvelin tarkoittaa dynaamisesti aktivoitavaa joko yhtä tai kokoelmaa useammasta COM-luokan toteutuksesta. Tämä sovellus tulee toimimaan ns. paikallisena palvelimena (*local server*) ja palvelimena tulee toimimaan yksi suoritettava (*executable*) ohjelma (*EXE*) [Rector, 1999, s.157-158]. COM-palvelimilta vaadittaville ominaisuuksille, kuten rekisteröinnille ja palvelimen elinkaaren hallinnalle, löytyy vaadittavat valmiit toteutukset ATL-luokkakirjastosta. Oman rekisteröintinsä palvelin suorittaa itse. Rekisteröinnissä käytetään hyväksi ”perinteisiä” komentokielen komentoja `RegServer` ja `UnRegServer` [Rector, 1999, s.159]. Myös elinkaarensa hallinnasta palvelin huolehtii itse. Tämä ilmenee käytännössä siten, että kun palvelin huomaa että yksikään asiakas ei sitä enää käytä, lopettaa palvelin itse itsensä.

6.1.3 Ongelmatilanteita MFC:n ja ATL:lla toteutetun COM-komponentin yhteensovituksessa

MFC:lla toteutetun pääluokan ja ATL:lla toteutettujen COM-rajapintojen yhteensovittamisesta ilmeni useita pieniä yhteensovittamisongelmia. Ongelmia tosin osattiinkin odottaa, sillä ATL:n käytön perustuminen moniperintään (*multiple inheritance*) ja MFC:n käytön perustuminen yksittäisperintään (*single inheritance*) tulisivat väistämättä tuottamaan ongelmia tekniikoita yhdistettäessä [Lam, 1996b, s. 333]. Ongelmia tuottivat mm. säieturvallisuus, MFC-luokkien eliniän epävakaus ja ATL:n ja MFC:n samannimiset määrittelyt eri asioille.

Säieturvallisuudessa ilmenneet ongelmat johtuivat MFC:n ja ATL:n eroista. ATL:stä perityissä luokissa ei säieturvallisuus ole automaattisena ominaisuutena, kun taas STA-tekniikan avulla (*Single Threaded Apartment*) luoduista MFC-luokista ominaisuus löytyy [DiLascia, 1999, s. 44]. Erona oletusarvoissa on tällöin se, että MFC:ssä säikeeseen ja sen ominaisuuksiin ei voida vaikuttaa toisesta säikeestä, kun taas ATL:ssä tämä on mahdollista. Ongelmia saattaisi ilmetä myös COM-rajapinnan käytön synkronoinnissa, varsinkin silloin, kun komentorajapintaa kutsutaan lyhyen ajan sisällä useasta asiakaskomponentista. Näiltäkin ongelmilta kuitenkin vältyttiin STA-tekniikan avulla. STA:n parametreilla luotu COM-komponentti on säieturvallinen ja näin ollen soveltuva käytettäväksi renderöintiluokan toteutuksessa. Synkronointiongelman ratkeaminen perustuu STA:n ominaisuuteen, jonka mukaan COM-rajapinta jakaa rajapinnan käytön yhdelle asiakaskomponentille kerrallaan [Platt, 1997, s. 17].

MFC:n ja ATL:n yhteiskäytössä ilmeni ongelmia myös muuttujien nimeämisessä. Käännösongelmat johtuivat ATL:n ja MFC:n päällekkäisyyksistä, samalla nimellä tarkoitettiin kussakin tekniikassa aivan eri asiaa. Tämä ilmeni esim. käännösvaiheessa lukuisina virheinä ja varoituksina. Päällekkäisyyksiä löytyi lopulta viisi kappaletta, niin metodien kuin muuttujienkin nimistä.

Kaikki päällekkäisyydet liittyivät viittausten (*reference*) lukumäärän laskemiseen. Ongelmista päästiin eroon käyttämällä esikääntäjää (*preprocessor*). Tämän avulla nimettiin ilmenneet päällekkäisyydet uudelleen ennen varsinaista sovelluskoodin kääntämistä. Käytännössä uudelleennimeäminen toteutettiin C++:n `#define`-makron avulla.

6.2 MFC dynaamisessa käyttöliittymän kehittämisessä

Tässä luvussa arvioidaan MFC-luokkakirjastoa osana dynaamista käyttöliittymäkehitystä. Luvussa kerrotaan myös MFC:n ominaisuuksista säieohjelmoinnissa. Vertailupohjana luokkakirjaston vaihtoehtoja tutkittaessa on käytetty TrollTechin kehittämää Qt-käyttöliittymäkirjastoa. Qt-käyttöliittymäkirjasto on lähdekoodiltaan vapaata, ja tarjoaa saman ohjelmointirajapinnan eri käyttöjärjestelmille (esim. Windows, Unix, Linux ja Mac OS) [Petreley, 2002, s.54]. Vertailupohjana on käytetty myös tekijän aikaisempia kokemuksia käyttöliittymäohjelmoinnista.

Jo UIML-renderöintikoneistoa suunniteltaessa ja sen toteuttamiseen käytettäviä tekniikoita valittaessa osattiin aavistella, että erittäin ratkaisevassa osassa tulisi olemaan käytettävä käyttöliittymäkirjasto. MFC-luokkakirjastoon päädyttiin, koska sen tiedettiin tarjoavan mahdollisuudet muihinkin tarkoituksiin kuin pelkkänä käyttöliittymäkirjastona olemiseen. Näin myös sovelluksen osista pää- ja säieluokka päätettiin toteuttaa MFC:n avulla.

Heti sovellusta suunniteltaessa ja viimeistään kehityksen alussa huomattiin, että MFC:n käyttö vaatii uudelta käyttäjältä paljon kirjaston luokkien ja ominaisuuksien opettelua. Seuraavissa luvuissa on kuvattu MFC:n vahvuuksia ja heikkouksia osaltaan myös luokkakirjaston helppokäyttöisyyden ja opittavuuden kannalta. Varsinkin heikkouksien osalta monet ominaisuudet ovat MFC:llä varmasti saavutettavissa, mutta asian toteuttaminen saattaisi vaatia kohtuuttoman paljon opettelua ja työtä.

6.2.1 MFC:n heikkouksia

MFC vaatii monilta osin käyttäjältään melko paljon kokemusta luokkakirjaston käytöstä. Ehkä parhaiten kokemattomuus ilmeni käyttöliittymäelementtejä dynaamisesti luotaessa. Tällöin toteuttaja tarvitsee useissa tapauksissa lukuisia parametreja alustuksiin, koska oletusarvoisesti MFC ei näitä tarjoa. Usein nämä parametrit ovat jotain MFC:n omia tietorakenteita (esim. C++:n tietueita).

Esimerkiksi yksinkertaisen dialogi-ikkunan luonti luokan `CDialog` avulla vaatii ensimmäiseksi parametrikseen viitteen resurssiin, joka täytyy dynaamisessa käyttöliittymän kehittämisessä luoda erikseen (katso esimerkki 14). Tämän tiedoston sisältö on tarpeettoman hankala ja työläs toteuttaa, verrattuna esimerkiksi Qt-käyttöliittymäkirjaston vastaavan `dialog`-elementin toteutustapaan (esimerkki 14).

Esimerkki 14. `Dialog`-elementin luonti MFC:n ja Qt-luokkakirjaston avulla [Prosise, 1999] ja [Trolltech, 2000].

```
// MFC:
CDialog dialog;
dialog.Create(LPCTSTR lpszTemplateName, CWnd* pParentWnd =
              NULL )
// Qt:
QDialog(QWidget * parent = 0, const char * name = 0,
         bool modal = FALSE, WFlags f = 0 )
```

Samankaltaisia tilanteita ilmeni useasti käyttöliittymäelementtejä alustettaessa. Tämä johtunee pitkälti siitä, että elementtejä on pääasiallisesti tarkoitus luoda nk. velhojen (*wizard*) avulla, jolloin alustukset hoitaa käytetty toteutusympäristö. Dynaamisessa luonnissa velhojen käyttö ei ole mahdollista.

Toinen tilanne, missä MFC:n dynaamisessa käytössä ilmeni ongelmia, oli metodien ylikuormittaminen. Ylikuormittaminen on dynaamisessa käytössä mahdotonta, jolloin haluttujen ominaisuuksien toteuttaminen metodeihin jouduttiin tekemään luokkien perinnän avulla. Perintää jouduttiin käyttämään niin käyttöliittymäelementtejä luotaessa kuin viestimekanismia kehitettäessäkin. Myös tällöin MFC:n soveltumattomuus dynaamiseen käyttöliittymäohjelmointiin ilmeni lukuisina parametreina ja hankalina rakenteina. Säieluokan (`CWinThread`) tapahtumien kiinniottamiseen tarkoitettu metodi ei toiminut monisäikeisessä sovelluksessa ennen kuin luokka oli peritty täysin uudeksi luokaksi ja metodi ylikuormitettu omalla toteutuksella. Ylikuormittamisen puutteesta johtuvat ongelmat pystytään siis korvaamaan perinnän avulla, mutta tämä tapa on erittäin työläs.

Myös MFC:n luokkahierarkian ja riippuvuuksien syvempi tuntemus olisi ollut monessa tapauksessa tarpeellista. Useat käyttöliittymäelementit vaativat toimiakseen toisia MFC-luokkien alustuksia, esimerkiksi ominaisuuslomaketta (`CPropertySheet`) ei voi alustaa tyhjäksi, sen täytyy sisältää vähintään yksi luokan `CPropertyPage`-esiintymä, joka taas tarvitsee useita hankalia parametreja ja muuttujia toimiakseen.

6.2.2 MFC:n vahvuuksia

Dynaamisessa käyttöliittymien kehittämisessä erittäin ratkaiseva tekijä on viestimekanismin toiminta. MFC tarjoaa toimintojen ja viestien kiinniottamiseen hyvät mahdollisuudet. Useat MFC:n luokat sisältävät erillisen viestisilmukan, johon kaikki tapahtumat ohjautuvat ja jossa ne on helppo jakaa tehtävänsä ja toimintansa mukaisesti. Kun alun hankaluuksista (katso luku 6.2.1) viestimekanismin kehityksessä dynaamisesti luotuun käyttöliittymään päästiin, toimi perinnän avulla toteutettu viestisilmukka tarkoituksensa mukaisesti. Esimerkiksi Qt-luokkakirjastoon verrattuna MFC:n tarjoamat mahdollisuudet ovatkin tässä suhteessa käyttökelpoisempia. Qt-luokkakirjaston avulla luodussa dynaamisessa käyttöliittymässä viestit tulee ottaa kaikki yksitellen kiinni ja näin määritellä toiminta jokaiselle viestille erikseen. MFC:ssä erillinen viestisilmukka hoitaa viestin kiinnioton ja siihen voidaan reagoida vain tarvittaessa. UIML-renderöintikoneiston tapauksessa viestisilmukkana toimii `CWinThread`-luokan metodi `OnCmdMsg`.

Myös MFC:n ominaisuuksia säieohjelmoinnissa voidaan pitää sen hyvänä puolena. Varsinkin UIML-renderöintikoneiston toimintaperiaatteiden ja luokkahierarkian vaatima säieturvallisuus on MFC:n luokissa toiminnaltaan varma ja kokemattomillekin toteuttajille helppo luoda. Myös MFC:n luokkien tarjoamat ominaisuudet synkronointiin (eli singleton-ominaisuuden toteutukseen) olivat erittäin käyttökelpoisia. Ongelmiakin säieturvallisuutta luotaessa kohdattiin, mutta ne johtuivat pitkälti puutteista MFC:n ja ATL:n yhteistoiminnassa (katso luku 6.1.3).

6.3 Sovelluksen lopputuloksen ja UIML-määrittelykielen arviointia

Tässä luvussa arvioidaan opinnäytteessä kehitetyn sovelluksen lopputulosta ja UIML:n käytöstä saatuja hyötyjä tämän sovelluksen osalta.

6.3.1 Sovelluksen lopputuloksen arviointia

Työmäärältään UIML-renderöintikoneiston kehittäminen muodostui ennakoitua suuremmaksi. Työmäärän suuruuteen vaikuttivat eniten hankaluudet tekniikoiden yhdistämisessä ja kokemattomuus MFC-luokkakirjaston käytössä.

Tämän lopputyön puitteissa sovelluksesta valmistui ensimmäinen versio, joten arviointi suoritetaan tämän version perusteella. Sovelluksen kehitystä on tarkoitus jatkaa ja myöskin mahdollisia vaihtoehtoisia tekniikoita tullaan tutkimaan.

Sovelluksen ensimmäisen version valmistuttua voi olla tyytyväinen lopputulokseen. Vaatimusmäärittelyssä asetetut tavoitteet täyttyivät lähes täysin. Sisäinen arkkitehtuuri valmistui täysin ja toimi moitteetta alun hankaluuksien jälkeen. Viestimekanismi sekä käyttöliittymältä asiakkaalle että päinvastoin toimi suunnitellusti. Myös säikeiden luonti ja käsittely toimivat vaatimusten mukaisesti ja käyttöliittymien sijoittaminen säikeisiin soveltui käyttötarkoitukseen erinomaisesti. Myös COM-rajapinnat ulospäin toimivat testien perusteella hyvin.

Vaatimusmäärittelyn *ei-toiminnalliset* vaatimukset (katso luku 4.1.2) toteutuivat täysin. Toiminnallisetkin vaatimukset (luku 4.1.2) toteutuivat tärkeimmiltä kohdiltaan, ainoastaan vaadituista käyttöliittymäelementeistä ominaisuuslomake (MFC:n luokka `CPropertySheet`) ja valikot (`CMenu`) jäivät tähän versioon toteuttamatta. Puutteet johtuvat MFC:n käyttöliittymäluokkien käytön hankaluudesta (katso luku 6.2.1).

Puutteena sovelluksen ensimmäisessä versiossa voidaan pitää sen osittaista sidonnaisuutta MFC-luokkakirjastoon. UIML-renderöintikoneiston toimiessa ideaalisti onnistuisi käyttöliittymäkirjaston vaihto vaivattomasti sanastoja muuttamalla. Kuitenkin sovelluksen ensimmäisessä versiossa vaihto vaatisi hieman muutoksia myös renderöintiluokan toteutukseen. Tämän ominaisuuden täydellinen toteuttaminen olisi vaatinut resursseja kohtuuttoman paljon, varsinkin kun toimivuudesta ei olisi sittenkään ollut takuita. Luokkakirjaston helpon vaihtamisen vaikea toteutettavuus johtuu pitkälti MFC:n huonoista ominaisuuksista dynaamisessa luonnissa, osaltaan toki myös sovelluksen toteuttajien kokemattomuudesta MFC-luokkakirjaston käytössä. Toisaalta esimerkiksi vaihtoehtoisen Qt-käyttöliittymäkirjaston avulla nämä ominaisuudet olisivat olleet huomattavasti helpommin ja vähemmällä vaivalla toteutettavissa.

6.3.2 UIML:n soveltuvuus käytäntöön

UIML oli tekijälle ennen UIML-renderöintikoneiston kehittämistä tuntematon määrittelykieli. Tämän lopputyön aikana niin UIML kuin yleisemminkin XML tulivat tutuiksi, ja niiden käytöstä saatavat hyödyt alkoivat näkyä käytännön työskentelyssäkin. UIML:n käytöstä saatua hyötyä on analysoitu tämän lopputyön teoriaosuudessa kuvatun listan (luku 2.1.2) perusteella.

Suurin UIML:n käytöstä saatu hyöty on sen helppokäyttöisyys verrattuna normaaliin käyttöliittymäohjelmointiin. Varsinkin itse määriteltävän nimistön avulla UIML-dokumentin luonti oli melko vaivatonta. Elementtien ja tunnisteiden vapaa nimeämismahdollisuus oli HTML:n käyttöön tottuneelle erittäin antoisaa, tosin nimistön määrittely ja siinä tapahtuneet muutokset vaativat huomiota myös nimistön dokumentoinnissa.

Muista luvussa 2.1.2 luetelluista hyödyistä sovelluksen pieni koko toteutui myös, tosin sovellus- ja testiympäristössä tästä ominaisuudesta saatu hyöty jäi minimaaliseksi. Käyttöliittymän jakautuminen kahteen osaan toteutui myös täysin, ja sekin on normaaliin käyttöliittymäohjelmointiin verrattuna selkeä etu. Tästäkin ominaisuudesta saatava käytännön hyöty ilmenee parhaiten vasta loppukäyttäjien renderöintikoneistolla operoidessa. Ominaisuus, jonka avulla UIML-dokumentissa voidaan viitata valmiisiin elementteihin (*template*) esimerkiksi URL:n avulla, ei kuulunut ensimmäisen version vaatimusmäärittelyyn, joten sen hyödyllisyydestä ei ole käytännön kokemusta.

UIML:n käytössä käyttöliittymien kuvaamiseen ilmeni myös muutamia hankaluuksia. Nämä hankaluudet esiintyivät enemmänkin työmäärän lisääntymisenä kuin varsinaisina UIML:n puutteina. Työmäärän suuruus ilmeni parhaiten elementti- ja tapahtumamäärältään suurissa käyttöliittymissä. Tällöin UIML-tiedostot kasvoivat pituudeltaan huomattavan suuriksi, ja niiden selkeys ja sitä mukaa muokattavuus huononivat. Varsinkin käsiteltävien tapahtumien paljous tekee tiedostoista hankalasti tulkittavia. Myös silloin kun käyttöliittymä sisältää useampia saman luokan elementtejä, voi oikean elementin löytäminen olla usein hidasta ja turhauttavaa.

Toisena UIML:n hankaluutena voidaan pitää suurissa käyttöliittymissä ”hienosäädön” vaikeutta. Elementtien kokojen ja sijaintien ilmoittaminen koordinaattien avulla saattaa joskus tuntua työläältä.

Eräänä parannuskeinona näille puutteille voisi pitää esimerkiksi kahdensuuntaisesti toimivaa käyttöliittymän generointityökalua. Työkalu generoisi piirretystä käyttöliittymästä automaattisesti UIML-tiedoston ja kyseistä UIML-dokumenttia muokattaessa näyttäisi muutokset piirrettävässä käyttöliittymässä. Tällöin tosin UIML:n hyvistä vahvuuksista paikkariippumattomuus ja siirrettävyys huononisivat, ja siirryttäisiin jälleen lähemmäksi perinteisempää käyttöliittymäohjelmointia. Toisena parannuskeinona havaittuihin puutteisiin voisi toimia jo edelläkin mainittu valmiiden mallien (*template*) käyttö. Tämän avulla elementtien uudelleenkäytettävyyys paranisi ja tästä johtuen UIML-dokumentit olisivat lyhyempiä ja selkeämpiä.

6.4 NPDI-prosessimallin toteutuminen käytännössä

Tässä luvussa arvioidaan opinnäytteessä käytetyn prosessimallin toteutumista. UIML-renderöintikoneiston kehittäminen toimi tässä suhteessa eräänlaisena ”pilottiprojektina” NPDI-prosessimallin käyttämisessä osana näinkin pienen sovelluksen kehittämistä.

6.4.1 Prosessin kulku

Prosessin alku sujui vauhdikkaasti ja eteni mallin mukaisesti varsin pitkälle. Alustusvaiheen alustustiimin kokoonpano selkeni nopeasti ja vaaditut dokumentitkin syntyivät vaivatta. Määrittelyvaihe sujui ongelmitta, vaikkakin tarkemman aikataulun muodostaminen osoittautui hieman työlääksi, koska prosessin kehityshenkilöt olivat osana muitakin ohjelmistonkehitysprojekteja ja näiden tarkkoja aikavaatimuksia ei ollut tiedossa.

Myös tulevat kesälomat hankaloittivat aikataulun laadintaa. Kehitysvaihe sujui aikataulun mukaisesti lähes loppuun saakka. Vaiheen lopussa paljon resursseja vaatineet ongelmat (esim. hankaluudet MFC:n käyttöliittymäluokkien kanssa, ks. luku 6.2.1) siirsivät kehitysvaiheen päätöstä hieman suunniteltua myöhemmäksi. Vahvistusvaiheen testaus suoritettiin sisäisesti sovelluksen kehittäjien kesken.

Tarvittavat muutokset niin käyttöliittymäelementteihin kuin sovelluksen luokka-arkkitehtuuriinkin tehtiin tämän testauksen perusteella. Markkinointi- ja jakeluvaihe jäi lopulta varsin suppeaksi, koska toteutunut versio UIML-renderöintikoneistosta oli vasta ensimmäinen. Vaiheen sisällöksi muodostui sovellusdokumenttien päivittäminen ajan tasalle ja ilmenneiden ongelmien kirjaaminen, jotta tulevaisuudessa vastaavia hankaluuksia ei koettaisi.

6.4.2 Huomioita prosessimallista

Prosessimallista saadut kokemukset olivat mielestämme varsin antoisia. Pelkästään NPDI-prosessimallin eduiksi kaikkia näitä kokemuksia ei voida laskea, sillä samoja ominaisuuksia löytyy toki muistakin prosessimalleista. Lähtökohtana voidaan pitää sitä, että yleensä yrityksessämme ei vastaavanlaisia pienempiä ohjelmistoprojekteja ole juurikaan minkään prosessimallin mukaan toteutettu.

NPDI:n hyviksi puoliksi koettiin melko tarkan aikataulun määrittäminen ja sen säännöllinen seuranta, dokumentoinnin monipuolisuus ja kattavuus sekä henkilöiden selkeä tehtävänjako. Tarkan aikataulun määrittäminen ja toteuttaminen onnistuivat projektissa erittäin hyvin. Suurimmat kiireet koettiin kehitysvaiheen lopussa, mutta pienellä viivästymisellä ja resurssien kasvattamisella siitäkin selvittiin.

Kuitenkin tarkka ja ehkä tiukkakin aikataulutus koettiin positiiviseksi seikaksi, motivaatio ja työteho pysyivät korkealla läpi projektin. Myös prosessimallin valmiit dokumenttipohjat todettiin varsin kattaviksi ja soveltuviksi myös osaksi pienempää sovellusprojektia. Varsinkin alustusvaiheen dokumentointi (*business plan*) oli ennestään varsin tuntematon käsite. Myös määrittely-/suunnitteluvaiheen dokumentaatio tuki erinomaisesti sovelluksen rakenteen suunnittelua. Se kuitenkin selkeytti kehitettävän sovelluksen tehtävää ja teknisiä lähtökohtia niin pääkehittäjille kuin muillekin alustustiimin jäsenille. Myös suunnitteludokumentin vaativuus ja monipuolisuus koettiin positiiviseksi seikaksi, varsinkin sovellusvaiheen ongelmia ratkottaessa.

NPDI:n ja yleisemmin muidenkin prosessimallien toteuttamisen hankaluuksista pienemmissä ohjelmistoprojekteissa voidaan mainita muutama. NPDI:n hankaluutena koettiin muutaman ihmisen työmäärän kasvaminen käytännössä melko suureksi ja myös lukuisista ydintiimin (*Core Team*) kokoontumista muutamaa pidettiin tarpeettomana. Dokumenttipohjat olivat kattavia ja mainittiin NPDI-prosessimallin hyvissä kokemuksissa, mutta joissain dokumenteissa liiallinen pikkutarkkuus ja sopimattomuus tämän kokoluokan projektiin vaativat valmiiden dokumenttipohjien muokkaamista.

NPDI-prosessimallin vaatiman työmäärän kasaantuminen muutamalle avainhenkilölle tiedettiin jo ennen projektin alkua. Käytännössä ongelmat johtuivat paljolti siitä, että ydintiimin johtaja (*Core team leader*) oli myös suuressa vastuussa sovelluksen kehittämisessä. Tällöin lukuisat dokumentit, palaverit ja muutkin johtajan tehtäväkenttään kuuluvat toiminnot olisivat vaatineet paljon resursseja, ja kehityspuoli olisi väistämättä kärsinyt tästä.

Tehtävien kasaantuminen näkyi parhaiten toteutusvaiheen lopussa, kun uusia ongelmia ilmeni ja aikataulussa pysyminen vaati resurssien keskittämistä kehitystehtäviin. Tämä ongelma johtui pitkälti projektin henkilömäärän pienuudesta. Mikäli johtajana olisi toiminut sovelluksen toteutuksesta erillinen henkilö, olisi dokumentit ja kokoontumiset ollut aikataulujen kiristyessäkin helpompi organisoida. Tässä vaiheessa myös lukuisat kokoontumiset vaiheiden lopuissa ja aluissa tuntuivat liiallisilta, muutaman kerran koko ydintiimin kokoontumista vaatinut asia hoidettiinkin hieman epävirallisemmin, esimerkiksi henkilökohtaisten keskusteluiden perusteella.

7 Yhteenveto

Tutkielman alussa käsiteltiin UIML-määrittelykielen teoriaa, MFC-luokkakirjastoa sekä muita käytettyjä tekniikoita. Myös käytettyä NPDI-prosessimallia kuvattiin tärkeimmiltä osiltaan. Kutakin aihetta kuvattiin niiden laajuuden vuoksi vain tässä opinnäytteessä tarvituilta osin. Tutkielman käytännön osuus muodostui renderöintisovelluksen kehittämisestä UIML-määrittelykielelle. Kehitettyä UIML-renderöintikoneistoa on kuvattu siten, että abstraktiotaso on melko korkealla. Tekniikoiden ja sovelluksen arvioinnissa pääpaino oli tekniikoiden soveltumisessa dynaamiseen käyttöliittymän kehittämiseen. Prosessimallia arvioitiin osana pienehköä ohjelmistoprojektia.

Teoriaosuuden kirjallisuuteen tutustuminen ja sovelluksen toteuttaminen käytännössä opettivat lopputyön tekijälle paljon varsinkin rakenteisten dokumenttien käytöstä ja dynaamisesti luotavien käyttöliittymien tarpeista ja vaatimuksista. UIML-määrittelykieli osoittautui käyttökelpoiseksi, ja muutkin valitut tekniikat soveltuivat tarkoitukseensa hyvin. Tekijän vähäisestä kokemuksesta johtuen joitain hankaluuksiakin sovellusta toteutettaessa ilmeni. Varsinkin MFC-käyttöliittymäkirjaston hyvä hallinta olisi vaatinut huomattavaa resurssien lisäystä. Osittain tästä johtuen myös työmäärä, minkä renderöintikoneiston toteuttaminen vastaamaan UIML:n asettamia tavoitteita vaatii, kasvoi suureksi. Kuitenkin alussa asetetut tutkimustavoitteet saavutettiin ja tulokset antoivat vastauksia esitettyihin kysymyksiin.

Koska tutkimus perustuu pitkälti UIML-renderöintikoneiston toteuttamiseen, rajoittuvat tulokset ja niiden analysointi käytettyihin tekniikoihin. Tästä johtuen vaihtoehtoisten tekniikoiden tutkiminen ja nyt saatujen tulosten kyseenalaistaminen jäivät pienelle huomiolle. Esimerkiksi UIML-renderöintikoneiston prototyypin kehittäminen useamman käyttöliittymäkirjaston avulla tarjoaisi enemmän vertailupohjaa MFC:n ominaisuuksia arvioitaessa.

Myös automaation asettamat rajoitteet (COM-tuen vaatimukset) saattavat vinouttaa tutkimustuloksia varsinkin MFC-luokkakirjaston käyttökelpoisuuden osalta. Näistä syistä johtuen voivat tutkimustulokset olla osittain rajoittuneita ja pitkälle meneviä johtopäätöksiä UIML:n käytöstä varsinkin vaihtoehtoisten tekniikoiden kanssa kannattaa välttää.

Tutkimusta on tarkoitus jatkaa kartoittamalla vaihtoehtoisia käyttöliittymäkirjastoja ja analysoimalla niiden ominaisuuksia. Tällöin saadaan vertailupohjaa MFC:lle ja mahdollisesti muillekin tutkimuksessa käytetyille tekniikoille. Myös sovellukselta vaaditaan tällöin uusia ominaisuuksia, etenkin luokkakirjaston vaihdon tulee olla entistä joustavampaa.

Pidemmällä aikavälillä jatkokehitykseen vaikuttaa osaltaan myös UIML:n käytön yleistyminen ja sen tuleva standardointi W3C:n (*the World Wide Web Consortium*) toimesta. Standardisoinnin avulla UIML tavoittaisi useampia ihmisiä ja tällöin myöskin erilaisia UIML:n ympärille kehitettyjä varusohjelmia (mm. editorit ja renderöintikoneistot) tulisi enemmän saataville. Tämä helpottaisi tulevaisuuden tutkimustyötä ja muodostaisi entistä enemmän vertailupohjaa omille tutkimustuloksille. Tällä hetkellä odotukset ovatkin UIML:n tarjoamien ominaisuuksien suhteen korkealla.

Lähteet

Abrams Marc, ”UIML: An Appliance-Independent XML User Interface Language”, Computer Networks, Vol. 31 No. 11-16 (May 1999), s. 1695-1708.

Agerbo Ellen ja Cornils Aino, ”How to preserve the benefits of Design Patterns”, ACM SIGPLAN Notices, Vol. 33 Issue 10 (Oct 1998), s. 134-143.

Booch Grady, Rumbaugh James ja Jacobson Ivar, ”The Unified Modeling Language User Guide”, Addison-Wesley, Reading Massachussetts, 1999.

Buschmann, Meunier, Rohnert, Sommerlad, Stal, ”Pattern-Oriented Software Architecture”, John Wiley & Sons, West Sussex England, 2001.

DiLascia Paul, ”More Reusable MFC Goodies”, Microsoft Systems Journal (MSJ), Vol. 14 No. 12 (Dec. 1999), s. 31-46.

Dung Nguyen, ”Design patterns for data structures”, ACM SIGCSE Bulletin, Vol. 30 No. 1 (Mar 1998), s. 336-340.

Grimes Richard ja Stockton Alex, ”Beginning ATL COM Programming”, Wrox Press Ltd, Birmingham UK, 1998.

Harmonia Inc., ”User Interface Markup Language (UIML) 3.0 Specification”, saatavilla WWW-muodossa osoitteessa <URL: http://www.uiml.org/specs/docs/uiml30-revised-02-12-02.pdf>, 8.2.2002.

Harold Elliotte Rusty, ”XML Bible”, Hungry Minds Inc., New York NY, 2001.

Honeywell Oy, ”NPDI (New Product Development and Introduction Process)”, saatavilla WWW-muodossa osoitteessa <URL: http://web.iac.honeywell.com/process/>, 17.4.1998.

IEEE Standards Committee, ”Draft: Recommended Practice for Transfer of Power Quality Data (P1159.3/D9)”, saatavilla WWW-muodossa osoitteessa <URL: http://grouper.ieee.org/groups/1159/3/docs/p1159-3-d9.pdf >, 1.8.2002.

Lam John, ”Building a COM object with ATL”, PC Magazine, Vol. 15 Issue 22 (1996), s.233-236.

Lam John, ” ATL: Rx for your COM headaches”, PC Magazine, Vol. 15 Issue 21 (1996), s.333-336.

Li Sing ja Economopoulos Panos, ”Professional COM Applications with ATL”, Wrox Press Ltd., Olton Birmingham, 1998.

Mace Scott, ”Weaving a Better Web”, Byte, Vol. 23 Issue 3 (March 98), s. 58-68.

Marchal Benoit, ”Applied XML Solutions”, Sams, Indianapolis Indiana, 2000.

Microsoft, ”MSDN Library, CWinApp:The Application Class", saatavilla World Wide Web -muodossa <URL: http://msdn.microsoft.com/library/>, viitattu 10.7.2002.

Musser David R., Derge Gillmer J. ja Saini Atul, ”STL Tutorial and Reference Guide, 2nd Edition”, Addison-Wesley, Reading Massachussetts, 2001.

Nentwitch Christian, ”A Consistency Checking and Smart Link Generation Service”, ACM Transactions on Internet Technology (TOIT), Vol. 2 Issue 2 (May 2002), s. 151-185.

Phanouriou Constantinos, ”UIML: A Device Independent User Interface Markup Language”, Virginia Polytechnic Institute and State University, Blacksburg Virginia, 2002.

Petreley Nicholas, ”Development on the Qt”, Computerworld, Vol. 36 Iss. 21 (May 2002), s. 54.

Petzold Charles, ”Programming Windows, 2nd Edition”, Microsoft Press, Redmond Washington, 1990.

Platt David, ”Give ActiveX-based Web-pages a boost with the apartment threading model” Microsoft Systems Journal (MSJ), Vol. 12 No 2 (Feb 1997), s. 17.

Pressman Roger S., ”Software Engineering, Vol. 4”, McGraw-Hill, USA, 1998.

Prosise Jeff, ”Programming Windows with MFC, 2nd Edition”, Microsoft Press, Redmond Washington, 1999.

Rector Brent, Sells Chris, ”ATL Internals”, Addison-Wesley, Reading Massachussets, 1999.

Rumbaugh James, ”Object-Oriented Modeling and Design”, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

Salminen Airi, ”Rakenteisen tekstin hallinta”, Tietojenkäsittelytieteiden julkaisuja: Opetusmonisteita OM-3, Jyväskylän yliopisto, Tietojenkäsittelyopin laitos, 1992.

Stroustrup Bjarne, ”C++ Programming Language, 3rd Edition”, Addison-Wesley, Reading Massachussetts, 1997.

Trolltech, ”Qt 3.1 Documentation”, saatavilla WWW-muodossa osoitteessa <URL: http://doc.trolltech.com/3.1/>, 2002.

Wick Michael, ”Using Design Patterns in CS1”, ACM SIGSE Bulletin, Vol. 33 Issue 1 (March 2001), s. 258-262.

Wilson Scott F., ”Analyzing Requirements and Defining Solution Architectures”, Microsoft Press, Redmond Washington, 1999.

Liitteet

Liite 1. Esimerkki UIML-sanastosta

```
<?xml version="1.0" ?>
<uiml>
 <template id="vocab">
  <presentation base="MFC_1.0_JanneKorhonen_1.0">
   <d-class id="cButton" used-in-tag="part" maps-type="class" maps-
    to="CButton">
    <d-property id="caption" maps-type="setMethod" maps-
     to="SetCaption">
      <d-param  type="string" />
    </d-property>
    <d-property  id="caption" maps-type="getMethod" maps-
     to="GetCaption" return-type="string" />
    <d-property  id="location" maps-type="getMethod" maps-
     to="GetLocation" return-type="CPoint" />
    <d-property id="location" maps-type="setMethod" maps-
     to="SetLocation" return-type="void">
      <d-param  name="x" type="int" />
      <d-param  name="y" type="int" />
    </d-property>
    <d-property  id="size" maps-type="getMethod" maps-to="GetSize"
       return-type="CPoint" />
    <d-property id="size" maps-type="setMethod" maps-to="SetSize"
       return-type="void">
       <d-param  name="width" type="int" />
       <d-param  name="height" type="int" />
    </d-property>
    <event  class="OnClick" />
   </d-class>
```

```
    <d-class id="cEdit" used-in-tag="part" maps-type="class" maps-
      to="CEdit">
      <d-property  id="text" maps-type="getMethod" maps-to="GetText"
        return- type="string" />
      <d-property id="text" maps-type="setMethod" maps-to="SetText"
        return-type="void">
        <d-param  type="string" />
      </d-property>
      <d-property  id="location" maps-type="getMethod" maps-
        to="GetLocation" return-type="CPoint" />
       <d-property id="location" maps-type="setMethod" maps-
        to="SetLocation" return-type="void">
          <d-param  name="x" type="int" />
          <d-param  name="y" type="int" />
       </d-property>
       <d-property  id="size" maps-type="getMethod" maps-to="GetSize"
         return-type="CPoint" />
      <d-property id="size" maps-type="setMethod" maps-to="SetSize"
        return-type="void">
          <d-param  name="width" type="int" />
          <d-param  name="height" type="int" />
       </d-property>
     </d-class>
   </presentation>
  </template>
</uiml>
```

Liite 2. Esimerkki XML-muotoisesta komentorakenteesta

```
<cmdGram>
  <command>
    <description>This is a description for this command</description>
      <detail name="cmdName">EndThread</detail>
      <detail name="TerminateOption">WAIT_OPTION1</detail>
  </command>
  <command>
    <description>Restructure command, UIML specific</description>
    <detail name="name">restructure</detail>
    <detail name="XML">
      <uiml>
        <template>
          <restructure at-part="btnOK" how="replace">
            <template>
              <part id="btnOK">
                <style>
                  <property name="caption">Okei</property>
                </ style>
              </ part>
            </ template>
          </ restructure>
        </ template>
      </ uiml>
    </ detail>
  </ command>
</ cmdGram>
```

Liite 3. Esimerkki virheiden määrittelystä XML-muodossa

```
<EEInfo srcComp="example_component.dll">
   <item>
      <description>Error description comes here.</description>
         <detail name="database">Example_db</detail>
         <detail name="server">example_server</detail>
   </item>
</EEInfo>
```