



TAMPEREEN TEKNILLINEN YLIOPISTO

MARKO FELIN
MICROSOFT VISION LAAJENTAMINEN JOUSTAVAKSI
TIETOKANTAPOHJAISEKSI MALLINNUSTYÖKALUKSI
Diplomityö

Tarkastajat: professori Kai Koskimies
yliassistentti Jari Peltonen
Tarkastajat ja aihe hyväksyty
Tieto- ja sähkötekniikan tiedekuntaneuvoston
kokouksessa 08.06.2011

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

FELIN, MARKO: Microsoft Visio laajentaminen joustavaksi tietokantapohjaiseksi mallinnustyökaluksi

Diplomityö, 60 sivua, 1 liitesivu

Syyskuu 2011

Pääaine: Ohjelmistotuotanto

Tarkastajat: professori Kai Koskimies ja yliassistentti Jari Peltonen

Avainsanat: Mallinnustyökalu, Microsoft Visio, laajennuskomponentti, tietokanta, työkaluintegrointi

Perinteiset ohjelmistojen mallinnustyökalut ovat jäykkiä, formaaleja ja syntaksiorientoituneita. Ne eivät tue vapaamuotoista luonnostelua tai käytetyn mallinnuskielen vastaisia kuvauksia. Mallinnustyö on kuitenkin luovaa, etenkin alkuvaiheessa, kun ratkaisuja vielä hahmotellaan. Tästä syystä käyttäjät turvautuvat muun muassa tavallisiin toimistosovelluksiin, jotka ovat joustavampia ja tukevat itse työtä paremmin. Toimistosovelluksilla tuotettu tieto ei kuitenkaan tule automaattisesti osaksi mallia, jolloin arvokasta työtä menetetään. Ohjelmistojen mallinnuksen varhaisessa työkalutuessa on näin ollen selkeitä puutteita, jotka kaipaavat täydentämistä.

Näitä ongelmia pyritään ratkaisemaan Tampereen teknillisen yliopiston Ohjelmistotekniikan laitoksella kehitetyllä Trinity-työkaluympäristöllä. Ympäristö integroi ole-massa olevia sovelluksia ja laajentaa niitä mallinnusominaisuuksilla. Ratkaisun tavoitteena on näiden sovellusten hyödyntäminen mallinnustyössä niin, että kaikesta tehdystä työstä tulee osa mallia. Tällä lähestymistavalla pyritään aikaistamaan mallinnuksen työkalutukea sekä kaventamaan mallinnustyökalujen ja toimistosovellusten välistä kuilua.

Trinity-ympäristön ensimmäiseksi integroitavaksi sovellukseksi valittiin Microsoft Visio, johon toteutettua laajennuskomponenttia tämä diplomityö käsittelee. Laajennuksen tavoitteena oli saada Visiosta ympäristössä toimiva joustava graafinen mallinnustyökalu, joka tukee myös luonnostelevaa työtä. Laajennuskomponentin olennaisin tehtävä on tallentaa Visiossa luodut mallit reaaliaikaisesti ympäristön tietokantaan ja pitää Vision käyttöliittymä synkronoituna tietokannan tilan kanssa. Muita keskeisiä vaatimuksia laajennukselle olivat mallinnuskielien helppo lisättävyys, toteutuksen geneerisyys ja usean samanaikaisen käyttäjän tuki.

Työn tuloksena saatiin aikaan Visio-laajennuskomponentti, joka vastaa edellä mainittuihin vaatimuksiin. Tämä kirjallinen osuus työstä esittelee laajennuskomponentin vaatimukset, määrittelyn, teknisen suunnittelun ja analyysin. Trinity-ympäristön kannalta Visio-laajennus havainnollisesti monipuolisesti sen lähestymistavan tuomia hyötyjä ja haasteita. Eräs keskeinen haaste on suorituskyky, joka nykyisellään on vielä vaatimaton ja kaippaa jatkokehitystä.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Master's Degree Programme in Information Technology

FELIN, MARKO: Extending Microsoft Visio into a flexible repository based modeling tool

Master of Science Thesis, 60 pages, 1 Appendix page

September 2011

Major: Software engineering

Examiners: professor Kai Koskimies and assistant professor Jari Peltonen

Keywords: Modeling tool, Microsoft Visio, extension, database, tool integration

Traditional software modeling tools are rigid, formal, and syntax-oriented. They do not support free-form sketching or informal descriptions that do not conform to the used modeling language. Modeling work, however, is creative especially in the beginning of work when solutions are still being outlined. For this reason, the users rely on regular office tools that are more flexible and support the actual work better. The information produced with office tools, however, does not automatically become a part of a model and valuable work is lost. There is clearly a need for flexible modeling tools that supports modeling in earlier phases of work.

Trinity is a modeling tool environment developed at Tampere University of Technology that pursues to find solutions to these problems. The environment integrates existing office tools and extends them with modeling capabilities. The goal of this solution is to utilize office tools for modeling purposes, so that all work becomes a part of a model. This approach aims to promote earlier tool support and narrow the gap between traditional modeling tools and office tools.

Microsoft Visio was chosen to be the first tool to be integrated into the environment. This was achieved with a Visio extension that is presented in this thesis. The aim of the extension was to utilize Visio in the environment as a graphical modeling tool that supports sketching. The fundamental task of the extension component is to save the models drawn in Visio to the database of the environment and keep the interface of Visio synchronized with the state of the database. Other essential requirements for the extension were the ease of adding new languages, generic implementation, and support for collaborative work.

As a result of the work, a Visio extension fulfilling the aforementioned requirements was implemented. This written part of the thesis presents the requirements, specification, design and evaluation of the extension. The extension illustrated the various advantages and challenges of the approach used in Trinity. One essential challenge is performance, which is currently modest and requires future development.

ALKUSANAT

Haluaisin kiittää kaikkia Trinity-projektissa olleita henkilöitä ja erityisesti Mikko Vartialaa osallistumisesta Visio-laajennuskomponentin suunnitteluun, toteutukseen ja testaamiseen. Kiitokset myös työn tarkastajille Kai Koskimiehelle ja Jari Peltoselle työn ohjaamisesta ja kirjallisen osuuden arvokkaista kommentteista. Lisäksi lämmin kiitos perheelle ja kaikille läheisille ihmisille, jotka ovat kannustaneet ja tukeneet tämän työn loppuun saattamisessa.

Tampereella, 17. elokuuta 2011

Marko Felin

SISÄLLYS

1. JOHDANTO	1
2. TAUSTA JA TEORIA	3
2.1. OHJELMISTOJEN MALLINTAMINEN	3
2.2. METAMALLIT JA MALLINNUSKIELIEN MÄÄRITTELY	3
2.3. MICROSOFT VISIO.....	5
2.3.1. <i>Yleiset ominaisuudet</i>	5
2.3.2. <i>Peruskäsitteet</i>	5
2.3.3. <i>Mallinnustuki</i>	6
2.3.4. <i>Automaattiorajapinta</i>	7
2.3.5. <i>Laajennuskomponenttitekniikat</i>	7
2.3.6. <i>ShapeSheet</i>	8
2.4. SUUNNITTELU- JA ARKKITEHTUURIMALLIT.....	9
2.4.1. <i>Tarkkailija-suunnittelumalli</i>	10
2.4.2. <i>Tarkkailija-suunnittelumalli .NET-tapahtumilla ja delegaateilla</i>	11
2.4.3. <i>Malli-näkymä-ohjain -arkkitehtuurimalli</i>	11
2.4.4. <i>Kerrosarkkitehtuuri</i>	12
3. JOUSTAVAT MALLINNUSTYÖKALUT	14
3.1. MALLINTAMISEN JOUSTAVA LUONNE	14
3.2. JOUSTAVAN MALLINNUSTYÖKALUYMPÄRISTÖN VAATIMUKSIA	15
3.2.1. <i>Työkaluintegraatio</i>	15
3.2.2. <i>Tiedon integraatio</i>	16
3.3. MALLINNUSTYÖKALUN VAATIMUKSIA	16
3.3.1. <i>Käytettävyys ja ominaisuudet</i>	16
3.3.2. <i>Osana ympäristöä</i>	17
3.3.3. <i>Visio-laajennus</i>	17
4. TRINITY – MALLINNUKSEN- JA TYÖKALUYMPÄRISTÖ	19
4.1. YLEISTÄ.....	19
4.2. YMPÄRISTÖN PERUSPALVELUT	19
4.2.1. <i>Hallintakäyttöliittymä</i>	20
4.2.2. <i>Informaatiopaneeli</i>	21
5. LAAJENNETTU MICROSOFT VISIO	23
5.1. KÄYTTÖLIITTYMÄ	23
5.2. MALLI- JA NÄKYMÄELEMENTIT	24
5.3. TYÖKALULLA MALLINTAMINEN.....	25
5.3.1. <i>Kaavion avaaminen</i>	26

5.3.2.	<i>Elementin luominen</i>	26
5.3.3.	<i>Elementin ominaisuuksien muokkaaminen</i>	28
5.3.4.	<i>Suhteen luominen</i>	28
5.3.5.	<i>Elementin kopioiminen</i>	29
5.3.6.	<i>Mallin katselmointi</i>	29
5.3.7.	<i>Näkymäasetukset</i>	30
5.3.8.	<i>Kaavion sulkeminen ja työn lopettaminen</i>	30
5.4.	ERIKOISTOIMINTOJA	31
5.4.1.	<i>Viivan luominen suoraan elementistä</i>	31
5.4.2.	<i>Vision UML-mallien tuominen Trinity-ympäristöön</i>	32
6.	JÄRJESTELMÄN TOTEUTUS	33
6.1.	SUUNNITTELUPERIAATTEET	33
6.2.	TRINITY-YMPÄRISTÖN ARKKITEHTUURI.....	33
6.2.1.	<i>Tietokanta</i>	34
6.2.2.	<i>Tietokantakomponentti</i>	36
6.2.3.	<i>Integraatioarkkitehtuuri</i>	36
6.3.	VISIO-LAAJENNUSKOMPONENTIN ARKKITEHTUURI.....	37
6.3.1.	<i>ViewController-kerros</i>	38
6.3.2.	<i>Model-kerros</i>	41
6.4.	LAAJENNUSKOMPONENTIN ERITYISPIIRTEITÄ	42
6.4.1.	<i>ShapeSheet-datan käsittely</i>	42
6.4.2.	<i>Toimintojen kumoaminen ja uudelleen tekeminen</i>	44
7.	VISIO-MALLINNUSTYÖKALUN LAAJENNETTAVUUS	47
7.1.	KIELIEN NÄKYMÄELEMENTTIEN MÄÄRITTELY VISIOSSA	47
7.1.1.	<i>Template-elementin metaominaisuudet</i>	48
7.1.2.	<i>Näkymäelementtiä kiinnostavat ominaisuudet</i>	50
7.2.	OHJELMAKOODIN LAAJENNETTAVUUS.....	51
7.2.1.	<i>Asemoijat ja väritystilat</i>	51
7.2.2.	<i>Näkymätyyppikohtaiset käsittelijät</i>	52
7.2.3.	<i>MarkerEvent-pluginit</i>	52
8.	ANALYYSI	54
8.1.	TOTEUTUKSEN ARVIOINTI.....	54
8.2.	TULEVAISUUS JA PARANNUSEHDOTUKSIA	56
8.3.	AIHEESEEN LIITTYVIÄ JULKAISUJA	57
9.	YHTEENVETO	60
	LIITE 1: TRINITYN METAMETAMALLI	61

LÄHTEET 62

TERMIT JA NIIDEN MÄÄRITELMÄT

API	Application Programming Interface, ohjelmointirajapinta. Rajapinta, jonka kautta sovellus mahdollistaa toisen sovelluksen käyttäen palveluita.
COM	Component Object Model. Microsoftin kehittämä teknologia ohjelmistokomponenttien tekemiseen.
COM Add-in	COM-teknologialla toteutettu laajennuskomponentti Microsoft Office-sovelluksiin.
DLL	Dynamic Link Library. Microsoftin toteutusteknologia jaetuille kirjastoille.
EMF	Eclipse Modeling Framework. Eclipse-pohjainen mallinnuskehys ja koodigenerointiympäristö.
GUID	Globally Unique Identifier. Ohjelmistoissa käytetty globaalisti uniikki 128-bittinen tunniste.
Master	Perusmuoto. Microsoft Vision muotojen alkuperäinen muoto, joka voidaan pudottaa Vision kaavioihin uudeksi muodoksi.
MOF	Meta-Object Facility. MOF on malliohjautuvan ohjelmistokehityksen standardi, jota käytetään UML:n määrittelyyn.
MUI	Management User Interface. Trinity-ympäristön hallintatyökalu.
MVC	Model-View-Controller, malli-näkymä-ohjain. Arkkitehtuurimalli, jonka idea on erottaa käyttöliittymä sovelluslogiikasta ja –datasta.
Shape	Muoto. Microsoft Vision kaavioissa oleva näkymäelementti.
ShapeSheet	Taulukko, jossa on tietoja Vision muodosta. Näitä tietoja ovat esimerkiksi muodon mitat ja ulkoasun määrittävät tyylit.
Stencil	Kaavain. Microsoft Vision osa, joka sisältää Master-elementtejä.
Template	Esimääritelty malli, jota voidaan käyttää valmiina pohjana jotakin elementtiä luotaessa.
Trinity	Tampereen teknillisen yliopistolla toteutettu mallinnus- ja työkaluympäristö.
UML	Unified Modeling Language. UML on ohjelmistojen mallinnukseen käytetty yleiskäyttöinen mallinnuskieli.
XMI	XML Metadata Interchange. XML-pohjainen standardi MOF-pohjaisille metamalleille.
XML	Extensible Markup Language. Tekstuaalinen metakieli, jonka avulla voidaan määritellä rakenteellisia merkkäuskieliä.

1. JOHDANTO

Ohjelmistojen mallintaminen on luonteeltaan luovaa, iteratiivista ja työn vaiheesta riippuvaista. Projektin alkuvaiheessa työ on luonnostelevaa ja hahmottelevaa, kun taas myöhemmässä vaiheessa malleista usein halutaan muodostaa täydellisiä ja tiukasti käytetyn mallinnuskielen mukaisia. Nykyiset mallinnustyökalut kuitenkin tukevat riittävästi vain jälkimmäistä ja rajoittavat luovaa työtä muun muassa pakottamalla mallit käytetyn kielen sääntöjen mukaisiksi. Lisäksi ne eivät yleisesti salli vapaamuotoisia elementtejä osana mallia tai tue muita luonnostelumenetelmiä. Perinteiset toimistosovellukset sen sijaan ovat joustavampia, mistä syystä niitä suositaan usein projektin alkuvaiheessa mallinnustyökalujen sijaan. Toimistosovelluksilla luotu tieto ei kuitenkaan tule osaksi lopullista mallia, vaan se joudutaan jälkikäteen lisäämään haluttuun malliin varsinaista mallinnustyökalua käyttäen. Ohjelmistojen mallinnuksen varhaisessa työkalutuessa on täten selkeä aukko, joka kaipaa täydentämistä.

Tampereen teknillisen yliopiston Ohjelmistotekniikan laitoksella on jo usean vuoden ajan tutkittu mallintamisen työkalutukea ja etsitty ratkaisuja muun muassa edellä mainittuun ongelmaan. Tämän tutkimuksen puitteissa kehitetään parhaillaan Trinity-mallinnus- ja työkaluympäristöä, joka integroi joukon olemassa olevia sovelluksia ja laajentaa niitä mallinnusominaisuuksilla. Ympäristön tavoitteena on mahdollistaa olemassa olevien ja tuttujen työkalujen, kuten toimistosovellusten, käyttämisen kussakin vaiheessa projektia niin, että kaikesta tehdystä työstä tulee aina osa jotakin mallia. Näin mallinnuksen työkalutukea voidaan laajentaa kattamaan kaikkia projektin vaiheita ja kuilua perinteisten mallinnustyökalujen ja toimistotyökalujen välillä saadaan kavennettua.

Trinity-ympäristön ensimmäiseksi integroitavaksi työkaluksi valittiin Microsoft Visio, johon toteutettua laajennuskomponenttia tämä työ käsittelee. Laajennetun Vision tarkoituksena on toimia joustavana graafisena mallinnustyökaluna, joka sallii myös epäformaalien mallien ja vapaamuotoisten elementtien luomisen ilman luovaa työtä hidastavia syntaksirajoitteita. Laajennuskomponentin olennaisimpana toiminnallisuutena on lisätä Visioon tarvittavat mallinnusominaisuudet ja tallentaa käyttäjien luomat mallit reaaliaikaisesti ympäristössä sijaitsevaan yhteiseen tietokantaan sekä pitää työkalun käyttöliittymä sen kanssa synkronoituna. Työssä esiteltyä toteutusta ei aloitettu puhtaalta pöydältä, vaan Vision rajapintaan tutustuttiin etukäteen ja laajennuskomponentista toteutettiin ensimmäinen prototyyppi TTY:n ohjelmistotuotannon projekti-työkurssilla talvella 2008–2009.

Luvussa kaksi esitetään työn ymmärtämisen kannalta tärkeimmät taustatiedot ja teoria. Luku kolme käsittelee työn ongelmaa ja vaatimuksia työkalulaajennokselle sekä sen ympäristölle. Luvussa neljä esitellään Trinity-ympäristön tarjoamia peruspalveluita,

joihin laajennuskomponentti nojaa. Laajennetun Vision käyttöä esitellään luvussa viisi, jonka jälkeen luku kuusi kuvaa ympäristön ja laajennuskomponentin arkkitehtuuriset ratkaisut. Luku seitsemän käsittelee laajennuskomponentin toteutuksen keskeiset yksityiskohdat. Luvussa kahdeksan työtä analysoidaan arvioimalla toteutusta, antamalla sille tulevaisuuden parannusehdotuksia ja esittelemällä muutama muu työn aiheeseen läheisesti liittyvä julkaisu. Lopuksi luku yhdeksän sisältää yhteenvedon.

2. TAUSTA JA TEORIA

Tässä luvussa esitellään työn kannalta tärkeimmät taustatiedot ja teoria lyhyesti. Nämä sisältävät ohjelmistojen mallintamiseen liittyvät käsitteet, laajennettavan Microsoft Vision esittelyn sekä työssä käytetyt suunnittelu- ja arkkitehtuurimallit.

2.1. Ohjelmistojen mallintaminen

Ohjelmistokehityksessä käytetään yleisesti erilaisia malleja kuvaamaan järjestelmiä ja niihin liittyviä ongelma-alueita. Mallien avulla tarkasteltava kohde voidaan yksinkertaistaa rajoittamalla johonkin näkökulmaan ja abstrahoimalla siitä epäolennaiset yksityiskohdat pois. Abstraktiotaso ja käytötapa voi vaihdella aina korkeimman tason arkkitehtuurista alimman tason yksityiskohtaiseen suunnitteluun. Näitä malleja voidaan käyttää hyödyksi ongelman ymmärtämisen lisäksi dokumenttien ja spesifikaatioiden laatimiseen, ihmisten välisen kommunikaation apuvälineenä tai jopa ohjelmakoodin generoimiseen. (Koskimies et al. 2004.)

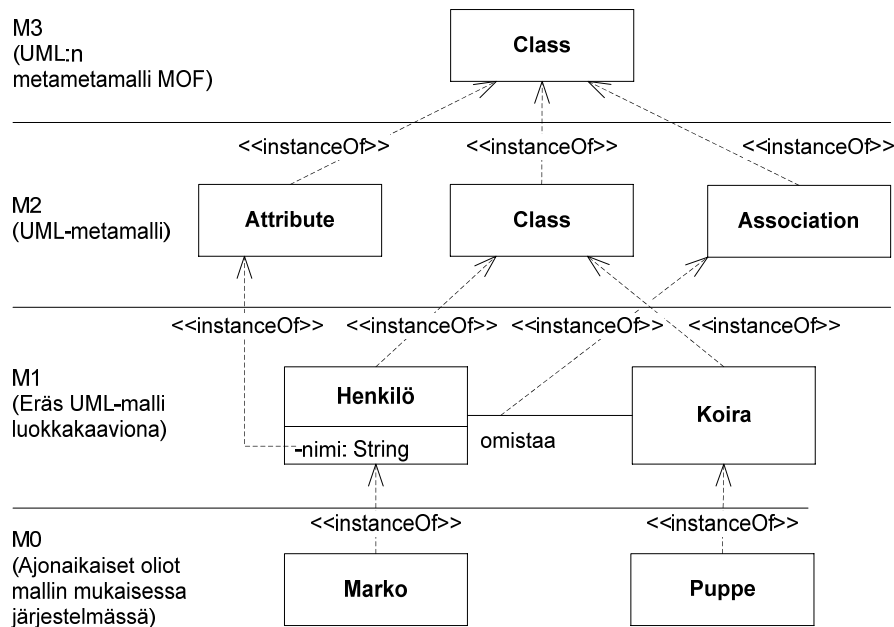
Mallit esitetään usein täsmällisesti määritellyillä kuvaustavoilla, joita kutsutaan *mallinnuskieliksi* (engl. modelling language). Mallinnuskielen esitysmuoto on yleensä tekstuaalinen tai graafinen ja se määritellään joko formaalisti, luonnollista kieltä käyttäen tai näiden yhdistelmänä. (Koskimies et al. 2004)

Nykypäivän ohjelmistoteollisuudessa ylivoimaisesti käytetyin ja tunnetuin graafinen mallinnuskieli on OMG:n vuonna 1997 standardoima UML (Unified Modeling Language) (OMG 2010a). Sen tämän hetken tuorein versio 2.3 koostuu yhteensä 14 erilaisesta kaaviotyypistä, jotka jaetaan karkeasti ottaen ohjelmiston rakennetta ja käyttäytymistä kuvaaviin kaavioihin. (OMG 2010b) Koska kyseessä on yksi mallinnuskieli, eri kaaviotyyppien elementtejä voidaan käyttää sekaisin samoissa kaavioissa, mikäli ne eivät ole UML:n metamallin vastaisia. Metamalleja ja kielten määrittelyä käsitellään seuraavassa kohdassa 2.2. UML:n yleisyydestä johtuen sitä käytetään tässä työssä esimerkki-kielenä kaikissa mallinnusta käsittelevissä kohdissa.

2.2. Metamallit ja mallinnuskielten määrittely

Kielten määrittelyssä on aina otettava kantaa sekä kielen rakenteeseen että sen merkitykseen. Ohjelmointikielten tapauksessa nämä määritellään tarkkaan ja yksiselitteisesti, mutta mallinnuskielten kohdalla merkitykselle jätetään tarkoituksella jonkin verran tulkinnanvaraa. Esimerkiksi UML:n väljä määrittely mahdollistaa sen soveltamisen eri vaiheissa ohjelmistokehitystä, kuten alkuvaiheen aikana tehtävässä epäformaalimmassa luonnostelussa. (Koskimies et al. 2004)

Mallinnuskielten rakenne voidaan jakaa abstraktiin ja konkreettiseen syntaksiin. UML:n kohdalla abstrakti syntaksi määrittelee mallielementtien loogiset suhteet ja konkreettinen syntaksi kaavioiden visuaalisen ulkoasu. Visuaalinen ulkoasu on olennainen vain ihmisen ymmärtämisen kannalta, joten UML:n standardi antaa konkreettisesti syntaksista vain suosituksia ja pienet variaatiot eivät ole ongelmallisia. Abstrakti syntaksi sen sijaan määrittää täsmällisesti käyttäen metamallia, joka kuvaa sallittujen mallien abstraktin rakenteen korkeamman tason mallilla. OMG:n tavoitteena ei kuitenkaan ole ollut standardoida ainoastaan UML, vaan kokonainen infrastruktuuri, jonka avulla voidaan kehittää muitakin samaan käsitteistöön nojautuvia mallinnuskieliä. Tämän seurauksena on kehitetty nelitasoinen metamalliarkkitehtuuri, jonka yksinkertaistettu kuvaus nähdään kuvassa 2.1. (Koskimies et al. 2004)



Kuva 2.1. UML:n mallitasot. Mukailtu lähteestä (OMG 2010c).

Metamallihierarkiassa alemman tason elementit ovat aina ylemmän tason luokkien ilmentymiä. Alin taso M0 sisältää ajonaikaiset järjestelmän oliot, jotka kuvataan tason M1 mallissa esimerkiksi kuvan 2.1 tapaan luokkakaaviolla. Seuraavalla tasolla M2 on puolestaan UML:n metamalli ja ylimpänä tasolla M3 UML:n metametamalli MOF (Meta Object Facility) (OMG 2011d). MOF on lopulta itsensä eräs ilmentymä ja nämä neljä tasoa riittävät muodostamaan riittävän kehyksen erilaisten samaan peruskäsitteistöön pohjautuvien mallinnuskielten määrittelyyn. (Koskimies et al. 2004)

UML on tarkoitettu yleiskäyttöiseksi ja monille kohdealueille soveltuvaksi kieleksi oliopohjaiseen suunnitteluun, mutta sitä voidaan halutessa erikoistaa tietyille kohdealueelle käyttäen sen tarjoamaa stereotyyppiin perustuvaa profiilimekanismia. (Koskimies et al. 2004.) Domain-spesifinen mallintaminen (Domain-Specific Modeling, DSM) on puolestaan menetelmä, jossa kielet suunnitellaan alusta asti geneerisyyden sijasta mahdollisimman ilmaisuvoimaiseksi ja helppokäyttöiseksi jollekin kohdealueelle. Tuotantokäytössä domain-spesifiset kielet tuovat yleiskäyttöisiin kieliin verrattuna tehok-

kuusetuja ja kustannussäästöjä, mutta niiden kehittäminen vaatii sekä vahvaa kohdealueen tuntemusta että ammattitaitoa kielien määrittelystä. (Mernik et al. 2005.) Domain-spesifisten kielien määrittelyssä metametamallin tulee tukea laajasti kaikkia mallinnuskielissä käytettyjä elementtejä, johon yleiskäyttöiset mallinnuskielet ovat usein liian matalalla abstraktiotasolla. Niitä kuitenkin käytetään yleisesti lähtökohtana DSM-järjestelmän metametamallia rakennettaessa. (Kelly & Tolvanen 2008 s. 367)

2.3. Microsoft Visio

Microsoft Visio on Office -tuoteperheeseen kuuluva grafiikka- ja piirustussovellus, joka sisältää laajan tuen erityyppisten kaavioiden luomiseen (Microsoft 2011a). Tässä luvussa käydään lyhyesti läpi Vision sisältämät perus- ja mallinnusominaisuudet sekä esitellään tämän työn kannalta keskeiset menetelmät sen laajentamiseen ohjelmointirajapintojen kautta. Ohjelmointirajapintoja on olennaisesti kaksi, automaatio- ja *ShapeSheet*-rajapinta. Nämä rajapinnat sekä mahdolliset laajennuskomponenttitekniikat esitellään omilla aliluvuissaan.

2.3.1. Yleiset ominaisuudet

Vision päätoiminnallisuus on erilaisten kaavioiden piirtäminen. Se sisältää tuen muun muassa erilaisten liiketoiminta-, vuo-, pohjapiirustus-, tietoverkko-, ohjelmisto- ja tietoverkkokaavioiden piirtämiselle valmiiden kaaviomallien (*template*) avulla. Nämä mallit sisältävät kaikki työkalut, muotoilut, asetukset ja elementit, joita tarvitaan tietyn tyyppisten kaavioiden muodostamiseen. (Microsoft 2011b)

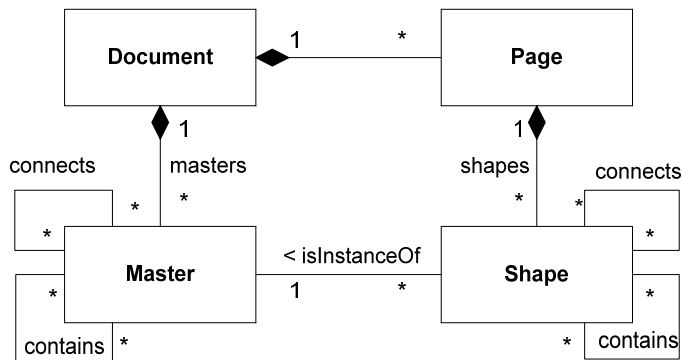
Laajan kaaviomallituen lisäksi Visiossa on edistyneet visuaaliset toiminnot ja työkalut, joilla kaavioiden piirtäminen on helppoa ja luontevaa. Se tukee valmiiden elementtien lisäksi vapaan vektorigrafiikan piirtämistä sekä elementtien ryhmittelyä, kerroksia, teemoja, asemointityökaluja ja monia muita nykyaikaisen grafiikkatyökalun perusominaisuuksia.

Visio on myös helposti muokattavissa uusilla kaaviomalleilla ja laajennuksilla, joilla toiminnallisuutta voidaan laajentaa ja integroida ulkoisiin sovelluksiin, kuten muihin Office-tuoteperheen sovelluksiin. Muokattavuutensa, monipuolisuutensa ja helpon lähestyttävyytensä vuoksi Visio onkin suosittu työkalu kaavioiden tekoon useissa yrityksissä (Haikala & Märijärvi 2006 s. 86).

2.3.2. Peruskäsitteet

Kaaviomalliin kuuluvien elementtien määrittely tapahtuu kaavaindokumenteissa (*Stencil*), jotka sisältävät joukon *perusmuotoja* (*Master*) (Kuva 2.2). Perusmuodot voivat sisältää toisia perusmuotoelementtejä ja ne voivat olla joko yksinkertaisia elementtejä tai toisia elementtejä yhdistäviä elementtejä. Kaavioelementit, eli *muodot* (*Shape*), luodaan pudottamalla perusmuotoja kaavioon, jolloin ne perivät perusmuotojen graafisen ulkoasun ja ominaisuudet. Muodot sisältävät lisäksi linkin sen perusmuotoon, jol-

loin ne päivittyvät kun niiden perusmuotoon tehdään muutoksia. Tämä linkki kuitenkin katoaa esimerkiksi sisältymissuhteita purettaessa. (Microsoft 2011c.) Yleisesti perusmuodon voidaan ajatella olevan muodon *template*-elementti, eli malli, joka määrittelee muodon ominaisuudet.



Kuva 2.2. Vision peruskäsitteet ja niiden väliset suhteet.

Yksittäinen kaaviomalli Visiossa on tyypillisesti yhdessä dokumentissa (*Document*), joka voi sisältää useita sivuja (*Page*), joihin varsinaiset kaaviot piirretään. Sekä perusmuodot sisältävä kaavain että koko kaaviomallin sisältämä dokumentti ovat tyyppiä *Document*, kuten kuvan 2.2 UML-kaaviosta nähdään. Vision elementtien ulkoasu ja käyttöä käsitellään tarkemmin laajennetun Vision esittelyn yhteydessä luvussa 5.

2.3.3. Mallinnustuki

Visio 2007 sisältää joukon kaaviomalleja myös ohjelmistojen mallintamiseen. Merkittävin ja yleisimmin käytetty näistä on UML-kaaviomalli, joka tukee aktiviteetti-, kollaboraatio-, komponentti-, käyttöönotto-, sekvenssi-, tila- ja luokkakaavionotaatioita.

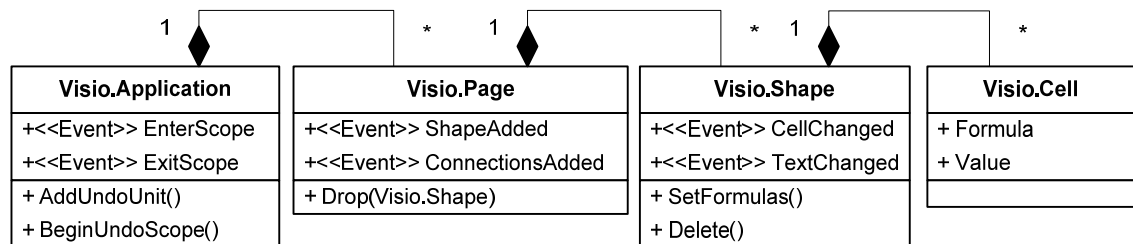
Pelkkien kaavioiden ja sen elementtien määrittelyn lisäksi UML-kaaviomalli sisältää erillisten mallien käsitteen monista muista Vision kaavioista poiketen. Näkymään piirretyistä elementeistä luodaan näkymäelementeistä erilliset mallielementit, joita voidaan selata ja muokata erillisestä malliselaimesta. Mallielementtejä voidaan käyttää uudelleen toisissa näkymissä raahaamalla niitä malliselaimesta näkymiin ja mallielementin kaikkiin näkymäelementteihin voidaan esimerkiksi navigoida helposti. Näin ollen Visio on valmis UML-kaaviomalli sisältää monia perinteisistä mallinnustyökaluista tuttuja ominaisuuksia.

UML-mallinnustuki on kuitenkin monella tapaa puutteellinen ja rajoitettu, kuten Peltonen et al. (2010) sanoo. Luodut mallit ovat sidottuja yksittäisiin Visio dokumentteihin, jolloin niitä ei voida esimerkiksi jakaa reaaliaikaisesti muihin työkaluihin, jotka tarjoisivat vaihtoehtoisia näkymiä malliin. Kaaviomalli sisältää tuen ainoastaan UML 1.4:lle, vaikka OMG on jo julkaissut siitä paljon uusia ominaisuuksia sisältävän version 2.3. Tämän lisäksi malleille ei voida tehdä minkäänlaisia eheystarkistuksia tai muita kehittyneempiä mallinnustyökalujen toimintoja. Yleisesti Visiolla mallintamiseen liittyy

myös samoja ongelmia kuin toimistotyökaluilla mallintamiseen yleensä. Näitä ongelmia käsitellään tarkemmin kohdassa 3.1.

2.3.4. Automaattiorajapinta

Vision automaattiorajapinta tarjoaa laajan oliomallin Vision ja sen elementtien, kuten muotojen ja dokumenttien, ohjelmalliseen käsittelyyn. Automaattiorajapinta on käytettävissä kaikilla *Automation*-teknologiaa (Microsoft 2011d) tukevilla ohjelmointikielillä, kuten Microsoft .NET-kielillä. Rajapinta koostuu luokista, jotka voivat sisältää joukon metodeja, tapahtumia ja ominaisuuksia. Karkeasti ottaen jokaista Vision elementtiä vastaa rajapinnassa yksi luokka, jolloin rajapinnasta muodostuu hierarkkinen rakenne Vision käyttöliittymähierarkian mukaisesti (Kuva 2.3). (Microsoft 2011e)



Kuva 2.3. Ote Vision automaattiorajapinnasta (Peltonen et al. 2010).

Kuvassa 2.3 nähdään yksinkertaistettu esimerkki automaattiorajapinnasta ja sen hierarkkisesta rakenteesta. Ylimmällä tasolla on koko Visio-sovellusta edustava *Application*-luokka, jonka dokumentit koostuvat useammasta sivusta (*Page*), jolla taas on useampia muotoja (*Shape*) ja niin edelleen. Hierarkia yltää alimmalle tasolle, eli yksittäisen muodon soluun, joka mahdollistaa käytännössä täydellisen kontrollin kaikkiin Vision sisältämiin elementteihin. (Microsoft 2011e)

Tätä rajapintaa käyttäen Vision toiminnallisuutta voidaan laajentaa halutulla tavalla erilaisten laajennuskomponenttien (2.3.5) ja *VBA*-makrojen (Visual Basic for Applications) avulla. *VBA*-makrot sisällytetään Vision dokumentteihin, ja ne soveltuvat sellaisenaan parhaiten yksittäisten monimutkaisten Visio-spesifisten toimintojen automatisointiin yksittäisessä dokumentissa. Itsenäisiä ja irrallisia laajennuskomponentteja varten Visio tarjoaa kaksi eri teknologiaa, jotka esitellään luvussa 2.3.5. (Microsoft 2011f)

2.3.5. Laajennuskomponenttitekniikat

Visio-laajennuskomponentit voidaan toteuttaa joko *add-on*- tai *COM add-in*-teknologioilla. Molemmat komponenttityypit voivat olla joko Vision kanssa samassa prosessissa ajettavia DLL-kirjastoja tai erillisiä EXE-sovelluksia. DLL-kirjastojen merkittävimpänä etuna EXE-sovelluksiin nähden on niiden suorituskyky, sillä EXE-sovellukset joutuvat ylittämään prosessirajat kommunikoidessaan Vision kanssa. (Microsoft 2011f)

Suurin ero *add-on-* ja *COM add-in-*komponenttien välillä on niiden ajaminen Visiosta. *Add-onit* voidaan käynnistää milloin tahansa ajonaikaisesti käyttäjän suorittamien toimintojen seurauksena, kuten käyttöliittymän painikkeita painamalla. *COM Add-init* sen sijaan ladataan ainoastaan kertaalleen Vision käynnistyksessä, eikä niitä voida enää sen jälkeen kutsua Vision toimesta. *COM Add-ineita* voidaan kuitenkin epäsuorasti kutsua *add-onien* tavoin lähettämällä niin sanottuja *MarkerEvent*-tapahtumia *QueueMarkerEvent*-laajennuskomponentin avulla. Tapahtumat ovat yleisesti ottaen ainoa kommunikointitapa Visiosta *COM add-inien* suuntaan, joten *COM add-inien* toiminta perustuu pitkälti automaattiorajapinnan tapahtumien kuunteluun ja niihin reagoimiseen. (Microsoft 2011f)

2.3.6. ShapeSheet

Kaikkien Vision kaavioissa esitettävien elementtien ulkoasu ja käyttäytyminen määritellään niiden *ShapeSheetissä*. *ShapeSheet* on taulukkomuotoinen rakenne, joka koostuu useisiin eri osioihin (*Section*) jaetuista riveistä (*Row*) ja niillä olevista soluista (*Cell*). Yksittäinen solu voi sisältää arvoja ja Microsoft Excel-tyyppisiä kaavoja, jotka määrittelevät jonkin *Shapen* sisältämän ominaisuuden, kuten pituuden, arvon ja käyttäytymisen. Kuvassa 2.4 nähdään erään *Shapen ShapeSheetistä* osa, joka määrittelee muun muassa *Shapen* perusdimensiot ja geometrian. (Microsoft 2011g)

Shape Transform					
Width	18.75 mm	PinX	25.625 mm	FlipX	0
Height	18.75 mm	PinY	195 mm	FlipY	0
Angle	0 deg	LocPinX	Width*0.5	ResizeMode	0
		LocPinY	Height*0.5		
User-defined Cells			Value		Prompt
User.LongFormula	IF(Width>20 mm,SETF(GetRef(User.IsLong),TRUE),SETF(GetRef(User.IsLong),FALSE))				No Formula
User.IsLong	FALSE				""
Connection Points	X	Y	DirX / A	DirY / B	Type / C
1	Width*0.5	Height*0.5	0 mm	25.4 mm	No Formula
2	Width*1	Height*0	-18.1786 mm	17.7397 mm	No Formula
3	Width*0	Height*0	18.1166 mm	17.8031 mm	No Formula
4	Width*1	Height*1	-17.8347 mm	-18.0855 mm	No Formula
5	Width*0	Height*1	17.8977 mm	-18.0231 mm	No Formula
Geometry 1					
Geometry1.NoFill	User.IsLong	Geometry1.NoLine	No Formula	Geometry1.NoShow	No Formula
Name	X	Y	A	B	C
1	MoveTo	Width*0			
2	LineTo	Width*1			
3	LineTo	Width*1			

Kuva 2.4. Visio ShapeSheet.

ShapeSheetiä voidaan käsitellä suoraan Vision käyttöliittymästä kehitystilan ollessa päällä tai ohjelmakoodista *ShapeSheet*-rajapinnan kautta. Rajapinnan kautta päästään käsiksi kaikkiin *ShapeSheetin* osioihin, riveihin ja soluihin, jolloin niitä voidaan ohjelmallisesti lukea tallennusta varten tai muokata käyttäytymään halutulla tavalla visuaalisesti.

Etenkin Excel-tyyppisten kaavojen käyttö soluissa tekee *ShapeSheetistä* tehokkaan työkalun kehittäjälle. Solut voivat lukea ja muokata toisia soluja niiden avulla, jolloin

Shapeista voidaan muokata halutulla tavalla dynaamisia ilman erillistä ohjelmakoodia. Esimerkiksi kuvassa 2 rivi *User.LongFormula* sisältää kaavan, joka asettaa sen alla olevan *User.IsLong*-rivin arvoksi *TRUE*, mikäli *Shapen* leveys ylittää 20 millimetriä. *Shapen* geometriaa esittävän *Geometry1*-osion *NoFill*-solun arvo on sidottu *User.IsLong*-rivin arvoon, jolloin *Shapen* ollessa yli 20 millimetriä leveä ei sen kyseistä geometriaa täytetä millään värillä. (Microsoft 2011g)

2.4. Suunnittelu- ja arkkitehtuurimallit

Suunnittelumallit ovat hyväksi havaittuja uudelleenkäytettäviä kuvauksia ratkaisusta tiettyihin ohjelmistojen suunnitteluun liittyviin ongelmiin. Ne perustuvat olemassa oleviin toimiviksi todettuihin suunnitteluratkaisuihin ja kannustavat hyödyntämään niiden parhaita käytäntöjä samantyyppisiin ongelmiin uudelleen myös tulevaisuudessa. Suunnittelumallit mahdollistavat lisäksi monimutkaisten ratkaisuiden dokumentoimisen ja niistä keskustelun suunnittelijoiden kesken tehokkaasti pelkästään niiden nimiä käyttäen. (Beck et al. 1996)

Ensimmäisen kirjan suunnittelumalleista julkaisi niin sanottu neljän koplana (*Gang of Four*) Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides vuonna 1994. Heidän kuuluisa teos *Design Patterns* luokittelee suunnittelumallit niiden tarkoituksen perusteella luonti-, rakenne- ja käyttäytymismalleihin. Luomismallit keskittyvät olioiden luomisprosessiin, rakennemallit luokkien tai olioiden muodostamiin suurempiin rakenteisiin ja käyttäytymismallit luokkien tai olioiden väliseen interaktioon ja vastuualueisiin. Toisaalta suunnittelumallit voidaan jakaa myös luokka- ja oliomalleihin sen perusteella, että soveltuuko malli pääasiassa luokkiin ja niiden staattisiin suhteisiin vai olioihin ja niiden ajonaikaisiin dynaamisiin suhteisiin. (Gamma et al. 1994)

Arkkitehtuurimallit puolestaan kuvaavat sovelluksen kaikkein korkeimman tason suunnitteluratkaisuja, jotka määrittelevät koko sovelluksen rakenteen. Tämä rakenne ohjaa kaikkea kehitystyötä sovelluksen komponenttien kommunikaatiosta sen laajentamiseen ja alikomponenttien suunnitteluun. Jokaisen arkkitehtuurimallin tavoitteena on saavuttaa jokin sovelluksen globaali ominaisuus, kuten käyttöliittymän mukautuvuus. (Buschmann et al. 1996.) Arkkitehtuurimalli voidaan myös nähdä suunnittelukielenä, joka tarjoaa kehittäjille sanaston ja kehyksen hyödyllisten ratkaisumallien kehittämiseksi tiettyihin ongelmiin (Monroe et al. 1997).

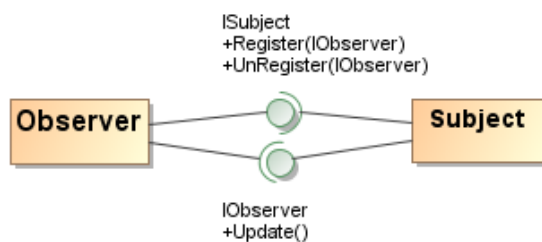
Tämän työn suunnittelussa ja toteutuksessa hyödynnettiin myös useita suunnittelu- ja arkkitehtuurimalleja, joista olennaisimmat esitellään lyhyesti seuraavissa aliluvuissa. Tarkkailija-suunnittelumallista käydään lisäksi läpi siitä .NET -teknologioilla toteutettu variaatio, johon tässä työssä toteutettu Visio-laajennuskomponentin toteutus tarkemmin ottaen nojaa.

2.4.1. Tarkkailija-suunnittelumalli

Tarkkailija-suunnittelumallia (observer pattern) hyödynnetään usein tilanteissa, joissa jokin tarkkailija (observer) on kiinnostunut jonkin toisen kohteen (subject) tapahtumisesta. Suunnittelumallin avulla voidaan saavuttaa Buschmann et al. (1996) mukaan muun muassa seuraavia hyötyjä:

- Palvelun pyytäjän (tarkkailijan) ja tarjoajan (kohteen) välistä suhdetta saadaan löyhemmäksi.
- Kohteella voi olla mielivaltaisen määrä tarkkailijoita, joiden määrä voi vaihdella ajonaikaisesti.
- Tarkkailijan ei tarvitse aktiivisesti tiedustella uutta tietoa kohteelta, joka olisi suorituskyvyn kannalta epätoivottavaa.

Olio-ohjelmoinnissa perinteinen tapa toteuttaa tarkkailija-suunnittelumalli on erillisten rajapintojen määrittäminen tarkkailijalle ja lähteelle. Lähteen rajapinta mahdollistaa tarkkailijoiden rekisteröitymisen kuuntelemaan tapahtumia sekä tämän rekisteröitymisen poistamisen. Tarkkailijan rajapinta toimii takaisinkutsurajapintana, jonka kautta lähde tiedottaa rekisteröityneitä tarkkailijoita tapahtumista. Kuva 2.5 esittää yhden mahdollisen toteutuksen suunnittelumallin rajapinnoista sekä niiden toteuttavista olioista. (Koskimies & Mikkonen 2005)



Kuva 2.5. Tarkkailija-suunnittelumallin yksinkertainen versio. Mukailtu lähteestä (Vartiainen 2010).

Käytännön toteutuksissa komponenttien välille muodostuu kuitenkin yleensä jonkinlaisia riippuvuuksia. Koskimies & Mikkosen (2005, s. 87) mukaan toteutuksissa on muun muassa otettava kantaa siihen, kuinka tapahtumat esitetään ja välitetään kuuntelijoille sekä käsitelläänkö tapahtumat asynkronisesti vai synkronisesti. Lisäksi Buschmann et al. (1996) mainitsevat suunnittelumallin ongelmakohtiksi mahdolliset runsaat ylimääräiset päivityskutsut, mikäli kohteella on paljon päivitettävää dataa, mutta tarkkailija on kiinnostunut vain pienestä osasta sitä.

2.4.2. Tarkkailija-suunnittelumalli .NET-tapahtumilla ja delegaateilla

Microsoft .NET sisältää tarkkailija-suunnittelumallin hyödyntämiseen suoraviivaisen ja helppokäyttöisemmän menetelmän tapahtumien (event) ja delegaattien (delegate) avulla. Kohteen ja tarkkailijan ei tarvitse toteuttaa erillisiä rajapintoja, vaan riittää että ne määrittelevät tapahtumat ja delegaatit. Delegaatti on funktio-osoitin, jonka tarkkailija sitoo tiettyyn kohteen tapahtumaan ja jota kohde kutsuu tapahtuman realisoituessa. Delegaatin tyyppi määritellään kohteen tapahtuman määrittelyssä, jolloin tarkkailijan tulee tietää delegaatin tyyppi ja toteuttaa sitä vastaava funktio, joka suoritetaan kohteen kutsuessa delegaattia (Ohjelma 2.1). (Microsoft 2011h)

```
// Kohde: Tapahtuman määrittely kohteen rajapinnassa.
event EventHandler<ModelPropertyChangedEventArgs> ModelModified;
// Kohde: Tapahtumasta ilmoittaminen tarkkailijoille.
this.ModelModified(this, new ModelPropertyChangedEventArgs (
    model.ThingIdentifier, property, newValue, null));

// Tarkkailija: Tapahtuman sitominen uuteen delegaattiin.
this.Database.ModelModified += new EventHandler
    <ModelPropertyChangedEventArgs>(Database_ModelModified);
// Tarkkailija: Tapahtuman käsittelijäfunktio.
void Database_ModelModified(object sender,
    ModelPropertyChangedEventArgs e);
```

Ohjelma 2.1. Esimerkki Tarkkailija-suunnittelumallista .NET -tapahtumia ja delegaatteja hyödyntäen.

Ohjelma 2.1 sisältää käytännön esimerkin tapahtumien ja delegaattien käytöstä. Kohteen *Database* tapahtumaan *ModelModified* sidotun delegaatin tyyppi esimerkissä on *EventHandler<ModelPropertyChangedEventArgs>*, jota vastaava delegaatti luodaan tapahtumaan rekisteröidytessä ja sidotaan funktioon *DataBase_ModelModified*.

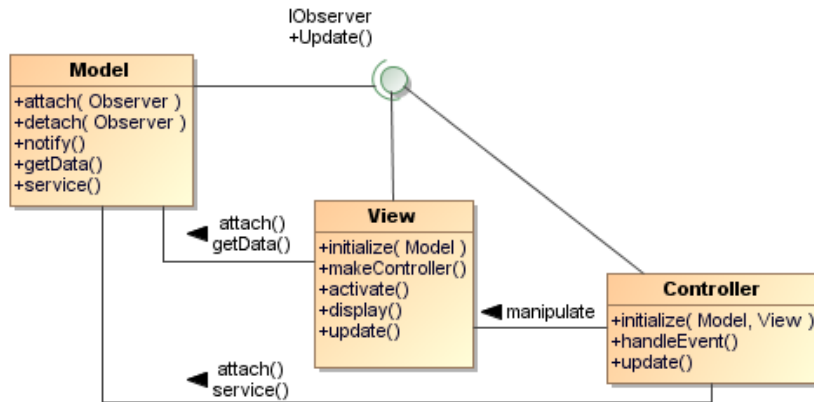
2.4.3. Malli-näkymä-ohjain -arkkitehtuurimalli

Malli-näkymä-ohjain -arkkitehtuurimallin (*MVC, model-view-controller*) perusideana on erottaa käyttöliittymä sovelluslogiikasta ja -datasta (Koskimies & Mikkonen 2005 s. 142). Buschmann et al. (1996) mukaan arkkitehtuuriratkaisulla saavutettavia etuja ovat muun muassa seuraavat:

- Sama informaatio voidaan jakaa useassa erilaisessa ikkunassa.
- Käyttöliittymää voidaan helposti muokata jopa ajonaikaisesti, eikä sen tulisi vaikuttaa sovelluksen ydintoimintaan.
- Käyttöliittymä heijastaa ohjelman uusinta tilaa välittömästi.

Arkkitehtuurimallista on olemassa useita erilaisia variaatioita, mutta olennaisesti malli jakaa sovelluksen kolmeen komponenttiin: malliin, näkymään ja ohjaimeen. Mallikomponentti sisältää sovelluksen ydininformaation ja -toiminnallisuuden, jonka eri

näkymät esittävät käyttäjälle. Näiden välissä toimii ohjaimia, jotka tulkitsevat näkymän tapahtumia, ohjaavat ne mallille ja pitävät käyttöliittymän ja mallin tilan synkronoitui-
na. Tapahtumien kuuntelu perustuu usein luvussa 2.4.1 kuvattuun Tarkkailija-
suunnittelumalliin ja jokaista käyttöliittymän näkymäkomponenttia vastaa tyypillisesti
yksi ohjain. Kuvassa 2.6 nähdään yksi variaatio malli-näkymä-ohjain -arkkitehtuurista.
(Buschmann et al. 1996)



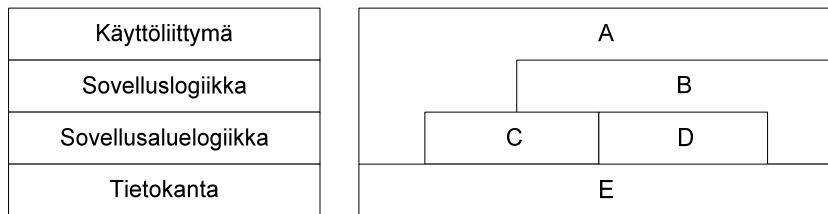
Kuva 2.6. Malli-näkymä-ohjain-arkkitehtuuri. Mukailtu lähteestä (Buschmann et al. 1996).

Kuvan 2.6 esimerkissä sekä näkymä että ohjain rekisteröityvät malliin kuuntelemaan tapahtumia. Eräs tyypillinen vuorovaikutusketju alkaa käyttäjän syötteestä, jonka ohjain havaitsee. Ohjain päivittää tapahtuman seurauksena mallia, joka ilmoittaa muutoksesta kuuntelijoille, jotka puolestaan päivittävät oman tilansa pyytämällä tuoreimman tiedon mallilta. Näkymä voi nyt tarvittaessa päivittää käyttöliittymänsä ja ohjain esimerkiksi estää tai sallia tiettyjä käyttöliittymän toimintoja. (Koskimies & Mikkonen 2005 s. 143)

2.4.4. Kerrosarkkitehtuuri

Kerrosarkkitehtuuri koostuu jonkin abstrahointiperiaatteen mukaan nousevaan järjestykseen järjestetyistä tasoista. Sen perusajatuksena on toteuttaa tietyllä tasolla oleva komponentti tai yksittäinen palvelu käyttäen hyväksi alemman kerroksen rajapinnan tarjoamia komponentteja tai palveluja. Tällainen puhdas kerrosarkkitehtuuri on kuitenkin melko harvinainen ja usein palvelukutsut joko joutuvat ohittamaan kerroksia tai kulkemaan alemmasta kerroksesta ylempään. (Koskimies & Mikkonen 2005 s. 126)

Kerrosten ohittamisten perusteena on yleisesti joko parempi suorituskyky tai yksinkertaisesti se, ettei jotakin palvelua ole tarjolla suoraan alemmalla kerroksella. Palvelukutsun suunta alemmalta kerrokselta ylempään puolestaan on vastoin arkkitehtuurin periaatteita ja johtaa alemman kerroksen riippuvuuteen yleimmästä kerroksesta. Joskus tällainen on kuitenkin välttämätöntä, ja yleisenä ratkaisuna riippuvuuden katkaisemiseksi käytetään takaisinkutsurakenteita (callback). (Koskimies & Mikkonen 2005 s. 126)



Kuva 2.7. Kuvaus ohituksia (oikealla) ja ei ohituksia (vasemmalla) sisältävästä kerrosarkkitehtuurista. Mukailtu lähteestä (Koskimies & Mikkonen 2005).

Yksi yleinen esimerkki kerrosarkkitehtuurista on neljään kerrokseen jaettu liiketoimintajärjestelmä, jossa käyttöliittymä ja sen sovelluslogiikka erotetaan laajemmasta sovellusaluelogiikasta ja tietokannasta (Kuva 2.7 vasemmalla). Ratkaisu mahdollistaa esimerkiksi uusien sovelluksien luomisen samalle sovellusalueelle suoraan kahden alimman kerroksen päälle ja tietokannan vaihtamisen kokonaan toiseen. (Koskimies & Mikkonen 2005 s. 128)

Kerrosarkkitehtuurin etuna on sen suhteellisen yksinkertainen kuvaustapa ja soveltuvuus hyvin sekä pieniin että suuriin järjestelmiin. Se kannustaa minimoivaan suunnitteluun sekä mahdollistaa sovelluskerrosten uudelleenkäytön ja vaihtamisen ilman muutoksia muihin kerroksiin, mikäli niiden rajapinnat vain suunnitellaan riittävän hyvin. Kerrosarkkitehtuurimalli toimii myös toisaalta dekompositioperiaatteena, mutta toisaalta myös sen ymmärtämistä helpottavana ryhmittelyperiaatteena. (Koskimies & Mikkonen 2005)

Merkittävin haittapuoli kerrosarkkitehtuurissa on sen suorituskyky, sillä palvelukutsuja joudutaan välittämään ylimääräisten kerrosten läpi suoran palvelukutsun sijasta. Toinen yleinen ongelma on poikkeusten käsittely, mikäli alimmilla kerroksilla syntyvät poikkeukset käsitellään vasta ylemmillä tasoilla, eikä niillä ole keinoja palautua virhetilanteesta. Tyypillinen tilanne on käyttöliittymästä käynnistetty toiminto, jonka epäonnistussa heitetty poikkeus palaa takaisin käyttöliittymäkerrokseen asti käyttäjälle mystisenä virheilmoituksena. (Koskimies & Mikkonen 2005 s. 131)

3. JOUSTAVAT MALLINNUSTYÖKALUT

Ohjelmistojen mallintaminen on luova ja iteratiivinen prosessi, jossa analysoidaan ja kommunikoidaan jonkin kohdealueen ongelmia ja niiden mahdollisia ratkaisuja. Tällainen prosessi vaatii joustavaa työkaluympäristöä, joka tukee työn eri vaiheita ja mahdollistaa haluttujen työkalujen käytön kussakin prosessin vaiheessa. Nykyiset mallinnustyökalut eivät kuitenkaan tue tätä riittävän hyvin, vaan ihmiset joutuvat mukautumaan ja joustamaan työkalujen sijasta. Tässä luvussa käsitellään mallintamistyön joustavaa luonnetta ja työkalutuen puutteen aiheuttamia ongelmia sekä esitellään vaatimuksia ympäristölle, jolla näitä ongelmia saadaan ratkaistua.

3.1. Mallintamisen joustava luonne

Ohjelmistojen mallintaminen koostuu usein monista eri vaiheista, joissa työn luonne vaihtelee suuresti. Alkuvaiheessa työ on luonteeltaan analysoivaa ja tutkivaa, kun kohdealuetta pyritään ymmärtämään ja ratkaisuvaihtoehtoja hahmotellaan vapaamuotoisilla tavoilla. Prosessin edetessä pidemmälle kohdealueesta muodostetaan formaalimpia malleja, joissa syntaksi, semantiikka ja rajoitteet ovat tiukemmin määriteltyjä. Näiden ääripäiden välillä myös mallinnustyökaluille asetetut vaatimukset muuttuvat radikaalisti työn luonteen mukana. Nykyiset työkalut eivät kuitenkaan kykene täysin vastaamaan kaikkiin näihin vaatimuksiin, vaan käyttäjät joutuvat turvautumaan useisiin eri työkaluihin, jotka vastaavat kullekin mallinnusvaiheelle ominaisiin tarpeisiin.

Etenkin luonnosteluvaiheissa työkalulta kaivataan mahdollisimman paljon ilmaisu- vapautta ja helppokäyttöisyyttä, jolloin käyttäjät turvautuvat muun muassa tavallisiin toimistosovelluksiin perinteisten mallinnustyökalujen sijasta. Mallinnustyökalut ovat toimistotyökaluihin verrattuna vaikeita oppia ja niiden käyttö on usein raskasta. Ne esimerkiksi saattavat pakottaa noudattamaan jonkin määritellyn metamallin rajoitteita, vaikka varhaisessa hahmotteluvaiheessa kohdealuetta ja sen rajoitteita ei vielä tunneta, eikä yksityiskohtaisia sääntöjä ole mahdollista edes muodostaa. Lisäksi mallinnustyökalun käyttö vaatii usein käytetyn kielen rakenteen tuntemista hyvin, jotta kaikki halutut ominaisuudet ja toiminnot löytyvät niiden käyttöliittymästä ilman ylimääräistä etsimistä. Epätäydellisiä malleja ei yleensä sallita, eikä hahmottelun luovaa luonnetta näin tueta riittävästi. Toimistosovellukset, kuten tekstinkäsittely-, lomake- ja kaaviotyökalut, puolestaan luovat monikäyttöisen alustan kaiken tyyppiselle tiedon kuvaukselle, eivätkä aseta turhia rajoitteita tai ota kantaa semantiikkaan.

Toimistosovelluksia käytettäessä törmätään kuitenkin moneen muuhun ongelmaan. Perinteisesti tämän tyyppisten sovelluksien yhteydessä käytetään tiedostomuodossa olevia dokumentteja, joiden sisältämä tieto on pääasiassa visuaalista informaatiota. Luodut

”mallit” eivät täten ole esimerkiksi käytettävissä dokumentin ulkopuolella ilman tiedon kopioimista. Useita näkymiä samaan malliin ei myöskään tästä syystä voida helposti muodostaa perinteisten mallinnustyökalujen tapaan. Lisäksi mallin eheyden ylläpitäminen on manuaalista ja pienenkin muutoksen päivittäminen kaikkialle voi olla työlästä. Jos ohjelmistotyökalulla tuotettu tieto halutaan vielä lopuksi siirtää varsinaiseen mallinnustyökaluun osaksi formaalimpaa mallia, on työ jälleen manuaalista ja jäljitettävyyss tiedon alkuperään menetetään.

Ohjelmistojen mallinnustyökalutuessa on siis selkeästi tarve ympäristölle, jolla saadaan yhdistettyä sekä perinteisten mallinnustyökalujen että toimistotyökalujen parhaat ominaisuudet. Tällä tavalla koko mallinnusprosessissa tuotettu tieto saadaan jo varhaisessa vaiheessa työkalutuen piiriin, jolloin epäformaalissakin muodossa oleva tieto voi olla osana mallia, uudelleenkäytettävissä ja jäljitettävissä. Seuraavissa luvuissa esitellään vaatimukset mallinnusympäristölle, jolla tähän tavoitteeseen voidaan päästä. Vaatimukset käsitellään sekä kokonaisen työkaluympäristön että yksittäisen ympäristöön liittyneen mallinnustyökalun näkökulmasta.

3.2. Joustavan mallinnustyökaluympäristön vaatimuksia

Vaatimusten lähtökohtana on idea integroida organisaatiossa yleisesti käytössä olevat työkalut niin, että niiden kunkin parhaita ominaisuuksia hyödyntämällä ja laajentamalla saadaan aikaan monipuolinen ja joustava mallinnusympäristö. Tämän ympäristön tulee mahdollistaa kuhunkin mallinnusprosessin vaiheeseen sopivimman työkalun käytön – esimerkiksi käyttäjille valmiiksi tutun piirto-ohjelman käytön luokkakaavioiden muokkaamisen ja tekstinkäsittelytyökalun hyödyntämisen käyttötapausten kuvaamisessa tai raporttien luomisessa. Järjestelmän tulee tukea myös kollaboratiivista työskentelyä, jolloin monet käyttäjät voivat samaan aikaan käsitellä samaa mallitietoa ja nähdä toisten käyttäjien tekemät muutokset reaaliaikaisesti sekä samassa että muissa mallin näkymissä. Vaatimukset voidaan jakaa työkalu- ja tietointegraatioihin, jotka esitellään tarkemmin seuraavissa kohdissa 3.2.1 ja 3.2.2.

3.2.1. Työkaluintegraatio

Työkalut tulee integroida ympäristöön laajentamalla niitä tarvittavilla mallinnusominaisuuksilla ja integraatiolla muihin ympäristön komponentteihin. Ne toimivat järjestelmässä vain yksittäisinä näkyminä malliin ja mahdollistavat mallintamisen pääosin niiden valmiiksi tarjoamien toimintojen avulla.

Työkaluympäristön hallintaan on oltava keskitetty työkalu, jonka kautta työkaluja sekä ympäristön malleja voidaan hallita. Hallintatyökalun tulee olla koko ajan nopeasti saatavilla siten, että sen kautta voidaan käynnistää haluttu työkalu mallintamista varten missä tahansa työvaiheessa. Hallintatyökalun tavoin kaikki ympäristön yhteiset komponentit tulee eriyttää työkaluista ja työkalut säilyttää mahdollisimman yksinkertaisina ja alkuperäisen kaltaisina.

Työkaluintegraation kontrolliosuuden tulee tukea toteutusteknologisesti heterogeenista joukkoa työkaluja, jotka voivat erota muun muassa toteutuskielten osalta toisistaan. Lisäksi uusien työkalujen liittämisen tulee olla helppoa ja mahdollista ajonaikaisesti niin, ettei olemassa oleviin komponentteihin aiheudu muutoksia.

3.2.2. Tiedon integraatio

Mallien tulee olla kaikkien ympäristön käyttäjien ja työkalujen saatavilla, joten ne tulee tallentaa johonkin jaettuun paikkaan, kuten keskitettyyn tietokantaan. Tämän keskitetyn tietokannan tulee myös huolehtia tiedon varmistuksesta, suojauksesta ja käyttöoikeuksien hallinnasta. Lisäksi kaiken tiedon tulee olla usean käyttäjän käytettävissä samaan aikaan ilman merkittäviä viiveitä tai lukituksia. Keskitetystä tiedon sijainnista huolimatta käyttäjien tulee pystyä tarvittaessa työskentelemään ja muokkaamaan malleja myös ilman jatkuvaa yhteyttä tähän tietokantaan.

Kaiken käyttäjän tekemän tiedon tulee tallentua automaattisesti ilman käyttäjän erillistä tallentamista ja tulla osaksi käsiteltävää mallia. Malli sisältää varsinaisen mallitiedon lisäksi näkymätiedon, kuten esimerkiksi näkymäelementtien graafisen ulkoasun, joka on usein mallitiedon ymmärtämisen kannalta merkittävää. Sekä malli- että näkymätiedon tulee olla aina käyttäjien saatavilla, ja niiden tulee päivittyä kaikkiin järjestelmän näkymiin välittömästi tietokannasta.

Samaan malliin tulee voida olla useita eri näkymiä, jolloin mallien tulee olla erillään näkymätiedosta. Ympäristön tukemien mallinnuskielten kuvaukset säilytetään myös samassa tietokannassa, mutta erillään malli- ja näkymätiedosta. Mallinnuskieliä tulee voida luoda ja päivittää tietokantaan pienellä vaivalla.

3.3. Mallinnustyökalun vaatimuksia

Luvussa 3.1 esitellyt ongelmat ja luvussa 3.2 asetetut koko työkaluympäristön vaatimukset muodostavat yksittäiselle työkalulle joukon vaatimuksia, jotka käydään tässä luvussa läpi. Vaatimukset on jaoteltu mallinnustyökalun käyttöön ja työkaluympäristöön liittyviin vaatimuksiin. Lopuksi esitellään myös tässä työssä toteutettua Visio-laajennusta koskevat erityisvaatimukset.

3.3.1. Käytettävyys ja ominaisuudet

Mallinnustyökalujen tulee ensisijaisesti tukea työtä, eivätkä ne saa lisätä tarpeettomia tai toisteisia työvaiheita. Niiden tulee sallia vapaamuotoisten elementtien luomisen osaksi mallia kaikilla työkalun olemassa olevilla toiminnoilla ja sallia haluttaessa rikkinäistenkin mallien luomisen.

Mallinnuskäyttöön muokatun työkalun käyttöliittymän ja käytettävyyden tulisi säilyä ennallaan. Käyttöliittymää tulee laajentaa ja muokata vain tarvittaessa niiltä osin, joilla tuetaan mallintamiseen liittyviä toimintoja. Tällaisia ovat esimerkiksi mallitietoa näyttävät paneelit, joita voidaan liittää käyttöliittymän oheen. Vastaavasti työkalun val-

miit ominaisuudet, toiminnot ja tavat käsitellä sen sisältämiä elementtejä säilytetään mahdollisimman koskemattomana, jotta työkalujen parhaita hyötyjä ei menetetä. Olenaisia säilytettäviä perustoiminnallisuuksia ovat muun muassa työkalujen tavat luoda uusia elementtejä, kopioida, poistaa, nimetä uudelleen sekä kumota ja tehdä toimintoja uudelleen.

Käytettävyyden säilyttämiseksi suorituskyvyn on oltava riittävä, vaikka malli- ja näkymätieto tallennetaan tietokantaan välittömästi ja tietoa päivitetään tietokannasta näkymään reaaliaikaisesti. Käyttöliittymän vasteajan tulisi olla niin lyhyt, että se ei häiritse luovaa työskentelyä liikaa. Ideaalitapauksessa eroa työkalun normaaliin käyttöön ei tulisi havaita, vaikka samaa näkymää muokataan yhtäaikaisesti monen käyttäjän toimesta. Poikkeuksellisen pitkistä latausoperaatioista on aina syytä ilmaista käyttäjälle selkeästi, jotta hän ei turhaan odota käyttöliittymältä välitöntä vastetta. Tarkkoja numeerisia vasteaikaavaatimuksia ei yksittäiselle työkalulle aseteta, sillä vasteaika on riippuvainen myös ympäristön tietokannan ja työkaluintegraation aiheuttamista viiveistä.

3.3.2. Osana ympäristöä

Merkittävin vaatimus yksittäiselle mallinnustyökalulle on sen toimivuus yhteen muun ympäristön kanssa. Työkaluintegraation näkökulmasta työkalun tulee tukea integraatiossa käytettyjä arkkitehtuurisia ratkaisuja ja kommunikaatiota muiden ympäristön työkalujen, kuten hallintatyökalun kanssa. Tietointegraation puolesta työkalun tulee käyttää ympäristössä käytettyä yhteistä tietokantaa ja vastata ympäristön asettamiin tavoitteisiin käyttäjän luoman tiedon automaattisesta ja reaaliaikaisesta päivittämisestä tietokantaan sekä sieltä takaisin työkalunäkymään.

Yhtenä ympäristön työkaluintegraation vaatimuksena on erottaa ympäristön yhteiset toiminnallisuudet erilleen työkaluista ja säilyttää työkalut mahdollisimman yksinkertaisina. Työkaluihin tehtävissä mallinnuslaajennuksissa näin ollen on pyrittävä minimalisitiin toteutuksiin niin, että ne toimivat vain välikomponentteina alkuperäisten työkalujen ja tietokannan välillä. Niiden tehtävänä on reagoida käyttäjien tekemiin toimintoihin työkalun käyttöliittymässä ja pitää näkymän tila synkronoituna tietokannassa sijaitsevan mallin kanssa.

3.3.3. Visio-laajennus

Työkaluympäristön yleisten vaatimuksien lisäksi Visio-mallinnustyökalulle asetettiin teollisuusyhteistyökumppanin esittämiä Visio-spesifisiä vaatimuksia ja toiveita. Nämä käydään lyhyesti läpi seuraavassa.

Visio on tarkoitettu kaavioiden piirtämiseen ja sisältää hyvät piirto-ominaisuudet, joten sillä tulee voida piirtää ja muokata ympäristön tietokannassa olevia malleja niitä hyväksi käyttäen. Sen tulee tukea uusien kielien lisäämistä mahdollisimman pienellä vaivalla ja mielellään ilman ohjelmakoodin muokkausta. Mallinnuskieliä tarvitaan useita ja ne muuttuvat tarpeiden mukaan, joten laajennuksen ydintoteutuksen tulisi olla riittävän geneerinen, jotta kielimuutokset eivät aiheuttaisi muutoksia toteutukseen. Perus-

vaatimuksena laajenukselle on tuki UML:n yleisimmille kaaviotyypeille, kuten luokka-, tila-, aktiviteetti-, sekvenssi- ja käyttötapauskaavioille. Käytännössä laajennuskomponentin tulee kuitenkin tukea kaikkia graafisia domain-spesifisiä mallinnuskieliä. Tämän lisäksi Vision omilla kaaviomalleilla luotujen UML-kaavioiden tuominen osaksi järjestelmää on haluttu ominaisuus.

Mallinnusominaisuuksien lisäksi malleja tulee voida katselmoida ja kommentoida vapaasti työkalujen käyttöliittymässä. Mallin näkymäelementtien visuaalista esitystä tulee voida muokata vapaasti Vision valmiilla ominaisuuksilla, kuten värityksillä, jonka lisäksi työkalun tulee tarjota valmiita väritystiloja esimerkiksi katselmointien havainnollistamiseen. Työkalun tulee lisäksi tarjota tarvittavissa tilanteissa automaattiset elementtien asemointiominaisuudet tai muuten helppokäyttöiset menetelmät elementtien asemointiin, kuten ryhmittelyyn ja liikuttamiseen.

4. TRINITY – MALLINNUS- JA TYÖKALUYMPÄRISTÖ

Trinity on Tampereen teknillisen yliopiston Ohjelmistotekniikan laitoksella kehitetty mallinnus- ja työkaluympäristö. Tässä luvussa esitellään yleisesti tämä ympäristö sekä sen tarjoamat palvelut siinä oleville työkaluille.

4.1. Yleistä

Trinity-ympäristöllä vastataan joustavan mallinnuksen haasteisiin integroimalla useita valmiiksi käytössä olevia työkaluja ja laajentamalla niitä tarvittavilla mallinnusominaisuuksilla. Näitä työkaluja ovat muun muassa Microsoft Office-tuoteperheen toimistotyökalut, jotka ovat käyttäjille ennestään tuttuja ja helposti laajennettavissa halutuilla toiminnallisuuksilla.

Ympäristön työkalujen pyrkimyksenä on vastata luvussa 3 esiteltyihin mallinnustyökalujen vaatimuksiin. Ne toimivat osana ympäristöä tarjoten vaihtoehtoisia näkymiä yhteisessä Trinity-tietokannassa sijaitseviin malleihin. Jokaisella työkalulla on tarkoitus suorittaa sellaisia työvaiheita ja toimintoja, joita varten se on olemassa ja joissa se on parhaimmillaan. Olennaista on kuitenkin se, että kaikki käsiteltävä tieto tallennetaan tietokantaan osaksi jotakin mallia. Lisäksi tallennus suoritetaan automaattisesti heti käyttäjän toimintojen jälkeen, joten erillistä tallennustoimintoa ei missään ympäristön työkalussa vaatimusten mukaisesti tarvita.

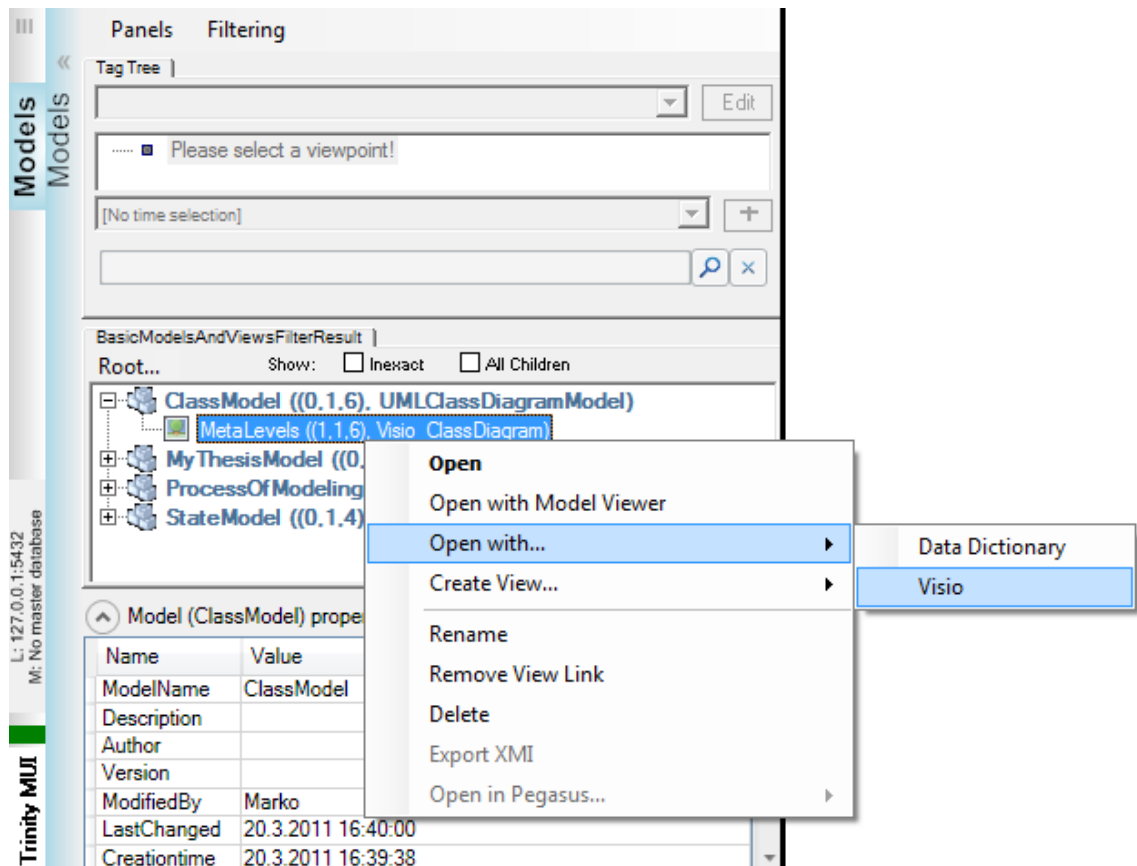
4.2. Ympäristön peruspalvelut

Trinityssä yksittäisten mallinnustyökalujen vastuulla on näkymien tarjoaminen malliin, mutta yleistä mallien hallintaa niillä ei voida tehdä. Malleja sekä niiden näkymiä luodaan ja hallitaan erillisen hallintatyökalun kautta, jonka kautta mallit avataan näkymien kautta muokattaviksi mallinnustyökaluihin. Työkaluihin voidaan lisäksi liittää yleiskäyttöinen Trinityn Informaatiopaneeli, jonka kautta voidaan lukea ja muokata suoraan lomakemuotoista mallitietoa tietokannasta.

Hallintakäyttöliittymä ja Informaatiopaneeli yhdessä tarjoavat yksittäisille työkaluille tarvittavat palvelut, jotta niiden avulla voidaan mallintaa Trinity-ympäristössä. Nämä komponentit esitellään seuraavaksi tarkemmin kohdissa 4.2.1 ja 4.2.2.

4.2.1. Hallintakäyttöliittymä

Trinity-ympäristön hallinta tehdään Management User Interface – hallintatyökalulla (MUI) (Kuva 4.1). MUI:lla voidaan luoda, poistaa, hakea ja organisoida tietokannassa olevia malleja ja näkymiä sekä avata niitä tuettuihin työkaluihin. Lisäksi sen kautta hallitaan ympäristöön liitettyjen työkalujen ja tietokantayhteyden asetuksia.



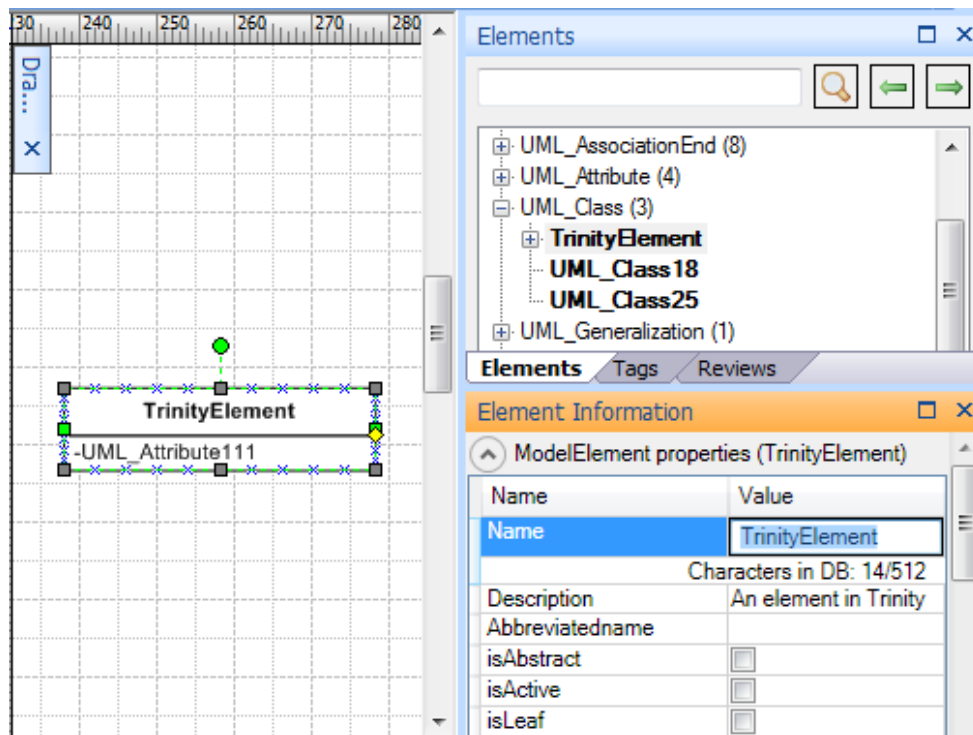
Kuva 4.1: Trinityn hallintakäyttöliittymä (MUI) ja näkymän avaaminen malliselaimesta.

Tämän työn kannalta olennaisin toiminnallisuus MUI:ssa on mallin näkymien avaaminen työkaluun. Kuva 4.1 sisältää esimerkin luokkakaavionäkymän avaamisesta Microsoft Visioon. MUI:n malliselaimesta valitaan haluttu malli ja sen alta haluttu näkymä, joka hiiren kakkospainikkeen valikosta voidaan joko avata oletustyökaluun ”Open”-toiminnolla tai ”Open with...”-toiminnolla johonkin tuetuista työkaluista. Kaikilla näkymillä on tietokannassa määriteltynä näkymätyyppi, joka riippuu työkalusta, jolla näkymää voidaan käsitellä. Yksi työkaluista on oletustyökalu, ja esimerkiksi kuvassa olevan näkymätyypin *Visio_ClassDiagram* tapauksessa tämä on Microsoft Visio.

Malliselaimen lisäksi MUI sisältää alareunassa Trinityn Informaatiopaneelin, joka mahdollistaa mallipuusta valittujen mallien ja näkymien ominaisuuksien muokkaamisen. Kuvassa 4.1 MUI on avatussa tilassa, mutta se voidaan sulkea näytön vasempaan laitaan ohueksi paneeliksi, josta se voidaan laajentaa näkyviin nopeasti milloin tahansa ympäristöä käytettäessä. Sieltä voidaan halutessa muun muassa raahata ja pudottaa elementtejä avoimiin työkaluihin sekä käynnistää katselmointisessioita käsiteltäviksi.

4.2.2. Informaatiopaneeli

Työkaluympäristö tarjoaa itsenäisen Informaatiopaneelin lomakemuotoisen mallitiedon lukemiseen ja muokkaamiseen. Paneeli voidaan liittää osaksi ympäristössä olevien työkalujen käyttöliittymiä, jossa se näyttää kulloinkin valittuna olevien elementtien tietoja (Kuva 4.2). Informaatiopaneelin olennaisena roolina työkaluissa on esittää mallin ja sen elementtien malli-informaatiota, kun työkalujen näkymät itse esittävät näiden näkymäinformaatiota, kuten sijainnit ja suhteet toisiinsa visuaalisesti.



Kuva 4.2: Trinity-Informaatiopaneeli liitettynä ympäristön työkaluun.

Paneeli sisältää kirjoitushetkellä neljä erilaista välilehdille jaettua perusikkunaa, joista jokaisella on selkeä vastuualueensa. Nämä ikkunat nähdään myös kuvassa 4.2, ja ne ovat:

- Mallielementtiselain (*Elements*), joka listaa avoimena olevan mallin kaikki mallielementit puumaisena hierarkiana. Selaimen kautta käyttäjän on mahdollista muun muassa etsiä elementtejä tai raahata ja pudottaa ympäristön työkaluihin uusia näkymäelementtejä olemasta olevista mallielementeistä.
- Yksittäisen elementin informaation näyttävä ikkuna (*Element Information*), joka esittää työkalunäkymässä kulloinkin valittuna olevan elementin näkymä- ja malli-informaation, sekä muun muassa sen viittaukset muihin elementteihin. Kun mitään ei ole valittuna, näyttää ikkuna käsittelyn alla olevan mallin ja näkymän informaation.
- Avainsanaikkunan, josta käyttäjä voi nähdä ja muokata valittuna oleviin elementteihin liitettyjä avainsanoja (*Tags*).
- Katselmointi-ikkuna, joka mahdollistaa katselmointitilojen hallinnan (*Reviews*).

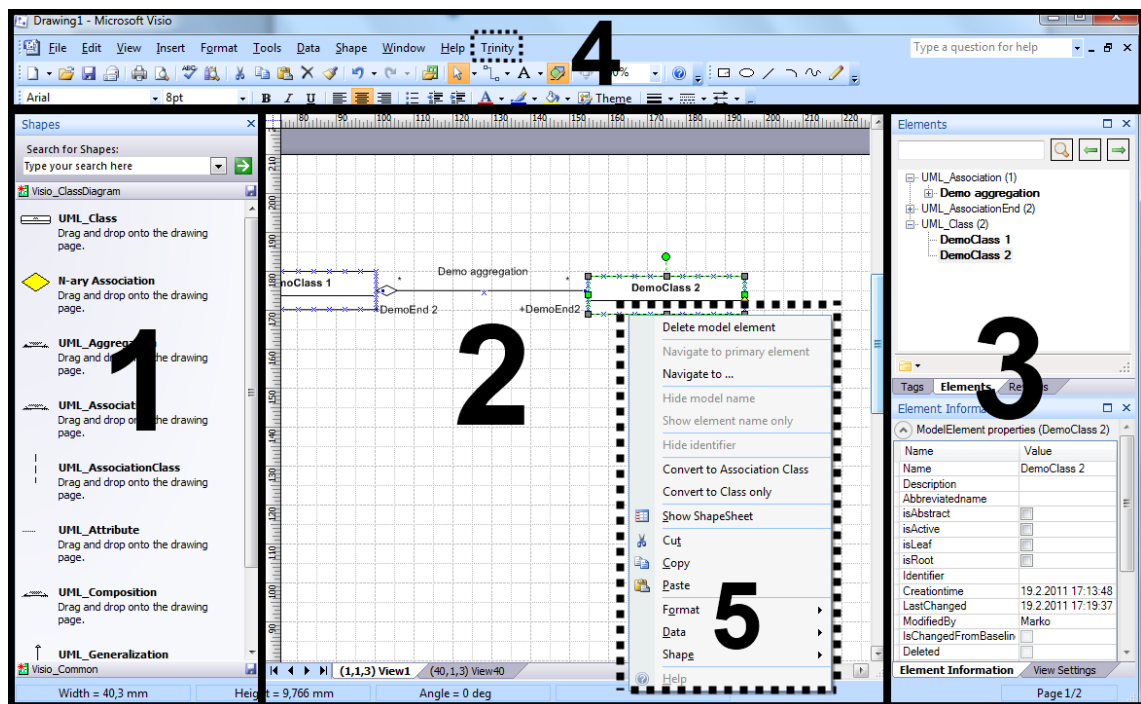
Informaatiopaneeli voi näiden perusikkunoiden lisäksi sisältää muita työkalun itse siihen lisäämiä ikkunoita. Ikkunat ovat täysin itsenäisiä komponentteja, jotka voidaan tarvittaessa myös irrottaa Informaatiopaneelistä muuhun käyttöön.

5. LAAJENNETTU MICROSOFT VISIO

Työssä suunniteltiin ja toteutettiin Microsoft Visioon laajennus, jolla Visiosta saadaan Trinity-ympäristössä toimiva joustava graafinen mallinnustyökalu. Tässä luvussa esitellään laajennetun Vision käyttöliittymä ja yleisimmät toiminnot, joilla mallintamista voidaan suorittaa.

5.1. Käyttöliittymä

Vision käyttöliittymä Trinity-laajennuksen kanssa on esitetty seuraavassa kuvassa (Kuva 5.1). Kuvaan on merkitty käyttöliittymän viisi merkittävintä osaa, joista alueet yksi ja kaksi kuuluvat perinteiseen Visio-käyttöliittymän, kun taas alueet kolme, neljä ja viisi sisältävät laajennuksen lisäämiä komponentteja.



Kuva 5.1. Laajennetun Vision käyttöliittymä.

Alue yksi on kaavainikkuna. Tämä ikkuna sisältää perusmuotoja, joita käyttäjä voi pudottaa piirtoalueelle (Kuva 5.1, alue 2) hiirellä raahaamalla. Kaavainikkuna voi sisältää useita kaavaimia, joista jokainen sisältää yhteen mallinnuskieleen liittyvät elementit. Kaavaimista vain yksi voi olla kerrallaan avoimena, jolloin muut ovat luhistuneina aktiivisen kaavaimen ylä- tai alapuolella. Oletuksena avoimena on sillä hetkellä aktiivisena olevan kaavion kieleen liittyvä kaavain.

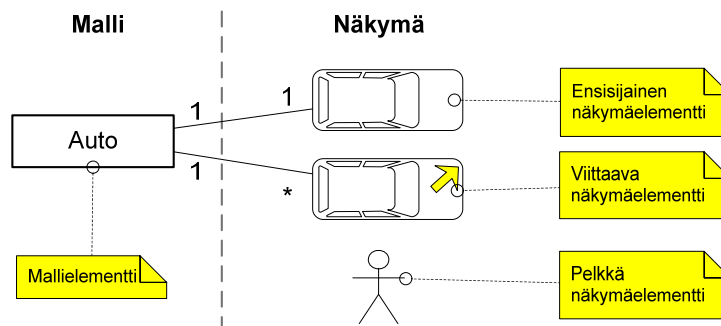
Alue kaksi sisältää piirtoalueen, johon kaaviot eli graafiset näkymät malliin muodostetaan. Kaavioita voidaan piirtää kuten Visiossa normaalisti, eli pudottamalla perusmuotoja kaavainikkunasta (Kuva 5.1, alue 1) piirtoalueelle tai piirtämällä vapaamuotoisia elementtejä Vision piirtotyökaluilla (katso 5.3.2). Alueella voi lisäksi olla useita sivuja (*Page*), joista jokainen sisältää yhden Trinity-ympäristön kaavion. Sivuja vaihdetaan ikkunan alareunassa sijaitsevista välilehdistä, joiden teksti ilmaisee sen sisältämän näkymän nimen ja johon kirjoittamalla tätä nimeä voidaan vaihtaa.

Alueella kolme sijaitsee kohdassa 4.2.2 esitelty Trinityn Informaatiopaneeli. Paneeli reagoi Visiossa tehtäviin elementtivalintoihin ja näyttää tämän elementin tiedot. Perusikkunoiden lisäksi laajennus lisää Informaatiopaneeliin ”View Settings”-ikkunan, jonka kautta voidaan muokata erityyppisiä tilapäisiä väritystiloja ja muita kaavion esitystapaan liittyviä asetuksia (katso kohta 5.3.7). Ikkuna ei muista ikkunoista poiketen liity mallitiedon esittämiseen, vaan ainoastaan Vision näkymäasetuksiin. Informaatiopaneeli sijaitsee kuvassa 5.1 käyttöliittymän oikeassa reunassa, mutta sekä paneeli että sen sisältämät ikkunat ovat vapaasti käyttäjän siirrettävissä ja telakoitavissa jokaiseen Vision käyttöliittymän reunaan. Vaihtoehtoisesti paneeli ja ikkunat voi leijua täysin vapaana ikkunaan Vision pääikkunan sisällä.

Vision yläreunassa sijaitsevaan työkalupalkkiin (Kuva 5.1, alue 4) on Vision normaaliin valikkojen lisäksi lisätty oma Trinity-valikko. Kaikki Trinity-kohtaiset aputoiminnot on lisätty vain ja ainoastaan tähän työkalupalkin valikkoon ja muita työkalupalkin valikoita laajennus ei muokkaa. Työkalupalkin lisäksi hiiren oikealla painikkeella toimivat tilannevalikot (Kuva 5.1, alue 5) sisältävät monia Trinity- ja elementtikohtaisia toimintoja, kuten navigoinnin muihin elementteihin.

5.2. Malli- ja näkymäelementit

Trinityssä näkymäelementtejä voi olla useita yhtä mallielementtiä kohden, kuten mallinnusympäristöissä yleensä. Erityyppisille näkymäelementeille on Trinityssä yleisesti käytössä omat terminsä, jotka on esitelty kuvassa 5.2.



Kuva 5.2. Trinityn mallinnustyökaluissa käytettyjen malli- ja näkymäelementtien termit.

Uutta mallielementtiä luotaessa mallinnustyökalussa luodaan niille ensimmäinen niitä esittävä näkymäelementti, josta tulee ensisijainen näkymäelementti. Ensisijaisia

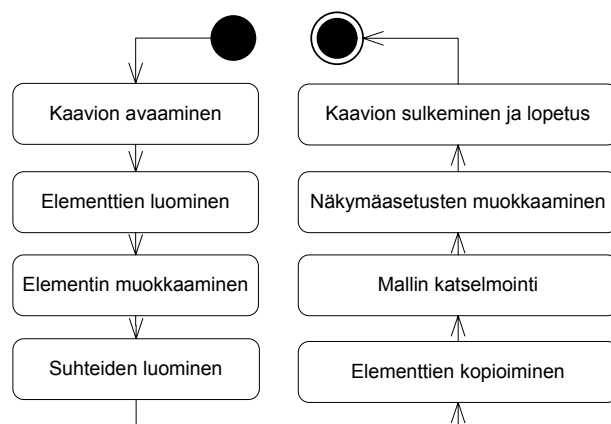
näkymäelementtejä voi olla korkeintaan yksi ja se merkitsee mallielementin kotipaikan mallissa. Mallielementtiä hallitaan ensisijaisen näkymäelementin kautta ja se voidaan esimerkiksi poistaa kokonaan vain poistamalla ensisijaisen näkymäelementin. Ympäristön periaatteiden mukaan jokaisella mallielementillä on oltava olemassa aina yksi ensisijainen näkymäelementti, jonka kautta siihen voidaan päästä käsiksi työkalunäkymien kautta.

Viittaavia näkymäelementtejä voi puolestaan olla lukematon määrä missä tahansa näkymässä ympäristössä. Niiden kautta ei kuitenkaan voida esimerkiksi poistaa mallielementtiä - ne vain viittaavat siihen nimensä mukaisesti. Viittaavat näkymäelementit esitetään Visiossa keltaisella nuolella elementin oikeassa yläreunassa.

Näkymäelementti voi olla myös viittaamatta mihinkään mallielementtiin, jolloin kyseessä on pelkkä näkymäelementti. Tällaisia ovat muun muassa vapaamuotoiset graafiset elementit, joita käyttäjä voi piirtää Vision yleisillä piirtotyökaluilla.

5.3. Työkalulla mallintaminen

Mallintaminen laajennetulla Visiolla ei pääpiirteissään eroa normaalista Vision käytöstä, mutta aikaisemmin esiteltyt vaatimukset sekä Trinity-ympäristön ominaisuudet ja suunnitteluperiaatteet tuovat siihen joitakin uusia ominaisuuksia ja lisävaiheita. Tässä luvussa esitellään laajennetulla Visiolla mallintamisen perusvaiheita (Kuva 5.3), jotka suorittamalla käyttäjä voi luoda yksinkertaisen kaavion. Vaiheiden tarkoitus on antaa kokonaiskuva siitä, miten mallintamistyö toteutetulla työkalulla ja Trinity-ympäristöllä käytännössä tehdään.



Kuva 5.3. Esimerkki yksinkertaisesta kaavionluomisprosessista Visio-mallinnustyökalulla.

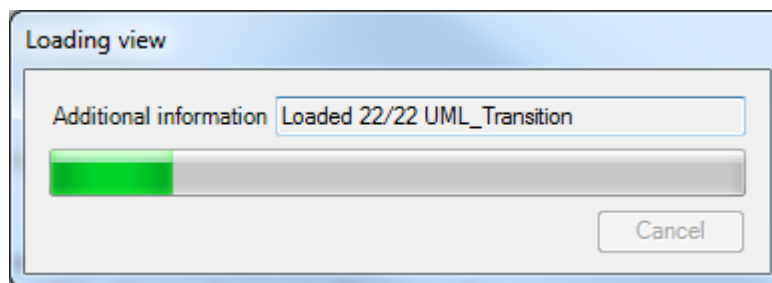
Esimerkkiprosessin jokainen vaihe käydään seuraavaksi yksitellen läpi, kukin omassa aliluvussa. Vaiheiden sisältämiin toimintoihin voi olla useita vaihtoehtoisia suoritustapoja, joista vain tärkeimmät esitellään. Toiminnallisuudet vaihtelevat hieman näkymätyypeittäin eikä esimerkki ota kantaa kaavion tyyppiin, mutta pääpaino on UML:n luokkakaavioiden luomisessa. Koska kaavioiden luominen ei kuulu mallinnustyökalun

itsensä vastuulle, oletetaan tyhjän kaavion olevan prosessin alussa valmiiksi luotuna tietokannassa.

5.3.1. Kaavion avaaminen

Käyttäjä voi avata kaavion Trinity-ympäristössä usealla eri tavalla. Yleisimmin tämä tehdään hallintakäyttöliittymän malli- ja näkymäselaimesta (4.2.1), Vision Trinity-työkaluvalikosta ”Open view”-toiminnolla tai Trinity URI-osoitteella esimerkiksi internet-selaimen linkistä. Mikäli Visio ei ole valmiiksi käynnissä, käynnistetään se automaattisesti, jonka jälkeen kaavio ladataan tietokannasta aktiiviseksi näkymäksi Visioon. Työkalupalkissa sijaitsevan ”Open view”-toiminnolla näkymän avaaminen edellyttää kuitenkin Vision olemista valmiiksi käynnissä.

Näkymän lisäksi sen näkymätyyppiin liittyvät *template*-elementit asetetaan aktiiviseksi kaavainikkunaan ennen itse näkymän avaamista. Mikäli avattava kaavio on Visiossa ensimmäinen kyseistä näkymätyyppiä, ladataan *template*-elementit tietokannasta. Uuden kaavion lataaminen Visioon on suhteellisen raskas operaatio, etenkin jos kaavio sisältää suuren määrän näkymäelementtejä. Tästä syystä käyttäjälle esitetään latauksen edistymistä ilmaiseva latauspalkki (Kuva 5.4).

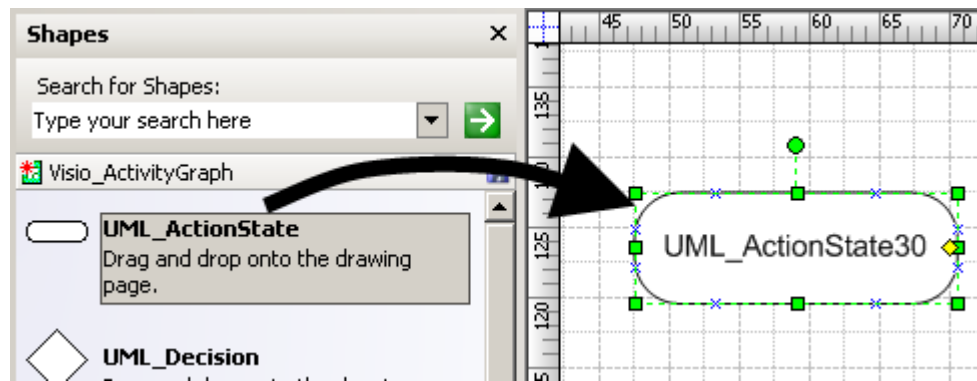


Kuva 5.4. Kaavion latauksen edistymistä ilmaiseva latauspalkki.

Lopputuloksena käyttäjällä on edessään valittu kaavio näkymäelementteineen sekä näkymätyyppiin liittyvät perusmuodot aktiivisena kaavainikkunassa. Mikäli jokin näkymäelementeistä on jollakin tavalla viallinen, esitetään se punaisella värillä. Eräs esimerkki tällaisesta on näkymäelementti, jonka mallielementti on poistettu ja sitä ei löydy tietokannasta.

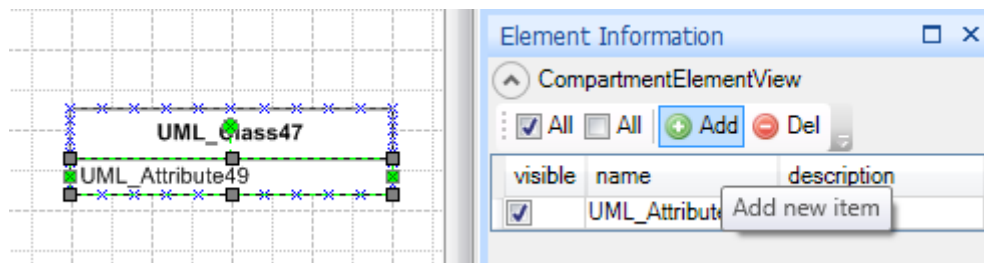
5.3.2. Elementin luominen

Uusi elementti luodaan yksinkertaisesti raahaamalla hiirellä halutun tyyppinen perusmuoto Vision kaavaimesta piirtoalueelle (Kuva 5.5). Käyttäjän pudottaessa tilaa esittävän perusmuodon luodaan tietokantaan kokonaan uusi tilaa esittävä instanssi elementistä, sisältäen sekä näkymä- että mallielementin. Pudotetusta näkymäelementistä tulee mallielementin ensisijainen näkymäelementti.



Kuva 5.5. Uuden elementin luominen hiirellä raahaamalla kaavaimesta.

Viittaavia näkymäelementtejä voidaan luoda raahaamalla mallielementtejä hiirellä ulkoisista komponenteista, kuten hallintakäyttöliittymästä tai Informaatiopaneelistä (Kuva 5.6). Lisäksi hallintakäyttöliittymästä voidaan pudottaa esimerkiksi kokonaisia näkymiä, jolloin käyttäjä valitsee dialogista haluamansa perusmuodon, joka luodaan edustamaan pudotettua näkymää kohdenäkymässä. Tällä mekanismilla mahdollistetaan olemassa olevien elementtien, näkymien ja käytännössä minkä tahansa Trinity-järjestelmän elementin uudelleenkäyttö ja niihin viittaaminen työkalunäkymissä. Toinen tapa luoda viittaavia näkymäelementtejä on elementtien kopioiminen, joka on esitelty kohdassa 5.3.5.



Kuva 5.6. Lapsielementin luominen Informaatiopaneelissa.

Lapsielementit, kuten UML-luokan attribuutit, voidaan luoda mallinnustyökalulla kahdella tavalla. Yleisimmin ne luodaan Informaatiopaneelissa ”Add”-toiminnolla, kun attribuuttiosio luokasta on valittuna (Kuva 5.6). Toinen tapa on luoda lapsielementti kuten muutkin elementit ja pudottaa sen tämän jälkeen haluttuun isäelementtiin, kuten luokan attribuuttiosioon.

Mallinnuskieleen liittyvien peruselementtien lisäksi käyttäjä voi luoda mitä tahansa vapaamuotoisia elementtejä osaksi mallia käyttäen Vision piirtotyökaluja. Näistä luodaan malliin oletuksena vain pelkät näkymäelementit, mutta Trinity-työkalupalkin valikosta se voidaan myöhemmin asettaa viittaamaan johonkin olemassa olevaan mallielementtiin ”Change modelement”-toiminnolla.

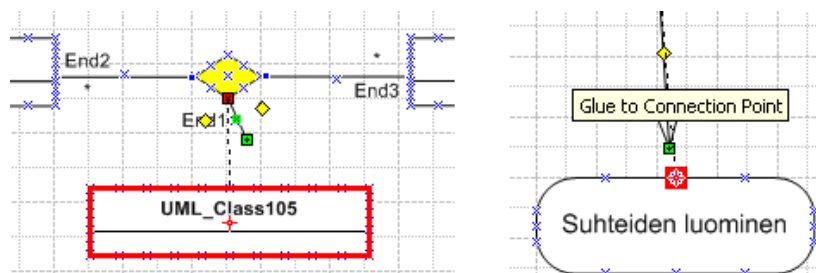
5.3.3. Elementin ominaisuuksien muokkaaminen

Sekä malli- että näkymäelementeillä on molemmilla ominaisuuksia, joita voidaan muokata mallinnustyökalussa. Kumpiinkin elementteihin liittyy kielessä määriteltyjä perusominaisuuksia kuten nimi, jonka lisäksi näkymäelementeillä on työkalukohtaisia graafisia ominaisuuksia, kuten sijainti ja koko.

Ominaisuuksia voidaan muokata joko kaaviosta tai Informaatiopaneelistä. Kaavion kautta muokataan pääasiallisesti näkymäelementin graafisia ominaisuuksia, kuten sijaintia ja kokoa, kun taas Informaatiopaneeli mahdollistaa tietokannassa määriteltyjen lomakemuotoisten ominaisuuksien muuttamisen. Osa tietokannassa määritellyistä mallitai näkymäelementin ominaisuuksista on sidottu myös graafisiin elementteihin, jolloin muun muassa elementin nimen muuttaminen voidaan tehdä suoraan graafiseen elementtiin kirjoittamalla. Graafisia elementtejä voidaan yleisesti käsitellä kuten Visiossa normaalistikin, eli esimerkiksi ryhmittely, syvyysjärjestys, ja väritykset ovat vapaasti käytävissä ja ne tallentuvat automaattisesti tietokantaan.

5.3.4. Suhteen luominen

Suhde-elementtejä, kuten aktiviteettikaavion siirtymiä tai luokkakaavion assosiaatioita, luodaan muiden elementtien tavoin kohdan 5.3.2 mukaisesti. Suhde-elementtien päitä voidaan yhdistää minkä tahansa kaavion elementtien välille, jolloin nämä elementit liittyvät toisiinsa kyseisellä suhteella. Suhde-elementit voidaan liittää muihin elementteihin joko dynaamisesti tai staattisesti (Kuva 5.7).



Kuva 5.7. Suhde-elementin liittäminen dynaamisesti (vasemmalla) ja staattisesti (oikealla).

Dynaaminen liittäminen muodostetaan viemällä viivan pää keskelle kohde-elementtiä, jolloin se korostetaan punaisella viivalla. Liittäminen ollessa dynaaminen viivaelementti on kiinnittynyt kohde-elementtiin vapaasti niin, että liittäminen vaihtelee tilanteen mukaan elementtejä liikutellessa. Staattisessa liittämässä viivan pää yhdistetään sen sijaan vain ja ainoastaan yhteen tiettyyn kohde-elementin siniseen liittämisspottiin.

Suhteilla voi olla useita päitä, josta esimerkkinä luokkakaavion assosiaatio vasemmassa kuvassa 5.7. Eksplisiittisten suhte-elementtien lisäksi suhteita luodaan myös implisiittisesti monissa tilanteissa, kuten luokkaelementtiä pudottaessa pakettielementtiin, jolloin niiden väliin luodaan isä-lapsisuhde. Näitä suhteita ei esitetä näkymissä erillisinä graafisina elementteinä.

5.3.5. Elementin kopioiminen

Elementit kopioidaan näkymässä Vision normaaleita kopiointimekanismeja, kuten kopioi ja liitä, käyttäen. Trinityn näkymä- ja mallielementtijaottelu (5.2) kuitenkin johtaa tilanteeseen, jossa elementtejä voidaan kopioida usealla eri tavalla. Nämä tavat on esitelty seuraavassa taulukossa 5.1.

Taulukko 5.1. Elementtien kopiointitavat.

Täyskopiointi	Sekä malli- että näkymäelementti kopioidaan kohdenäkymään ja uusi näkymäelementti asetetaan viittaamaan uuteen mallielementtiin.
Viittauskopiointi	Vain näkymäelementti kopioidaan ja asetetaan viittaamaan alkuperäiseen mallielementtiin, eli luodaan uusi viittaava näkymäelementti.
Normaali kopiointi	Mikäli kopioitu näkymäelementti on ensisijainen, suoritetaan täyskopiointi, muussa tapauksessa viittauskopiointi.
Siirtäminen	Elementti siirretään kokonaan kohdenäkymään. Käytännössä suoritetaan täyskopiointi ja alkuperäinen kopioitu elementti poistetaan.

Yhden kaavion sisällä elementtiä kopioitaessa tehdään aina automaattisesti täyskopiointi. Tämä johtuu siitä, että on harvinaista luoda viittauksia saman näkymän sisällä ja kopiointi on usein nopein tapa luoda uusia samantyyppisiä elementtejä Visiossa. Viittaavia näkymäelementtejä voidaan kuitenkin luoda luvussa 5.3.2 esitetyllä tavalla pudottamalla mallielementtejä näkymään ulkoisista komponenteista, kuten Informaatiopaneelista.

Kopioitaessa kaavioiden välillä käyttäjä saa valita käytetyn kopiointimenetelmän itse. Valinta tarjotaan esittämällä dialogi sen jälkeen, kun näkymäelementti on kopioitu kohdenäkymään. Kopiointi voidaan tässä vaiheessa myös perua sulkemalla dialogi.

5.3.6. Mallin katselmointi

Trinity tarjoaa katselmointia ja muita kommentointitarpeita varten erilliset katselmointimallit ja -sessiot sekä niihin liittyvät hallintatyökalut. Katselmointimalleja ja -sessioita luodaan ja hallitaan hallintakäyttöliittymässä, mutta itse katselmointi suoritetaan mallinnustyökalussa.

Mallinnustyökalun puolella katselmointitilaa hallitaan Informaatiopaneelin kommentti-ikkunasta. Ikkunasta voidaan valita näkyviin ja aktiiviseksi niitä katselmointisessioita, jotka liittyvät auki olevan malliin ja jotka ovat ajanhetkellä avoimia. Yhteen sessioon liittyy joukko kommenttielementtejä, jotka esitetään avoimena olevassa näkymässä kommenttilappuina kun sessio valitaan näytettäväksi. Sessioita voi olla useita kerrallaan näkyvissä, mutta yhden niistä on aina lisäksi oltava aktiivisena, eli sessiona johon uudet lisättävät kommentit tallennetaan.

Kun ensimmäinen sessio valitaan aktiiviseksi, avataan kaavain, josta katselmointikommenttielementtejä voidaan pudottaa kaavioon. Kommentteja käsitellään ja liitetään elementteihin kaaviossa kuten muitakin elementtejä. Katselmointiominaisuus tukee vain Trinityn omia kommenttielementtejä, eikä Vision valmiita katselmointiominaisuuksia tai elementtejä ole integroitu Trinityyn. Katselmointisessioiden sulkemisen jälkeen myös kommenttikaavain suljetaan, sillä katselmointikommentteja ei ole tarkoitus käyttää kuin katselmointien yhteydessä.

5.3.7. Näköasetukset

Mallinnustyökalussa voidaan käyttää värityksiä havainnollistamaan näköelementtien tilaa tai ominaisuuksia. Väritystilat asetetaan Informaatiopaneelin ”näköasetukset”-ikkunasta ja valittavissa olevia tiloja on neljä: katselmointi, vertailukohta, oletus ja ”näytä poistetut elementit”. Tiloista kolme ensimmäistä ovat vaihtoehtoisia toisensa poissulkevia ja viimeinen aina valittavissa oleva asetus.

Katselmointiväritystilassa elementit väritetään riippuen niiden luomis- tai muokkaamisajankohdasta suhteessa niihin liittyvän katselmointisession aloitusajankohtaan. Session alun jälkeen muokattuja elementtejä merkitään sinisellä, uusia elementtejä vihreällä ja muuttumattomia ruskealla värillä, jolloin katselmoinnin seurauksena elementteihin tehtyjä päivityksiä on helppo seurata.

Vertailukohtatila on muuten samanlainen kuin katselmointiväritystila, mutta session sijasta elementtien aikaleimoja verrataan niin sanottuun *baseline*-malliin, jonka pohjalta malli on luotu. Oletustilassa elementit värjätään niin kuin käyttäjä ne on itse asettanut tai jos elementti on muuten käyttäjän huomiota kaipaava, esimerkiksi vioittunut. ”Näytä poistetut elementit” näyttää nimensä mukaisesti näkömästä poistetut elementit punaisella värillä.

5.3.8. Kaavion sulkeminen ja työn lopettaminen

Koska kaikki käyttäjän tekemät toiminnot tallennetaan automaattisesti tietokantaan, ei käyttäjän tarvitse tallentaa työtään erikseen ennen lopettamista. Näin ollen Visio tai yksittäinen kaavion sisältämä sivu Visiossa voidaan yksinkertaisesti sulkea työskentelyn päätteeksi. Suljettaessa vain yksittäistä näkömää suljetaan myös tähän kieleen liittyvä kaavain, mikäli kaavio oli viimeinen tätä näkömätyyppiä oleva avoin kaavio.

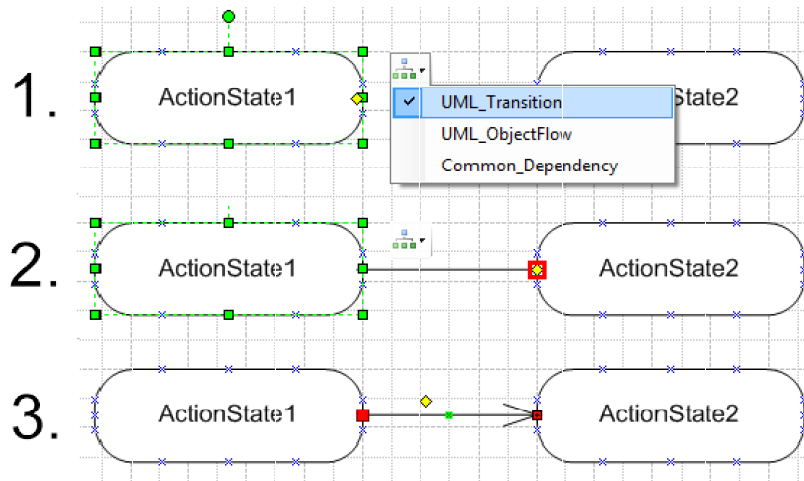
Työkalu ei oletuksena suljettaessa kysy käyttäjältä laajennuksella luodun kaavion tallentamisesta kuvana sen tarpeettomuudesta johtuen. Kaavio voidaan kuitenkin tallentaa normaalisti Visio-tiedostona ennen sulkemista Vision tallentamistoiminnoilla. Tässä tapauksessa kaavio kuitenkin on vain irrallinen kuva, joka ei enää ole yhteydessä Trinity-ympäristöön.

5.4. Erikoistoimintoja

Laajennus lisää Visioon mallinnustuen lisäksi myös muita työkalun käyttöä huomattavasti helpottavia aputoimintoja. Näitä ovat muun muassa nopeampi tapa luoda viiva-elementtejä ja Visioon valmiilla kaaviopohjilla luotujen UML-kaavioiden tuonti Triniin. Nämä toiminnot esitellään seuraavassa.

5.4.1. Viivan luominen suoraan elementistä

Visiossa kaikki elementit, mukaan lukien viivat, luodaan normaalisti pudottamalla perusmuotoja kaavaimesta piirtoalueelle. Linkkejä luotaessa elementtien välille tämä on kuitenkin hidasta, sillä viivaelementit on tämän jälkeen erikseen vielä liitettävä haluttujen elementtien välille. Suoraan viivan vetäminen elementistä toiseen on huomattavasti nopeampi ja intuitiivisempi tapa linkkien luomiseen, mutta Visio 2007 ei tue kyseistä ominaisuutta. Tästä syystä laajennuksessa käytettyihin elementteihin on lisätty valinnainen aputoiminto, jolla käyttäjä voi valita halutun viivatyyppin ja vetää sen suoraan elementistä toiseen (Kuva 5.8). Vaikka toiminnallisuus on Visioon normaalista käytöstä poikkeavaa, ovat siitä saadut käyttökokemukset olleet positiivisia.



Kuva 5.8. Viivaelementin vetäminen elementistä toiseen.

Haluttu viivatyyppi valitaan valitun elementin viereen ilmestyvää kuvaketta klikkaamalla ja valitsemalla esille tulevasta listasta jokin vaihtoehtoista (Kuva 5.8, vaihe 1). Tämä lista näyttää kaikki kyseiseen kaaviotyyppiin kuuluvat viivatyyppit. Jokaiselle elementille on määritetty kuitenkin yleisimmin käytetty viivatyyppi oletukseksi, jolloin tätä vaihtoa ei useimmiten ole tarve suorittaa. Viivatyyppin valinnan jälkeen viiva piirretään vetämällä elementissä oleva keltainen piste joko kiinni toiseen elementtiin (Kuva 5.8, vaihe 2) tai vapaasti mihin tahansa kaaviota, jonka jälkeen vaiheessa 1 valittu suhde-elementti luodaan tähän väliin (Kuva 5.8, vaihe 3).

5.4.2. Vision UML-mallien tuominen Trinity-ympäristöön

Vision valmiita UML-kaavioita käytetään usein sellaisenaan mallintamistarkoituksessa ja yhtenä toiveena ympäristölle oli mahdollisuus hyödyntää niitä suoraan myös *Trinityssä*. Tätä varten Visio-laajennus mahdollistaa Vision valmiiden UML-luokkakaavioiden tuomisen helposti osaksi Trinityä muuntamalla ne Trinityyn luokkakaavioiksi. Muunnos on yksisuuntainen, joten Trinity UML-luokkakaavioita ei voida palauttaa takaisin Visioon valmiin UML-kaaviomallin mukaiseksi kaavioksi.

Muunnos suoritetaan valitsemalla haluttu alue tai koko kaavion valmiista Vision UML-kaaviosta ja kopioimalla se leikepöydälle. Tämän jälkeen kopioidut elementit liitetään Trinity UML-luokkakaavionäkymään Trinityyn työkalupalkin valikosta toiminnolla ”Import Visio UML Class Diagram Elements”. Laajennus tulkitsee kopioidut elementit ja korvaa ne Trinityyn vastaavilla elementeillä sekä tallentaa ne tietokantaan uusina elementteinä.

6. JÄRJESTELMÄN TOTEUTUS

Tässä luvussa käydään läpi Trinity-työkaluympäristön toteutuksen suunnitteluperiaatteet sekä ympäristön ja työssä toteutetun Visio-laajennuksen arkkitehtuurit. Trinity on kokonaisuutena toimiva järjestelmä, jossa koko ympäristön arkkitehtuuriset ratkaisut vaikuttavat myös siihen liitettyjen yksittäisten työkalujen toteutukseen.

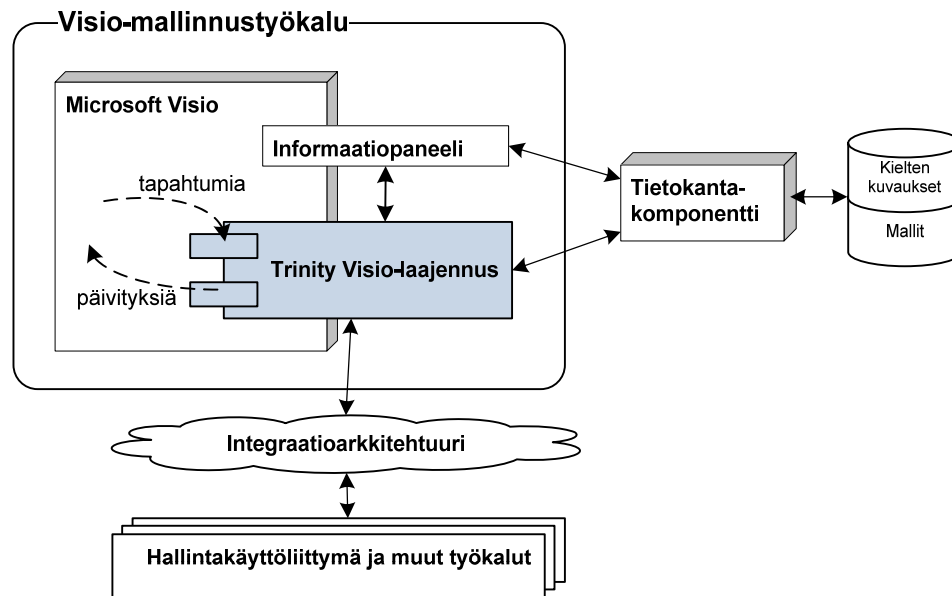
6.1. Suunnitteluperiaatteet

Ympäristönlajuisesti keskeisenä suunnitteluperiaatteena on toteutuksen geneerisyys. Geneerisyydellä pyritään mahdollisimman joustavaan ja laajennettavaan ympäristöön, johon uusien komponenttien ja toiminnallisuuksien lisääminen on helppoa ja nopeaa. Laajennettavuuden osalta pyritään etenkin uusien työkalujen ja mallinnuskielien lisäämisen sekä olemassa olevien muokkaamisen helpouteen. Ympäristön tulee tukea heterogeenistä ja mielivaltaista joukkoa työkaluja, joten kieliriippumaton kommunikaatio ja tietotyypin välittäminen on merkittävä suunnitteluperiaate etenkin ympäristön kontrolli-integraation tasolla.

Geneerisyyden ja laajennettavuuden lisäksi tärkeä periaate on toteutuksen ja komponenttien keskittäminen yhteen paikkaan sekä niiden modulaarisuus. Samoja komponentteja pyritään käyttämään mahdollisimman monessa paikassa sekä toteutuksen tehokkuuden että käyttäjälle tarjottavan käyttökokemuksen konsistenttiuden parantamiseksi. Geneerisyydestä huolimatta toteutukset pyritään pitämään mahdollisimman yksinkertaisina ja rivimääriltään pieninä, jotta ylläpidettävyys säilytetään riittävän korkealla tasolla.

6.2. Trinity-ympäristön arkkitehtuuri

Trinityn korkean tason arkkitehtuuri on esitetty kuvassa 6.1. Kaikki järjestelmän sovellukset keskustelevat toistensa kanssa integraatioarkkitehtuurin välityksellä ja liittyvät tietokantaan erillisellä tietokantakomponentilla. Tietokanta, tietokantakomponentti ja integraatioarkkitehtuuri on kuvattu tarkemmin kohdissa 6.2.1, 6.2.2 ja 6.2.3.



Kuva 6.1. Korkean tason kuvaus Trinity-ympäristön arkkitehtuurista.

Visio-mallinnustyökalu kuvassa 6.1 esittää ympäristöön liitettyjen mallinnustyökalujen yleistä rakennetta ja liityntää muuhun järjestelmän komponentteihin. Työkaluihin, kuten Microsoft Visioon, toteutetut laajennuskomponentit reagoivat laajennettavan työkalun käyttöliittymän tapahtumiin ja päivittävät muutokset tietokantaan sekä vastaavasti päivittävät työkalun näkymää, kun tietokannan tila muuttuu. Myös hallintakäyttöliittymä nähdään ympäristössä yhtenä työkaluna huolimatta sen erityisestä ja keskeisestä roolista.

Työkalulaajennukset voivat halutessaan liittää Informaatiopaneelin työkalun käyttöliittymään. Informaatiopaneelin luominen ja työkalunäkymässä tapahtuvien valintojen päivittäminen on laajennuskomponenttien vastuulla ja Informaatiopaneeli itse on työkalusta riippumaton. Informaatiopaneeli tarjoaa ja vaatii tietyt rajapinnat, joiden kautta tiedon päivittäminen suoritetaan työkalun ja sen välillä. Nämä rajapinnat on kuvattu kohdassa 6.3.2.

6.2.1. Tietokanta

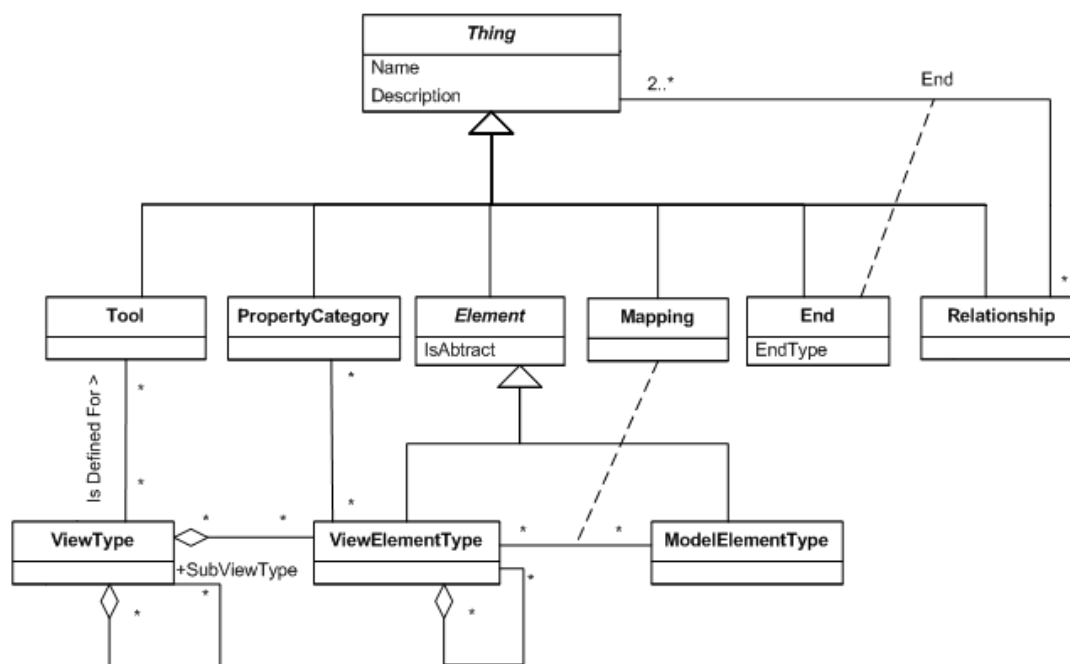
Trinityn tietokantapalvelimena käytetään avoimen lähdekoodin PostgreSQL:ää (PostgreSQL 2011). Se on oliorelaatietietokanta, jota työkaluympäristön sovellukset käyttävät luvussa 6.2.2 kuvatun erillisen oliorajapinnat tarjoavan tietokantakomponentin kautta.

Tietokanta sisältää olennaisimpana kaikki mallit, kuvaukset tuetuista kielistä ja tietoa ympäristöstä sekä siihen integroiduista työkaluista. Tietokannan rakenne perustuu metatasohierarkiaan, joka on esitetty taulukossa 6.1.

Taulukko 6.1. Metatasot Trinityssä.

Metametataso	Kieli metamallien (mallinnuskielien) määrittelyyn.
Metataso	Mallinnuskielet, kuten UML-luokkakaaviokieli.
Mallitaso	Metatason mallinnuskielillä luodut mallit, kuten UML-luokkakaaviomalli, sekä niiden näkymäinformaatio.

Kaikki käyttäjien luomat mallit tallennetaan alimmalle mallitasolle, joka sisältää mallitiedon lisäksi kaiken näkymäinformaation. Mallien mahdollinen sisältö ja rakenne määritellään metatasolla kuvatuilla mallinnuskielillä, jotka puolestaan määritellään ylimmän metametatason kielellä. Kuva 6.2 sisältää otteen metametamallin käsittekaaviosta, joka on esitelty kokonaisuudessaan liitteessä 1.



Kuva 6.2. Ote Trinityn metametamallin käsittekaaviosta.

Tietokannan metatasohierarkian ylimmällä tasolla olevalla metametamallilla tulee pystyä kuvaamaan kaikki ympäristön käsitteet mukaan lukien työkalut, joten se on määritelty melko abstraktiksi. Metametataso on lisäksi ainoa kiinteä osa Trinityn tietokantarakennetta. Sekä meta- että mallitason rakenteet generoidaan automaattisesti ylemmän tason sisällön perusteella, jolloin uusia kieliä ja työkaluja voidaan lisätä järjestelmään dynaamisesti ilman olemassa olevien rakenteiden muuttamista.

Keskitetty tietokanta sijaitsee erillisellä palvelinkoneella, jonka lisäksi jokaisella työasemalla on replikoitu paikallinen tietokanta, joka synkronoidaan keskitetyn tietokannan kanssa. Kaikki käyttäjän tekemät muutokset suoritetaan ensin keskitetylle palvelimelle konfliktien välttämiseksi, jonka jälkeen onnistuneet päivitykset päivitetään paikalliseen tietokantaan. Lukuoperaatiot suoritetaan ensisijaisesti suoraan paikalliseen

tietokantaan ja tiedot päivitetään keskitetystä tietokannasta vain, mikäli tietoa ei ole vielä replikoitu tai se on päivittynyt. Näin optimoidaan tietokantayhteydestä aiheutuneita viiveitä merkittävästi. Nykyinen tietokantatoteutus ei salli päivitysten tekoa ensin paikalliseen tietokantaan ja jälkikäteen synkronoimista keskitettyyn tietokantaan, mutta tulevaisuudessa tähän tultaneen pyrkimään.

6.2.2. Tietokantakomponentti

Tietokantakomponentti (*O/R mapper*) tarjoaa kaikille työkaluille helppokäyttöiset oliorajapinnat Trinityn relaatiopohjaisen tietokannan käsittelyyn. Se osaa tulkita luvussa 6.2.1 esiteltyjä tietokannan metarakenteita ja dynaamisesti luoda niiden perusteella oikeanlaisen oliomallin.

Tietokantakomponentin käyttö perustuu kiinteisiin rajapintoihin, jotka perustuvat Trinityn metametamalliin. Olennaisimpia rajapintoja ympäristön mallinnustyökalujen kannalta ovat esimerkiksi *IModel*-, *IView*-, *IModelElement*-, *IViewElement*- ja *IRelationship*-rajapinnat, joiden avulla pystytään käsittelemään varsinaisia malleja, näkymiä, elementtejä ja niiden välisiä suhteita. Vaikka mallinnuskieli tai käytetty tietokanta-ajuri vaihtuvat, myös mallitason rajapinnat säilyvät ennallaan eikä työkaluihin näin aiheudu muutoksia. Lisäksi tietokantakomponentin kerrosrakenne tukee käytetyn tietokanta-ajurin vaihtamista ilman muutoksia

Tietokantakomponentti sisältää lisäksi transaktioiden hallinnan, tietokantojen välisen synkronoinnin, tietokantatapahtumista tiedottamisen ja plugin-mekanismien työkalukohtaisen näkymäinformaation käsittelemiseen. Etenkin Vision tapauksessa taulukko-muotoisen näkymäinformaation, eli *ShapeSheetien*, käsittely on merkittävä osa tietokantakomponentin käyttöä. Komponentin rajapintojen käyttöä esitellään tarkemmin Visio-laajennuksen toteutusta käsittelevässä kohdissa 6.3, 6.4 ja 7.

6.2.3. Integraatioarkkitehtuuri

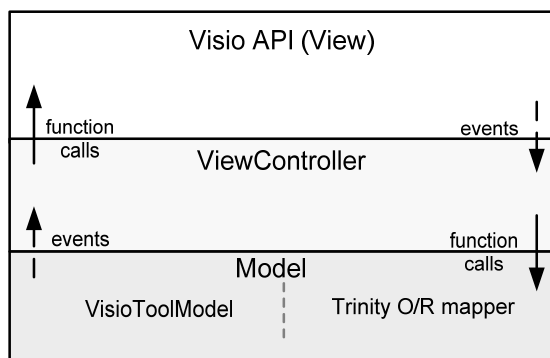
Ympäristön työkaluintegraation kontrolli on toteutettu agenttiarkkitehtuurilla. Agenttiarkkitehtuuri on autonomisiin agenteihin perustuva integraatioarkkitehtuuri, jossa niin kutsutut agentit liikkuvat ympäristön sijainnista toiseen suorittaen sille määrättyä tehtävää (Peltonen & Vartiala 2009). Arkkitehtuurin periaatteena on toiminnallisuuslogiikan sijoittaminen yhteen paikkaan, eli agenttiin, hajautetussa järjestelmässä.

Trinityssä yksi tavanomainen agentti on kaavion avaamisen suorittava agentti, joka voidaan käynnistää hallintakäyttöliittymästä avaamaan kaavion johonkin ympäristön työkaluista. Trinityn agenttiarkkitehtuuri on lisäksi toteutettu kieliriippumattomasti, jolloin eri kielillä toteutettujen työkalujen rajapintojen integroiminen ympäristöön on helppoa. Käytännössä kieliriippumattomuus tarkoittaa käytettyjen tietotyyppien kieliriippumattomuutta ja jokainen kieli vaatii oman toteutuksen arkkitehtuurista. Tämän työn tekohetkellä Trinityn agenttiarkkitehtuuri on toteutettu C#- ja Java-ohjelmointikielillä.

6.3. Visio-laajennuskomponentin arkkitehtuuri

Laajennus toteutettiin yhtenä *COM Add-in* DLL:nä Microsoft Visioon versioon 2007. Laajennuskomponenttitekniologiaksi valittiin *COM Add-in* erityisesti sen helppokäyttöisyyden ja paremman suorituskyvyn vuoksi suhteessa *Add-on*-teknologiaan, kuten luvussa 2.3.5. esiteltiin. Toteutuskielenä käytettiin Trinity-projektissa yleisesti käytössä olevaa C#:n versiota 3.5 (Microsoft 2011i).

Laajennuskomponentin arkkitehtuuri perustuu malli-näkymä-ohjain -arkkitehtuuriin, kuten kuvan 6.3 korkean tason kuvauksesta nähdään. Arkkitehtuuri on lisäksi jakautunut kerroksiin, joten sen voidaan nähdä noudattavan myös kerrosarkkitehtuurin periaatteita. Puhtaasta kerrosarkkitehtuurista poiketen sovelluksen ylempi kerros ei käytä pelkästään alemman kerroksen palveluita, vaan ohjaimet käyttävät ratkaisussa myös ylemmällä tasolla olevia näkymän palveluita. Kutsut kohdistuvat kuitenkin välittömästi ylä- tai alapuolella olevaan kerrokseen, joten tasojen ohituksia ratkaisussa ei ole. Kerrosarkkitehtuurina tarkastellen laajennuskomponentti voitaisiin jakaa myös neljään kerrokseen, jolloin tietokantakomponentti olisi alimmainen kerros *Model*-kerroksen alla. Tällöin funktiokutsut ohittaisivat *Model*-kerroksen kulkiessaan *ViewController*-kerroksesta tietokantakomponentin kerrokseen.



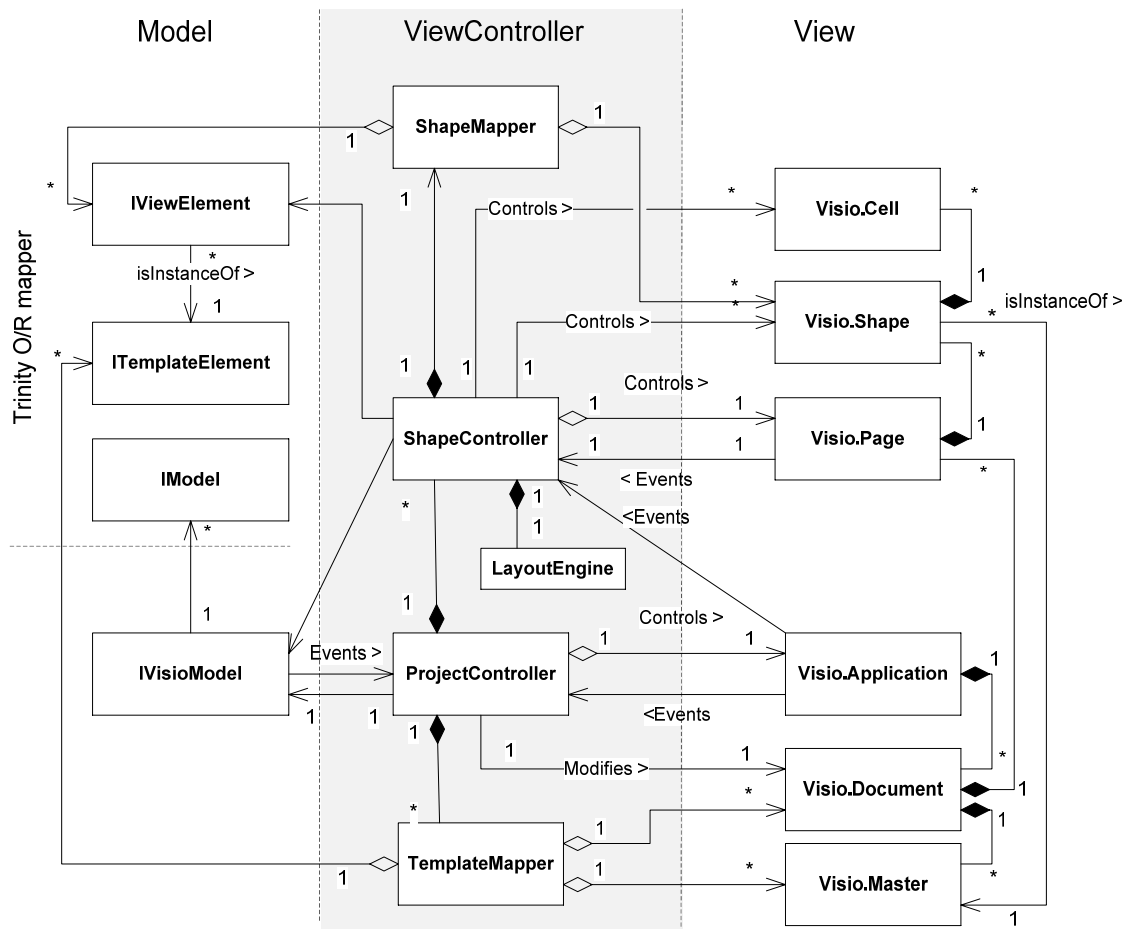
Kuva 6.3. Laajennuskomponentin kerrostunut malli-näkymä-ohjain -arkkitehtuuri. Mukailtu lähteestä (Peltonen et al. 2010).

Arkkitehtuurin *ViewController*-kerros, eli ohjain, toimii välikomponenttina reagoiden näkymä- ja malli-kerroksen tapahtumiin ja pitäen niiden tilat synkronoituina. Tapahtuman seurauksena *ViewController* päivittää vastakkaisen kerroksen tilaa kutsumalla sen funktioita. Tapahtumien ja funktiokutsujen suunnat on osoitettu nuolilla kuvassa 6.3. Tapahtumien kuuntelussa käytetään kaikkialla luvussa 0. esitellyjä .NET:in tapahtumia ja delegaatteja.

Näkymäkerros (*View*) koostuu yksinkertaisesti pääasiassa Visioon automaattiorajapinnan (2.3.4) tarjoamista rajapinnoista. Se esittää graafisesti *Model*-kerroksen tilan, eli mallin, ja mahdollistaa sen muokkaamisen Visioon käyttöliittymän elementtien kautta. Tämän lisäksi näkymäkerros sisältää kaikki käyttäjälle esitettävät dialogit, joilla jokaisella on vastaava kontrolleri *ViewController*-kerroksessa. Monimutkaisempien *ViewController*- ja *Model*-kerrosten rakenne ja toiminta kuvataan seuraavaksi yksityiskohtaisemmin kohdissa 6.3.1 ja 6.3.2.

6.3.1. ViewController-kerros

ViewController-kerros toimii sovelluksen malli-näkymä-ohjain-arkkitehtuurin ohjaimena ja on vastuussa mallin ja näkymän tilan synkronoinnista. Kerroksen olennaisimmat luokat ovat *ProjectController*, *ShapeController* ja *LayoutEngine* (Kuva 6.4).

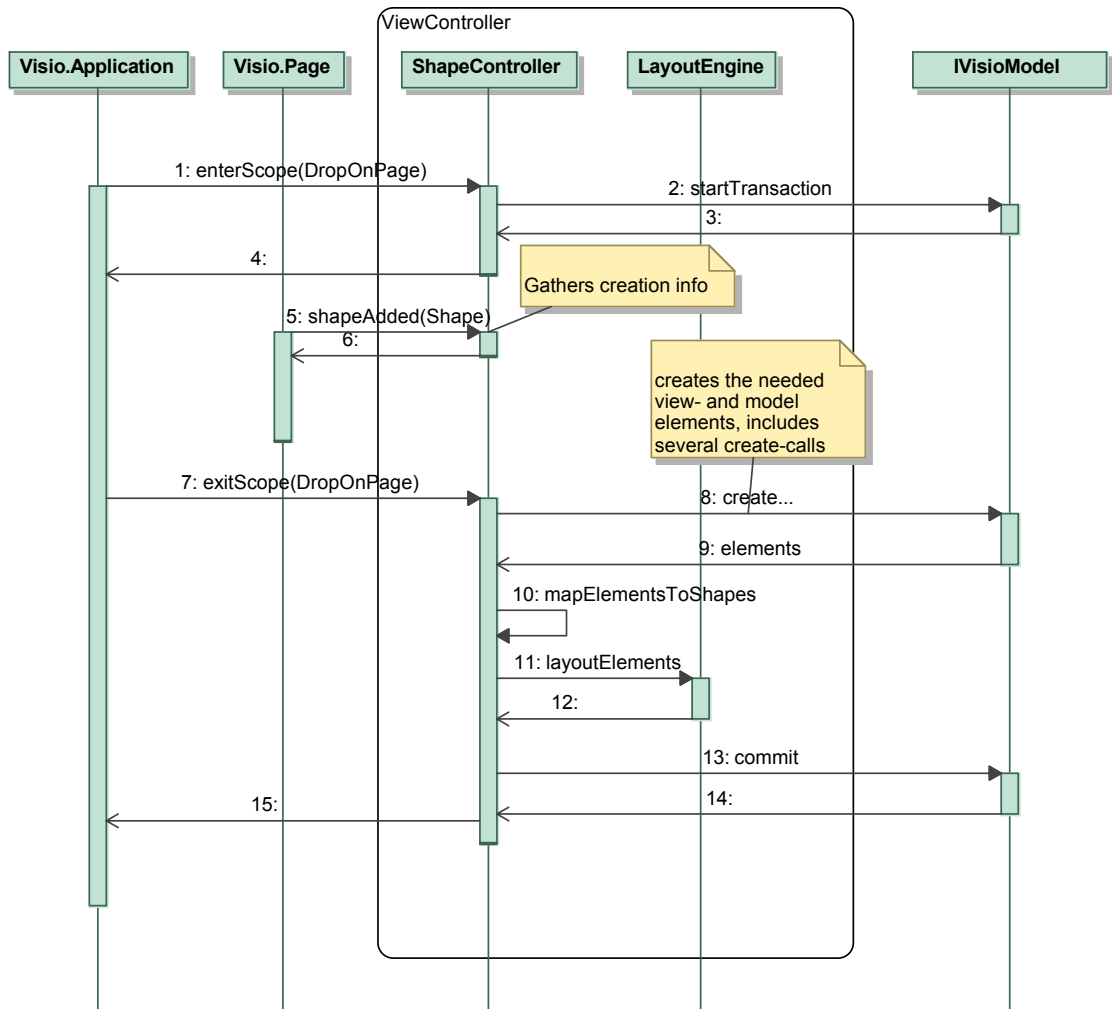


Kuva 6.4. ViewControllerin olennaiset suhteet malliin ja näkymään.

ProjectController-luokan vastuulla on Vision korkeimman tason elementtien eli koko sovelluksen ja sen dokumenttien hallinta. Tätä varten *ProjectController* kuuntelee Vision *Application*-luokan lähettämiä tapahtumia esimerkiksi sivun vaihtumisesta ja sulkemisesta. *ProjectController* on myös vastuussa kaikkien *Model*-kerroksesta tulevien tapahtumien kuuntelusta, niihin reagoimisesta ja välittämisestä tarvittaessa eteenpäin *ViewController*-kerroksessa. *ProjectController* hallitsee lisäksi *Stencil*-olioita ja niiden sisältämiä *Master*-elementtejä, jotka se linkittää tietokantakomponentin vastaaviin *ITemplateElement*-rajapinnan toteuttaviin olioihin *TemplateMapper*-komponentilla. Kaaviota avattaessa ja vaihdettaessa *ProjectController* pitää huolen siitä, että oikea *Stencil* on aina avoinna.

ProjectController luo jokaista sivua kohden yhden *ShapeController*-olion, joka on vastuussa kyseisen sivun sisältämän kaavion elementeistä. *ShapeController* linkittää *ShapeMapper*-luokan avulla jokaisen sivulla olevan *Shape*-elementin mallissa olevaan

näkymäelementtiin (*IViewElement*) ja pitää ne synkronoituina. *ShapeController* toimii näin laajennuskomponentin keskeisimpänä moottorina, sillä muodot ovat sovelluksen merkittävin rajapinta käyttäjän ja mallin välillä. Tapahtumat *Shape*-olioiden muutoksista *ShapeController* saa sen hallitseman *Page*-olion tapahtumien kautta. Tämän lisäksi *ShapeController* kuuntelee joitakin *Application*-luokan tapahtumia, jotka liittyvät yksittäisiin *Shape*-elementteihin ja joiden käsittely kuuluu täten loogisesti sen vastuulle. Kuva 6.5 näyttää esimerkin *ViewController*-kerroksen toiminnasta tilanteessa, kun käyttäjä pudottaa uuden elementin kaavioon.

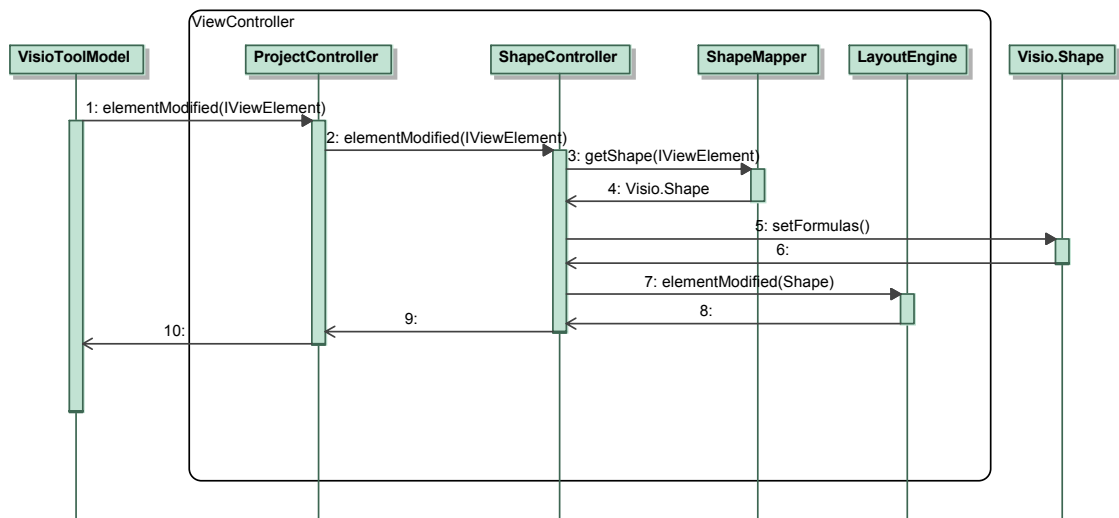


Kuva 6.5. *ViewController*in toiminta elementin luomisessa. Mukailtu lähteestä (Peltonen et al. 2010).

Visio sitoo kaikki sen sisäiset komennot kokonaisuuksiin niin sanotuilla *Scopeilla*, joiden sisällä myös kaikki komentoon liittyvät tapahtumat lähetetään. Nämä *Scopet* sidotaan usein myös suoraan tietokantatransaktioihin niin, että yksi *Scope* aiheuttaa yhden transaktion. Kuvan 6.5 esimerkissä uuden muodon luominen aiheuttaa uuden *DropOnPage*-scopen, josta *ProjectController* saa Visiolta tapahtuman *EnterScope*. *ProjectController* ohjaa tapahtuman käsittelyn aktiivisen sivun *ShapeController*-instanssille, joka aloittaa *Scopea* varten vastaavan tietokantatransaktion mallikerroksen *IVisioModel*-

rajapinnan metodilla ja jää odottamaan *ShapeAdded*-tapahtumaa. Selkeyden vuoksi *ProjectController* on jätetty pois kuvasta. *ShapeAdded*-tapahtuman syntyessä *ShapeController* kerää luodun *Shape*-elementin tiedot talteen ja esimerkiksi tulkitsee, että onko kyseessä uusi vai kopioitu elementti. *Scopen* päätyttyä *Visio* lähettää *ExitScope*-tapahtuman, jonka seurauksena *ShapeController* kutsuu *VisioToolModel*-luokan elementinluomismetodia, joka luo pudotettua *Shape*-oliota vastaavat elementit tietokantaan. Tuloksena *ShapeController* saa tietokantakomponentin *IViewElement*-olion, joka linkitetään *ShapeMapper*-olion avulla vastaavaan *Shape*-olioon. Lopuksi *LayoutEngine* tarvittaessa asemoi ja värjää luodun *Shapen* ja transaktio päätetään onnistuneesti kutsuamalla *IVisioModel*-rajapinnan *commit*-metodia. Uudet elementit luodaan ja tietokanta-transaktio päätetään vasta *Scopen* päätteeksi siksi, että monimutkaisemmissa tilanteissa yhden *Scopen* sisällä voi olla useita tapahtumia, jotka halutaan mukaan samaan transaktioon. (Peltonen et al. 2010)

Kaikki *Vision* käyttöliittymässä tapahtuvat muutokset käsitellään karkeasti ottaen samoilla periaatteilla kuin edellisessä esimerkissä. Muotoa liikuteltaessa kutsutaan *IVisioModel*-rajapinnan *elementModified*-metodia, poistettaessa *elementDeleted*-metodia ja niin edelleen. Tietokannan muutokset päivitetään *Visio*-näkömään vastaavasti *IVisioModel*-rajapinnan tapahtumien perusteella, kuten seuraavassa esimerkissä (Kuva 6.6) nähdään.



Kuva 6.6. *Vision* päivittäminen mallin muuttuessa.

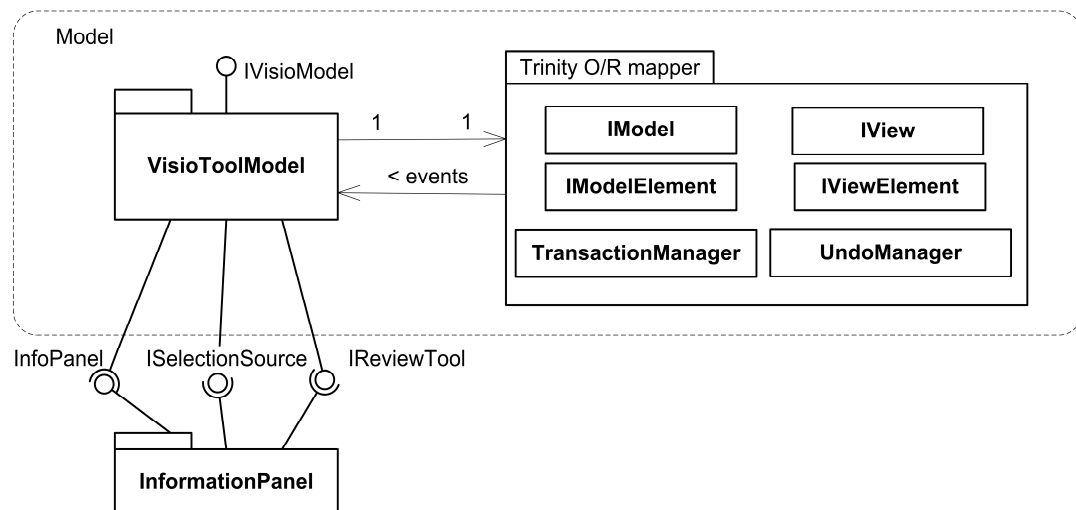
Mallin muutosten päivittäminen *Visio*-näkömään on *ViewController*-kerrokselle yksinkertaisempi operaatio siinä mielessä, että siinä ei tarvita tietokantatransaktioita. Kuvan 6.6 esimerkissä *ShapeController* yksinkertaisesti käsittelee *IVisioModel*-rajapinnan *elementModified*-tapahtuman ja päivittää muuttunutta *IViewElement*-oliota vastaavan *Shape*-olion uudella tiedolla. Muuttunut ominaisuus on osana tapahtuman parametrejä ja mikäli *Shape* on kiinnostunut siitä, asetetaan uusi arvo sen *ShapeSheetiin*. Muodon näyttämien ominaisuuksien määrittely on esitelty tarkemmin kohdassa 7.1. Muutoksen lopuksi *LayoutEngine* asemoi muodon tarvittaessa uudelleen.

ViewController-kerroksen luokkien toteutukset ovat geneerisiä siten, että kaikki *Shapet* käsitellään samoilla *ShapeController*- ja *ProjectController*-olioilla mallinnuskielestä riippumatta. Tähän päästään määrittelemällä *Shapejen Shapheetissä* kaikki riittävä tieto elementistä kohdassa 7.1 esitellyillä menetelmillä. Näin saavutetaan luvussa 3.3 esitetty vaatimus geneerisyydestä ja minimaalisista koodimuutoksista kielten muuttuessa. Varjopuolena tästä seuraa luonnollisesti jonkinasteinen suorituskyvyn heikentyminen.

Joissain tilanteissa *ViewController* käyttää tietokantakomponentin rajapintoja myös suoraan ohi *Model*-kerroksen, mikäli se on suorituskyvyn kannalta järkevää. Esimerkiksi *IViewElement*-näkömäälelementtien ominaisuuksia luetaan suoraan niiden rajapinnan kautta, sillä *ShapeController* sisältää näkömäälelementit jo valmiiksi ja lukuoperaatioita suoritetaan tiheästi.

6.3.2. Model-kerros

Model-kerros koostuu olennaisesti kahdesta moduulista, *VisioToolModel*-luokasta ja luvussa 6.2.2 esitellystä tietokantakomponentista (*Trinity O/R mapper*). Kerroksen rakenne nähdään kuvassa 6.7.



Kuva 6.7. Komponentin *Model*-kerroksen keskeiset komponentit ja rajapinnat.

VisioToolModel toteuttaa kolme merkittävää rajapintaa: *IVisioModel*-, *IReviewTool*- ja *ISelectionSource*-rajapinnat. Rajapinnoista ensimmäinen on rajapinta *ViewController*-kerrokseen (6.3.1) ja kaksi jälkimmäistä ovat Informaatiopaneelin (4.2.2) vaatimia rajapintoja. *ISelectionSource*-rajapinnan kautta Informaatiopaneeli voi rekisteröityä kuuntelemaan Vision elementtien valintoja, ja *IReviewTool*-rajapinta tarjoaa menetöt katselmointisessioiden näyttämiseen työkalussa. Visio puolestaan saa tapahtumat Informaatiopaneelistä sen *InfoPanel*-rajapinnan kautta.

VisioToolModelin roolina *Model*-kerroksessa on tarjota apufunktioita tietokannan mallien ja näkymien monimutkaisempaan käsittelyyn, kuten esimerkiksi näkymien avaamiseen ja elementtien luomiseen. *VisioToolModel* ei itsessään sisällä luotavia mal-

leja tai niiden tilaa, vaan ne ovat tietokantakomponentin vastuulla. *ViewControllerin* kerroksen luokat muokkaavat mallia pääasiassa *IVisioModel*-rajapinnan kautta ja *VisioToolModel* suorittaa tarvittavat kutsut tietokantakomponentin rajapintaan. *VisioToolModel* myös käsittelee kaikki tietokantakomponentin kiinnostavat tapahtumat ja lähettää *IVisioModel*-rajapinnan kautta vastaavat *ViewControllerille* paremmin räätälöidyt tapahtumat.

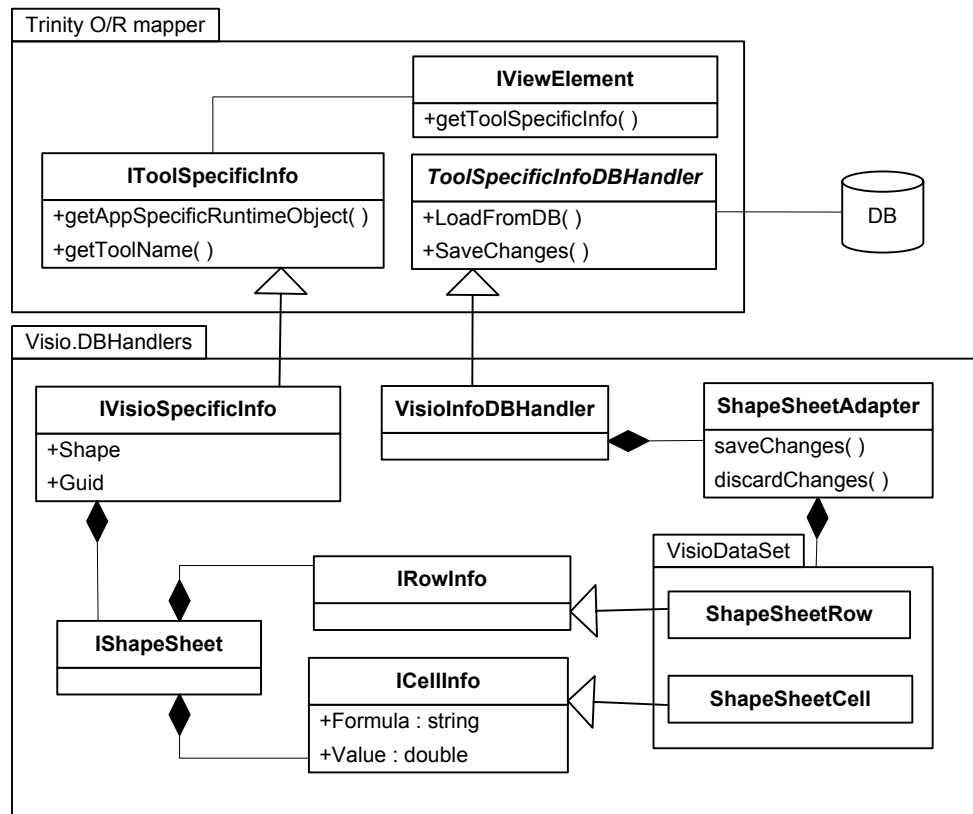
Tietokantakomponenttia suoritetaan eri säikeessä kuin Visiota, jolloin *VisioToolModel* joutuu ottamaan huomioon muutamia olennaisia tekijöitä sen tapahtumien käsittelyssä. Normaaleissa tilanteissa tietokantakomponentin ilmoittaessa tapahtumasta suoritus voitaisiin siirtää Visioon säikeeseen suoraan lähettämällä *MarkerEvent*-tapahtuma ennen päivitysten tekoa Visioon *Application*-luokan *QueueMarkerEvent*-metodilla. Mikäli käyttäjällä on kuitenkin esimerkiksi modaalinen dialogi auki Visiossa, aiheuttaa rajapintakutsu rinnakkaisuuspoikkeuksen. Ratkaisuna käytetään *COM*-rajapinnan *CoRegisterMessageFilter*-metodia, jonka avulla rajapintakutsu voidaan suorittaa asynkronisesti käyttäjän suljettua modaalisen dialogin. Näin ollen ulkoisia muutoksia ei voida modaalisen dialogin ollessa auki päivittää Visio-näkymään. Rinnakkaisuuden lisäksi *VisioToolModel* joutuu tulkitsemaan tietokantatapahtumien tyyppiä, sillä tietokantakomponentti lähettää tapahtumat myös paikallisista muutoksista, joihin ei useimmissa tapauksissa haluta reagoida.

6.4. Laajennuskomponentin erityispiirteitä

Laajennuskomponentin arkkitehtuurisista ratkaisuista voidaan erottaa kaksi merkittävää yksityiskohtaa: *ShapeSheet*-datan käsittely sekä toimintojen kumoaminen ja uudelleen tekeminen (*undo* ja *redo*). Näiden ratkaisuperiaatteet ja haasteet esitellään seuraavaksi kohdissa 6.4.1 ja 6.4.2.

6.4.1. ShapeSheet-datan käsittely

ShapeSheet-datan tallentamiseksi Trinityn tietokantaan tulee työkalun tarjota näkymädatan käsittelyyn tarvittava toiminnallisuus tietokantakomponentille. Tämä tarkoittaa työkalukohtaisen näkymädatan *IToolSpecificInfo*-rajapinnan ja abstraktin *ToolSpecificInfoDBHandler*-luokan konkreettista toteuttamista. *IToolSpecificInfo* on rajapinta, jonka työkalut tarjoavat tietokantakomponentin *plugin*-mekanismille, joka puolestaan linkittää sen generiseen *IViewElement*-elementtiin. *ToolSpecificInfoDBHandler* puolestaan vastaa tämän työkalukohtaisen tiedon automaattisesta lataamisesta ja tallentamisesta tietokantaan. Visio-laajennuskomponentissa nämä toteutetaan *Model*-kerroksen *Visio.DBHandlers*-moduulissa (Kuva 6.8).



Kuva 6.8. ShapeSheet-datan käsittelyn olennaisimmat rajapinnat ja luokat Visiossa ja tietokantakomponentissa (O/R mapper).

Visio-laajennuskomponentin ohjelmakoodissa näkymädata saadaan mekanismien ansiosta suoraan *IViewElement*-olion *getToolSpecificInfo*-metodilla, joka palauttaa *IVisioSpecificInfo*-rajapinnan toteuttavan olion. Visio-laajennuskomponentissa *ShapeSheet*-dataa käytetään *IShapeSheet*-rajapinnan kautta, joka kätkee taakseen itse datan tallentavan *VisioDataSet*-tietorakenteen. Tietorakenne perustuu *DataSetiin*, joka on Microsoft ADO.NET:in (Microsoft 2011j) tarjoama taulukkomuotoiselle datalle tarkoitettu säiliö. *VisioDataSetin* sisällön hallinnasta vastaa *ShapeSheetAdapter*, jota *VisioInfoDBHandler* kutsuu tietokantakomponentin kutsuessa sen tallennus- tai latausmetodeja.

ShapeSheet-datan käsittelyyn liittyy joitakin haasteita ja ongelmakohtia, jotka johtuvat sen tietokantaan tallentamisesta. Visiossa *Shapella* on yhden sivun kontekstissa uniikki Visio sisäinen ID, jolla se tunnistetaan ja jolla siihen viitataan esimerkiksi *ShapeSheetin* kaavoissa. ID:tä ei voida muokata luomisen jälkeen ja se vaihtuu, kun kaavio ladataan uudestaan tietokannasta ja *Shapet* luodaan alusta alkaen uudelleen. Tästä seurauksena *ShapeSheetin* kaavoissa olevat ID-viittaukset menevät sekaisin. Ratkaisuna ongelmaan jokaiselle *Shapelle* generoidaan *GUID* (Globally Unique Identifier), joka tallennetaan *ShapeSheetin* *User.Guid*-soluun. Kaikki viittaukset toisiin soluihin muutetaan ID-viittauksista *GUID*-viittauksiksi ennen tietokantaan tallentamista ja *GUID*-viittauksesta takaisin ID-viittaukseksi kaavion avaamisen jälkeen, kun *Shapejen* ID:t on tiedossa. Sama *GUID* toimii lisäksi tietokannassa yksikäsitteisenä tunnisteena sitä vastaavaan *IViewElement*-elementtiin linkattaessa. (Peltonen et al. 2010)

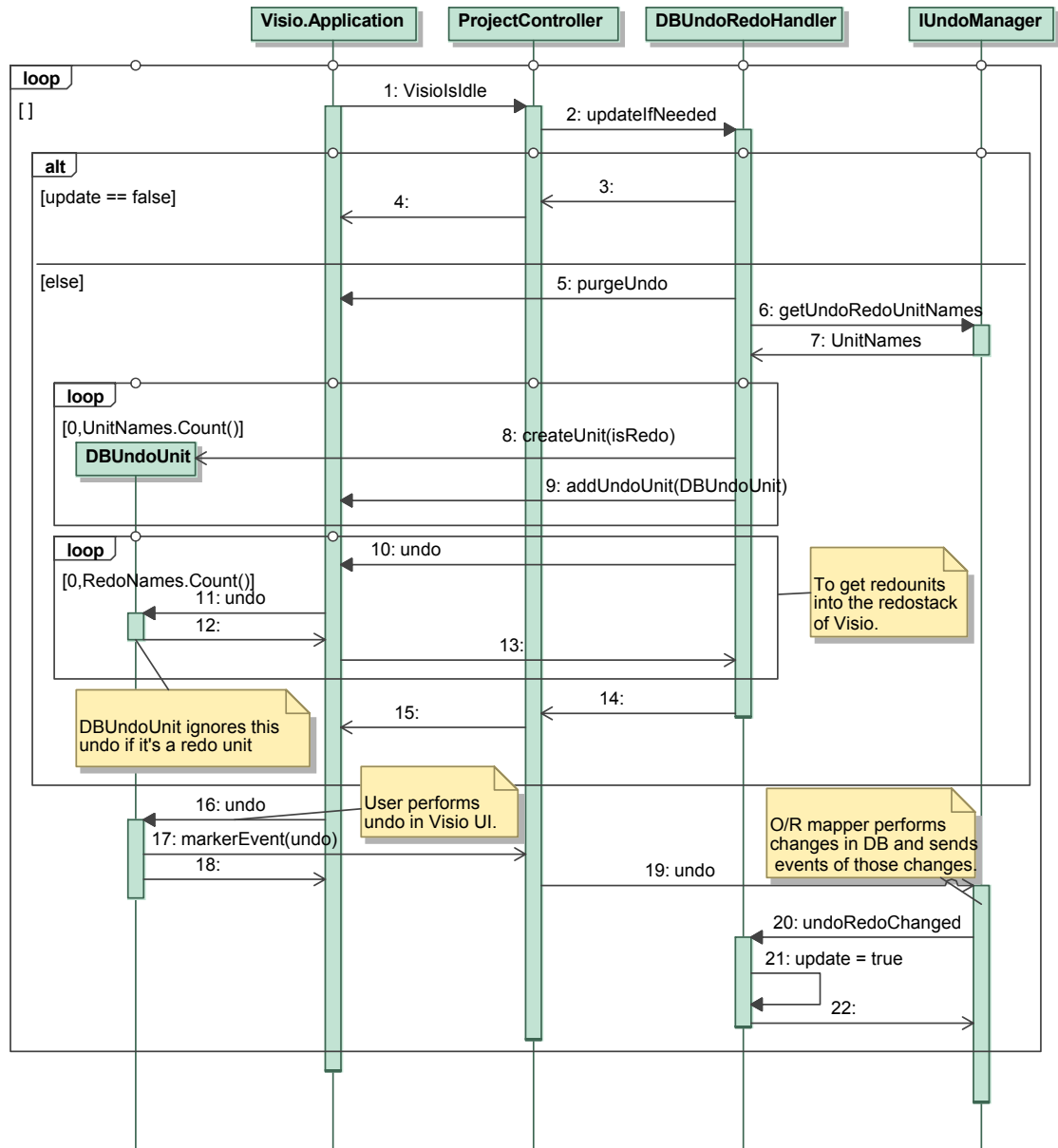
Toinen haaste liittyy *ShapeSheet*-solujen arvojen asettamisen heikkoon suorituskykyyn. Yksittäisen solun arvon asettaminen *Cell*-luokan *setFormula*-metodilla on erittäin raskasta, jolloin sen käyttöä on vältetty toteutuksessa mahdollisimman paljon. Useita solujen arvoja voidaan asettaa yhdellä kertaa käyttäen *setFormulas*-metodia, jota käyttämällä suorituskyky paranee monikymmenkertaisesti. Kun *setFormula*-metodia joudutaan käyttämään, on myös tehokkaampaa lukea solun arvo ennen asettamista ja varmistaa, että arvo on muuttunut, ettei sitä aseteta turhaan.

6.4.2. Toimintojen kumoaminen ja uudelleen tekeminen

Visiossa jokainen käyttäjän suorittama kumottavissa oleva komento aiheuttaa automaattisesti Vision sisäisen *IVBUndoUnitin*, joka kuvaa muutoksen kaiken sisällön ja säilyttää tarvittavat tiedot sen kumoamiseksi. Laajennuskomponenttia käytettäessä muutokset kuitenkin ovat Vision käyttöliittymämuutoksia laajempia ja saattavat aiheuttaa tietokannassa monimutkaisiakin mallimuutoksia, joiden kaikkien tulee olla käyttäjän kumottavissa. Sama koskee myös muita ympäristön työkaluja, mistä syystä Trinityn tietokantakomponenttiin on toteutettu ”kumoa ja tee uudelleen”-toiminnallisuudelle (*undo* ja *redo*) yleinen tuki, jota hyödynnetään myös Vision laajennuksessa.

Tietokantakomponentti tarjoaa *IUndoManagerin*, joka luo automaattisesti yksittäisiin tietokantatransaktioihin sidottavia *undo*- ja *redo*-yksiköitä. Vision automaattisesti luomat *IVBUndoUnitit* korvataan yksinkertaisesti näillä tietokantaoperaatioita vastaavilla *undo*- ja *redo*-yksiköillä, jolloin kaikki tietokantaan tehdyt muutokset saadaan mukaan Vision *undo*- ja *redo*-komentoihin.

Vision *undo*- ja *redo*-pinot voidaan tyhjentää ja niihin voidaan lisätä uusia *IVBUndoUnit*-rajapinnan toteuttavia olioita ohjelmallisesti. Vision omien *undo*- ja *redo*-pinojen korvaaminen tietokantakomponentin yksiköillä toteutetaan laajennuskomponentin *ViewController*-kerroksen *DBUndoRedoHandler*-luokassa aina, kun tietokantakomponentin *UndoManager* lähettää tapahtuman sisältönsä muutoksesta. Kuva 6.9 esittää esimerkin tapahtumaketjusta kokonaisuudessaan.



Kuva 6.9. Undo- ja redo-toiminnallisuuden yleinen tapahtumaketju.

Vision käyttöliittymän *undo*- ja *redo*-pinot päivitetään vain tarvittaessa Vision ollessa toimeettomana. Kun päivitys on tarpeen, haetaan tietokantakomponentin *undo*- ja *redo*-yksiköistä niiden nimet, joille jokaiselle luodaan oma *DBUndoUnit*-olio. *DBUndoUnit* on *IVBUndoUnit*-rajapinnan toteuttama laajennuskomponentin luokka, joka sisältää *undo*- ja *redo*-metodit. Vision automaatorajapinta sisältää metodin vain *undo*-yksikön lisäämiseksi, joten sekä *undo*- ja *redo*-yksiköt joudutaan lisäämään ensin *undo*-pinoon. Tämän jälkeen *redo*-yksiköiden *undo*-metodia kutsutaan manuaalisesti, jolloin ne siirtyvät *redo*-pinoon. *Redo*-yksikkö ei kuitenkaan reagoi tähän *undo*-kutsuun mitenkään. Kun käyttäjä suorittaa *undo*- tai *redo*-toiminnon, aiheuttaa *DBUndoUnit* Vision *MarkerEvent*-tapahtuman, jonka *ProjectController* käsittelee ja välittää kutsun *IUndoManager*ille. Tietokantakomponentti suorittaa muutokset tietokannassa, jonka jälkeen *IUndoManager* ilmoittaa sen sisällön muutoksesta *DBUndoHandler*ille, joka lopuksi

päivittää Vision *undo*- ja *redo*-pinot kun Visio on taas joutilaana. *DBUndoUnit* kierrättää *undo*- ja *redo*-kutsut *MarkerEvent*-tapahtuman kautta, jotta mahdollisesti kesken olevat operaatiot suoritetaan ensin loppuun ja suoritus siirtyy Vision säikeeseen.

7. VISIO-MALLINNUSTYÖKALUN LAAJENNETTAVUUS

Visio-mallinnustyökalun merkittävin laajennettavuusominaisuus on uusien mallinnuskielien lisääminen. Tässä luvussa esitellään keskeiset menetelmät sekä kielien näkymäelementtien määrittelyyn Visiossa että laajennuskomponentin perustoiminnallisuuden laajentamiseen ohjelmakoodissa. Ohjelmakoodilaajennuksilla mahdollistetaan muun muassa kielikohtaisia erityistoimintoja ja automaatiota.

7.1. Kielien näkymäelementtien määrittely Visiossa

Trinity-ympäristössä mallinnuskielien abstrakti syntaksi ja kaaviotyypit määritellään tietokannan metamalleissa (katso kohta 6.2.1), jolloin mallinnustyökalujen vastuulle jää niiden työkalukohtaisten konkreettisen syntaksin määrittely. Käytännössä tämä tarkoittaa työkalun *template*-näkyäelementtien luomista ja linkittämistä kielessä määriteltyihin elementtityyppeihin tietokantakomponentin avulla. Visiossa *template*-näkyäelementit ovat *Master*-elementtejä (perusmuotoja), joiden määrittelyn keskeisimmät kohdat esitellään seuraavaksi.

Visio-mallinnustyökalun elementtien näkymädata on lähes yksinomaan luvussa 2.3.6 kuvattua *ShapeSheet*-dataa, joka määritellään aina *Master*-elementeissä (Kuva 7.1). *Master*-elementin *ShapeSheet* tallennetaan tietokantaan kokonaisuudessaan ja ladataan sieltä sen näkymätyypin avauksen yhteydessä. *Master*-elementti luodaan Visiossa sen valmiilla *Master*-muokkaimella piirtämällä sen graafiset muodot ja määrittelemällä sille halutut mallinnuskieleen liittyvät ominaisuudet, jonka jälkeen se voidaan lisätä johonkin tietokannassa olevan näkymätyyppiin laajennuskomponentin tarjoamalla kehittäjille suunnatulla *Master*-lisäysdialogilla. Sama Visio Trinity-valikosta löytyvä dialogi toimii myös olemassa olevan *Master*-elementin päivittämiseen.

User-defined Cells	
User.Guid	"BC8E1F67-5009-43e7-9F72-C6355D0109FA"
User.Text	User.TextSource
User.MetaProperties	"MMType=UML_SimpleState"
User.ModelProperties	"name,baselineref_type"
User.name	"SimpleState"
User.TextSource	User.name
User.isprimaryreference	TRUE
User.ViewProperties	"isprimaryreference"

Kuva 7.1. Esimerkki näkyäelementin ominaisuuksien määrittelystä *ShapeSheet*issä.

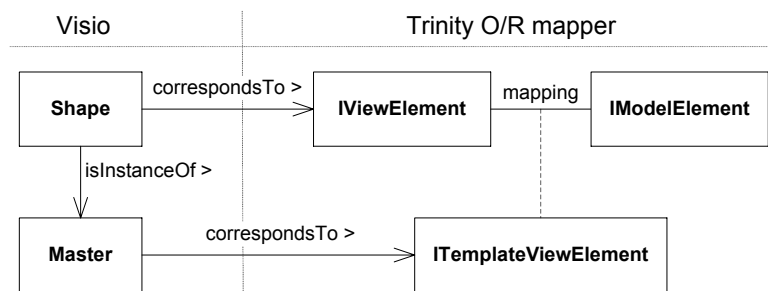
Kaikki kieleen liittyvät ominaisuudet määritellään *ShapeSheet*-datan ”User-defined Cells”-osiossa (Kuva 7.1). Olennaisimmat määriteltävät ominaisuudet ovat:

- *Master*-elementtiä vastaavan *ITemplateViewElementin* nimi (esimerkiksi UML-luokka).
- Suhteet, joita luodaan mallielementtien välille, kun sen *Shape* yhdistetään toiseen *Shapeen*.
- Ominaisuudet, joista *Shape* on kiinnostunut ja käsittelee jotenkin (esimerkiksi mallielementin nimi-ominaisuus, joka halutaan näyttää).

Yllä mainituista ominaisuuksista kaksi ensimmäistä tallennetaan *ShapeSheet*-datan lisäksi suoraan *ITemplateViewElementin* metatietoihin, jolloin määriteltyjä ominaisuuksia ei ole tarve jäsentää aina *ShapeSheet*-datasta ja niitä voidaan uudelleenkäyttää lapsielementtejä luotaessa *Informaatiopaneelista*. Sen sijaan *Shape*-elementtiä kiinnostavien ominaisuuksien määrittelyjä ei tallenneta tietokantakomponentin elementteihin, sillä ne ovat puhtaasti pelkästään Vision näkymäelementteihin liittyviä ominaisuuksia. Molemmien tyyppisten ominaisuuksien määrittely *ShapeSheet*issä esitellään seuraavaksi omissa aliluvuissaan.

7.1.1. Template-elementin metaominaisuudet

Template-elementin ominaisuudet määritellään *ShapeSheetin* *User.MetaProperties*-rivillä puolipisteillä toisistaan eroteltuina. Yksittäisen ominaisuuden nimi ja arvo erotetaan toisistaan puolipisteellä. *Template*-elementin tyyppi määritellään avainsanalla *MMType* (Taulukko 7.1). Kuvan 7.1 esimerkissä elementin tyyppi määritellään *UML_SimpleState*, jolloin *Master*-elementti linkitetään tietokantakomponentin vastaavan nimiseen *ITemplateViewElement*-elementtiin, joka puolestaan linkittyy tämän tyyppiseen metametamallin näkymä- ja mallielementtiin. Tämän perusteella kohdemalliin luodaan tämän tyyppiset elementit, kun uusi *Shape* luodaan Visiossa. Jos *MMType* jätetään määrittelemättä, luodaan ainoastaan malliin ainoastaan näkymäelementti. Esimerkiksi vapaamuotoisia elementtejä varten jokaisessa näkymätyypissä on geneerinen *Master*-elementti, jolla ei määritellä *MMTypeä*. Käyttäjän piirtämä vapaa *Shape* korvataan tällä *Master*-elementillä ja kaikki *Shapen* graafiset ominaisuudet kopioidaan sille. Vision *Master*- ja *Shape*-elementtien suhteet tietokantakomponentin elementteihin on esitetty havainnollisemmin kuvassa 7.2.



Kuva 7.2. Vision *Master*- ja *Shape*-elementtien linkittyminen tietokantakomponentin elementteihin.

Kaaviossa olevia *Shape*-elementtejä voidaan yhdistää toisiinsa joko viiva-*Shapeilla* tai muodostamalla erityyppisiä isä-lapsi-suhteita. Mahdolliset suhdetyypit ja niiden määrittelyyn käytetyt avainsanat on lueteltu taulukossa 7.1.

Taulukko 7.1. Olennaiset avainsanat template-elementin meta-ominaisuuksien määrittelyyn User.MetaProperties-rivillä.

Määriteltävä ominaisuus	Tarvittavat avainsanat	Selite
Metameta-elementin tyyppi	<i>MMType</i> =Tyyppi	Kielessä määritellyn mallielementin nimi. Itse template-elementin nimi voi olla eri.
Yksinkertainen suhde, jossa yksi mahdollinen suhde.	<i>ConnectionName</i> =suhteen_nimi; <i>ConnectionEndName</i> = viiva_pää; <i>ConnectionOtherEndName</i> =toinen_pää;	Lyhyempi määrittelytapa viivasuhteelle, jossa molemmat päät ja niiden suhteet ovat samat. Määritellään yhdistävässä elementissä (esimerkiksi UML-assosiaatio).
Viivasuhde, jossa eri suhde viivan päissä.	<i>BeginConnectionName</i> =alkupää; <i>BeginConnection_OtherEndName</i> =alk1; <i>BeginConnection_LineEndName</i> =alk2; <i>EndConnectionName</i> =loppupää; <i>EndConnection_OtherEndName</i> =lop1; <i>EndConnection_LineEndName</i> =lop2" <i>ConnectionType</i> =NoChildren;	Esimerkki tällaisesta suhteesta on UML-periytyminen, jossa suhteiden päät ovat eri roolissa (<i>general</i> ja <i>specifc</i>). Lisämääre <i>ConnectionType</i> tarkoittaa, että viivalla ei ole erillisiä päitä (kuten esimerkiksi UML-assosiaation päät). Määritellään viiva-Masterissa
Dynaaminen isä-lapsi-suhde	<i>RelationshipName</i> =suhteen_nimi <i>ParentEndName</i> = isä_pää <i>ChildEndName</i> = lapsi_pää <i>SubElementType</i> =lapsen_tyyppi	Vain yksi mahdollinen suhde. Määritellään isäelementissä. Määre <i>SubElementType</i> määrittelee isäelementtiin luotavien elementtien tyyppin.
Dynaaminen isä-lapsi-suhde (useita mahdollisia)	<i>RelationshipNames</i> =suhde1, suhde2, suhde3;	Käytetään tilanteissa, kun suhde riippuu lapsielementin tyyppistä. Määritellään isäelementissä. Annetuista vaihtoehtoista oikea suhde ja päiden nimet tulkitaan metamallin <i>parent</i> - ja <i>child</i> -rooleista.
Staattinen isä-lapsi-suhde	<i>CreationConnectionName</i> =suhteen_nimi <i>CreationConnectionEndNameThis</i> =lapsi <i>CreationConnectionEndNameOther</i> =isä	Staattinen yhden suhde yhteen. Määritellään lapsielementissä.

Viivasuhteilla on aina kaksi päätä, joihin voidaan määritellä joko samat tai eri suhteet. Viivasuhteet määritellään aina viivaelementissä, kun taas isä-lapsisuhteessa määrittelevä osapuoli riippuu suhteen tyyppistä. Esimerkiksi UML:n luokkaelementtiin attribuuttia luotaessa kyseessä on niin sanottu dynaaminen isä-lapsi-suhde, jossa luokan näkymäelementtiin voi luoda ja poistaa mielivaltaisen määrän attribuutteja. Dynaamiset lapsisuhteet määritellään aina isäelementissä. Staattinen suhde voi olla puolestaan tilanteessa, jossa elementti omistaa aina täsmälleen yhden elementin lapsena. Tämä suhde luodaan heti elementtejä luotaessa eikä sitä koskaan poisteta ilman molempien osapuolien poistamista. Suurin osa moniosaisista Master-elementeistä määritellään sisäkkäisil-

lä elementeillä ja yhdistetään toisiinsa tätä suhdetta käyttäen. Staattisen suhteen määrittely tehdään lapsielementissä.

7.1.2. Näkymäelementtiä kiinnostavat ominaisuudet

Ominaisuudet, joista muoto on kiinnostunut, määritellään pilkulla eroteltuina taulukossa 7.2 esitellyillä *ShapeSheetin* riveillä. Ominaisuudet voivat olla muodon edustaman näkymäelementin, mallielementin, mallin, näkymän tai jonkin suhteen tiettyssä roolissa olevan toisen elementin ominaisuuksia, joiden arvoja muoto tarvitsee halutun tiedon esittämiseen.

Taulukko 7.2. *ShapeSheet*-rivit *Shapea* kiinnostavien ominaisuuksien määrittelyyn.

Rivi	Sisältö
<i>User.ModelProperties</i>	Shapeen liittyvän mallielementin (mikäli sellainen on olemassa) ominaisuudet.
<i>User.ViewProperties</i>	Shapeen liittyvän näkymäelementin ominaisuudet.
<i>User.ModelsProperties</i>	Mallin, jossa vastaava näkymäelementti sijaitsee, ominaisuudet.
<i>User.ViewsProperties</i>	Mallin, jossa vastaava näkymäelementti sijaitsee, ominaisuudet.
<i>User.RelEndRoleProperties</i>	Jonkin suhteen määritellyssä roolissa olevan elementin ominaisuudet. Ominaisuus voi olla usean suhteen päässä, joten roolit voidaan ketjuttaa alla olevan BNF:n mukaisesti: $\langle arvo \rangle ::= \langle rooli \rangle [_ \langle rooli \rangle]^* _ \langle ominaisuus \rangle$

Jokaista lueteltua ominaisuutta varten on lisäksi luotava oma *User*-rivi, johon laajennuskomponentin *ShapeController*-komponentti tallettaa kyseisten ominaisuuksien arvot kaavion avauksessa tai arvon muuttuessa tietokannassa (katso kohta 6.3.1, Kuva 6.6). Mikäli *Master*-elementti ei sisällä riviä jollekin ominaisuudelle, luodaan sellainen automaattisesti *Shapen ShapeSheetiin*. Tämä ei kuitenkaan ole optimaalista ja rivin määrittely *Master*-elementissä on suotavaa suorituskyvyn parantamiseksi ja tiedon duplikoimisen välttämiseksi.

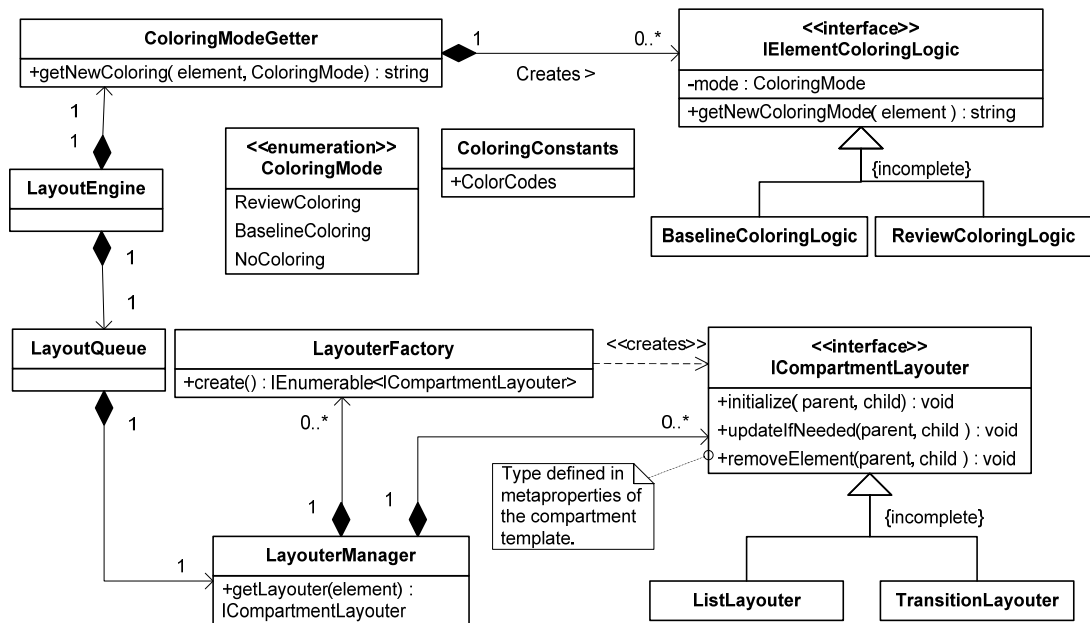
Yleisin määriteltävä ominaisuus on nimi, joka halutaan näyttää *Shapessa* ja jota voidaan muokata *Shapeen* kirjoittamalla. Laajennuskomponentin toteutuksessa yksi *Shape* sisältää yhden tekstikentän (*TextField*), joka linkitetään johonkin ominaisuuteen. Ominaisuus määritellään *User.TextSource*-rivillä, ja sen arvoksi asetetaan viittaus halutun ominaisuuden *User*-riviin. Varsinaisen *TextField*-kentän sisällöksi voidaan haluta asettaa muutakin tekstiä kuin ominaisuuden arvo, joten tekstikentän sisältö määritellään erikseen *User.Text*-rivillä, joka voi sisältää *User.TextSource*-sisällön lisäksi muuta tekstiä.

7.2. Ohjelmakoodin laajennettavuus

Yksinkertaisimmillaan Visio-mallinnustyökalua voidaan laajentaa uusilla mallinnuskielillä luvussa 7.1 kuvatuilla kielen näkymäelementtien määrittelyllä, jolloin laajennuskomponentin ohjelmakoodiin ei tarvita muutoksia. Monimutkaisempien kielten tapauksissa voidaan kuitenkin tarvita lisäautomaatiota, kuten asemoiteja (layouting) tai muita lisätoimintoja. Tätä varten laajennuskomponentti tarjoaa seuraavissa aliluvuissa esiteltyt laajennusmekanismit. Jokainen mainittavista mekanismeista sijaitsee *ViewController*-komponentissa omassa nimiavaruudessaan ja niiden hallinnasta vastaa *ProjectController*.

7.2.1. Asemoijat ja väritystilat

Laajennuskomponentin yksittäisten elementtien asemoinnit ja värjäykset toteutetaan *ViewController*-komponentin *Layouts*-nimiavaruudessa, jonka olennainen sisältö on esitelty kuvassa 7.3. Toimintojen suorittamisesta vastaa *LayoutEngine*-luokka, johon lisätään jonoon elementtien asemoimispyyntöjä, jotka ajetaan haluttuna ajanhetkenä.



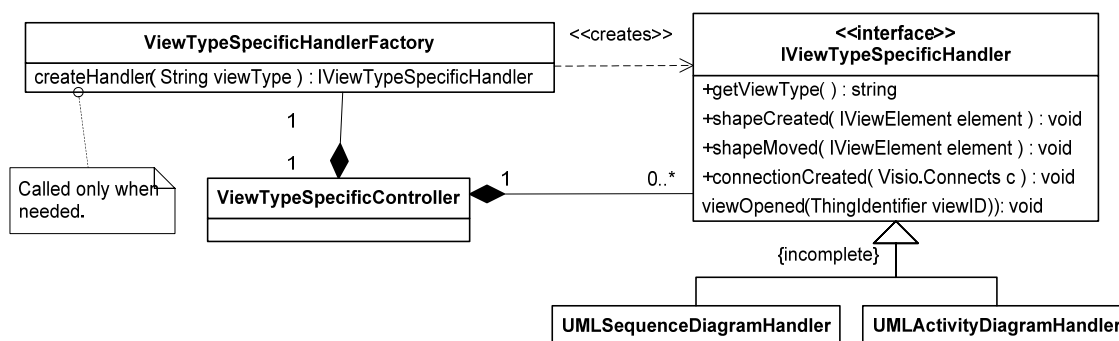
Kuva 7.3. Asemoinnin keskeiset luokat.

Elementtien asemoiminen on tarpeen tilanteessa, jossa jonkin elementin johonkin osion (compartment) sisään voidaan pudottaa uusia elementtejä tai poistaa niitä, kuten UML-luokan attribuuttiosioon. Uusien asemoijien lisääminen tapahtuu yksinkertaisesti toteuttamalla *ICompartmentLayouter*-rajapinnan luokka, jonka instanssin luominen lisätään *LayouterFactory*-luokan *create*-metodiin. Halutun asemoijan nimi määritellään *template*-elementin meta-ominaisuuksissa (katso kohta 7.1.1) avainsanalla ”CompartmentType”.

Uusien väritystilojen lisääminen onnistuu *IElementColoringLogic*-rajapinnan luokan lisäämisellä *ColoringModeGetter*-luokan *getNewColoringMode*-metodiin. Tämä luokka sisältää logiikan, jolla elementti värjätään ja sen tulee palauttaa jokin värityskoodivakio, jotka määritellään *ColoringConstants*-luokassa. *ColoringMode*-enumeraattori on logiikkaa vastaava moodi, joka valitaan *Informaatiopaneelin* ikkunasta ja välitetään *LayoutEngine*-oliolle.

7.2.2. Näkymätyyppikohtaiset käsittelijät

Näkymätyyppikohtaiset käsittelijät tulevat tarpeeseen, kun halutaan monimutkaista näkymäautomaatiota tai mallitason operaatioita. Tarpeet ovat usein näkymätyyppikohtaisia ja laajennuskomponentti tarjoaa uusien käsittelijöiden lisäämiseen yksinkertaisen tehtaan ja rajapinnan sen *ViewTypeSpecificHandlers*-nimiavaruudessa (Kuva 7.4)

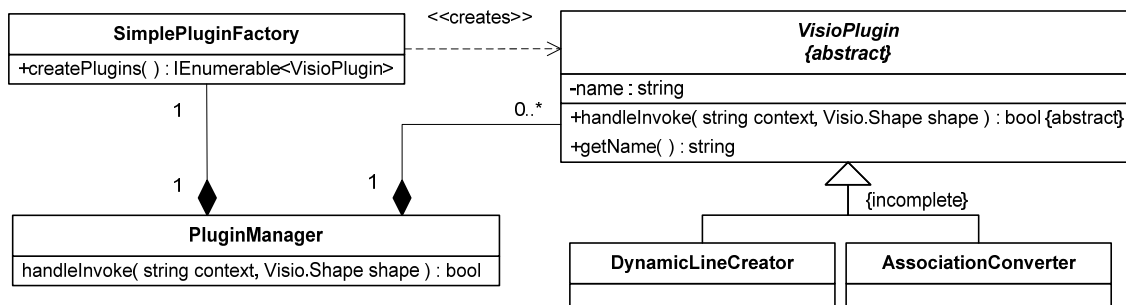


Kuva 7.4. Näkymäkohtaiset käsittelijät.

Käsittelijän tulee toteuttaa *IViewTypeSpecificHandler*-rajapinta, joka sisältää menet- dit useimmin tarvittavien tilanteiden käsittelyyn. Näitä tilanteita ovat esimerkiksi ele- menttien luomiset, liikuttelut ja yhdistämiset sekä kaavioiden avaamiset. Käsittelijä lisä- tään laajennuskomponenttiin *ViewTypeSpecificHandlerFactory*-luokassa, joka luo täs- mälleen yhden käsittelijän jokaiselle näkymätyypille, mikäli sille sellainen on toteutettu.

7.2.3. MarkerEvent-pluginit

MarkerEvent-tapahtumat ovat ainoa tapa kutsua Visiosta *add-in*-laajennus- komponentteja suoraan, kuten luvussa 2.3.5 mainittiin. Tästä syystä esimerkiksi kon- tekstimenuista käynnistettävät toiminnot ja luvussa 5.4.1 esitelty viivojen piirtotoiminto on toteutettu niiden avulla. Laajennuskomponentti tarjoaa yksinkertaisen *plugin*- menetelmän uusien vastaavanlaisten toimintojen lisäämiseen (Kuva 7.5).



Kuva 7.5. MarkerEvent-pluginit.

Kaikki *pluginit* periytyvät abstraktista luokasta *VisioPlugin* ja luodaan *SimplePluginFactory*-luokassa sovelluksen käynnistyessä. *Pluginit* yksilöidään niiden nimien perusteella ja kutsuttava *plugin* määritellään *MarkerEvent*-tapahtumassa. *MarkerEvent*-tapahtumat sisältävät aina merkkijonon *ContextString*, joka sisältää avain-arvo-pareja ($\langle ContextString \rangle ::= ['/' \langle avain \rangle ' = ' \langle arvo \rangle ']^*$). Kaikki *plugin*-kutsut määritellään avainsanan ”cmd” arvolla ”PluginInvoke” ja haluttu *plugin* avainsanalla ”plugin”, kuten esimerkiksi ”/cmd=PluginInvoke /plugin=AssociationConverter”. Muut parametrit riippuvat *pluginista*, ja niiden tulkinta on *pluginien* itsensä vastuulla.

8. ANALYYSI

Tässä luvussa arvioidaan diplomityössä Trinity-projektiin toteutetun Visio-laajennuskomponentin toteutusta ja pohditaan, miten sitä tulevaisuudessa tullaan tai tulisi kehittää eteenpäin. Tämän jälkeen esitellään lisäksi muutama työn aihepiiriin liittyvä julkaisu ja vertaillaan niitä Trinity-projektin ja sen mallinnustyökalujen toteutuksen ja lähestymistapaan.

8.1. Toteutuksen arviointi

Työssä toteutettu Visio-laajennuskomponentti on tiukasti osa isompaa Trinity-projektia, joka pyrkii vastaamaan muun muassa joustavan mallinnuksen haasteisiin integroidulla mallinnustyökaluympäristöllä. Tästä syystä myös laajennuskomponentin vaatimukset pohjautuivat merkittävästi ympäristön asettamiin vaatimuksiin ja näiden toteutumista on perusteltua arvioida yhdessä. Taulukko 8.1 kerää yhteen kaikki luvussa 3 esitellyt yksittäiset vaatimukset sekä niiden täyttymisen kolmiportaisella asteikolla. Asteikon miinusmerkki ilmaisee ettei vaatimus täyttynyt, nolla osittaista täyttymistä ja plus-merkki täyttymistä. Arviot ovat osin subjektiivisia, mutta ne pyritään perustelemaan seuraavassa.

Työn tavoitteena oli Vision valjastaminen yhdeksi graafiseksi ja joustavaksi mallinnustyökalunäkymäksi Trinity-ympäristöön. Tähän tavoitteeseen päästiin Visioon toteutetulla *COM Add-in* -laajennuskomponentilla, joka on toistaiseksi osoittautunut toimivaksi ratkaisuksi. Visio tarjosi kaiken tarvittavan toiminnallisuuden monipuolisen ja kohtalaisen helppokäyttöisen rajapintansa kautta, eikä ylitsepääsemättömiin ongelmiin törmätty. Visio-laajennukselle asetetut vaatimukset saatiin täytettyä kiitettävästi ja sen käytöstä on saatu jo positiivisia kokemuksia myös teollisuusyhteistyökumppanilta. Komponentin toteutus aloitettiin UML 1.4:n kaaviotyypeillä ja kirjoitushetkellä kaaviotukea laajennetaan version 2.3:n kaavioihin. Uusien mallinnuskielien lisääminen nykyisellä toteutuksella on osoittautunut yksinkertaiseksi, sillä se ei vaadi muutoksia ohjelmakoodiin. Käytännössä kaikki työ liittyy Vision *template*-elementtien ja niiden *ShapeSheet*-kaavojen luomiseen. Poikkeus tähän ovat asemoinnit ja ylimääräiset näkymäautomaatiot, jota varten laajennuskomponenttiin toteutettiin yksinkertaiset laajennusmekanismit. Perustoteutus riittää geneerisyytensä ansiosta kuitenkin yleisimpiin tilanteisiin, eikä laajennusmekanismeja ole tarve hyödyntää kovinkaan usein.

Geneerisyyden kääntöpuolena on kuitenkin osittain runsas monimutkaisen ohjelmakoodin määrä, joka etenkin *ShapeController*-luokan kohdalla on jonkinasteinen ongelma. *ShapeController* toimii keskeisimpänä moottorina Vision rajapinnan käsittelijänä ja sen vastuulle on kertynyt liiankin moni perustoiminnallisuus. Muutosten tekeminen tiettyihin olemassa oleviin toiminnallisuuksiin on tästä syystä osoittautunut aikaa vie-

väksi. Nykyinen versio luokasta on kuitenkin varsin toimiva ja vakaa, eikä muutoksille ole ollut vähään aikaan tarvetta. Muilta osin toteutus on ylläpidettävyydeltään selkeästi korkeammalla tasolla.

Vision käyttöliittymä ja ominaisuudet saatiin säilytettyä vaatimusten mukaan alkuperäisen kaltaisena, eikä mallinnusominaisuuksien käyttö vaadi juurikaan opettelua Visiota aikaisemmin käyttäneelle. Käyttöliittymää muokattiin vain mallitietoa näyttävällä Informaatiopaneelilla ja lisäämällä Vision valikoihin uusia Trinity-kohtaisia toimintoja. Myös vapaamuotoiset graafiset elementit tallennetaan osaksi mallia ja niitä voidaan hyödyntää myöhemmin, joten joustavan mallinnuksen luonnosteluvaatimuksia saatiin tältä osin täytettyä kiittävästi. Lisäksi piirrettyjen mallien ei vaadita noudattavan kielien metamallia, vaan sen tarkistus jätetään muiden ympäristön komponenttien vastuulle.

Taulukko 8.1. Luvussa 3 asetettujen vaatimusten täyttymisen arviointi.

Vaatus	Toteutuminen
<i>Ympäristön työkalu – Käytettävyys ja ominaisuudet</i>	
Ei lisää ylimääräisiä vaiheita työhön	+
Vapaamuotoisten elementtien salliminen	+
Metamallin vastaisten mallien luominen	+
Alkuperäisen käyttöliittymän ja toiminnallisuuden säilyttäminen	+
Suorituskyvyn riittävyys ja käyttökokemuksen säilyminen	0
Indikaattorit poikkeuksellisen pitkistä latausoperaatioista	+
<i>Ympäristön työkalu – Osana ympäristöä</i>	
Tuki tietokannalle ja tiedon päivittämiselle reaaliaikaisesti	+
Tuki kommunikaatiolle muun ympäristön työkalujen kanssa	+
Yhteisen toiminnallisuuden eriyttäminen ja hyödyntäminen	+
<i>Visio-spesifiset</i>	
Tuki mallien luomiselle ympäristön tietokantaan	+
Uusien kielten lisäämisen helppous	+
Toteutuksen geneerisyys	+
Katselmointituki	+
Valmiit väritystilat sekä vapaat muotoilut	+
Asemoinnit ja automaatio, sekä niiden laajennettavuus	+
<i>Ympäristö – työkaluintegraatio</i>	
Yksittäinen työkalu yhtenä näkymänä malliin	+
Reaaliaikainen kollaboratiivinen työskentely	+
Erillinen mallien hallinta, irrallaan mallinnustyökaluista	+
<i>Ympäristö – tietointegraatio</i>	
Mallit keskitetyssä paikassa	+
Offline-työskentely	-
Mallinnuskielten lisääminen järjestelmään mahdollista helposti	+
Kaikki tieto osa mallia ja sen päivitys kaikkialle välittömästi	+

Ympäristön vaatimuksena sen työkaluille oli tuki kollaboratiiviselle työskentelylle ja tiedon tallentamiselle erilliseen tietokantaan, joihin Visio-laajennuskomponentti kykenee vastaamaan onnistuneesti. Suurin osa tästä toiminnallisuudesta perustuu yhteisen tietokannan ja tietokantakomponentin toteutukseen, jolloin haasteellisimmaksi osuudeksi tältä osin muodostui ulkoisten muutosten päivittäminen Visioon samaan aikaan, kun käyttäjä tekee jotain sen käyttöliittymässä. Näihin ongelmiin löydettiin ratkaisut (katso kohta 6.3.2), mutta vasteaika kärsii niistä jonkin verran. Kokonaisuudessaan suurimmat suorituskykyongelmat johtunevat kuitenkin keskitetyn tietokannan viiveistä ja suuresta määrästä ympäristön komponentteja, jotka sitä käyttävät. Työkalua käytettäessä eron perinteisen Vision käyttöön huomaa, joten asetettuja suorituskykyvaatimuksia ei vielä tältä osin saatu täysin täytettyä. Työkalu on tästä huolimatta käyttökokemukseltaan tyydyttävällä tasolla.

Laajennuskomponentin toteutuksessa nojataan suurilta osin vaatimusten mukaisesti ympäristön yhteiseen toiminnallisuuteen tietokantakomponentin, Informaatiopaneelin ja hallintakäyttöliittymän muodossa. Laajennuskomponentin toteutuksen alkuvaiheessa yleistä toiminnallisuutta ei kuitenkaan ollut vielä paljoa, joten sitä on jälkikäteen eriytetty kokemusten perusteella Visio-laajennuskomponentista tietokantakomponenttiin ja sen laajennuksiin. Tällaisia ovat muun muassa elementtien kopioimistoiminnot, joita kirjoitushetkellä hyödynnetään myös muissa ympäristön työkaluissa. Itse Visio-laajennuksen toteutus saatiin säilytettyä yksinkertaisena niin, että se toimii vain väli-komponenttina Visio-näkymän ja tietokannan mallien välillä. Tällainen lähestymistapa aikaisen työkalutuen edistämiseksi on työn aikana osoittautunut varsin tehokkaaksi ja joustavaksi.

8.2. Tulevaisuus ja parannusehdotuksia

Lähitulevaisuudessa Visio-laajennuskomponenttia tullaan jatkokehittämään laajentamalla sitä uusilla kielillä, joista ensisijaisia ovat kaikki UML2.3:n kaaviotyypit. Tämän lisäksi metamallin sääntöjen tarkistuksesta vastaava sääntökomponentti on työn alla ja sille tultaneen toteuttamaan jonkinlainen esitysmuoto myös Visio-mallinnustyökaluun.

Työssä toteutettu laajennuskomponentti on Vision versiolle 2007, mutta kirjoitushetkellä uusin versio 2010 on jo julkaistu (Microsoft 2011k). Visio 2010 tuo mukanaan joukon uusia ominaisuuksia, joilla nykyisen toteutuksen käytettävyyttä saataisiin parannettua. Näitä ovat esimerkiksi uudenlaiset säiliöt (containers), niistä erikoistetut listat (lists), älykkäämmät asemoinnit ja monitasoisten kontekstivalikoiden tuki. Säiliöiden avulla päästäisiin eroon muun muassa sisäkkäisten elementtien ongelmista, jossa isä-elementin venyttäminen liikuttaa aina lapsielementtejä ja useita lapsielementtejä ei voida valita kerrallaan hiirellä maalaamalla. Listat puolestaan sisältävät kehittyneet ominaisuudet lapsielementtien luomiseen ja järjestelemiseen. Näihin on nykyisellään tarvittu oma toteutus, joka ei vastaa käytettävyydeltään 2010:n valmiita ominaisuuksia. Visio 2010:n API sisältää pääasiassa vain laajennuksia 2007:n API:iin, joten siirtyminen siihen lienee mahdollista pienellä vaivalla ja on täten kaikin puolin suositeltavaa.

Parannettavaa laajennuskomponentissa on luvussa 8.1 esille tuotu suorituskyky sekä erinäiset riippuvuussuhteet. Heikko suorituskyky on eniten riippuvainen tietokantayhteydestä, mutta myös Vision rajapinnan käytössä on optimoimisen varaa. Esimerkiksi asemointien tapauksissa useita *ShapeSheet*-solujen arvoja kerralla asettavaa *setFormulas*-metodia ei käytetä kaikkialla lähinnä sen monimutkaisemman toteutuksen vuoksi. Tämä vaivannäkö voisi kuitenkin olla suuremmissa kaavioissa merkityksellinen käyttökokemuksen kannalta.

Tämän hetken toteutus on varsin riippuvainen tietokantakomponentin tarjoamista rajapinnoista, sillä niitä käytetään useassa paikassa suoraan. Esimerkiksi *ViewController*-kerros käyttää sen palveluita, vaikka ne olisi järkevämpää suorittaa *Model*-kerroksen avulla. Tämä ei kuitenkaan ole suuri ongelma, sillä tietokantakomponentti itsessään on toteutettu niin, että sen rajapinnat eivät ole riippuvaisia käytetystä tietokannasta ja niiden ei näin oleteta muuttuvan helposti. Toinen riippuvuus juontuu siitä, että laajennuskomponentti on toteutettu yhtenä DLL:nä, joka sisältää kaiken toiminnallisuuden. Näin ollen esimerkiksi toteutettuja *template*-elementtejä ei voida käyttää hyväksi ilman laajennuskomponenttia, mikäli ne nojaavat sen asemointitoimintoihin. Elementtien asemoinnit ja muut laajennuskomponentin laajennukset voitaisiin tulevaisuudessa toteuttaa esimerkiksi erillisinä *add-on*-komponentteina, jolloin niitä voitaisiin hyödyntää modulaarisemmin ja uusia lisätä myös ajonaikaisesti.

8.3. Aiheeseen liittyviä julkaisuja

Käsite joustava mallinnustyökalu on melko uusi, mutta siihen viittaavia tutkimuksia on tehty viime aikoina yhä enenevässä määrin. Mallinnustyökalujen integraatioita sen sijaan on tutkittu laajemmin jo 1980-luvulta asti (Wicks & Devar 2007). Seuraavassa esitellään lyhyesti muutama näihin aiheisiin liittyvä julkaisu.

Ossher et al. (2010) esittelevät käsitteen “flexible modeling tools” uutena tutkimuskohteena, joka pyrkii yhdistämään perinteisen mallinnustyökalujen ja toimistosovellusten parhaita puolia Trinityn tavoin. Kirjoittajat tunnistavat samoja varhaisen mallinnustyökalutuen ongelmia ja perinteisten mallinnustyökalujen puutteita, mistä syystä projektin esimäärittelyvaiheessa käyttäjät turvautuvat mieluiten toimistotyökaluihin ja tehtyä työtä menetetään. Heidän mukaan parhaat puolet voitaisiin yhdistää joko muokkaamalla mallinnustyökaluja joustavammaksi, lisäämällä toimistotyökaluihin mallinnusominaisuuksia tai luomalla kokonaan uusi työkalu. Kirjoittajat esittävät yleisen käsitteellisen arkkitehtuurin, jonka avulla tavoitteeseen voidaan päästä, valittiin edellä mainituista toteutustavoista mikä hyvänsä. Karkeasti ottaen arkkitehtuuri erottaa näkymäkerroksen useat eri näkymät malleista, linkittää ne toisiinsa ja erottaa rakenteellisten kohdealuekohtaiset rajoitteet sekä niiden tarkistelun erilliseksi opastavaksi komponenttikseen. Tämän lisäksi se sallii vapaamuotoisten elementtien lisäämisen osaksi mallia.

Trinity-projektin ja Visio-mallinnustyökalun ratkaisut toimivat suurelta osin samoilla periaatteilla. Ossher et al. kuitenkin rajoittuu yksittäisen työkalun tarkasteluun ilman kollaboratiivisuuden tai keskitetyn tietokannan tuomia hyötyjä. Tutkimus perustelee

kuitenkin joustavien mallinnustyökalujen tarpeen hyvin ja luo perustaa niiden jatkotutkimukselle.

Volz & Jablonski (2010) pyrkii kaventamaan piirustus- ja mallinnustyökalujen välistä kuilua omalla joustavalla OMME-mallinnustyökaluympäristöllään. Se on Trinityn tavoin metamallipohjainen ja perustuu olemassa olevien työkalujen, ohjelmistokehityksen ja kirjastojen hyödyntämiseen. OMME:n lähestymistapa on Trinity-ympäristön kanssa hyvin samankaltainen, mutta rajoittaa mallien muokkaamisen yhteen työkaluun. Ympäristö on nimittäin toteutettu Eclipse-alustalle (Eclipse 2011), jossa myös kaikki graafinen editointi suoritetaan. Kaikki mallit säilötään *EMF*:ään, mutta jonkinlainen tiimituki sekä keskitetty mallivarasto on tekijöiden mukaan myös tulossa tulevaisuudessa. Näiden käytön reaaliaikaisuuteen tai kollaboraatiomahdollisuuksiin ei sen sijaan oteta kantaa. Ympäristön keskeisin kontribuutio on tuki epäformaalille metamallintamiselle sallivassa toimistotyökalumaisessa ympäristössä, joka on laajennettavissa myös kehittyneemmällä mallinnusparadigmoilla.

MOFLON (Amelunxen et al. 2008) puolestaan on perinteinen metamallipohjainen työkaluintegraatioympäristö, joka integroi mallinnustyökalut muuntamalla niiden sisäiset mallit MOF 2.0-standardin mukaisiksi malleiksi. Sen toiminta perustuu mallimuunnokset tarjoavaan kehikseen, jonka avulla työkalujen mallit ovat keskenään vaihtokelpoisia. Olemassa olevia työkaluja ratkaisussa ei muokata lainkaan, joka on selkeä etu Trinityyn verrattuna. Toisaalta vapaamuotoisempien työkalujen edut jäävät saavuttamatta, joka on vastoin Trinityn perusvaatimuksia. Mallimuunnoksista johtuen se ei myöskään ilmeisesti tue reaaliaikaista kollaboratiivista työskentelyä.

Joustaviin mallinnustyökaluihin läheinen tutkimuskohde ovat niin sanotut älykkäät toimistotyökalut (smart office tools), jossa toimistotyökalujen uudelleenkäyttö- ja konsistenttiusongelmia pyritään ratkaisemaan erillisellä sisällön rakenteella (content model). Desmond et al. (2010) esittelevät esivaatimusanalysoinnin tueksi tarkoitetun BIT-Kit-mallinnustyökalun, joka tunnistaa ja kategorisoi käyttäjän vapaasti piirrettyjä visuaalisia kaavioelementtejä ja mahdollistaa niiden liittämisen myöhemmin haluttuihin kohdealueen käsitteisiin. Näitä käsitteitä voidaan uudelleenkäyttää erityyppisissä kaavioissa ja työkalu huolehtii sisällön eheyden hallinnasta sekä tarjoaa kohdealuekohtaista opastusta perinteisten mallinnustyökalujen tapaan. Vaikka ratkaisu rajoittuu yhteen toimistotyökaluun, on se hyvä askel eteenpäin varhaisen työkalutuen edistämisen kannalta ja kaventaa perinteisten mallinnustyökalujen ja toimistotyökalujen välistä kuilua.

Myös Kern & Kühne (2009) hyödyntävät Microsoft Visiota mallinnustyökaluna integroimalla sen Eclipse Modeling Frameworkin (*EMF*) kanssa. Tekijät tunnistavat Visiosta kolme metatasoa; mallit, *Stencilit* (metamalli) ja *Stencilien* käsitteellisen rakenteen (metametamalli), jotka linkitetään *EMF*:n vastaaviin tasoihin. Ratkaisu hyödyntää linkityksessä Vision *XML*-tallennustoimintoa ja *EMF*:n *XMI*-tiedostoja, joiden avulla mallit muunnetaan työkalujen välillä. Linkin tavoitteena on hyödyntää Visiota graafisena kaaviotyökaluna *EMF*:ää näiden kaavioiden mallien prosessointiin, kuten validointiin tai koodigenerointiin. Muunnos on mallitasolla kaksisuuntainen ja tukee myös Vision näkymätiedon, kuten elementin sijainnin ja koon, tallentamista osaksi *EMF*-mallia. Trini-

tyn ratkaisuista poiketen toteutus ei kuitenkaan jälleen tue reaaliaikaista kollaboratiivista työskentelyä tiedostopohjaisista mallimuunnoksistaan johtuen.

9. YHTEENVETO

Tässä diplomityössä suunniteltiin ja toteutettiin Microsoft Visioon COM Add-in -laajennuskomponentti, jolla Visiosta saadaan Trinity-työkaluympäristössä toimiva joustava tietokantapohjainen mallinnustyökalu. Joustavuudella tarkoitetaan sitä, että se pyrkii yhdistämään vapaamuotoisten työkalujen ja perinteisten mallinnustyökalujen parhaita ominaisuuksia. Ratkaisu sallii normaalin mallintamisen lisäksi vapaamuotoisten elementtien lisäämisen osaksi mallia eikä pakota käyttäjää noudattamaan tiukkoja sääntöjä mallinnustyökalujen tavoin. Näin se tukee luonteeltaan luonnostelevaa työtä ja edistää mallinnuksen varhaisempaa työkalutukea. Ympäristön mallit sijaitsevat erillisessä tietokannassa, jonka tilan laajennuskomponentti pitää synkronoituna Visio-näkymän kanssa ja mahdollistaa mallien muokkaamisen reaaliaikaisesti. Se tukee lisäksi useita samanaikaisia käyttäjiä samaan näkymään ja käytännössä mielivaltaista joukkoa mallinnuskielemiä. Se tarjoaa myös yksinkertaiset menetelmät uusien kielten lisäämiseen ja perustoinnallisuuden laajentamiseen erilaisilla asemointi- ja automaatiokomponenteilla.

Laajennuskomponentin toteutuksesta saatiin geneerinen ja kohtalaisen yksinkertainen hyödyntäen tunnettuja suunnittelumalleja ja ympäristön tarjoamia valmiita palveluita mahdollisimman paljon. Geneerisyyden vastapainona suorituskyky ei ole tällä hetkellä halutulla tasolla, ja osa toteutuksesta on monimutkaistunut hieman liikaa. Suorituskyky on kuitenkin riippuvainen muusta ympäristöstä, etenkin tietokantayhteydestä, ja sitä on arvioitava kokonaisuutena. Tulevaisuudessa komponentin sisäisiin rakenteisiin ja tietokantakomponenttiin liittyviä riippuvuussuhteita tulisi arvioida tarkemmin, mikäli laajennuskomponentin toiminnallisuutta halutaan esimerkiksi eriyttää muihin tarkoituksiin. Nykyisiin vaatimuksiin nähden laajennuskomponentti on kuitenkin toimiva ja tarjoaa hyvän pohjan sen jatkokehitykselle Trinity-projektin puitteissa.

Työssä tarkasteltiin myös muita aihepiirin tutkimuksia, joissa vastaavaa Vision tai muun olemassa olevan toimistotyökalun laajentamista mallinnustyökaluksi samassa mittakaavassa ei tullut vastaan. Etenkin reaaliaikaisen samanaikaisen työskentelyn tuki on melko ainutlaatuinen, sillä muut tutkitut ympäristöt perustavat mallien jakamisen pääasiassa mallimuunnoksiin. Perinteisten mallinnustyökalujen ja toimistotyökalujen välisen kuilun kaventaminen tuntuu kuitenkin olevan melko ajankohtainen aihe, joka osaltaan nostaa tämän työn merkitystä.

LÄHTEET

Amelunxen C., Klar F., Königs A., Rötschke T., Schürr A., Metamodel-based Tool Integration with MOFLON. 2008. Proceedings of the 30th international conference on Software engineering, Leipzig, Germany, May 10 - 18, 2008. New York, NY, USA 2008, ACM Press. pp. 807-810.

Beck, K., Crocker, R., Meszaros, G., Vlissides, J., Coplien, J., Dominic, L., Paulisch, F. 1996. Industrial experience with design patterns. Proceedings of the 18th international conference on Software engineering. May, 1996. Washington, IEEE Computer Society. pp. 103–114.

Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., Pattern-Oriented Software Architecture, Volume 1, A System of Patterns. Chichester, Wiley, 1996, 457 s.

Gamma, E., Helm, R., Johnson R., Vlissides, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 395 s.

Haikala, I., Märijärvi J. 2006. Ohjelmistotuotanto. Yhdestoista painos. Helsinki, Talentum Media Oy. 440 s.

Heiko Kern and Stefan Kuhne. 2009. Integration of Microsoft Visio and Eclipse Modeling Framework Using M3-Level-Based Bridges. Proceedings of the 2nd ECMDA Workshop on Model-Driven Tool & Process Integration at Fifth European Conference on Model-Driven Architecture Foundations and Applications 2009, Enschede, The Netherlands, June 24, 2009. pp. 35-46.

Koskimies K., Koskinen J., Maunumaa M., Peltonen J., Selonen P., Siikarla M., Systä T. (2004). UML työvälineenä ja tutkimuskohteena. Tietojenkäsittelytiede, Issue 21, pp. 19-51.

Mernik M., Heering J., Sloane M. When and How to Develop Domain-Specific Languages ACM Computing Surveys, Vol. 37, No. 4, December 2005, pp. 316–344.

Kelly S., Tolvanen J-P. 2008. Domain-Specific Modeling: Enabling Full Code Generation. Hoboken, New Jersey. Wiley-IEEE Computer Society Press. 448p.

Microsoft Corporation. Visio 2010:n perustoiminnot [WWW]. [viitattu 22.5.2011]. Saatavissa: http://office.microsoft.com/fi-fi/visio-help/visio-2010-n-perustoiminnot-HA101835290.aspx?CTT=5&origin=HA010370238#_Toc254806997.

Microsoft Corporation. Get started using Visio [WWW]. [viitattu 22.5.2011]. Saatavissa: <http://office.microsoft.com/en-us/visio-help/get-started-using-visio-2007-HA010232688.aspx>.

Microsoft Corporation. Understanding Visio shapes and masters [WWW]. [viitattu 22.5.2011]. Saatavissa: <http://office.microsoft.com/en-us/visio-help/understanding-visio-shapes-and-masters-HA001182260.aspx>.

Microsoft Corporation. Overview of Automation [WWW]. [viitattu 22.5.2011]. Saatavissa: <http://msdn.microsoft.com/en-us/library/ms221251.aspx>.

Microsoft Corporation. Welcome to the Microsoft Office Visio 2007 SDK [WWW]. [viitattu 22.5.2011]. Saatavissa: <http://msdn.microsoft.com/en-gb/library/ms409183%28office.12%29.aspx>.

Microsoft Corporation. Overview of Add-ons and COM Add-ins in Visio 2007 [WWW]. [viitattu 22.5.2011]. Saatavissa: <http://msdn.microsoft.com/en-us/library/bb851468%28office.12%29.aspx>.

Microsoft Corporation. Visio 2007 ShapeSheet Reference [WWW]. [viitattu 22.5.2011]. Saatavissa: [http://msdn.microsoft.com/en-gb/library/ms427024\(office.12\).aspx](http://msdn.microsoft.com/en-gb/library/ms427024(office.12).aspx).

Microsoft Corporation. Exploring the Observer Design Pattern [WWW]. [viitattu 22.5.2011]. Saatavissa: <http://msdn.microsoft.com/en-us/library/ee817669.aspx>.

Microsoft Corporation. The C# Language [WWW]. [viitattu 15.5.2011]. Saatavissa: <http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>.

Microsoft Corporation. ADO.NET Datasets WWW]. [viitattu 15.5.2011]. Saatavissa: [http://msdn.microsoft.com/en-us/library/8s3saad7\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/8s3saad7(VS.71).aspx).

Microsoft Corporation. Visio 2010:n kotisivu [WWW]. [viitattu 17.5.2011]. Saatavissa: <http://office.microsoft.com/fi-fi/visio/>.

Object Management Group. Unified Modeling Language [WWW]. [viitattu 15.5.2011]. Saatavissa: <http://www.uml.org/>.

Object Management Group. UML 2.3 Infrastructure Specification [WWW]. [viitattu 15.5.2011]. Saatavissa: <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>.

Object Management Group. UML 2.3 Infrastructure Specification [WWW]. [viitattu 25.4.2011]. Saatavissa: <http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/>.

Object Management Group. OMG's MetaObject Facility (MOF) Home Page [WWW]. [viitattu 25.4.2011]. Saatavissa: <http://www.omg.org/mof/>.

Ossher, H., Bellamy, R., Simmonds, I., Amid, D. Anaby-Tavor, A., Callery, M., Desmond, M., de Vries, J., Fisher, A., Krasikov, S. 2010. Flexible Modeling Tools for Pre-Requirements Analysis: Conceptual Architecture and Research Challenges. OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, Reno/Tahoe, Nevada, USA, October 17-21, 2010. New York, NY, USA, ACM Press.

Peltonen J., Vartiala M., An agent based architecture style for application integration, *Annales Univ. Sci. Budapest., Sect. Comp.*, vol. 31, pp. 3–22, 2009.

PostgreSQL: The world's most advanced open source database [WWW]. [viitattu 21.4.2011]. Saatavissa: <http://www.postgresql.org/>.

The Eclipse Foundation. Eclipse - The Eclipse Foundation open source community website [WWW]. [viitattu 18.5.2011]. Saatavissa: <http://www.eclipse.org/>.

Vartiala, M. 2010. Design and Implementation of an Agent-Based Architecture for a Process Support System. Master's Thesis. Tampere. Tampereen teknillinen yliopisto. 50p.

Volz, M., Jablonski, S. 2010. OMME – A Flexible Modeling Environment. SPLASH 2010 Workshop on Flexible Modeling Tools. Reno, Nevada, USA, October 18, 2010.

Wicks, M.N., Dewar, R.G. A new research agenda for tool integration. *Journal of Systems and Software*, 80(2007)9, pp. 1569-1585.