KÄÄNTEISTEKNIIKKA OHJELMISTOJEN UUDISTAMISEN APUVÄLINEENÄ

Miia-Maarit Karhunen
Pro gradu -tutkielma
Kuopion yliopisto
Tietojenkäsittelytieteen laitos
27.5.2003

Tiivistelmä

KUOPION YLIOPISTO, informaatioteknologian ja kauppatieteiden tiedekunta
Tietojenkäsittelytieteen koulutusohjelma
Tietojenkäsittelytiede

KARHUNEN, MIIA-MAARIT S.: Käänteistekniikka ohjelmistojen uudistamisen apuvälineenä
Pro gradu -tutkielma, 91 s.
Tutkielman ohjaaja: Professori Anne Eerola, FT
Toukokuu 2003

Avainsanat: käänteistekniikka, takaisinmallinnus, uudistaminen, ohjelmistojen ymmärtäminen

Ohjelmistojen uudistamisella käsitetään lähes kaikki keinot, jotka tähtäävät olemassa olevan ohjelmiston muuttamiseen uuteen muotoon. Ohjelmistojen uudistaminen on tärkeä osa nykyistä ohjelmistotuotantoa, jossa on tarve käyttää hyväksi olemassa olevia järjestelmiä.

Käänteistekniikka eli takaisinmallinnus on menetelmä, jonka avulla analysoidaan olemassa olevaa järjestelmää ja pyritään esittämään se korkeammalla abstraktiotasolla. Käänteistekniikan avulla pystytään selvittämään esimerkiksi ohjelmiston suunnittelun pohjalla olleita vaatimuksia käyttämällä hyväksi olemassa olevaa dokumentaatiota ja ohjelmiston lähdekoodia. Käänteistekniikkaa voidaan luonnehtia takaperinmeneväksi ohjelmistotuotannoksi. Se on tärkeä osa ohjelmistojen uudistamista.

Tutkielmassa on määritelty käänteistekniikan ja uudistamisen tärkeimmät termit. Tämän jälkeen on perehdytty uudistamisen ja käänteistekniikan eri toteuttamistapoihin. Tutkielmassa on esitelty myös käänteistekniikassa apuna käytettäviä välineitä sekä perehdytty UML-kaavioihin ja näiden välisten yhteyksien hyödyntämiseen käänteistekniikassa ja ohjelmien uudistamisessa. Tutkielmassa on käsitelty myös käänteistekniikan käyttökohteita ja myös sen ongelma-alueita.

Tutkielmassa päädytään siihen lopputulokseen, että käänteistekniikkaa voidaan soveltaa useissa kohteissa pitkin ohjelmistotuotantoprosessia. Käyttökohteita ovat laadunvarmistus, kuten testaus ja tarkastus, uudelleendokumentointi, vanhan järjestelmän uudistaminen ja uudelleenkäyttö sekä opetus.

Esipuhe

Tämä tutkielma on tehty PLUG-IT-tutkimusprojektin TEHO-osaprojektissa. Tutkimusprojektin rahoittajina toimivat Tekes sekä ohjelmistoyritykset ja sairaanhoitopiirit.

Haluan kiittää erittäin lämpimästi ohjaajaani FT Anne Eerolaa. Hänen kannustuksensa ja huomionsa olivat erittäin tärkeitä koko tutkielman luomisprosessin ajan. Lisäksi haluan kiittää tarkastajana toiminutta FT Erkki Pesosta hyvistä havainnoista.

Kaunis kiitos tuesta tulevalle aviomielelleni Jarsille.

Kuopiossa 27.5.2003

_____

Miia-Maarit Karhunen

Sisällysluettelo

1 Johdanto

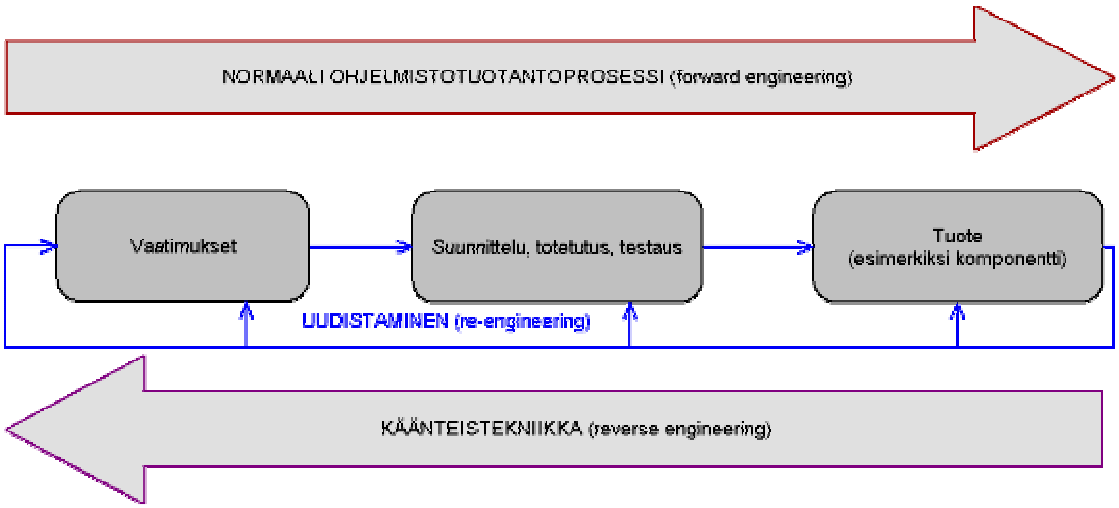
Ohjelmistoteollisuus käyttää noin 70 % kapasiteetistaan vanhojen ohjelmistojen päivitykseen ja vain noin 30 % uusien ohjelmien luomiseen. Eräs syy tähän on se, että ohjelmistoja käyttävät tahot haluavat käyttää vanhoja ohjelmistoja mahdollisimman pitkään. Useat suuret ja kriittiset ohjelmistot ovat peräisin 1950- ja 1960-luvuilta. Näiden korvaaminen kokonaan uusilla ohjelmistoilla on erittäin kallis ja työläs prosessi. Tämän lisäksi kyseiset järjestelmät sisältävät paljon tärkeää toimialalogiikkaa ja tietoa, joka olisi saatava talteen ja hyötykäyttöön, etteivät tärkeät tiedot katoasi ikiajoiksi [MüR98]. Edellä mainittujen syiden vuoksi ohjelmistoteollisuudella on suuri tarve entistä tehokkaammille keinoille uudistaa ohjelmistoja. Samalla myös uudelleenkäytettävyyden merkitys on korostunut. On alettu ymmärtää se tosiasia, että ei ole tehokasta rakentaa ohjelmistoa kokonaan alusta alkaen.

Ylläkuvattujen ongelmien kanssa kamppailevat myös terveydenhuollon tietojärjestelmät. Terveydenhuollon tietojärjestelmiksi kutsutaan sellaisia järjestelmiä, joita käytetään terveydenhuollossa esimerkiksi potilastietojen kirjaamiseen ja ylläpitoon sekä näiden tietojen välitykseen [Myk98]. Terveydenhuollon tietojärjestelmien uudistamis- ja integrointitarpeet ovat suuria senkin vuoksi, että terveydenhuollossa käytetään monia eri tekniikoilla toteutettuja, iältään vaihtelevia sekä useilta eri valmistajilta tulevia tietojärjestelmiä. Tietojärjestelmien lukumäärä etenkin yliopistollisissa sairaaloissa on suuri. Esimerkiksi Kuopion yliopistollisessa sairaalassa on tällä hetkellä käytössä noin 170 eri järjestelmää [KoE02]. Tavalliseen potilastietojen käsittelyynkin tarvitaan kymmeniä erilaisia ja eri-ikäisiä järjestelmiä, jotka ovat lisäksi keskenään osittain tai kokonaan yhteensopimattomia. Tätä uudistamis- ja integrointiongelmaa ja sen ratkaisuvaihtoehtoja onkin pohdittu jo jonkin aikaa Tekesin, yritysten ja sairaanhoitopiirien rahoittamassa PLUG-IT-projektissa.

Ohjelmien uudistamisella (re-engineering) tarkoitetaan vanhojen (legacy), mutta yrityksen kannalta tarpeellisten ohjelmien muokkaamista nykytilannetta vastaavaan, entistä tehokkaampaan muotoon [ChC90]. Tämä voidaan tehdä muuttamalla ohjelma uudemmalle ohjelmointikielelle tai muuten helpommin ylläpidettäväksi käyttäen nykyaikaisempia suunnitteluratkaisuja [Har01]. Ohjelmien uudistamisen tarve voi syntyä esimerkiksi silloin, kun vanhaan ohjelmaan on tehty useita päivityksiä ja päivitykset ovat muuttaneet ohjelman rakenteen vaikeaselkoiseksi. Tämän lisäksi dokumentointi

on saatettu unohtaa kokonaan tai se on tehty huolimattomasti. Uudistamisen tarve voi syntyä myös silloin, kun halutaan lisätä ohjelmistoon uusia toimintoja.

Uudistamisprosessin eräs merkittävimmistä vaiheista uudistamisen toteutusvaiheeksi kutsuttu vaihe, jossa käytetään apuna käänteistekniikkaa (reverse engineering). Käänteistekniikkaa kutsutaan myös takaisinmallintamiseksi. Käänteistekniikalla tarkoitetaan käänteistä ohjelmistotuotantoprosessia, jossa ohjelman vaatimukset pyritään selvittämään jopa pelkän ohjelman lähdekoodin avulla [ChC90]. Käänteistekniikkaa on käytetty apuna esimerkiksi Cobol-kielisten ohjelmien muokkaamisessa Java-kielelle. Tämän lisäksi käänteistekniikka on todettu käyttökelpoiseksi olio-ohjelmien testauksessa [KuH99]. Kuva 1 esittää uudistamisen, käänteistekniikan ja normaalin ohjelmistotuotannon välisen suhteen.



Kuva 1 Uudistaminen, käänteistekniikka ja normaali, etenevä ohjelmistotuotantoprosessi

Käänteistekniikkaan liittyy kiinteästi siinä apuna käytettävät erilaiset välineet. Välineiden tarkoituksena on helpottaa ohjelman rakenteen ja ohjelman pohjana olleiden vaatimusten selvittämistä. Välineet auttavat esimerkiksi löytämään yhteyksiä suunnittelussa käytettävien UML (unified modelling language) -kaavioiden sekä kaavioiden ja ohjelmakoodin välille. Oikeiden välinevalintojen avulla pystytään vähentämään itse takaisinmallinnusprosessiin kuluvaa aikaa.

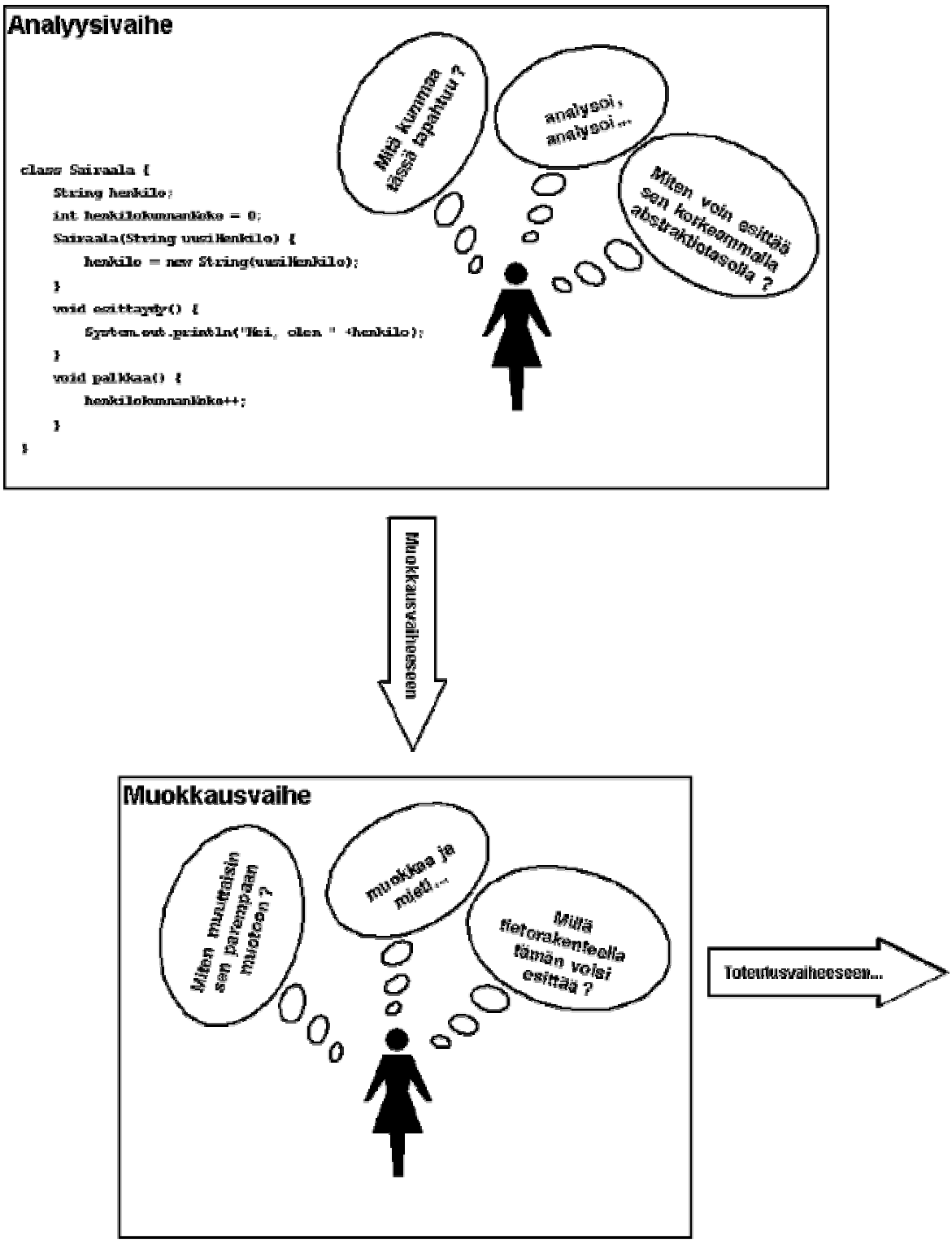
Tässä tutkielmassa perehdytään tarkemmin käänteistekniikkaan. Aluksi luvussa 2 käsitellään ohjelmien uudistamista. Sen jälkeen luvussa 3 selvennetään, mitä käänteistekniikalla tarkoitetaan ja mitä vaiheita siihen kuuluu. Lisäksi kerrotaan käänteistekniikan eri osa-alueista. Luvussa 4 tuodaan esille eri välineitä, joiden avulla käänteis-

tekniikan toteuttamista voidaan helpottaa. Luvussa 5 esitellään yhteyttä UML-kaavioiden ja käänteistekniikan välillä. Luvussa 6 kuvataan käänteistekniikan ongelmia ja lisäksi tuodaan esille kritiikkiä, jota on esitetty käänteistekniikkaa kohtaan. Lopuksi luvun 7 pohdinnassa pyritään kuvaamaan, kuinka käänteistekniikkaa voitaisiin hyödyntää yleisesti ja nostetaan esille sen tarjoamat mahdollisuudet terveydenhuollon tietojärjestelmien integrointiin ja uudistamiseen.

2 Ohjelmien uudistaminen

Ohjelman uudistaminen on prosessi, josta voidaan erottaa ainakin kaksi päätoimintoa. Ensinnäkin uudistettaessa analysoidaan kohteena olevaa järjestelmää. Toinen päätoiminnoista on analysoinnin jälkeen tapahtuva järjestelmän muokkaaminen uuteen muotoon. Analysointivaiheessa tutkitaan, miten ohjelma toimii ja tutkimisen pohjalta pyritään kuvaamaan ohjelmaa korkeammalla abstraktiotasolla. Korkeammalla abstraktiotasolla tarkoitetaan esimerkiksi ohjelman lähdekoodin perusteella luotavia kaavioita, muun muassa erilaisia UML-kaavioita, tai vaikkapa pseudo- eli luonnoskoodia, jolla tarkoitetaan ohjelmien esittämiseen kehitettyä formaalia, lausekieltä muistuttavaa kieltä [Atk99]. Analyysivaiheessa voidaan ajatella palattavan ohjelmiston alkuperäiseen suunnitteluvaiheeseen.

Uudistamisen toisessa vaiheessa eli muokkausvaiheessa muokataan selvitettyjä, ohjelman pohjalla olleita suunnitteluratkaisuja parempaan muotoon. Tässä vaiheessa tulokseen voidaan pyrkiä käyttämällä mahdollisimman paljon suunnittelumalleja (design patterns). Tämän jälkeen on vuorossa uudistamisen vaihe, jossa ohjelma toteutetaan uudestaan. Kuvassa 2 esitetään uudistamisen päävaiheet analysointi ja muokkaus.



Kuva 2 Uudistamisen kaksi päävaihetta: analysointi ja muokkaus

Joissakin tapauksissa ohjelman rakennetta ei sinällään parannella, vaan uudistaminen koskee pelkästään ohjelmakoodin muuntamista jollekin toiselle ohjelmointikielelle esimerkiksi Cobol-kieleltä C++-kieleksi. Tässäkin tapauksessa joudutaan miettimään ohjelman rakennetta uudelleen. Ohjelman rakenteen miettiminen sisältyy ohjel-

makoodin muuntamisvaiheeseen. Tällä tavalla uudistettaessa aluksi analysoidaan, mitä ohjelma tekee ja muokkausvaiheessa selvitetään, kuinka se voidaan esittää toisella ohjelmointikielellä. Uutta ohjelmointikieltä valittaessa tulee huomioida, että kaikilla ohjelmointikielillä ei voida esittää samoja asioita. Etenkin tässä uudistustavassa voidaan käyttää tehokkaasti hyväksi automaattisia apuvälineitä, jotka pystyvät generoimaan ohjelmointikieltä joksikin toiseksi ohjelmointikieleksi.

2.1 UUDISTAMISEN MÄÄRITTELYJÄ JA NIIDEN VERTAILUA

Ohjelmien uudistamista, kuten lähes kaikkia tietojenkäsittelytieteeseen kuuluvia termejä, ovat eri tahot määritelleet useilla eri tavoilla. Määritelmä, johon suurin osa uudistamista koskevista muista määrittelyistä pohjautuu, on Chikofskyn ja Crossin luoma. Chikofsky ja Cross ovat esittäneet taksonomiassaan vuonna 1990 seuraavanlaisen uudistamista koskevan määritelmän: "Uudistamisessa tutkitaan kohteena olevaa järjestelmää ja muutetaan se uuteen muotoon sekä toteutetaan tämä uusi muoto” [ChC90].

Eräs melko usein käytetyistä määritelmistä on Robert S. Arnoldin tekemä, jonka mukaan *uudistamisella* pyritään ensinnäkin parantamaan *ohjelmiston ymmärtämistä* (program comprehension) ja toiseksi parantamaan itse ohjelmistoa lisäämällä sen *uudelleenkäytettävyyttä*, *ylläpidettävyyttä* sekä *kehitettävyyttä* (evolvability) [Arn93]. Uudelleenkäytettävyydellä tarkoitetaan sitä, että ohjelmistoa tai sen osia voidaan hyödyntää sellaisenaan tai vähin muutoksin tulevaisuudessa samassa tai jossakin muussa yhteydessä [Atk99]. Ylläpidettävyydessä on kyse siitä, että ohjelmistoa on helppoa ylläpitää ja päivittää. Tällä tarkoitetaan esimerkiksi uusien toimintojen lisäämistä ohjelmistoon. Muutettavuus tarkoittaa sitä, että ohjelmaa on helppoa muuttaa tarpeiden muuttuessa.

Hausi Müller on luultavasti käyttänyt Chikofskyn ja Crossin määritelmää hyväkseen luodessaan oman määrittelynsä. Hausi Müller esittää, että uudistaminen on prosessi, jossa tutkitaan kohteena olevaa ohjelmistoa ja muokataan se uuteen muotoon [Mül97]. Erään määritelmän mukaan uudistamisessa on kyse siitä, että muutetaan ohjelman tietorakenteita tai sisäisiä mekanismeja, mutta ei muuteta sen toiminnallisuutta [Arn93].

Määrittelijät eivät ymmärrä uudistamista täysin samalla tavoin. Chikofsky ja Cross määrittelevät uudistamisen kolmivaiheiseksi prosessiksi. Ensimmäiseksi tutkitaan, sitten muokataan ja lopuksi vielä toteutetaan. Arnoldin mukainen uudistaminen liittyy tiiviisti ohjelman ymmärtämiseen [Arn93]. Lisäksi Arnoldin mielestä eräs uudistami-

sen tärkeimmistä tavoitteista on yleisesti ohjelman laadun parantaminen. Tämä tulee esille siinä, että halutaan parantaa ohjelmiston laatutekijöinä pidettäviä ominaisuuksia: ylläpidettävyyttä, uudelleenkäytettävyyttä sekä muutettavuutta. Arnold on myös itse selittänyt, että hänen määritelmässään ohjelmisto käsitetään laajasti [Arn93]. Ohjelmisto ei koostu pelkästään lähdekoodista, vaan lisäksi siihen kuuluvat niin mahdollisesti olemassa oleva dokumentaatio kuin suunnittelu- ja analyysitiedot.

Müllerin määritelmän mukaan uudistamisella pyritään saattamaan ohjelmisto uuteen, parempaan muotoon. Eli tässäkin määritelmässä ajetaan periaatteessa takaa entistä laadukkaampaa ohjelmistoa. Müllerin määritelmään liittyy lisäksi Arnoldin määritelmän tavoin pyrkimys ohjelmiston ymmärtämiseen; ohjelmistoa tutkimalla pyritään ymmärtämään ohjelmistoa ja sen tarkoitusta. Lisäksi Arnoldin ja Müllerin määritelmissä ei suljeta pois ohjelmiston toiminnallisuuden muuttamista. Müllerin määritelmässä ei kuitenkaan tuoda selkeästi esille sitä, kuluuko toteuttamisvaihe uudistamiseen vai ei. Yleensä toteuttamisvaihe on selkeästi osa uudistamista.

Viimeisessä määritelmässä taas uudistaminen koetaan melko suppeana; siinä ei muuteta ohjelman toiminnallisuutta, eikä ymmärtämisen käsite nouse mitenkään esille. Näistä kolmesta esimerkkimääritelmästä laajimpana ja kattavimpana pidetään Arnoldin tekemää määrittelyä [Arn93, Har01].

2.2 MITEN UUDISTAMINEN VOIDAAN TEHDÄ?

Voidaan kysyä, milloin on kyse uudistamisesta. Tähän erään vastauksen antaa Linda H. Rosenbergin tekemä selvitys, jonka mukaan uudistaminen voidaan toteuttaa kolmella erilaisella tavalla [Ros96, Har01]:

1. Yhtäkkinen uudistaminen (big bang approach).
2. Vähittäinen uudistaminen (incremental approach).
3. Kehittävä uudistaminen (evolutionary approach).

Seuraavassa käydään läpi näitä uudistamistapoja sekä uudistamisen vaiheita ja ongelma-alueita, Rosenbergin tekemän selvityksen mukaisesti [Ros96].

2.2.1 Yhtäkkinen uudistaminen

Yhtäkkisessä uudistamisessa korvataan *olemassa oleva järjestelmä* (existing system) kokonaisuudessaan *kohdejärjestelmällä* (target system) (Kuva 3). Olemassa olevalla järjestelmällä tarkoitetaan alkuperäistä, uudistettavaa järjestelmää ja kohdejärjestelmällä taas sitä järjestelmää, joka uudistuksen tuloksena syntyy. Yhtäkkinen uudista-

mistapa on käytössä silloin, kun järjestelmä on korvattava välittömästi uudella. Tällainen tilanne voi olla esimerkiksi silloin, kun järjestelmäarkkitehtuuri muuttuu. Koska järjestelmä tuodaan uuteen ympäristöön kerralla, rajapintojen luominen uusien ja vanhojen osien välille on tarpeetonta, sillä käytännössä ei ole olemassa vanhoja ja uusia osia.

Kuva 3 Yhtäkkinen uudistaminen [Ros96]

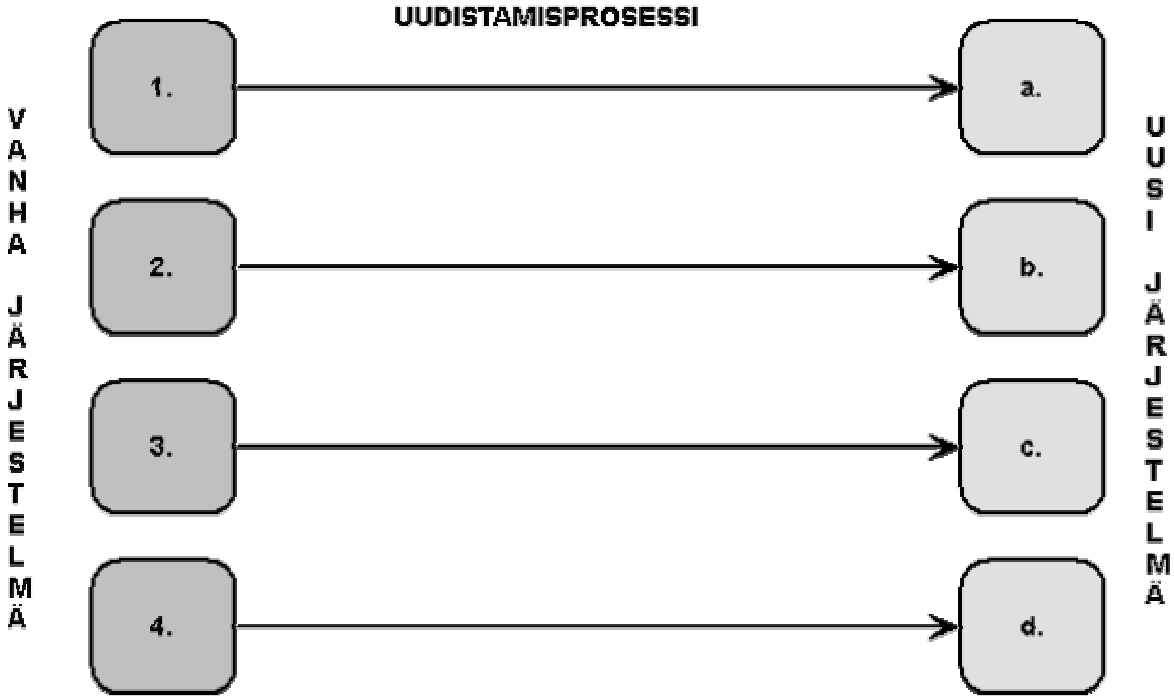
Yhtäkkinen uudistaminen ei kuitenkaan sovi kaikkiin tapauksiin. Etenkin suurille järjestelmille tällainen lähestymistapa on käyttökelvoton; tarvittaisiin liikaa resursseja, kuten rahaa, aikaa ja henkilötyötä, ennen kuin uusi järjestelmä olisi valmis korvaamaan olemassa olevan järjestelmän. Lisäksi uudistuksen tuloksena syntyy *monoliittinen sovellus*, joka ei ole muodoltaan paras mahdollinen. Monoliittisella sovelluksella tarkoitetaan sovellusta, joka koostuu vain yhdestä osasta. Sitä ei ole siis jaettu osiin tai komponentteihin. Monoliittisen sovelluksen ohjelmakoodi on usein niin sanottua spagettikoodia, jonka toiminnallisuudesta on vaikea ottaa selkoa. Tämän vuoksi monoliittisen sovelluksen osittainen uudelleenkäyttäminen on lähes mahdotonta ja myös sen ylläpito on hankalaa.

Yhtäkkiseen uudistamiseen liittyy suuria riskejä, sillä kohdejärjestelmän tulee olla eheä ja pystyä toimimaan rinnakkain olemassa olevan järjestelmän kanssa, jotta uudistamisen onnistuminen voidaan testata ja todeta. Etenkin rinnakkainen toiminta saattaa olla vaikeaa ja liian kallista toteuttaa. Suurin vaikeus on kuitenkin muutosten hallinnassa. Tämä johtuu siitä, että ennen kuin kohdejärjestelmä on täysin valmis, saatetaan joutua tekemään muutoksia nykyiseen eli olemassa olevaan järjestelmään ja nämä muutokset täytyy tietenkin tehdä myös uuteen eli kohdejärjestelmään.

2.2.2 Vähittäinen uudistaminen

Vähittäisessä uudistamisessa järjestelmän osia (component) uudistetaan yksi kerrallaan, eli kuten nimikin jo sanoo, vähitellen. Osien uudistaminen tapahtuu niiden alkuperäisen rakenteen perusteella. Kuvassa 4 esitetään vähittäisen uudistamisen toteutusidea. Koska osat uudistetaan erikseen, ne ovat nopeammin käyttökelpoisia kuin yhtäkkisessä uudistamisessa. Lisäksi mahdolliset virheet löytyvät helpommin, koska

jokainen uudistettu osa on pyritty määrittelemään hyvin. Vähittäisen uudistamisen edut ovat helposti asiakkaankin havaittavissa, sillä uudistettuja osia päästään nopeasti käyttämään. Toisaalta asiakkaiden on helppoa huomata, jos järjestelmän toiminnallisuus on uudistuksen seurauksena huonontunut tai muuttunut. Asiakkaiden on myös helpompi tottua muutokseen, sillä uudistettujen osien ei ole tarkoitus vaikuttaa vielä uudistamattomiin, alkuperäisiin osioihin. Tämän lisäksi vähittäisesti uudistettu järjestelmä on tulevaisuudessa uudelleenkäytettävämpi ja helpommin ylläpidettävä kuin yhtäkkisen uudistamisen tuotoksena syntyvä monoliittinen sovellus.

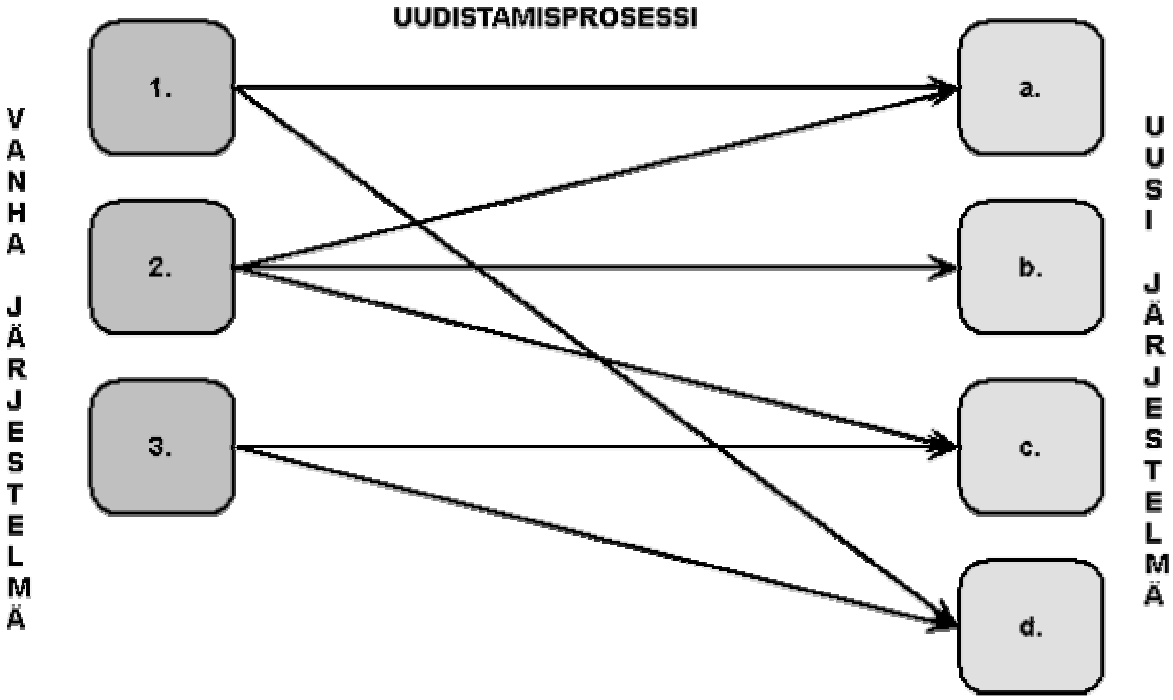


Kuva 4 Vähittäinen uudistaminen [Ros96]

Vähittäisessä uudistamisessa on omat ongelmansa ja riskitekijänsä. Koko järjestelmän uudistaminen vähittäisesti voi olla ajallisesti pitkäkestoista. Vähittäisesti uudistettaessa etenkin laajasta järjestelmästä syntyy useita väliaikaisia versioita, ja sen vuoksi versionhallinnan tulee olla huolellista. Laajoja järjestelmiä on runsaasti esimerkiksi terveydenhuollon käytössä. Lisäksi vähittäisessä uudistamisessa järjestelmän rakennetta ei voida kokonaisuudessaan uudistaa, vaan uudistaminen kohdistuu kunkin osan sisäiseen rakenteeseen. Tämän vuoksi uudistajien tulee tuntea tarkoin nykyisen järjestelmän osien rakenne ja suunnitella kohdejärjestelmän rakenne erittäin huolellisesti. Näistä seikoista huolimatta vähittäinen uudistaminen on riskittömämpi vaihtoehto kuin yhtäkkinen uudistaminen.

2.2.3 Kehittävä uudistaminen

Kehittävässä uudistamisessa uudistetaan järjestelmää osa kerrallaan kuten vähittäisessä uudistamisessakin. Tässä uudistustavassa uudistettavat osiot valitaan kuitenkin niiden toiminnallisuuden, eikä niiden rakenteen perusteella. Eli kohdejärjestelmään lisätään toiminnallisesti toistensa kaltaisia osia tarpeen mukaan (Kuva 5). Toiminnallisesti toistensa kaltaisia ovat esimerkiksi kaikki sellaiset osat, jotka ovat tekemisissä tulostustoimintojen kanssa.



Kuva 5 Kehittävä uudistaminen (Linda H. Rosenbergia mukaillen)

Kehittävässä uudistamisessa uudistajat voivat keskittyä tunnistamaan toiminnallisia osia riippumatta siitä, missä ne rakenteellisesti sijaitsevat. Tämän lisäksi kehittävän uudistamisen tuloksena saadaan modulaarisesti suunniteltu järjestelmä ja samalla osien ulottuvuutta voidaan lisätä vähän kerrallaan. Tämän kaltaista uudistamistapaa kannattaa käyttää etenkin silloin, kun halutaan muuttaa olemassa oleva järjestelmä komponentti- tai olioperustaiseksi kohdejärjestelmäksi. Tällaisen muutoksen edessä ovat monet nykyisin käytössä olevat järjestelmät, esimerkiksi terveydenhuollon järjestelmät.

Kehittävän uudistamistavan haittapuolena on se, että uudistajan tulee ensin kyetä tunnistamaan samankaltaiset toiminnot koko järjestelmästä ja määritellä ne sen jälkeen yhtenä uutena toiminnallisena osana. Uudistajan tulee siis olla erittäin hyvin perillä järjestelmän eri toiminnoista. Lisäksi uudistajan tulee kyetä hahmottamaan, mil-

loin ja millaisia toimintoja on järkevää yhdistää. Koska alkuperäisen järjestelmän toiminnallisia osia on muutettu arkkitehtuuristen sijasta, järjestelmän vastausajat saattavat pidentyä. Lisäksi rajapintojen ja niiden määrittelyn kanssa saattaa tulla ongelmia.

2.3 UUDISTAMISEN VAIHEET

Uudistaminen ei tietenkään onnistu ilman hyvää suunnittelua. Linda H. Rosenbergin tekemän selvityksen mukaan uudistaminen onnistuu parhaiten, jos sen tekee seuraavien viiden vaiheen kautta [Ros96, Har01]:

1. Ryhmän muodostaminen (re-engineering team formation).
2. Mahdollisuuksien arviointi (project feasibility analysis).
3. Analyysi ja suunnittelu (analysis and planning).
4. Uudistamisen implementointi eli toteuttaminen (re-engineering implementation).
5. Testaus ja käyttöönotto (testing and transition).

Ensimmäisessä vaiheessa tarkoituksena on muodostaa ryhmä, jonka tavoitteena on uudistaa jokin järjestelmä tai ohjelmisto. Ryhmän jäseniltä vaaditaan monenlaisia taitoja. Ensinnäkin heidän tulee hallita ohjelmiston kehitysprosessi kokonaisuudessaan. Toiseksi jäsenten on kyettävä hankkimaan ja testaamaan uusia uudistamisessa mahdollisesti käytettäviä työkaluja. Lisäksi heidän tulee valvoa, että yrityksen työntekijät kykenevät käyttämään työkaluja tehokkaasti. Kolmanneksi ryhmän tulee tiedottaa ja markkinoida uudistamisprosessista yrityksen muille työntekijöille sekä neljänneksi käyttää tarvittaessa ryhmän ulkopuolisten työntekijöiden asiantuntemusta apunaan uudistamisessa. Yleisesti voidaan sanoa, että ryhmän jäsenten tulee olla omatoimisia, ryhmätyöskentelyyn kykeneviä markkinointihenkisiä henkilöitä, jotka hallitsevat lisäksi koko ohjelmiston kehitysprosessin.

Toisessa vaiheessa käsitellään uudistamisen mahdollisuuksia. Tässä vaiheessa ryhmä arvioi, mitä yritys tarvitsee ja haluaa olemassa olevasta järjestelmästä. Samalla tutkitaan, mitä rajoituksia nykyisessä järjestelmässä on. Ryhmä määrittelee myös, mihin uudistamisella pyritään eli halutaanko esimerkiksi ensisijaisesti parantaa ohjelmiston laatua vai nostaa sen liiketaloudellista arvoa. Ryhmän tulee laskea näille uudistamistavoitteille jonkinlainen mitattavissa oleva arvo. Tämän arvon avulla ryhmä laskee uudistamisen kannattavuuden yritykselle: uudistuksesta oletettavasti syntyviä säästöjä verrataan uudistamisen kuluihin.

Kolmannessa vaiheessa analysoidaan nykyistä järjestelmää, määritellään kohdejärjestelmän ominaisuudet ja luodaan testausympäristö (testbed), jonka avulla lopuksi varmistetaan, että toiminnallisuus siirretään oikein nykyisestä järjestelmästä kohdejärjestelmään. Ensimmäiseksi siis analysoidaan olemassa olevaa järjestelmää käyttäen apuna järjestelmän lähdekoodia ja sen mahdollista dokumentaatiota. Analysoinnissa pyritään tunnistamaan ohjelman ominaispiirteet. Tällöin kiinnitetään huomiota nykyisen järjestelmän ominaisuuksista etenkin ylläpidettävyyteen ja toiminnallisuuteen, joiden avulla voidaan todistaa kohdejärjestelmän tarpeellisuus. Kun ryhmä on analysoinut olemassa olevan järjestelmän ja sen laadulliset ominaisuudet, siirrytään määrittelemään kohdejärjestelmää ja sen ominaisuuksia. Määrittelyn jälkeen ryhdytään luomaan testiympäristöä, jonka avulla voidaan todistaa kohdejärjestelmän ja nykyisen järjestelmän olevan yhtäpitävät. On tärkeää, että uuden eli kohdejärjestelmän toiminnot ovat *jäljitettävissä* olemassa olevasta eli vanhasta järjestelmästä. Jäljitettävyydellä tässä tarkoitetaan sitä, että uuden järjestelmän toiminnoista on johdettavissa selkeä polku vanhan järjestelmän toimintoihin.

Neljännessä vaiheessa toteutetaan uudistaminen takaisinmallinnuksen ja etenevän ohjelmistotuotannon (forward engineering) vaiheiden avulla. Takaisinmallinnuksessa käydään järjestelmää läpi siirtyen abstraktiotasolta toiselle ja selvitetään samalla nykyisen järjestelmän merkittävät toiminnot. Edellisessä vaiheessa tapahtuvan analysoinnin voidaan myös ajatella olevan takaisinmallinnusta. Tässä neljännessä vaiheessa on mahdollista käyttää apuna erilaisia työvälineitä (tools). Työvälineiden käytettävyys (usability) tulee selvittää huolellisesti ennen niiden käyttöönottoa. Samalla tulee varmistua siitä, että työvälineet soveltuvat kyseiseen takaisinmallinnusprosessiin ilman suurempia muutoksia.

Kun takaisinmallinnuksen avulla on päästy halutulle abstraktiotasolle, aloitetaan etenevä ohjelmistotuotanto (forward engineering), jossa vaihe vaiheelta rakennetaan kohdejärjestelmää. Tämä vaihe tulee tehdä huolellisesti, ettei vahingossa muuteta ohjelmiston toiminnallisuutta vääränlaiseksi. Virheelliset toiminnallisuuden muutokset estetään laadunvarmistus- ja tuotteenhallintatekniikoiden avulla. Esimerkiksi tarkastus (software inspection) [GiG93] voisi toimia hyvänä laadunvarmistustekniikkana.

Viidennessä ja viimeisessä uudistamisen vaiheessa testataan kohdejärjestelmää ja otetaan se käyttöön. Testauksen tavoitteena on varmistua siitä, ettei uudistaminen ole aiheuttanut järjestelmään toiminnallisia virheitä. Uudistamisen tuloksena syntynyttä järjestelmää testataan samoilla tekniikoilla kuin kokonaan uuttakin järjestelmää testat-

taisiin. Kohdejärjestelmän tulee täyttää ne vaatimukset, jotka asetettiin sille analyysi- ja suunnitteluvaiheessa (3. vaihe). Rosenbergin mukaan järjestelmän dokumentaation päivittämisestä tulee huolehtia sen jälkeen, kun järjestelmä on todettu riittävän oikeelliseksi. Kuitenkin enemmän hyötyä voitaisiin saavuttaa, jos järjestelmän dokumentaatiota päivitettäisiin ja korjailtaisiin pitkin uudistusprosessia. Jos muutosten kirjaaminen unohtuu silloin, kun muutokset tehdään, voi jälkikäteen syntyvä dokumentaatio olla lähes hyödytön.

2.4 RISKITEKIJÄT UUDISTAMISESSA

Vaikka uudistamista käytetään osittain riskien vähentämiseksi, siihen itsessään liittyy monia erityyppisiä riskejä [Ros96]. Tämän vuoksi ennen uudistamisprosessin alkamista tulee riskit analysoida tarkoin. Riskien analysointi liittyy kiinteästi lähes jokaisen ohjelmistotuotannossa käytettävän prosessin toteuttamiseen [HaM01]. Uudistamisen riskit liittyvät sen eri vaiheisiin ja myös uudistamisessa käytettäviin apuvälineisiin.

Uudistamisprosessi sisältää erilaisia prosesseille tyypillisiä riskejä ja lisäksi pelkästään uudistamisprosessille tyypillisiä riskejä. Prosesseille tyypillisenä riskinä voidaan pitää väärien arvioiden ja päätelmien tekemistä toteuttamisen suunnitteluvaiheessa. Esimerkiksi ei pystytä arvioimaan, mitä hyötyjä on saavutettavissa oikea-aikaisella rahankäytöllä. Päätös sinällään kalliiden apuvälineiden hankkimisesta voi vähentää uudistamisen kokonaiskuluja huomattavasti, sillä manuaalisesti eli ilman apuvälineitä tehty uudistaminen voi tulla erittäin kalliiksi. Manuaalisessa uudistamisessa työtunnit kuluvat sellaisiin työvaiheisiin, jotka hoituisivat automatisoiduilla välineillä hetkessä. Apuvälineet eivät kuitenkaan auta tilanteessa, jolloin uudistuksen tuomat hyödyt on arvioitu alun perin väärin perustein. Huonosti suunniteltu uudistusprosessi on myös sellainen, jossa keskitytään luomaan kalliita ja hienon näköisiä, mutta merkitykseltään vähäisiä dokumentaatioita.

Uudistuksen lopputulokseen voi myös liittyä suuria suunnittelullisia riskejä. Tällöin uudistamisen tavoitteita ei kyetä saavuttamaan väärän uudistamistekniikan takia, eli on esimerkiksi valittu vähittäinen uudistaminen toteutustavaksi, vaikka kehittävä uudistustapa olisi ollut tilanteeseen sopivampi. Lisäksi valittu uudistamistapa voi olla sellainen, ettei se sovellu kyseisen yrityksen päämääriin, aikatauluihin tai varoihin. Tämä riskitekijä on tyyppiesimerkki etenkin uudistusprosessiin liittyvästä riskistä. Muita mahdollisia uudistuksen lopputulokseen liittyviä uhkakuvia ovat seuraavat:

uudistetun järjestelmän suorituskyky jää riittämättömäksi tai vanha järjestelmä tulee vanhanaikaiseksi ennen kuin uudistaminen on kyetty viemään loppuun asti.

Vastuun puuttuminen on melko tyypillinen kaikenlaisiin prosesseihin liittyvä riski. Tämä riskityyppi kohdataan useimmiten uudenlaisten prosessien käyttöönotossa tai muulla tavoin vaativissa prosesseissa. Jos tämä riski toteutuu, uudistaminen jää ajelehtimaan, ja kukaan ei halua tai kykene ottamaan siitä vastuuta. Esimerkkinä todella epäonnistuneesta vastuun hallinnasta on sellainen tilanne, jossa projektin mennessä huonosti myös yrityksen johtoporras vetäytyy käynnissä olevasta uudistamisprojektista.

Erilaiset laatuasiat ja niiden huomioon ottaminen ovat prosesseille tyypillisiä riskejä. Laatuasioihin liittyvät riskit ovat esimerkiksi seuraavanlaisia: projektin laadunvarmistusta ei tehdä kunnollisesti, sovellusalueen asiantuntijoita ei kuunnella tai tuotteen versioidenhallinta jää huolimattomaksi. Näiden riskien toteutumista on mahdollista välttää hyvillä laadunhallintamenettelyillä sekä suunnittelulla ja aikatauluttamisella [HaM01].

Henkilöstö ja yrityksen sisäiset toimintatavat aiheuttavat omanlaiset riskinsä. Suurin osa näistä riskeistä on tyypillisiä lähes kaikille prosesseille. Henkilöstöpuolella ongelmia voivat aiheuttaa taitojen, kokemuksen ja motivaation puute. Etenkin ohjelmoijilla voi olla motivoitumisongelmia uudistamisprosessissa. Esimerkiksi ohjelmoijan voi olla vaikeaa ymmärtää, miksi jokin ohjelma uudistetaan, vaikka se on toimiva. Toisaalta ohjelmoija voi kokea uudistamisen vääräksi ohjelmanparannustavaksi; hänen mielestään parempi tapa saattaisi olla kokonaan uuden ohjelman rakentaminen.

Toimintatavoista voi löytyä ongelmia tavoitteiden ja välitavoitteiden sekä päämäärien asettamisen kanssa. Myös liian aikainen sitoutuminen koko järjestelmän uudistamiseen kuuluu toimintatapojen riskeihin. Tällöin ei välttämättä ymmärretä, miten laajasta projektista on kysymys. Toimintatapojen aiheuttamiin riskeihin kuuluu myös työvälineiden suunnittelematon käyttö. Esimerkiksi jotain työvälinettä käytetään vain sen vuoksi, että sitä on käytetty ennenkin, vaikka se ei sovellu tämän tyyppiseen uudistamisprojektiin.

Uudistusprosessille tyypillisiä riskitekijöitä löytyy niin takaisinmallinnusvaiheesta, etenevästä ohjelmistotuotantovaiheesta sekä työvälineiden käytöstä ja niiden valinnasta. Takaisinmallinnusvaiheessa vaikeuksia voi syntyä tarpeellisen tiedon tunnistamisen kanssa. Ohjelmakoodista voi olla vaikeaa löytää suunnittelu- ja vaatimustietoja sekä tärkeitä liiketaloudellisia tietoja. Tämän lisäksi olioita tai komponentteja on voitu

tunnistaa väärin tai riittämättömin perustein ohjelmakoodista. Osa tiedoista voi myös jäädä tunnistamatta tai mikä vielä pahempaa: hyödytöntä tietoa tunnistetaan hyödyllisenä tietona. Tunnistamisen lisäksi käytetty ohjelmointikieli voi tuoda ongelmia. Kaikki ohjelmointikielet eivät ole soveliaita esittämään vaatimus- ja suunnittelumäärittelyissä tarvittavaa abstraktia tietoa.

Seuraavat riskitekijät liittyvät etenkin etenevään ohjelmistotuotantovaiheeseen. Uusi järjestelmä ja uusi ympäristö aiheuttavat tämän vaiheen suurimmat riskit. Tunnistamisvaiheessa löydetyt oliot voivat olla sopimattomia uuteen järjestelmään ja toisaalta taas olemassa olevat tietorakenteet saattavat olla siirtokelvottomia uuteen ympäristöön. Näiden lisäksi ongelmia voi aiheutua huolimattomasti tehdystä pohjatyöstä sekä uusien vaatimusten ja toiminnallisuuksien tahattomasta tai tahallisesta lisäilystä.

Kuten edellä jo mainittiin, eräät uudistamisen riskitekijät liittyvät siinä apuna käytettäviin työvälineisiin (tools). Nämä riskit ovat sellaisia, jotka liittyvät lähes kaikkiin prosesseihin, joissa käytetään automatisoituja välineitä. Eräs tyypillisimmistä riskin aiheuttajasta tässä riskikategoriassa on työvälineiden toiminnallisuus. Tällä tarkoitetaan sitä, että työvälineet voivat toimia eritavoin, kun on odotettu. Riskejä voi aiheutua myös sellaisesta tilanteesta, ettei työvälineen toimittaja vastaa tuotteidensa puutteista. Näiden lisäksi työvälineisiin liittyy myös sellainen riskitekijä, että tarvittavia työkaluja ei ole välttämättä saatavilla tai ne ovat vasta varhaisversioasteella.

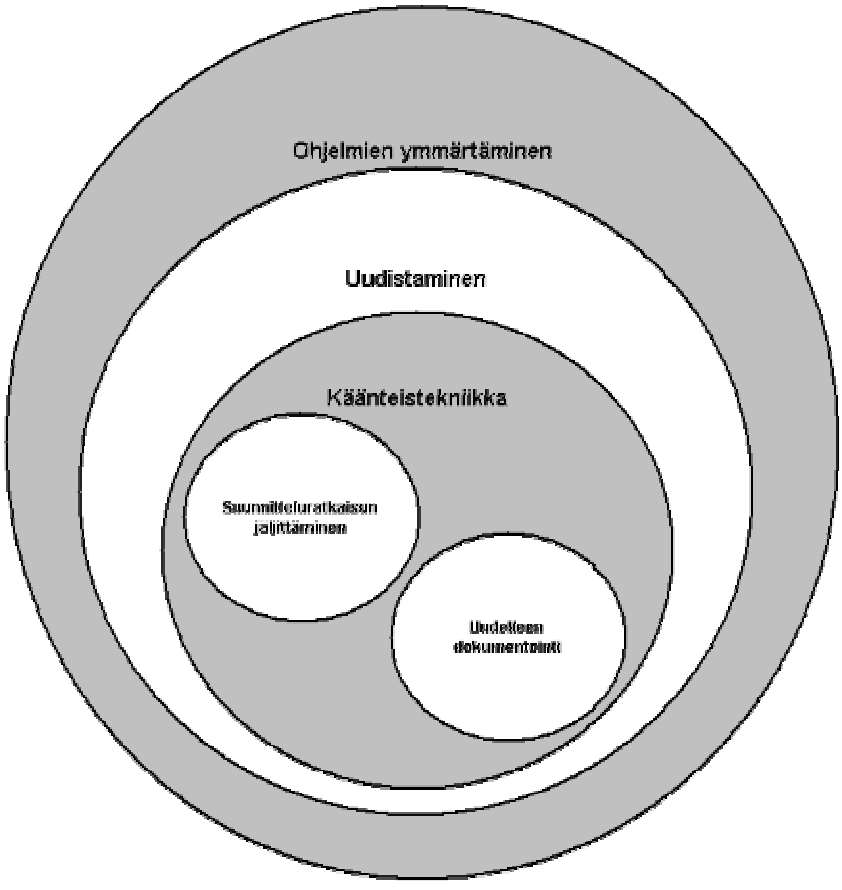
Uudistusprosessi tuntuu olevan täynnä erilaisia riskejä. Riskejä löytyy useamman tyyppisiä ja ne koskettavat uudistusprosessin eri osa-alueita. Kuitenkin tulee ottaa huomioon, että riskien olemassa olo on tyypillistä mille tahansa prosessille. Uudistuksen riskien määrä ei varmaankaan eroa huomattavasti muista ohjelmistotuotannon prosesseista. Hyvällä riskienhallintastrategialla riskit on mahdollista huomioida ja siten niiden toteutuminen voidaan estää. Tällä tavoin ongelmien ilmaantuminen minimoidaan uudistamisprosessissa.

3 Käänteistekniikka

Käänteistekniikka (reverse engineering) on merkittävä vaihe ohjelmien uudistamisessa. Käänteistekniikassa eli takaisinmallinnuksessa on kyse takaperin (reverse) menevästä ohjelmistotuotannosta, jossa vanhojen dokumenttien ja ohjelman lähdekoodin tai jopa pelkästään lähdekoodin avulla voidaan jäljittää ohjelmiston alkuperäiset vaatimukset.

Alun perin käänteistekniikkaa eli takaisinmallinnusta käytettiin laittomiin keinoihin. Käänteistekniikan avulla purettiin toisen valmistajan valmistamia laitteita osiin ja siten pyrittiin selvittämään laitteen valmistussalaisuudet. Tämän vuoksi käänteistekniikka nähdään edelleen hieman epäilyttävänä tekniikkana. Tätä ongelmaa tarkastellaan lisää luvussa 6.2 Ongelmat laillisuuden kanssa.

Käänteistekniikka kuuluu ohjelmistotekniikan alueeseen, jossa tutkitaan ohjelmien ymmärtämistä (program comprehension). Kuvassa 6 esitetään, miten käänteistekniikka sijoittuu ohjelmien ymmärtämisen osa-alueeksi. Ympyröiden kokosuhteet eivät ole verrannollisia eri aihealueiden suhtautumiseen toisiinsa.



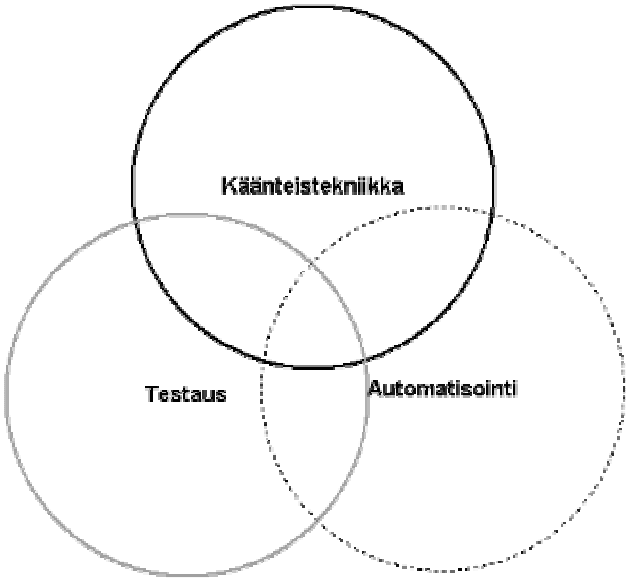
Kuva 6 Käänteistekniikka ohjelmien ymmärtämisen osa-alueena

3.1 YLEISTÄ

Käänteistekniikkaa, kuten muitakin ohjelmistotekniikan käsitteitä, ovat eri henkilöt määritelleet hieman eri tavoin. Eräs useimmin käytetyistä on E. Chikofskyn ja J. H. Crossin määrittely vuodelta 1990. Heidän mukaansa *käänteistekniikka* on prosessi, jossa analysoidaan jotakin tiettyä järjestelmää [ChC90]. Analysoimalla tunnistetaan järjestelmän osat ja osien väliset suhteet sekä sen jälkeen luodaan järjestelmästä kuvaus, joka on muodoltaan erilainen tai abstraktiotasoltaan korkeampi. Käänteistekniikkaan siis kuuluu ensinnäkin tarpeellisen tiedon löytäminen sekä toiseksi tiedon muuntaminen korkeammalle abstraktiotasolle. Käänteistekniikka ei siis itsessään sisällä ohjelmiston uudelleenrakentamista vaan sen tarkoituksena on pelkästään tutkia ja analysoida olemassa olevaa järjestelmää [ChC90]. Uudistaminen sisältää myös tuon ohjelmiston uudelleenrakentamisen. Käänteistekniikan avulla löydettyjä tietoja voidaan hyödyntää pitkin ohjelmistokehityksen elinkaarta [RaC99]. Esimerkiksi käänteistekniikan avulla voidaan helpottaa projektien hallintaa sekä tietenkin ohjelmiston osien uudelleenkäyttöä ja myös ohjelmistojen kehittämistä laadukkaammiksi tuotteiksi. Käänteistekniikkaa on myös käytetty hyväksi erilaisten järjestelmien rakenteen parantamisessa.

Käänteistekniikkaa voidaan käyttää myös hyväksi olio-ohjelmien testauksessa [KuH99]. Tällöin käytetään hyväksi graafista mallia nimeltä *olioperustainen testausmalli* (Object-Oriented Test Model), joka on käänteistekniikkaa käyttävä malli. Mallin avulla pyritään auttamaan testaajia ymmärtämään oliopohjaisten ohjelmien osien rakennetta ja osien välillä vallitsevia suhteita. Tämän lisäksi mallin pyrkimyksenä on helpottaa testaajia testitapausten ja -skeneerioiden luomisessa sekä tehostaa testausstrategioita ja näiden luomista.

Tämän mallin avulla voidaan havaita, että käänteistekniikka, testaus ja automatisointi liittyvät toisiinsa kuvan 7 esittämällä tavalla. Käänteistekniikasta lainatun algoritmin avulla voidaan luoda automatisoitu väline, jota voidaan käyttää hyväksi testauksessa.



Kuva 7 Käänteistekniikan suhde testauksen ja automatisoinnin kanssa

Käänteistekniikan eli takaisinmallinnuksen tarkoituksena on tukea ohjelmien ymmärtämistä [Sys00]. *Ohjelmien ymmärtämisellä* tarkoitetaan tässä kykyä ymmärtää, mitä ohjelma tekee ja millaiset suunnitteluratkaisut ovat ohjelman taustalla. Tätä päämäärää voidaan helpottaa eri tavoin, kuten esimerkiksi ylläpidon ja uudelleenkäytön avulla sekä huolehtimalla dokumentaatiosta ja lisäksi uudistamalla tarpeen mukaan kohteena olevaa järjestelmää. Ohjelmien ymmärtämistä voidaan tukea myös tuottamalla *suunnittelumalleja* (design patterns) olemassa olevasta järjestelmästä. Suunnittelumalli on määritelmänsä mukaan yleinen ja todistettavissa oleva ratkaisu jossakin tietyssä aihepiirissä yleisesti esiintyvään arkkitehtuuri- tai suunnitteluongelmaan [Eer02]. Voidaan arvioida, että mitä enemmän suunnittelumalleja on löydettävissä järjestelmästä, sitä laadukkaampi se on [Sys00]. Tämä johtuu siitä, että suunnittelumallit ovat osaa hyvää ohjelmointikäytäntöä. Ne ovat sitä silloinkin, kun suunnittelumallia käyttänyt ohjelmoija ei ole tehnyt sitä tietoisesti. Ohjelmistotekniikan kannattaisikin investoida ohjelmien ymmärtämisen tutkimiseen ja sen tekniikoihin. Siten olisi mahdollista vähentää vanhojen järjestelmien uudistamiseen ja kehittämiseen liittyviä kuluja ja riskejä [MüJ00].

Käänteistekniikasta eli takaisinmallinnuksesta voidaan erottaa kaksi eri puolta sen perusteella, miten ohjelmaa tarkastellaan. Tarja Systä nimittää ohjelmiston staattisen rakenteen mallintamista *staattiseksi takaisinmallinnukseksi* (static reverse engineering) ja dynaamisen käyttäytymisen mallintamista *dynaamiseksi takaisinmallinnukseksi* (dynamic reverse engineering). [Sys00]

Jaottelu voi tapahtua myös sen perusteella, minkä tiedon perusteella takaisinmallinnus suoritetaan. Voidaan puhua *koodin takaisinmallinnuksesta* (code reverse engineering) ja *tiedon takaisinmallinnuksesta* (data reverse engineering) [MüJ00].

Lisäksi takaisinmallinnuksen voidaan katsoa muodostuvan kahdesta hieman erityyppisestä osa-alueesta: *suunnitteluratkaisun jäljittämisestä* (design recovery) ja *uudelleendokumentoinnista* (re-documentation) [ChC90].

Käänteistekniikkaa voidaan siis tarkastella eri näkökulmista ja sen perusteella erottaa siitä erilaisia muotoja. Monesti nämä muodot voivat sisältää osittain tai kokonaan jonkun toisen tavan tehdä takaisinmallinnusta. Esimerkiksi suunnitteluratkaisun jäljittämisessä voidaan käyttää hyväksi niin staattista kuin dynaamistakin takaisinmallinnusta.

3.2 STAATTINEN JA DYNAAMINEN TAKAISINMALLINNUS

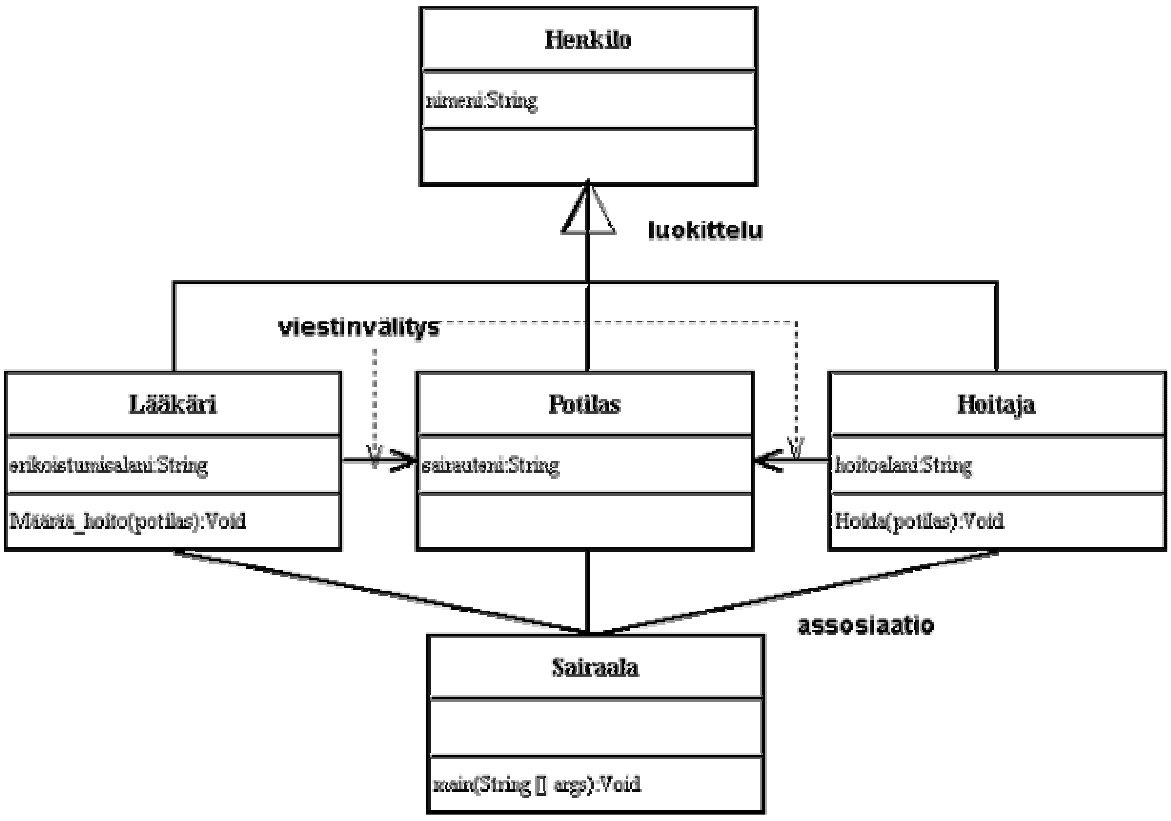
Ohjelmistoa takaisinmallinnettaessa tulee ottaa huomioon niin ohjelmiston staattinen kuin dynaaminenkin puoli [Sys00]. Molemmat puolet tuovat takaisinmallinnuksessa tarvittavaa tietoa ohjelmiston osista ja niiden välisistä suhteista. Staattista ja dynaamista puolta on syytä tarkastella erikseen, mutta myös yhdessä. Tarpeellisen tiedon löytymiseksi tarvitaan moniulotteista ohjelmiston tarkastelua. Lisäksi pelkän tiedon käyttöarvo on erittäin pieni, ellei sitä voida esittää jollakin kuvailevalla tai luettavalla tavalla. Juuri takaisinmallinnusvälineiden avulla saadaan muodostettua graafisia kuvauksia järjestelmistä. Seuraavassa tarkastellaan esimerkkikoodia (Esimerkki 1) ja siitä saatavaa tietoa.

Esimerkki 1. Ohjelmakoodi

```
public class Sairaala {
     public static void main(String[] args) {
          laakari lastenlaakari = new laakari("Minna");
          hoitaja lastensairaanhoitaja = new hoitaja("Mikko", "lastensairaanhoito");
          potilas lapsi = new potilas("Maiju", "vesirokko" );
          lastenlaakari.Maaraa_hoito(lapsi);
          lastensairaanhoitaja.Hoida(lapsi);
     }
}

public class henkilo {
     protected String nimeni;

     public henkilo(String nimi) {
          nimeni = nimi;
```

```
    }
}

public class laakari extends henkilo {
    private String erikoistumisalani;

    public laakari(String nimi) {
        super(nimi);

    }
    public laakari(String nimi, String erikoistumisala) {
        super(nimi);
        erikoistumisalani = erikoistumisala;
    }

    public void Maaraa_hoito(potilas p) {

    }

}

public class hoitaja extends henkilo {
    private String hoitoalani;

    public hoitaja(String nimi){
        super(nimi);
    }

    public hoitaja(String nimi, String hoitoala) {
        super(nimi);
        hoitoalani = hoitoala;
    }

    public void Hoida(potilas p) {

    }
}

public class potilas extends henkilo {
    private String sairauteni;

    public potilas(String nimi){
        super(nimi);
    }

    public potilas(String nimi, String sairaus) {
        super(nimi);
        sairauteni = sairaus;
    }

}
```

Staattisessa takaisinmallinnuksessa etsitään ohjelmasta tietoa sen perusteella, millainen ohjelman rakenne on ohjelmakoodin mukaan [Sys00]. Tässä takaisinmallinnustyypissä tarkastellaan siis pelkästään ohjelman lähdekoodia. Staattisen mallinnuksen tarkoituksena on kuvata ohjelmiston osien eli esimerkiksi luokkien pysyvät, syötteistä riippumattomat suhteet [KoM96]. Tällaisia suhteita ovat muun muassa osakokoonpano-, assosiaatio- sekä periytymissuhde. Staattista tietoa jostakin tietystä ohjelmakoodista voidaan esittää luokkakaavioiden avulla (Kuva 8). Luokkakaavio antaa tietoa kohteena olevan ohjelman staattisista elementeistä ja niiden sisällöstä sekä niiden välisistä suhteista. Staattista tietoa ovat esimerkiksi luokat, rajapinnat, muuttujat ja metodit. Suhteita taas ovat muun muassa metodien väliset kutsut sekä luokkien väliset jatkuvuussuhteet (extension relationship). Staattisen tiedon erottelussa ohjelmakoodista voidaan eräänä keinona käyttää kielioppiin perustuvia kääntäjiä (parsers based on grammar). Staattisessa takaisinmallinnuksessa käytettäviä tapoja ovat esimerkiksi syntaksi- ja tyyppitarkastukset sekä kontrolli- ja tietovirta-analyysit.

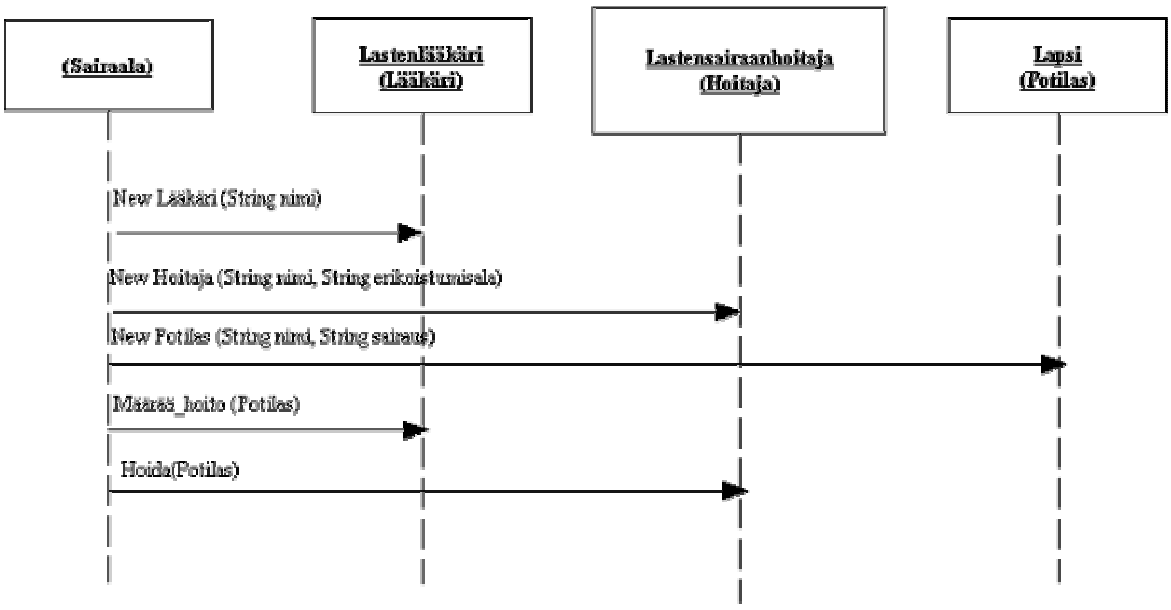


Kuva 8 Staattista tietoa ohjelmakoodista: esimerkki 1 liittyvä luokkakaavio

Dynaamisessa takaisinmallinnuksessa kiinnitetään huomiota siihen, miten ohjelma käyttäytyy ajonaikana [Sys00]. Dynaamisessa takaisinmallinnuksessa tarkastellaan siis ohjelman ajonaikaista toimintaa, ei pelkästään lähdekoodia, kuten staattisessa ta-

kaisinmallinnuksessa. Ohjelman ajonaikaisen toiminnan perusteella yritetään ohjelmasta löytää haluttua tietoa. Dynaamisen takaisinmallinnuksen apuvälineinä käytetään tapahtumien *tallentajia* (event recorder), *tulkintaohjelmia* (debugger) sekä *profiloijia* (profiler). Yleensä dynaamisen analyysin avulla saadaan selvyyttä esimerkiksi ohjelman ajonaikaiseen toimintaan ja muistin hallintaan sekä lisäksi saadaan tietoa peräkkäisistä tapahtumista (event).

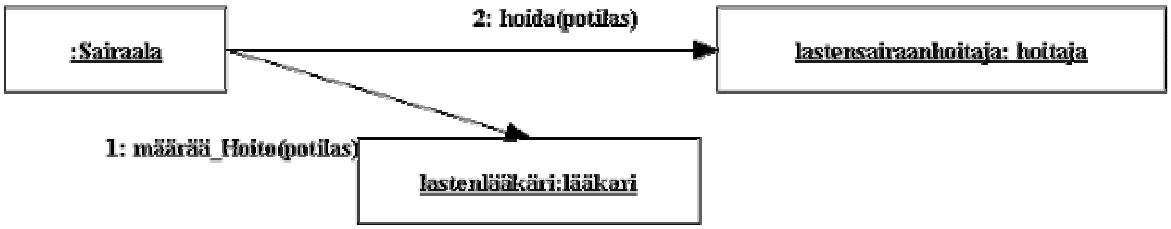
Dynaaminen takaisinmallinnus on tärkeä osa oliopohjaisten järjestelmien tutkimista. Esimerkkejä oliopohjaisten ohjelmien dynaamisesta käytöksestä ovat ajonaikainen sitominen ja olioiden luominen. Näitä dynaamisia piirteitä on lähes mahdotonta ymmärtää pelkästään ohjelman lähdekoodia tutkimalla. Löydetyn dynaamisen tiedon kuvaamisessa voidaan käyttää hyväksi esimerkiksi tapahtumasekvenssikaavioita eli skenaarioita (kuva 9) sekä tilakaavioita. Tapahtumasekvenssikaavioiden ja tilakaavioiden välillä vallitsee selkeä suhde; tapahtumasekvenssikaavioista voidaan johtaa tilakaavioita. Tämä tapahtuu ottamalla erikseen käsittelyyn jokainen tapahtumasekvenssikaavion olio (pystyviiva) ja sen tapahtumat. Olioon tulevat tapahtumat aiheuttavat tilakaavioissa tarkasteltavat tilasiirtymät [HaM98]. Tilakaaviot sisältävät toteutuksen kannalta kaiken sen tiedon, mitä tapahtumasekvenssikaaviokin. Tämä ei päde päinvastaiseen suuntaan. Tapahtumasekvenssikaavio on kuitenkin yleensä havainnollisempi kuin tilakaavio. Yhteyden selvittämiseen tila- ja tapahtumasekvenssikaavioiden välillä on Tampereen yliopistossa kehitetty SCED-väline [KoM96].



Kuva 9 Dynaamisen tiedon mallintaminen ohjelmakoodista: esimerkki 1 liittyvä tapahtumasekvenssikaavio

Staattista ja dynaamista tietoa voidaan kuvata myös samassa näkymässä [Sys00]. Tällainen kuvaustapa on hyödyllinen silloin, kun halutaan löytää yhteydet staattisen ja dynaamisen tiedon välillä. Lisäksi yhdistämällä molemmat piirteet samaan näkymään voidaan parantaa kuvaksen laatua ja varmistaa, että ohjelma tai sen osa on ymmärretty oikein. Esimerkiksi moniperintää (polymorphism) ei voida täysin ymmärtää staattisen analyysin avulla, sillä ohjelman lähdekoodista voidaan nähdä vain mahdolliset vaihtoehdot metodikutsulle. Kun tällaiseen näkymään lisätään dynaaminen, ajonaikana tapahtuva tarkastelu, on löydettävissä oikea metodikutsu kyseessä olevaan tilanteeseen.

Toisaalta molempien piirteiden mahduttaminen samaan näkymään ei aina onnistu, sillä staattiset ja dynaamiset piirteet ovat erilaisia suhteessa toisiinsa. Useimmissa järjestelmissä staattiset abstraktiot ovat yleensä alijärjestelmiä ja dynaamiset abstraktiot taas käyttötapauksia. Esimerkiksi sairaalajärjestelmästä löytyvää staattista tietoa ovat osastot sekä hoitohenkilökunta ja dynaamista tietoa taas erilaiset käyttötapaukset kuten uuden potilaan tietojen lisääminen. Tietyn tyyppisten yhdistettyjen kuvausten luominen on mahdollista; esimerkiksi UML:n mukaisella yhteistyökaaviolla (collaboration diagram) voidaan kuvata niin staattisia kuin dynaamisiakin ominaisuuksia. Tulee kuitenkin huomioida, että yhteistyökaavion sisältämä tietomäärä kasvaa helposti liian suureksi ja sen vuoksi siitä tulee vaikeasti hallittava ja epäselvä. Kuvassa 10 on yhteistyökaavio, joka pohjautuu osaan edellä esitetyn esimerkki 1 ohjelmakoodia.



Kuva 10 Yhteistyökaavio osasta esimerkki 1 ohjelmakoodia

Rakennettaessa kuvausta, jossa huomioidaan niin staattiset kuin dynaamiset piirteet, tulee jo aikaisessa vaiheessa päättää, kumpi puolista on kuvauksen pohjana ja kumpi tulee taas täydentämään kuvausta. Kuvausta ei voida rakentaa siten, että kumpikin piirre olisi kuvauksessa täysin tasaveroisesti esitettynä. Lisäksi on huomioitava, että mitä enemmän tietoa lisätään samaan näkymään, sitä epäselvempi näkymästä tulee.

Luvussa 5 UML-kaaviot ja käänteistekniikka keskitytään hieman tarkemmin eri UML-kaavioihin ja niiden käyttämiseen takaisinmallinnuksessa.

3.3 KOODIIN KESKITTYVÄ TAKAISINMALLINNUS

Ohjelmistotuotanto on pitkään pyörinyt ohjelmakoodin ympärillä. Koodi tuntuu olevan monelle ohjelmistoalan ammattilaiselle tärkein asia. Käänteistekniikkakin on painottunut ohjelmakoodin perusteella tehtäväksi. Yleensä takaisinmallinnuksesta puhuttaessa puhutaankin juuri ohjelmakoodin takaisinmallinnuksesta. Edellä esitellyt staattinen ja dynaaminen takaisinmallinnus ovat osa koodiin keskittyvää takaisinmallinnusta. Samoin myöhemmin esiteltävät suunnitteluratkaisun jäljittäminen ja uudelleendokumentointi pohjautuvat yleensä ainakin osittain juuri koodiin keskittyvään takaisinmallinnukseen.

Koodiin keskittyvässä takaisinmallinnuksessa tuotetaan ohjelmasta tietoa ohjelmakoodin perusteella. Tähän takaisinmallinnustyyppiin soveltuvat hyvin erilaiset automaattiset takaisinmallinnusvälineet. Ohjelmakoodista tuotettava tieto voi olla esimerkiksi UML-kaavioita tai vapaamuotoisia ohjelman rakennetta kuvaavia kaavioita.

Koodin perusteella tapahtuva takaisinmallinnus on ollut tyypillisin tapa suorittaa tiedon etsimistä [MüJ00]. Tämä johtuu useimmiten siitä, että ainoa olemassa oleva tieto ohjelmistosta sijaitsee ohjelmakoodissa. Lisäksi ohjelmakoodia pidetään eräänä luotettavimmista tiedon lähteistä. Tulee kuitenkin muistaa, ettei ohjelmakoodi ole aina luotettavaa. Monesti ohjelmakoodin sekaan unohtuu tarpeettomia koodinpätkiä, joissa ei ole mitään tarpeellista toiminnallisuutta. Samoin ohjelmakoodin sekavuus voi aiheuttaa väärinymmärryksiä ohjelman toiminnasta.

3.4 TIETOON KESKITTYVÄ TAKAISINMALLINNUS

Tiedon takaisinmallinnus pyrkii selvittämään, mitä tietoa on tallennettu järjestelmiin ja kuinka tätä tietoa voidaan hyödyntää erilaisissa yhteyksissä [MüJ00]. Tiedon takaisinmallinnus yhdistää rakenteisen tiedon analyysitekniikat täsmällisiin tiedonhallintakäytäntöihin [Aik98]. Jos takaisinmallinnuksessa keskitytään joko järjestelmä- tai organisatoorisiin tietoihin, tulisi puhua tiedon takaisinmallinnuksesta. Peter H. Aikenin mukaan tiedon takaisinmallinnus tarjoaa tehokkaita keinoja tilanteiden ratkaisuun, joissa

- tutkitaan koko järjestelmää koskevaa tiedon käyttöä,
- tarkastellaan ongelmia, jotka aiheutuvat monimutkaisesta tiedon siirrosta tai rajapinnoista tai

- uudistamisen tavoitteiden saavuttaminen vaatii mieluummin strategista kuin toiminnallista analysointia [Aik98].

Tietokantojen takaisinmallinnus (database reverse engineering) on eräs tiedon takaisinmallinnuksen paremmin jäsentynyt osa-alue. Tietokantojen takaisinmallinnuksessa voi olla kyse esimerkiksi siitä, että vanha tietokanta pitää muuttaa toiseen muotoon.

Tiedon takaisinmallinnus on pitkään ollut perinteisen koodin takaisinmallinnuksen varjossa, mutta vähitellen kiinnostus on kasvanut [MüJ00]. Suurimmat syyt, miksi tiedon takaisinmallinnusta on hyljeksitty, ovat seuraavat:

1. perinteinen erottelu tietokantajärjestelmien ja ohjelmistotekniikkayhteisöjen välillä ja
2. tutkijoiden olettama, että koodin takaisinmallinnus on haastavampaa ja mielenkiintoisempaa.

Viime aikoina lisääntyneeseen kiinnostukseen tiedon takaisinmallinnusta kohtaan ovat vaikuttaneet suuret tietoihin liittyvät muutokset kuten esimerkiksi yhteisvaluutan käyttöönotto Euroopan Unionin alueella ja myös vuosituhannen vaiheen aiheuttamat muutokset laajoihin tietojärjestelmiin.

Järjestelmissä on tallennettuna paljon tietoa, josta on apua uusien järjestelmien luomisessa [MüJ00]. Esimerkiksi tiedollisen dokumentaation avulla pystytään suunnittelemaan huolellisesti uuden järjestelmän rakentaminen. Tällöin pystytään välttymään ongelmilta, joita on syntynyt aiemmin, kun vanha järjestelmä on korvattu uudella. Esimerkkinä tällaisesta ongelmasta voisi mainita Suomen verotuksen myöhästymisen yli vuodella, kun verohallinnon järjestelmä uusittiin.

Tiedon takaisinmallinnuksen avulla voidaan myös huolehtia ohjelmistojen laadusta [MüJ00]. Jos tietorakenteet ovat huonosti suunniteltuja, siitä yleensä on seurauksena huonosti toteutettu järjestelmä. Tätä tietoa voidaan hyödyntää etenkin silloin, kun yritys tekee hankintapäätöksiä valmiista ohjelmiston osista tai komponenteista.

3.5 SUUNNITTELURATKAISUN JÄLJITTÄMINEN

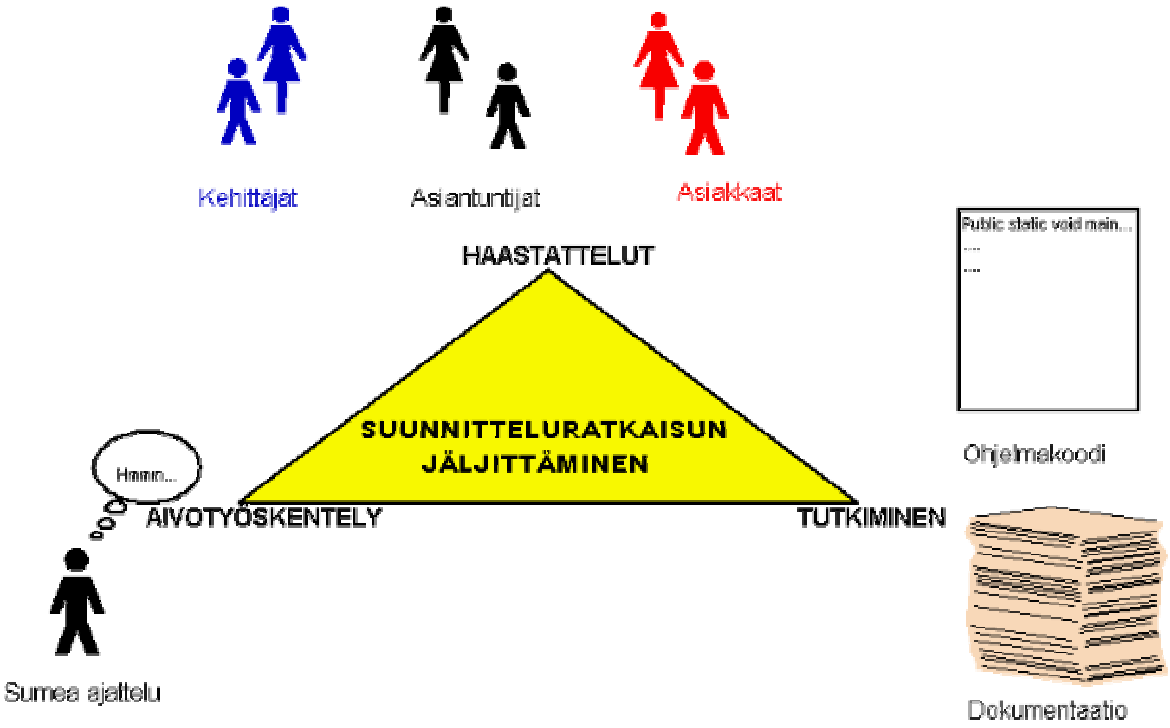
Ted J. Biggerstaff on vuonna 1989 määritellyt *suunnitteluratkaisun jäljittämisen* seuraavasti: suunnitteluratkaisun jäljittämisessä luodaan abstraktio siitä, kuinka ohjelma on suunniteltu [Big89]. Tässä käytetään apuna ohjelman lähdekoodia, suunnitteludokumentaatiota, henkilökohtaisia kokemuksia sekä yleistä tietämystä kyseisestä sovellusalasta ja ongelmasta.

Chikofsky ja Cross taas määrittelivät saman asian seuraavana vuonna näin: suunnitteluratkaisun jäljittämisessä on kyse

1. alakohtaisen tietämyksen,
2. ulkopuolisen tiedon sekä päätelmien tai
3. sumean ajattelun

yhdistämisestä havaintoihin, joita on tehty kohteena olevasta järjestelmästä [ChC90].

Suunnitteluratkaisun jäljittämisessä siis käytetään hyväksi kaikkea mahdollista tietoa, mitä järjestelmästä on olemassa (Kuva 11). Tietoa pyritään etsimään lähdekoodin ja muun dokumentaation lisäksi myös haastattelemalla järjestelmää kehittämässä olleita henkilöitä sekä mahdollisesti myös asiakkaita Tämän lisäksi sumealla ajattelulla ja yleisellä alakohtaisella tietämyksellä on oma sijansa tässä prosessissa.



Kuva 11 Suunnitteluratkaisun jäljittäminen

Tärkeänä apuna suunnitteluratkaisun jäljittämisessä toimii niin kutsuttu *vapaamuotoinen tieto* (informal information). Tällä tarkoitetaan ohjelmointikielen rakenteiden liittämistä sovellusalueeseen esimerkiksi nimeämiskäytäntöjen ja ohjelmakoodin kommentoinnin avulla. Tätä valotetaan seuraavaksi Java-ohjelmointikielisten koodiesimerkkien avulla, joista ensimmäisessä on käytetty huonoa nimeämistapaa ja toisessa taas hyvää.

Jos ohjelmakoodissa ei ole käytetty kuvaavaa nimeämistapaa (Esimerkki 2), koodin toiminnan ymmärtäminen on erittäin hankalaa. Tällaisesta koodista voi yleensä ymmärtää vain esimerkiksi, mikä funktio kutsuu mitäkin tai millaiset muuttujat; esimerkiksi tyypiltään taulukkoja vai merkkejä, kuuluvat funktioihin. Huonosti nimettyä tai kommentoitua ohjelmakoodia on vaikea käyttää hyväkseen suunnitteluratkaisun jäljittämisessä. Siitä ei saada irrotettua sellaisia tietoja, jotka pystyttäisiin yhdistämään ohjelmakoodia tutkivan henkilön olemassa olevaan tietämykseen. Tämän vuoksi ohjelman merkitys eli semantiikka ei tule selväksi ja ohjelmaa ei voida täysin ymmärtää.

Esimerkki 2. Huonosti nimetty ohjelmakoodi

```
class xy {
    String m1;
    int m2 = 0;
    xy(String m) {
        m1 = new String(m);
}
    void zz() {
        System.out.println("Hei, olen " +m1);
    }
    void ww() {
        m2++;
    }
}
```

Toisessa esimerkissä (Esimerkki 3) on sama ohjelmakoodi kirjoitettu käyttäen kuvaavampaa nimeämistapaa, ja sen vuoksi ohjelmakoodin toiminta on ymmärrettävää. Hyvin nimetty ja kommentoitu ohjelmakoodi on liitettävissä alakohtaiseen tietämykseen. Jopa vähänkin ohjelmoinut ihminen kykenee ymmärtämään, mistä on kysymys. Voidaan sanoa, että ohjelmakoodin semantiikka on selvä. Tässä käytetyt esimerkkikoodit ovat melko yksinkertaisia, mutta niiden avulla pystytään näkemään, mikä merkitys hyvällä nimeämis- ja kommentointikäytännöllä on.

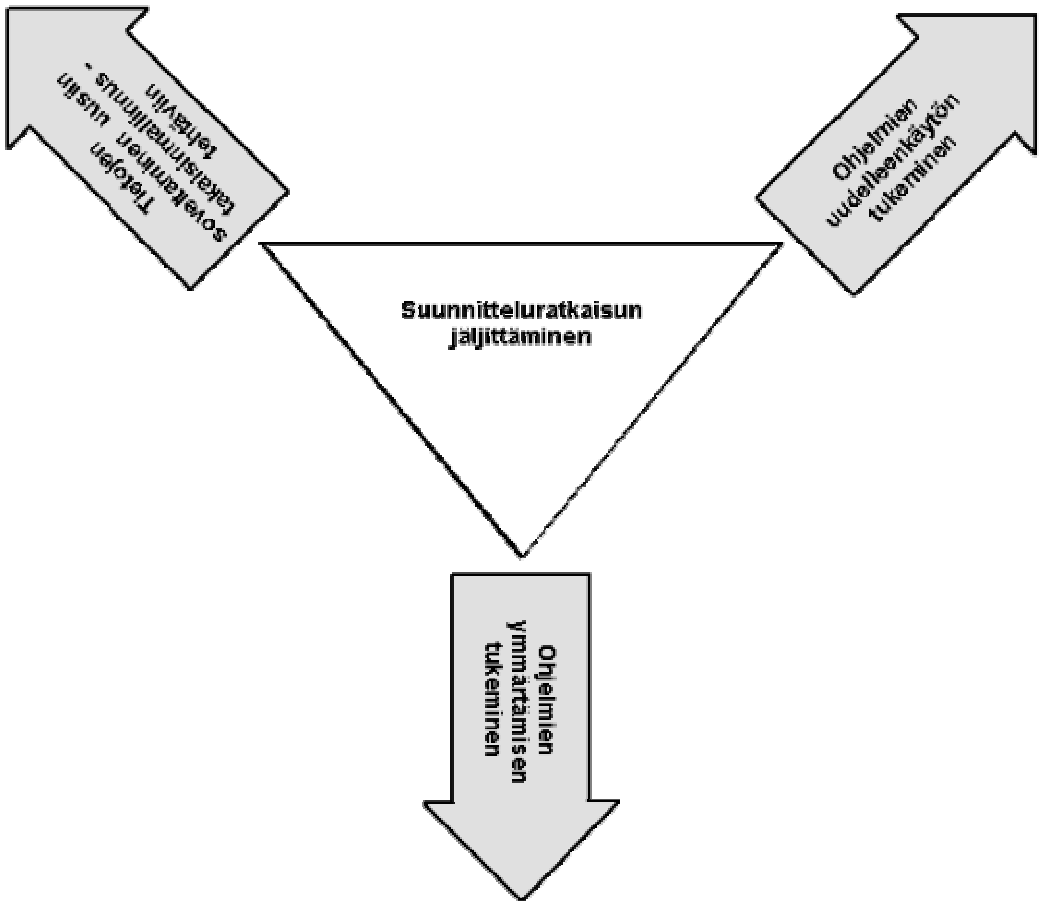
Esimerkki 3. Ohjelmakoodi, jossa on käytetty hyvää nimeämistapaa

```
class Sairaala {
    String henkilo;
    int henkilokunnanKoko = 0;
    Sairaala(String uusiHenkilo) {
        henkilo = new String(uusiHenkilo);
    }
    void esittaydy() {
```

```
        System.out.println("Hei, olen " +henkilo);
    }
    void palkkaa() {
        henkilokunnanKoko++;
    }
}
```

3.5.1 Suunnitteluratkaisun jäljittämisen tavoitteet

Suunnitteluratkaisun jäljittämisen tarkoituksena on tuottaa materiaalia, jonka avulla ohjelmiston kanssa tekemisissä oleva henkilö voi täysin ymmärtää ohjelman käyttäytymistä eli mitä ohjelma tekee, miten se sen tekee ja miksi se toimii niin kuin toimii [Big89]. Yleisesti suunnitteluratkaisun jäljittämiselle voidaan ajatella kolme päätavoitetta: ensinnäkin pyritään tukemaan ohjelmien ymmärtämistä, toiseksi tuetaan ohjelmien uudelleenkäyttöä ja kolmanneksi voidaan soveltaa saatuja tietoja joihinkin tuleviin takaisinmallinnustehtäviin (Kuva 12).



Kuva 12 Suunnitteluratkaisun jäljittämisen tavoitteet

Mielenkiintoista on se, että suunnitteluratkaisun jäljittämistä tehdään usein vahingossa; esimerkiksi uudet ylläpitäjät joutuvat selittämään itselleen ohjelman rakennetta ja muutosten aiheuttamia vaikutuksia ohjelmistoon ennen kuin voivat tehdä vaadittuja muutoksia. Tällöin he siis pyrkivät ymmärtämään ohjelmaa ja jäljittävät suunnitteluratkaisuja oikeastaan tietämättään.

Ensimmäisen tavoitteen, eli ohjelman ymmärtämisen, saavuttamisessa on paras aloittaa laajojen rakenteiden ymmärtämisestä [Big89]. Laajoilla rakenteilla tarkoitetaan muun muassa tärkeimpiä tietorakenteita, moduulirakenteita tai pääkomponentteja. Tämän jälkeen voidaan siirtyä analysoimaan suunnittelurakenteita ja pyrkiä kuvaamaan nämä rakenteet ohjelmakoodia abstraktimmassa muodossa kuten erilaisina kaavioina ja käsitteinä sekä suunnitteluperusteina. Seuraavassa on kysymyksiä, joiden esittäminen helpottaa ohjelman ymmärtämistä.

1) Mistä moduuleista, komponenteista tai muista osista ohjelma koostuu?
2) Mitkä ovat ohjelman merkittävimmät tietoalkiot?
3) Mitä erilaisia tuotoksia ja dokumentteja on syntynyt ohjelmistotuotantoprosessin aikana?
4) Mitä vapaamuotoisia suunnitteluabstraktioita on löydettävissä?
5) Miten suunnitteluabstraktiot ovat suhteessa ohjelmakoodiin?

Eräät ohjelmointikielet tukevat moduuliajattelua tai komponenttiajattelua. Tämän vuoksi ohjelman osat on helposti tunnistettavissa ohjelman lähdekoodin perusteella. Ellei ohjelmointikieli tue tällaisia rakenteita, ohjelman osien tunnistamiseen tulee käyttää hyväksi yhdistelmää erilaisista tiedoista. Esimerkiksi voidaan yhdistää kokemus tältä kyseiseltä sovellusalueelta, lähdekoodi sekä tieto siitä, miten yritys yleensä on toiminut tällaisen ohjelmointiongelman yhteydessä.

Tietoalkioiden löytämisessä tärkein apuväline on kokemus kyseiseltä sovellusalueelta. Mitä enemmän kokemusta on, sen helpompaa tärkeimpien tietoalkioiden tunnistamisesta tulee. Esimerkiksi mitä enemmän jäljittäjä tietää terveydenhuollon potilastietojärjestelmistä, sitä helpompaa on tunnistaa kyseisen järjestelmän merkitykselliset tietoalkiot.

Ohjelman ymmärtämiseen kuuluu suunnittelutietojen kuvaaminen eri keinoin. Eri yritykset hoitavat sen omalla tavallaan. Osa takaisinmallintajista käyttää esimerkiksi UML-kaavioita ja toiset taas erilaisia kuvauskieliä tai vapaamuotoisia kaavioita. Vapaamuotoisiksi kutsutaan sellaisia kaavioita, jotka eivät noudata mitään tunnettua standardia tai suositusta.

Ohjelman ymmärtämiseen parhaalla mahdollisella tavalla tarvitaan kaikki mahdollinen saatavilla tai löydettävissä oleva tieto olemassa olevasta ohjelmistosta. Nämä muut vapaamuotoisiksi suunnitteluabstraktioiksi kutsutut tiedot eivät välttämättä ole niin muodollisia ja hyvin määriteltyjä kuin oikeat tuotokset ja dokumentaatio, mutta niistäkin saadaan tärkeää tietoa. Vapaamuotoisia suunnitteluabstraktioita voi löytyä muun muassa jonkun tietyn tahon kokousmuistiinpanoista tai muistioista. Tällaiset epäviralliset dokumentit voivat sisältää perustelut sille, miksi jokin tietty suunnitteluratkaisu on aikanaan tehty.

Kun tuotokset ja muut suunnitteluabstraktiot on kyetty löytämään, pitää tarkastella niiden ja ohjelmakoodin välisiä suhteita. Esimerkiksi halutaan tietää, mitä kohtaa ohjelmakoodissa kuvaa suunnitteluvaiheessa tehty tietovirtakaavio. Kun pystytään löytämään silta abstraktioiden ja ohjelmakoodin välille, voidaan luoda viitekehys, jonka sisällä pystytään järjestelemään niin ohjelmakoodin liittyvät tiedot kuin rakenteet, jotka auttavat ymmärtämään ohjelmakoodin yksityiskohtia.

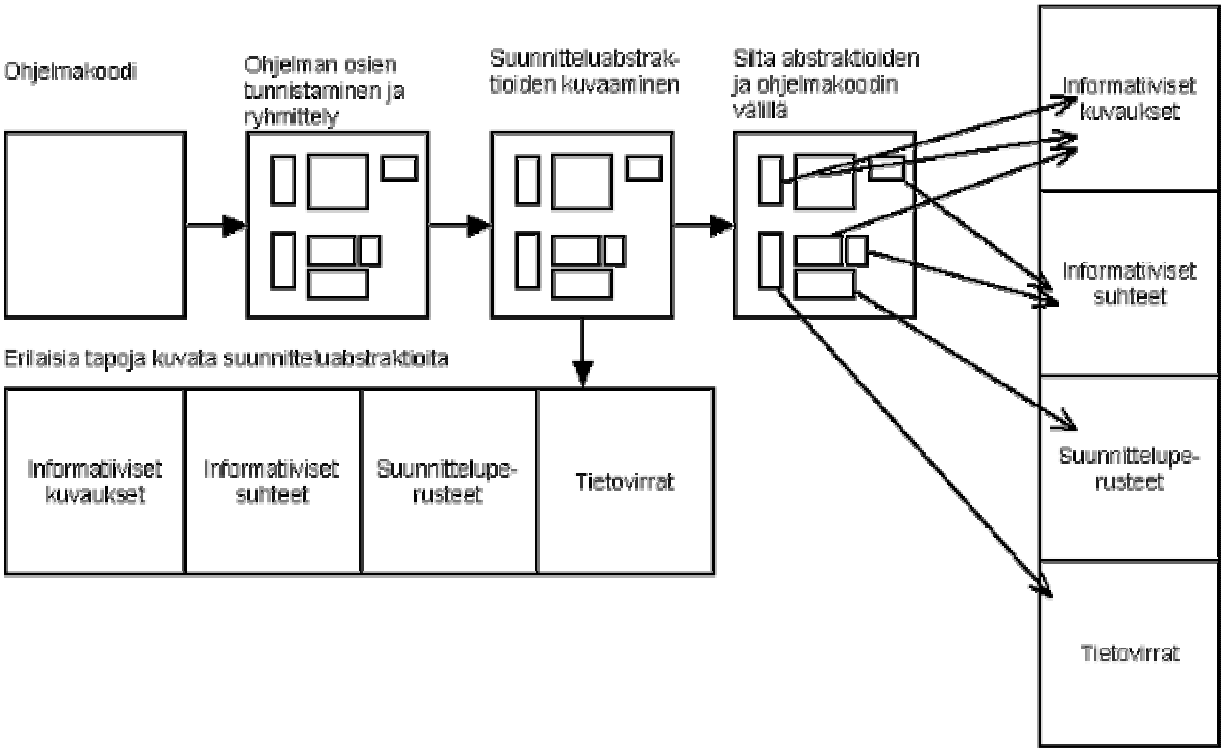
Toisen tavoitteen saavuttamisessa eli uudelleenkäytön tukemisessa voidaan käyttää seuraavanlaista menetelmää. Edellisessä vaiheessa löydettyjä suunnittelukomponentteja voidaan käyttää hyväksi esimerkiksi komponenttikirjastojen avulla. Tällöin joudutaan usein yleistämään komponentteja uudelleenkäyttömahdollisuuksien lisäämiseksi. Yleisesti suunnitteluratkaisun jäljittämisen avulla saatuja tietoja voidaan käyttää hyväksi, kun rakennetaan uudelleen joitakin samanlaisia komponentteja tai kun jäljitetään samankaltaisia komponentteja jostakin toisesta järjestelmästä.

Kolmatta tavoitetta eli tulosten soveltamista käytännössä voidaan hyödyntää siten, että etsitään samankaltaisuuksia jostakin toisesta järjestelmästä. Harvoin löydetään toisiaan täysin vastaavia malleja, mutta osittainenkin vastaavuus auttaa uudessa tilanteessa. Esimerkiksi työmäärää pystytään vähentämään osittaisen vastaavuuden avulla. Lisäksi löydettäessä uusista ohjelmista samankaltaisia malleja, voidaan luotua mallia laajentaa ja tehdä siitä entistä mukautuvampi.

3.5.2 Suunnitteluratkaisun jäljittämisen vaiheet

Suunnitteluratkaisun jäljittämisen alkaessa tekijöillä on yleensä jonkin asteinen tietämys tai olettama, millaisesta ohjelmasta ja sovellusalueesta on kyse. Tämän avulla suunnitteluratkaisun jäljittämistä tekevä asiantuntija pystyy asettamaan odotuksia ohjelmalle ja esimerkiksi sen rakenteelle. Muun muassa voidaan päätellä, millaisia osia tai toimintoja ohjelmasta löytyy.

Kuvassa 13 on esitetty Ted J. Biggerstaffin käsitys suunnitteluratkaisun jäljittämisen eri vaiheista ja niihin liittyvistä seikoista. Kuvatut vaiheet soveltuvat sellaisenaan etenkin C-kielen ymmärtämiseen. Muiden ohjelmointikielten tarkasteluun vaiheita tulee hieman muuttaa.



Kuva 13 Suunnitteluratkaisun jäljittämisen vaiheet Ted J. Biggerstaffin mukaan [Big89]

Ted J. Biggerstaff on esittänyt seuraavanlaisen esimerkin, jota seurataan tässä luvussa aina jokaisen suunnitteluratkaisun jäljittämisen vaiheen kohdalla. Alkutilanne on se, että asiantuntija tietää sovellusalueena toimivan ikkunointijärjestelmän samanaikaiset prosessit. Tästä hän pystyy olettamaan järjestelmän koostuvan ainakin prosessitaulusta, ikkunataulusta, ikkunoiden käsittelymoduulista ja prosessien käsittelymoduulista.

Ensimmäisessä vaiheessa asiantuntija luo itselleen mielikuvan siitä, millainen järjestelmä mahdollisesti on. Samalla tavoin hän rakentaa ilmentymät alijärjestelmille. Tällä tavoin pystytään luomaan yleiskuva järjestelmän mahdollisesta *arkkitehtuurista*. Järjestelmän arkkitehtuurilla tarkoitetaan järjestelmän ohjelmiston osia ja osien välisiä suhteita, riippuvuuksia sekä järjestelmälle asetettuja rajoituksia [Eer02]. Nämä kuvi-

tellut kuvaukset järjestelmästä voivat löytyä suorasti tai epäsuorasti järjestelmän koodista.

Suoria vastaavuuksia ohjelmakoodista voidaan löytää kielellisen mallintunnistuksen avulla. Kielellistä mallintunnistusta käytetään löytämään jonkun tietyn rakenteen erilaiset kirjoitusasut. Esimerkiksi prosessitaulun ilmentymiä voidaan etsiä koodista erilaisten prosessitaulua tai sen osia mahdollisesti tarkoittavien kirjainyhdistelmien avulla, kuten "pro", "tbl" ja "prc".

Epäsuoria vastaavuuksia taas voidaan löytää alirakenteiden ilmentymien kautta. Esimerkin kaltaisen järjestelmän alirakenteena voi toimia prosessitaulu. Prosessitaulukin voidaan jakaa pienempiin osiin, kuten esimerkiksi tilaan ja nimeen sekä prosessin sijaintiin.

Suureen järjestelmään voi olla vaikeaa sovittaa luotuja malleja. Monesti tuloksena on sellaisia positiiviselta vaikuttavia löydöksiä, jotka loppujen lopuksi paljastuvat vääriksi. Tästä syntyy turhaa ajanhukkaa. Näiden väärien positiivisten löydösten välttämiseksi voidaan eräänä keinona käyttää olemassa olevia tietoja samankaltaisista ohjelmista, samasta sovellusalueesta tai kyseessä olevasta ohjelmointikielestä. Eräissä ohjelmointikielissä voidaan olettaa, että tietyt määrittelyt löytyvät jostakin ennalta määrätystä kohdasta ohjelmakoodia. Esimerkiksi C-kielisessä ohjelmassa prosessitaulun määrittely löytyy oletettavasti otsikkotiedostosta (header file). Tällaisen tiedon perusteella haluttujen tietojen, tässä tapauksessa prosessitaulun määrittelyä koskevat tiedot, etsintä voidaan rajoittaa koskemaan vain tiettyjä tiedostoja ja siten voidaan löytää haluttu määrittelykoodi.

Suunnitteluratkaisun jäljittämisessä on hyödyllistä käyttää apuna automaattisia välineitä, sillä ne säästävät aikaa. Välineiden avulla voidaan etsiä esimerkiksi yhteyksiä määrittelykoodin ja alirakenteiden ilmentymien välille. Välineitä voidaan käyttää myös pelkästään määrittelykoodin etsimiseksi otsikkotiedostoista. Tulee kuitenkin huomioida, että välineiden avulla ei yleensä löydetä kuin osittaisia vastaavuuksia, ja tarkempi tutkiminen jää siten jäljittämistä tekevän henkilön tehtäväksi. Osittaiset vastaavuudet luovat kuitenkin hyvän pohjan suunnitteluratkaisun jäljittämiselle. Automatisoinnin avulla saatujen tietojen pohjalta on jäljittäjän hyvä jatkaa työtänsä.

3.6 UUDELLEENDOKUMENTOINTI

Uudelleendokumentaatiota pidetään vanhimpana ja yksinkertaisimpana käänteistekniikan muotona. Määritelmän mukaisesti *uudelleendokumentoinnissa* on kyse semant-

tisesti samantasoisen esityksen luomisesta jonkin materiaalin pohjalta tai entisen esityksen tarkistamisesta [ChC90]. Tämä tehdään koko ajan samalla abstraktiotasolla. Kuten nimikin jo kertoo; uudelleendokumentoinnissa luodaan joko kokonaan uusia dokumentteja olemassa olevasta järjestelmästä tai korjataan vanhoja dokumentteja ohjelman nykyistä muotoa vastaaviksi. Uudelleendokumentoinnin tarkoituksena on yksinkertaisesti saattaa järjestelmän dokumentaatiot siihen tilaan, missä niiden pitäisikin olla.

Uudelleendokumentointi ei välttämättä tarkoita koko dokumentaation luomista, vaan suppeassa muodossa sillä voidaan tarkoittaa pelkästään kommenttien lisäämistä ohjelmakoodiin. Uudelleendokumentointi voi olla myös esimerkiksi pseudokoodin luomista ohjelmakoodista [Har01]. Pseudokoodilla tarkoitetaan ohjelmakoodin esittämistä normaalia kieltä lähempänä olevassa muodossa.

Uudelleendokumentoinnin tarve tulee silloin, kun jostain syystä järjestelmän dokumentit ovat joko

- kadonneet,
- niitä ei ole ikinä ollut tai
- ne ovat jääneet päivittämättä.

Nämä ovat erittäin tyypillisiä ongelmia ohjelmistotuotannossa, sillä dokumenttien tekeminen tai niiden päivittäminen koetaan yllättävän usein turhaksi tai liian paljon resursseja vieväksi toiminnaksi. Monesti esimerkiksi ohjelmakoodin kommentointikin tehdään vasta jonkin aikaa ohjelman valmistumisen jälkeen ja silloin on jo voitu unohtaa, miksi jokin tietty ratkaisu on tehty. Tätä voi akateemisessa maailmassa verrata sellaiseen tilanteeseen, jossa väitöskirjaansa kirjoittava henkilö lisäilee lähdeviitteet vasta jonkin aikaa kirjoitusurakan päättymisen jälkeen.

Dokumentoinnin hyljeksiminen johtuu usein siitä, että dokumentteja laadittaessa tai ohjelmaa muutettaessa ei pystytä näkemään ajanmukaisten dokumenttien tärkeyttä ohjelman myöhemmän kehityksen ja ylläpidon kannalta. Kunnollisesti tehty dokumentaatio voisi toimia apuna myös uusien tuotteiden luomisessa. Dokumentoinnin tärkeys jää helposti taka-alalle etenkin silloin, kun samat henkilöt toimivat pitkään jonkun tietyn ohjelmiston parissa. He eivät tule välttämättä ajatelleeksi, että joskus jokin muukin taho saattaa joutua työskentelemään kyseisen ohjelmiston parissa sitä päivittäen, uudistaen tai muokaten.

Uudelleendokumentoinnista voidaan löytää kaksi erilaista suuntausta. Silloin, kun pyritään muuttamaan ohjelmakoodia pseudokoodiksi, on yleensä kyse *suunnittelupää-*

tösten tunnistamisesta ja silloin, kun tarkoituksena on rakentaa uudelleen ohjelmiston arkkitehtuuri, on kyse *rakenteellisesta uudelleendokumentoinnista*.

3.6.1 Suunnittelupäätösten tunnistaminen

Ohjelman tekovaiheessa tehdään mitä erilaisimpia suunnittelupäätöksiä [LeO90]. Tämän kaltaisia päätöksiä ovat esimerkiksi, miten mallintaa ratkaistavaa ongelmaa, mitä rajoituksia ohjelmointikieli ja laiteympäristö luovat sekä mitä ohjelmointikieltä milloinkin kannattaa käyttää. Suunnittelupäätökset ovat laadultaan erilaisia. Toiset vaikuttavat paljon ohjelman muihin osiin ja toiset taas ovat lähes riippumattomia muista päätöksistä. Ongelmana suunnittelupäätöksissä on se, että niitä hyvin harvoin dokumentoidaan perusteellisesti. Yleensä suunnittelupäätöksen seuraukset on nähtävissä vain lähdekoodissa. Tämän seurauksena ohjelman ylläpito ja uudistaminen vaikeutuvat. Näiden vaiheiden onnistumiseen tarvitaan tietämys ohjelman suunnittelupäätöksistä.

Ohjelmistotuotanto koostuu eri vaiheista, joihin liittyy kiinteästi useamman tyyppisiä suunnittelupäätöksiä. Erilaisia suunnittelupäätöksiä tehdään esimerkiksi seuraavissa ohjelmistonkehitysvaiheissa [LeO90]:

- kokoaminen ja osiin jakaminen,
- kapseloiminen ja limittäminen,
- yleistäminen ja erikoistaminen sekä
- esitysmuoto ja sen valinta.

Kokoamista tapahtuu esimerkiksi silloin, kun rakennetaan lauseke muuttujien ja funktioiden avulla. Toisaalta kokoamista on myös se, kun kootaan ohjelma komponenttien avulla. Ohjelman kokoaminen voidaan tehdä myös kirjastofunktioiden avulla.

Osiin jakaminen eli osittaminen on taas tyypillisin ohjelmistotuotantoprosessin aikana tehdyistä suunnittelupäätöksistä. Osittamisesta on kyse silloin, kun jaetaan jokin tehtävä, esimerkiksi laskenta, helpommin suoritettaviin ja ymmärrettäviin osiin esimerkiksi aliohjelmiksi. Näitä osia voidaan käyttää sitten kokoamalla hyväksi. Kokoamiseen ja osiin jakamiseen liittyviä suunnittelupäätöksiä tukevat ohjelmistossa käytettävät tieto- ja kontrollirakenteet.

Kapseloinnissa rajoitetaan pääsemistä käsiksi toteutuksen yksityiskohtiin. Tällaisten rajojen vetäminen kuuluu kiinteästi ohjelmien rakentamiseen. Kapselointi on suunnittelupäätös, jonka tarkoituksena on helpottaa ohjelman ylläpitoa ja ymmärtämistä. Tämä tapahtuu siten, että kapseloinnin avulla voidaan toteutusta muuttaa ilman, että kut-

surajapinnan muuttaminen tulee tarpeelliseksi. Kapseloinnin ajatus on tänä päivänä käytössä niin olio- kuin komponenttipohjaisessa ohjelmistotuotannossa, jossa kapseloinnin avulla pyritään parantamaan ohjelmien modulaarisuutta [Atk99].

Limittämistä voidaan pitää kapseloinnin vastakohtana. Limittämisessä on kyse siitä, että toimintoja kiedotaan yhteen ja tällä tavoin lisätään tehokkuutta. Esimerkiksi samassa silmukassa voi olla hyödyllistä laskea sekä taulukon suurin alkio että tämän kyseisen alkion indeksi. Limittämisen haittapuolena on se, että koodi voi tulla vaikeammin ymmärrettäväksi sekä ylläpito saattaa vaikeutua.

Yleistämistä voidaan tehdä oikein tapahtuvan parametrisoinnin avulla. Silloin suunnitteluvaiheessa tulee päättää, mitä kannattaa parametrisoida. Suunnittelupäätöksenä yleistämisessä on kyse siitä, että yleistämisen avulla toisaalta täytetään ohjelmalle asetetut vaatimukset, mutta toisaalta lievennetään joitakin ohjelmalle asetettuja rajoituksia. Esimerkiksi ohjelmalta voidaan vaatia, että se laskee yhteen tiettyyn lukujoukkoon kuuluvia lukuja. Tällöin voidaan käyttää yleistettyä funktiota, joka laskee yhteen myös muihinkin lukujoukkoihin kuuluvia lukuja.

Erikoistamisessa on kyse siitä, että ohjelman määrittelystä johdetaan rajoittuneempi vaihtoehto kuin yleistämisessä. Tällöin optimoidaan algoritmi sopimaan juuri kyseessä olevaan ohjelmointikieleen ja ongelma-alueeseen. Optimoinnilla parannetaan ohjelman suorituskykyä, mutta samalla ymmärrettävyys voi kärsiä. Voidaan ajatella, että yleinen komponentti on käyttökelpoisempi kuin erikoistunut, mutta toisaalta taas erikoistunut komponentti tuo ohjelmaan lisää tehokkuutta. Lisäksi erikoistuneita komponentteja on helpompaa testata kuin yleisiä. Näistä tekijöistä johtuen päätökset siitä, millaisia komponentteja käytetään, tulee tehdä huolellisesti ja tapauskohtaisesti. Erikoistamisesta tapahtuu silloin, kun päätetään, että rajoitettu taulukko on riittävä tapa pinon esittämiseksi tämän tyyppisessä tapauksessa.

Esitysmuotoa valittaessa mietitään, missä muodossa ongelman ratkaisu on parasta esittää. Esitysmuodon valitsemista ei tule sotkea edellä esitettyyn erikoistamiseen. Esitysmuodosta on kyse silloin, kun päätetään esittää pino kiinteämittaisen taulukon ja indeksimuuttujan avulla.

Kun suunnitteluratkaisut on tunnistettu ohjelmakoodista, tullaan seuraavan haastavan tehtävän eteen: löydetyt ratkaisut tulisi esittää sellaisessa muodossa, että ylläpitoon ja uudelleenkäyttöön erikoistuneet henkilöt pystyisivät hyödyntämään tietoja tehokkaasti. Esitystapa riippuu tiiviisti suunnitteluratkaisujen esitysmuodosta. Huono

esitysmuoto, jota ylläpitäjät eivät esimerkiksi ymmärrä, voi tehdä suunnitteluratkaisujen tunnistamisen hyödyttömäksi.

Suunnittelupäätösten avulla ohjelmistoja voidaan ymmärtää paremmin ja samalla helpottaa ylläpitoa ja uudelleenkäyttöä. Ihanteellinen tilanne olisi, jos suunnittelupäätökset löytyisivät alun perin dokumentoituna, mutta harvoin tilanne on sellainen. Jos suunnittelupäätökset kuitenkin saadaan selvitettyä tehokkaasti jälkikäteen, niiden avulla voidaan helpottaa huonosti dokumentoitujen ohjelmistojen uudelleenkäyttöä ja ylläpitoa.

3.6.2 Rakenteellinen uudelleendokumentointi

Ohjelmiston olemassa olevan arkkitehtuurin selvittämistä käänteistekniikkaa käyttäen kutsutaan rakenteelliseksi uudelleendokumentoinniksi [WoT95]. Rakenteellisen uudelleendokumentoinnin tuloksena saadaan hahmotettua kohteena olevan järjestelmän muoto sekä palautettua osa ohjelmiston arkkitehtuurisuunnittelutiedoista. Se ei itsessään sisällä ohjelmakoodin uudelleenrakentamista. Takaisinmallinnukseenhan ei kuulu ohjelmakoodin uudelleenrakentaminen, vaan takaisinmallinnuksessa on kyse pelkästään tietojen keräämisestä ja analysoinnista. Takaisinmallinnuksen avulla, jota siis rakenteellinen uudelleendokumentointi edustaa, mallinnettuja tietoja käytetään kuitenkin yleensä hyväksi ohjelmiston toteuttamiseen uudessa muodossa. Rakenteellisen uudelleendokumentoinnin avulla selvitettyjä tietoja käytetään hyväksi muun muassa arkkitehtuurin uudelleenrakentamisessa.

Rakenteellisen uudelleendokumentoinnin tarve syntyy silloin, kun ohjelmistoa on muutettu tai päivitetty ja nämä muutokset ovat jääneet kirjaamatta dokumentaatioon. Muutosten seurauksena ohjelmiston arkkitehtuuri on saattanut muuttua, jonka seurauksena olemassa oleva dokumentaatio ei enää vastaakaan olemassa olevaa arkkitehtuuria. Rakenteellisen uudelleendokumentaation avulla selvitetään tapahtuneet muutokset. Tämä tapahtuu yleensä vertaamalla dokumentoitua arkkitehtuuria olemassa olevaan arkkitehtuuriin. Rakenteellisessa uudelleendokumentoinnissa hyödynnetään lisäksi arkkitehtuuri- ja suunnittelumalleja (architectural and design patterns).

Rakenteellinen uudelleendokumentointi soveltuu etenkin laajojen järjestelmien ymmärtämiseen [WoT95]. Laajoissa järjestelmissä on erityisen tärkeää ymmärtää koko järjestelmän arkkitehtuurista rakennetta eikä vain jotakin yksittäistä algoritmia.

Rakenteelliseen uudelleendokumentointiin on kehitetty takaisinmallinnusympäristö nimeltä Rigi [WoT95]. Tästä ympäristöstä kerrotaan enemmän seuraavassa automaattisia apuvälineitä käsittelevässä luvussa.

4 Välineet käänteistekniikan tukena

Ohjelmistojen ymmärtämiseen eli siten käänteistekniikkaankin kuuluvat kiinteästi erilaiset apuvälineet (tools), menetelmät ja näiden kehittäminen. Välineet tukevat käänteistekniikan eri vaiheita ja ne yleensä auttavat tärkeän tiedon keräämisessä. Välineiden tarkoituksena on yleensä tuottaa abstraktiotasoltaan korkeamman asteista tietoa kohteena olevasta ohjelmistosta [ToP01]. Esimerkiksi osaa välineistä voidaan käyttää apuna analysoitaessa ohjelman koodia. Toisia taas käytetään, kun muodostetaan korkeamman tason kuvauksia ohjelmiston rakenteesta.

Näitä välineitä ovat olleet kehittämässä niin ohjelmistoteollisuus, kuten esimerkiksi Nokia, kuin myös akateemiset tahot muun muassa University of Victoria Kanadassa ja Tampereen yliopisto sekä Tampereen teknillinen korkeakoulu täällä Suomessa [Sys00].

4.1 AUTOMATISOINTI JA KÄÄNTEISTEKNIIKKA

Tietotekniikan liiton Atk-sanakirja vuodelta 1999 määrittelee *automaation* seuraavasti: "Sellaisen tekniikan käyttäminen, jonka avulla toiminta tapahtuu ilman ihmisen ohjaavaa tai suorittavaa osuutta, tavallisesti suorittimen tai tietokoneen ohjauksessa" [Atk99]. *Automatisoinnissa* on taas kyse automaation toteuttamisesta. Automatisoinnista puhutaan paljon tämän päivän tietojenkäsittelytieteessä, niin testauksen kuin ohjelmien analysoinnin yhteydessä. Automatisoinnin tarkoituksena on helpottaa tehtävää työtä. Jos tarkastellaan automatisointia yleisesti, siihen ryhdyttäessä tulee huomioida useita seikkoja. Jotta automatisointi voidaan tehdä, tulee ensinnäkin olla olemassa manuaalinen tapa tehdä kyseinen työ. Esimerkiksi testauksen automatisointiin ei kannata lähteä, ellei ole olemassa toimivaa tapaa suorittaa prosessia manuaalisesti [Zam98]. Pitää siis olla olemassa menetelmä, jolla työ voidaan tehdä käsin. Joskus voi riittää myös tieteellinen malli, jonka pohjalta voidaan vahvasti olettaa, että automatisointi olisi tehtävissä kyseisen mallin avulla. Toiseksi automatisoinnissa tulee huomioida sen kannattavuus: automatisointi kannattaa harvoin silloin, kun sillä saavutettava taloudellinen tai ajallinen hyöty on pieni. Kolmanneksi automatisoinnissa tulee ottaa huomioon se, onko olemassa valmiita välineitä automatisoinnin suorittamiseksi. Jos välineitä ei ole, tulee tarkoin miettiä, kannattaako ryhtyä itse rakentamaan sellaista.

Neljänneksi pitää myös ottaa huomioon, onko automatisoitava tehtävä sellainen, joka toistuu usein nyt ja tulevaisuudessa.

Automatisoidun käänteistekniikan tarkoituksena on helpottaa ohjelmiston ymmärtämistä sekä ylläpidettävyyttä. Välineiden avulla voidaan ensinnäkin erottaa järjestelmän abstraktiot sekä suunnitteluelementit ohjelman lähdekoodista. Tämän jälkeen näiden eroteltujen abstraktioiden ja elementtien avulla voidaan tuottaa toiminnallisia kuvauksia järjestelmän rakenteellisista osista toisistaan eroavilla abstraktiotasoilla sekä oliopohjaisia malleja ja dokumentaatioita järjestelmästä [RaC99]. Automatisoinnista ei kuitenkaan saa tulla itsetarkoitus; kaikkia vaiheita ei ole syytä täysin automatisoida. Se ei ole edes mahdollista. On seikkoja, joiden havaitsemiseen tarvitaan tietokoneiden ulottumattomissa olevaa inhimillistä ajatteluky kyä.

4.2 VÄLINEITÄ, MENETELMIÄ JA YMPÄRISTÖJÄ

Erilaisia välineitä ja niistä koostuvia ympäristöjä tai työvälineistöjä on kehitetty käänteistekniikan tueksi. Ne keskittyvät käänteistekniikan eri sovellusalueisiin sekä ne tukevat eri ohjelmointikieliä. Yleensä välineet ovat keskittyneet esimerkiksi jonkin tietyn tyyppisen tiedon suodattamiseen (extraction) ohjelmakoodista, mutta niillä ei voida onnistuneesti suodattaa kokonaisen ohjelmiston lähdemallia (source model) [KaC97]. Yhdellä välineellä ei voida tutkia kaikkia mahdollisia olemassa olevia ohjelmointikieliä tai ratkaista kaikkia takaisinmallinnustehtäviä. Lisäksi vain harvoilla välineillä voidaan tukea dynaamista takaisinmallinnusta, välineet ovat yleensä keskittyneet staattiseen takaisinmallinnukseen.

Käänteistekniikkaan tarkoitetut ympäristöt tai työvälineistöt, esimerkiksi Rigi (luku 4.2.4.) ja Dali (luku 4.2.6), on useimmiten suunniteltu helposti muunneltaviksi ja täydennettäviksi eli käyttäjä voi muokata näitä kuhunkin tilanteeseen sopivaksi. Ne sisältävät useita välineitä tai tekniikoita, joiden avulla voidaan saada paljon luotettavampia takaisinmallinnustuloksia kuin vain yhtä välinettä tai tekniikkaa käyttämällä.

Staattisen takaisinmallinnuksen välineet ovat yleensä keskittyneet luomaan graafisia esityksiä ohjelman lähdekoodista suodatetusta tiedosta [Sys00]. Staattiseen takaisinmallinnukseen keskittyvät välineet käyttävät tiedon suodattamisessa yleensä hyväkseen jäsentimiä tai sitten erilaisia sanastollisia (lexical) tekniikoita [KaC97]. Sanastollisia tekniikoita käyttävät välineet ovat useimmiten helpommin muunneltavia kuin jäsentimiä hyödyntävät, mutta toisaalta ne ovat tehottomampia. Osa staattisen takaisinmallinnuksen välineistä sisältää tilan, jossa suodatetun tiedon pohjalta luotuja

näkymiä voidaan muokata. Näkymiä voidaan esimerkiksi täydentää sellaisella tiedolla, jota ei ole kyetty automaattisesti löytämään. Lisäksi eräät välineet tukevat korkeamman tason mallien, esimerkiksi järjestelmäarkkitehtuurin, luomista kohteena olevasta ohjelmistosta. Tällainen väline on esimerkiksi Rigi.

Staattiseen takaisinmallinnuksen tarkoitettujen välineiden pohjalla käytetään usein hyväksi ohjelmointikielten rakenteen ja erilaisten mittareiden (meters) tuntemusta. Mittareita käytetään yleensä laadunvarmistuksen apuvälineinä. Takaisinmallinnuksessa mittareiden avulla voidaan esimerkiksi löytää ohjelmiston monimutkaisimmat rakenteet tai osat. Lisäksi mittareita voidaan käyttää erilaisten suunnitteluvirheiden löytämiseen. Eräät mittareista auttavat selvittämään ohjelmiston monimutkaisuutta tai uudelleenkäytettävyyttä. Esimerkki tällaisesta on mittari, joka tutkii oliopohjaisen ohjelmiston perintään liittyvää hierarkiaa.

Dynaamisen takaisinmallinnuksen välineet käyttävät hyväkseen erilaisia viestinvälityskaavioita kuvaamaan ohjelmiston ajonaikaista käyttäytymistä. Ohjelmistoissa tapahtuu paljon erilaista viestinvälitystä lyhyenkin käytön aikana eli ohjelmiston ajonaikaisesta käyttäytymisestä kertyy paljon tietoa. Tämän vuoksi suurin haaste dynaamisten takaisinmallinnusvälineiden rakentamisessa on ratkaisun kehittäminen löydetyn tiedon hallintaan ja käsittelyyn.

Välineiden avulla löydettyä tietoa voidaan käyttää hyväksi erilaisiin tarkoituksiin, esimerkiksi uudelleenkäyttöön sekä projektinhallintaan. Samoin tätä tietoa voidaan käyttää järjestelmän laadun parantamiseen. Esimerkiksi eräät takaisinmallinnusvälineet tähtäävät arkkitehtuurien uudelleenrakentamisessa tarpeellisen tiedon muodostamiseen. Lisäksi saatujen tietojen avulla voidaan tehostaa laadunvalvontaa kuten esimerkiksi testausta. Kuitenkin tulee muistaa, että takaisinmallinnuksen suorittamiseen tarvitaan myös manuaalista työtä; pelkät välineet eivät riitä onnistuneen lopputuloksen tuottamiseksi.

Takaisinmallinnusvälineet ja menetelmät ovat eläneet jo jonkin aikaa eräänlaista murroskautta [Sys00]. Tämä johtuu uusien ohjelmointikielten esiinmarssista, jonka seurauksena tulevaisuuden perinneohjelmat on kirjoitettu oliopohjaisilla kielillä, kuten Java ja C++, kun taas tämän päivän perinneohjelmat ovat yleensä Cobol- tai C-pohjaisia. Tästä johtuen nykyisin kehitettävät menetelmät ja välineet keskittyvät yhä useammin oliopohjaisten ohjelmistojen takaisinmallintamiseen. Lisäksi tulevaisuuden perinneohjelmat tulevat olemaan komponenttipohjaisia nykyisten ollessa monoliittisia

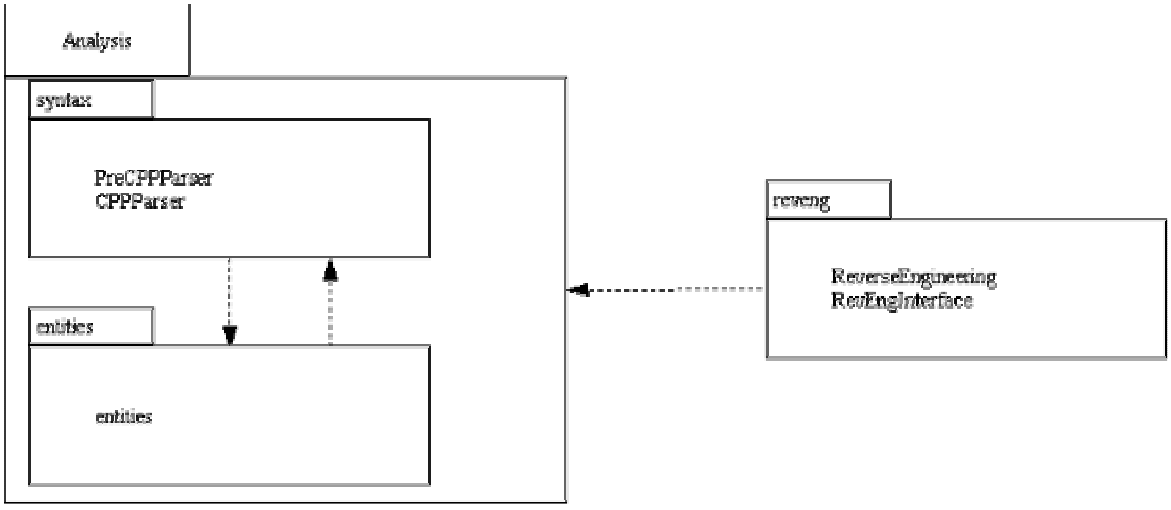
sovelluksia. Tästä voidaankin vetää se johtopäätös, että takaisinmallinnusta ja sen erilaisia tekniikoita tulee kehittää jatkuvasti muun tietojenkäsittelytieteen kehittyessä.

Välineitä ovat kehittäneet niin akateemiset tahot kuin yrityksetkin. Osa välineistä on tietenkin kehitetty yritysten ja yliopistojen yhteistyönä. Esimerkiksi Nokia ja Tampereen yliopistot ovat tehneet tällä saralla yhteistyötä kehittäessään muun muassa SCED-välinettä [KoM96]. Microsoft on taas kehitellyt omiin tuotteisiinsa takaisinmallinnusominaisuuksia [Ste01]. Näillä tuotteilla voidaan takaisinmallintaa Microsoftin kehittämiä ohjelmointikieliä.

Takaisinmallinnukseen käytettäviä välineitä löytyy lukuisia. Yksinkertaisella Internet-haulla on löydettävissä kymmeniä, ellei satoja erilaisia. Suurin ongelma tässä välineviidakossa on varmasti se, kuinka löytää joukosta sopivin väline. Seuraavaksi esitellään lyhyesti muutamia erilaisia ja eri tahojen tekemiä välineitä sekä menetelmiä ja ympäristöjä.

4.2.1 RevEng

Takaisinmallinnusväline RevEng on kehitetty tuottamaan UML:n mukaisia luokkakaavioita C++-ohjelmakoodista suodatettujen tietojen perusteella [ToP01]. Väline on kehitetty yhteistyössä sveitsiläisen CERN-tutkimuskeskuksen ja italialaisen ITC-irst-tutkimuskeskusen välillä.



Kuva 14 RevEng-takaisinmallinnusväline [ToP01]

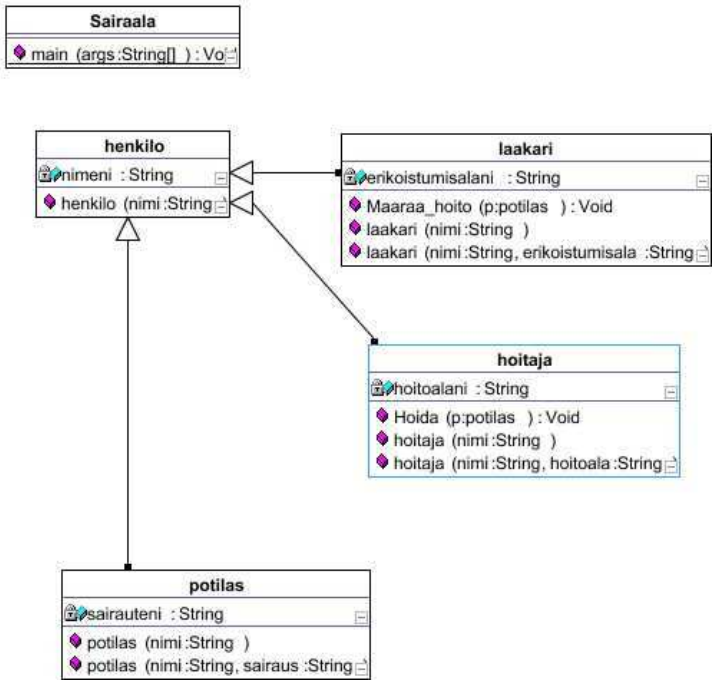
RevEng koostuu kahdesta pääosasta; *analysis*-pakkauksesta, jonka tarkoituksena on analysoida C++-koodia ja *reveng*-pakkauksesta, joka mallintaa luokkakaaviot analysoidun tiedon perusteella (Kuva 14). Analysis-pakkaus sisältää *entities*- ja *syntax*-alipakkaukset. Näistä ensimmäinen koostuu C++-entiteeteistä eli -olioista (entity) ja

toinen taas sisältää syntaksin analysoimiseen tarvittavat jäsennin-luokat. RevEng-pakkaus sisältää kaksi luokkaa; ReverseEngineering ja RevEngInterface. Näistä ensimmäinen käyttää hyväkseen jäsennyksen aikana syntyneitä olioverkkoja (net of objects) ja poimii sieltä luokkakaavioiden luomiseen tarvittavat entiteetit. Tällaisia ovat esimerkiksi luokkiin, parametreihin ja metodeihin liittyvät entiteettetit. Näiden avulla luodaan tiedot luokkakaaviosta ja välitetään ne DOT-välineelle luokkakaavion lopullisen ilmiasun laskemista varten. Tämän jälkeen RevEngInterface-luokka mahdollistaa kaavion näyttämisen.

RevEng-välineen avulla luodut luokkakaaviot on todettu melko yhtäpitäviksi todellisuuden kanssa. Tämä johtuu siitä, että RevEng tarkastelee löytämiensä olioiden tyyppejä sekä luokkien sisäisiä suhteita. Takaisinmallinnuksen avulla tuotettujen UML-kaavioiden luotettavuuden kasvaessa takaisinmallinnuksen käyttäminen lisääntyy eri ohjelmistotuotannon vaiheissa. Samalla kasvavat tietenkin ohjelmistojen ymmärryksen taso ja sen seurauksena ohjelmistojen luotettavuus ja muut laatutekijät lisääntyvät.

4.2.2 FUJABA

FUJABA on lyhennys sanoista From UML to Java And Back Again. Sen nimi kuvaakin erinomaisesti, mistä kyseisessä ympäristössä on kyse. FUJABA on Paderbornin yliopistossa kehitetty prototyyppi ympäristöstä, joka tukee sekä takaisinmallinnusta, että etenevää ohjelmistotuotantoa [Fuj03]. Ympäristön avulla voidaan tuottaa UML-kaavioiden perusteella runko ohjelman lähdekoodille ja taas lähdekoodin perusteella FUJABA tuottaa rungon luokka- ja aktiviteettikaavioille. FUJABA:n tuottamat luokkakaaviot (Kuva 15) ovat melko selkeitä, aktiviteettikaaviot taas ovat hieman epäselviä ja puutteellisia. Kaavioiden lisäksi FUJABA:n tarkoituksena on löytää suunnittelumalleja lähdekoodista.



Kuva 15 FUJABA-välineen luoma luokkakaavion runko luvussa 3.2 esitetystä esimerkkikoodista

FUJABA-ympäristö on alun perin kehitetty tukemaan molempiin suuntiin menevää ohjelmistotuotantoa (round-trip engineering). Kyseistä ympäristöä on hyödynnetty tämän lisäksi esimerkiksi hajautettujen järjestelmien tutkimuksessa. Lisäksi FUJABA-ympäristöä on käytetty materiaalina tutkittaessa myöhemmin esiteltävän Shimba-ympäristön käyttökelpoisuutta.

4.2.3 SCED

SCED-väline on alun perin kehitetty Tampereen yliopiston ja Tampereen teknillisen korkeakoulun yhteistyönä [KoM96]. SCED on skenaarioiden eli tapahtumasekvenssikaavioiden käyttöä tukeva työkalu. Sen avulla voidaan tuottaa tilakaavioita, jotka hyväksyvät annetut skenaariot eli tapahtumasekvenssikaaviot sekä skenaarioita, jotka ovat annettujen tilakaavioiden mukaisia. Tämän lisäksi SCED-väline asettelee ja sieventää tilakaaviot UML:n pohjalla olleen OMT-notaation mukaisesti. Välineen alkuperäinen tarkoitus oli automatisoida skenaarioita käyttävää oliopohjaista ohjelmistotuotantoa.

SCED koostuu skenaarioeditorista ja tilakaavioeditorista kuten monet muutkin oliopohjaiset CASE-välineet. Se eroaa kuitenkin muista oliopohjaisista CASE-välineistä skenaariota ja tilakaavioita integroivan osioidensa vuoksi. Skenaarioista

tilakaavioita luovaan osaan on sovellettu Biermannin ja Krishnaswamyn BK-algoritmia, jonka avulla voidaan automaattisesti luoda ohjelma, kun ensin tiedetään ohjelman suorittamien perustoimenpiteiden jonot tarpeeksi monissa esimerkkiajoissa [KoM96]. Tämän lisäksi tulee tuntea perustoimenpiteiden suoritusten välillä vallitsevat muuttujien arvoja koskevat ehdot. Algoritmin tuloksena syntyy vuokaaviona esitetty ohjelma, joka muistuttaa läheisesti tilakaaviota.

SCED-välineen [Sce98] kokeilun tuloksena voidaan sanoa, että sillä on melko helppoa luoda tapahtumasekvenssikaaviota ja tämän jälkeen generoida jostakin tapahtumasekvenssikaavion luokasta tai oliosta tilakaavio. Tilakaavio sisältää riittävästi tietoa siinä kuvattavan luokan tai olion tilasiirtymistä. Esimerkki SCED-välineen käytöstä ja käytön tuloksista esitetään luvussa 5.8.6.

Tilakaavioista saadaan luotua melko täydellisiä skenaariokaavioita. Tässä pitää huomioida se, että skenaariokaaviossa tuodaan esille myös sellaisia tahoja, joiden käyttäytyminen on jäänyt määrittelemättömäksi. Monesti tällainen tilakaaviota vaille jäävä taho on käyttäjä.

Kokeilun perusteella tämäkin ominaisuus on melko helppokäyttöinen. Yhdistelemällä useamman tilakaavion tietoja saadaan käytännöllisiä tapahtumasekvenssikaavion runkoja. SCED-väline toimii kuitenkin paremmin toiseen suuntaan tapahtuvassa mallinnuksessa.

Skenaariokaavioita voidaan myös luoda tai niitä täydentää SCED-välineen ulkopuolisen syötteen avulla esimerkiksi graafisen käyttöliittymän tai ohjelman lähdekoodin perusteella. Tämän vuoksi SCED-väline soveltuukin tekijöidensä mielestä myös takaisinmallinnuksen apuvälineeksi. SCED-välinettä onkin käytetty Nokialla tukemaan skenaarioihin perustuvaa oliopohjaista ohjelmakehitystä.

4.2.4 Rigi

Puoliautomaattinen Rigi-ympäristö on kehitetty Rigi-tutkimusprojektissa helpottamaan rakenteellisen tiedon ymmärtämistä [WoT95]. Rigi-välineen voidaan ajatella koostuvat kahdesta erityyppisestä osiosta. Sillä voidaan tunnistaa ohjelmiston osasia ja niiden välisiä suhteita ja toisaalta sen avulla voidaan erottaa ohjelmistosta suunnitteluinformaatiota ja järjestelmän abstraktioita. Näitä varten Rigi tarjoaa kolmenlaisia takaisinmallinnuksessa apuna käytettäviä välineitä:

1. Jäsentäjä (parser), joka tukee perinnejärjestelmille tyypillisiä imperatiivisia ohjelmointikieliä, kuten Cobolia ja C:tä. Uusimmat Rigi-versiot tukevat myös

Java-ohjelmointikieltä. Rigi-järjestelmässä on myös LATEX-jäsentäjä, jota voidaan käyttää apuna dokumentaation ymmärtämisessä. Jäsentäjä on väline, joka jakaa ohjelman kieliopin mukaisiin rakenteisiin.

2. Kuvauskanta (repository), jonne voidaan tallentaa lähdekoodista suodatettu tieto, on tietokanta, joka sisältää tietojärjestelmien kuvauksia.
3. Interaktiivinen kaavioeditori, jonka avulla voidaan käyttää hyväksi ohjelmistosta saatavia kuvauksia.

Rigi-järjestelmä keskittyy perinnejärjestelmien takaisinmallinnuksessa seuraavanlaisten vaatimusten täyttämiseen [WoT95]. Vaatimukset esitetään tekstissä kursivoiduin kirjaimin. Ensinnäkin Rigin avulla pyritään tuottamaan *dynaamisia kuvauksia* järjestelmästä. Toiseksi Rigi pyrkii olemaan mahdollisimman *joustava*. Kolmanneksi Rigi on pyritty rakentamaan sellaiseksi, että tärkeät päätökset jäävät sitä käyttävän analysoijan tehtäviksi, se on siis *ihmiskeskeinen* järjestelmä. Takaisinmallinnusprosessissa täytyy muistaa se, ettei kaikkea voi automatisoida. Neljänneksi Rigi esittää järjestelmästä näkökulmaltaan *erilaisia näkymiä*, joita tarvitaan myöhemmässä tutkimuksessa. Viidenneksi Rigissä on pyritty huomioimaan *skaalautuvuus* eli järjestelmän tulisi pystyä mukautumaan tietomäärän muutoksiin.

Rigi-ympäristön käytöstä on havaittu seuraavana esitettäviä hyötyjä [WoT95]. Rigi-järjestelmän avulla ohjelmiston ylläpitäjät saavat visuaalisen ja siten myös konkreettisen kuvauksen ylläpitämänsä järjestelmän loogisesta rakenteesta. Aiemmin heillä on ollut käytössään vain mielikuva kyseisestä rakenteesta. Tämän lisäksi Rigin luomat näkymät korostavat niitä ohjelmiston osa-alueita, jotka tarvitsevat enemmän huomiota osakseen; esimerkiksi keskeisiä ohjelmisto-osia, joilla on suuri määrä riippuvaisuuksia. Rigin esille tuomat näkymät ohjelmistosta tuovat ohjelmiston tarkasteluun ja ylläpitoon objektiivisen näkökulman aiemman subjektiivisen näkökulman lisäksi tai sijasta. Tämä johtuu siitä, että näkymät pohjautuvat ohjelmiston lähdekoodiin, eivätkä vanhoihin, päivittämättömiin ohjelmistodokumentteihin, kuten aiempi subjektiivinen näkemys. Lisäksi Rigin luomat näkymät ohjelmiston rakenteesta tekevät rakenteen ymmärrettäväksi myös muille ulkopuolisille tahoille, ainakin kokeneille ohjelmistojen analysoijille.

Rigi-ympäristöä on käytetty hyväksi erilaisten järjestelmien, esimerkiksi Cobol-kielisen potilastietojärjestelmän ja C-kielisen hiukkaskiihdyttimen valvontajärjestelmän, rakenteellisessa uudelleendokumentoinnissa [WoT95].

Rigi-järjestelmän muunneltavuuden, laajennettavuuden ja monikäyttöisyyden ansiosta sitä ja sen pohjalla olevia tietoja, on käytetty hyväksi muissa takaisinmallinnusvälineissä ja -menetelmissä, kuten kappaleessa 4.2.6 esiteltävässä Dali-työvälineistössä, kappaleessa 4.2.7 esiteltävässä Shimba-ympäristössä ja kappaleessa 4.2.8 esiteltävässä ARM-menetelmässä.

4.2.5 inSight

Takaisinmallinnusväline inSight pyrkii auttamaan ohjelmiston osasten välisten suhteiden selvittämisessä [RaC99]. Se tukee C-, C++- ja Protel-ohjelmointikieliä. inSight-välineen tarkoituksena on auttaa ohjelmistoarkkitehtejä ja -kehittäjiä ymmärtämään olemassa olevaa ohjelmistoa ja siten helpottaa heidän koko ohjelmiston elinkaaren ajan kestävää kehitys- ja ylläpitotyötään. inSight takaisinmallintaa ohjelmiston automaattisesti pohjautuen sekä ohjelmiston staattiseen eli rakenteelliseen että dynaamiseen eli käytökselliseen puoleen.

inSight etsii ja erottelee epäsuoria riippuvuuksia ohjelmiston entiteettien välillä. Tähän käytetään tietovirta-analyysiä. Toiseksi inSight etsii ja erottelee dynaamisen rakenteen esittämiseen tarvittavia suhteita prosessifunktioiden välillä. Dynaamisen rakenteen esittäminen kaavioiden avulla on tärkeää, jotta voidaan ymmärtää, mitkä funktiot toimivat missäkin prosessissa, kun ohjelmaa suoritetaan.

inSight-välinettä voidaan käyttää hyväksi ohjelmiston vahvuuksien ja heikkouksien määrittämisessä, ohjelmakoodin ja arkkitehtuurin uudelleenkäytettävyyden selvittämisessä sekä myös ohjelmakoodin tarkastuksissa (inspection). Eli välineen tulisi parantaa laadunvalvontaa ja helpottaa ohjelmistojen arviointia sekä mittausta.

inSight-välinettä on käytetty ohjelmistokehityksen apuna telekommunikaatioon ja tietoliikenneverkkoihin erikoistuneessa Nortelissa, jonka järjestelmät ovat Protel-, C- ja C++-pohjaisia [RaC99]. Sieltä saadun palautteen mukaan inSight on helpottanut ohjelmistojen ymmärtämisessä ja ohjelmistojen jatkuvassa laadunparantamisessa. Tämän lisäksi suunnittelijoiden koulutusajat heille uudenlaisiin tehtäviin ovat lyhentyneet inSight-välineen käyttöönoton jälkeen.

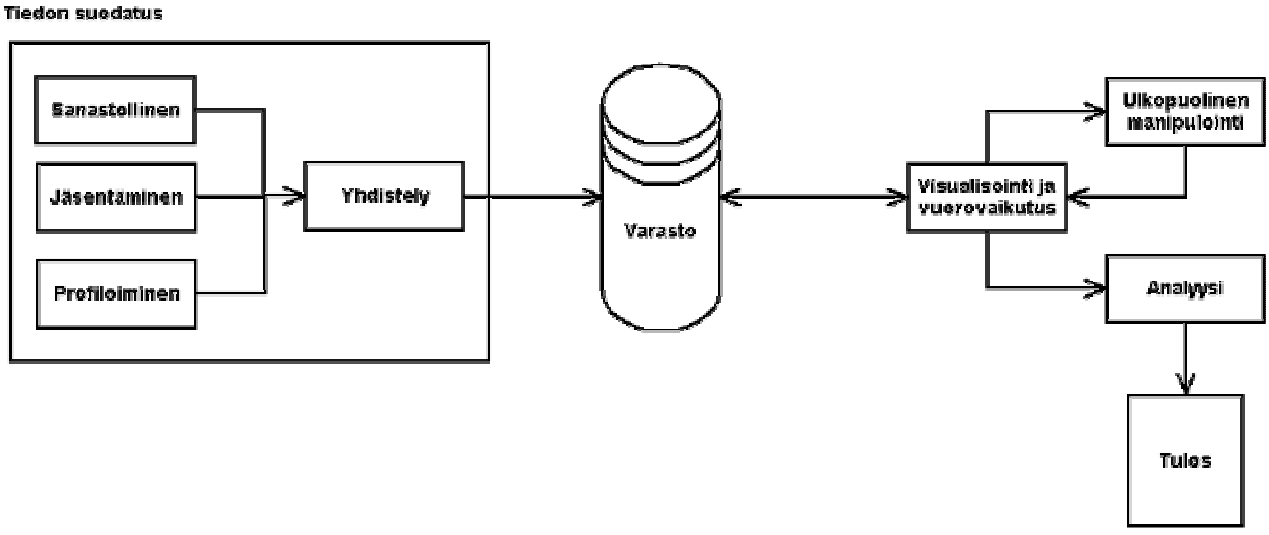
4.2.6 Dali

Dali on työvälineistö (workbench), jonka avulla voidaan ymmärtää ohjelmiston toteutettua arkkitehtuuria [KaC97]. Dali on kehitetty avoimeksi työympäristöksi, jonka kukin käyttäjä voi muokata haluamaksensa. Dalin tarkoituksena on auttaa käyttäjää

määrittelemään arkkitehtuurimallit ja sen jälkeen liittämään lähdekoodista saatu tieto näihin malleihin. Eli Dali-väline tukee arkkitehtuurien analysointia ja sitä taas käytetään apuna arkkitehtuurin uudelleenrakentamisessa (recontstruction) ja uudelleendokumentoinnissa. Dali-välineen voidaan ajatella soveltuvan erittäin hyvin etenkin rakenteiseen uudelleendokumentointiin. Dali-työvälineistössä on hyödynnetty osaa Rigi-välineestä.

Kazmanin ja Carrieren mukaan Dali-välineistössä yhdistyy kolme erilaista tekniikkaa arkkitehtuurinäkymän uudelleenrakentamiseksi (Kuva 16).

1. Sen avulla voidaan löytää arkkitehtuurisia osia ohjelmiston lähdemateriaalista, esimerkiksi lähdekoodista.
2. Dalia voidaan käyttää yhdistämään toteutettu arkkitehtuuri, joka saadaan selville ensimmäisen tekniikan avulla, suunniteltuun arkkitehtuuriin käyttäjän määrittelemien arkkitehtuurimallien (architectural pattern) avulla.
3. Dalin avulla voidaan visualisoida toisen vaiheen aikana syntynyt arkkitehtuuri, eli lähdekoodista poimittu tieto, joka on järjestelty arkkitehtuurimallien avulla, käyttäjän tarkastelunkohteeksi.



Kuva 16 Dali-työvälineistö Kazmanin ja Carrieren mukaan [KaC97]

Dali tarjoaa keinot, joiden avulla voidaan selvittää ohjelman konkreettista mallia (concrete model). Konkreettinen malli sisältää ohjelmiston elementtejä, esimerkiksi olioita ja muuttujia, elementtien välisiä suhteita, kuten oliokutsuja sekä lisäksi elementtien ja suhteiden attribuutteja, esimerkiksi olio O on tyyppiä T. Järjestelmän konkreettinen malli voi koostua useista erityyppisistä näkymistä; staattisen rakenteen, dynaamisen luonteen tai rakennusaikaisen (build-time) rakenteen kuvauksesta. Tässä

vaiheessa käytetään hyväksi kokoelmaa erilaisista tiedon suodatustekniikoista, joiden avulla pyritään kokoamaan totuudenmukainen kuva järjestelmän koostumuksesta.

Kun järjestelmän koostumus on selvitetty, se on järkevää tallentaa tietovarastoon. Dali-välineistössä tietovarasto koostuu kahdesta osasta: SQL (Structured query language)-tietokannasta, joka toimii mallin esisijaisena varastona, sekä sovelluskohtaista tiedostoista, joiden avulla voidaan siirtää tietoa eri välineiden välillä. Välineet voivat olla yhteydessä SQL-tietokantaan joko suoraan, rajapintojen kautta tai sovelluskohtaisten tiedostojen asettamien sääntöjen mukaisesti.

Tietovarastoon tallentamisen jälkeen voidaan Dalin avulla kartuttaa tietovarastoon tallennettujen suhteiden kokoelmaa. Tämä tapahtuu johtamalla uutta tietoa olemassa olevasta tiedosta. Dali tarjoaa joustavan ympäristön suhteiden johtamiselle.

Tämän jälkeen päästään käyttämään hyväksi Dalin vuorovaikutusominaisuuksia. Dalin avulla voidaan mallia muuttaa

- suoran manipuloinnin tai
- ulkoisen manipuloinnin ja analysoinnin sekä
- varaston synkronoinnin avulla.

Suoran manipuloinnin avulla kukin arkkitehti voi luoda oman tulkintansa suodatettujen tietojen pohjalta. Ulkoinen manipulointi ja analysointi tarjoaa mahdollisuuden suodatettujen tietojen käsittelyyn Rigi-välineen ulkopuolella. Malli viedään ensin välineen ulkopuolelle ja analysoidaan se siellä halutulla välineellä. Tämän jälkeen analysoitu malli saatetaan takaisin Rigin haltuun. Rigi tarjoaa työskentelyyn tarkoitetun kopion SQL-varastoon tallennetusta suoran manipuloinnin avulla luodusta mallista. Tämän vuoksi on tärkeää, että työskentelykopioon tehdyt muutokset voidaan tehdä myös alkuperäiseen malliin. Tätä ominaisuutta kutsutaan varaston synkronoinniksi. Varaston synkronointi on toteutettu RCL-koodin avulla, joka päivittää halutut muutokset SQL-varastoon. Muutokset voivat olla joko suoran manipuloinnin avulla tehtyjä tai sitten ulkoisen manipuloinnin ja analysoinnin tuloksia.

Dali-työympäristö on erittäin joustava. Dalin ainoa pysyvä rajoitus on tietokannan käyttö keskusmallin varastointiin. Teoriassa muut käytettävät välineet ja ominaisuudet ovat kunkin käyttäjän itsensä päätettävissä. Todellisuudessa Dalin tekijät ovat antaneet suosituksensa käytettävistä välineistä. Kuitenkin kukin käyttäjä voi päättää itse, haluaako käyttää suositettuja välineitä kussakin vaiheessa.

Dali-työympäristö helpottaa arkkitehtuurin suodattamista ja uudelleenrakentamista. Kuitenkin tulee huomioida, että tällainen työ ei onnistu pelkästään jollakin tietyllä välineellä, vaan onnistuneeseen lopputulokseen tarvitaan myös ihmisen kosketusta.

4.2.7 Shimba

Shimba on takaisinmallinnusympäristö, joka soveltuu Java-kielisten järjestelmien takaisinmallinnukseen. Tämä ympäristö yhdistää kaksi takaisinmallinnuksen välinettä, Rigi ja SCED, joiden käyttötarkoituksena on analysoida ja visualisoida olemassa olevan ohjelmiston staattista ja dynaamista toiminnallisuutta [SyK01]. Shimba tukee niin ohjelmakoodin tutkimista, visualisointia kuin analysointia.

Yleensä käänteistekniikassa käytettävät välineet tai ympäristöt ovat keskittyneet joko ohjelmiston käyttäytymisen tai rakenteen ymmärtämiseen, Shimba-ympäristön tarkoituksena on auttaa ymmärtämään, miten käyttäytyminen ja rakenne ovat riippuvaisia toisistaan. Tämä tapahtuu ensinnäkin jakamalla Rigi-välineen tuottamat staattista riippuvaisuutta kuvaavat kaaviot osiin SCED-välineen tuottamien sekvenssikaavioiden avulla. Toinen Shimba-ympäristön tarjoama keino on Rigin tuottamien kaavioiden käyttäminen SCED sekvenssikaavioiden luomisen ohjaajana ja samalla sekvenssikaavioiden abstraktiotason kohottajana. Shimban tarkoituksena on siis yhdistää staattisen ja dynaamisen takaisinmallinnuksen tekniikat ja käyttää niitä rinnakkain. Tällä tavoin takaisinmallinnuksessa suodatettua tietoa voidaan tutkia ja käyttää tehokkaammin hyväksi.

Shimba-ympäristön tarkoituksena on tarjota välineet kolmenlaisiin tehtäviin. Seuraavaksi ratkaistavat tehtävät esitetään kysymysten kautta. Ensimmäinen tehtävä käsittelee ohjelman ymmärtämistä, jolloin etsitään vastausta seuraaviin kysymyksiin:

1. Mitkä ovat ohjelmiston staattiset osaset ja kuinka ne ovat riippuvaisia toisistaan? Kuinka näitä osasia käytetään ohjelmiston ajonaikana?
2. Millainen on ohjelmiston korkeantason rakenne? Kuinka ohjelmiston korkeantason komponentit keskustelevat keskenään?
3. Toistuvatko tietyt yleiset käyttäytymismallit ohjelman ajonaikaisessa käyttäytymisessä? Mitä ne mallit ovat ja milloin ne esiintyvät?
4. Kuinka paljon eri komponentteja käytetään ohjelman ajonaikana?

Nämä kysymykset käsittelevät paljolti ohjelmistorakennetta eli ohjelmiston eri osia ja osien välisiä suhteita. Tätä aluetta voidaan kutsua ohjelman arkkitehtuurin selvittämiseksi eli ohjelman arkkitehtuurin uudelleenrakentamiseksi.

Toista tehtävää, johon Shimba on tarkoitettu, voidaan kutsua päämäärähakuiseksi takaisinmallinnukseksi. Tällöin pyritään vastaamaan muun muassa tämänkaltaisiin kysymyksiin:

1. Kuinka jonkin tietty komponentti toimii ja kuinka se vaikuttaa koko järjestelmän käyttäytymiseen?
2. Milloin poikkeus tai virhe tapahtui? Mitä tapahtui ennen sitä ja missä järjestyksessä?
3. Kuinka poikkeavan käytöksen aiheuttava komponentti on rakennettu?

Näillä kysymyksillä pyritään selvittämään eri osien toiminnallisuutta ja myös suunnitteluratkaisuja. Tässä vaiheessa on siis kysymys suunnitteluratkaisun jäljittämisestä sekä käytöksen selvittämisestä.

Kolmanneksi Shimbaa voidaan käyttää olion tai metodin täydellisen käytöksen tutkimiseen. Tällöin voidaan vastata seuraaviin kysymyksiin:

1. Millainen on olion tai metodin dynaaminen kontrollivirta ja kokonaisvaltainen käyttäytyminen?
2. Kuinka jokin tietty olion tila on saavutettavissa ja kuinka suoritus jatkuu sen jälkeen?
3. Mihin viesteihin olio reagoi tietyssä tilassa koko elinkaarensa ajan?
4. Mitä olion metodeja kutsutaan ohjelman suorituksen aikana?

Näiden kysymysten avulla selvitetään tarkemmin ohjelman staattista ja dynaamista käytöstä.

Shimba-ympäristön käyttökelpoisuutta on tutkittu jo aiemmin tässä luvussa esitellyn Paderbornin yliopistossa kehitetyn FUJABA-ympäristön avulla. FUJABA on laajahko ympäristö, joka koostuu noin 700 java-kielisestä luokasta. Tämän tutkimuksen tulokset osoittavat, että vaikkakin Shimban käyttäminen ja sen tuottamat tulokset eivät ole ongelmattomia, Shimba on käyttökelpoinen väline. Shimban tärkeimpänä ominaisuutena voidana pitää sen kokonaisvaltaisuutta. Sen avulla pystytään luomaan käsitys niin järjestelmän dynaamisesta kuin sen staattisesta puolesta.

4.2.8 ARM

ARM (Architecture reconstruction method) on puoliautomaattinen analyysimenetelmä arkkitehtuurien uudelleenrakentamiseksi. [GuA99] Arkkitehtuurin uudelleenrakentamisen tarve syntyy silloin, kun olemassa oleva arkkitehtuuri ei enää vastaa alkuperäistä, suunniteltua arkkitehtuuria. Järjestelmään tehdyt muutokset voivat aiheuttaa arkki-

tehtuurin muuttumisen. Arkkitehtuurin muuttumisen seurauksena järjestelmä ja sen rakenne voivat muuttua toiminnallisesti heikommiksi. Arkkitehtuurin uudelleenrakentamisen eräänä vaiheena voidaan pitää luvussa 3.6.2. esiteltyä rakenteellista uudelleendokumentointia.

ARM käyttää hyväkseen arkkitehtuurimallien (architectural patterns) tunnistamista. Menetelmä koostuu Guon, Atleen ja Kazmanin mukaan 4 eri vaiheesta:

1. Konkreettisen suunnitelman kehittäminen mallientunnistukseen (pattern recognition plan).
2. Lähdemallin (source model) löytäminen.
3. Arkkitehtuurimallien (pattern) ilmentymien tutkiminen ja arvioiminen.
4. Arkkitehtuurin uudelleenrakentaminen ja analysoiminen.

ARM-menetelmässä käytetään edellä esitettyjä Dali- ja Rigi-välinettä sekä Rigi-välineen pohjalla olevaa Rigi Standart Format -määrittelykieltä. Määrittelykieltä käytetään ensimmäisessä vaiheessa kuvaamaan malleja. Tämän jälkeen käytetään Dali-välinettä kääntämään arkkitehtuurimallien määrittelyt mallikyselyiksi (pattern query). Toisessa vaiheessa Dalia käytetään lähde-elementtien ryhmittelyyn. Kolmannessa vaiheessa Dalin avulla tutkitaan arkkitehtuurimallien ilmentymiä. Neljännessä vaiheessa taas käytetään Rigi-välinettä visualisoimaan saavutettuja tuloksia.

4.2.9 Microsoft Visio 2000

Visio on Microsoftin valmistama kaupallinen käänteistekniikkaa hyödyntävä väline. Se on väline Microsoftin omilla ohjelmointikielillä tehtyjen ohjelmien staattiseen takaisinmallinnukseen. Vision avulla voidaan takaisinmallintaa Microsoft Visual C++ 6.0, Microsoft Visual Basic 6.0 sekä Microsoft Visual J++ -ohjelmointikieliä [Ste01]. Visio-välineen takaisinmallinnusominaisuus tähtää ohjelman staattista rakennetta kuvaavien UML-kaavioiden esimerkiksi luokkakaavioiden tuottamiseen ohjelmakoodista.

Väline käyttää eri tapoja staattisen tiedon keräämiseen eri ohjelmointikielillä tehdystä ohjelmista. J++- ja Visual Basic -kielissä käytetään hyväksi kehitysympäristön oliomallia ja C++-kielisistä ohjelmista tieto kerätään selaustietoja sisältävistä tiedostoista (.BSC). J++-kielisistä ohjelmista saadaan kattavimmat kaaviot, sillä oliomalli, josta tieto kerätään on siinä kattavin. C++-kielen selaustietoja sisältävät tiedostot ovat tiedon keräämisen kannalta heikoimmat.

4.3 ARVIOINTIA VÄLINEISTÄ JA NIIDEN HYÖDYLLISYYDESTÄ

Kuten edellisestä luvusta 4.2 on tullut ilmi, käänteistekniikkaväline-termin alle mahtuu erilaisia ja eritasoisia välineitä. Yhteistä näille kaikille välineille on kuitenkin se, että niillä pyritään tavalla tai toisella helpottamaan ohjelmien ymmärtämistä. Karkeasti välineet voidaan jaotella lähdekoodista UML-kaavioita tuottaviin välineisiin ja lähdekoodista ohjelman rakennetta kuvaavia kaavioita luoviin välineisiin. Ensimmäiseen ryhmään kuuluvat edellä esitellyistä välineistä RevEng, Fujaba sekä Microsoft Visio 2000, jotka kaikki luovat UML-luokkakaavioita ohjelman lähdekoodin perusteella. UML-luokkakaaviosta ja sen rakenteesta on lisätietoa luvussa 5.4. Näiden välineiden lisäksi tähän ryhmään voidaan liittää SCED-väline, joka generoi UML-tapahtumasekvenssikaaviosta UML-tilakaavioita ja päinvastoin. Näitä UML-kaavioita tarkastellaan tarkemmin luvuissa 5.2 ja 5.3. Toiseen ryhmään kuuluvat taas muut edellä esitellyt välineet.

Ensimmäisen ryhmän välineet antavat ohjelmasta tietoa UML-kaavioiden muodossa. Oikeastaan jokainen ohjelmistoammattilainen kykenee tulkitsemaan UML-kaavioita ja siten ne auttavat ohjelmiston rakenteen ymmärtämisessä. Esimerkkinä tällaisesta välineestä voidaan tarkastella FUJABA-välinettä. FUJABA-välineen luomat luokkakaaviot esittävät selkeästi, mitä luokkia kyseisessä ohjelmassa on ja lisäksi sen, mitä ja minkä tyyppisiä attribuutteja sekä metodeja ja näiden parametreja kullakin luokalla on. Lisäksi FUJABA generoi luokkakaavioon automaattisesti periytymissuhteet (extends). FUJABA-väline luo tämän lisäksi aktiviteettikaavioiden runkoja jokaisen luokan jokaisesta konstruktorista sekä metodista.

FUJABA-välineen luomien luokka- ja aktiviteettikaavioiden avulla voidaan tarkastella ohjelman luokkien rakennetta sekä verrata välineen luomia kaavioita mahdollisesti dokumentaatiossa oleviin kaavioihin. Tällä tavoin voidaan esimerkiksi todeta, kuinka hyvin ohjelmakoodi on toteuttanut ohjelman pohjalla olleet vaatimukset ja suunnitelmat. Lisäksi nämä kaaviot selkeyttävät ohjelman luokka-koostumusta. Tätä tietoa voidaan käyttää hyväksi tarkasteltaessa laajaa ohjelmakoodia. FUJABA-välineen luokkakaavioiden generointi -ominaisuutta voidaan käyttää hyväksi ohjelmoinnin sekä tietojärjestelmien suunnittelun opetuksessa. Aloittelevat tietojenkäsittelyn opiskelijat pystyvät välineen avulla ymmärtämään yhteyden, joka vallitsee suunnitteluvaiheessa luotavien kaavioiden ja lopullisen ohjelmakoodin välillä. Kokemuk-

sen ja opiskelijakommenttien perusteella näiden kahden yhteyden hahmottaminen tuntuu olevan yllättävän hankalaa.

Tulee kuitenkin huomioida, että välineiden luomat kaaviot eivät ole täydellisiä. Esimerkiksi FUJABA ei pysty poimimaan ohjelman lähdekoodista kaikkea täydelliseen luokkakaavioon vaadittavaa informaatiota, kuten muita suhteita kuin periytymissuhde. Tällaisen tiedon poimimiseen tulee manuaalisesti tarkastella ohjelmakoodia sekä mahdollista dokumentaatiota.

Toisen tyyppisistä välineistä voidaan tarkastella Rigi-välinettä, joka keskittyy ohjelman rakenteen ymmärtämiseen eli rakenteellisen tiedon hankkimiseen ohjelmasta sekä ohjelman rakenteellisen kuvauksen luomiseen. Rigi-väline on oletettavasti erittäin onnistunut rakenteellisen tiedon kerääjä, sillä sitä on käytetty muiden välineiden ja ympäristöjen rakentamisessa. Se, kuten muutkin tämän tyypin välineet, luo graafisia kuvauksia ohjelman loogisesta rakenteesta. Esimerkiksi Java-kielisestä ohjelmasta se generoi kuvauksen, jossa näkyvät luokkien, parametrien, metodien ja attribuuttien väliset suhteet. Tällaisen kuvauksen perusteella ohjelman rakennetta pystytään selvittämään ja esimerkiksi uudelleendokumentoimaan ohjelmistoa. Monestihan ohjelman dokumentaatio on saattanut jäädä jalkoihin ohjelmiston laajentuessa. Tällöin sen uudelleendokumentointi on erittäin tärkeää. Lisäksi Rigin luomia kuvauksia voidaan käyttää hyväksi mietittäessä ohjelmiston tai sen osan uudelleenkäytettävyyttä. Jos ohjelmisto tai sen osa sisältää paljon riippuvuuksia, sen uudelleenkäyttö voidaan useimmiten unohtaa. Tietenkin Rigin luomat kuvaukset toimivat myös apuna ohjelman rakenteen parantamisessa eli kuvausten perusteella voidaan nähdä esimerkiksi mistä riippuvuuksista olisi hyvä päästä eroon. Rigi-väline sopii myös hyvin ohjelmoinnin opetukseen. Sen avulla voidaan havaita, kuinka pienessäkin ohjelmassa voi olla paljon erilaisia riippuvuuksia.

Rigi-välineen suurin heikkous on sen käytön opetteluun kuluva aika. Etenkin Windows-sukupolven henkilölle välineen käyttöönotto saattaa tuoda ongelmia. Lisäksi laajasta ohjelmasta syntyvä kuvaus on erittäin sekava, sillä laajassa ohjelmistossa on tietenkin paljon erilaisia rakenneosia ja näiden välisiä riippuvuuksia.

Välineet ja niiden käyttö ei ole ongelmatonta, kuten edellisissä esimerkeissä on tullut ilmi. Eräs tyypillinen välineiden käyttöä rajoittava tekijä on niiden kieliriippuvuus. Suurin osa välineistä tukee vain yhtä tai kahta ohjelmointikieltä. Jos siis ollaan tekemisissä järjestelmän kanssa, joka koostuu eri kielillä toteutetuista osista, tarvitaan käyttää useampia käänteistekniikkavälineitä. Lisäksi päätöksen tekeminen siitä, mikä

väline on toimivin ja kannattavin aina käsillä olevassa tapauksessa voi muodostua hankalaksi. On pidettävä mielessä se, että välineiden käytöllä tulee saavuttaa mitattavissa olevaa hyötyä.

Välineitä tulisi kehittää monipuolisempaan suuntaan siten, että samalla välineellä olisi mahdollista luoda niin UML-kuvauksia kuin myös ohjelmiston rakenteen kuvauksia. Lisäksi olisi tärkeää, että välineet rakennettaisiin tukemaan useampia ohjelmointikieliä. Esimerkiksi siten, että välineen hankkija voisi päättää, minkä kaikkien ohjelmointikielten tuen hän haluaa välineeseensä.

Yleisesti ottaen voidaan kuitenkin sanoa, että välineet tarjoavat avun ohjelmistojen tarkasteluun ohjelmistotuotantoprosessin eri vaiheissa. Välineiden avulla voidaan saada nopeasti kuvaus ohjelmiston koostumuksesta ja siten ohjelmakoodin manuaaliseen selaamiseen ja selvittelyyn kuluva aika säästyy. Lisäksi väline on tarkempi kuin ihminen; se ei jätä välistä yhtäkään koodiriviä, ellei sitä käsketä niin tekemään.

Kuitenkaan pelkästään välineillä ei tehdä ihmeitä. Prosessi, jossa välinettä käytetään, koostuu myös muistakin tekijöistä, kuten esimerkiksi siinä työskentelevistä henkilöistä. Kaikkien tekijöiden tulee olla kunnossa, jotta prosessi voidaan suorittaa menestyksekkäästi.

5 UML-kaaviot ja käänteistekniikka

Useat edellisessä kappaleessa esitetyt välineet tuottavat ja käyttävät takaisinmallinnuksessa hyväkseen UML-kaavioita. esimerkiksi käyttötapaus- tai sekvenssikaavioita. UML-kaaviot ovat käytetyimpiä olio-ohjelmien suunnittelun välineitä. Niiden avulla järjestelmää voidaan tarkastella eri näkökulmista ja myös abstraktiotasoltaan erilaisista asetelmista [SeK01]. UML-kaavioiden välillä vallitsee riippuvuuksia, joiden avulla kaavioista voidaan muodostaa toisen tyyppisiä kaavioita. Osa riippuvuuksista on vahvempia ja selkeämpiä, osa taas heikompia. Riippuvuuksista johtuen johonkin tiettyyn kaavioon tehdyt muutokset vaikuttavat myös muihin kaavioihin.

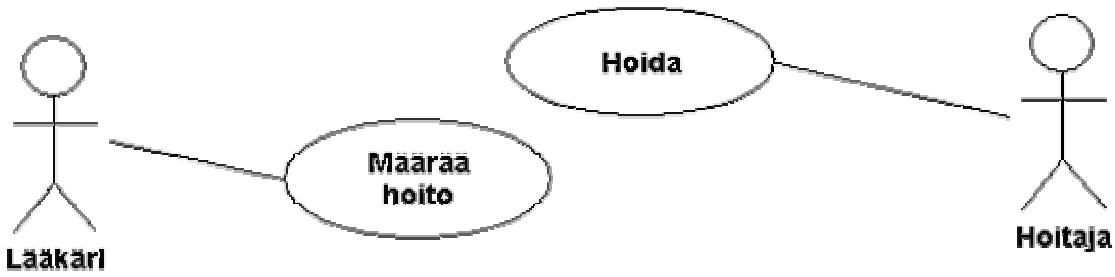
Riippuvuuksia voidaan käyttää hyväksi takaisinmallinnuksessa. Niiden avulla voidaan luoda esimerkiksi puuttuvaa dokumentaatiota jostakin järjestelmästä. Selvitettyjen yhteyksien ja näistä luotujen kaavioiden avulla voidaan myös tukea testausta tai muuta arviointia. Alkuperäisiä suunnittelun pohjana olleita kaaviota verrataan käänteistekniikan avulla selvitettyihin kaavioihin ja tarkastellaan toteuttaako järjestelmä sille alun perin asetetut vaatimukset.

Seuraavaksi esitellään eräitä UML-kaaviotyyppejä ja tämän jälkeen kerrotaan näiden välisistä erivahvuisista riippuvaisuuksista.

5.1 KÄYTTÖTAPAUSKAAVIO

Käyttötapauskaavio (use case diagram) on havaittu monessakin mielessä erittäin hyödylliseksi [BoV00] kuvausmuodoksi niin käänteistekniikassa kuin muuallakin ohjelmistotuotannossa. Yleensä dynaamisen takaisinmallinnuksen tuloksena tuotetaan juuri käyttötapauksia [Sys00].

Käyttötapauskaaviot luodaan ohjelmalle asetettavien vaatimusten perusteella. Suurin osa näistä vaatimuksista saadaan selville keskustelemalla eri käyttäjätahojen kanssa [FoS97]. Käyttötapauksissa kuvataan ohjelman toiminnallisuus eli käyttötapaukset (use case) ja ohjelmaa käyttävät tahot eli käyttäjät (actor) [HaM01]. Kuvassa 17 on yksinkertainen käyttötapauskaavio tai osa käyttötapauskaaviota, joka on voinut olla luvun 3.2 esimerkki 1 olevan ohjelmakoodin pohjalla. Lääkäri sekä Hoitaja ovat kuvan käyttäjiä ja Määrää hoito sekä Hoida ovat käyttötapauksia.



Kuva 17 Yksinkertainen käyttötapauskaavio

Martin Fowlerin mukaan käyttötapausten kuvaamisessa on tärkeintä huolehtia siitä, että kaikki mahdolliset käyttötapaukset tulevat kuvatuiksi sekä samalla ymmärretyiksi [FoS97]. Käyttäjien kuvaaminen ei ole niin tärkeää. Tärkeintä käyttötapauksissa on siis se, että tyypiltään erilaisen käyttäjien tarpeet ja päämäärät tulevat varmasti huomioiduiksi.

Käyttötapaukset on todettu hyödyllisiksi välineiksi ohjelmistotuotannossa seuraavien syiden perusteella [BoV00]. Ensinnäkin niin ohjelmiston käyttäjät kuin kehittäjätkin pystyvät ymmärtämään käyttötapauksia. Monet muut kuvaustavat ovat käyttäjien kannalta vaikeammin ymmärrettäviä. Toiseksi käyttötapausten avulla pystytään käsittämään, mikä on todella tärkeää tarkastelun kohteena olevassa ohjelmistossa. Kolmas käyttötapausten hyöty koskee suunnittelutapaa, jossa käyttötapaukset ovat keskeisessä asemassa. Tällainen suunnittelutapa mahdollistaa eri toteutus- ja suunnitteluvaihtoehtojen huomioimisen, tukee iteratiivista ohjelmiston kehitysprosessia sekä helpottaa suunnittelutietämyksen uudelleenkäyttöä. Rational Unified Process eli RUP tukee tällaista *inkrementaalista suunnittelua*. Inkrementaalisella tarkoitetaan tässä tapauksessa käyttötapausten joukkoon perustuvaa suunnittelua [Kru01].

Käyttötapauskaavioihin liittyy käyttötapausten kirjallinen kuvaaminen. Tätä kirjallista kuvaamista voidaan kutsua *skenaariokuvaukseksi*. Pelkkä kuvallinen ilmaisu ei siis riitä. Käyttötapausten keskeisimpänä ideana voidaan pitää niiden yksinkertaisuutta ja ymmärrettävyyttä, näitä ominaisuuksia käyttötapausten kirjallinen kuvaaminen tukee. Käyttötapausten tarkoituksena on olla eri asianosaistahojen ymmärrettävissä, siten ne tukevat myös takaisinmallinnusta.

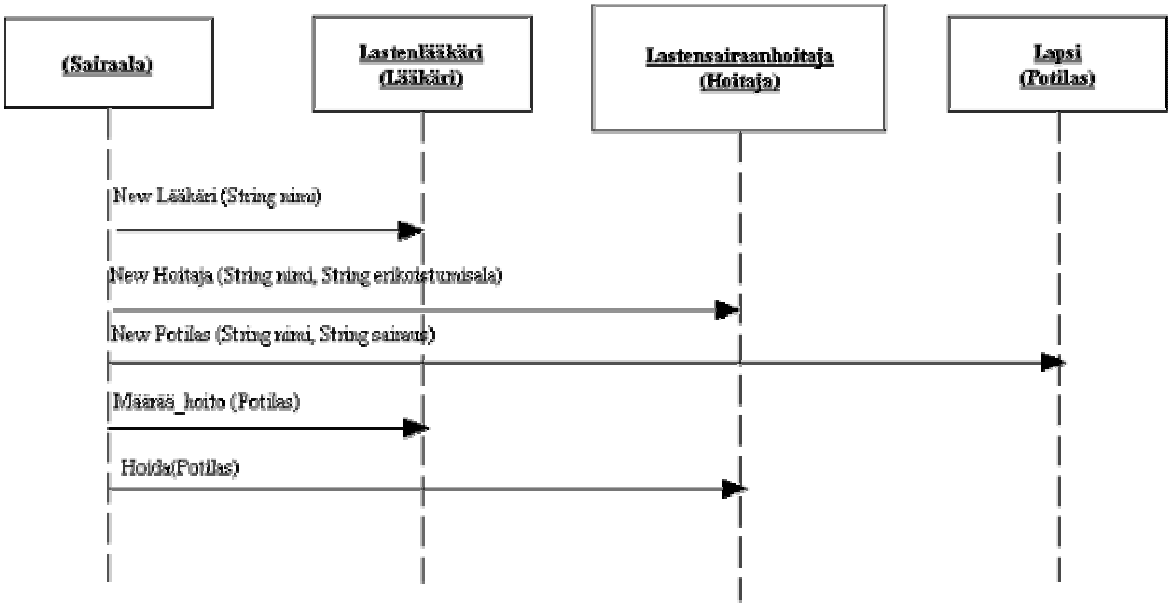
UML-notaation mukaan käyttötapauskaaviossa kuvataan käyttäjien ja käyttötapausten välisten suhteiden lisäksi myös käyttötapausten välisiä suhteita. Esimerkiksi muut käyttötapauksen voivat käyttää (include) jotakin tiettyä käyttötapausta. Samoin käyttötapausten välillä voi olla laajennussuhteita (extends). Näiden käyttöä tulee kuitenkin

pyrkiä välttämään, sillä ne saattavat hankaloittaa käyttötapausten ymmärtämistä ja tuhota siten käyttötapausten perusidean ymmärrettävyydestä.

5.2 TAPAHTUMASEKVENSSIKAAVIO

Tapahtumasekvenssikaavio (sequence diagram) eli lyhyemmin sekvenssikaavio on vuorovaikutuskaavio, jota kutsutaan joissakin yhteyksissä myös skenaarioiksi. Vuorovaikutuskaaviolla tarkoitetaan kaaviota, jolla kuvataan olioiden välistä vuorovaikutusta. Skenaario-nimitystä käytetään yleensä tapahtumasarjan kuvaukselle eli esimerkiksi käyttötapauskaavion aukikirjoitettua kuvausta voidaan kutsua siinä mielessä skenaarioksi.

Tapahtumasekvenssikaaviota käytetään kuvaamaan tapahtumaketjua [HaM01]. Se koostuu johonkin tiettyyn vuorovaikutukseen liittyvistä olioista ja niiden välisistä viesteistä. Sekvenssikaaviossa esiintyvät pystysuorat viivat kuvaavat jotakin tiettyä oliota sekä sen tilan muutoksia ja poikkisuorat nuolet viestinvälitystä olioiden välillä. Kuvassa 18 oleva tapahtumasekvenssikaavio liittyy samaan ohjelmakoodiin kuin edellisessä kappaleessa oleva yksinkertaistettu käyttötapauskaaviokin. Samaa kuvaa on käytetty jo aiemmin luvussa 3.2 olevan ohjelmakoodiesimerkin yhteydessä. Tässä tapahtumasekvenssikaaviossa eri olioita ovat Sairaala, Lääkäri, Hoitaja sekä Potilas. Esimerkkinä tapahtumasta voidaan pitää sitä, kun Sairaala-luokka välittää Lääkäri-oliolle viestin: Maaraa_hoito(Potilas).

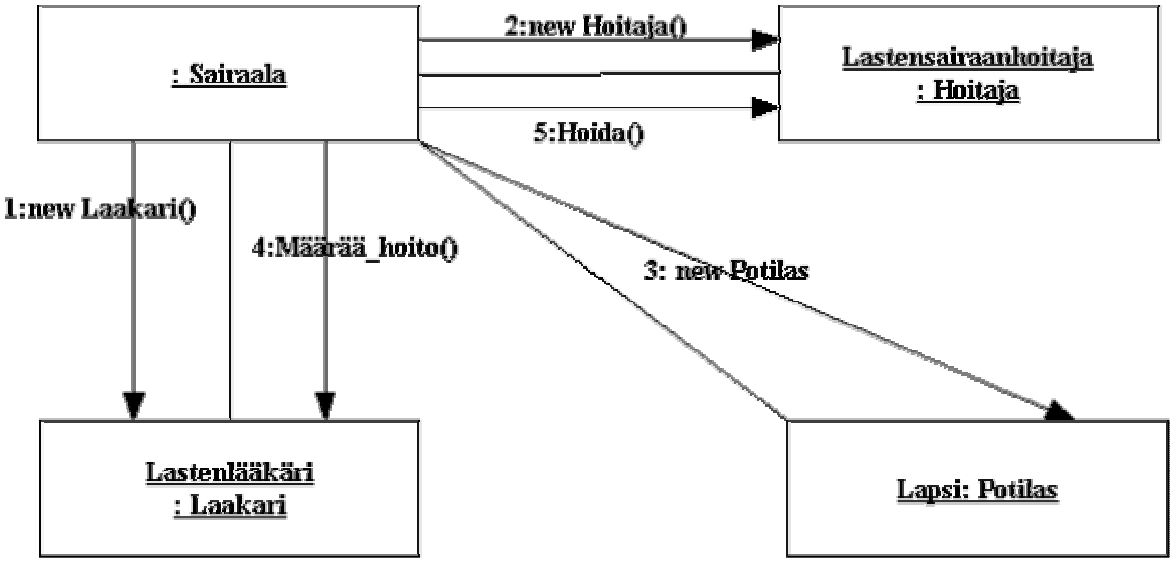


Kuva 18 Tapahtumasekvenssikaavio

Sekvenssikaavio kuvaa usein olion vuorovaikutusta jonkun tietyn käyttötapauksen yhteydessä. Sekvenssikaavion avulla voidaan kuvata esimerkiksi viestinvälitystä eri sovellusten välillä. Tapahtumasekvenssikaaviota ei kuitenkaan käytetä kuvaamaan kaikkia mahdollisia tilanteita, vaan useimmiten tapahtumasekvenssikaaviolla kuvataan normaalijärjestetystä. Voidaan ajatella, että käyttötapauskaavion skenaariokuvaukset ovat aukikirjoitettuja tapahtumasekvenssikaavioita.

5.3 YHTEISTYÖKAAVIO

Yhteistyökaavio (collaboration diagram) on tyypiltään vuorovaikutuskaavio, kuten tapahtumasekvenssikaaviokin [FoS97]. Yhteistyökaavio koostuu olioita ja luokkia kuvaavista laatikoista, viestejä kuvaavista nuolista sekä assosiaatioita kuvaavista viivoista. Lisäksi käytetään apuna numerointia, joka kuvaa tapahtumajärjestystä.



Kuva 19 Yhteistyökaavio

Yhteistyökaaviolla kuvataan lähes samoja asioita kuin tapahtumasekvenssikaaviollakin. Tämän lisäksi yhteistyökaaviossa on huomioitu olioiden ja luokkien staattinen riippuvaisuus toisistaan. Kuvassa 19 oleva yhteistyökaavio liittyy luvun 3.2 esimerkkiohjelmakoodiin. Siitä saadaan selville esimerkiksi, että Sairaala-luokassa tapahtuu uuden Lääkäri-olion luomiskutsu ja että Sairaala-luokkassa kutsutaan Hoitaja-oliota Hoida()-kutsulla. Lisäksi yhteistyökaaviosta selviää, että Hoitaja, Potilas ja Lääkäri ovat staattisessa suhteessa Sairaala-luokan kanssa.

Yhteistyökaavio ja edellä esitelty tapahtumasekvenssikaavio kuvaavat samaa toimintaa ja ovat siis sisällöltään täysin samanlaisia. Ne toimivat toisensa korvaavina kaavioina.

5.4 LUOKKAKAAVIO

Luokkakaavio (class diagram) on lähes kaikkilla käytettävä oliopohjaisen ohjelmistokehityksen kuvaustapa [FoS97]. Voidaan sanoa, että lähes kaikissa tunnetuissa oliopohjaisissa kuvausmetodeissa käytetään jonkinlaista luokkakaaviota. Monet tahot pitävätkin luokkakaaviota tärkeimpänä oliopohjaisen ohjelmistokehityksen mallinnusmenetelmänä.

Luokkakaavio kuvaa ohjelmiston staattista rakennetta ja käytöstä. Sitä käytetään kuvailemaan järjestelmän erilaisia oliotyyppejä, luokkien *operaatiota* ja *attribuutteja* sekä näiden välillä vallitsevia staattisia suhteita, kuten *assosiaatiota* (assosiation) ja *alityyppejä* (subtype). Attribuuteilla tarkoitetaan luokan ominaisuuksia, operaatiot ovat taas prosesseja, joista luokka huolehtii. Assosiaatiolla kuvataan luokkien ilmentymien välisiä suhteita ja alityypeillä kuvataan taas periytymis-suhdetta. Lisäksi luokkakaaviossa kuvataan rajoituksia olioiden välisille yhteyksille.

Luokkakaavion kuvaamisessa voidaan käyttää kolmea erilaista katsantokantaa. Nämä ovat *käsitteellinen* (conceptual), *määrittelyllinen* (specification) ja *toteutusnäkökulmaa* (implementation). Eri näkökulmat vaikuttavat vahvasti siihen, mitä ja miten luokkakaaviossa esitetään. Katsantokannat eivät kuulu muodolliseen UML:ään, mutta ne huomioimalla mallintamisesta saadaan enemmän irti.

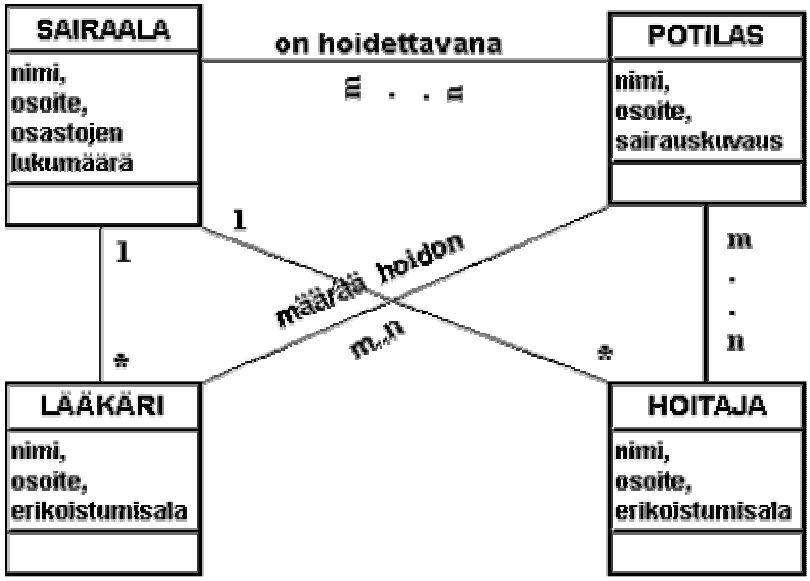
Luokkakaaviota käytetään paljon. Niiden käyttäminen ei kuitenkaan ole täysin ongelmatonta. Luokkakaavioiden käytössä on hyvä huomioida seuraavat seikat:

- Liian monimutkaisen notaation käyttäminen ei kannata.
- Luokkakaavion kuvausnäkökulma kannattaa valita huolellisesti sen mukaan, missä vaiheessa ohjelmistotuotantoprosessia ollaan menossa.
- Toteutusta ei kannata lähteä kuvaamaan liian varhaisessa vaiheessa. Käsitteellinen ja määrittelyllinen kuvaus tulee tehdä aluksi huolella.
- Kaavioiden luomisessa kannattaa mieluummin keskittyä kuvaamaan tärkeimmät seikat kuin kuvata kaikki mahdollinen. Tällä tavoin voidaan päästä tilanteeseen, jossa kaaviota todellakin käytetään ja päivitetään kehityksen myötä.

Seuraavissa kappaleissa käsitellään eri näkökulmia luokkakaavioihin hieman tarkemmin. Lisäksi kuvataan assosiaatioiden, attribuuttien ja operaatioiden merkitys eri näkökulmien kannalta.

5.4.1 Käsitteellinen näkökulma

Käsitteellisessä näkökulmassa on Fowlerin mukaan kyse siitä, että kuvataan kohteena olevaan sovellusalaan liittyviä käsitteitä. Näillä käsitteillä on selvä yhteys myöhemmin luotaviin luokkiin. Yhteys ei kuitenkaan ole sellainen, että nämä käsitteet esiintyisivät suoraan luokkina. Käsitteellisessä kuvauksessa tulee pyrkiä minimoimaan yhteydet ohjelmistoon, joka tulee toteuttamaan järjestelmän. Tästä johtuen käsitteellinen kuvaus on ohjelmointikieliriippumaton. Käsitteellistä kuvausta voidaan käyttää niin analysointivaiheessa kuin varhain määrittelyprosessissa. Tämän tason kaavioita luovat siis järjestelmän analysoijat.

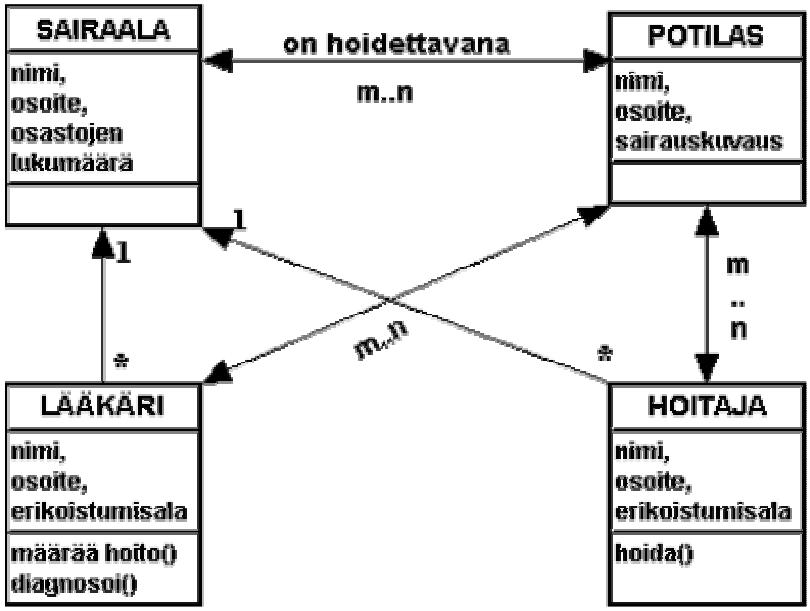


Kuva 20 Käsitteellinen yksinkertainen luokkakaavio

Käsitteellisessä kuvauksessa (Kuva 20) assosiaatiot kuvaavat luokkien välisiä käsitteellisiä suhteita. Esimerkiksi Lääkäri voi hoitaa monta Potilasta ja Potilas voi olla monen Lääkärin hoidettavana. Attribuutit ilmaisevat tässä kuvaustyypissä sitä, että jollakin käsitteellä on tietty ominaisuus. Esimerkiksi Potilaalla on Nimi. Käsitteellisellä tasolla operaatiota kannattaa käyttää kuvaamaan tietylle luokalle kuuluvia periaatteellisia vastuita.

5.4.2 Määrittelyllinen näkökulma

Määrittelyllistä näkökulmaa käytetään hyväksi määrittelyvaiheessa. Tässä näkökulmassa ryhdytään kuvaamaan jo ohjelmistoa, muttei kuitenkaan sen toteutusta (implementation), vaan rajapintoja (interface). Oliopohjaisessa ohjelmistokehityksessä erotellaan selkeästi rajapinnat ja toteutus toisistaan. Tämä erottelu kuitenkin unohdetaan usein käytännössä, koska olio-ohjelmointikielissä nämä molemmat osat ovat yhdistettyinä samaan luokkaan. Kuitenkin oliopohjaisen ohjelmoinnin vaikuttavin tekijä tulisi olla luokan rajapinta, ei sen toteutus. Rajapintojen merkitys korostuu etenkin nykyään suositussa komponenttipohjaisessa suunnittelussa.



Kuva 21 Määrittelyllinen luokkakaavio

Määrittelyllisessä kuvauksessa (kuva 21) assosiaatio edustaa vastuita. Esimerkiksi voi olla olemassa Lääkäriin liittyviä metodeja, joiden avulla voidaan tietää, ketä Potilasta tietty Lääkäri hoitaa. Attribuutit kertovat tällä kuvauksen tasolla, että jokin olio voi ilmaista ominaisuutensa ja lisäksi ominaisuuksia voidaan asettaa. Esimerkiksi Lääkäri-olio voi ilmaista ulospäin Nimensä ja Nimi voidaan asettaa jonkin metodin avulla. Määrittelyllisessä kuvauksessa operaatiot vastaavat tyypille kuuluvista yleisistä metodeista eli esimerkiksi Lääkäri-olio voi määrätä hoidon tai diagnosoida sairautta.

5.4.3 Toteutusnäkökulma

Toteutusnäkökulmassa kuvataan luokkia ja niiden toteutusta. Tätä kuvaustapaa käytetään useimmin, vaikka määrittelyllisen näkökulman käyttö olisi enemmän oliopohjaisen ohjelmistokehityksen mukaista ja siten parempi vaihtoehto. Toteutusnäkökulman

käyttäminen on suositeltavaa silloin, jos halutaan kuvata ohjelmaa jonkin tietyn toteutustekniikan kannalta. Toteutusnäkökulmalla vastataan esimerkiksi kysymykseen "miten ohjelmisto toteutetaan java-kielellä".

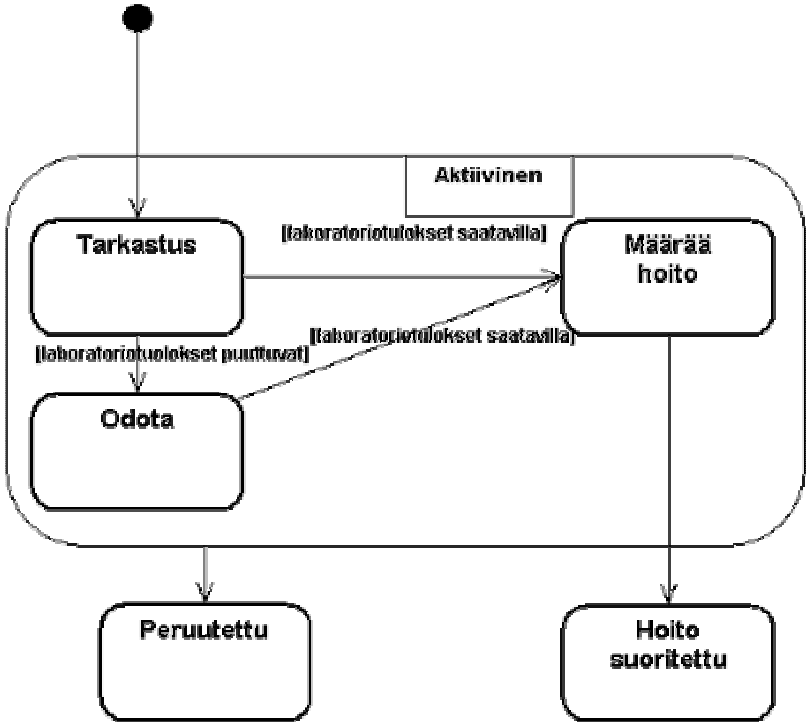
Toteutusnäkökulman mukaisessa kuvauksessa assosiaatiolla ilmaistaan kahden luokan välisiä osoittimia, jotka voivat toimia molempiin suuntiin tai vain jompaankumpaan suunnista. Toteutusvaiheessa tulee päättää, mihin suuntiin osoittimet toimivat. Esimerkiksi onko Lääkärin ja Sairaalan välinen osoitin yksi- vai kaksisuuntainen.

Toteutusnäkökulmassa attribuutin rooli on seuraavanlainen: se kertoo, että luokassa on kenttä jollekin ominaisuudelle eli esimerkiksi Lääkäri-luokassa on kenttä Nimi-ominaisuudelle. Tällä tasolla operaatiot vastaavat niin yleisistä, suojatuista kuin yksityisistäkin metodeista.

5.5 TILAKAAVIO

Tilakaavioiden avulla kuvataan järjestelmän käytöstä [FoS00]. Ne koostuvat tiloista, joita olio voi saavuttaa, tilasiirtymistä eli kuinka olion tila voi muuttua sekä toiminnoista, jotka mahdollisesti liittyvät näihin. *Tiloja* voidaan kutsua *toiminnaksi* tai *aktiivisuudeksi* (activity) ja *tilasiirtymiä* taas *toimenpiteeksi* (action). Nämä molemmat ovat yleensä jonkin metodin synnyttämiä prosesseja. mutta luonteiltaan hieman erilaisia.

Tilasiirtymät eli toimenpiteet ovat prosesseina luonteeltaan nopeasti tapahtuvia ja niitä ei voi keskeyttää. Tilat eli toiminnat voivat olla pitkäkestoisempia ja lisäksi eräät tapahtumat voivat keskeyttää käynnissä olevan prosessin. Jos tilasiirtymään ei liity tapahtumaa, se ilmentyy heti, kun tilan toiminta on suoritettu. Tilasta voi samanaikaisesti lähteä vain yksi tilasiirtymä. Tätä ehtoa vartioivat tilakaaviossa *tilasiirtymävahdit* (guards), joita kuvataan yleensä hakasulkeisiin ([]) asetetulla tekstillä. Lisäksi tilakaaviossa voi olla niin kutsuttu *ylätila* (superstate). Tällaista tilaa käytetään silloin, kun useammasta tilasta tulee päästä samanlaiseen toimintoon. Esimerkiksi tilanteessa, jossa monesta tilasta tulee päästä peruutus-tilaan (cancel). Lisäksi ylätilasta on kyse silloin, kun halutaan nimetä eräät tilat ja tilasiirtymät niiden välillä jollakin yläkäsitteellä; esimerkiksi käsitteellä aktiivinen. Tilakaaviossa on usein myös *tapahtumia* (event), jotka johtavat tiettyyn tilaan. Esimerkiksi peruutettu-tapahtuma johtaa peruutus-tilaan.



Kuva 22 Tilakaavio lääkärin vastaanotolta

Kuvassa 22 on kuvattu yksinkertainen tilakaavio, joka esittää lääkärinvastaanoton lääkärin toimintaa. Tilakaaviossa on kuvattu muutama tilasiirtymävahti sekä ylitiloja, kuten peruutettu- ja aktiivinen-tilat.

Kappaleessa 5.2 esitelty tapahtumasekvenssikaavio ja tilakaavio ovat läheisessä yhteydessä toisiinsa. Tapahtumasekvenssikaaviossa kuvatut johonkin olioon tai luokkaan tulevat väliset viestit voidaan ajatella tilakaaviossa jonkin olion tai luokan tilasiirtymänä. Tapahtumasekvenssikaaviossa kuvatut olion tai luokan tilat voidaan taas ajatella suoraan tilakaavion tiloina.

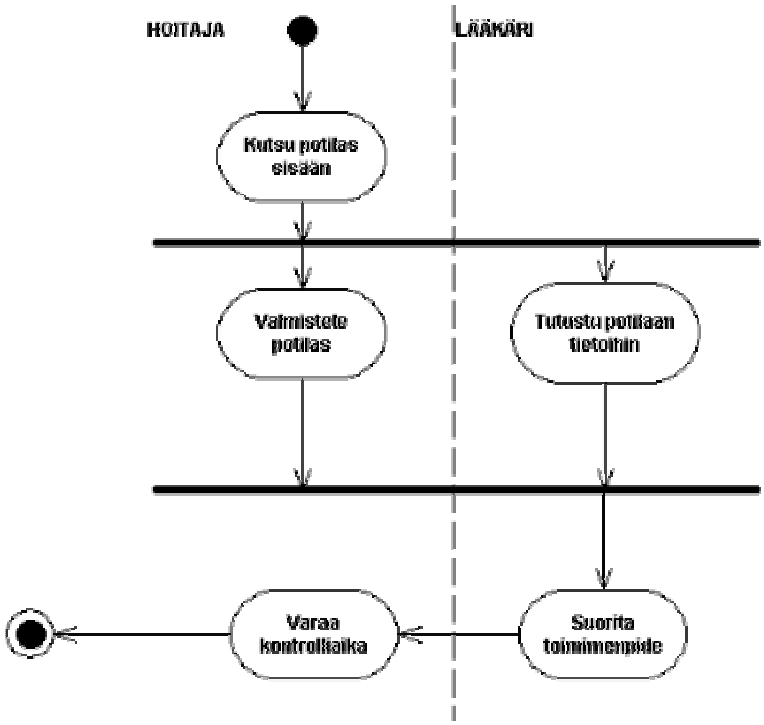
Tilakaaviota voidaan kuvata Märijärven ja Haikalan tapaan seuraavasti: olio odottaa jossakin tietyssä tilassa herätettä, joka siirtää sen toiseen tilaan tai mahdollisesti takaisin samaan tilaan [HaM01]. Useimmiten tilakaaviolla kuvataan jonkun tietyn luokan jotakin tiettyä oliota ja sen elinkaarta. Tilakaaviot ovat erittäin käyttökelpoisia silloin, kun halutaan kuvata jonkin tietyn olion matkaa käyttötapauksesta, jotka yleensä muodostavat tilakaavion tilan, toiseen käyttötapaukseen. Tilakaaviolla ei kuitenkaan ole helppoa kuvata useamman olion yhteistyötä vaativaa käytöstä. Parhaiten tilakaavioita käytetään silloin, kun niillä pyritään kuvaamaan jossakin luokassa tapahtuvaa mielenkiintoista käytöstä. Tällöin tilakaaviot mahdollisesti auttavat ymmärtämään, mitä tapahtuu ja miksi. Niitä ei siis kannata käyttää kuvamaan joka-ainoan luokan käytöstä.

5.6 AKTIVITEETTIKAAVIO

Aktiviteettikaaviolla kuvataan peräkkäisiä toimintoja. Kuvaamisessa käytetään apuna ehdollista ja rinnakkaista käytöstä. Aktiviteettikaavio on tilakaavion erikoistapaus, jossa kuvataan toiminnallisia tiloja. Aktiviteettikaavio koostuu *toiminnallisista tiloista* (activity state), joita voidaan kutsua lyhyesti myös toiminnoiksi, *haarautumakohdista* (branch), *yhdistimistä* (merge). *haaraumista* (fork) sekä *liittimistä* (join). Aktiviteettikaaviossa voidaan myös käyttää apuna *uimalinjoja* (swimlanes) ja *tahdistuspalkkeja* (synchronizing bar).

Toiminnallinen tila on aivan kuin tilakaavion tila. Se kuitenkin kuvaa aina jotakin selkeää toiminnallisuutta. Toiminta voi tarkoittaa tosielämän toimintaa kuten potilaan hoitamista tai ohjelmiston toimintaa kuten luokan metodia. Haarautumakohta on tilanne, jossa yksi tilasiirtymä jakaantuu useammaksi. Vain yksi näistä useasta siirtymästä voi tulla valituksi. Aktiviteettikaaviossakin tätä tilasiirtymän valintaa vartioi tilasiirtymävahti. Se tilasiirtymä valitaan, joka täyttää vaihtoehdon. Yhdistin tulee esille silloin, kun kaksi tai useampia tilasiirtymiä yhdistyy yhdeksi. Yhdistin on siis haaraumakohdan vastakkainen toiminto. Haarauma on kyseessä silloin, kun yksi tilasiirtymä jakautuu kahdeksi tai useammaksi rinnakkaisesti toteutettaviksi tilasiirtymiksi. Liitin (join) toimii haarauman vastakappaleena. Haaraumien ja liittimien lukumäärän tulee aina täsmätä.

Kuvassa 23 on kuvattu aktiviteettikaavio toimenpidekäynnistä. Kaaviossa on kuvattu rinnakkain hoitajan ja lääkärin mahdolliset aktiviteetit. Lisäksi apuna kuvauksessa on käytetty uimalinjoja sekä haaraumia ja liittimiä.



Kuva 23 Aktiviteettikaavio toimenpidekäynnistä

Aktiviteettikaaviot toimivat parhaiten kuvatessaan rinnakkaista käytöstä. Niiden vahvuutena voidaankin pitää sitä, että ne kannustavat rinnakkaiseen toimintaan ja sen vuoksi ne toimivat hyvänä apuvälineenä monisäikeisten (multithreaded) sovellusten suunnittelussa ja ymmärtämisessä. Suurimpana aktiviteettikaavion heikkoutena voidaan pitää sitä, että ne eivät kuvaa selkeästi toiminnan ja olioiden yhteyksiä. Tämän vuoksi ne eivät ole toimivimpia välineitä olioiden yhteistyön, olion käytöksen muutosten tai ehdollisen logiikan kuvaamiseen. Toisaalta ne toimivat erittäin hyvin, kun halutaan analysoida käyttötapauksia. ymmärtää asiankäsittelyä (workflow) sekä reaalimaailman aktiviteetteja tai kuvailla monimutkaista peräkkäisalgoritmia.

5.7 KOMPONENTTIKAAVIO

Komponenttikaavio kuuluu UML-kaavioiden ryhmittelyssä fyysisiin kaavioihin. Fyysisiä kaavioita on kaksi; komponenttikaavion ja sijoittelukaavio (deployment diagram). Komponenttikaavio kuvaa ohjelmiston komponentteja ja niiden välisiä riippuvaisuuksia. Komponentti voidaan määritellä toteutusta kuvaavaksi ohjelmiston fyysiseksi osaksi, jolla on rajapintoja [HaM01]. Riippuvuudet komponenttien välillä kuvaavat kuinka muutokset johonkin komponenttiin tulevat vaikuttamaan muihin komponentteihin. Riippuvaisuuksia voi olla monenlaisia kuten esimerkiksi kommunikointi

(communication) ja kääntäminen (compilation). Komponentin rajapinta määrittelee sen tarjoamat palvelut ja samalla se peittää komponentin toteutuksen muilta. Komponentit kommunikoivat toistensa kanssa rajapintojen kautta. Rajapintaa voidaan pitää eräänlaisena sopimuksena; se määrittelee komponentin tarjoamien palveluiden lisäksi oletukset, joita komponentti tekee ympäristöstään tai käyttäjän toiminnasta johtuen, rajoitukset, joita komponentin toteutus asettaa sekä komponentin toiminnan poikkeustilanteissa.

Useimmiten komponenttikaavio ja ohjelmiston sekä laitteiston välisiä fyysisiä suhteita kuvaava sijoittelukaavio esitetään yhdistettynä kaaviotyyppinä. Fyysisiä kaavioita käytetään kuvaamaan ohjelmiston loogisesta tiedosta poikkeavaa fyysistä tietoa.

5.8 UML-KAAVIOIDEN VÄLISIÄ YHTEYKSIÄ

Kuten jo tämän luvun alussa todettiin, UML-kaavioiden välillä vallitsee tyypiltään erilaisia riippuvaisuuksia. Eräät riippuvuudet ovat lähes yksi-yhteen -riippuvaisuuksia eli riippuvaisuus on erittäin vahva. Suurin osa riippuvaisuuksista on kuitenkin hieman heikompia. Kuitenkin selvää on, että jos johonkin kaavioon tehdään muutoksia, se yleensä vaikuttaa tavalla tai toisella myös muihin kaavioihin [SeK01]. Esimerkiksi käyttötapausten määrän kasvaessa voidaan pitää selvänä, että muutoksia syntyy niin vuorovaikutuskaavioon kuin luokkakaavioonkin. Vuorovaikutuskaavioon syntyy lisää olioiden ja luokkien välisiä viestejä, ja luokkakaaviossa taas luokkien sekä niiden operaatioiden määrään tulee muutoksia.

Selonen, Koskimies ja Sakkinen ovat luokitelleet riippuvuudet neljään erilaiseen ryhmään [SeK01]:

- täydellinen riippuvaisuus (full transformation)
- vahva riippuvaisuus (strong transformation)
- tuettava riippuvaisuus (supported transformation)
- heikko riippuvaisuus (weak transformation).

Kun kaavioiden välisiä riippuvaisuuksia käytetään hyväksi takaisinmallinnuksessa, tulee huomioida, että tuloksena syntyy useampia erilaisia vaihtoehtoja. Tämä johtuu siitä yksinkertaisesta syystä, että kaikkien kaavioiden välillä ei todellakaan vallitse täydellinen tai edes vahva riippuvaisuus. Tästä huolimatta riippuvuuksien hyödyntäminen on yleensä kannattavaa. Jos mietitään takaisinmallinnuksen hyödyntämistä vaa-

timusten toteutumisen varmistamisessa, on riittävää, jos kyetään löytämään yksi vaihtoehto, joka toteuttaa alkuperäiset vaatimusmäärittelyssä käytetyt kaaviot.

Seuraavissa aliluvuissa tullaan esittelemään näitä riippuvaisuuksia tarkemmin. Samalla pyritään perustelemaan, miksi joidenkin kaavioiden välillä vallitsevat tietyt riippuvaisuudet. Lisäksi käsitellään esimerkkiä, jossa näytetään kuinka tapahtumasekvenssikaavioden ja tilakaavioiden välistä yhteyttä on hyödynnetty Tampereen yliopiston ja Tampereen teknillisen korkeakoulun kehittämässä SCED-takaisinmallinnusvälineessä.

5.8.1 Täydellinen riippuvaisuus

Täydellisellä riippuvaisuudella tarkoitetaan sitä, että kahden kaavion tiedot vastaavat lähes täysin toisiaan. Täydellisessä riippuvaisuudessa hyväksytään vain merkitykseltään vähäisten tietojen poisjäänti tai mahdollinen lisäys siirryttäessä lähdekaaviosta (source diagram) kohdekaavioon (target diagram). Lähdekaaviolla tarkoitetaan kaaviota, josta lähdetään liikkeelle ja kohdekaaviolla taas kaavioita, johon prosessissa päädytään. Vähäisinä tietoina voidaan pitää esimerkiksi yhteistyökaaviossa olevia linkkejä eli assosiaatiota. Tämä johtuu siitä, että yleensä olioiden välillä vallitsee assosiaation lisäksi myös kommunikaatiota. Siten niiden välinen suhde tulee selväksi jo viestin välityksen avulla.

Täydellinen riippuvaisuus onnistuu vain sellaisten kaavioiden välillä, jotka ovat semanttisesti eli merkityksellisesti lähellä toisiaan. Tällaisia kaavioita ovat Selosen, Koskimiehen ja Sakkisen mukaan vuorovaikutuskaaviot eli tapahtumasekvenssikaaviot ja yhteistyökaaviot sekä tilakaaviot (statechart diagram) ja aktiviteettikaaviot (activity diagram).

Tapahtumasekvenssikaavion ja yhteistyökaavion välinen täydellinen riippuvaisuus on nähtävissä selvästi. Molemmat kaaviot kuvaavat täysin samoja asioita. Lisäksi mikä tärkeintä kaavioiden käyttämä näkökulma on samanlainen. Molemmat kaaviot pyrkivät kuvaamaan vuorovaikutussuhteita olioiden ja luokkien välillä. Lisäksi UML:n metamalli (metamodel) ei erottele näitä kaaviotyyppejä lainkaan toisistaan.

Tilakaavioiden ja aktiviteettikaavioiden välinen yhteys ei välttämättä ole aivan yhtä selvä kuin vuorovaikutuskaavioiden väliset yhteydet. Tilakaavion ja aktiviteettikaavion välistä yhteyttä voidaan perustella ajatuksella, jonka mukaan aktiviteettikaavio on tilakaavion erikoistapaus. Tästä voidaan olla eri mieltä. Aktiviteettikaavion voidaan

myös ajatella kuvaavan todellisen elämän aktiivisia tapahtumia, kun taas tilakaavion voidaan ajatella kuvaavan pelkästään laitteiston tai ohjelman saavuttamia tiloja.

5.8.2 Vahva riippuvaisuus

Vahvalla riippuvaisuudella tarkoitetaan tilannetta, jossa lähdekaavio ja kohdekaavio vastaavat melkein toisiaan. Tällä tarkoitetaan sitä, että kohdekaavio sisältää erittäin paljon samaa tietoa kuin lähdekaavio, jonka joku olisi voinut luoda samasta järjestelmän osasta. Tällainen riippuvaisuus vallitsee esimerkiksi lähdekoodin ja tapahtumasekvenssikaavion välillä sekä tapahtumasekvenssikaavion ja luokkakaavion välillä. Muita vahvoja riippuvaisuuksia on tilakaavion ja sekvenssikaavion välillä, tilakaavion sekä luokkakaavion välillä, luokkakaavion ja lähdekoodin välillä sekä päinvastoin ja tilakaavion ja lähdekoodin välillä.

Tilakaavion ja tapahtumasekvenssikaavion välinen vahva riippuvaisuus on todistettu tilakoneen synteesi -algoritmin avulla. Algoritmi on nimeltään BK-algoritmi [KoM96]. Tätä algoritmia on käytetty hyväksi SCED-takaisinmallinnusvälineessä, jonka avulla voidaan luoda sekvenssikaavioista tilakaavioita ja myös päinvastoin.

Tapahtumasekvenssikaavion ja luokkakaavion välinen vahva riippuvaisuus on osoitettu UML-metamallin avulla [SeK01].

Luokkakaavion ja lähdekoodin molempaan suuntaan kulkevat vahvat riippuvaisuudet lienevät selvät. Tietenkin tämän riippuvaisuuden syntymiseen tarvitaan luokkakaaviosta oikea näkökulma. Näitä näkökulmia selvitettiin edellä luokkakaaviosta kertovassa luvussa 5.4. Käsitteellisen luokkakaavion ja lähdekoodin välinen yhteys ei ole samalla tavalla vahva kuin toteutuksellisen luokkakaavion ja lähdekoodin. Määrittelyllisen luokkakaavion ja lähdekoodin välinen suhde voi olla vahva.

Tilakaavion ja luokkakaavion välinen vahva riippuvaisuus voidaan todeta kahdella tapaa. Ensinnäkin käyttäjän määrittelemät säännöt voidaan kuvata luokkiin pohjautuvalla tilakaavioesityksellä. Toinen polku tilakaaviosta luokkakaavioon kulkee seuraavasti: luokkien välisten kommunikaatioyhteyksien staattinen sievennetty kuvaus eli projektio voidaan esittää luokkakaavion avulla.

Tilakaavion ja lähdekoodin välinen vahva riippuvaisuus lienee jälleen selvä. Tilakaaviossahan kuvataan jossakin luokassa olevan jonkun tietyn olion saavuttamia tiloja ja tilasiirtymiä. Nämä on siirrettävissä lähes suoraan lähdekoodiin. Koska aktiviteettikaavion ja tilakaavion välillä vallitsee täydellinen riippuvaisuus, voidaan olettaa, että aktiviteettikaaviosta päästään samalla perusteella lähdekoodiin.

5.8.3 Tuettava riippuvaisuus

Tuettava riippuvaisuus vallitsee kahden kaavion välillä silloin, kun lähdekaavioissa mukana olevan tiedon avulla voidaan selvittää kohdekaavion koostumusta. Tällaista mukana olevaa tietoa ovat esimerkiksi kaavioon merkityt arvot tai muut mahdolliset kommentit. Jollei lähdekaaviosta löydy näitä tietoja, riippuvaisuus on heikko tai täysin käyttökelvoton. Tuettava riippuvaisuus voi vallita esimerkiksi luokkakaavion ja käyttötapauskaavion välillä.

Luokkakaavion ja käyttötapauskaavion välinen tuettava riippuvaisuus löytyy silloin, luokkakaavion luokkiin on liitetty kommentit siitä, mihin käyttötapauksiin ne ottavat osaa. Ellei näitä kommentteja löydy, riippuvaisuus ei ole tämän tasoinen. Riippuvaisuus voi silti olla heikkona olemassa.

Tuettava riippuvaisuus löytyy myös luokkakaavion ja yhteistyökaavion sekä komponenttikaavion välillä. Näihin molempiin tuettaviin riippuvaisuuksiin vaaditaan jälleen, että luokkakaavion luokat on varustettu riittävin kommentein. Kun kyse on luokkakaavion ja yhteistyökaavion välisestä riippuvaisuudesta, luokista tulee löytyä kommentit niistä yhteistyökaavion solmuista (nodes), joihin ne kuuluvat. Luokkakaavion suhteessa komponenttikaavioon vaaditaan luokkiin kommentit niistä komponenteista, joihin luokat kuuluvat.

5.8.4 Heikko riippuvaisuus

Heikossa riippuvaisuudessa tarkasteltavana oleva lähdekaavio sisältää vain hieman kohdekaavion luomisessa tarvittavaa tietoa. Kuitenkin näitä tietoja on sen verran, että niitä voidaan käyttää hyväksi uuden kaavion luomisessa. Heikko riippuvaisuus voi vallita luokkakaavion ja tapahtumasekvenssikaavion välillä. Kuten aiemmin todettiin, näiden kaavioiden välillä vallitsee toiseen suuntaan vahva riippuvaisuus.

Luokkakaavion ja tapahtumasekvenssikaavion välisessä heikossa riippuvaisuudessa on kyse siitä, että luokkakaavion luokkien perusteella voidaan luoda eräänlainen runko tapahtumasekvenssikaaviolle. Heikkoja riippuvaisuuksia voidaan löytää myös muiden kaavioiden väliltä. Heikko riippuvaisuus on mahdollinen esimerkiksi silloin, kun tuettavaan riippuvuuteen vaadittavia riittäviä kommentteja ei löydy.

5.8.5 Muita oletettavia riippuvaisuuksia

Tämän luvun alussa on esitetty, että voidaan olettaa muutosten käyttötapauskaaviossa aiheuttavan muutoksia muuallakin. Tämä on tietenkin totta, mutta tämän kaltaisia

riippuvaisuuksia ei voida pitää Selosen, Koskimiehen ja Sakkisen mukaan riittävän selkeinä tai suorina.

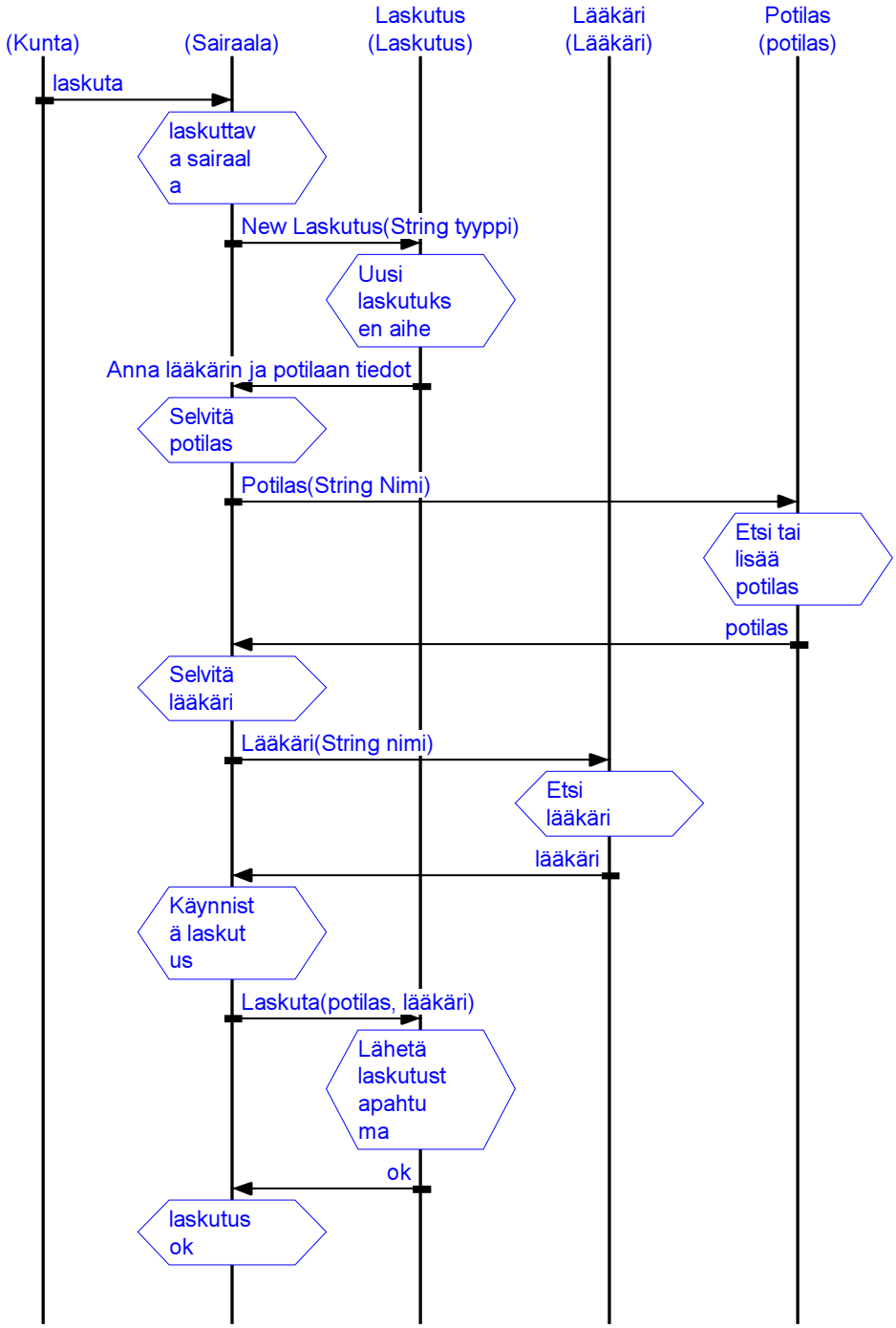
Selonen, Koskimies ja Sakkinen nostavat esille yhteydet käyttötapauskaavion ja niiden merkitystä tarkentamaan käytettävien aktiviteettikaavion, yhteistyökaavion ja tapahtumasekvenssi kaavion välillä. Heidän mukaansa UML ei tarjoa riittävän selkeitä tietoja näiden yhteyksiä riittävään määrittämiseen. Tätä voidaankin pitää eräänä UML:n heikkoutena.

Muitakin yhteyksiä kaavioiden välillä voidaan olettaa olevan olemassa. Kuitenkaan, jos kaaviot eivät itsessään tarjoa riittävän selkää informaatiota, yhteyden määrittäminen riittävän luotettavasti kaaviosta toiseen on mahdotonta ja sen vuoksi tätä yhteyttä on jopa vaarallista ryhtyä käyttämään takaisinmallinnusmielessä. Tietenkin vertailemalla kaavioita toisiinsa ja yhdistelemällä eri kaavioiden tietämystä, voidaan ainakin tarkastaa joidenkin saavutettujen tulosten järkevyyttä.

5.8.6 Esimerkki tapahtumasekvenssikaavion ja tilakaavion välisestä yhteydestä

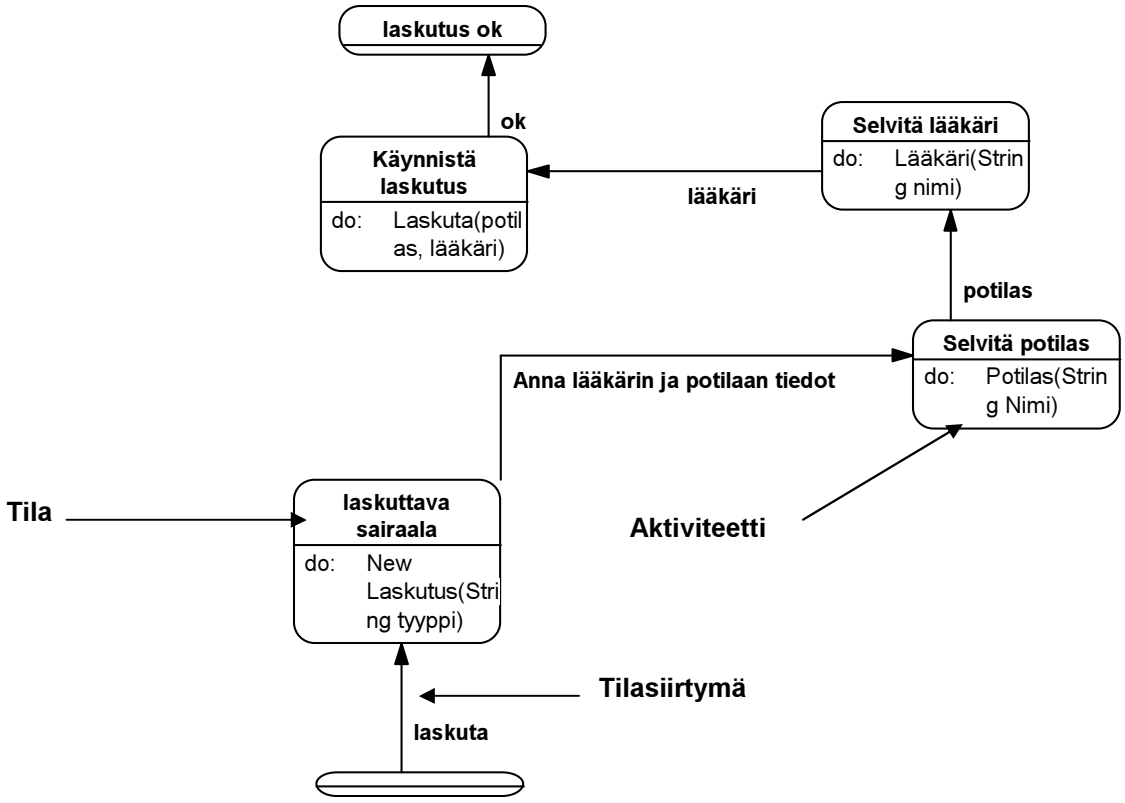
Kuten aiemmin todettiin tapahtumasekvenssikaavion ja tilakaavion välillä vallitsee vahvariippuvaisuus. Tätä riippuvaisuutta on hyödynnetty Tampereen yliopiston ja Tampereen teknillisen korkeakoulun yhteistyöstä syntyneessä SCED-välineessä [Sce98, KoM96]. SCED-väline on esitelty luvussa 4.2.2.

Kuvassa 24 on SCED-välineellä piirretty tapahtumasekvenssikaavio Laskutustoimenpiteestä.



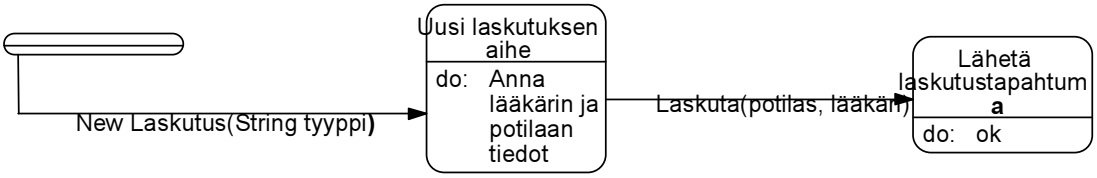
Kuva 24 SCED-välineellä tehty Laskutus-tapahtumasekvenssikaavio

Tapahtumasekvenssikaavion piirtämisen jälkeen, voidaan valita jokin kaavion luokista tai olioista, josta halutaan generoida tilakaavio. Tässä tapauksessa tilakaavio haluttiin generoida Sairaala-luokasta ja Laskutus-oliosta. SCED-takaisinmallinnusvälineen Sairaala-luokasta luoma tilakaavion runko esitetään kuvassa 25. Sairaalaluokan tilakaavioon tulee paljon tiloja, sillä Laskutus-tapahtumasekvenssikaaviossa Sairaala-luokalla on eniten tiloja.



Kuva 25 SCED-välineellä Laskutus-tapahtumasekvenssikaaviosta luotu Sairaala-luokan tilakaavio

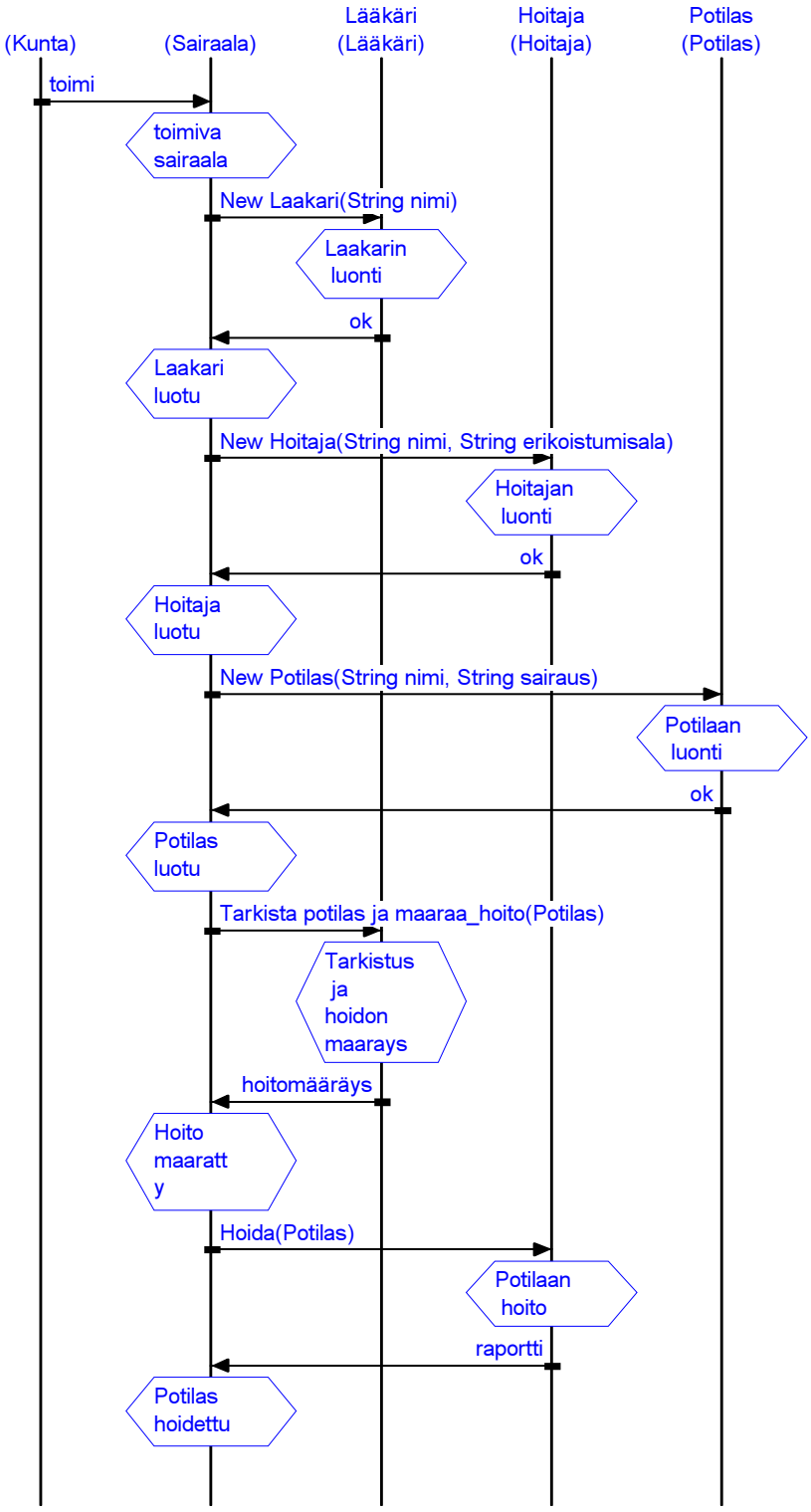
Seuraavassa kuvassa 26 esitetään SCED-välineen luoma tilakaavio Laskutus-oliosta. Laskutus-olion tilakaavio on suppea, koska Laskutus-olion tilojen määrä Laskutus-tapahtumasekvenssikaaviossa on pieni.



Kuva 26 SCED-välineellä Laskutus-tapahtumasekvenssikaaviosta luotu Laskutus-olion tilakaavio

Kuten kuvista 24, 25 ja 26 huomaamme, tapahtumasekvenssikaaviosta saadaan riittävästi tietoa melko ymmärrettävän tilakaavion rungon automaattiseen luomiseen. Tilakaavioihin on saatu siirrettyä kaikki mahdollinen tapahtumasekvenssikaaviosta selville saatava tieto. Tilakaaviota selventäisi, jos aktiviteettia kuvaavassa DO-kohdassa kerrottaisiin, minne käsky menee. Tilakaaviota saadaan tarkennettua halut-

tuun suuntaan SCED-välineen muokkausominaisuuksilla. Muokkausominaisuudet parantavat SCED-välineellä saatuja lopputuloksia.

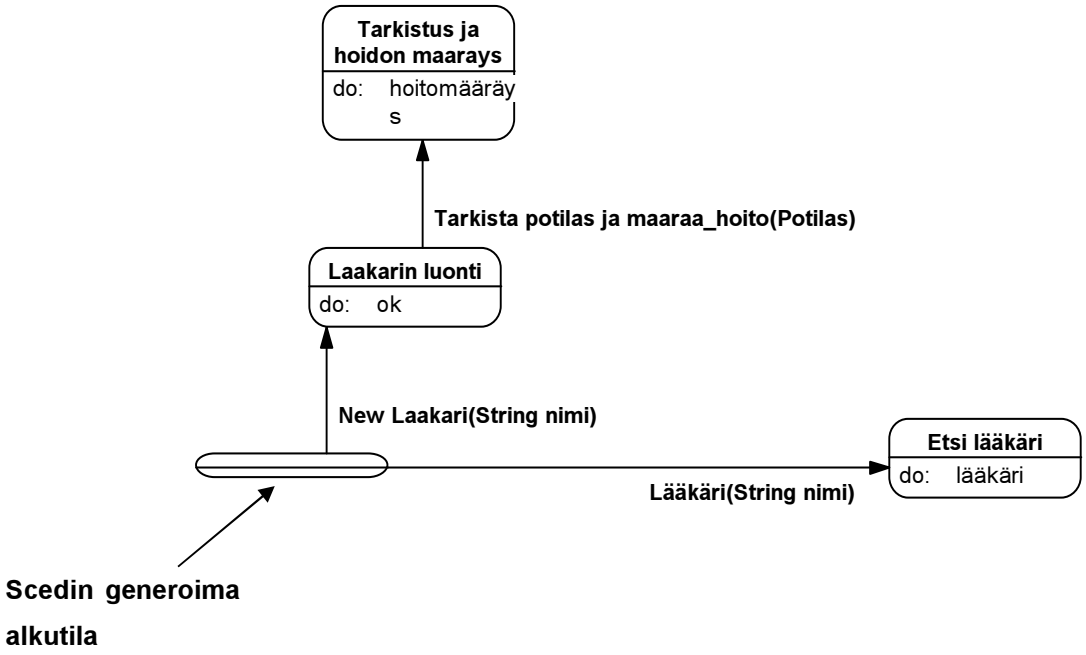


Kuva 27 SCED-välineellä tehty Sairaala-tapahtumasekvenssikaavio

Kuvassa 27 on kuvattu Sairaala-tapahtumasekvenssikaavio, jota käytetään hyväksi seuraavassa toisen tyyppisessä tilakaavioiden generointia koskevassa esimerkissä.

SCED-välineellä on mahdollista yhdistää useammassa sekvenssikaaviossa esiintyvän olion tai luokan tilakaavio. SCED luo samaan kuvaa kaksi erillistä tilakaaviota, joista voi luoda yhden tilakaavion yhdistämällä jonkin tilan. Esimerkiksi Laskutus-tapahtumasekvenssikaaviosta ja Sairaala-tapahtumasekvenssikaaviosta voidaan yhdistää Laakari-olion tilat samaan tilakaavioon.

Kuvassa 28 esitetään tilakaavio, joka on saatu valitsemalla Lääkäri-olio Sairaala- ja Laskutus-tapahtumasekvenssikaavioista ja yhdistämällä näistä syntyvien tilakaavioiden tyhjä alkutila. Tämä tyhjä alkutila on se tila, josta uuden Lääkäri-olion luomiskäsky syntyy.



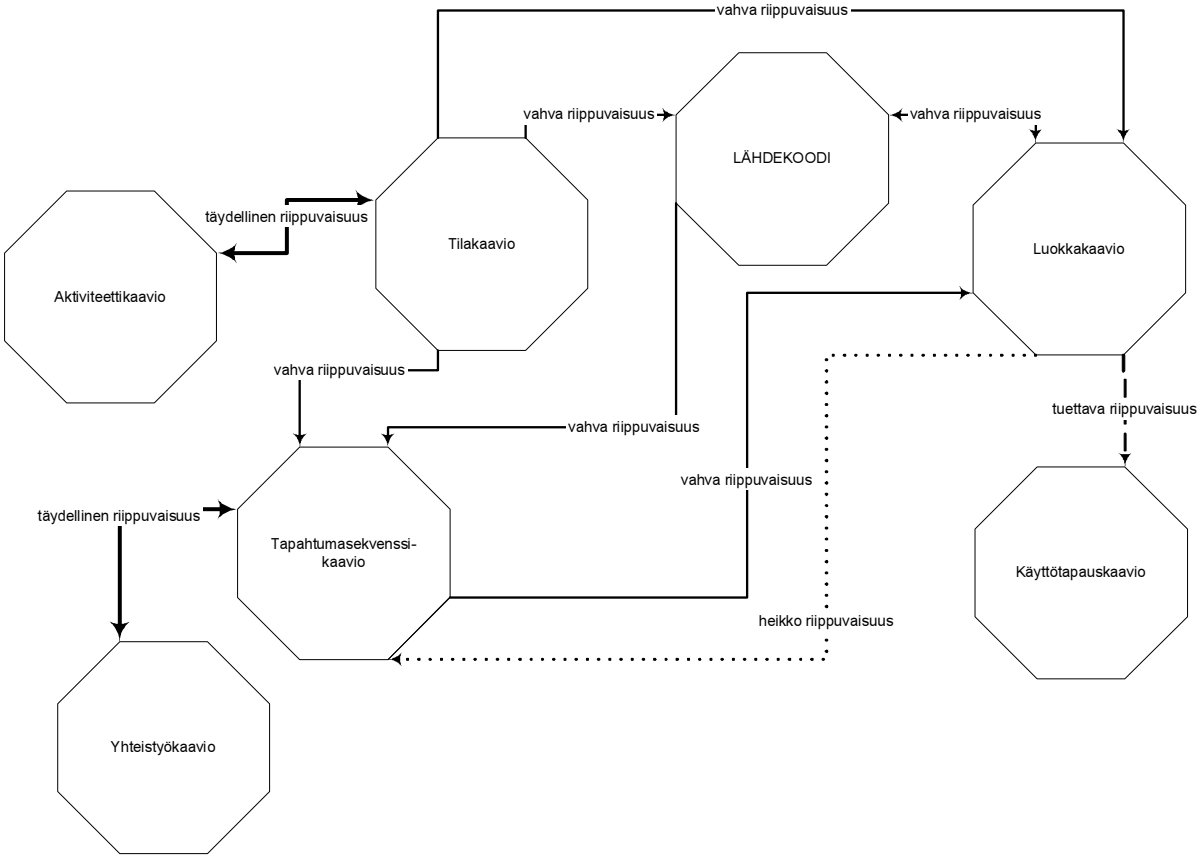
Kuva 28 SCED:in luoma yhdistetty tilakaavio Laakari-oliolle

Yhdistetystä kaaviosta nähdään, ettei SCED-väline sinällään synkronoi eri tapahtumasekvenssikaavioista saatua tietoa. Toisaalta tämä on erittäin ymmärrettävää; tapahtumasekvenssikaaviot eivät yleensä tarjoa synkronointiin vaadittavia tietoja.

Nämä esitetyt esimerkit valottavat tapahtumasekvenssikaavioiden ja tilakaavioiden välisiä yhteyksiä. Samalla ne selventävät, mitä SCED-takaisinmallinnusvälineellä voi tehdä. Jos ajatellaan UML-kaavioiden välisiä suhteita ja niiden käyttämistä takaisinmallinnuksessa, kahden kaavion välisten yhteyksien automatisoitu luominen on hyvä alku.

5.9 UML-KAAVIOIDEN VÄLISTEN YHTEYKSIEN HYÖDYNTÄMINEN

UML-kaaviot ovat eräs ohjelmistotuotannon käytetyimmistä mallinnusmenetelmistä. Niiden avulla ohjelmistoja pystytään kuvaamaan ja suunnittelemaan informatiivisesti. Kuten tässä luvussa aiemmin on todettu, UML-kaavioden sekä UML-kaavioiden ja ohjelmakoodin välillä vallitsee erilaisia riippuvuuksia. Toiset näistä riippuvuuksista ovat vahvoja, toiset taas heikompia. Kaavioita ei kuitenkaan ole kehitetty niin tieteellisen tarkasti, että voitaisiin löytää täysin pitävä riippuvuusketju esimerkiksi käyttötapausten ja ohjelmakoodin välille. Kuten edellä on esitetty, jonkinlaisia melko selkeitä riippuvuusketjuja on kuitenkin löydettävissä.



Kuva 29 UML-kaavioiden välisiä yhteyksiä [SeK01]

Kuvassa 29 on kuvattu eräiden UML-kaavioiden ja ohjelman lähdekoodin välisiä riippuvuuksia Selosen, Koskimiehen ja Sakkisen mukaisesti. Kuvasta voidaan nähdä, että esimerkiksi yhteistyökaaviosta kulkee vahvana pidettävä riippuvuusketju tapahtumasekvenssikaavion ja luokkakaavion kautta ohjelman lähdekoodiin. Tietenkin jotain informaatiota katoaa matkalla, sillä tapahtumasekvenssikaavion ja luokkakaavion

sekä luokkakaavion ja lähdekoodin välinen riippuvuus ei ole täydellinen. Kaavion mukaan reitti toiseenkin suuntaan on olemassa: lähdekoodista tulisi pystyä päätymään yhteistyökaavioon tapahtumasekvenssikaavion kautta.

UML-kaavioiden välisiä yhteyksiä on käytetty ja voidaan käyttää jatkossakin takaisinmallinnuksen automatisoimiseen. Tällä hetkellä eri välineet tarjoavat automatisoituja apukeinoja myös muiden kuin edellisessä kappaleessa esitettyjen tila- ja tapahtumasekvenssikaavioiden välisten yhteyksien selvittämiseksi. Osa välineistä käyttää hyväkseen ohjelmakoodin ja tiettyjen kaavioiden, kuten luokkakaavio ja aktiviteettikaavio, välisiä yhteyksiä ja osa taas luo SCED:in tavoin kaavioita toisten kaavioiden pohjalta.

Kaavioiden välisten suhteiden tutkiminen ja tutkimustulosten käyttäminen automatisoitujen välineiden kehityksessä on tärkeää, sillä tällä tavoin saataisiin tehokkaita apuvälineitä ohjelmistojen takaisinmallinnukseen. Välineet toimisivat hyvänä tukena pitkin ohjelmistotuotantoprosessia. Niitä voitaisiin käyttää tukena niin vanhan ohjelman uudistamisessa kuin uuden ohjelman testauksessa tai myös ohjelmakoodin tarkastuksessa.

6 Käänteistekniikan ongelma-alueet

Käänteistekniikan hyödyntämistä ei ole koettu helpoksi. Ongelmia syntyy muun muassa sen käyttöönotossa sekä ymmärtämisessä ja eräät tahot ovat nostaneet esille myös laillisuuskysymyksiä.

6.1 VAIKEUDET KÄÄNTEISTEKNIIKAN KÄYTTÖÖNOTOSSA

Käänteistekniikan, kuten muidenkin menetelmien, käyttöönottoon vaikuttavat monet tekijät. Jos käyttöönottoprosessi on hankala ja kallis, hyvänkin menetelmän käyttöönotto saattaa jäädä. Käänteistekniikan käyttöönottoon liittyy tiettyjä ongelmakohtia, joita tullaan käsittelemään seuraavaksi.

Käänteistekniikan eräänä ongelma-alueena pidetään sen määrittelyjen moninaisuutta sekä käyttökelpoisten tutkimustulosten puutetta [BeR00]. Käänteistekniikan määrittelyn taso liikkuu melko korkealla abstraktiotasolla. Tämän vuoksi voi olla vaikeaa sanoa, milloin todellakin on kyse käänteistekniikasta. Käänteistekniikkatutkimusta on tehty melko paljon, mutta sen tuloksia ei ole pystytty hyödyntämään riittävästi ohjelmistoteollisuudessa. Tästä johtuen muita menetelmiä käytetään mieluummin perinnejärjestelmien käsittelyyn, muun muassa XML (Extensive Markup Language)-kääreitä.

Käytännön ongelmia takaisinmallinnukseen tulee siitä, että useimmiten ohjelmistot ovat huonosti dokumentoituja [Sys00]. Tilanne voi olla myös sellainen, että ohjelmistoa alun perin kehittäneitä henkilöitä ei enää saada kiinni. Tästä johtuen ainoa luotettava lähdemateriaali ohjelmistosta on sen lähdekoodi. Lähdekoodin tulkinta ei ole aina yksinkertaista, sillä se sisältää usein sellaisia rakenteita, joiden merkitys ja toiminnallisuus jäävät hieman epäselväksi.

Eräs suurimmista ongelmista käänteistekniikan alueella on tehokkaiden automaattisten apuvälineiden puute. Tämän vuoksi käänteistekniikan toteuttaminen on usein melko arvokasta ja hidasta. Käänteistekniikkavälineitä kehitetään kuitenkin kokoajan esimerkiksi Victorian yliopistossa Kanadassa.

Ongelmia aiheuttaa myös takaisinmallinnusprosessin puutteellinen määrittely. Tämän vuoksi prosessi toteutetaan joka kerta eri tavoin [MüJ00]. Näin takaisinmallinnuksesta syntyy turhia kuluja. Jotta takaisinmallinnuksesta saataisiin tuottavampaa, sen toistettavuuteen tulisi kiinnittää huomiota. Lisäksi takaisinmallinnusprosessia tulisi tehostaa ja samalla parantaa prosessin kypsyysastetta.

Eräänä käänteistekniikan ongelmana voidaan pitää sitä, että tutkimus on yleensä painottunut pelkästään koodin perusteella tehtävään takaisinmallinnukseen (code reverse engineering). Joskus tärkeämpää tietoa voisi löytyä tiedon takaisinmallinnuksen (data reverse engineering) avulla [BeR00, MüJ00].

Käänteistekniikkaa on kritisoitu myös siitä, että sen avulla pyritään ratkomaan turhan suuria ongelmia [WeH95]. Tämän lisäksi tutkijat ja kehittäjät lupaavat käänteistekniikan avulla asiakkaalleen liian suuria ja mullistavia tuloksia. Esimerkiksi todellisuudessa laajan vanhan ohjelmiston takaisinmallinnus voi olla turhan vaativa ja kallis prosessi. Kritiikin mukaan ongelmallista on myös se, että käänteistekniikan avulla ei välttämättä voida tehokkaasti selvittää laajan ohjelmiston takana olleita ajatuksia tai syitä siihen, miksi johonkin ratkaisuun on päädytty. Suurin ongelma ei kuitenkaan ole itse käänteistekniikka vaan suunnittelijat, jotka eivät ole alun perin tehneet suunnittelutyötään kunnolla. Tänä päivänäkin suunnittelua tehdään heikkotasoisesti ja sen seurauksena järjestelmät ovat erittäin sekavia. Kritiikissä myönnetään kuitenkin, että käänteistekniikka on tietyissä tilanteissa erittäin käyttökelpoista ja sen vuoksi käänteistekniikan tutkimusta kannattaa jatkaa.

6.2 ONGELMAT LAILLISUUDEN KANSSA

Käänteistekniikan historiallinen tausta juontaa juurensa laitteistopuolelle. Alun perin elektronisten laitteiden valmistajat pyrkivät parantamaan tuotteidensa laatua purkamalla niin omia kuin toistenkin valmistajien laitteita osiin [ChC90]. Tällä purkamisella yritettiin yleisesti selvittää, miten tuote on valmistettu. Kilpailijoiden tuotteita purkamalla pyrittiin selvittämään tietenkin valmistus- ja suunnittelusalaisuuksia. Tästä menneisyydestä johtuen käänteistekniikalla on osittain laiton leima otsassaan.

Osa ohjelmistojen kehittäjistä on sitä mieltä, että käänteistekniikka on laitonta [BeL98]. He perustelevat syytöksiään sillä, että käänteistekniikan avulla on helppoa selvittää toisen valmistajan tuotteen suunnittelusalaisuudet. Tällä tarkoitetaan sitä, että epärehelliset ammattilaiset selvittävät käänteistekniikan avulla ensin, miten ja miksi kilpailija ohjelmisto toimii, muuttavat sen jälkeen ohjelmistoa hieman omiin tarpeisiinsa soveltuvaksi. Tämän jälkeen ohjelmistoa alettaisiin myydä uutena, mullistavana tuotteena, joka on ominaisuuksiltaan parempi kuin kilpailijan versio. Jos näin todellisuudessa tapahtuu, silloin ollaan tekemisissä tekijänoikeusrikkomuksen kanssa.

Voidaan kuitenkin olettaa, että suurin osa käänteistekniikan käytöstä kohdistuu nykyään ohjelmistotalojen omiin tuotteisiin. Suurinta osaa ohjelmistotaloista kiinnostaa

enemmän, kuinka heidän omat huipputuotteensa on rakennettu ja kuinka niitä pystytään hyödyntämään tulevaisuudessa uudelleenkäytön tai uudistamisen kautta.

Tämän lisäksi käänteistekniikkaa hyödynnetään laillisesti ja hyvässä mielessä myös muilla saroilla kuin pelkästään ohjelmistotaloissa. Esimerkiksi opiskelijoita voidaan opettaa ohjelmoimaan siten, että puretaan jonkin valmiin ohjelman lähdekoodi osiin. Näiden osien avulla opiskelijat näkevät, kuinka ohjelma on toteutettu ja siten he oppivat käyttämään samoja suunnittelumenetelmiä [BeL98]. Lisäksi tällä menetelmällä voidaan opettaa opiskelija havaitsemaan erot hyvän ja huonon suunnitteluratkaisun välillä.

7 Pohdintaa

Ohjelmistojen uudistamisen ja takaisinmallinnuksen erilaiset tekniikat ovat mielenkiintoinen ja maailmalla jonkin verran tutkittu aihe. Tässä tutkielmassa on pyritty antamaan yleiskuva molemmista aiheista. Samalla on haluttu tuoda esille takaisinmallinnuksen välineitä sekä UML-kaavioiden ja takaisinmallinnuksen yhteyttä. Tässä loppupohdinnassa tuodaan yhteenvedonomaisesti esille takaisinmallinnuksen käyttöalueita ja mahdollisia kehityskohteita.

Takaisinmallinnusta voidaan hyödyntää monenlaisissa yhteyksissä, niin vanhojen ohjelmistojen uudistamisessa kuin myös uusien ohjelmistojen testauksessa. Vanhojen ohjelmien uudistamisessa takaisinmallinnuksen avulla saadaan selvitettyä ohjelmiston rakennetta ja mahdollisesti uudelleendokumentoimaan sitä. Uusien ohjelmien testauksessa takaisinmallinnusta voidaan käyttää hyväksi siten, että verrataan takaisinmallinnuksen avulla tuotettuja UML-kaavioita suunnitteluvaiheessa luotuihin kaavioihin.

Näiden lisäksi eräs mielenkiintoinen käyttökohde on tietojenkäsittelyn opetus. Ohjelmakoodia analysoimalla ohjelmoinnin tekniikat ja suunnittelun merkitys nousevat selkeästi esille. Siten siis *ymmärrys*, johon käänteistekniikan avulla pyritään, kasvaa. Ohjelmakoodista UML-kaavioita generoivien takaisinmallinnusvälineiden avulla opiskelijoille tarkentuu, mitä UML-kaavioilla todellisuudessa kuvataan ja miksi niiden luominen ohjelmiston suunnitteluvaiheessa on tärkeää.

Käänteistekniikan perustella voidaan luoda erilaisia menetelmiä ja välineitä, joita yritykset voivat käyttää jokapäiväisessä työssä hyväkseen. Tätä onkin jo tehty; esimerkiksi Rational Rose -välineistössä on valmiina käänteistekniikkaa tukevia ominaisuuksia. Suuret tietotekniikkayritykset ovat myös ottaneet käyttöönsä eri tahojen kehittämiä käänteistekniikkavälineitä. Esimerkiksi Nokia on ottanut käyttöönsä Tampereen yliopiston ja Tampereen teknillisen korkeakoulun tutkimuksen tuloksena syntyneitä välineitä.

Olemassa olevat välineet tarjoavat melko monipuolisen valikoiman eri tilanteisiin sopivia toimintoja. Näitä eri toimintoja ja välineitä yhdistelemällä voidaan luoda melko hyvä ketju lähdekoodista eri UML-kaavioihin ja näiden välisiin yhteyksiin. Kun saadaan luotua tietty ketju ohjelman lähdekoodista eri kaavioihin, siitä kyetään ensinnäkin ymmärtämään ohjelmiston staattista ja dynaamista toimintaa. Lisäksi tätä ketjua voidaan käyttää apuvälineenä testauksessa, kuten yllä on kerrottu. Ketjun avulla voi-

daan nähdä ovatko alkuperäiset suunnittelu- ja määrittelyvaiheessa luodut UML-kaaviot toteutuneet lopullisessa ohjelmakoodissa. Eli tällä tavoin kyetään toteamaan jollakin tasolla, toteuttaako ohjelmisto sille alun perin asetetut vaatimukset. Oletettavasti välineiden luomat kaaviot eivät täysin vastaa alkuperäisiä kaavioita, mutta jo selkeät yhdenmukaisuudet auttavat todistamaan vaatimusten ja ohjelmakoodin vastaavuuden.

Tietenkään ketjun luominen ei ole täysin ongelmatonta. Esimerkiksi eri välineet luovat erilaisia kaaviomuotoja, jotka eivät ole suoraan yhteensopivia. Välineet kaipaavat siis lisäkehitystä. Etenkin toivottavaa olisi, että UML-kaavioden välisiä yhteyksiä hyödynnettäisiin enemmän käänteistekniikkavälineissä. Siten etenkin ohjelmien uudelleendokumentointi helpottuisi, samoin kuin laaduntarkkailu.

Testauksen lisäksi käänteistekniikkaa voidaan hyödyntää muissakin laaduntarkkailumenetelmissä. Eräs käyttökohde on ohjelmistojen tarkastusmenettely. Käänteistekniikkavälineiden avulla luotujen ohjelmien rakenteen kuvauksen avulla voidaan tarkastaa esimerkiksi se, onko ohjelmakoodin perusteella tehtävä ohjelmistokuvaus täydellinen. Monestihan jokin luokka tai joku luokan metodeista saattaa jäädä vahingossa dokumentoimatta tai dokumentaatiosta voi löytyä sellainen luokka, jota lopullisessa ohjelmassa ei ole olemassakaan. Ohjelmistokuvausta ja välineen luomaa rakennekuvausta tai rakennekuvauksia voidaan verrata toisiinsa ja huomata siten poikkeumat.

Eräs sovellusalue, jossa käänteistekniikkaa voitaisiin hyödyntää monistakin eri syistä, on terveydenhuollon tietojärjestelmät. Tämä aihe otettiin esille jo Johdanto-osuudessa. Eräs syy tähän on se, että terveydenhuollossa on käytössä paljon jo iäkkäitä järjestelmiä, jotka tarvitsevat kipeästi uudistamista tai korvaavan järjestelmän. Esimerkiksi sairaaloissa paljon käytetty Musti-tietojärjestelmäperhe on peräisin 1980-luvulta [Myk98]. Koska järjestelmät ovat iäkkäitä, tästä yleensä seuraa se, että järjestelmät eivät ole muodoltaan parhaimpia mahdollisia ja lisäksi järjestelmissä käytetyt ohjelmointikielet eivät ole nykyisin kovinkaan monen alan ammattilaisen hallitsemia. Lisäksi järjestelmiä kehittäneet henkilöt ovat useimmiten tavoittamattomissa.

Takaisinmallinnuksen puolesta puhuu myös se, että näiden perinnejärjestelmien sisältä löytyy paljon käyttökelpoista ja jopa erittäin tärkeää tietoa, jota ei löydy mistään muualta. Tieto on kuitenkin sellaista, että se on saatava siirrettyä tulevaan järjestelmään. Tällöin järjestelmää pitää tarkastella ja analysoida tarkoin, että tärkeä tieto saadaan kerättyä talteen. Tulee laatia huolellinen uudistamisstrategia ja lähteä toteuttamaan sitä. Oletettavasti tällaiseen uudistamiseen kuuluu esimerkiksi uudelleendoku-

mentointia ja suunnitteluratkaisun jäljittämistä. Näissä vaiheissa hyvänä apuna toimivat erilaiset käänteistekniikkavälineet. Kuten luvun 4 perusteella voi päätellä, takaisinmallinnusvälineitä on saatavilla useita. Näistä osa on ilmaislevityksessä ja osa taas maksullisia.

Terveydenhuollon järjestelmissä tarvitaan myös tietokantojen uudistamista. Tässä apukeinoksi tulee tietokantojen takaisinmallinnus. Lisäksi on havaittu, että terveydenhuollon järjestelmien uudelleenkäytettävyyttä ja integroitavuutta tulisi lisätä. Integroitavuuden selvittämisessä käänteistekniikalla on oma roolinsa. Sen avulla voidaan selvittää olemassa olevien järjestelmien rakenne ja siten päättää, miten integrointi on mahdollista toteuttaa.

Ohjelmistojen monoliittisuudesta tulisi päästä eroon.. Terveydenhuollon järjestelmien nykyisin olemassa olevat arkkitehtuurit kaipaavat etenkin tässä suhteessa parannusta. Mykkänen on määritellyt tavoitearkkitehtuurin terveydenhuollon järjestelmille [Myk00]. Käänteistekniikan ja sen automatisoitujen menetelmien, esimerkiksi Rigi-pohjaiset välineet, avulla voidaan lähteä selvittämään ja parantamaan ohjelmistojen nykyistä arkkitehtuuria tähtäimenä Mykkäsen määrittelemä tavoitearkkitehtuuri.

Käänteistekniikalle löytyy paljon käyttökohteita. Sen käyttökohteet vaihtelevat laadunparannuksesta ja uudelleendokumentoinnista järjestelmien koko arkkitehtuurin uudistamiseen. Takaisinmallinnuksen ja sen tekniikoiden sekä apuvälineiden kehittäminen vaatii kuitenkin lisää tutkimusta ja panostusta. Tietotekniikka-alan kehittyessä myös takaisinmallinnusta tulee kehittää. Eräs tärkeimmistä kehityskohteista on helppokäyttöisen ja vakaan käänteistekniikkavälineen luominen. Unelmien käänteistekniikka väline olisi sellainen, että sille tuotaisiin syötteenä ohjelman lähdekooditiedostot ja se generoisi yhdellä napin painalluksella niin näkymän ohjelman rakenteesta kuin kaikki ohjelmakoodista poimittavissa tai johdettavissa olevat UML-kaavionäkymät.

Käänteistekniikkaan ja uudistamiseen tulee panostaa myös alan koulutuksessa, jotta tietämys tekniikoista leviäisi myös pieniin ja keskisuuriin tietotekniikka-alan yrityksiin. Ilman hyvää koulutusta tulevia tutkijoita ja kehittäjiä on vaikeaa löytää.

Lähteet

[Aik98] Aiken Peter H.: Reverse Engineering of Data. IBM Systems Journal, Vol. 37 Issue 2, 1998.

[Arn93] Arnold Robert S.: Software Reengineering. IEEE Computer Society Press, 1993.

[Atk99] ATK-sanakirja 1999. Tietotekniikan liitto. Suomen Atk-kustannus Oy. Gummerus Kirjapaino Oy, 1999.

[BeL98] Behrens Brian C., Levary Reuven R.: Practical Legal Aspects of Software Reverse Engineering. Communications of the ACM, Vol. 41, No. 2, February 1998.

[BeR00] Bennett Keith, Rajlich Vaclav: Software Maintenance and Evolution: A Roadmap. Proceedings of the conference on The Future of Software Engineering, Limeric, Ireland. ACM Press, 2000.

[Big89] Biggerstaff Ted J.: Design Recovery for Maintenance and Reuse. IEEE, Computer, Vol. 22, No. 7, July 1989.

[BoV00] Bojic Dragan, Velasevic Dusan: Reverse Engineering of Use Case Realizations in UML. Proceedings of the ACM Symposium on Applied Computing in Como Italy. ACM Press, 2000.

[ChC90] Chikofsky Elliot, Cross James: Reverse engineering and design recovery: A taxonomy. IEEE Software, Vol. 7, No. 1, January/February 1990.

[Eer02] Eerola Anne: Arkkitehtuurit ja suunnittelumallit. Kurssirunko, Kuopion yliopisto, kevät 2002.

[FoS97] Fowler Martin, Scott Kendall: UML Distilled. Applying the Standard Object Modeling Language. Addison Wesley Longman, Inc. 1997.

[FoS00] Fowler Martin, Scott Kendal: UML Distelled. Second Edition. A Brief Guide to the Standard Object Modeling Language. Addison Wesley Longman, Inc. 2000.

[Fuj03] FUJABA-homepage. University of Paderborn. Viitattu 11.03.2003. Saatavilla Internet-osoitteesta http://www.uni-paderborn.de/cs/fujaba .

[GiG93] Gilb Tom, Graham Dorothy: Software Inspection. Addison-Wesley, Pearson Education, 1993.

[GuA99] George Yanbing Guo, Joanne M. Atlee, Rick Kazman, A Software Reconstruction Architecture Method, Software Architecture, Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), 1999. Viitattu 7.11.2002. Saatavilla Internet-osoitteesta: http://www.sei.cmu.edu/staff/rkazman/wicsa1-arm.pdf .

[HaM98] Haikala Ilkka, Märijärvi Jukka: Ohjelmistotuotanto. Suomen Atk-kustannus Oy. Gummerus kirjapaino Oy, Jyväskylä, 1998.

[HaM01] Haikala Ilkka, Märijärvi Jukka: Ohjelmistotuotanto. Suomen Atk-kustannus Oy. RT-Print Oy, Pieksämäki, 2001.

[Har01] Harsu Maarit: Ohjelmien analysoinnin ja uudelleenmuokkauksen tekniikat. Tampere University of Technology, Software Systems Laboratory Reports, Report 25, 2001.

[KaC97] Kazman Rick, Carriere s. Jeromy: Playing Detective: Reconstructing Software Architecture from Available Evidence. Technical Report, Software Engineering Institute (SEI), Carnagie Mellon University, 1997.

[KoE02] Korpela Mikko, Eerola Anne, Korhonen Maritta, Mykkänen Juha, Turunen Pekka: Terveydenhuollon tietojärjestelmien yhteensopivuutta kehitetään kolmikantayhteistyöllä. Sairaaloiden ja terveyskeskuksien ammattilehti. Sairaala, numero 5, 2002.

[KoM96] Koskimies Kai, Männistö Tatu, Systä Tarja, Tuomi Jyrki: SCED: Skenaariot ohjelmistokehityksen apuna. Tietojenkäsittelytieteen Seura ry:n julkaisu. Tietojenkäsittelytiede, numero 8, elokuu 1996.

[Kru01] Kruchten Philippe: The Rational Unified Process: An Introduction, Second Edition. Addison-Wesley, Pearson Education, 2001.

[KuH99] Kung David, Hsia Pei: A Reverse Engineering Approach for Software Testing of Object-Oriented Programs, IEEE Symposium on Application-Specific Systems and Software Engineering and Technology, March 24 - 27, Richardson, Texas, 1999.

[LeO90] LeBlanc Richard J. Jr, Ornburn Stephen B., Rugaber Spencer: Recognizing Design Decisions in Programs. IEEE Software, Vol. 7, No. 1, January/February 1990.

[MüJ00] Müller Hausi, Jahnke Jens, Smith Dennis, Storey Margaret-Anne, Tilley Scott, Wong Kenny: Reverse Engineering: A Roadmap. Proceedings of the conference on The future of Software engineering, 2000.

[Mül97] Müller Hausi: Reverse Engineering Strategies for Software Migration. Proceedings of the 19th international conference on Software engineering, ACM Press, 1997.

[MüR98] Müller Hausi, Reps Thomas, Snelting Gregor: Program Comprehension and Software Reengineering. Proceedings of the Dagstuhl Seminar, March 9th-13th, Seminar No 98101, Report No 204, 1998.

[Myk98] Mykkänen Juha: Selaintekniikkaa käyttävien terveydenhuollon tietojärjestelmien arkkitehtuurit. Pro Gradu -tutkielma. Tietojenkäsittelytieteen ja sovelletun matematiikan laitos. Kuopion yliopisto, 1998.

[Myk00] Mykkänen Juha: Komponentti-FixIT. Terveydenhuollon komponenttipohjainen sovellustuotanto – toiminnallisuus, arkkitehtuuri, siirtymästrategiat ja välineet. Kuopion yliopiston selvityksiä C. Luonnontieteet ja ympäristötieteet 7. Kuopion yliopiston painatuskeskus, 2000.

[RaC99] Rajala Norman, Campara Djenana, Mansurov Nikolai: inSight – Reverse Engineer Case Tool. Proceedings of the 21st international conference on Software engineering , Los Angeles, California, United States. IEEE Computer Society Press, 1999.

[Ros96] Rosenberg Linda H.: Software Re-engineering. Technical report, NASA. 1996. Viitattu 08.08.2002. Saatavilla Internet-osoitteesta: http://satc.gsfc.nasa.gov/support/reengrpt.PDF .

[Sce98] SCED Version 2.1.6. Tampere University of Technology, University of Tampere. 1998. Hankittu 05.03.2003. Saatavilla Internet-osoitteesta: http://www.cs.tut.fi/~tsysta/sced/ .

[SeK01] Selonen Petri, Koskimies Kai, Sakkinen Markku: How to Make Apples from Oranges in UML. Proceedings of the 34th Hawaii International Conference on System Sciences, IEEE Computer Society 2001.

[Ste01] Step-by-Step Guide to Reverse Engineering Code into UML Diagrams with Microsoft Visio 2000. Microsoftin Internet-sivusto.Viitattu 10.12.2002. Saatavilla Internet-osoitteesta: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvisio00/html/revengcode.asp .

[SyK01] Systä Tarja, Koskimies Kai, Müller Hausi: Shimba – an environment for reverse engineering Java software systems. Software – Practice and Experience, 31, 2001.

[Sys00] Systä Tarja: Static and Dynamic Reverse Engineering Techniques for Java Software Systems. Academic Dissertation. Department of Computer and Information Sciences, University of Tampere. Tampereen yliopistopaino Oy, 2000.

[ToP01] Tonella Paolo, Potrich Alessandra: Reverse Engineering of the UML Class Diagram from C++ Code in Presence of Weakly Typed Containers. IEEE International Conference on Software Maintenance, Florence, Italy, November 2001.

[Zam98] Zambelich Keith: Totally Data-Driven Automated Testing, A White Paper. Viitattu 7.11.2002. Saatavilla Internet-osoitteesta: http://www.sqa-test.com/w_paper1.html .

[WeH95] Weide Bruce W., Heym Wayne D., Hollingsworth Joseph E.: Reverse Engineering of Legacy Code Exposed. Proceedings of the 17th international conference on Software engineering , Seattle, Washington, United States. ACM Press, 1995

[WoT95] Wong Kenny, Tilley Scott R., Müller Hausi A., Storey Margaret-Anne D.: Structural Redocumentation: A Case Study, IEEE Software, Special issue on legacy software systems, 1995.