MetaEdit+:n käyttö kehitysmenetelmän mallintamisessa

Miika Nurminen & Annemari Auvinen
{minurmin,annauvi}@jyu.fi

4.11.2008

Seminaarityössä käsitellään ohjelmistotyön tukivälineitä kehitysmenetelmän mallintamisen näkökulmasta. CASE-työvälineistä keskitytään MetaEdit+-ohjelmistoon. Työssä käsitellään menetelmäkehityksen historiaa ja oletettuja hyötyjä, verrataan menetelmäkehitystä mallipohjaiseen kehitykseen ja menetelmän kuvausmahdollisuuksia (Meta)CASE-työvälineillä. Käytännön esimerkkinä käydään läpi MetaEdit+:ssa olevan valmiin kehitysmenetelmän sovittamista uudelle sovellusalueelle.

1 Johdanto

Ohjelmistotekniikan ja tietojärjestelmien kehittämisen alkuvaiheista alkaen ohjelmistotyön tuottavuutta on pyritty parantamaan nostamalla tiedon abstraktiotasoa. Ohjelmointikielten tasolla konekielen ja assemblerin syrjäyttivät ensin proseduraaliset kielet, jotka sittemmin ovat jääneet useimmilla sovellusalueilla funktio- ja erityisesti oliokielten varjoon. Järjestelmien monimutkaisuutta on yritetty hallita myös erilaisilla kehitysmenetelmillä, jotka tarjoavat projektille standardikäytäntöjä, yhtenäisen notaation ja dokumentaatiotavan, sekä välineitä projektin hallintaan. Sekä ohjelmistoprosessin eri vaiheita että varsinaisia kehitysmenetelmiä on myös pyritty tukemaan erilaisin ohjelmistoin. Ohjelmistotyön tukivälineet (CASE, Computer Aided Software Engineering), menetelmäkehityksen tukivälineet (CAME, Computer Aided Method Engineering) ja viimeisimpänä metamallipohjaiset työkalut (MetaCASE).

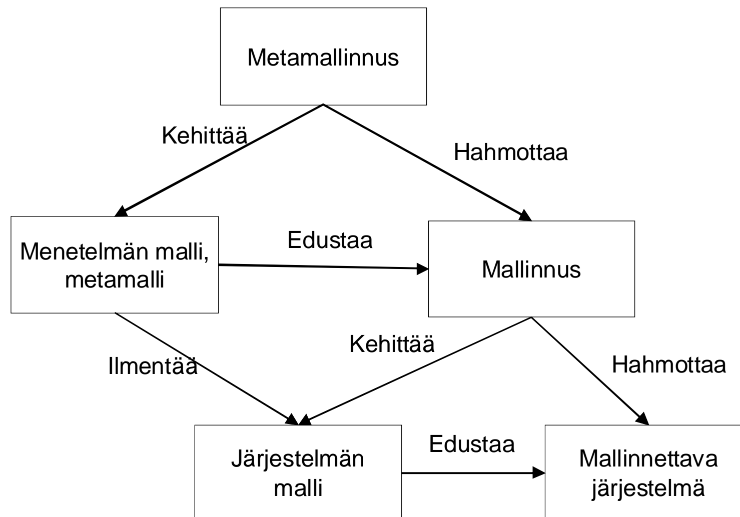
Seminaarityössä käsitellään ohjelmistotyön tukivälineitä kehitysmenetelmän mallintamisen näkökulmasta. Luvussa 2 käsitellään kehitysmenetelmiä ja niiden tukea eri CASE-työvälineillä yleisesti, luvussa 3 keskitytään menetelmäkehitykseen. Luvussa 4 käydään läpi esimerkki menetelmän sovittamisesta MetaEdit+-ympäristön avulla. Luku 5 on yhteenveto.

2 Tietojärjestelmien kehitysmenetelmät ja CASE-työvälineet

Avison & Fitzgerald (2006) määrittelevät CASE-työvälineen olevan *mikä tahansa integroitu tietokonejärjestelmä, joka on erityisesti suunniteltu tukemaan merkittävää osaa tietojärjestelmän kehitysprosessista ja siihen liittyvien tehtävien ja prosessien hallinnasta*. Eritasoisia CASE-työvälineitä on ollut olemassa 1970-luvulta alkaen ja niiden on toivottu parantavan erityisesti ohjelmistotyön tuottavuutta samaan tapaan kuin CAD (*Computer Aided Design*) -ohjelmistot ovat tehostaneet laitteisto- ja muuta tuotantosunnittelua. 1970- ja 80-lukujen työkalujen yleisenä ongelmana oli, että ne oli suunniteltu vain tiettyä suunnittelumenetelmää silmälläpitäen, jolloin tietty CASE-ympäristö hankittaessa kehitysorganisaation oli sopeuduttava sen käyttämään malliin. CASE-ympäristöjen joustamattomuus ja yhteensopimattomuus yhdistettynä suureen määrään tarjolla olevia menetelmiä eivät pääsääntöisesti johtaneet toivottuihin odotuksiin tuottavuuden parantumisesta (Kelly & Tolvanen 2008).

1990-luvun lopulla UML-kieli saavutti laajan suosion ja on siitä asti ollut käytössä useissa eri suunnittelu- ja kehitysympäristöissä sekä erityisesti oliosuuntautuneiden menetelmien osana.

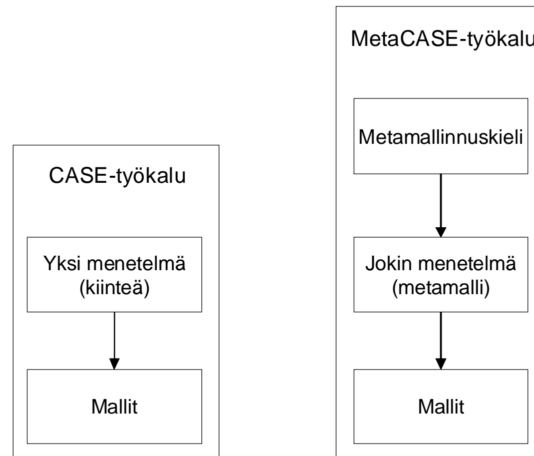
Teknisenä ongelmana eri UML-työkalut kommunikoivat kuitenkin keskenään huonosti, koska eri valmistajat tulkitsevat standardiksi tarkoitettua XMI-siirtoformaattia eri tavoin ja käyttävät omia epästandardeja laajennuksiaan (Lundell *et al.* 2006). UML:ää on myös kritisoitu, koska sen geneerisyys vaatii ylimääräisen tulkinnallisen kerroksen mallinnettaessa sovellusalueen käsitteitä verrattuna sovellusaluespesifisiin kieliin (Kelly & Tolvanen 2008).



Kuva 1. Metamallinnus ja mallinnus (Tolvanen 1998)

Toinen 1990-luvulla kehittynyt ja 2000-luvulla vähitellen yleistynyt tekniikka ovat MetaCASE-työkalut, joissa ideana on sisällyttää samaan työkaluun sekä menetelmän mukaisten kuvausten editointi, että muokattava metamalli, joka kuvaa malleissa käytettävän menetelmän (=notaation). Tämä lisää ohjelmiston joustavuutta oleellisesti verrattuna yhteen malliin sidottuihin CASE-työkaluihin ja periaatteessa korvaa ne, kunhan MetaCASE-työkalun metamallinnuskieli on tarpeeksi ilmaisuvoimainen menetelmän kuvaukseen (käytännössä ongelmia voi tulla esim. olemassa olevien mallien siirrettävyyden tai käyttöliittymäongelmien takia). Kuva 1 havainnollistaa metamallinnuksen suhdetta mallinnukseen ja kehitettävään tietojärjestelmään, Kuva 2 esittää CASE- ja MetaCASE-työvälineiden periaatteellista eroa. Varhaisissa MetaCASE-työkaluissa metamallit kuvattiin usein tekstinä ja piti mahdollisesti kääntää, ennen kuin malleja voitiin käyttää, mikä hidasti menetelmien sovittamista (Kelly & Tolvanen 2008). MetaEdit (Smolander *et al.* 1991) oli ensimmäinen graafisen metamallinnusympäristön sisältävä MetaCASE-työväline. Huomattavaa on, että yleisten menetelmäkehityksen työvälineiden (CAME) ei välttämättä tarvitse olla MetaCASE-tyylisiä (Kelly 1997), mutta kaikki MetaCASE-työkalut sisältävät CAME-ominaisuuksia (Marttiin 1998).

Metamallinnusta voidaan soveltaa varsinaisten tietojärjestelmien kehitysmenetelmien alueen lisäksi periaatteessa minkä tahansa muunkin sovellusalueen kehitysmenetelmään edellyttäen, että keskeiset käsitteet ovat abstrahoitavissa. Menetelmätasolla esimerkkejä tällaisista ovat liiketoimintaprosessien mallintaminen (Marttiin 1998) ja asiakirjojen rakenteistaminen (Lehtinen 1998). Sovellukset ovat vielä monipuolisemmat tarkasteltaessa tietojärjestelmien mallinnusta: sovellusaluepohjaisessa mallinnuksessa (*DSM, Domain Specific Modeling*) ei välttämättä sitouduta mihinkään perinteiseen kehitysmenetelmään, vaan mallinnetaan sovellusalue alusta alkaen omalla notaatiolla. Yhdistämällä sovellusaluemalli erityisasiantuntijan koodaamaan raporttigeneraattoriin ja sovellusaluekehykseen voidaan saavuttaa merkittäviä tuottavuushyötyjä, koska sovelluksen muu koodi voidaan generoida automaattisesti sovellusaluemallista raporttigeneraattorin avulla. Sovellusaluekehys vastaa integroinnista ajoympäristöön ja perinteiseen tapaan toteutettuihin uudelleenkäytettäviin komponentteihin (Kelly & Tolvanen 2008).



Kuva 2. CASE- ja MetaCASE-työvälineiden peruserot (Tolvanen 2000).

Sovellusaluepohjaista mallinnusta voidaan pitää yhtenä lähestymistapana mallipohjaiselle kehitykselle (*MDE, Model Driven Engineering*) (Saraiva & de Silva 2008). MDE:ssä *malleja* pidetään ohjelmistokehitysprosessin perustana vastakohtana koodille tai dokumentaatiolle. Toinen merkittävä lähestymistapa MDE:hen on MDA (*Model Driven Architecture*) (Kleppe *et al.* 2003), joka perustuu UML:n sovellusaluekohtaiseen sovittamiseen profiilien avulla ja mallien muunnoksiin. Huomattavaa on, että MetaCASE-työkalu soveltuu yhtä lailla järjestelmäkehitykseen valitulla (tai mahdollisesti muokatulla) kehitysmenetelmällä kuin sovellusaluemallinnukseen, koska sekä sovellusalueen käsitteet että menetelmän mallinnuselementit voidaan kuvata MetaCASE:n metamallissa. Periaatteessa myös MDA voitaisiin sisällyttää MetaCASE-ympäristöön, joskin UML:n metamalli lisättynä MDA-muunnossäännöillä ja kyselykielellä on huomattavasti tyypillistä sovellusaluekohtaista kieltä monimutkaisempi.

Vaihtoehtona metamallinnukselle kehitysmenetelmiä voidaan mallintaa myös ontologian avulla (Leppänen 2005). Ontologioiden formaalisuusaste vaihtelee asiasanastoista loogisiin teorioihin, mutta usein käytetyn (tietojenkäsittelytieteissä – vastakohtana filosofian laajemmalle käsitteelle) määritelmän mukaan ontologia on formaali, eksplisiittinen määrittely yhteisestä käsitteistöstä (*shared conceptualization*) tietämyksen kuvaamiseen (Gruber 1993). Tietämyksen kuvaaminen ontologioita käyttäen mahdollistaa mm. koneellisen päättelyn ja mallien yhtenäisyyden automaattisen tarkastelun. Ontologioilla ja metamalleilla on selviä yhtäläisyyksiä tavoitteidensa osalta (ja tietyn sovellusalueen metamallia voisi pitää yhtä lailla sovellusaluespesifisenä ontologiana), mutta erilaisista tutkimusperinteistä johtuen menetelmien terminologia ja käsittelytapa poikkeavat toisistaan ja mallintamistapojen täsmällinen suhde on epäselvä. Leppäsen (2005) mukaan metamallinnus luo, laajentaa, muokkaa ja integroi ”mallien malleja”, jotka kuvaavat tai määräävät tiettyä sovellusaluetta. Ontologiat toimivat yhtenäistävinä viitekehyksinä eri näkymiin. Molemmat menetelmät soveltavat kieliä (metamallit, ontologiat) artefaktien esittämiseen. Kielet spesifioidaan metadatamalleilla metamallinnuksessa ja metaontologioilla ontologiakehityksessä. Molemmat menetelmät sallivat vaihtelua artefaktien formaalisuudessa.

3 Menetelmäkehitys

Valittaessa tietojärjestelmän kehitysmenetelmää organisaation käyttöön jokin seuraavista strategioista on mahdollinen (Tolvanen 1998): käsikirjalähestymistapa (*text book approach),* tapahtumapohjainen (*contingency approach)* tai menetelmäkehitys (*method engineering*):

- Käsikirjalähestymistavassa tietty menetelmä otetaan sellaisenaan käyttöön – tyypillistä, jos menetelmä on aiemmin tuntematon organisaatiossa tai kehittäjien menetelmätietämys on vähäistä
- Kontingenssilähestymistavassa (*contingency approach*) menetelmän osat valitaan suuremmasta tekniikoiden joukosta niiden ominaisuuksien (esim. tekniset, organisationaaliset, inhimilliset) perusteella (Tolvanen 2000).
- Menetelmäkehityksessä (*method engineering)* valinta tehdään yhdistämällä olemassa olevia menetelmien osia komponenttityylisesti ja määrittelemällä uusia komponentteja. Brinkkemper (1996) määrittelee menetelmäkehityksen olevan *tiede menetelmien, tekniikoiden ja työkalujen suunnitteluun, kehitykseen ja sovittamiseen tietojärjestelmien kehityksen tueksi.*

Menetelmäkehitys on havaittu tarpeelliseksi, koska geneeriset tai edes sovellusaluekohtaiset menetelmät on havaittu todellisissa kehitysprojekteissa liian joustamattomiksi. Useiden tutkimusten perusteella yli puolet organisaatioiden käyttöönottamista menetelmistä on sovitettuja tai itse kehitettyjä (Tolvanen 2000). Lisäksi useissa organisaatioissa menetelmiä ei välttämättä sovelleta käytännössä kirjattujen ohjeiden mukaisesti, jos kehittäjät kokevat menetelmän liian raskaaksi tai menetelmä ei muuten sovi organisaatiokulttuuriin.

3.1 Näkökulmia menetelmiin

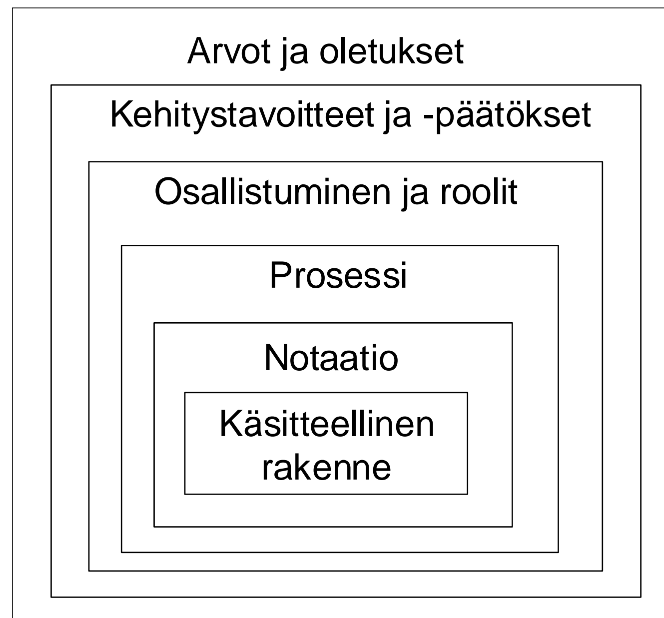
Brinkkemperin (1996) määritelmän[1] mukaan tietojärjestelmän kehitysmenetelmä on *tieteelliseen ajatteluun perustuva lähestymistapa järjestelmäkehitysprojektiin, koostuen säännöistä ja ohjeista, jotka on rakenteistettu järjestelmällisesti osaksi kehitysaktiviteetteja vastaamaan kehitettäviä tuotteita*. Menetelmät ovat siis monitahoisia entiteettejä, joita voidaan jaotella ja tarkastella useista eri näkökulmista. Menetelmiä on kehitetty eri vaiheisiin ohjelmistokehitysprosessia, eri sovellusalueille, erilaisten lähestymistapojen ja paradigmojen tueksi, sekä erilaisia organisationaalisia rakenteita silmälläpitäen (Tolvanen 2000). Menetelmien monimuotoisuuden huomioon ottaen on selvää, että kaikkia menetelmän osa-alueita ei voida formalisoida tai ylipäänsä tehostaa tietokoneavusteisesti. Arvioitaessa tekijöitä, joita CASE- ja CAME-työkaluilla voidaan tukea tarkastellaan Tolvasen (1998) jaottelua menetelmätietämyksen eri tyypeistä (ks. Kuva 3):

- **Käsitteellinen rakenne**. Määrittelee avainkäsitteet, jotka ovat usein sovellusaluespesifisiä. Joissakin menetelmissä (esim. UML-kieltä käyttävät menetelmät) käsitteellinen rakenne on formalisoitu, useimmissa ei. Metamallintamisella mallinnetaan pääosin käsitteellistä rakennetta.
- **Notaatio**. Käsitteiden esitystapa. Samoja käsitteitä voi esittää useilla tavoilla, esim. graafit vs. matriisit. Esitystavat voivat olla formaaleja (esim. loogiset säännöt), puoliformaaleja (rakenteiset esitystavat) tai vapaamuotoisia.
- **Prosessi**. Ohjeet kehitysprosessiin liittyvien suunnittelu- ja johtamisprosessien läpiviemiseksi.
- **Osallistuminen ja roolit**. Menetelmän tulisi tunnistaa kehitysprojektiin liittyvät roolit ja vastuut (käyttäjä, projektipäällikkö, toteuttaja, omistaja jne).

---

[1] Muita vaihtoehtoisia määritelmiä on runsaasti, esim. Tolvanen (1998), Leppänen (2005), Avison & Fitzgerald (2006). Brinkkemperin määritelmä valittiin yksinkertaisuuden vuoksi ja siksi, että määritelmä toimii samassa viitekehyksessä edellä esitetyn menetelmäkehityksen määritelmän kanssa.

- **Kehitystavoitteet ja päätökset**. Kehitystavoitteissa kuvataan menetelmän tunnistamat (tekniset ja ei-tekniset) hyvät käytännöt ja kehitysratkaisut.
- **Oletukset ja arvot**. Menetelmän taustalla olevat (usein implisiittiset) oletukset, ”maailmankuva”.



Kuva 3. Menetelmätietämyksen tyypit (Tolvanen 1998).

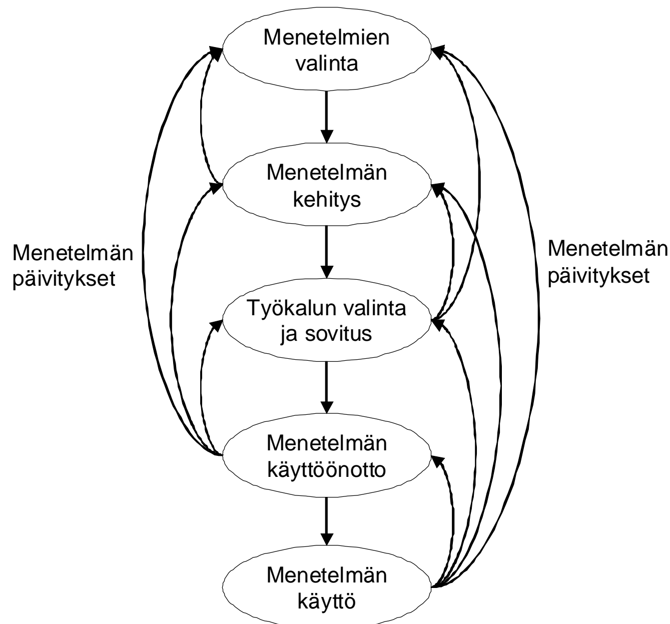
Metamallipohjainen menetelmän kuvaaminen keskittyy pääosin menetelmän käsitteelliseen rakenteeseen ja jonkin verran notaatioon ja prosessiin, mutta muut menetelmätietämyksen tyypit jäävät useimmiten metamallintamisen ulkopuolelle (Tolvanen 2000). Siksi MetaCASE-työkalujen osalta on osittain harhaanjohtavaa puhua täysiverisestä menetelmäkehitysympäristöstä, koska erityisesti menetelmän ei-teknisiä piirteitä ei yleensä huomioida. Täsmällisempää olisi puhua esim. mallinnuskielen tai notaation kehitysympäristöstä kuin varsinaisesta menetelmäkehityksestä. Esimerkiksi MetaEdit+:n dokumentaatiossa puhutaan UML-”menetelmästä”, vaikka kyseessä on suhteellisen menetelmäriippumaton (joskin pääosin oliolähestymistapaan liittyvä) kuvauskieli, joka ei sinänsä ota kantaa kehitysmenetelmään tai siihen, miten erilaisia UML-kaavioita pitäisi soveltaa kehitysprosessissa. Toisaalta esim. RUP (*Rational Unified Process*) on geneerinen kehitysmenetelmä, joka käyttää UML-kieltä. Käytännön sovelluskehityksessä kuitenkin projektin omaksumilla käytännöillä (esim. ketterien menetelmien asiakaslähtöisyys, testauslähtöinen kehitys, pariohjelmointi) on kuitenkin paljon perustavanlaatuisempi merkitys kuin dokumenteissa käytetyllä notaatiolla.

Metamallinnuksen yleisistä rajoitteista huolimatta menetelmäkuvaukset ovat kuitenkin hyödyllisiä niiden ominaisuuksien osalta, joita niillä pystytään kuvaamaan. Metamallit mahdollistavat ohjelmallisen tuen seuraaville piirteille (Tolvanen 2000):

- **Abstraktio ja mallinnus**. Mallinnuseditorin ja muiden malliin pohjautuvien työkalujen generointi.
- **Yhtenäisyystarkistukset ja ylläpidettävyys**. Mallien oikeellisuus on tarkastettavissa automaattisesti, eri mallinnuselementtejä voidaan linkittää toisiinsa useissa eri näkymissä.
- **Mallien muunnokset**. Koodin generointi, testimateriaali, vienti muihin työkaluihin
- **Tarkastukset**. Dokumentaation generointi, raportit.

3.2 Menetelmän sovittaminen

Edellyttäen, että menetelmää käyttöönotettaessa käsikirja- tai tapahtumapohjainen lähestymistapa ei riitä organisaatiolle, menetelmä täytyy kehittää paikallisesti. Kuva 4 esittää vaiheita, jotka voidaan tunnistaa uutta menetelmää kehitettäessä (Tolvanen 1998).



Kuva 4. Menetelmäkehityksen vaiheet organisaatiossa (Tolvanen 1998).

1. **Menetelmien valinta**. Jokaiseen tietojärjestelmäprojektiin sisältyy päätös projektissa käytettävistä menetelmistä – myös menetelmän (tietoinen tai tiedostamaton) käyttämättä jättäminen on päätös.
2. **Menetelmän kehitys**. Askeleessa 1 valitut menetelmät yhdistetään tai uusia menetelmiä luodaan kehitysprojektin tarpeiden mukaisesti. Tämä käsittää menetelmät osittavien komponenttien ja niiden välisten suhteiden määrityksen.
3. **Työkalun valinta ja sovitus**. Valitaan tai mukautetaan CASE-työväline, joka tukee askeleessa 2 kehitettyjä tai sovitettuja menetelmiä.
4. **Menetelmän käyttöönotto**. Koulutus, pilottiprojektit ym.
5. **Menetelmän käyttö**. Menetelmän ja työkalujen laajamittainen käyttö kehitysprojekteissa.

Käytännössä menetelmäkehityksen prosessi ei etene täysin suoraviivaisesti, vaan menetelmätietämyksen lisääntyessä voidaan palata takaisin aiempiin vaiheisiin ja parantaa menetelmää. Menetelmäkehitys voi olla inkrementaalista, pienin parannuksin etenevää tai radikaalia, ennen kehitysprojektin alkua kerralla määrättyä (Tolvanen 1998). Esimerkkinä tarkemmasta yksittäisestä kehitysprosessista luvussa 4.3 on kuvattu MetaEdit+:n tukema menetelmäkehitysprosessi.

Teknisesti ja käsitteellisesti haastavin vaihe liittyy varsinaiseen kehitysaskeleeseen (2) menetelmien monimuotoisuuden vuoksi. Rolland (2008) tunnistaa edelleen menetelmäkehityksen ongelmakohdiksi menetelmäkomponenttien määrittelyn, niiden haun ja kokoamisen, ja ehdottaa ratkaisuksi standardin rajapinnan kehittämistä, jotta menetelmiä voitaisiin hakea web-sovelluspalveluiden ja SOA:n tyylisesti palveluina. Metamallinnus tukee menetelmäkomponenttien yhdistämistä (Tolvanen 2000) edellyttäen, että eri menetelmät on kuvattu samaa metametamallia

käyttäen. Yleisemmässä tarkastelussa on kuitenkin todennäköistä, että tarvitaan ontologian kaltaista kokonaisvaltaisempaa rakennetta (Leppänen 2005), koska metamallipohjainen lähestymistapa kuvaa menetelmästä vain yhden näkökulman.

4 MetaEdit+

MetaEdit (Smolander *et al.* 1991) ja sen seuraaja MetaEdit+[2] (Kelly *et al.* 1996) ovat SYTI- ja sittemmin MetaPHOR-projekteissa (Lyytinen *et al.* 1994) kehitettyjä MetaCASE-ympäristöjä, jotka on sittemmin kaupallistettu MetaCase-yrityksessä. 1990-luvulla MetaEdit(+):aa markkinoitiin erityisesti menetelmäkehityksen työvälineenä – erilaisia muokattavia kehitysmenetelmiä sisältävänä tietovarastona ja mallinnustyökaluna. 2000-luvulta alkaen ohjelmistotuotelinjojen (Tolvanen & Kelly 2001) myötä painopiste on siirtynyt perinteisestä menetelmäkehityksestä sovellusaluepohjaiseen mallinnukseen, jossa sovelluksen koodi generoidaan automaattisesti sovellusaluemallista raporttigeneraattorin avulla (Kelly & Tolvanen 2008). Tässä luvussa keskitytään kuitenkin MetaEdit+:n käyttöön perinteisemmässä mielessä – monipuolisena menetelmäkantana. Luku perustuu pääosin MetaEdit 3.0:n testaukseen ja dokumentaatioon.

MetaEdit+-ympäristö koostuu kahdesta osasta: MetaEdit+ on CASE-työkalu ja sisältää joukon menetelmiä, joiden pohjalta voidaan myös generoida koodia tai muita raportteja. MetaEdit+ Method Workbenchin avulla menetelmiä voidaan muokata tai luoda omia, jotta ne täyttävät organisaation tarpeet. MetaEdit+-ympäristössä voidaan siis luoda oma menetelmä ja käyttää sitä suunnittelussa.

MetaEdit+-ympäristö tukee useita samanaikaisia käyttäijä. MetaEdit+-sovelluksessa voidaan työskennellä useassa projektissa samanaikaisesti ja dataa voidaan jakaa ja uudelleenkäyttää näiden projektien välillä. Projektit voivat perustua samaan tai eri menetelmiin Sama data voidaan esittää graafisena kaaviona, matriisina tai taulukkona. MetaEdit+ on myös alustariippumaton.

MetaEdit+-sovelluksen ydin on objektikuvauskanta, joka usean käyttäjän tapauksessa sijaitsee palvelimella. Kanta sisältää kaiken informaation olemassa olevista metodeista ja malleista. Monikäyttäjäympäristössä käyttäjän suorittamat muutokset ovat muiden käyttäjien saavutettavissa. Eri käyttäjien samanaikaiset muutokset samaan dataan on estetty lukoilla.

4.1 Perustoiminnot

MetaEdit+ sisältää seuraavat työkalut.

1. Information management -työkalut:

- *Launcher*- työkaluilla hallitaan ympäristön työkaluja.
- *Repository browsers* ja *Graph Manager*, joilla voidaan selata objektikuvauskantaan tallennettua tietoa.
- *Property dialogs*, joilla voidaan katsella ja muokata yksittäisen suunnitteluelementin tietoja.
- *Info Tools* tarjoaa tietoa tietyn suunnitteluelementin käytöstä eri projekteissa, graafeissa ja graafien kuvauksissa.

[2] http://www.metacase.com/MetaEdit.html

2. **Editorit** ovat menetelmäspesifisen informaation katselemiseen, muokkaamiseen ja hallinnoimiseen tarkoitettuja työkaluja. Niillä voidaan katsella tai muokata olemassa olevaa kuvausta tai luoda uusi kuvaus, tuoda olemassa olevia graafielementtejä kuvaukseen, tehdä raportteja, lisätä uusia objekteja tai tuoda niitä muista graafeista, muokata objekteja, lisätä ja siirtää suhteita, lisätä ja muokata rooleja, määritellä hajotelmia (*decompositions*) ja tarkentavia linkkejä graafien välille (*explosions*). Yhdessä editorissa tehdyt muutokset tallennetaan kuvauskantaan ja muutokset näkyvät sitä kautta muissakin editoreissa.

- *Diagram Editor* -työkalulla graafeja luodaan, hallitaan ja ylläpidetään kaaviomuodossa.
- *Matrix Editor* -työkalulla käsitellään graafeja esittämällä ne matriisimuodossa.
- *Table Editor* -työkalulla esitetään graafin elementit tabulaattori- tai lomakemuotoisessa muodossa. Graafin elementit esitetään riveinä ja elementin ominaisuudet sarakkeina.

3. **Output-työkalut** sisältävät raportointi- ja koodingenerointityökalut**.**

- MetaEdit+ sisältää valmiita raportteja.
- *Report Browser* -työkalulla voidaan muokata raportteja tai luoda omia raportteja. Työkalulla voidaan määritellä, mitkä osat suunnitteludatasta halutaan tulostaa, missä formaatissa ja mihin kohteeseen. Tällä työkalulla voidaan tehdä raportteja, dokumenttien ja koodin generointia. Omien raporttien tekemiseen käytetään Report Definition -kieltä.

4. **Method development -työkalut**, jotka muodostavat ympäristön metaCASE-osan.

Työtilainstanssit ovat yksityisiä ja yksittäiset käyttäjät voivat konfiguroida niitä. Käytetty menetelmä määrittelee ne työkalut, jotka ovat kulloinkin käytössä. Yksittäinen käyttäjä voi osallistua useaan projektiin, joten käyttäjän sallitaan työskentelevän useassa työtilainstanssissa ja siirtyvän niiden välillä.

MetaEdit+ sisältää suuren määrän ennalta määriteltyjä kehitysmenetelmiä. Nämä mallit sisältävät tuen neljään mallinnusalueeseen: liiketoimintamallinnukseen, järjestelmien arkkitehtuurien suunnitteluun ja hallintaan, rakenteisiin menetelmiin ja oliopohjaisiin menetelmiin. Kehitysmenetelmiä voidaan käyttää jopa samaan aikaan ja dataa voidaan linkittää ja käyttää eri menetelmien välillä. Myös olemassa olevien menetelmien muokkaus on helppoa, koska MetaEdit+ perustuu metamalleihin.

MetaEdit+-sovelluksella voidaan tuottaa menetelmäriippumattomia raportteja, kuten objektilistoja graafiin kuuluvista objekteista tai ominaisuuslistoja objektien ominaisuuksista. Graafien dokumentaatio voidaan generoida HTML- tai Word-muodossa. Koodigenerointia varten on sovelluksessa ennalta määriteltynä raportit C++-, Smalltalk-, CORBA IDL, Java-, Object Pascal (Delphi) ja SQL-kielille.

4.2 GOPRR-metametamalli

MetaEdit+ käyttää GOPRR (Graph-Object-Property-Relationship-Role) –datamallia[3]. GOPRR-metatyypit voidaan kuvata seuraavasti:

[3] MetaEdit+:n 4-versiossa metamalliin on lisätty myös portit, mutta koska seminaarityössä käytettiin ohjelman aiempaa versiota, porttien käsittely sivuutetaan.

- **Graafi** (*graph*) on joukko objekteja, suhteita, rooleja ja näiden liitoksia (*bindings)*. Graafi osoittaa, mitkä objektit suhde yhdistää minkäkin roolin kautta. Esimerkiksi graafeja ovat *Data Flow Diagram* ja *Object Diagram.*
- **Objekti** (*object*) on asia, joka on olemassa riippumatta suhteista tai rooleista, esimerkiksi *Class*. Kaikki objektien ilmentymät tukevat uudelleenkäytettävyyttä.
- **Ominaisuus** (*property*) on kuvaava tai määrittelevä ominaisuus, esim. *nimi.*
- **Suhde** (*relationship*) on yhteys objektiryhmien välillä, esim. *Data Flow*. Suhde liitetään objekteihin roolien kautta
- **Rooli** (*role*) määrittelee, kuinka objektit osallistuvat suhteessa, esim. rooleja ovat *Flows from* ja *Flows to.*

GOPRR-metatyyppejä käytetään sekä tyyppi- että ilmentymätasolla. Graafin tyyppi voi siis olla *Data Flow Diagram*, minkä esiintymä olisi tietty Data Flow Diagram, esim. Myyntijärjestelmä. Graafityypit sisältävät objektityyppejä, kun taas graafit sisältävät objekteja.
GOPRR-metamalli mahdollistaa:

- Rekursiiviset rakenteet,
- Mallinnettavien käsitteiden yleistämisen ja erikoistumisen,
- Polymorfiset mallinnuskonseptit,
- Erilaiset kuvaukset samasta käsitteellisestä kuvauksesta (esim. graafinen, matriisi, teksti),
- Eri kuvauksien väliset yhteydet ja
- Säännöt mallin eheyden tarkistamiseksi.

GOPRR-malli on oliopohjainen ja sisältää sekä abstraktin että konkreettisen rakenteen perinnän, polymorfismin, kuormituksen, sekä luokien ja olioiden erottelun. GOPRR on suunniteltu myös uudelleenkäytettäväksi. Sekä tyypit että objektin instanssit, suhde, ominaisuus ja graafi voidaan uudelleenkäyttää eri graafissa tai projektityypissä.

4.3 Menetelmäkehitys MetaEdit+:ssa

MetaEdit+ tarjoaa seuraavat menetelmäkehitystyökalut:

- **Property Tool**, jolla voidaan luoda ominaisuustyyppejä, kuten merkkijonoja tai tekstikenttiä, joita muut menetelmäkomponentit käyttävät.
- **Object Tool**, jolla voidaan määritellä objektityyppejä, kuten *Process*, *State* tai *Class*, jotka ovat menetelmien peruskomponentteja.
- **Relationship Tool**, jolla määritellään suhdekomponentit objektityyppien välille. Esimerkiksi *Flow* on suhdekomponentti *Data Flow* –kuvaajassa.
- **Role Tool**, jolla määritellään komponentit, jotka yhdistävät suhdetyypit ja objektityypit.
- **Graph Tool**, jolla hallitaan menetelmämäärityksiä. Menetelmät koostuvat objekti-, suhde- ja roolityypeistä, jotka on määritelty edellä mainituilla työkaluilla, sekä säännöistä, joilla määritellään, miten näitä tyyppejä voidaan yhdistellä.

- **Symbol Editor** -työkalulla määritellään ja muokataan graafin elementtien (objektien, suhteiden ja roolien) graafista kuvausta. Työkalu on eräänlainen piirtotyökalu, jolla voidaan määritellä symbolille muoto, väri ja tekstikentät.
- **Dialog Editor** -työkalulla muokataan elementtien ominaisuuksien muokkaamiseen käytettyjen dialogien ulkoasua.
- **Report Browser** -työkalulla määritellään eheystarkistukset sekä menetelmän dokumentaation ja koodin generointi.
- **Metamodel Browser** -työkalulla selataan menetelmämäärityksiä ja esitetään niiden sisältämät tyypit ja tyyppien väliset suhteet. Tällä työkalulla menetelmän kehittäjä pääsee käsiksi kaikkiin menetelmämäärityksiin, jotka ovat saatavilla avoimissa projekteissa.
- **Type Manager** -työkalua käytetään viemään menetelmämäärityksiä muihin MetaEdit+-kuvauskantoihin sekä tarpeettomien menetelmämääritysten poistamiseen.

Graafityökalun ja muiden menetelmäkehitystyökalujen välinen ero on siinä, että muut työkalut käsittelevät menetelmän yksittäistä komponenttia, mutta graafityökalu yhdistää nämä yksittäiset komponentit ja muodostaa niistä mallinnusmenetelmän.

MetaEdit+:ssa olevia metodeita voidaan siis muokata ja olemassa olevia metodimäärityksiä voidaan uudelleenkäyttää kehitettäessä uusia. Uuden menetelmän määrittämisprosessi koostuu MetaEdit+:ssa seuraavista vaiheista (Tolvanen 1998):

1. Identifioidaan ja määritellään menetelmän objektityypit ja määritellään niiden ominaisuudet.
2. Identifioidaan menetelmän suhdetyypit ja määritellään niiden ominaisuudet.
3. Määritellään menetelmän roolityypit ja niiden ominaisuudet.
4. Määritellään tarvittavat symbolit objekteille, suhteille ja rooleille.
5. Määritellään graafityypit ja lisätään niihin objekti-, suhde- ja roolityypit.
6. Määritellään graafityypeissä suhteiden liitokset.
7. Määritellään rajoitteet objekteille, jotka osallistuvat suhteisiin tai rooleihin.
8. Määritellään graafityypissä objektityyppien tarkentavat linkit graafien välille sekä hajotelmat
9. Määritellään tarkistukseen, mallien dokumentoimiseen ja koodin generointiin liittyvät raportit.
10. Näitä vaiheita voidaan suorittaa iteratiivisesti ja osittain samanaikasesti. Tyyppien määrityksiä voidaan myös muokata myöhemmin.

Esimerkki UML-kielen muokkauksesta:
Esimerkissä laajennetaan UML:n tilakaaviota laajentamalla State-tyyppiä niin, että se sisältää myös tiedon luokasta, jota tila kuvaa. Samalla tavalla voitaisiin luoda täysin uusi menetelmäkin.

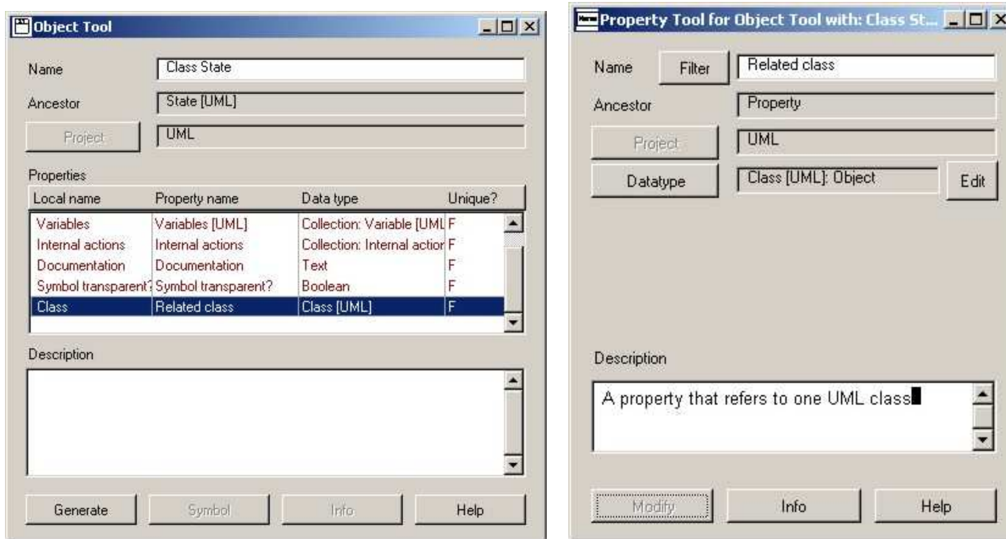
1. Objektityypin määrittely:

Tehdään uusi State-objektityyppi tilakaavioon. Käytetään olemassa olevaa UML:ssä käytössä olevaa State-tyyppiä ja muokataan sitä. Valitaan **Metamodel->Object tool**. **Name**-kenttään

avautuvasta valikosta valitaan **Make Descendant** ja otetaan pohjaksi State [UML] ja määritellään objektille nimi Class State. Ominaisuuksissa näkyy kaikki State [UML] -objektin ominaisuudet.

2. *Lisätään uusi ominaisuus objektille.*

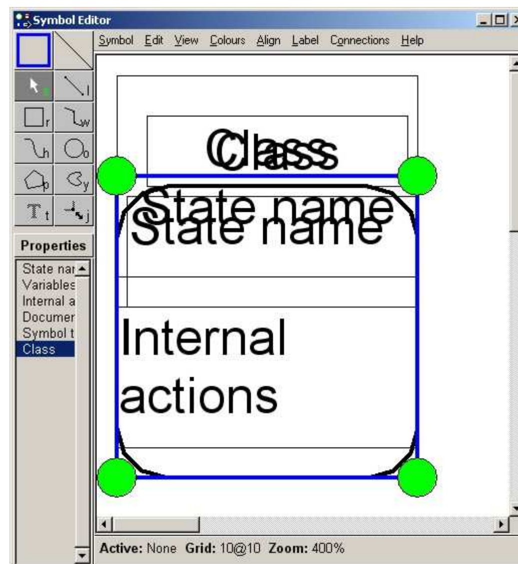
Lisätään ominaisuus, jonka arvo viittaa Class [UML] -objektiin. Ominaisuuslistan päältä aukeavasta valikosta valitaan **Add Property** ja edelleen **New Property Type**. Annetaan ominaisuudelle nimi Related class ja **Datatype**-painikkeen kautta pystytään määrittelemään ominaisuuden datatyypiksi **Object** ja Class [UML]. Kuvauksen määrittämisen jälkeen ominaisuus voidaan luoda **Generate**-painikkeella. Tämän jälkeen ominaisuudelle voi antaa vielä lokaalin nimen, esim. Class. Uusi ominaisuus näkyy objektin ominaisuuslistassa alimpana. Objektityyppi voidaan luoda **Generate**-painikkeella. Objektin ja ominaisuuden luonnissa käytetyt työkalut on esitetty kuvassa 5.



Kuva 5. Object Tool ja Property Tool –työkalut.

3. *Määritellään objektille symboli:*

Object Tool -työkalun **Symbol**-painikkeella päästään määrittelemään symboli (Kuva 6). Vasemmasta reunasta löytyvästä työkalurivistä lisätään halutut kuviot ja siirretään kuvioon alhaalta löytyvät, tarvittavat ominaisuudet Class, State name ja Internal actions. Yhteyspisteet voidaan määritellä valitsemalla kuvio ja **Connections -> Add Points for Selection**.



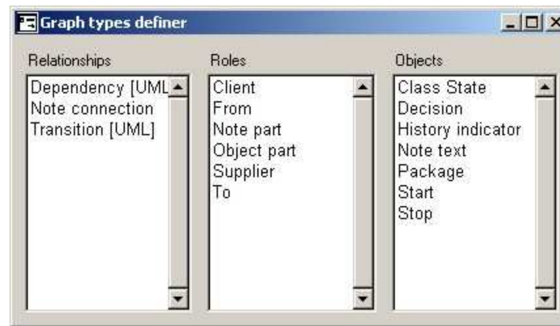
Kuva 6. Symbolin määritys Symbol editor -työkalulla.

4. *Määritellään suhdetyypit ja roolit:*

Suhdetyyppejä pääsee lisäämään, muokkaamaan ja katselemaan valitsemalla **Metamodel -> Relationship Tool** ja rooleja **Metamodel -> Role Tool.** Suhteita tai rooleja ei tässä tapauksessa tarvitse muutella. Aukeava ikkuna on samanlainen kuin objektityypin määrittelyssäkin ja myös suhteille ja rooleille voidaan määritellä symbolit symbolieditorilla.

5. *Määritellään graafityyppi:*

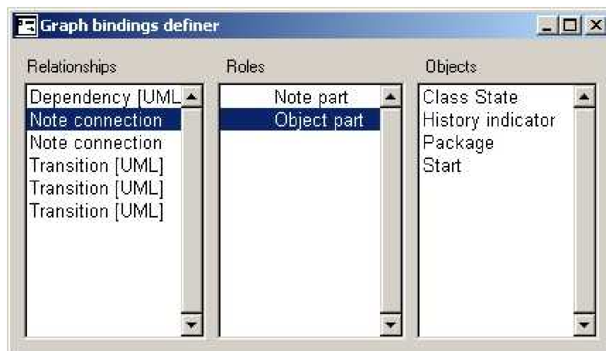
Graafityypin määrittelemiseksi valitaan **Metamodel -> Graph Tool** ja **name**-kohdasta valikosta **Make Descendant.** Valitaan tyypiksi State Diagram [UML]. Annetaan graafille nimi, kuvaus ja luodaan se **Generate**-painikkeella. Tämän jälkeen voidaan **Types**-painikkeella avautuvassa ikkunassa (Kuva 7) määritellä, mitä objekti-, suhde- ja roolityyppejä käytetään metodissa. Lisätään Class State -objekti valitsemalla objektilistan päältä aukeavasta valikosta **Add** ja Class State. Vastaavasti valitaan State [UML] objektilistasta ja aukeavasta valikosta **Delete**. Nyt graafiin on lisätty uusi objekti ja poistettu vanha.



Kuva 7. Graafityypin määrittelyssä määritellään siihen kuuluvat objekti-, suhde- ja roolityypit.

6. *Määritellään liitokset eli yhteydet objektityyppien välillä:*

Jokainen liitos sisältää yhden suhdetyypin, vähintään kaksi roolityyppiä ja yhden tai useamman objektityypin jokaiselle roolille. Liitos siis määrittelee, mitkä objektit voivat olla missäkin roolissa tietyssä suhteessa.

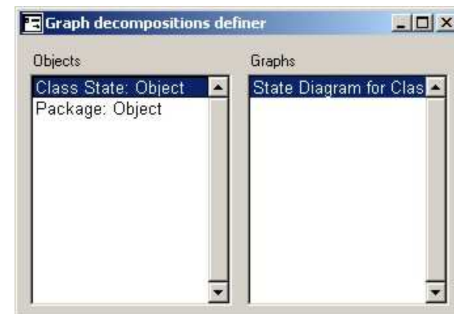


Kuva 8. Liitostyökalulla määritellään objektityyppien väliset yhteydet.

Liitostyökalu (Kuva 8) aukaistaan graafityökalun **Bindings**-painikkeella. Lisätään Note connection –suhteen Object part -roolin objektiksi Class State ja poistetaan sieltä State [UML]. Eli valitaan Note connection ja roolilistasta Object part ja objektilistan päällä aukeavasta valikosta valitaan **Add...** ja Class State. Vastaavasti valitaan State [UML] objektilistasta ja valikosta **Delete**. Nyt State [UML] on korvattu yhdessä liitoksessa uudella Class State -objektilla. Myös muut State [UML]-objektit on korvattava Class State -objektilla jokaisen liitoksen jokaisessa roolissa.

7. *Valitaan mahdolliset objektien, suhteiden ja roolien tarkentavat linkit graafien välille ja mahdolliset objektien hajotelmat.*

Tarkentavat linkit graafien välillä määrittelevät, mihin graafityyppiin tietyn tyyppiset objektit, suhteet ja roolit voidaan linkittää. Hajotelmat määrittelee, mitkä objektityypit voidaan toiminnallisesti osittaa.

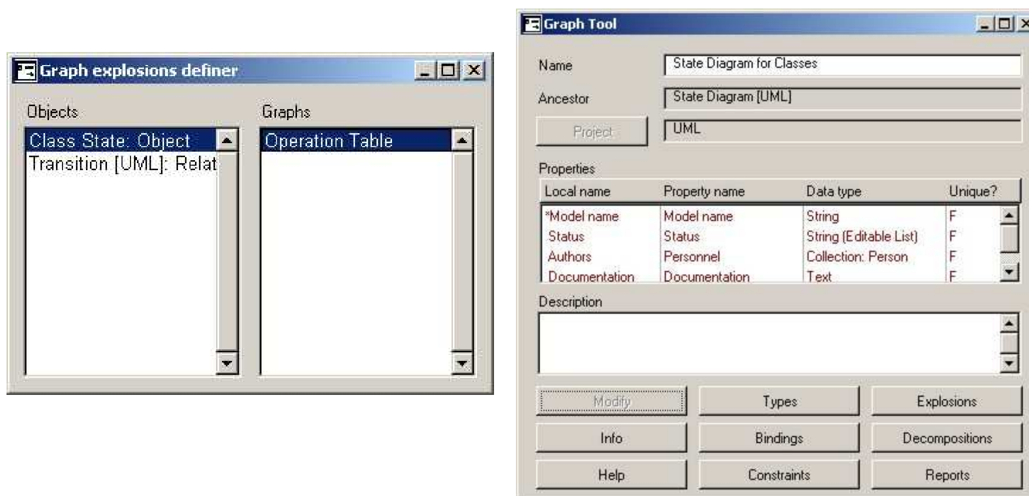


Kuva 9. Hajotelmien määrittelyikkuna.

Decomposition-ikkuna (Kuva 9) saadaan auki graafityökalun **Decompositions**-painikkeella. Lisätään luotu objektilistaan. Valitaan **Add...** objektilistan päällä aukeavasta valikosta, mikä aukaisee listan graafityyppiin

kuuluvia objekteja, suhteita ja roolityyppejä. Valitaan listasta Class State. Lisätään graafit valitsemalla uusi Class State –objekti aktiiviseksi objektilistasta ja graafilistassa valitaan **Add...** ja aukeavasta listasta State Diagram for Classes. Poistetaan State [UML] -objektityyppi valitsemalla se objektilistasta ja valitsemalla **Delete**. Seuraavaksi muutetaan Package-objektin hajotelma osoittamaan uuteen graafiin. Valitaan Package-objektityyppi objektilistasta, **Add...** graafilistan päällä aukeavasta valikosta ja State Diagram for Classes. Valitaan vanha State Diagram [UML] graafilistasta ja poistetaan se valitsemalla **Delete**.

Korvataan kaikki tarkentavat linkit vanhasta State [UML]-objektista uuteen State class -objektiin. Määrittelyyn päästään graafityökalun **Explosions-**painikkeella. Valitaan State [UML] objektilistasta (kuva 6). Poistetaan se valitsemalla **Delete** aukeavasta valikosta.. Lisätään Class State valitsemalla **Add…** aukeavasta valikosta ja sen jälkeen objekti avautuvasta dialogista. Graafilistassa valitaan **Add…** ja avautuvasta ikkunasta Operation Table.



Kuva 10. Vasemmalla tarkentavien linkkien määritysikkuna ja oikealla graafityökalu.

8. *Lopuksi painetaan Modify-painiketta graafityökalussa ja hyväksytään muutokset (*Kuva 10*).*
9. *Menetelmä on valmis käytettäväksi:* Käynnistetään jokin editoreista (Diagram, Matrix tai Table) ja luodaan uusi State Diagram for Classes –graafi.

4.4 MetaEdit+:n arviointia

MetaEdit+:n etuina voidaan pitää sitä, että 4. versiosta alkaen ohjelmistossa on API-rajapinta ja tuki web-sovelluspalveluille, mikä mahdollistaa helpon integroinnin mistä tahansa ympäristöstä. Tässä versiossa on myös suora mallien XML-muotoinen tuonti/vienti. MetaEdit+ sisältää käsitteiden monipuolisen kuvaustavan, kun graafeja voidaan kuvata kaavioina, taulukkoina ja matriiseina. Raportteja voi tulostaa html- ja doc-muodossa. Ryhmätyön tuki on etu samoin kuin se, että omia käsitteitä ja sääntöjä voidaan päivittää myös kehitysaikana.

MetaEdit+:n heikkoutena voidaan pitää käyttöliittymää ja varsinkin ponnahdusvalikoiden toimintaa. Hiiren painikkeet eivät toimi Windows-ympäristössä ohjeissa annetulla tavalla. Ilman avustusta on myös mahdotonta lähteä käyttämään työkalua. Raporttityökalun avulla on tietyin edellytyksen teoriassa mahdollista generoida toimiva sovelluksen koodi, mutta tämä vaatii erityisasiantuntemusta ja on niin työlästä, ettei todennäköisesti kannata, ellei saman sovellusaluemallin pohjalta voida generoida useampia sovelluksia.

Saraiva & de Silva (2008) ovat vertailleet eri MDE-työkaluja. Vertailuun oli otettu mukaan Enterprise Architect, MetaSketch, MetaEdit+ ja Microsoftin DSL-työkalut. Kuvattavana oli sosiaalisen verkon metamalli ja työkaluja arvioitiin mm. tuettujen siirtoformaattien, mallin transformaatiotuen, kieli- ja kirjastometaforien käytön, käyttäjän muokattavissa olevien loogisten tasojen, metamallin syntaksin ja semantiikan määrittelyn tuen sekä meta-metamallien koon perusteella. Julkaisussa todettiin, että vaikka MetaEdit+ perustuu yksinkertaiseen ja joustavaan GOPRR-metametamalliin, niin se ei sisällä käyttäytymiseen liittyviä ominaisuuksia, mikä vaikuttaa niiden metamallien joukkoon, joita työkalulla voidaan määritellä. MetaEdit+ ei myöskään tue mallien transformaatioita, mutta se tarjoaa raportointitavan, jolla voidaan luoda tekstipohjaisia tuotoksia mallin kuvauskannan tiedoista. Vertailluista työkaluista ainoastaan MetaEdit+ ja Enterprise Architect eivät tukeneet metamallimääritysten vientiä.

MetaEdit+:n soveltuvuutta on testattu myös ProLaatuPro-projektissa (Nurminen & Penttinen 2005) IT-tiedekunnan toimintaprosessien kuvaamiseen ja kuvauksiin liittyvien tietojen tallentamiseen. Ohjelman merkittävimpänä etuna todettiin olevan oman metamallin ja sääntöjen määrityksen, mutta toisaalta sekä mallinnuskäyttöliittymässä että raporttien generoinnissa oli loppukäyttäjän kannalta puutteita. Se ei myöskään tukenut mallintamista tietyllä kaaviotyypillä, mikä olisi ollut keskeinen vaatimus prosessityökalulle. Ohjelmiston valmiit raportit eivät myöskään tarjonneet tietoa muodossa, joka olisi mahdollistanut helpon jatkokäsittelyn ilman, että huomattava osa raportin generointiin tarvittavasta koodista olisi pitänyt käytännössä toistaa prosessien julkaisujärjestelmän toteutusta varten. Toisaalta ohjelma oli muilta ominaisuuksiltaan liiankin monipuolinen sovellus projektin hallinnon tarpeita ajatellen.

5 Yhteenveto

Seminaarityössä käsiteltiin ohjelmistotyön tukivälineitä kehitysmenetelmän mallintamisen näkökulmasta. MetaCASE-työvälineitä on mahdollista käyttää menetelmäkehityksen apuna mallintamalla kehitysmenetelmän metamalli MetaCASE-ympäristöön. Tämän jälkeen työvälineellä voidaan tehdä menetelmän mukaisia malleja tietojärjestelmän kehittämiseksi. Ongelmana kuitenkin on, että menetelmästä voidaan CASE-ympäristön puitteissa ottaa käyttöön vain käsitteellinen rakenne ja notaatio – näkymä menetelmään on rajallinen. Menetelmän oikeaa soveltamista siihen kuuluvien käytäntöjen puitteissa, projektinhallintaa tai muita ”inhimillisiä” tekijöitä monipuolisinkaan CASE-työkalu ei pysty korvaamaan, vaikka tietokoneen käsiteltävissä olevan datan käsittelyssä ja rutiinitehtävien automatisoinnissa siitä voi olla hyötyä.

Perinteistä menetelmien metamallinnusta suuremmat tuottavuushyödyt saatetaan saada sovellusaluekeskeisellä mallintamisella, jolloin yleiskäyttöisen menetelmän sijaan mallinnetaan sovellusalue alusta alkaen omalla notaatiolla, mikä helpottaa kommunikointia sovellusalueen asiantuntijoiden kanssa. DSM-lähestymistapa vaatii kuitenkin erityisasiantuntemusta raporttigeneraattorien toteutuksen muodossa, jos sovellusaluemalleista halutaan saada myös ajettavaa koodia aikaan. Jos samaa sovellusalueen metamallia käytetään useammissa sovelluksissa, generaattorien toteutus tulee yleensä perinteisiä kehitysmenetelmiä edullisemmaksi. Kaiken kaikkiaan MetaEdit+ on monipuolinen ja skaalautuva ympäristö erilaisia mallinnustehtäviä varten sekä tietojärjestelmien kehittämistä että muitakin sovelluksia ajatellen, mutta käytettävyydeltään testatussa 3.0-versiossa oli parantamisen varaa. GOPRR-metametamalli on joustava ja ymmärrettävä, mutta ei sisällä käyttäytymiseen liittyviä ominaisuuksia, mikä lisää raporttigeneraattoriin toteutettavan koodin määrää joillakin mallinnuskielillä.

Lähteet

Avison D., Fitzgerald G. 2006. Information Systems Development – Methodologies, technologies & tools, 4th edition, McGraw-Hill.

Brinkkemper S. 1996. Method engineering: engineering of information systems development methods and tools. Information and Software Technology 38(4), 275-280.

Gruber T. 1993. Toward principles for the design of ontologies used for knowledge sharing. Tekninen raportti KSL-93-04, Knowledge Systems Laboratory, Stanford University. http://ksl.stanford.edu/KSL_Abstracts/KSL-93-04.html

Kelly, S. 1997. Towards a Comprehensive MetaCASE and CAME Environment. Väitöskirja, Tietojenkäsittelytieteiden laitos, Jyväskylän yliopisto.

Kelly, S., Lyytinen, K., Rossi, M. 2006. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment, in Proceedings of the 8th International Conference on Advanced Information Systems Engineering, CAiSE'96.

Kelly S., Tolvanen J.-P. 2008. Domain-specific modeling - enabling full code generation, Wiley Interscience.

Kleppe A., Warmer J., Bast W. 2003. MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley.

Lehtinen A. 1998. Tietokoneavusteinen mallinnus rakenteisten asiakirjastandardien kehittämisessä. Pro Gradu, Tietojenkäsittelytieteiden laitos, Jyväskylän yliopisto.

Leppänen M. 2005. An Ontological Framework and a Methodical Skeleton for Method Engineering. Väitöskirja, Tietojenkäsittelytieteiden laitos, Jyväskylän yliopisto.

Lundell B., Lings B., Persson A., Mattsson A. 2006. UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2. 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006).

Lyytinen K., Kerola P., Kaipala J., Kelly S., Lehto J, Liu H., Marttiin P., Oinas-Kukkonen H., Pirhonen J., Rossi M., Smolander K., Tahvanainen V.-P., Tolvanen J.-P. 1994. MetaPHOR: Metamodeling, Principles, Hypertext, Objects and Repositories – Final Report. Tekninen raportti TR-7, Jyväskylän yliopisto, Oulun yliopisto. http://metaphor.it.jyu.fi/loppurap/

Marttiin P. 1998. Customisable Process Modeling Support and Tools for Design Environment. Väitöskirja, Jyväskylän yliopisto.

Nurminen M., Penttinen T. 2005. MetaEdit+ -arviointi. Tekninen raportti (ProLaatuPro-projekti), Informaatioteknologian tiedekunta, Jyväskylän yliopisto. http://prosessit.it.jyu.fi/raportit/MetaEditArvio.pdf

Rolland C. 2008. Method Engineering: Towards Methods as Services. International Conference on Software Process (ICSP 2008).

Saraiva J. de Sousa, da Silva A.R. 2008. Evaluation of MDE tools from a metamodeling perspective, Journal of Database Management 19(4), 21-46.

Smolander, K., Lyytinen, K., Tahvanainen, V., and Marttiin, P. 1991. MetaEdit - A Flexible Graphical Environment for Methodology Modelling. In Advanced Information Systems Engineering, CAiSE'91.

Tolvanen J.-P. 1998. Incremental Method Engineering with Modeling Tools - Theoretical Principles and Empirical Evidence. Väitöskirja, Tietojenkäsittelytieteiden laitos, Jyväskylän yliopisto.

Tolvanen J.-P. 2000. TJTL95 Metamodeling and Method Engineering. Luentomateriaali. Tietojenkäsittelytieteiden laitos, Jyväskylän yliopisto. http://users.jyu.fi/~jpt/ME2000/

Tolvanen, J.-P., Kelly, S. 2001. Modelling Languages for Product Families: A Method Engineering Approach. Proceedings of OOPSLA workshop on Domain-Specific Visual Languages.