DEVINDER SINGH

MITTAUSOHJELMISTON SPESIFIKAATIOKOMPONENTIN ANALYYSI JA KEHITTÄMINEN

Diplomityö

Tarkastaja: professori Kai Koskimies
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-
neuvoston kokouksessa
09. toukokuuta 2012

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
SINGH, DEVINDER: **MITTAUSOHJELMISTON SPESIFIKAATIOKOMPONENTIN ANALYYSI JA KEHITTÄMINEN**
Diplomityö, 56 sivua
Joulukuu 2012
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Kai Koskimies
Avainsanat: Tietomalli, metatieto, arkkitehtuuri, takaisinmallinnus

Ohjelmistoarkkitehtuurilla on suuri merkitys ohjelmistojen kehityksessä. Se takaa, että ohjelmiston rakenne pysyy modulaarisena ja helposti ymmärrettävänä. Arkkitehtuuri helpottaa järjestelmän testausta, ylläpitoa ja kehittämistä. Ohjelmistoarkkitehtuuri jää useimmiten dokumentoimatta johtuen aikatauluongelmista. Seurauksena on, että eri henkilöt saattavat tehdä hyvin erilaisia olettamuksia siitä, mitkä asiat kuuluvat ohjelmiston arkkitehtuuriin ja mitkä eivät. Pidemmällä aikavälillä arkkitehtuuridokumentaation puuttuminen johtaa ohjelmiston ja sen arkkitehtuurin rapautumiseen.

Ohjelmiston kehitysprosessissa tietomallin dokumentointi on yhtä tärkeä kuin ohjelmistoarkkitehtuurin dokumentointi. Muutos tietomallissa johtaa muutoksiin ohjelmakoodiin ja muutokset ohjelmakoodissa saattavat vaatia muutoksia tietomalliin. Kummassakin tapauksessa koituu kustannuksia.

Tässä työssä perehdytään mittausohjelmiston spesifikaatiokomponenttiin. Kysymyksessä on spesifikaatiokomponentin uudistaminen (re-engineering). Mittausohjelmisto koostuu useasta eri komponentista ja toimii tietyn tuotteen kehityksen tukena. Tämän työn tavoitteena on kuvata spesifikaatiokomponentin arkkitehtuuri käyttäen apuna takaisinmallinnustyökalua. Tämän lisäksi on tarkoitus mallintaa ja analysoida spesifikaatiokomponentin tietomalli. Analyysin perusteella esitetään paranneltu tietomalli.

Aluksi työssä perehdytään mittausohjelmistoon ja sen eri komponentteihin. Tarkoituksena on antaa yleiskuvaus mittausohjelmistosta ja kertoa kunkin komponentin vastuut. Sen jälkeen esitetään lyhyesti työssä sovellettua takaisinmallinnustyökalua ja kuvataan miten sen soveltaminen onnistui. Sitten arvioidaan aikaansaatua arkkitehtuuria käyttäen skenaariopohjaiseen arviointiin perustuvaa menetelmää ja esitetään lyhyesti arvioinnin tulokset. Tämän jälkeen kuvataan spesifikaatiokomponentin tietomalli ja tuodaan esille sen puutteet. Esitettyihin puutteisiin ehdotetaan ratkaisuvaihtoehdot, joiden pohjalta kuvataan paranneltu tietomalli. Sitten esitetään ratkaisuvaihtoehdot, jotka hyödyntävät spesifikaation tietoa raporttien ja raporttipohjien muodostamiseen. Tämä on työn keskeisimpiä tuloksia. Lopuksi kuvataan ja arvioidaan toteutettua ratkaisuvaihtoehtoa.

Tässä työssä saatujen havaintojen perusteella takaisinmallinnustyökalu ei yksinään riitä ohjelmiston ymmärtämisessä ja takaisinmallintamisessa. Työkalujen ohella tarvitaan ohjelmakoodin tutkimista ja analysointia. Analysointia helpottavat ohjelmiston järkevä rakenne ja siinä sovelletut koodauskäytännöt. Komponenttien välinen riippumattomuus helpottaa komponenttien ylläpitämistä ja muutosten tekemistä. Samaan instanssiin liittyvien tietojen hajautus useaan luokkaan ei aina kuitenkaan ole järkevin ratkaisu muutostenhallinnan kannalta.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master’s Degree Programme in Information Technology
SINGH, DEVINDER: ANALYZING AND IMPROVING SPECIFICATION COMPONENT IN MEASUREMENT SOFTWARE
Master of Science Thesis, 56 pages
December 2012
Major: Software engineering
Examiner: Professor Kai Koskimies
Keywords: Data model, metadata, architecture, reverse engineering

Software architecture has an important role in software development. It ensures that the structure of the software remains modular and easy to understand. Architecture makes system testing, maintenance and development easier. Software architecture is often not documented due to scheduling restrictions. The result is that different people may make different assumptions about what falls within the software architecture and what not. In the longer term the lack of architecture documentation leads to the erosion of the software and its architecture.

In the software development process data model documentation is as important as software architecture documentation. A change in the data model leads to changes in the code and changes to the code may require changes in the data model. In both cases, the cost is affected.

This thesis focuses on the specification component of a measurement software. An aim is to re-engineer the specification component. The measurement software contains several components and it is used for a specific product development. An aim of this work is to model and analyze data model of specification component. Based on the analysis the thesis presents an improved data model and describes the further development ideas. Moreover, we intend to describe the architecture of the specification component. A reverse engineering tool is used to analyze the architecture.

This thesis is first going to introduce the measurement software and its components. The purpose is to give a general description of the measurement software, and describe each component's responsibility. After that a reverse engineering tool is presented briefly and a short summary is given on how suitable the tool was. Then, the architecture is evaluated using a scenario-based evaluation method and the results achieved are presented briefly. In addition, the data model of the specification component is described and its deficiencies are highlighted. Solutions to the deficiencies are proposed and based on the solutions an improved data model is described. Then, solutions that take advantage of the specification information to form reports and report templates are described. These are the main results of the work. Lastly, implementation of the one solution is described and analyzed.

The results of this work show that a reverse engineering tool alone is not sufficient to understand software. Beside tools, we need to study and analyze the code. The structure of the software and the applied practices help in the analysis. Independence between components makes maintenance more easier. Spreading information across multiple classes, however, is not always the most sensible solution in terms of management of change.

ALKUSANAT

Tämän diplomityön on tarjonnut ja rahoittanut Atostek Oy. Kiitokset Atostekille työn aiheesta ja rahoituksesta.

Haluan kiittää työn tarkastajaa professori Kai Koskimiestä ja työn ohjaajaa Risto Pitkästä loistavasta työn ohjauksesta ja työn sisältöä koskevista neuvoista. Kiittäisin Harri Järveä, Petteri Roposta ja Tuomas Kujalaa hyvien neuvojen tarjoamisesta. Kiittäisin myös Tuomas Fjällströmiä työni oikolukemisesta ja Atostek Oy:n kaikkia työntekijöitä mukavan ja opettavaisen työympäristön tarjoamisesta.

Erityiskiitokset vanhemmilleni, veljelleni ja hänen puolisolleen, joiden tuki ja kannustus on vienyt minua eteenpäin urallani. Lopuksi haluan kiittää puolisoani kaikesta tuesta sekä veljenpoikaani ja poikaani mielenkiintoisista hetkistä.

Pirkkalassa 06.11.2012

Devinder Singh

SISÄLLYS

1 Johdanto....1
2 Mittausohjelmiston ja sen komponenttien kuvaus....3
2.1 Yleiskuvaus mittausohjelmistosta....3
2.2 Mittausohjelmiston rakenne....4
2.2.1 Protokollakomponentti....4
2.2.2 Mittauskomponentti....4
2.2.3 Raporttikomponentti....5
2.2.4 Spesifikaatiokomponentti....6
2.2.5 Hallintatyökalu....6
3 Tietomallinnus....8
3.1 Yleistä....8
3.2 Metatieto....9
3.3 Rakenteiset dokumentit....10
3.3.1 XML....10
3.3.2 XSLT....11
4 Takaisinmallinnus....14
4.1 Yleistä....14
4.2 Suunnitteluratkaisun jäljittäminen ja uudelleendokumentointi....15
4.3 Staattinen ja dynaaminen takaisinmallinnus....15
4.4 Yleiskuvaus takaisinmallinnustyökaluista....16
4.5 Takaisinmallinnustyökalun valinta....17
5 Spesifikaatiokomponentin arkkitehtuuri....18
5.1 Työnkuvaus....18
5.2 Takaisinmallintaminen....18
5.3 Arkkitehtuurin yleiskuvaus....20
5.4 Spesifikaatiokomponentin arkkitehtuuriratkaisut....21
5.5 Komponenttien välinen kommunikointi....23
6 Arkkitehtuurin arviointi....24
6.1 ATAM....24
6.2 Arvioinnin tarkoitus spesifikaatiokomponentissa....25
6.3 Spesifikaatiokomponentin skenaariot....25
6.4 Arviointi....26
6.5 Arvioinnin tulokset....28
7 Spesifikaatiokomponentin tietomalli....29
7.1 Nykyinen tietomalli....29
7.2 Tietomallin arviointi....33
8 Jatkokehitysideat....37
8.1 Tämänhetkinen raportin muodostamisprosessi....37

8.2 Raportin muodostaminen spesifikaatiosta....39
8.3 Spesifikaation muuttaminen raporttipohjaksi....41
8.4 Spesifikaatiopohjan osien käyttäminen raporttipohjassa....43
8.5 Johtopäätökset....45
9 Toteutuksen arviointi....47
9.1 Ratkaisuvaihtoehdon toteutuksen kuvaus....47
9.2 Toteutetun ratkaisuvaihtoehdon arviointi....48
9.2.1 Arvioinnin suoritus....48
9.2.2 Arvioinnin tulokset....49
9.3 Vertailu ratkaisuvaihtoehtoon kolme....51
10 Yhteenveto....52
Lähteet....54
Liite 1: XSLT-muunnossäännöt

MERKINNÄT, TERMIT JA NIIDEN MÄÄRITELMÄT

ATAM	Architecture Tradeoff Analysis Method on menetelmä ohjelmistoarkkitehtuurin arvioimiseen. Menetelmä on tarkoitettu useiden eri laatuominaisuuksien arviointiin.
CSS	Cascading Style Sheetsin avulla voidaan määritellä dokumentin visuaalista esitystapaa.
Dublin Core	Metatietosanastostandardi informaatioresurssien kuvaamiseen.
DTD	Document Type Definition on rakenteisen dokumentin rakennetta määrittävä kieli. Sen avulla määritellään dokumentin sallitut elementit ja attribuutit.
HTML	HyperText Markup Language on merkkauskieli, jonka avulla voidaan kuvata hypertekstiä eli hyperlinkkejä sisältävää tekstiä.
Kursiivi	Ensimmäisen kerran tekstissä esiintyvät uudet termit merkitään kursiivilla.
MPM	Maintenance Prediction Method on menetelmä ohjelmistoarkkitehtuurin arvioimiseen. Menetelmä on tarkoitettu ylläpidettävyyteen liittyvien laatuominaisuuksien arviointiin.
OCL	Object Constraint Language on määrittelykieli rajoitteiden kuvaamiseen.
OML	Open Modeling Language on graafinen mallinnuskieli, joka on kehitetty kuvaamaan olio-pohjaisia järjestelmiä.
RDF	Resource Description Framework on W3C:n standardoima malli tiedon vaihtamiseen web-ympäristössä.
Reverse engineering	Analysointiprosessi, jonka avulla tunnistetaan järjestelmän eri osia ja näiden osien välisiä suhteita sekä esitetään järjestelmä korkeammalla abstraktiotasolla.
SAAM	Software Architecture Analysis Method on menetelmä ohjelmistoarkkitehtuurin arvioimiseen. Menetelmä on tarkoitettu muunneltavuuteen ja toiminnallisuuteen liittyvien laatuominaisuuksien arviointiin.
SGML	Standard Generalized Markup Language on metakieli, jonka avulla voidaan määritellä merkkauskieliä.
Takaisinmallinnus	Katso reverse engineering määritelmä.

`Tasalevyinen kirjasintyyppi`	Luokkien ja prosessien nimet merkitään tasalevyisellä kirjasintyypillä.
UML	Unified Modeling Language on graafinen mallinnuskieli, joka sisältää 13 erilaista kaaviota. Kaavioiden avulla kuvataan järjestelmän ja ohjelmiston rakennetta, käyttäytymistä ja vuorovaikutusta.
W3C	World Wide Web Consortium on kansainvälinen yhteisö, joka kehittää ja ylläpitää WWW:n standardeja.
XML	Extensible Markup Language on rakenteellinen kieli, joka erottaa dokumentin loogisen ja fyysisen rakenteen.
XPath	XML Path Language on kyselykieli XML-dokumenttien osien osittamiseen ja tiedon hakuun.
XSLT	Extensible Stylesheet Language on sääntöpohjainen muunnoskieli.

1 JOHDANTO

Järjestelmän ohjelmistoarkkitehtuurin dokumentointi on tärkeää ohjelmistoja kehitettäessä. Mitä laajemmasta ja monimutkaisemmasta järjestelmästä on kyse, sitä tärkeämpi on sen arkkitehtuurin dokumentointi. Ohjelmiston arkkitehtuuri määrittelee järjestelmän keskeiset osat ja niiden välisen vuorovaikutuksen sekä sisältää tehdyt ratkaisut ja käsitteet. Arkkitehtuuri helpottaa eri osapuolten välistä kommunikointia sekä järjestelmän testausta, ylläpitoa ja kehittämistä. [12, s. 18–19]

Johtuen aikatauluongelmista ohjelmistoarkkitehtuuri jää useimmiten dokumentoimatta tai se dokumentoidaan liian abstraktilla tasolla. Mikäli arkkitehtuuria ei ole dokumentoitu, seurauksena on, että eri henkilöt saattavat tehdä hyvin erilaisia olettamuksia siitä, mitkä asiat kuuluvat ohjelmiston arkkitehtuuriin ja mitkä eivät. Ennemmin tai myöhemmin arkkitehtuuridokumentaation puuttuminen johtaa ohjelmiston ja sen arkkitehtuurin rapautumiseen. [12, s. 20]

Tietomallin dokumentointi ja ylläpitäminen on yhtä tärkeä osa ohjelmiston kehitysprosessia kuin järjestelmän ohjelmistoarkkitehtuurin dokumentointi. Tietomalli määrittelee järjestelmän tietosisällön rakenteen, joka kuvataan käsitteiden ja niiden välisten suhteiden avulla. Pienikin muutos tietomallissa heijastuu ohjelman koodiin saakka, jolloin siitä koituu kustannuksia. Täysin sama pätee päinvastoin eli muutokset ohjelmakoodissa saattavat vaatia muutoksia tietomalliin. Mikäli tietomallista ei ole olemassa dokumenttia, on hankalaa lisätä uusia käsitteitä ja tehdä muutoksia olemassa olevaan tietokantarakenteeseen. Tämä johtuu osittain siitä, että rakenne on hankalasti hahmotettavissa ja tieto saattaa olla hajautettu useaan eri tietokantaan. [13, s. 20–21]

Tämän työn tarkoituksena on tarkastella mittausohjelmistoa, joka koostuu useasta eri komponentista. Mittausohjelmisto toimii tietyn tuotteen kehityksen tukena. Tuotekehitysprosessin lähtökohtana ovat tuotteelle asetetut vaatimukset, ja niiden toteutumista seurataan mittaustulosten perusteella. Mittaustulokset raportoidaan, jolloin niistä saadaan yksityiskohtaista tietoa siitä, miten hyvin tarkasteltava tuote toteuttaa annetut vaatimukset. Tämän tiedon perusteella kehitystiimi voi parannella tuotetta kunnes se on riittävän hyvä.

Mittausohjelmisto koostui alun perin vain yhdestä komponentista, joka oli mittauskomponentti. Mittauskomponentin tarkoituksena on suorittaa määriteltyjä toimenpiteitä tuotteelle ja kerätä mittausdataa. Tämän jälkeen siihen integroitiin raportointikomponentti, jonka avulla raportoidaan tehtyjä mittauksia taulukko- ja kuvaajamuodossa. Kehityksen jatkuessa ohjelmistoon integroitiin spesifikaatiokomponentti, jonka avulla määritetään tuotteen eri ominaisuuksille asetetut vaatimukset spesifikaatiomuodossa.

Tieto liikkuu eri komponenttien välillä, mutta dokumenttia tehdyistä päätöksistä tai suunnitteluratkaisuista ei ole olemassa. Tiedon rakenne on muodostunut vähitellen tällä tavoin evoluutiomaisesti.

Tässä työssä kysymyksessä on mittausohjelmiston spesifikaatiokomponentin uudistaminen (*re-engineering*). Tarkoituksena on kuvata spesifikaatiokomponentin arkkitehtuuri, eli selvitetään komponentin sidokset muihin mittausohjelmiston komponentteihin ja niiden väliset vuorovaikutustavat. Arkkitehtuurin kuvaamiseen käytetään apuna takaisinmallinnustyökalua. Staattisen takaisinmallinnuksen (*reverse engineering*) ja takaisinmallinnustyökalun avulla aikaansaatua arkkitehtuuria arvioidaan käyttäen skenaariopohjaiseen arviointiin perustuvaa menetelmää. Arvioinnin perusteella pyritään löytämään arkkitehtuurin heikkoudet ja vahvuudet. Arkkitehtuurin lisäksi mallinnetaan ja analysoidaan spesifikaatiokomponentin tietomalli. Analyysin paljastamiin puutteisiin ehdotetaan ratkaisuvaihtoehdot, joiden perusteella esitetään paranneltu tietomalli. Tietomallin uudelleenmallinnuksen lisäksi esitetään ratkaisuvaihtoehdot, jotka hyödyntävät spesifikaation tietoa raporttien ja raporttipohjien muodostamiseen. Tämän lisäksi toteutetaan yksi kyseisistä ratkaisuvaihtoehdoista.

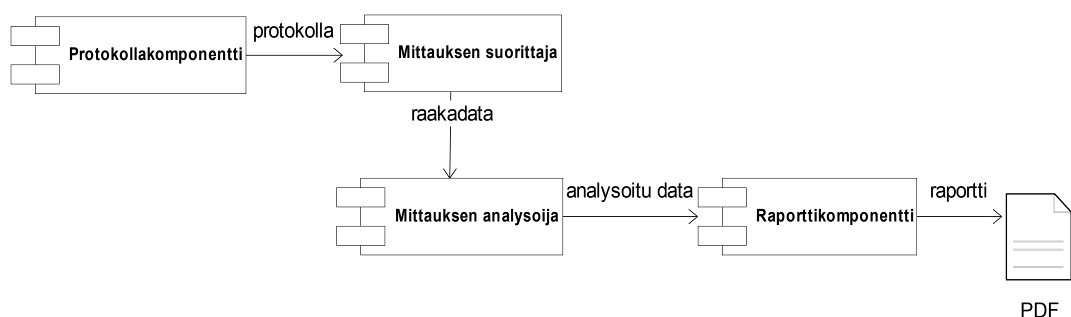
Luvussa 2 perehdytään mittausohjelmiston toimintaympäristöön ja siinä oleviin komponentteihin. Luvussa 3 esitellään tietomalliin liittyvä käsitteet ja notaatiot. Luvussa 4 esitellään yleisesti takaisinmallintamista ja takaisinmallinnustyökalut, jotka soveltuisivat käytettäväksi. Luvussa 5 kerrotaan miten valittu takaisinmallinnustyökalu soveltui, kuvataan spesifikaatiokomponentin arkkitehtuuria ja siinä käytetyt ratkaisupäätökset. Luvussa 6 arvioidaan spesifikaatiokomponentin arkkitehtuuria ja esitellään arvioinnin tulokset. Luvussa 7 esitellään spesifikaatiokomponentin tietomalli ja sen parannukset. Luvussa 8 esitellään ratkaisuvaihtoehdot, jotka muodostavat raportteja ja raporttipohjia spesifikaatio tiedon perusteella. Luvussa 9 kuvataan ja arvioidaan ratkaisuvaihtoehdon toteutusta. Luvussa 10 esitellään yhteenveto työn tuloksista.

2 MITTAUSOHJELMISTON JA SEN KOMPONENTTIEN KUVAUS

2.1 Yleiskuvaus mittausohjelmistosta

Mittausohjelmisto, mittalaitteet ja mittausympäristö muodostavat mittausjärjestelmän, joka mahdollistaa tuotteen ominaisuuksien mittaamisen. Mittausympäristö kuvaa mitattavan tuotteen ominaisuuksiin ulkoisesti vaikuttavia olosuhteita. Näitä olosuhteita kuvataan erillisillä parametreilla, joita voivat olla esimerkiksi lämpötila tai ilmanpaine. Mittauksen aikana kontrolloidaan mittausympäristöä ja ohjataan mittalaitteita, jotka mittaavat tuotteen ominaisuuksia.

Mittausohjelmisto sisältää mittauksen suunnittelun, suorituksen, raakadatan analysoinnin ja raportin luomisen. Lueteltu järjestys on tavanomainen suoritusjärjestys, mutta toiminnot ovat toteutettu toisistaan erillisinä ohjelmina, jolloin useita mittaukseen liittyviä toimintoja voi suorittaa samanaikaisesti. Esimerkiksi voidaan luoda uusi mittaus tai tarkastella edellisen mittauksen tuloksia, kun mittaus on jo käynnissä. Eri osille voidaan antaa seuraavat nimet: protokollakomponentti, raporttikomponentti ja mittauskomponentti, joka kostuu mittauksen suorittajasta ja mittauksen analysoijasta. Kuvassa 2.1 on esitetty tavanomainen suoritusjärjestys sekä ohjelmien saamat syötteet ja tulosteet.



Kuva 2.1. *Yleiskuva mittausohjelmiston suoritusjärjestyksestä.*

Protokollakomponentilla suunnitellaan mittaus eli määritetään suoritettavat mittausvaiheet ja niiden parametrit. Mittauksen suorittaja tulkitsee protokollan vaiheet ja suorittaa mittaukset ohjaamalla tarvittavia mittalaitteita. Tuloksena saadaan mittauksesta raakadata. Mittauksen analysoija lukee raakadatan ja suorittaa sille määritettyjä laskentoja. Laskentatavat määritellään mittauksen suunnittelun yhteydessä protokollaan. Raportti-

komponentin avulla koostetaan yksi tai useampi raportti mittauksen analysoidusta datasta. Raporttiin voidaan myös sisällyttää mittauksen raakadata.

Edellä mainittujen osien lisäksi mittausohjelmisto sisältää spesifikaatiokomponenttin. Spesifikaatiokomponentin avulla määritellään verifiointidokumentti, joka sisältää arvorajat tuotteen ominaisuuksille. Vertaamalla mittaustuloksia verifiointidokumentissa oleviin arvorajoihin voidaan päätellä, täyttääkö tuotteen ominaisuus sille asetetut vaatimukset.

2.2 Mittausohjelmiston rakenne

2.2.1 Protokollakomponentti

Protokollakomponentti on mittausten suunnitteluun tarkoitettu komponentti. Komponentin avulla voidaan uusien mittausten suunnittelun lisäksi muokata ja poistaa olemassa olevia mittauksia. Luotavaan mittaukseen määritellään mittauksen tyyppi, mittausympäristöön liittyvät parametrit, mittauksen vaiheet ja laskentatavat. Mittaus luodaan luetellussa järjestyksessä. Mittauksen tyypin avulla mittauksen suorittaja tulkitsee mitä mittalaitteita tarvitsee ohjata. Yksi mittaus koostuu yhdestä tai useammasta mittausvaiheesta. Mittalaitteiden ohjaukseen ja mitattavan asentoon liittyvät parametrit määritellään kunkin mittausvaiheen yhteydessä. Viimeisenä on laskentatavan valintavaihe.

Laskentatapa on toimenpide, jota sovelletaan raakadataan mittauksen jälkeen. Toimenpide voi olla esimerkiksi arvon muuttaminen mittayksiköstä toiseen tai uuden arvon laskeminen. Aina ei tarvitse valita laskentatapaa, mikäli mittaustulos on sellaisenaan käytettävissä, mutta useimmiten näin ei kuitenkaan ole. Laskentatavasta riippuen parametrina voi olla joko yksittäinen mittaus tai joukko mittauksia.

2.2.2 Mittauskomponentti

Mittauskomponentti sisältää sekä mittauksen suorittajan että analysoijan. Mittauskomponentti suorittaa mittausprosessin mitattavalle tuotteelle. Mittausprosessi voidaan kuvailla tapahtumaketjuna, joka koostuu vaiheista. Jokaisen vaiheen aikana suoritetaan osa prosessista. Mittausprosessi alkaa valmisteluvaiheella, jonka jälkeen siirrytään mittausvaiheeseen, analyysivaiheeseen ja lopuksi raportointivaiheeseen.

Valmisteluvaiheessa valmistellaan mittalaitteet, mitattava tuote ja mittausympäristöä mittauksen suoritusta varten. Mittauksen suorittaja valitsee mitattavan tuotteen ja suoritettavan mittauksen. Jos mittalaitteet tarvitsee kalibroida, se tehdään tämän vaiheen yhteydessä.

Mittausvaiheessa tuotteelle suoritetaan varsinaiset mittaukset ja vaiheen tuloksena saadaan mittausdataa tuotteen ominaisuuksista. Mittausvaihe koostuu useista mittaustapahtumista. Mittaustapahtumassa tuotteelle suoritetaan yksittäinen mittaus, jonka tulokseksi saadaan raakadataa. Tuloksena saatua dataa tallennetaan tietokantaan myöhempää

käyttöä varten. Mittaustapahtumaan liittyy erilaisia parametreja, esimerkiksi mitattavan tuotteen tilaa, mittausympäristöä, mittaus- ja laskentatapaa kuvaavia parametreja.

Analyysivaiheessa muunnetaan raakadata mittaustulokseksi käyttäen mittaustapahtumassa määriteltyä laskentatapaa. Laskentatapa kuvaa millaisia laskentoja mittaustapahtuman tulokselle suoritetaan. Analysoinnissa käytetään raakadatan lisäksi tuotteeseen, mittausympäristöön ja mittaustapaan liittyviä parametreja. Analysoitua dataa tallennetaan myös tietokantaan. Mittauksen raakadataa ja analysoituja tuloksia käytetään raporttien muodostamiseen.

Raporttivaiheessa muodostetaan raakadatasta ja analysoiduista tuloksista raportti. Raportissa esitetään erilaisia tietoja, kuten mittauksen nimi, mittaustulos joko numeerisena arvona tai kuvana sekä tuotteeseen ja mittausympäristöön liittyviä parametreja.

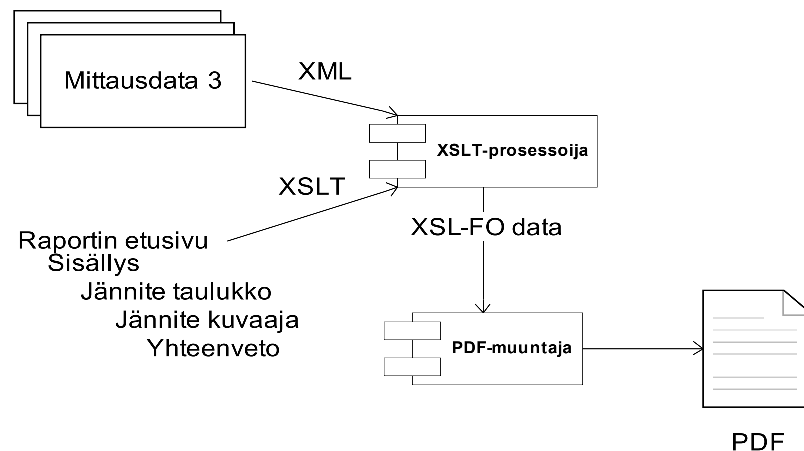
2.2.3 Raporttikomponentti

Raporttikomponentin avulla muodostetaan raporttipohjia ja koostetaan mittauksista raportteja näiden avulla. Raporttipohja koostuu useista raporttimallipohjista. Yksittäinen raporttimallipohja on XSLT-tyylitiedosto, joka sisältää muunnossääntöjä, joiden avulla mittausdata muunnetaan uuteen muotoon. *XSLT (Extensible Stylesheet Language Transformations)* on XSL (Extensible Stylesheet Language) -kieliperheeseen kuuluva sääntöpohjainen muunnoskieli. Se tarjoaa mahdollisuuden käsitellä XML-dokumentin dataa. *XML (Extensible Markup Language)* on rakenteellinen kieli, joka erottaa dokumentin loogisen ja fyysisen rakenteen. XML:stä ja XSLT:stä on kerrottu tarkemmin kohdassa 3.3.

Raporttimallipohjat jaetaan kahteen kategoriaan: yleisiin ja mittauskohtaisiin mallipohjiin. Yleiset mallipohjat sisältävät raportin yleisiä osia, kuten etusivun, sisällysluettelon, luku- ja alilukuotsikot. Mittauskohtaiset mallipohjat sisältävät mittaukseen liittyvien tietojen muunnossäännöt, kuten mittaustulosten esittäminen taulukko- tai kuvaajamuodossa.

Raporttipohjan koostaminen useista mallipohjista tuo monia etuja yhteen mallipohjaan verrattuna. Yhtenä etuna on, että päällekkäisen tiedon määrä vähenee, kun samaa mallipohjaa voidaan käyttää useassa eri raporttipohjassa. Toisena etuna on muutostenhallinta, sillä muutos voidaan tehdä yksittäiseen mallipohjaan usean eri raporttipohjan sijaan.

Kuvassa 2.2 on esitetty tarkempi XSLT-muunnoksen etenemisprosessi.



Kuva 2.2. *PDF-muotoisen raportin muodostamisprosessi.*

Valitut mittausdatat haetaan tietokannasta XML-muotoisena, jonka *XSLT-prosessoija* saa yhtenä sisääntulona. Toisena sisääntulona prosessoija saa raporttipohjan. Näiden avulla prosessoija muodostaa XSL-FO-muotoista dataa, joka annetaan *PDF-muuntajalle*. PDF-muuntaja tuottaa lopullisen PDF-dokumentin.

2.2.4 Spesifikaatiokomponentti

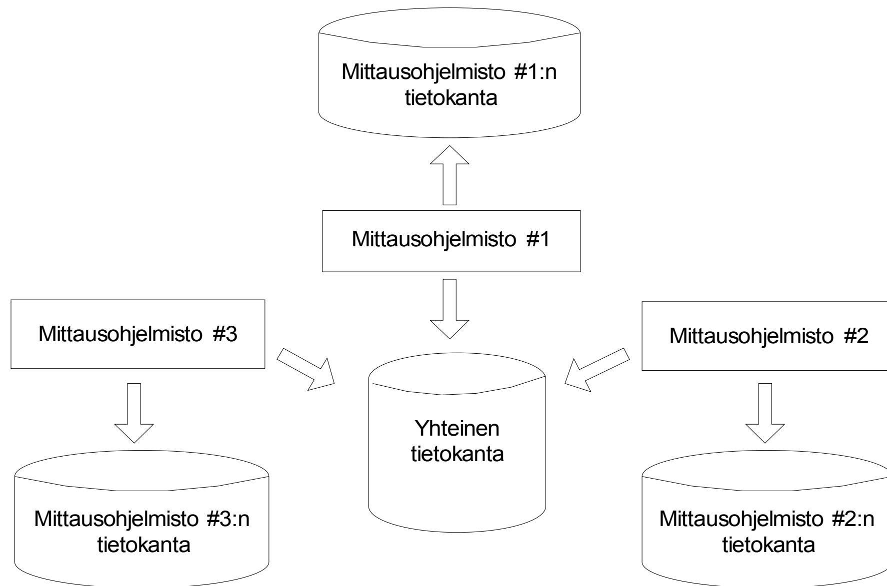
Spesifikaatiokomponentin avulla muodostetaan spesifikaatiopohjia ja spesifikaatioita eli verifiointidokumentteja. Spesifikaatiopohjaan määritellään erilaisten elementtien avulla spesifikaation rakenne. Rakenteeseen kuvataan tuotteen eri ominaisuudet. Spesifikaatiopohja on tyhjä, kunnes se liitetään spesifikaatioon. Spesifikaatiota luotaessa valitaan spesifikaatiopohja, jolloin spesifikaatio noudattaa kyseisen pohjan rakennetta. Spesifikaatioon on mahdollista täydentää ominaisuuksille arvorajat, jolloin saadaan aikaan dokumentti, jota voidaan käyttää ominaisuuksien verifioinnissa.

Verifiointidokumentin ja mittaustulosten avulla voidaan päätellä, täyttääkö tuotteen ominaisuus sille asetetut vaatimukset. Johtopäätöksen tekemistä helpotetaan yhdistämällä mittaustulokset ja verifiointiarvot raporttiin. Verifiointiarvot luetaan mukaan raporttiin XSLT-muunnossääntöjen avulla. Spesifikaatio, josta arvot luetaan valitaan, raportin luonnin yhteydessä.

2.2.5 Hallintatyökalu

Hallintatyökalu on mittausohjelmistosta erillinen komponentti, jonka avulla käsitellään tietokannassa olevia tietueita, kuten mittauslaboratorio, protokollia ja kaikkea muuta, jota mittausohjelmiston avulla luodaan. Mittauspisteiden erottelua tarvitaan, koska eri mittauspisteissä sijaitsevat erilaiset mittauslaitteet. Mittausta suorittaessa käyttäjä valitsee, missä mittauspisteessä mittausta ollaan suorittamassa. Komponentin avulla vaihdetaan myös käytettävää tietokantaa.

Tietokanta on jaettu kahteen osaan: yhteiseen ja ohjelmistokohtaiseen tietokantaan. Yhteisen tietokannan avulla voidaan jakaa tietoa eri palvelimissa toimivien mittausohjelmistojen kesken. Kuva 2.3 selventää tietokantojen käyttöä. Jokainen mittausohjelmisto käyttää yhteistä tietokantaa ja lisäksi niillä on myös oma tietokantansa käytössä.



Kuva 2.3. *Jako yhteiseen ja tuotekohtaiseen tietokantaan.*

Hallintatyökalun avulla luodaan uusia mittauspisteitä ja mittalaitteita sekä muokataan niiden tietoja. Komponenttia käytetään myös protokollien tilan muuttamiseen. Protokollilla on useita tiloja: *Work, Draft, R&D* ja *Official*. Work tilassa olevaa protokollaa on mahdollista muokata, mutta sillä ei voi suorittaa mittauksia. Mittausten suorittamista varten protokollan tila tulee olla joko Draft tai R&D. Official tila kertoo, että protokolla on hyväksytty viralliseen verifiointikäyttöön. Komponenttia voidaan käyttää protokollien tilojen muuttamisen lisäksi myös niiden poistamiseen.

3 TIETOMALLINNUS

3.1 Yleistä

Ohjelmiston tietosisältöä kuvataan tietomallilla, joka on abstrakti kuvaus sovellusalueen käsitteistä, niiden välisistä suhteista, ominaisuuksista ja rajoitteista. Tietomalli toimii järjestelmän tietosisällön kuvauksena ja kommunikoinnin apuvälineenä. Tietomalli on käytettävästä tietokantahallintajärjestelmästä riippumaton eli toteutusnäkökohdat jätetään huomiotta. Näin pyritään saamaan yhteinen näkemys kohdealueesta eri sidosryhmien välille, jotta päätelmien tekeminen olisi helpompaa ja selkeämpää. [2, s. 418–419]

Tietomalli on suhteellisen pieni osa koko järjestelmää. Kuitenkin siihen tehtävien pienten muutosten vaikutus saattaa olla merkittävä järjestelmän kannalta. Suunnittelulla on tärkeä rooli myös tietomallin yhteydessä. Hyvin suunnitellun tietomallin ylläpitäminen on helpompaa ja kustannustehokkaampaa. Jo pienillä muutoksilla voidaan luoda merkittäviä säästöjä. [20, s. 8–10]

Hyvän tietomallin kriteerit ovat muun muassa seuraavat: täydellisyys, oikeellisuus, ymmärrettävyys, riittävyys ja normaalisuus. Täydellisyydellä tarkoitetaan, että tietomalli tukee tarvittavan tiedon tallentamista, toisin sanoen kaikki kohdealueen olennaiset piirteet ovat otettu huomioon. Oikeellisuudella tarkoitetaan, että tietomalli on todellisuuden, eli kohdealueen, vääristämätön esitys. Ymmärrettävyydellä tarkoitetaan, että tietomalli kuvataan siten, että se on helposti luettavissa ja ymmärrettävissä. Riittävyydellä tarkoitetaan, että tietomalli kuvaa kaiken tarvittavan riittävän tarkasti eikä lisämääreitä tarvita. Normaalisuudella tarkoitetaan, että tietomalli ei sisällä toisteista tietoa. Uusia vaatimuksia tulee ja vanhat muuttuvat. Nämä aiheuttavat muutoksia tietomalliin.

Tietomallintamiseen on olemassa useita erilaisia mallinnustapoja ja tekniikoita. Yleensä niissä käytetään jotain kuvauskieltä, kuten luonnollista, formaalia tai graafista kieltä. Graafisista kielistä eniten käytettyjä ovat Chenin notaatio sekä *OML:n (Open Modeling Language)* ja *UML:n (Unified Modeling Language)* luokkakaavionotaatiot. Tässä työssä käytetään UML:n luokkakaavionotaatiota, koska sitä käytetty laajamittaisesti tietokantojen yhteydessä ja se on tutuin edellä mainituista notaatioista.

Kaikkia sovellusalueen rajoitteita ei ole järkevää kuvata graafisilla kuvausmenetelmillä. Liian yksityiskohtaisten rajoitteiden kuvaaminen vaikuttaa tietomallin selkeyteen ja ymmärrettävyyteen. Tästä syystä graafista kuvausmenetelmää täydennetään usein muilla tavoin. Yleensä käytetyin tapa on tekstuaalinen kuvaustapa, kuten luonnollinen

kieli tai rajoitekieli. Rajoitekieliä ovat muun muassa *OCL (Object Constraint Language)* ja UML:n tekstuaalinen rajoitekieli. [2, s. 418-419]

3.2 Metatieto

Metatieto on yleisen määritelmän mukaan tietoa tiedosta eli kuvailevaa ja määrittävää tietoa aineistosta. Metatieto auttaa aineiston identifioimisessa, hakemisessa ja hallitsemisessa. Metatiedoille on useita erilaisia luokitteluja. Yleisen luokittelun perusteella metatiedot jaetaan kolmeen kategoriaan: kuvailevaan, rakenteelliseen ja hallinnolliseen.

Kuvaileva metatieto kuvaa sisällön merkitystä. Otsikko, tiivistelmä, kirjoittaja ja avainsanat ovat esimerkkejä kuvailevasta metatiedosta. Näiden tietojen avulla aineisto on helposti haettavissa ja tunnistettavissa. Rakenteellinen metatieto ilmaisee kuinka aineistossa olevat objektit ovat järjestetty. Esimerkiksi rakenteellinen metatieto ilmaisee kuinka kirjan sivut ovat järjestetty, jotta niistä muodostuu kappaleita. Hallinnollinen metatieto antaa tietoa aineiston hallitsemiseen. Aineiston luontipäivämäärä, tyyppi ja oikeudet ovat esimerkkejä hallinnollisesta metatiedosta. [22, s. 3; 17, s. 1–6]

Metatieto on monipuolinen väline, jota voidaan hyödyntää yksittäisen aineiston, kokoelman tai laajan aineiston jonkin osa-alueen kuvaamiseen. Metatiedon luonnin tärkein syy on relevantin tiedon saaminen aineistosta. Tämän lisäksi metatieto helpottaa aineiston järjestämisessä, yhteensopivuudessa, integroimisessa, identifioimisessa ja tukee aineiston arkistointia ja säilytystä. [22, s. 3–4]

Säilytykseen liittyy monia uhkia. Alun perin aineistoja alettiin muuttamaan digitaaliseen muotoon, jotta voitaisiin parantaa aineistojen saatavuutta ja käytettävyyttä. Myös digitaaliseen tietoon liittyy monia aineistoa vaarantavia uhkia, kuten laite-, sovellus- ja tiedonsiirtovirheet. Digitaalinen tieto voi joko korruptoitua tai muuttua tarkoituksellisesti tai tahattomasti. Aineistoon liittyviä uhkia voidaan pyrkiä estämään useiden eri tekniikoiden avulla, kuten migraation ja emuloinnin avulla. Kuitenkin avainasemassa on metatieto, joka takaa aineiston pitkäaikaissäilyvyyden ja -käytettävyyden. Metatiedon avulla voidaan jäljittää mistä aineisto on alun perin tullut ja miten se on ajan myötä muuttunut. [22, s. 3–4]

Useiden vuosien ajan metatietoja on pyritty standardoimaan. Standardointi tuli välttämättömäksi, kun metatietojen tallentaminen, käsittely ja käyttö siirrettiin tietokoneille. Internetin myötä huimaa vauhtia kasvavat tietosisällöt ovat vaatineet metatietostandardien kehittämistä, jotta tietokoneohjelmat pystyvät hyödyntämään metatietoa. Metatietojen standardointia varten on kehitetty useita suosituksia. Niitä voidaan hyödyntää kehitettäessä metatietostandardia tietyn ympäristön tarpeisiin. Tärkeimpiä suosituksia ovat XML*, RDF (Resource Description Framework)* ja *Dublin Core*. XML ja RDF tarjoavat mallin ja syntaksin metatietojen esittämiseen. Dublin Core puolestaan on suositus verkkojulkaisuille, ja sitä on käytetty organisaatio- ja sovellusaluekohtaisten metatietoskeemojen kehittämisessä. [17, s. 7; 21, s. 36]

3.3 Rakenteiset dokumentit

3.3.1 XML

XML (Extensible Markup Language) on rakenteellinen kieli, joka erottaa dokumentin loogisen ja fyysisen rakenteen. Rakenteisten dokumenttien kirjoittamisen lisäksi XML:n avulla voidaan määritellä merkkauskieliä. *W3C:n (World Wide Web Consortium)* XML Working Group aloitti XML:n kehityksen vuonna 1996 ja ensimmäinen versio (XML 1.0) julkaistiin vuonna 1998. Tämä versio on ollut siitä lähtien W3C:n suositus. Toinen versio (XML 1.1) julkaistiin vuonna 2004, mutta erot ovat vähäiset. [3]

XML on johdettu *SGML:stä (Standard Generalized Markup Language, ISO-8879)*, joka on ollut *ISO-standardi (International Organization for Standardization)* vuodesta 1986. SGML:n avulla voidaan määritellä merkkauskieliä. Tunnetuin ja käytetyin SGML:llä määritetty merkkauskieli on *HTML (HyperText Markup Language)*. XML:n oli tarkoitus poistaa SGML:n heikkoudet. XML:stä saadut keskeiset hyödyt ovat:

- informaation siirrettävyyden helppous,
- laitteisto- ja ohjelmistoriippumattomuus,
- laajentamisen helppous,
- yhteensopivuus SGML:n ja HTML:n kanssa,
- kieltä käsittelevien ohjelmien kirjoittamisen helppous,
- dokumenttien luettavuus ja
- dokumenttien tekemisen helppous. [3]

XML:n syntaksi on varsin yksinkertainen, joten se on helposti opittavissa ja sovellettavissa. On olemassa kuitenkin muutamia syntaksisääntöjä, joita tulee noudattaa XML-dokumentteja luotaessa. Alla on esimerkki XML-dokumentin rakenteesta.

```
<?xml version="1.0" encoding="utf-8"?>                    (1)
<!DOCTYPE album SYSTEM "music-album.dtd">                 (2)
<albums>                                                  (3)
    <album artist="First Artist" year="1985">             (4)
          <name>Dire Straits</name>                       (5)
          <tracks>7</tracks>                              (6)
          <!-- Kommentti tähän                            (7)
    </album>                                              (8)
</albums>                                                 (9)
```

XML-dokumentti on oltava joko oikein muodostettu tai oikeellinen eli validi, jotta se olisi käyttökelpoinen. Dokumentti on oikein muodostettu, jos se täyttää kaikki XML:n syntaksisäännöt. Tämä on käyttökelpoinen XML-dokumentti, mutta aina se ei kuitenkaan riitä. Ennen pitkää tulee tarkistaa tiedon järkevyys. Dokumentti on oikeellinen, kun se on sekä XML-syntaksin että *DTD:nsä (Document Type Definition)* mukainen. [3] Kun tieto on tarkoitus vaihtaa kahden eri sovelluksen välillä käyttäen XML:ää, onkin

järkevää luoda dokumentille tyyppimääritys. Näin molemmat osapuolet voivat varmistua dokumentin rakenteesta ja sen elementtien sisältämien tietojen tyypeistä.

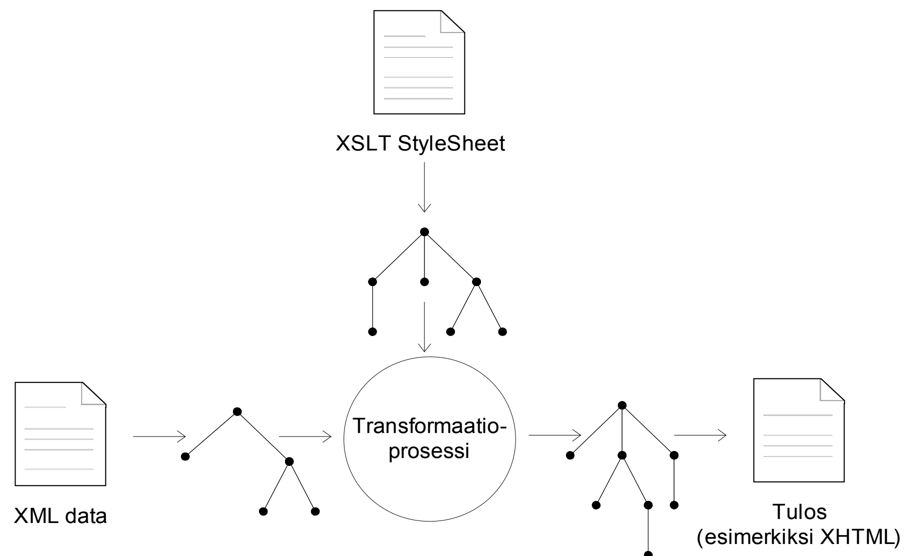
DTD (Document Type Definition) kuvaa XML-dokumentin sanaston ja kieliopin määrittelevät säännöt. DTD määrää elementit, joita voidaan käyttää XML-dokumentissa. Elementtien lisäksi se määrittelee elementteihin kuuluvat ominaisuudet ja elementtien väliset suhteet ja arvojen arvoalueet. DTD voidaan määritellä joko XML-dokumentin alkuun tai erilliseen tekstitiedostoon. Jälkimmäinen tapa on suositeltavaa sekä luettavuuden että selkeyden kannalta. [3]

XML-skeema kuvaa myöskin XML dokumentin rakenteen, elementit, attribuutit, attribuuttien tietotyypit ja niiden sallitut arvojoukot samoin kuin DTD. Erona DTD:hen on, että XML-skeema on itsessään XML dokumentti. XML-skeema on kuitenkin DTD:hen verrattuna monipuolisempi, sillä se esimerkiksi määrittelee perustyypit tarkemmin kuin DTD:ssä. Lisäksi se noudattaa samaa kielioppia kuin XML. Tämä ehkäisee mahdollisia virheitä, jotka voivat aiheutua eri kielioppien käytöstä. [16]

3.3.2 XSLT

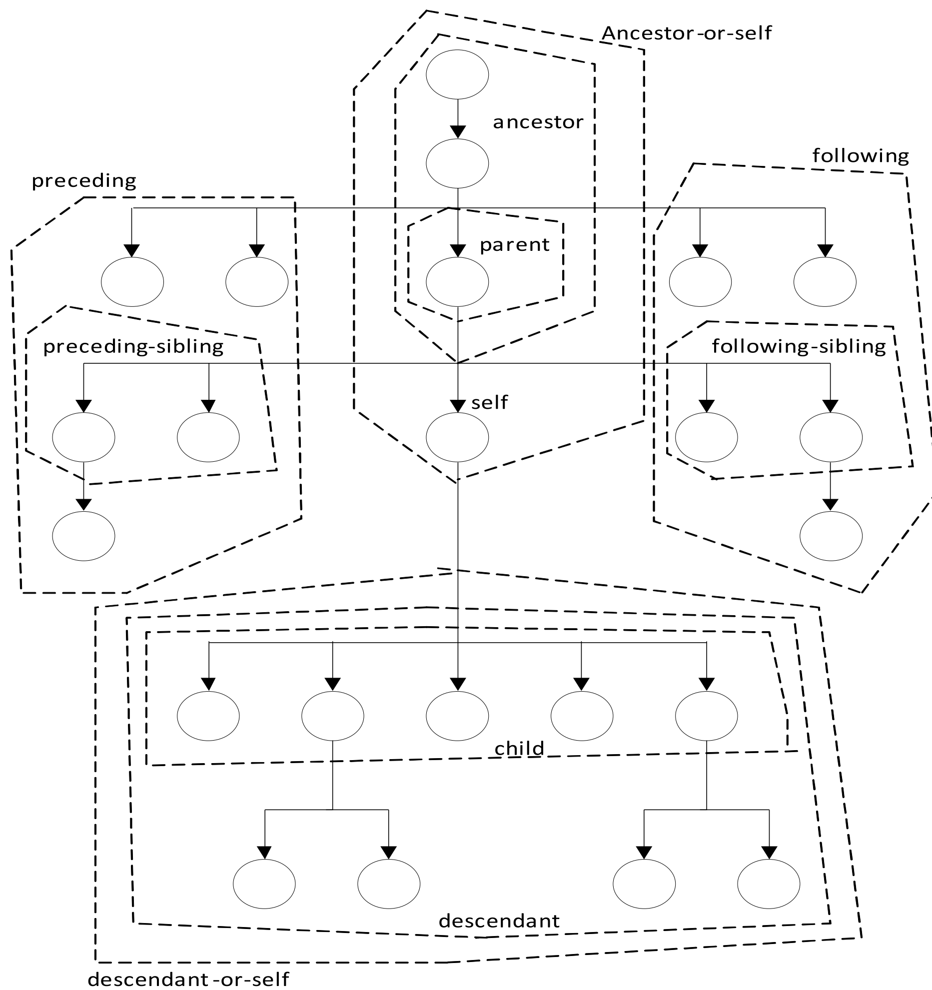
XML-dokumentit ovat tekstitiedostoja, joten ne eivät sellaisenaan sovellu datan esittämiseen. Kun halutaan esittää dataa eri muodossa, dataa tulee käsitellä. Eräs tapa XML-dokumenttien datan käsittelyyn on XSLT (XSL Transformations). XSLT on XSL (Extensible Stylesheet Language)-kieliperheeseen kuuluva sääntöpohjainen muunnoskieli. Nimestä huolimatta XSLT ei ole tyylikieli, koska se ei määrittele visuaalisia esitystapoja, kuten esimerkiksi *CSS (Cascading Style Sheets)*. XSLT:n 1.0 versio on ollut käytössä vuodesta 1999 lähtien ja saavuttanut laajan tuen eri ympäristöissä. Uudempi versio XSLT 2.0 tarjoaa paljon uusia ominaisuuksia, mutta niitä ei ole vielä kuitenkaan laajasti tuettu. Dokumentit tulee tulkita ennen kuin sen sisältämää tietoa voidaan käyttää. [15, s. 6–7]

XSLT 1.0 määrittelee XSLT-prosessin toiminnallisuuden, joka muuntaa lähdedokumentin uuteen muotoon. Dokumentit jäsennetään yleensä puumaiseksi rakenteeksi, josta käytetään nimitystä jäsennyspuu. Kuvassa 3.1 on esimerkki siitä, kuinka lähdedokumentti muutetaan uuteen muotoon.



Kuva 3.1. *XSLT -muunnosprosessi. [9]*

Käytännössä XSLT-prosessi ottaa muutettavan datan ja tehtävän muutoksen syötteinään. Muunnossääntöjen perusteella muodostetaan lähdedokumentista kohdedokumentti. [15, s. 4–6] Vaikka XSLT tarjoaa kielen muunnosten tekemiseen, niin se ei yksinään riitä. XSLT:n avulla ei voida viitata jäsennyspuun osiin. Viittausta tarvitaan, jotta dokumentin tietoja voitaisiin hakea. *XPath (XML Path Language)* on kyselykieli, jonka avulla voidaan hakea tietoja XML-dokumentista, toisin sanoen viitata jäsennyspuun osiin. Kuvasta 3.2 nähdään miten XPath-jäsennyspuu näkee muita osia suhteessa ”*Self”*-solmuun.



Kuva 3.2. *XPath-jäsennyspuun osat. [15, s. 4]*

XPath on mittausohjelmistossa tärkein muunnostekniikka XSLT:n ohella. Tätä tekniikkaa käytetään raporttikomponentin yhteydessä. Esimerkiksi mittausdata tuloksia haetaan XML-dokumentista juuri XPath-kyselyillä.

4 TAKAISINMALLINNUS

4.1 Yleistä

Takaisinmallinnuksesta eli käänteistekniikasta on monien muiden ohjelmistotekniikan käsitteiden tapaan useita määritelmiä. Yksi yleisemmin käytetyistä on E. Chikofskyn ja J. H. Crossin määrittely vuodelta 1990. [6, s. 3] Heidän mukaansa takaisinmallinnus on analysointiprosessi, jonka avulla pyritään saamaan käsitys ohjelmasta. Analysointiprosessin avulla pyritään tunnistamaan ohjelman eri osia ja näiden osien välisiä suhteita sekä esittämään ohjelma korkeammalla abstraktiotasolla. Takaisinmallinnuksen tarkoituksena on tutkia ja analysoida olemassa olevaa ohjelmaa. [8, s. 117]

Ohjelmiston evoluution aikana siihen kohdistuu useita ylläpidollisia toimenpiteitä, kuten uusien ominaisuuksien lisäämistä, vanhojen korjaamista ja parantamista sekä ohjelman laadun parantamista. Evoluutiomainen kasvu johtaa ohjelman rakenteen monimutkaistumiseen ja dokumentaation rappeutumiseen. Ajan myötä tarvittavan ja olemassaolevan tiedon välinen kuilu kasvaa. Mitä suuremmaksi tämä kuilu kasvaa, sen hankalampi ohjelmistokehittäjien on tehdä ohjelmaan laajoja rakenteellisia ja arkkitehtuurillisia uudistuksia. Puutteellisen tiedon johdosta ohjelmistokehittäjät eivät myöskään pysty arvioimaan tehtävien muutosten vaikutusta ohjelman eri osiin. Lähdekoodi on ainoa luotettava tiedonlähde evoluutiomaisen kehityksen jälkeen. Takaisinmallinnus on tekniikka, jonka avulla saadaan jäljitettyä kadonnutta tietoa ohjelmasta sen lähdekoodia tutkimalla. [10, s. 50]

Takaisinmallinnuksen hyötyjä kadonneiden tietojen jäljittämisen lisäksi ovat muun muassa monimutkaisuuden hallinta, vaihtoehtoisten näkymien tarjoaminen, sivuvaikutusten paljastaminen, korkeamman abstraktiotason esitysten tuottaminen ja uudelleenkäytön mahdollistaminen. Monimutkaisuuden hallintaan tarjotaan työkaluja, jotka helpottavat olennaisten tietojen suodattamisessa suurista ja monimutkaisista järjestelmistä. Vaihtoehtoiset näkymät ovat graafisia kuvauksia ohjelmasta, jotka edistävät sen ymmärtämistä. Erilaisia näkymiä ovat esimerkiksi tieto- ja kontrollivuokaavio sekä rakennekartta. Myöskin korkeamman abstraktiotason esitykset edistävät ohjelman ymmärtämistä, koska ne ovat lähempänä ihmisen ajattelumaailmaa. Ylläpidolliset toimenpiteet voivat aiheuttaa tahattomia sivuvaikutuksia, jolloin ohjelma ei toimi tarkoitetulla tavalla. Sivuvaikutusten paljastamisen ansiosta tällaiset virheelliset toiminnot huomataan jo aikaisessa vaiheessa. Useimmiten halutaan hyödyntää olemassa olevia ohjelmia, niiden

osia ja koodia. Takaisinmallinnus auttaa löytämään sellaiset osat, jotka soveltuvat uudelleenkäytettäviksi. [8, s. 118–120; 5, s. 328]

4.2 Suunnitteluratkaisun jäljittäminen ja uudelleendokumentointi

Takaisinmallinnuksesta voidaan erottaa kaksi osa-aluetta: suunnitteluratkaisun jäljittäminen ja uudelleendokumentointi. Suunnitteluratkaisun jäljittämisen tarkoituksena on tuottaa tietoa siitä, mitä ohjelma tekee, miten se sen tekee ja miksi se toimii juuri tietyllä tavalla. Suunnitteluratkaisujen ja -päätösten jäljittäminen tapahtuu tarkastelemalla ohjelmakoodia. Suunnitteluratkaisu voi liittyä siihen, miten järjestelmä on jaettu alijärjestelmiin ja miten moduulit on jaettu aliohjelmiin. Suunnitteluratkaisu voi lisäksi liittyä esimerkiksi esitysmuodon, tietorakenteiden ja muuttujien valintaan. Samalla, kun suunnitteluratkaisuja tunnistetaan, tapahtuu uudelleendokumentointi.

Uudelleendokumentoinnissa olemassaolevalle ohjelmalle tuotetaan uusia dokumentteja tai korjataan vanhoja. Kommenttien tuottaminen ohjelmakoodiin tai ohjelmakoodin muuttaminen pseudokoodiksi ovat esimerkkejä uudelleendokumentoinnista. Pseudokoodilla tarkoitetaan ohjelmakoodin esittämistä muodossa, joka on lähempänä ihmisen ymmärtämää kieltä. Yhtenä uudelleendokumentoinnin osa-alueena on rakenteellinen uudelleendokumentointi, joka tarkoittaa ohjelman arkkitehtuurin rakenteen uusimista. Toisena osa-alueena on suunnittelupäätösten tunnistaminen, jonka tarkoituksena on muuttaa ohjelmakoodi pseudokoodiksi. [8, s. 120–123] Uudelleendokumentoinnin tarve syntyy, kun ohjelman dokumentit ovat kadonneet, niitä ei ole olemassa tai ne eivät ole ajan tasalla. Nämä ovat tyypillisiä ongelmia ohjelmistotuotannossa. Eräs syy ongelmiin on, että dokumenttien luominen ja päivittäminen koetaan turhaksi ja resursseja kuluttavaksi toiminnaksi. [11, s. 38]

4.3 Staattinen ja dynaaminen takaisinmallinnus

Takaisinmallinnus voidaan jakaa kahteen osaan, staattiseen ja dynaamiseen. Ohjelmaa on syytä tarkastella molemmista näkökulmista, sillä molemmat tuovat tietoa ohjelman osista ja niiden välisistä suhteista. Staattisessa takaisinmallinnuksessa ohjelman tutkiminen ja analysointi tapahtuu tarkastelemalla pelkästään ohjelman lähdekoodia. Staattisen takaisinmallinnuksen avulla on tarkoitus kuvata ohjelman staattiset eli pysyvät osat, kuten esimerkiksi luokat ja niiden suhteet. Luokkien väliset suhteet ovat muun muassa osakokoonpano-, assosiaatio- ja periytymissuhde. Staattista tietoa voidaan esittää esimerkiksi luokkakaavioiden avulla. [8, s. 124–127]

Dynaamisessa takaisinmallinnuksessa kiinnitetään huomiota ohjelman ajonaikaiseen käyttäytymiseen. Toisin sanoen tarkastellaan ohjelman ajonaikaista toimintaa lähdekoodin ohella. Dynaaminen takaisinmallinnus on tärkeässä roolissa, kun tutkitaan olipohjaisia järjestelmiä. Oliopohjaisissa ohjelmissa on runsaasti dynaamisia piirteitä, kuten

ajonaikainen sitominen ja olioiden luominen. Näitä piirteitä ei pysty ymmärtämään pelkästään lähdekoodia tutkimalla. Dynaamista tietoa voidaan esittää esimerkiksi tapahtumasekvenssikaavioiden avulla. [8, s. 127–128; 11, s. 24–28]

4.4 Yleiskuvaus takaisinmallinnustyökaluista

Takaisinmallinnuksen tueksi on kehitetty erilaisia työkaluja. Ne keskittyvät eri sovellusalueisiin ja tukevat eri ohjelmointikieliä. Nykyisin työkalut ovat monipuolisia ja joustavia, ja niiden avulla saadaan suodatettua monipuolista tietoa analysoitavasta ohjelmasta. Työkaluihin liittyy kuitenkin rajoituksia. Yksi rajoittavin tekijä on ohjelmointikieli. Yhdellä työkalulla ei voida tutkia kaikkia mahdollisia ohjelmointikieliä. Myöskään kaikki työkalut eivät tue dynaamista takaisinmallinnusta, vaan ne keskittyvät yleensä staattiseen takaisinmallinnukseen.

Tässä työssä tarkasteltava mittausohjelmisto on kehitetty C#-ohjelmointikielellä, joka on rajoittavin tekijä valittaessa takaisinmallinnustyökalua. Toinen työkalun valintaan vaikuttava tekijä on lisensointi. Tässä työssä ei käytetä maksullisia työkaluja lukuun ottamatta niiden evaluointiversioita. Takaisinmallinnuksen työkaluja löytyy lukuisia: StarUML, Metamill, Enterprise Architect ja Altova Umodel ovat takaisinmallinnustyökaluja, jotka tukevat C# -ohjelmointikieltä ja ovat saatavissa joko ilmais- tai evaluointiversiona. Ongelmana on kuitenkin löytää työkalu, joka soveltuu parhaiten käyttötarkoitukseen. Seuraavaksi esitellään lyhyesti nämä potentiaaliset työkalut.

StarUML on avoimen lähdekoodin mallinnustyökalu. StarUML:n kehityksen tarkoituksena oli korvata maksullisia mallinnustyökaluja, kuten Rational Rose. StarUML tukee UML 2.0 standardia ja useita ohjelmointikieliä, kuten C++, Delphi, C# ja VB.Net. [19]

Metamill on mallinnustyökalu, joka pohjautuu UML 2.3 -standardiin. Metamillin avulla on mahdollista luoda useita erityyppisiä graafisia kuvauksia, kuten käyttötapaus-, luokka-, komponentti-, tila-, kommunikaatio- ja aktiviteettikaavioita. Metamill tukee sekä etenevää että takautuvaa mallinnusta. Toisin sanoen mallista voidaan generoida lähdekoodia ja lähdekoodista mallia. Metamill tukee seuraavia ohjelmointikieliä: C++, Java, C# ja VB.Net. Metamill on maksullinen työkalu, mutta siitä on saatavilla 30 päivän evaluointiversio. [14]

Sparx Systemsin Enterprise Architect on UML 2.4 standardiin perustuva mallinnustyökalu. Se tarjoaa mallinnuksen lisäksi tukea jäljitettävyys-, ylläpidettävyys- ja testaustoimenpiteisiin. UML-standardin lisäksi työkalu tukee BPMN-, SysML- ja monia muita spesifikaatioita. Enterprise Architect on monipuolinen ja laajasti tuettu mallinnustyökalu, jonka käytettävyyteen on kiinnitetty erityisen paljon huomioita. Enterprise Architect on maksullinen, mutta siitä on myöskin saatavilla 30 päivän evaluointiversio. [18]

Altova UModel on mallinnustyökalu, joka pohjautuu UML 2.0 standardiin. Työkalun avulla on mahdollista luoda useita erityyppisiä graafisia kuvauksia samoin kuin Meta-

mill työkalussa. Altova Umodel tukee Metamillin tapaan sekä etenevää että takautuvaa mallinnusta. Työkalu tukee seuraavia ohjelmointikieliä: C++, Java ja C#. Altova UModel on maksullinen työkalu, mutta siitä on myöskin saatavilla 30 päivän evaluointiversio. [1]

4.5 Takaisinmallinnustyökalun valinta

Tässä työssä sovelletaan Sparx Systemsin Enterprise Architect -mallinnustyökalua. Työkalun valinta perustui lähinnä kokeiluihin. Enterprise Architectin lisäksi kokeiltiin StarUML ja Metamill.

Ensin kokeiltiin StarUML-mallinnustyökalua, joka asentui ongelmitta. Työkalun käynnistys ei onnistunut johtuen puuttuvasta kirjastosta. Puuttuvaa kirjastoa ei haettu, koska vaihtoehtoina olivat kaksi muuta mallinnustyökalua. Seuraavaksi kokeiltiin Metamill-mallinnustyökalua. Metamillin käyttöliittymä oli varsin vanhanaikainen eikä se osannut muodostaa luokkien välisiä suhteita. Metamill-työkalun evaluointiversiossa on lisäksi rajoitteena, että se voi mallintaa maksimissaan kahtakymmentä luokkaa, mikä on varsin rajoittava tekijä suurissa ohjelmistoissa. Viimeisenä kokeiltiin Enterprise Architect -työkalua, joka toimi moitteettomasti. Kokeilujen perusteella kävi ilmi, että työkalu oli sopiva spesifikaatiokomponentin tapauksessa. Altova Umodel -työkalua ei yritetty asentaa, koska se vaati rekisteröintiä jopa evaluointiversiolle.

5 SPESIFIKAATIOKOMPONENTIN ARKKITEHTUURI

5.1 Työnkuvaus

Spesifikaatiokomponentin arkkitehtuuria lähdettiin mallintamaan takaisinmallinnustyökalun avulla. Takaisinmallinnustyökalun lisäksi tutkittiin ohjelmakoodia, jonka pohjalta täydennettiin arkkitehtuurikuvausta.

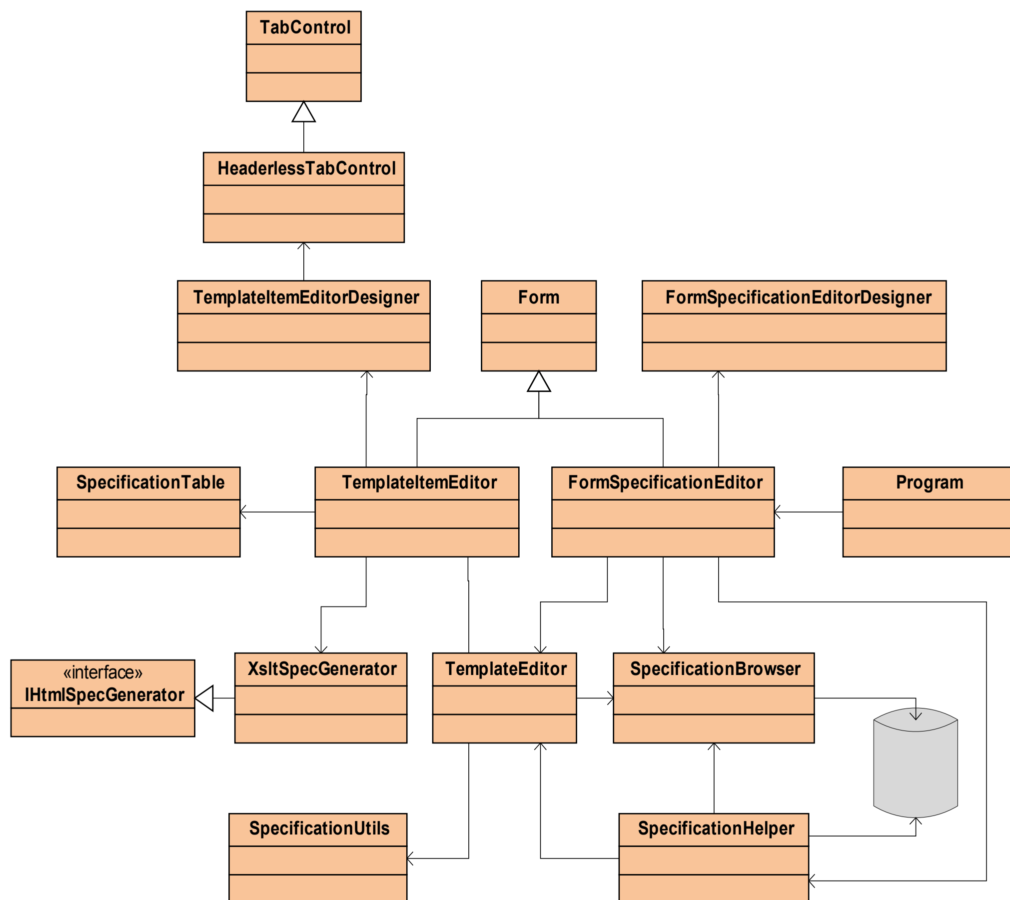
Tässä luvussa aluksi esitellään, miten takaisinmallintaminen soveltui työn tavoitteisiin ja mitkä oli työkalun keskeiset puutteet. Sitten kuvataan arkkitehtuurin eri osia tarkemmin ja esitellään arkkitehtuuriratkaisuja, joita havaittiin takaisinmallinnustyökalun ja lähdekoodi analysoinnin perusteella.

5.2 Takaisinmallintaminen

Tässä kohdassa keskitytään siihen, miten Enterprise Architect -mallinnustyökalun käyttö soveltui työn tavoitteisiin. Tässä luvussa esitetään lisäksi arkkitehtuurikuvaus, joka aikaansaatiin takaisinmallinnustyökalun avulla.

Kohdassa 4.5 mainittiin, että muissa takaisinmallinnustyökaluissa havaittiin puutteita, muttei Enterprise Architect -työkalukaan ole puutteeton. Esimerkiksi työkalu ei suoraan osannut muodostaa kaikkia luokkien välisiä suhteita, mutta suurimman osan se kuitenkin osasi. Tämän lisäksi periytymissuhde oli merkitty UML-notaatiosta poiketen luokan oikeaan yläkulmaan. Merkintätapa kuitenkin selvisi tutkimalla analysoitavan ohjelman lähdekoodia. Hyvänä puolena työkalussa on, että kooditiedostoja ei tarvitse valita yksi kerrallaan, vaan voidaan valita kokonainen kansio, jossa on useita tiedostoja. Työkalu käy automaattisesti kaikki tiedostot ja alikansiot läpi muodostaen luokkakaavion. Tiedostot voidaan kuitenkin valita myös yksitellen, mikäli ne eivät ole samassa kansiossa. Työkalu ei automaattisesti muodosta muita kaavioita kuin luokkakaavion, joten kaikki muu joudutaan tekemään manuaalisesti. Enterprise Architect tarjoaa monipuolisen mallintamisen, joten sillä voidaan tuottaa esimerkiksi tavallisia UML-kaavioita. Näin ollen työkalu soveltuu muuhunkin kuin pelkästään takaisinmallintamiseen.

Kuvassa 5.1 on esitetty spesifikaatiokomponentin luokkakaaviorakenne. Luokkakaaviosta on poistettu muuttujat ja funktiot luettavuuden ja havainnollisuuden parantamiseksi.



Kuva 5.1. *Spesifikaatiokomponentin rakenne luokkakaaviona.*

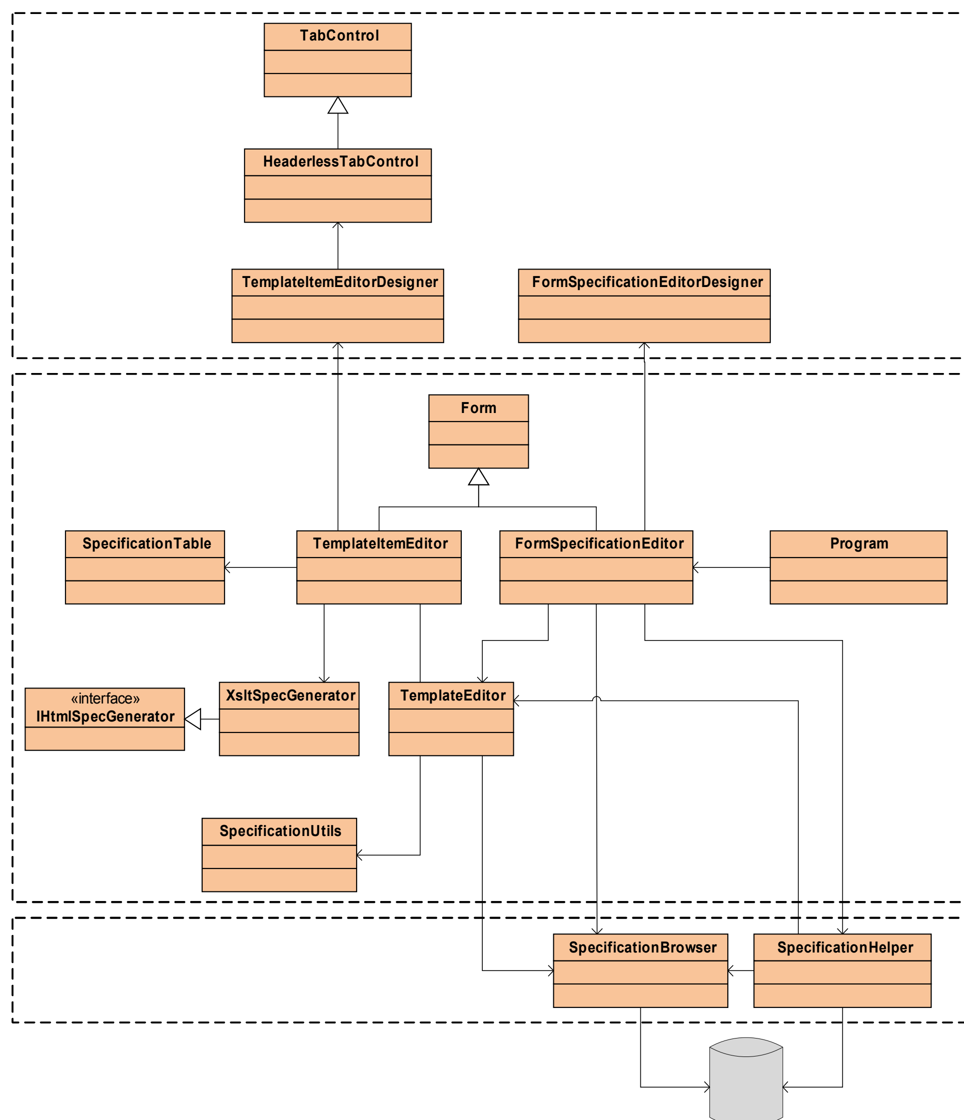
Takaisinmallinnustyökalun lisäksi sovellettiin staattista takaisinmallinnusta eli ohjelmakoodin tutkimista ja analysointia. Staattisen takaisinmallinnuksen havainnot käytettiin arkkitehtuurikuvauksen parantamiseen ja puuttuvien osien lisäämiseen. Työkalun tuottaman mallin keskeisin puute oli tietokantakommunikoinnin puuttuminen. Tämä saattoi johtua siitä, että mittausohjelmistossa tietokannan mallintamiseen käytettiin *Entity Framework* -oliomallinnusta. Entity Framework -kehyksen avulla järjestelmän tietokanta voidaan määritellä entiteettiluokkina. Entity Framework -kehyksen käyttäminen ei kuulu tämän diplomityön aiheeseen. Ohjelmakoodista selvisi nopeasti, mitkä komponentin osat käyttivät tietokantaa entiteettien kautta. Vaikkakin entiteetit tarjoavat mahdollisuuden hyödyntää tietokantaa missä tahansa ohjelmiston kohdassa, näin ei menetelty. Tietokantakommunikointi on jätetty kahden luokan vastuulle. Toinen luokista vastaa spesifikaation päivitykseen, poistoon ja tallentamiseen liittyvistä toimenpiteistä. Toinen taas vastaa spesifikaatiopohjan samoista toimenpiteistä. Luokkakaaviosta on selitetty tarkemmin kohdassa 5.3.

Yhteenvetona voidaan todeta, ettei takaisinmallinnustyökalu yksinään riitä ohjelmiston ymmärtämisessä ja takaisinmallintamisessa. Työkalujen ohella tarvitaan manuaalista staattista takaisinmallinnusta. Ohjelmakoodin tutkimista ja analysointia edesauttavat ohjelmiston järkevä rakenne ja siinä sovelletut koodauskäytännöt. Mittausohjelmiston ta-

pauksessa kaikki komponentit olivat omina projekteinaan. Näin ei tarvinnut perehtyä koko mittausohjelmiston koodiin vaan spesifikaatiokomponentin ohjelmakoodi oli eroteltu erillisen projektin alle. Tietokantakommunikointia ei takaisinmallinnustyökalun avulla ollut mahdollista havainnollistaa.

5.3 Arkkitehtuurin yleiskuvaus

Kuvassa 5.2 on esitetty paranneltu versio kuvasta 5.1. Kuvasta 5.2 nähdään, että luokat on jaettu kolmeen osaan. Tämä erottelu auttaa päättelemään, mitä arkkitehtuurityylejä spesifikaatiokomponentissa on käytetty.



Kuva 5.2. *Spesifikaatiokomponentin rakenne jaettuna osiin.*

Seuraavaksi esitellään kunkin luokan vastuu sekä luokkien väliset suhteet ja kommunikointitavat. `Program` on pääohjelma, joka käynnistää `FormSpecification-`

Editorin. FormSpecificationEditor on ohjain käyttäjälle näytettävästä käyttöliittymästä. Pääkäyttöliittymän toteuttaa FormSpecificationDesigner, sen vastuulla on käyttöliittymäkomponenttien asettelu käyttöliittymässä sekä tapahtumakäsittelyn asettaminen. Tapahtumankäsittely tapahtuu ohjainluokassa. TemplateItemEditor on toinen ohjainluokka, joka on toteutus TemplateItemEditorDesigner käyttöliittymälle. FormSpecificationEditorin avulla luodaan, muokataan ja poistetaan spesifikaatioita. TemplateItemEditorin avulla luodaan ja muokataan spesifikaatiopohjaan luotavat elementit, kuten luku-, teksti- ja tauluelementit. TemplateItemEditor toteuttaa TabControllista periytettyä luokkaa, jolla on tarkoitus piilottaa Tabin headerit. TemplateItemEditorin käyttämä SpecificationTable on toteutus taulun muodostamiselle spesifikaatiopohjaan. Taulun luonti on keskeinen osa, joten se on erotettu muusta toteutuksesta. Lisäksi se on monimutkaisempi kuin muut spesifikaatiopohjaan luotavat elementit. Näin monimutkaisemman elementin ylläpidettävyys paranee ja muutosten tekeminen on helpompaa.

XsltSpecGeneratorin avulla luodaan spesifikaation XML:stä HTML. Spesifikaatiokomponentti tarjoaa mahdollisuuden luoda spesifikaatiopohja visuaalisesti, joten tähän tarvitaan HTML-muotoa. TemplateEditor on pääkäyttöliittymän osa, jolla luodaan spesifikaatiopohjia. Tämä käyttää apunaan TemplateItemEditoria elementtien luonnin aikana. SpecificationUtils tarjoaa apufunktioita, joita käytetään XML:n sisentämiseen ja XML-dokumentin luomiseen. Luokat SpecificationHelper ja SpecificationBrowser huolehtivat tietokannan ja ohjelman välisestä kommunikoinnista. Vaikka spesifikaatiokomponentissa on käytetty Entity Framework -kehystä, vastuu tietojen päivittämisestä, lisäämisestä ja poistamisesta tietokannasta on jätetty näille kahdelle luokalle: SpecificationHelperin vastuulla on spesifikaatiopohjaan liittyvien muutosten päivittäminen tietokantaan ja SpecificationBrowserin vastuulla spesifikaation liittyvien muutosten päivittäminen tietokantaan.

5.4 Spesifikaatiokomponentin arkkitehtuuriratkaisut

Kuvan 5.2 ja ohjelmakoodi analysoinnin perusteella pystyy päättelemään spesifikaatiokomponentissa käytetyn arkkitehtuurityylin. Toisaalta aluksi oli hieman vaikeaa erottaa onko käytetty kerrosarkkitehtuuria vai MVC-arkkitehtuuria. Vaikeus johtui siitä, että molempien arkkitehtuurityylien ominaisuudet esiintyivät spesifikaatiokomponentissa. Molemmissa arkkitehtuurityyleissä on kolme osaa: käyttöliittymä, logiikka ja tietokanta.

Kerrosarkkitehtuurityylissä perusajatuksena on, että nämä kolme osat ovat organisoituneet kerroksiksi ja ylemmän tason kerros käyttää alemman tason kerroksien palvelui-

ta. Alemman tason kerroksien palveluita on tarkoitus käyttää kerroksittain, mutta joskus voidaan myöskin ohittaa jokin kerros. Tämä ei kuitenkaan ole suositeltavaa.

MVC eli näykymä-malli-ohjain -arkkitehtuurityylissä käyttöliittymä erotetaan sovelluslogiikasta. Tarkoituksena on jakaa vastuut. Mallin vastuulla on tarjota sovellukseen liittyvät loogiset toiminnot ja tiedot, rekisteröidä sovelluksen tilasta kiinnostuneet näkymäkomponentit ja ilmoittaa niille tilan muutoksista. Näkymän vastuulla on huolehtia sovelluksen tilan näyttämisestä käyttäjälle. Ohjaimen vastuulla on ottaa vastaan käyttäjän komentoja ja muuntaa ne sovelluksen toiminnoiksi. [4, s. 125-144]

Tarkalleen ottaen spesifikaatiokomponentin arkkitehtuuri noudattaa MVC-arkkitehtuurityyliä, koska siinä on selkeästi erotettu käyttöliittymä muusta sovelluslogiikasta, samaan tietoon on toteutettu kaksi näkymää ja luokkien vastuut vastaavat malli-näkymä-ohjain vastuita. MVC-arkkitehtuurityylin käyttämisestä seuraa muun muassa seuraavat edut:

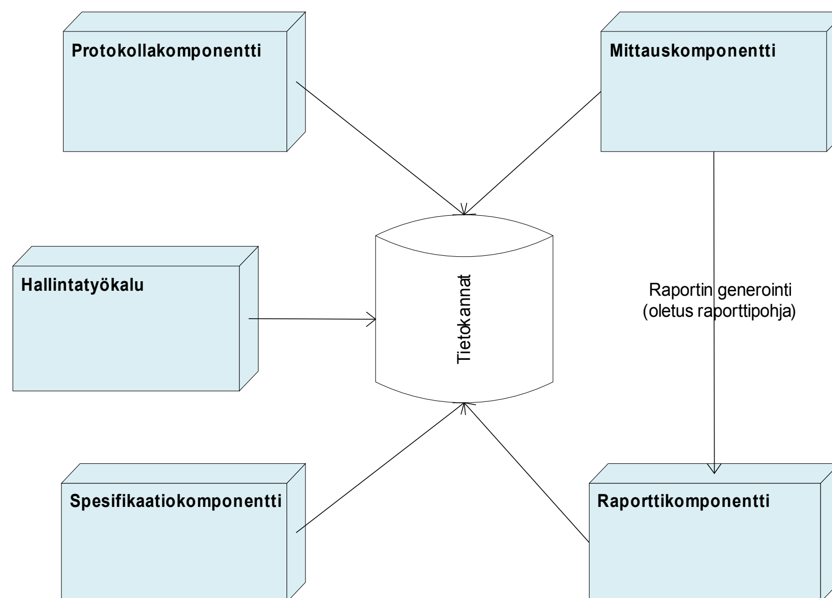
- käyttöliittymän ulkoasu on helposti vaihdettavissa,
- samaan tietoon voidaan toteuttaa useita näkymiä,
- rakenne on selkeä ja
- ohjelmistoon on helppo tehdä laajennuksia.

Ohjaimena toimivat spesifikaatiokomponentin tapauksessa `SpecificationHelper` ja `SpecificationBrowser`. Ohjaimen ansiosta tietokantahallintajärjestelmä on mahdollista vaihtaa ilman suurempia ongelmia. Tämä johtuu siitä, että muutokset, joiden avulla kommunikointi uuden tietokantahallintajärjestelmän kanssa saataisiin toimimaan, kohdistuisivat ainoastaan ohjaimeen.

Spesifikaatiokomponentissa on sovellettu MVC-arkkitehtuurityylin lisäksi Tila-, Tehdasmetodi- ja Tarkkailija-suunnittelumalleja [7, s. 107-116, s. 293-304, s. 305-314]. Tila-suunnittelumallia käytetään tilanteissa, joissa olion käyttäytyminen riippuu sen tilasta. Tässä yhteydessä olion käyttäytyminen on myöskin riippuvainen sen tilasta. Lisäksi spesifikaatiolla on myöskin oma tilansa, joka vaikuttaa sitä käsittelevän olion käyttäytymiseen. Tehdasmetodia käyttämällä luokka voi siirtää ilmentymien luonnin aliluokille. Tässä tapauksessa tätä on sovellettu siten, että käyttöliittymä luokalle annetaan vastuu luoda käyttöliittymä ja sen osat sekä määrittää osien ominaisuudet. Näin ohjainluokkaan ei tarvitse tehdä muutoksia, mikäli halutaan vaihtaa jonkin osan paikkaa tai ominaisuutta. Tarkkailija-suunnittelumalli on yleinen tapahtumapohjaisessa kommunikoinnissa, jota on käytetty spesifikaatiokomponentissa. Oliot voivat rekisteröityä tarkkailemaan tietyt tapahtumat. Kun lähde lähettää tapahtuman, siitä ilmoitetaan kiinnostuneille olioille, jotka tekevät tarvittavat toimenpiteet.

5.5 Komponenttien välinen kommunikointi

Mittausohjelmisto koostuu viidestä komponentista, jotka toimivat toisistaan riippumattomasti. Kukin komponentti käyttää muiden komponenttien aikaansaamia tuotoksia tai tulosteita tietokannan kautta. Tietokantoja on kaksi: toisessa on protokolliin ja mittauksiin liittyvät tiedot ja toisessa raporttiin ja spesifikaatioon liittyvät tiedot. Molempien tietokantojen välillä on riippuvuuksia, esimerkiksi protokollien yksityiskohtaiset tiedot ovat toisessa ja yleiset tiedot toisessa kannassa. Nämä on yhdistetty toisiinsa vierasavaimilla. Tietyissä tapauksissa komponentit käyttävät molempia tietokantoja, joten niiden käyttöä ei tässä yhteydessä erotella. Kuva 5.3 havainnollistaa komponenttien keskinäistä kommunikaatiota.



Kuva 5.3. *Mittausohjelmiston komponenttien välinen kommunikointi.*

Kuvasta 5.3 nähdään, etteivät komponentit suoraan riipu toisistaan. Mittauskomponentti käyttää mittaustulosten raportoimiseen raporttikomponenttia. Tämä luo paikallisen instanssin raporttikomponentista ja kutsuu raportin generointi funktiota. Raporttipohjana käytetään oletukseksi asetettua pohjaa. Raporttipohja asetetaan oletukseksi raporttikomponentin avulla. Spesifikaation valinta ei ole tässä yhteydessä mahdollista. Tämä on heikkous, koska verifiointiarvot ovat yhtä tärkeässä asemassa kuin mittaustulokset.

Mikäli raportingenerointifunktioon tehdään muutoksia, tämä ei vaadi suuria toimenpiteitä. Mittauskomponentin tulee ainoastaan huolehtia kutsua funktiota oikeilla parametreilla. Kääntäjä kuitenkin antaa virheilmoituksen, mikäli funktion parametrit ovat väärin. Näin ollen mittaus- ja raporttikomponentin välinen yhteys on hyvin löyhä.

6 ARKKITEHTUURIN ARVIOINTI

6.1 ATAM

SAAM (Software Architecture Analysis Method), ATAM (Architecture Tradeoff Analysis Method) ja *MPM (Maintenance Prediction Method)* ovat skenaariopohjaisia menetelmiä ohjelmistoarkkitehtuurin arvioimiseen. Tässä työssä käytetään ATAM-menetelmää, koska se soveltuu useiden eri laatuominaisuuksien arviointiin toisin kuin SAAM- ja MPM-menetelmää. SAAM on lähinnä tarkoitettu käytettäväksi muunneltavuuteen ja toiminnallisuuteen liittyvien laatuominaisuuksien arviointiin. MPM on kehitetty toisaalta ylläpidettävyyden arviointiin. [12, s. 228]

ATAM-menetelmä koostuu esittely-, analyysi-, testaus- ja raportointiosiosta. Arviointiprosessi kestää kolme päivää ja siihen osallistuvat arviointiryhmä, ulkoiset sidosryhmät, kuten asiakkaan edustajat ja sisäiset sidosryhmät. Sisäiset sidosryhmät ovat olleet mukana arkkitehtuurin suunnittelussa. Toisaalta arviointiryhmä muodostuu henkilöistä, jotka eivät ole olleet mukana ohjelmiston kehityksessä ja näin edustavat neutraalia näkemystä arkkitehtuurista. [12, s. 229]

Esittelyosiossa esitellään arviointiin osallistuville ATAM-menetelmä, liiketoiminnalliset tavoitteet, rajoitteet, järjestelmän vaatimukset sekä arvioitavaa arkkitehtuuria. Rajoitteita ovat esimerkiksi standardit, käytettävissä olevat resurssit, tuetut ohjelmistoalustat ja valmiiden komponenttien käyttö. Arkkitehtuuriin vaikuttaneiden keskeisten suunnitteluratkaisujen tuominen esille on olennainen osa esittelyosiota. [12, s. 229–230]

Analysointiosiossa tunnistetaan olennaiset arkkitehtuuriratkaisut, jotka toteuttavat laatuominaisuuksia. Laatuominaisuuksia täsmennetään laatupuun avulla. Laatupuu kuvaa ohjelmiston laatuominaisuuksia ja niihin liittyviä skenaarioita. Skenaario on esimerkki tilanteesta, jossa kyseessä oleva laatuominaisuus tulee esille. Jokaiseen skenaarioon liittyy kaksi parametria. Toinen parametrista kuvaa skenaarion tärkeyttä ja toinen toteutuksen vaikeutta ohjelmiston kannalta. Kun laatupuu skenaarioineen on olemassa, liitetään skenaariot ja arkkitehtuuriratkaisut toisiinsa. Tämän avulla voidaan tunnistaa riskit, turvalliset ratkaisut, tasapainokohdat ja herkkyyskohdat. Riski muodostuu arkkitehtuurillisesta ratkaisusta, joka mahdollisesti voi johtaa jonkin laatuominaisuuden huonontemiseen. Turvallinen ratkaisu on sellainen arkkitehtuurillinen päätös, joka parantaa laatuominaisuuksia. Turvallisesta ratkaisusta käytetään myöskin nimitystä ei-riski. Herkkyyskohta on arkkitehtuurillinen ratkaisu, joka on kriittinen jonkin laatuominaisuuden kannalta. Jos päätöksestä luovutaan, niin se voi johtaa laatuominaisuuden heikenty-

miseen. Tasapainokohta on ratkaisu, joka vaikuttaa useampaan kuin yhteen laatuominaisuuteen. [12, s. 230–232]

Testausosion tarkoituksena on täydentää analyysin tuloksia sidosryhmien näkemyksillä. Aluksi sidosryhmät tuottavat uudet skenaariot omien näkemystensä perusteella. Tämän jälkeen arviointiryhmä ja sidosryhmät yhdessä priorisoivat uudet skenaariot. Priorisoidut skenaariot liitetään laatupuuhun joko olemassa oleviin skenaarioihin tai luodaan uusi laatuominaisuushaara, johon skenaario liitetään. Skenaariot analysoidaan samoin kuin aikaisemmat skenaariot. Testausosiossa on pyrkimyksenä saavuttaa yhteinen näkemys siitä, mikä on tärkeää ohjelmiston kannalta ja mihin laatuominaisuuksiin tulee kiinnittää erityistä huomiota. [12, s. 232–233]

Raportointiosiossa esitetään arvioinnin tulokset kaikille arviointiin osallistujille. Tulokset, joita esitellään ovat olennaiset arkkitehtuurilliset ratkaisut, laatupuu, skenaariot ja niihin liittyvät riskit, turvalliset ratkaisut, herkkyys- ja tasapainokohdat. Tulokset esitetään lisäksi raporttina. [12, s. 233–234]

6.2 Arvioinnin tarkoitus spesifikaatiokomponentissa

Arvioinnilla pyritään spesifikaatiokomponentin tapauksessa siihen, mitkä ovat spesifikaatiokomponentin keskeiset ratkaisupäätökset ja miten ne vaikuttavat muutoksiin. Skenaariot pyritään valitsemaan siten, että ne tukevat spesifikaatiokomponentin ratkaisupäätösten esille tuomista. Siksi skenaarioiksi valitaan sekä vanhojen tietojen muuttamista että uusien ominaisuuksien lisäämistä. Tavoitteena on tuoda esille sellaiset ratkaisupäätökset, jotka ovat joidenkin ominaisuuksien toteutumisen kannalta kriittisiä, turvallisia ja riskejä aiheuttavia.

6.3 Spesifikaatiokomponentin skenaariot

Tässä kohdassa esitetään spesifikaatiokomponenttiin liittyvät skenaariot. Skenaariot koskevat uusien ominaisuuksien lisäystä ja suorituskykyä. Taulukkoon 6.1 on koottu käytettävät skenaariot ja laatuominaisuudet.

Taulukko 6.1. *Skenaariot ja niihin liittyvät laatuominaisuudet.*

Skenaario	Laatuominaisuus
Raportin luominen spesifikaatiosta (ID631)	Suoristuskyky
Virheellisen tiedon korjaaminen (ID632)	Muokattavuus
Uudentyyppisen ominaisuuselementin lisääminen (ID633)	Muokattavuus
Keskeytyneen spesifikaatiopohjan luonnin jatkaminen (ID634)	Suoristuskyky

Raportin luominen spesifikaatiosta. On saatu uusi versio spesifikaatiosta ja täytyy tuottaa sitä vastaava raportti. Spesifikaation vanhasta versiosta ei ole olemassa mitään tallenteita. Raportti pitää saada muodostettua uudesta spesifikaatiosta alle kolmessa tunnissa.

Virheellisen tiedon korjaaminen. Spesifikaation uudemmassa versiossa on muuttuneita kohtia. Muutokset on korostettu uudessa versiossa. Nämä muutokset pitää saada korjattua alle tunnissa.

Uudentyyppisen ominaisuuselementin lisääminen. Spesifikaatiopohjaan halutaan lisätä uusi kuvaajaelementti. Lisäyksen syynä on, ettei tuotteen ominaisuus ole mahdollista esittää olemassa olevilla elementeillä. Uuden kuvaajaelementin lisääminen pitää saada toteutettua viikossa.

Keskeytyneen spesifikaatiopohjan luonnin jatkaminen. Kahdessa viikossa pitäisi toteuttaa keskeytyneen spesifikaatiopohjan luonti. Keskeytys voi olla käyttäjästä johtumaton, kuten esimerkiksi sähkökatkos.

6.4 Arviointi

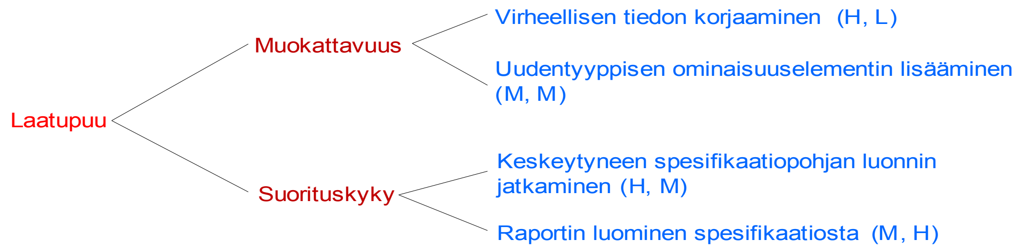
Spesifikaatiokomponentin arkkitehtuurin analysoinnin yhteydessä havaittiin seuraavat arkkitehtuuriratkaisut:

- MVC-arkkitehtuurityyli,
- Tila-suunnittelumalli,
- Tehdasmetodi-suunnittelumalli ja
- Tarkkailija-suunnittelumalli.

Näiden lisäksi spesifikaatiokomponentin arkkitehtuuriratkaisuihin voidaan lukea raporttien generoiminen XSLT:n ja XPathin avulla, ja spesifikaatiopohjan generoiminen visuaalisesti tai XML:n avulla.

Muokattavuus ja suorituskyky ovat laatuominaisuudet, joita käytetään skenaarioiden arvioinneissa. Muokattavuudella tarkoitetaan, miten arkkitehtuuri tukee uusien ominaisuuksien lisäystä. Tämän laatuominaisuuden avulla arvioidaan arkkitehtuurin soveltuvuutta tukea uudentyyppisiä spesifikaatioita ohjelmiston kehityksen jatkuessa. Suorituskyvyllä tarkoitetaan spesifikaatiopohjan luontiin käytettävää aikaa. Hyvä suoritusaika nopeuttaa spesifikaatiopohjien luontia.

Kuvassa 6.1 on laatupuu, josta nähdään laatuominaisuudet ja niihin liitetyt skenaariot. Skenaarioiden perässä oleva merkintä kertoo skenaarion priorisoinnista. Ensimmäinen arvo kertoo, kuinka tärkeä skenaario on ja toinen sitä, kuinka vaikea se on toteuttaa. Priorisointi merkintä on kolmiportainen asteikko, jossa High (H) edustaa korkeinta arvoa, Medium (M) keskimmäistä arvoa ja Low (L) matalinta arvoa.



Kuva 6.1. *Laatupuu skenaarioineen.*

Raportin luominen spesifikaatiosta. Skenaarion toteutuminen vaatii muutoksia tietomallissa ja ohjelmakoodissa. Tässä tilanteessa tulee toteuttaa raportin luominen suoraan spesifikaatiosta, jota ei ole tuettu tämänhetkisessä arkkitehtuurissa. Raporttien generoiminen XSLT:n ja XPathin avulla on herkkyyskohta tämän skenaarion kannalta. Raportti muodostuu XML:stä, joten spesifikaation XML:ää on mahdollista käyttää apuna raportin muodostamisessa.

Virheellisen tiedon korjaaminen. Skenaarion toteutumiseksi tarvitsee ainoastaan muokata spesifikaatiopohjan rakenne, joka voidaan tehdä joko visuaalisesti tai muokkaamalla XML-koodi. Nämä muutokset voi tehdä kuka tahansa, koska muutoksia ohjelmakoodiin ei tarvita. Mikäli spesifikaatiopohjaa on käytetty useissa spesifikaatioissa, muutokset heijastuvat automaattisesti kaikkiin spesifikaatioihin. Spesifikaatiopohjan generoiminen visuaalisesti tai XML:n avulla voidaan pitää herkkyyskohtana tämän skenaarion kannalta. Muutosten tekeminen ei onnistuisi halutussa ajassa, jos muutokset täytyisi tehdä ohjelmakoodiin.

Uudentyyppisen ominaisuuselementin lisääminen. Skenaarion toteutuminen vaatii muutoksia spesifikaatiokomponentin `TemplateItemEditoriin`, joka vastaa elementtien lisäämisestä. Tämän luokan tulee tukea uuden ominaisuuselementin lisäämistä, ja lisäksi tarvitaan toteutus ominaisuuselementin ominaisuuksien mukaisesti. Tämä voidaan toteuttaa omana luokkanaan, jos toteutus on monimutkainen, tai se voidaan integroida `TemplateItemEditoriin`. Ominaisuuselementin tiedot tulee lisätä myöskin skeemaan, jotta XML:n validointi onnistuisi uuden ominaisuuden osalta. MVC-arkkitehtuurityyli on turvallinen ratkaisu tämän skenaarion kannalta. Toisaalta spesifikaatiopohjan generoiminen visuaalisesti tai XML:n avulla on tämän skenaarion kannalta riski. Riski sen takia, että spesifikaatiopohja tulee validoida molemmissa tapauksissa. Visuaalisesti estetään elementin lisääminen tiettyihin paikkoihin ja XML:n tapauksessa tämä tulee tehdä skeeman avulla.

Keskeytyneen spesifikaatiopohjan luonnin jatkaminen. Skenaarion toteutuminen vaatii muutoksia spesifikaatiokomponentissa. Tällä hetkellä muutokset säilytetään niin kauan, kunnes käyttäjä tallentaa niitä. Mikäli käyttäjä haluaa tehdä toista toimintoa, niin näytetään varoitus tallentamattomasta datasta. Tallentamattoman datan tallentaminen pysyväismuistiin olisi järkevää sähkökatkosten ja muiden kaatumisten varalta. Tallentamaton data voidaan lukea pysyväismuistista ja antaa käyttäjälle mahdollisuus palata

kaatumista edeltäneeseen tilanteeseen. Spesifikaatiopohjasta tarvitsee ainoastaan tallentaa metatiedot ja spesifikaatiopohjan XML. Spesifikaatiopohjan muodostuminen XML:stä helpottaa tiedon palautumista keskeytyksen jälkeen. Spesifikaatiopohjan generoiminen XML:n avulla on turvallinen ratkaisu tämän skenaarion kannalta.

6.5 Arvioinnin tulokset

Taulukossa 6.2 on esitetty yhteenveto skenaarioissa esiintulleista arkkitehtuurillisista ratkaisusta ja niiden vaikutuksista.

Taulukko 6.2. *Arkkitehtuuriratkaisut ja niiden vaikutus.*

Arkkitehtuuriratkaisut	Vaikutus
Raporttien generoiminen XSLT:n ja XPathin avulla	Herkkyyskohta
Spesifikaatiopohjan generoiminen visuaalisesti tai XML:n avulla	Herkkyyskohta, turvallinen ratkaisu, riski
MVC ja ohjelmiston järkevä rakenne	Turvallinen ratkaisu

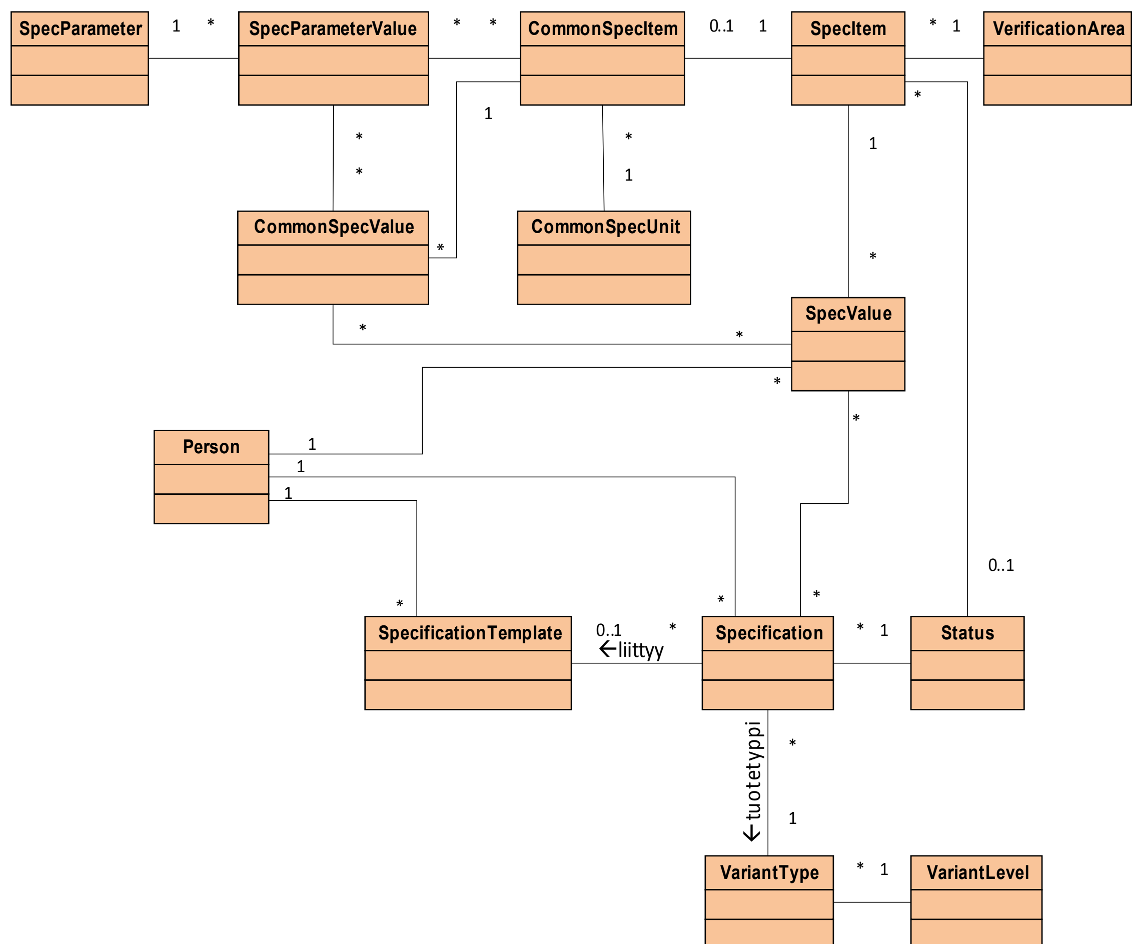
Taulukosta 6.2 nähdään, että spesifikaatiopohjan generoiminen visuaalisesti tai XML:n avulla on riski. Tämä kuitenkin on myös herkkyyskohta ja turvallinen ratkaisu muiden skenaarioiden osalta. Loppujen lopuksi siitä on kuitenkin enemmän hyötyä kuin haittaa.

Arkkitehtuurin kannalta ei tullut esiin sellaisia riskejä, jotka vaatisivat arkkitehtuurin uudistamista. Tämänhetkinen arkkitehtuuri on hyvä, mutta tietomalliin tarvitaan muutoksia, jotta raporttien muodostamista ja niihin käytettävää aikaa voitaisiin minimoida. Lisäksi tämä edistäisi uudelleenkäytettävyyttä. Arviointi soveltui hyvin tavoitteisiin, sillä saatiin esille spesifikaatiokomponentissa sovelletuista ratkaisupäätöksistä ja teknii-koista lyhyt yhteenveto. Yhteenvedon perusteella voidaan päätellä, mitkä ovat spesifikaatiokomponentin kriittisiä ratkaisupäätöksiä.

7 SPESIFIKAATIOKOMPONENTIN TIETOMALLI

7.1 Nykyinen tietomalli

Kuvassa 7.1 on esitetty spesifikaatiokomponentin tietomalli UML-kaaviona. Tietomallissa ei ole esitetty ominaisuuksia, jotta se pysyisi yksinkertaisena luettavuuden ja ymmärrettävyyden kannalta. Data tallennetaan relaationaalisen muodon lisäksi XML-muodossa



Kuva 7.1. Spesifikaatiokomponentin tietomalli.

Spesifikaatiopohja (SpecificationTemplate) on mahdollista luoda, kun on tiedossa verifioitavat ominaisuudet, vaikka mittauksia ei olisi vielä suoritettu. Spesifikaatiopohjaan liittyy rakenteen lisäksi metatietoja, joita ovat nimi, luontipäivämäärä, versio, kuvaus ja tekijä (Person). Kuten metatiedoista huomataan, tässä on käytetty sekä kuvailevaa että hallinnollista metatietoa. Spesifikaatiopohjan rakenne tallennetaan tietokantaan XML-muodossa, jonka rakenne kuvataan myöhemmin. Ohjelma myöskin käsittelee dataa XML-muodossa, joten se on sellaisenaan hyödynnettävissä ilman käsittelyyn liittyviä toimenpiteitä.

Tuotteen ominaisuuksia kuvataan taulukkomaisesti. Taulukon sarakkeet ovat kahdenlaisia: pelkkää tekstiä sisältäviä ja numeerisia, joihin lisätään verifiointiarvot. Sarakkeet, joihin lisätään verifiointiarvot ovat SpecItem-tyyppisiä. SpecItem sisältää VerificationArea:n, joka kertoo millaisilla mittalaitteilla mittaukset on suoritettu. VerificationArea toimii validoivana tietona, jotta mittausta suorittaessa samantyyppisillä mittalaitteilla arvot olisivat verrattavissa. SpecItem tarkentaa sarakkeen sisältöä erilaisilla parametriarvoilla (SpecParameterValue). Nämä arvot kertovat esimerkiksi millaisissa olosuhteissa verifiointiarvo pätee. SpecParameter kertoo tarkemmin SpecParameterValue:n tyypin. Tässä tulee hyvin esiin uudelleenkäytettävyys, koska parametrin tyyppi ja arvo on erotettu toisistaan omiin tauluihinsa. Näin samaa tyyppiä voidaan käyttää useamman verifiointiarvon yhteydessä. CommonSpecItem sisältää lisätietoa SpecItem:istä, kuten arvon tyypin ja yksikön (CommonSpecUnit). Sama yksikkö voi liittyä useaan arvoon, joten se on eriytetty omaan tauluun. SpecValue:en tiedot on myöskin jaettu kahteen tauluun. CommonSpecValue sisältää tarkempaa tietoa liittyen SpecValue-instanssiin.

Spesifikaatiopohjan ollessa valmis se voidaan liittää spesifikaatioon (Specification). Kaikki muutokset, jotka tehdään spesifikaatiopohjaan, heijastuvat automaattisesti spesifikaatioon, johon se on liitetty. Spesifikaatioon liittyy spesifikaatiopohjan lisäksi seuraavia metatietoja: nimi, versio, kuvaus, tekijä, luomispäivämäärä, status ja tuotetyyppi. Statuksen avulla havainnollistetaan missä tilassa dokumentti on. Dokumentin tila määrää dokumentin käsittelyoikeudet. Tiloja ovat *Work, Draft, R&D* ja *Official*. Esimerkiksi Work-tilassa oleva dokumentti on mahdollista muokata, muttei Draft-tilassa olevaa. Tuotetyyppi (VariantType) kuvaa tuotteita, joihin verifiointidokumentti on sovellettavissa. VariantLevel on korkeamman tason kuvaus tuotteen tyypistä.

Aiemmin mainittiin, että data on relaationaalisen muodon lisäksi XML-muodossa. Kuvassa 7.2 on esimerkki spesifikaatiopohjan rakenteesta XML-muodossa. Tämän lisäksi esitetään spesifikaatio, jossa kyseistä rakennetta on käytetty.

```
<specification>
  <title>New specification template</title>
  <section title="Johdanto" number="1">
    <p>Mittaukset tulee olla tehty...</p>
  </section>
  <section title="Verifioitavat ominaisuudet" number="2">
    <subsection title="Jälkikaiunta" number="2.1">
      <p>Alla oleva taulukko kuvaa äänen jälkikaiuntaa.</p>
      <table>
        <caption>Jälkikaiunta-aikoja</caption>
        <tr>
          <th>Parameter</th>
          <th>Min</th>
          <th>Nominal</th>
          <th>Definition and setup</th>
        </tr>
        <tr>
          <td>A = 30 m2</td>
          <td>
            <specitem
                guid="9743b425-b411-457a-94d4-be46061c308b"
                name="Aikaero"
                verificationareaguid="aa4c82c1-6c12">
             <specParameterValue
                guid="9e33d1c6-fdcd-4a43-affc-f794eaee1ca5"
                type="Area" value="30" />
             <specParameterValue
                 guid="680e0043-bb3f-4b47-90cf-2b26d922cd3d"
                 type="ValueType" value="Min" />
             <value datatype="Num" />
            </specitem>
          </td>
          <td>
            <specitem
                guid="9743b425-b411-457a-94d4-be46061c308b"
                name="Aikaero" verificationareaguid="aa4c82c1-6c12">
             <specParameterValue
                 guid="9e33d1c6-fdcd-4a43-affc-f794eaee1ca5"
                 type="Area" value="30" />
             <specParameterValue
                 guid="33abb8fa-628c-47d9-8d05-08f99cfed255"
                 type="ValueType" value="Nominal" />
             <value datatype="Num" />
            </specitem>
          </td>
          <td>Jälkikaiunta arvioitava kaavalla:<br />T=(0.161 s/m)*V/A </td>
        </tr>
      </table>
    </subsection>
  </section>
</specification>
```

***Kuva** 7.2. Spesifikaatiopohjan rakenne XML-muodossa.*

Kuvasta 7.2 näkyy, että XML:n rakenne on yksinkertainen. `Specification` on juurielementti, jonka lapsielementteinä voivat olla spesifikaation nimen lisäksi useita ensimmäisen tason otsikoita (`section`). XML rakenteessa on ensimmäisen tason otsikon lisäksi toisen (`subsection`) ja kolmannen (`subsubsection`) tason otsikoita. Nämä voivat esiintyä XML-rakenteessa tietyssä järjestyksessä, kuten missä tahansa dokumentissa. Toisin sanoen kolmannen tason otsikon tulee olla toisen tason otsikon lapsielementti, ja toisen tason otsikon tulee olla ensimmäisen tason otsikon lapsielementti. Kaikilla otsikkotyypeillä on kaksi attribuuttia, `title` ja `number`. `Title`-attribuutti kertoo otsikon nimen ja `number`-attribuutti numeroinnin. Kukin otsikko voi sisältää alemman tason otsikon lisäksi taulukoita (`table`) ja tekstikappaleita (`p`). Taulukon sarakkeet, joihin verifiointiarvot tulevat, sisältävät `specitem`-lapsielementin. Tämä elementti kertoo, minkä tyyppinen arvo sarakkeeseen hyväksytään ja minkälaisia esiehtoja siihen liittyy. Tällä hetkellä sarakkeen arvon tyyppi voi olla numeerinen, merkkijono tai totuusarvo. Kuvan esimerkissä taulukon sarakkeet hyväksyvät ainoastaan numeerisen arvon (`value`) ja molempien esiehtona (`specParameterValue`) on, että huoneen pinta-ala on 30 m^2. Tämän lisäksi molemmilla sarakkeilla on toisistaan eroava esiehto. Toinen hyväksyy tyyppinään minimiarvon ja toinen nimellisarvon.

XML:n oikeellisuudesta huolehditaan ohjelmallisesti validoimalla se spesifikaation skeemaa vasten. Spesifikaation skeema on XML-skeema, joka auttaa XML:n syntaksin ymmärtämisessä, koska siitä näkee nopeasti elementtien ominaisuudet ja niiden väliset suhteet ilman ohjelmakoodiin tutustumista tai ohjelman käyttöä. Näin käyttäjä voi muodostaa spesifikaatiopohjan XML-rakenteen jollain muulla itselleen sopivalla editorilla ja kopioida sitten rakenteen spesifikaatiopohjaan.

XML-rakenne on hyvä esimerkki metatiedon käytöstä. Taulukon sarakkeisiin määritetyt metatiedot eli `specItem`- ja `specParameterValue`-elementit auttavat ymmärtämään kunkin taulun sarakkeen tarkoituksen. Mikäli taulun sarakkeet olisi määritelty ilman metatietoja, sarakkeen rajoituksia olisi erittäin haastavaa saada esitettyä järkevällä tavalla. Lisäksi rajoituksia olisi saattanut puuttua kokonaan, jolloin ei oltaisi voitu hahmottaa miksi sarakkeen arvo on sellainen kuin se on. Pelkkien rivi- ja sarakeotsikoiden perusteella hahmotus olisi ollut myöskin haastavaa.

Kuvassa 7.3 on spesifikaatio, joka käyttää kuvan 7.2 rakennetta pohjanaan. Spesifikaatio sisältää rakenteen lisäksi verifiointiarvot, joita hyödynnetään raportoinnin yhteydessä.

1. Johdanto

Mittaukset tulee olla tehty...

2. Verifioitavat ominaisuudet

2.1. Jälkikaiunta

Alla oleva taulukko kuvaa äänen jälkikaiuntaa.

Jälkikaiunta-aikoja

Parameter	Min	Nominal	Definition and setup
A = 30 m2	2	3	Jälkikaiunta arvioitava kaavalla: T=(0.161 s/m)*V/A

***Kuva** 7.3. Spesifikaatio verifiointiarvoineen.*

Spesifikaation luomisen jälkeen tarvitsee vain täydentää verifiointiarvoja taulukon sarakkeisiin. Taulukon sarakkeisiin ei ole mahdollista syöttää väärän tyyppisiä verifiointiarvoja johtuen sarakkeen tyyppimäärityksestä.

7.2 Tietomallin arviointi

Spesifikaatiokomponentin kehittäminen jatkuu edelleen, jolloin vaatimusten muuttuminen on todennäköistä. Spesifikaatiokomponentin tietomallia voidaan helposti muuttaa vastaamaan muuttuneita vaatimuksia johtuen sen joustavasta rakenteesta. Spesifikaatiokomponentin tietomallin rakenne on myöskin suunniteltu tukemaan uusia ominaisuuksia. Esimerkiksi sillä voidaan kuvata muutakin kuin verifiointidokumentteja. Tietomalli on helposti laajennettavissa, jolloin tällaisia ominaisuuksia voidaan toteuttaa varsin joustavasti.

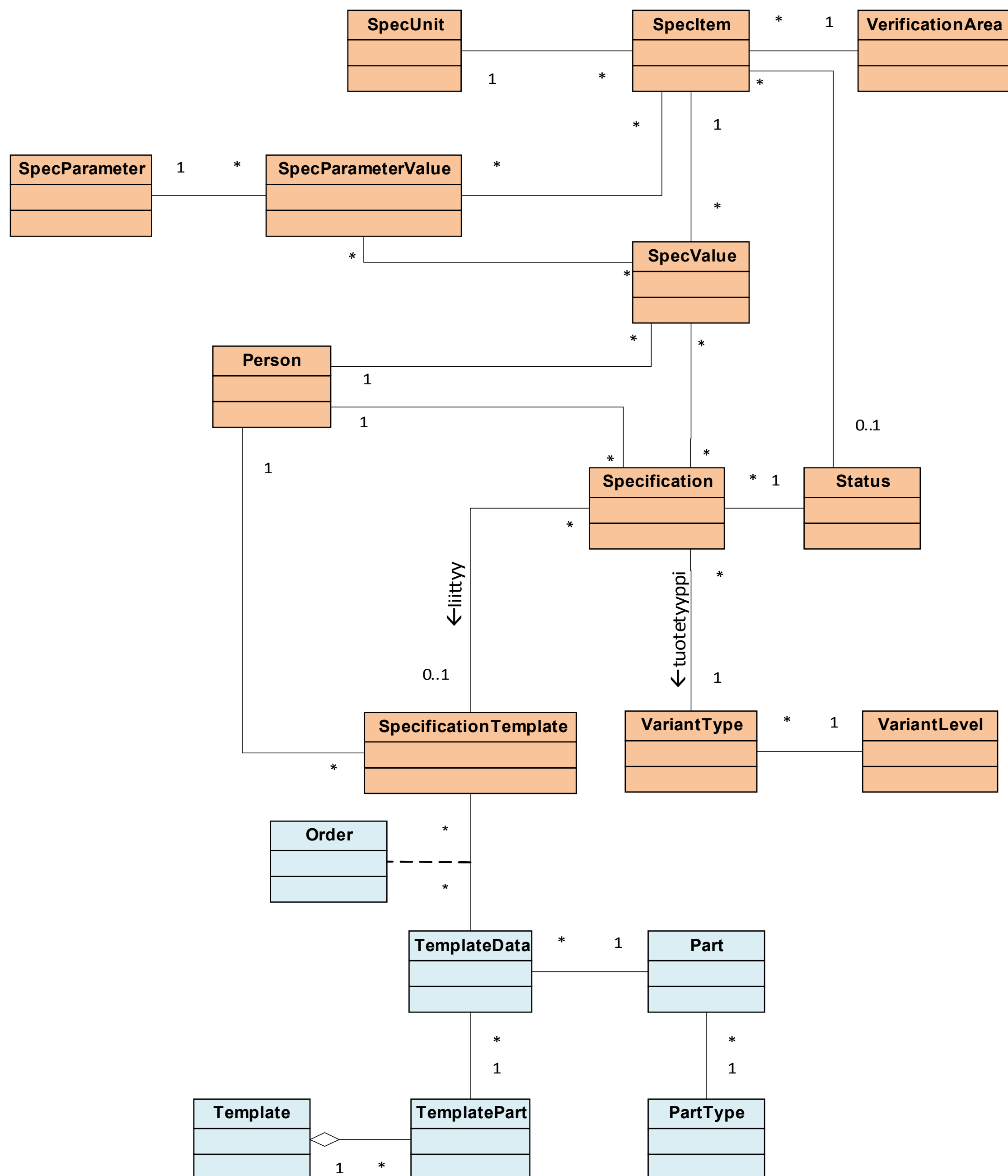
Tietomallista löytyy kuitenkin muutamia puutteita. `SpecItemiin` ja `Specificationiin` liittyvä `Status` on toteutettu vierasavaintyyppisenä. Viittauksena on kuitenkin käytetty `Status`-nimeä eikä numeerista arvoa, esimerkiksi GUID-arvoa, jota on käytetty muissa vierasavaimissa. Tässä luokkien välisen riippuvuuden luominen tapahtuu merkkijonon avulla eikä numeerisen arvon avulla. Tästä seuraa, että mikäli `Status` -nimeä joudutaan myöhemmin muuttamaan tai päivittämään, muutos on tehtävä erikseen jokaiseen luotuun instanssiin, joissa on käytetty `Status`-nimeä viittauksena. Numeerisen viittausarvon käyttö ei vaadi kyseisiä päivityksiä muualle kuin `Status`-taulun instanssiin. Toinen puutteellisuus liittyy `SpecItemiin` ja `CommonSpecItemiin`. Tietojen hajauttaminen kahteen luokkaan ei tässä yhteydessä tuo mitään lisäarvoa. Näin ollen tiedot olisi parempi yhdistää yhteen luokkaan, jolloin on helpompi ja nopeampi tehdä päivityksiä ja muutoksia kyseisen luokan tietoihin.

Tietomallista löytyy tällä hetkellä luokkia, joita ei ole hyödynnetty mitenkään. Tällaiset luokat, joita ei ole tarkoitus käyttää jatkossa, olisi hyvä poistaa kokonaan. Tietomallin ymmärrettävyyttä ja luettavuutta olisi näin ollen mahdollista hieman parantaa.

XML:n ansiosta spesifikaatiopohjan kopiointi on yksinkertaista, koska käyttäjä voi kopioida vanhan spesifikaatiopohjan XML:n uuden spesifikaatiopohjan XML:n tilalle, tehdä tarvittavat muutokset ja tallentaa uuden pohjan. Kopiointi on tarpeen, jos esimerkiksi tuotteesta tehdään uusi versio, joka sisältää vain muutamia uusia ominaisuuksia. Tällöin uudelle versiolle voidaan muodostaa spesifikaatiopohja vanhan version spesifikaatiopohjasta. Näin pienellä vaivalla saadaan aikaiseksi uusi spesifikaatiopohja, jota voidaan muokata vastaamaan uutta versiota. Tässä tavassa on edun lisäksi myöskin haittapuolensa. Eräänä haittana on virheen kopioituminen. Mikäli kopioitavassa datassa on virhe, se kopioituu muihinkin spesifikaatiopohjiin. Näin ollen virhe joudutaan korjaamaan kaikkiin spesifikaatiopohjiin, joihin virheellinen data on tullut kopioitua. Tämä on varsin työläs ja hankala prosessi, sillä käyttäjän tulee olla tietoinen siitä, mihin kaikkiin spesifikaatiopohjiin virheellinen data on kopioitu. Toisena haittana on, että rakenteen osat eivät ole uudelleenkäytettäviä.

Kopioinnista johtuvat ongelmat voitaisiin ratkaista siten, että spesifikaatiopohja koostettaisiin useista pienemmistä osista samoin kuin raporttipohja. Toisin sanoen spesifikaatiopohjan rakenne muodostuisi yksittäisistä luvuista, jotka on tallennettu omina instansseinaan. Näistä yksittäisistä instansseista muodostettaisiin spesifikaatiopohjan rakenne. Näin muutostenhallinta helpottuu ja nopeutuu. Muutostenhallinnan lisäksi päällekkäisen tiedon määrä vähenee, kun samaa data voidaan käyttää useassa eri spesifikaatiopohjassa.

Kuvassa 7.4 on esitetty paranneltu versio tietomallista, jossa on huomioitu edellä kuvatut puutteellisuudet ja parannusvaihtoehdot. Uudesta tietomallista on poistettu `CommonSpecItem`- ja `CommonSpecValue`-luokat ja niiden tiedot on yhdistetty `SpecItem`- ja `SpecValue`-luokkiin. `CommonSpecUnit` on nimetty `SpecUnit`:ksi selkeyden vuoksi. Tämän lisäksi spesifikaatiopohjan rakenne on suunniteltu muodostuvan pienemmistä osista, jotka on korostettu tietomalliin erivärisyydellä.



***Kuva 7.4.** Spesifikaatiokomponentin paranneltu tietomalli.*

Uuden tietomallin mukaan spesifikaatiopohja muodostuu useista `TemplateData`-instansseista. Nämä instanssit sisältävät spesifikaatiopohjan dataa eli ensimmäisen, toisen, kolmannen tason otsikoita, tekstikappaleita ja taulukoita. Kukin instanssi esiintyy tietyssä järjestyksessä spesifikaatiopohjassa. Järjestykseen liittyvät tiedot esitetään assosiaatioluokan (`Order`) avulla. `Part` kertoo lisätietoa liittyen dataan. Tässä tapauksessa se kertoo, onko kyseessä otsikko vai sisältö. Mikäli kyseessä on otsikko, niin `PartType` kertoo, minkä tyyppisestä otsikosta on kyse eli onko kyseessä ensimmäisen, toisen vai kolmannen tason otsikko.

Tuotteen ominaisuutta voidaan tutkia eri tavoin. Otetaan esimerkiksi jälkikaiuntaa, joka voidaan mitata tyhjässä huoneessa tai huoneessa, jossa on huonekalut. Näin meillä olisi kaksi eri variaatiota samasta ominaisuudesta, joiden verifiointiarvot tulee sisällyttää spesifikaatioon. Tämän takia tietomallissa esiintyy `Template`, joka koostuu useasta eri osasta (`TemplatePart`). Tämä kuvaa sitä, että `Template` on korkeamman tason nimike ominaisuudelle, jota verifioidaan ja `TemplatePart` sisältää sen eri variaatiot. Esimerkin tapauksessa `Template` instanssin nimi olisi ”Jälkikaiunta” ja sen `TemplatePart` instanssien nimet ”Tyhjä huone” ja ”Huone huonekaluineen”.

8 JATKOKEHITYSIDEAT

8.1 Tämänhetkinen raportin muodostamisprosessi

Tämänhetkisessä ohjelmiston versiossa raporttikomponentti muodostaa raportin, johon spesifikaation verifiointiarvot koostetaan lukemalla ne spesifikaation XML:stä. Raportti vastaa spesifikaation rakennetta, johon on lisätty mittausarvot ja havainnollisuuden vuoksi graafeja mittausarvoista.

Kuvassa 8.1 on raportti, joka sisältää aiemmin esitellyn jälkikaiunta esimerkin mittaustulosta. Kuten kuvasta 8.1 nähdään, se on rakenteeltaan täsmälleen samanlainen kuin spesifikaatio (kuva 7.3).

1. Johdanto

Mittaukset tulee olla tehty...

2. Verifioitavat ominaisuudet

2.1. Jälkikaiunta

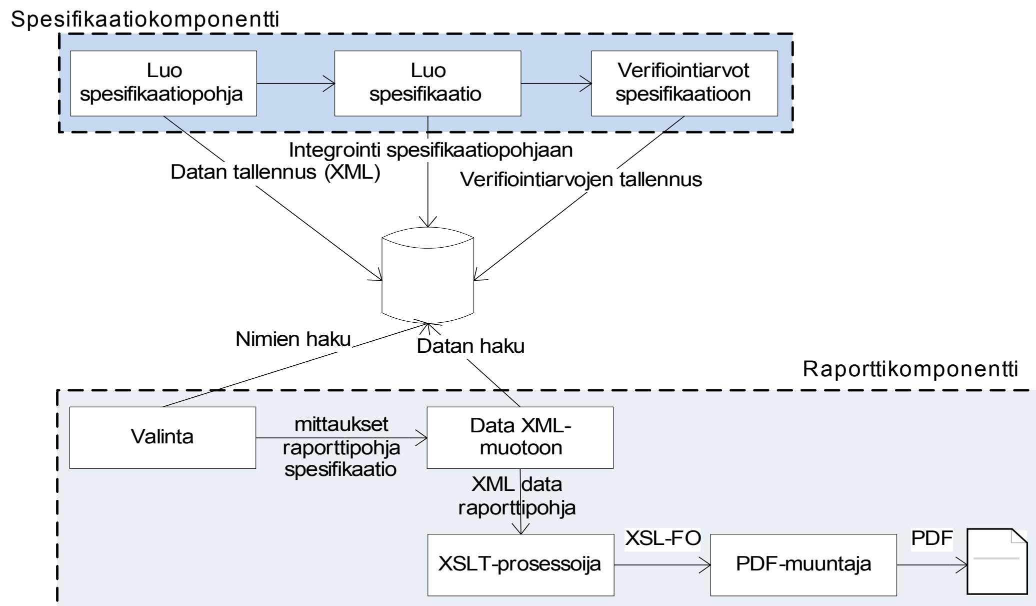
Alla oleva taulukko kuvaa äänen jälkikaiuntaa.

Jälkikaiunta-aikoja

Parameter	Spec value		Measurement value	Definition and setup
	Min	Nominal		
A = 30 m2	2	3	2.4	Jälkikaiunta arvioitava kaavalla: T=(0.161 s/m)*V/A

Kuva 8.1. *Raportti jälkikaiunta-ajan mittauksesta.*

Raportin ja spesifikaation välillä on kuitenkin pieni ero, joka näkyy taulukosta. Taulukko sisältää sekä spesifikaation verifiointiarvoja että mittaustulosta. Verifiointiarvot on luettu spesifikaatiosta ja mittaustulos suoritetuista mittauksista. Tässä tapauksessa käyttäjä joutuu tekemään kaksinkertaisen työn siten, että hän muodostaa ensin spesifikaatiopohjan, jonka jälkeen hänen tulee muodostaa raporttipohja. Raporttipohjassa luetaan muodostetusta spesifikaatiosta verifiointiarvoja XSLT- ja XPath-kyselyillä. Tämänhetkistä raportin muodostamisprosessia on havainnollistettu kuvaan 8.2.



Kuva 8.2. *Tämänhetkinen raporttikomponentin ja spesifikaatiokomponentin välinen kommunikointi raportin muodostamisessa.*

Kuvassa 8.2 näkyy, että raportin muodostaminen sisältää kaksi prosessia. Toinen on osa spesifikaatiokomponenttia ja toinen osa raporttikomponenttia. Prosessit ovat löyhästi riippuvaisia toisistaan tietokannan kautta. Ylempi prosessi tulee suorittaa kokonaisuudessaan ensin, jos verifiointiarvoja halutaan tuoda näkyviin raporttiin. Alemman prosessin suoritus vaatii, että mittaukset on suoritettu ja tarvittava raporttipohja on luotu sekä tarvittaessa spesifikaatio on luotu.

Ylempi prosessi voidaan suorittaa, kun on tiedossa uuden spesifikaation rakenne ja tiedot, joita tulee mahdollisesti tukea seuraavassa ohjelmiston julkaisuversiossa. Luontiprosessi on yksinkertainen. Ensimmäinen prosessi kuvaa, että saadun spesifikaation pohjalta muodostetaan spesifikaation rakenne (`Luo spesifikaatiopohja`), joka tallennetaan XML-muodossa tietokantaan. Sitten voidaan luoda spesifikaatio (`Luo spesifikaatio`), jolle määritetään metatietoja. Tässä luodaan spesifikaatiopohjan ja spesifikaation välille yhteys. Viimeisenä täytetään verifiointiarvot (`Verifiointiarvot spesifikaatioon`). Kuten kuvasta näkyy, jokainen prosessi on kiinteästi yhteydessä tietokantaan.

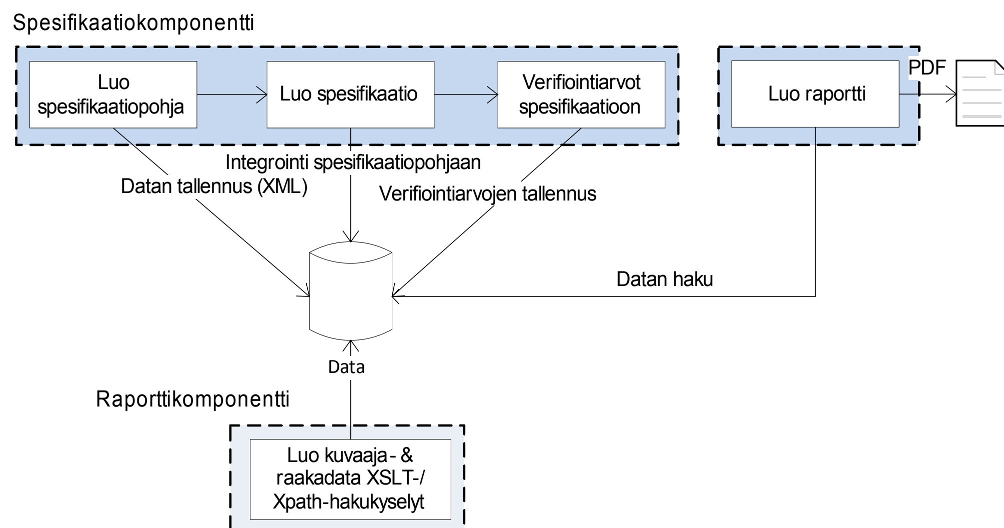
Alemman prosessin suoritus käynnistyy, kun halutaan luoda raportti. Raportin luonnin ensimmäisessä vaiheessa valitaan käytettävä raporttipohja, mittaukset ja spesifikaatio (`Valinta`). Tämän jälkeen ohjelmallisesti suoritetaan seuraavat prosessit, joista ensimmäisenä valitut mittaukset ja spesifikaatio muutetaan XML-muotoon (`Data XML-muotoon`). Periaatteessa ainoastaan mittausdata muutetaan XML-muotoon, koska spesifikaatio on jo valmiiksi XML-muodossa. Mittausdatan SQL-hakukysely on toteutettu siten, että se palauttaa datan XML-muodossa, joten tähän ei tarvita ohjelmallista toteutusta. Toisena raporttipohjaan luetaan mittaus- ja verifiointiarvoja määritettyjen XPath-

ja XSLT-muunnossääntöjen avulla sisääntulona saadusta XML-datasta (`XSLT-prosessoija`). Raporttipohja muutetaan tämän jälkeen XSL-FO-muotoon, josta generoidaan PDF-dokumentti PDF-muuntajan avulla (`PDF-muuntaja`).

Työn tehostamiseksi on suunniteltu muutamia ratkaisuvaihtoehtoja. Näiden ratkaisujen avulla käyttäjän ei tarvitse erikseen määritellä raporttipohjaa, joka vastaisi täsmälleen spesifikaation rakennetta. Tässä yhteydessä esitetään nämä ratkaisuvaihtoehdot sekä niiden hyvät ja huonot puolet. Lopuksi esitetään, mikä ratkaisuvaihtoehdoista olisi järkevin toteuttaa. Ratkaisuvaihtoehdot on jaettu sen mukaan, mitä tietomallia käytetään, eli onko käytössä tämänhetkinen tietomalli vai tehdäänkö tietomalliin kuvaa 7.4 vastaavat muutokset.

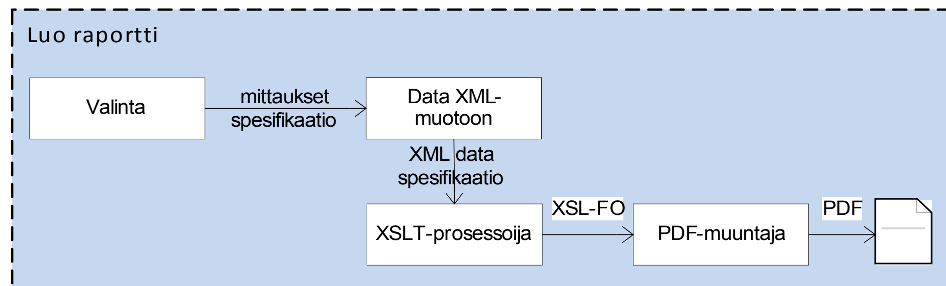
8.2 Raportin muodostaminen spesifikaatiosta

Oletetaan ensin, että tietomalliin ei lisätä uusia luokkia tai sen rakennetta ei muuteta kovin paljon. Ensimmäinen ratkaisuvaihtoehto on, että raportti muodostetaan spesifikaation perusteella. Kuvassa 8.3 on esitetty hahmotelma ensimmäisestä ratkaisuvaihtoehdosta.



Kuva 8.3. *Hahmotelma ratkaisuvaihtoehdosta yksi: raportin muodostaminen spesifikaatiokomponentin avulla.*

Kuten kuvasta 8.3 nähdään, spesifikaatiokomponentin alkuosa on samanlainen kuin tämänhetkisessä ratkaisuvaihtoehdossa. Tässä ratkaisuvaihtoehdossa on kuvattu lisäksi raportin muodostamisprosessia (`Luo raportti`). Tämä toteutettaisiin ohjelmallisesti siten, että ohjelma käy spesifikaation jokaisen osan läpi ja lisää niihin tarvittavat mittausarvot. `Luo raportti`-prosessia on kuvattu tarkemmin kuvassa 8.4.



Kuva 8.4. *Generate report-prosessin tarkempi kuvaus.*

Kuvasta 8.4 nähdään, että toteutus on vastaavanlainen kuin tämänhetkisessä raporttikomponentin toteutuksessa. Kuitenkin toteutus eroaa siltä osin, että siitä puuttuu raporttipohja kokonaisuudessaan. Lisäksi kun tehtyjen mittausten data muutetaan XML-muotoon, mittausarvot luetaan käyttäen XPath-kyselyitä. XPath-kyselyihin tarvittavat parametrit luetaan spesifikaation verifiointiarvojen parametreista eli `specParameterValue`-elementeistä. Tämänhetkisten parametrien avulla mittausarvoa ei voida hakea yksikäsitteisesti. Näin ollen verifiointiarvoille tulee määrittää lisää parametreja, joiden perusteella mittausarvojen hakukysely olisi yksikäsitteinen. Tällä hetkellä tietomallista löytyy suurin piirtein kaikki tarvittavat tiedot, joiden avulla mittausarvojen haku olisi mahdollista. Kuitenkin tarvitaan täysin uusi kytkös mittaustyyppeihin, joka kertoisi, minkätyyppisiin mittauksiin kyseinen verifiointiarvo liittyy. Mittausarvoa haettaessa olennainen parametri on mittaustyyppi. Tämän jälkeen muodostettu XML, jossa ovat mittausarvot mukana, muutetaan XSL-FO-muotoon ja annetaan PDF-muuntajalle PDF-dokumentin generoimiseksi.

Edellä mainitulla tavalla raportin luominen onnistuisi, mutta se ei yksinään riitä, koska tämänhetkisessä spesifikaatiossa ei ole mahdollista esittää graafeja tai mittausten raakadataa, jotka ovat oleellisia osia raportissa. Tämän vuoksi ohjelmallisen komponentin toteuttamisen lisäksi spesifikaatiokomponentin tulee tukea graafisten kuvaajien ja mittausten raakadatan lisäämistä spesifikaatiopohjaan. Kuvaajalle olisi erittäin vaikeaa antaa sellaisia parametreja, joiden perusteella voitaisiin luoda XSLT-muunnossäännöt ja XPath-kyselyt, jotka muodostaisivat kuvaajan raporttiin. Tämä olisi myöskin haasteellista toteuttaa ohjelmallisesti. Näin ollen parempi ratkaisu olisi lisätä tuki kuvaaja- ja raakadataelementeille spesifikaatioon, joiden sisältö ei olisi muuta kuin viittaus tietomallissa olevaan instanssiin. Instanssi sisältäisi XSLT-muunnossäännöt ja XPath-kyselyt, joiden avulla kuvaaja tai raakadata muodostetaan mittauksista. Tässä siis toimitaan samoin kuin tällä hetkellä eli raporttikomponentin avulla muodostetaan kuvaajan ja raakadatan muodostamiselle XSLT-muunnossäännöt ja XPath-kyselyt (`Luo kuvaaja- & raakadata XSLT-/XPath-hakukyselyt`), joihin tehdään viittaus spesifikaatiopohjan kuvaaja- ja raakadata-elementeistä (kuva 8.3). Näin ohjelmakomponentti, joka muuttaa spesifikaation raportiksi, voi ajaa jo toteutettua toteutusta kuvaajan ja raakadatan muodostamiselle ja antaa sille syötteeksi viittauksen päässä olevan datan. Näin saa-

daan toteutettua raportti, jossa on oleelliset asiat, ohjelmallisen komponentin ja tietomalliin tehtävillä pienillä lisäyksillä.

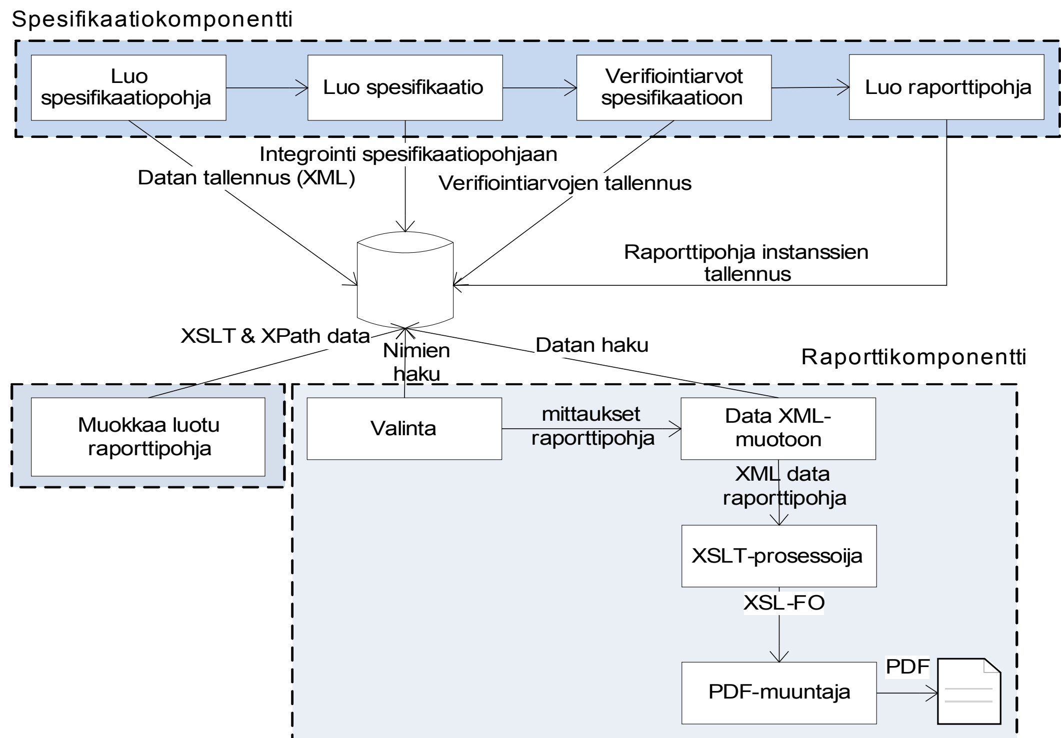
Kuvaajien ja mittausten raakadatojen esittäminen raportissa tuo pienen rajoituksen. Rajoitus liittyy siihen, että raporttikomponentin avulla tulee muodostaa tarvittavat kuvaaja- ja raakadatakyselyt ja liittää ne spesifikaatiopohjaan ennen kuin raportin muodostaminen olisi mahdollista. Toisaalta tämä ei ole ehdoton rajoitus.

Tämä ratkaisuvaihtoehto poistaa kaksinkertaisen työmäärän, jossa ensin luotiin spesifikaatiopohja ja tämän jälkeen vielä vastaava raporttipohja. Nyt raporttipohjaa voidaan käyttää mukautettujen raporttien muodostamiseen. Tämä ratkaisuvaihtoehto on yhtä hyvin käytettävissä parannellun tietomallin kanssa. Paranneltu tietomalli ei tuo suuria hyötyjä tähän ratkaisuvaihtoehtoon, koska sekä parannellun että tämänhetkisen tietomallin yhteydessä joudutaan tekemään samat toiminnot.

Tämän ratkaisuvaihtoehdon huonona puolena on, että spesifikaatiopohjaa ei ole mahdollista muokata muulta osin kuin mitä vaihtoehtoja spesifikaatiokomponentti tarjoaa. Raporttikomponentti taas tarjoaa XSLT- ja XPath-sääntöjen avulla minkä tahansa tyyppisen toteutuksen. Asiakkaalta voi tulla muutostarve uudentyyppisen mittausdatan esittämiseen. Mikäli uutta mittausdataa tuetaan spesifikaatiokomponentin avulla, se vaatii muutoksia sekä ohjelmaan että tietomalliin. Raporttipohjan yhteydessä tämä ei olisi ongelma, koska raporttipohjaan tehdään uusi osa, joka käsittelee kyseistä ominaisuutta. Tietenkin tämäkin vaatii muutoksia tietomalliin, muttei ohjelmaan. Ratkaisuvaihtoehto on herkkä muutoksille.

8.3 Spesifikaation muuttaminen raporttipohjaksi

Toisessa ratkaisuvaihtoehdossa ei muodosteta raporttia suoraan spesifikaatiopohjasta. Ratkaisuvaihtoehto toteuttaa erillisen ohjelmakomponentin, muuntimen (`Luo raporttipohja`), joka muuntaa spesifikaation raporttipohjaksi. Kuvaan 8.5 on hahmoteltu ratkaisuvaihtoehtoa.



Kuva 8.5. *Hahmotelma ratkaisuvaihtoehdosta kaksi: spesifikaation muuttaminen raporttipohjaksi.*

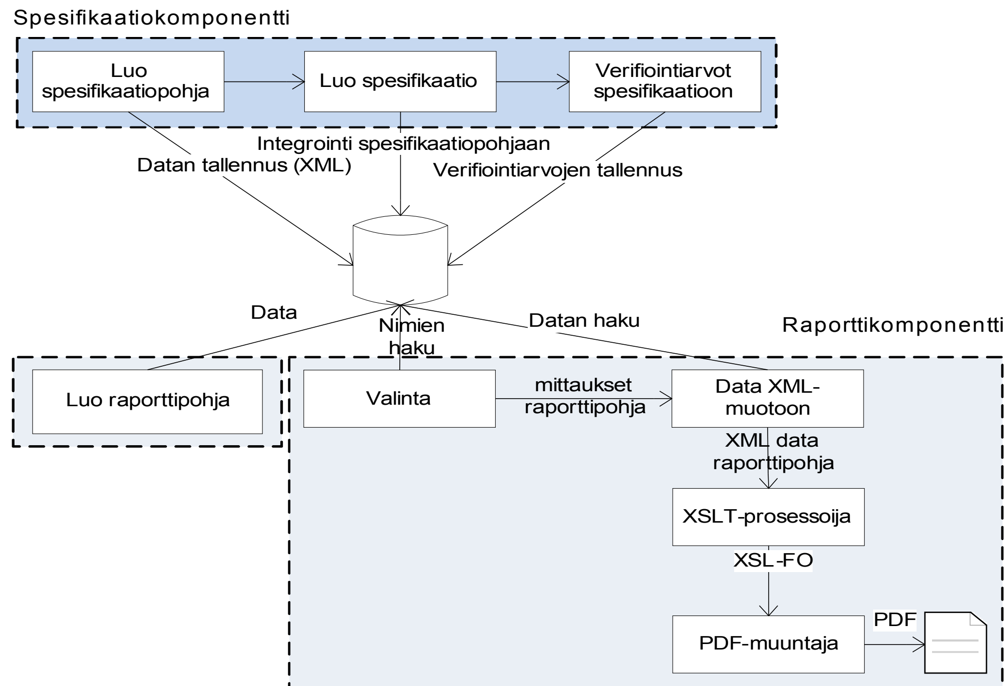
Alkuosa pysyisi tässäkin ratkaisuvaihtoehdossa ennallaan, mutta lisänä tulee muuntimen toiminnallisuus. Muuntimen tarkoituksena olisi muodostaa jokaisesta spesifikaation osasta raporttipohjan osa. Muunnosten yhteydessä muodostettaisiin myöskin mittausarvojen hakua varten tarvittavat XPath-kyselyt. XPath-kyselyitä varten tulee tehdä samat toimenpiteet kuin edellisessä ratkaisuvaihtoehdossa eli verifiointiarvoille tulee määrittää lisää parametreja mittausarvojen hakua varten. Näin ollen myöskin tämä ratkaisuvaihtoehto vaatii tietomalliin lisäyksiä. Muodostetut raporttipohjan osat lisätään raporttikomponentin tietomalliin liittyviin luokkiin omina instansseina. Näistä instansseista muodostetaan raporttipohja, joka nimetään samaksi kuin spesifikaation `Title`-arvo. Nimen perusteella raporttipohjista on helposti tunnistettavissa spesifikaatiokohtaiset pohjat. Tämän jälkeen raporttipohjaan on mahdollista tehdä muutoksia (`Muokkaa luotu raporttipohja`) lisäämällä graafeja ja mittauksen raakadataa samalla tavalla kuin raporttipohjaan tällä hetkellä. Tämä ei muuttaisi toimintatapaa oleellisesti, koska raportin muodostamisprosessin loppuosa pysyy samanlaisena kuin tällä hetkellä. Ratkaisuvaihtoehto on käytettävissä yhtä hyvin parannellun tietomallin kanssa. Paranneltu tietomalli auttaa hieman muuntimen toiminnallisuuden toteutuksessa, sillä muuntimen ei tarvitse erikseen erottaa spesifikaation osia toisistaan vaan ne olisivat omina instansseinaan valmiiksi. Nämä yksittäiset instanssit voidaan kääntää raporttipohjan instansseiksi.

Tässä vaihtoehdossa muutostenhallinta olisi varsin työlästä, koska muutos joudutaan tekemään spesifikaation lisäksi raporttipohjan osiin. Toisaalta muutoksien jälkeen olisi parempi muodostaa raporttipohja uudelleen muuttuneesta spesifikaatiopohjasta. Tällöin joudutaan poistamaan edeltävä versio tietokannasta. Mikäli tietoja ei muisteta poistaa, tietokantaan kerääntyy vanhentunutta tietoa. Tämä voi aiheuttaa muistiin liittyviä ongelmia varsinkin kun mittausohjelmistoa voidaan käyttää erilaisten tuotteiden ominaisuuksien mittaamiseen.

Tämän ratkaisun hyötynä on, että spesifikaatiokomponentin ja raporttikomponentin vastuut pysyvät ennallaan. Spesifikaatiokomponentin avulla muodostetaan spesifikaatioita. Raporttikomponentin avulla täydennetään spesifikaatioita ja muodostetaan raportteja. Tämän ratkaisuvaihtoehdon avulla vältytään ylimääräisen sidonnan lisäämisestä spesifikaatiokomponentin ja raporttikomponentin välille, jota ensimmäinen ratkaisuvaihtoehto vaatisi. Toisin sanoen graafeja ja mittausten raakadataa ei tarvitse erikseen lisätä spesifikaatiopohjaan.

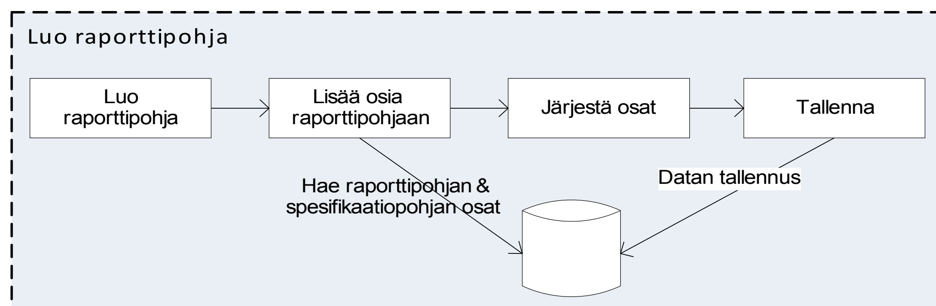
8.4 Spesifikaatiopohjan osien käyttäminen raporttipohjassa

Kolmas ratkaisuvaihtoehto vaatii rakenteellisia muutoksia tietomalliin ja muutoksia ohjelmakoodiin. Toisin sanoen tämä vaihtoehto liittyy paranneltuun tietomalliin eli tietomallin rakenne muutetaan aiemmin esitellyn tietomallin (kuva 7.4) mukaiseksi. Tällöin raporttipohjan osia ei tarvitse erikseen muodostaa ohjelmakomponentin avulla. `TemplateData`-instanssit, joita luodaan spesifikaatiopohjan muodostamisen yhteydessä, voidaan käyttää pienillä muutoksilla raporttipohjan muodostamiseen. Kuvaan 8.6 on hahmoteltu kolmas ratkaisuvaihtoehto.



Kuva 8.6. *Hahmotelma ratkaisuvaihtoehdosta kolme: spesifikaatiopohjan osien käyttäminen raporttipohjaa muodostaessa.*

Kuvasta nähdään, että spesifikaatiokomponentin toiminnallisuus ei poikkea tämänhetkisestä toteutuksesta millään tavalla. Kuitenkin muutokset tietomallissa johtavat ohjelmallisiin muutoksiin. Ohjelman tulee tukea ainakin osien muodostamista, niiden muokkaamista ja liittämistä spesifikaatiopohjaan. Raportin muodostamisen kannalta samoja `TemplateData`-instansseja voidaan käyttää raporttipohjan muodostamiseen (`Luo raporttipohja`), joita käytetään spesifikaatiopohjan muodostamiseen. Pieniä korjauksia kuitenkin joudutaan tekemään XML:ään, koska tämänhetkisen raporttikomponentin yhteydessä oleva XML eroaa spesifikaatiokomponentin kautta muodostetusta XML:stä. `Luo raporttipohja`-prosessia on kuvattu tarkemmin kuvassa 8.7.



Kuva 8.7. *Luo raporttipohja-prosessin tarkempi kuvaus.*

Kuvasta 8.7 nähdään, että ensin luodaan raporttipohja, johon voidaan lisätä `TemplateData`-instanssit ja/tai XSLT-muunnossäännöistä ja XPath-kyselyistä koostuvat

instanssit. Tämän jälkeen järjestetään lisätyt osat ja tallennetaan muodostettu raporttipohja. Näin saadaan aikaiseksi raporttipohja, josta voidaan muodostaa raportti mittausarvoineen. Ohjelmallisesti tulee kuitenkin toteuttaa mittausarvojen haku verifiointiarvoparametrien perusteella. Toteutus olisi kuitenkin vastaava kuin muissakin vaihtoehdoissa, mutta suoritusjärjestys olisi poikkeava. Suoritus voidaan tehdä kahdessa vaiheessa joko silloin, kun `TemplateData`-instanssi liitetään osaksi raporttipohjaa (`Luo raporttipohja`) tai kun muodostetaan raportti (`Data XML-muotoon`) eli ajonaikaisesti. Ajonaikaisessa vaihtoehdossa on huonona puolena se, että se joudutaan suorittamaan joka kerta, kun raporttia luodaan. Toinen vaihtoehto kuitenkin joudutaan tekemään uudelleen, kun dataan tehdään muutoksia. Näin ollen ensimmäinen vaihtoehto on parempi, jos otetaan huomioon suorituskyky ja toinen vaihtoehto on parempi, jos otetaan huomioon muutostenhallinta. Suorituskyky ei tule huononemaan nykyisestään paljon, joten muutostenhallinnan kannalta ajonaikainen vaihtoehto on kuitenkin parempi tässä tapauksessa.

Tämä ratkaisuvaihtoehto yhdistää raporttikomponentin ja spesifikaatiokomponentin hyvin löyhästi. Kun raporttipohja luodaan, niin siihen liitetään `TemplateData`-instanssit. Liittämisellä tarkoitetaan, että raporttipohjaa vastaavasta luokkainstanssista tehdään viittaus `TemplateData`-instansseihin, jotka liittyvät kyseiseen raporttipohjaan. Raporttipohjan kautta ei kuitenkaan voida muokata `TemplateData`-instansseja vaan muutokset tulevat spesifikaatiokomponentin kautta. Tämä helpottaa myöskin muutostenhallintaan liittyvää ongelmaa. Tämä siksi, että vasta ajonaikaisesti haetaan `TemplateData`-instanssin data, jolloin kaikki viimeisimmät muutokset, joita spesifikaatioon on tehty, tulee otettua huomioon raporttia luodessa. Mikäli vasta ajonaikaisesti muodostetaan XPath-kyselyt, joiden avulla haetaan mittausarvot, tämä vaihtoehto ei tee mitään ylimääräistä raporttipohjan luonnin yhteydessä. Tietenkin tässä ratkaisuvaihtoehdossa tulee raporttikomponentin toteuttaa tuki `TemplateData`-instanssin integroimiselle osaksi raporttipohjaa tai mahdollisuus valita spesifikaatio, jonka kaikki `TemplateData`-instanssit integroidaan raporttipohjaan. Raporttipohjaan tulisi tällöin näkyviin `TemplateData`-instanssin nimi ja versio.

8.5 Johtopäätökset

Työn tehostamiseksi on suunniteltu kolme eri ratkaisuvaihtoehtoa, joita esiteltiin edellisissä kohdissa. Tässä esitetään lyhyesti yhteenveto siitä, mikä ratkaisuvaihtoehdoista olisi järkevin toteuttaa. Taulukossa 8.1 on esitetty vertailutiedot eri ratkaisuvaihtoehtojen osalta.

Taulukko 8.1. *Ratkaisuvaihtoehtojen vertailu.*

	Ratkaisuvaihtoehto 1	**Ratkaisuvaihtoehto 2**	**Ratkaisuvaihtoehto 3**
Työmäärä	Pienenee (+)	Pienenee (+)	Pienenee (+)
Muutostenhallinta	Helpottuu (+)	Hieman vaikeutuu (-)	Helpottuu (+)
Tietomalli	Ei muutoksia (+)	Ei muutoksia (+)	Tarvitaan muutoksia (-)
Komponenttien vastuunjako	Muuttuu (-)	Ei muutosta (+)	Ei muutosta (+)
Raporttipohjan muuttaminen	Ei mahdollista (-)	Mahdollista (+)	Ei mahdollista (-)
Osien uudelleen-käytettävyys	Ei mahdollista (-)	Ei mahdollista (-)	Mahdollista (+)

Aiemmin esitettyjen ratkaisuvaihtoehtojen hyvien puolien ja taulukossa 8.1 esitetyn vertailun perusteella ratkaisuvaihtoehdoista joko ratkaisuvaihtoehto kaksi tai kolme olisi järkevin toteuttaa. Nämä ratkaisuvaihtoehdot tuottavat eniten verrattuna niistä koituviin kustannuksiin. Ratkaisuvaihtoehto kaksi olisi kuitenkin järkevää toteuttaa ainoastaan tämänhetkisen tietomallin tapauksessa, koska tämä tuottaisi eniten hyötyä. Parannellun tietomallin yhteydessä ei ole järkevää toteuttaa ratkaisuvaihtoehtoa kaksi, koska muuntimen toiminnallisuus olisi käytännössä aivan turha. Data olisi muutenkin käytettävissä pienillä muutoksilla XML-rakenteeseen raporttipohjien muodostamisessa ja lisäämällä raporttikomponenttiin integrointi. Integrointi antaisi mahdollisuuden valita spesifikaation osia osaksi raporttipohjaa.

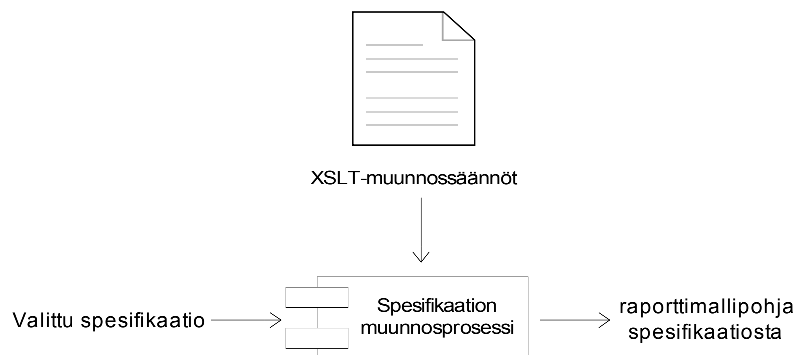
Yhteenvetona voidaan todeta, että kolmas ratkaisuvaihtoehto on paras mahdollinen, koska se helpottaa muutostenhallinnassa, ja sama data olisi käytettävissä raporttipohjan muodostamisessa. XML:ään tarvittavat muutokset olisi hyvä tehdä ajonaikaisesti, koska muuten datasta joudutaan säilömään useita versioita. Tämä vaikeuttaa muutostenhallintaa ja ylläpidettävyyttä. Ajonaikaisen sidonnan hyötynä olisi lisäksi, että spesifikaatiodataan tehdyt muutokset heijastuisivat automaattisesti raporttipohjassa, koska data on tarkoitus hakea vasta raportin muodostamisvaiheessa. Raportin muodostamisvaiheessa mittausarvot haettaisiin toteutetun ohjelmakomponentin avulla. Mittausarvojen haku hidastaa raportin muodostamisprosessia, mutta tämä kuitenkin kompensoituu ylimääräisen työn poistumisella ja työn nopeutumisella prosessin muissa osissa.

9 TOTEUTUKSEN ARVIOINTI

9.1 Ratkaisuvaihtoehdon toteutuksen kuvaus

Tässä diplomityössä toteutettiin yksi ratkaisuvaihtoehdoista, joka helpottaa raporttipohjan muodostamista. Suurin hyöty toteutuksesta saadaan, kun raporttipohjan rakenne vastaa täsmälleen spesifikaation rakennetta. Eri ratkaisuvaihtoehdoista on kerrottu tarkemmin luvussa 8. Tässä työssä toteutettiin ratkaisuvaihtoehto kaksi (kohta 8.3), joka muuntaa spesifikaation raporttipohjaksi. Syynä valintaan oli ratkaisuvaihtoehdon pienempi työmäärä, koska se ei vaadi muutoksia tietomalliin. Pienen työmäärän lisäksi ratkaisuvaihtoehto kaksi tuottaisi eniten hyötyä tämänhetkisen tietomallin tapauksessa.

Toteutus tehtiin siten, että ensin spesifikaation XML muutetaan raporttipohjan hyväksymään muotoon. Tämä aikaansaatiin luomalla XSLT-muunnossääntöjä (Liite 1: XSLT-muunnossäännöt) sisältävä tiedosto, jota vasten spesifikaation XML muutetaan uuteen muotoon. Kuvaan 9.1 on hahmoteltu spesifikaation muunnostoiminnallisuutta.



Kuva 9.1. *Spesifikaation muunnosprosessi.*

XSLT-tiedoston avulla spesifikaation XML käydään läpi ja muutetaan sen tiedot raporttipohjan tarvitsemaan muotoon eli raporttimallipohjaksi. Otsikot ja tekstikappaleet on suoraviivaista muuttaa, koska tekstikappaleet ovat sellaisenaan käytettävissä ilman muutoksia ja otsikoista pitää ainoastaan poistaa numeroinnista vastaavat attribuutit. Spesifikaatiossa ja raporttimallipohjassa tulokset esitetään taulukkomuodossa, spesifikaatiossa kuitenkaan ei eroteltu taulukon otsikko- ja sisältösoluja. XSLT-muunnossäännöissä on varsin hankalaa tehdä erottelu otsikko- ja sisältösolujen välillä ilman yksikäsitteistä juurielementtiä. Näin ollen spesifikaation XML:ään tehtiin muutos taulukkojen osalta. Muutoksessa taulukkojen otsikkosolujen juurielementiksi laitettiin *thead*-elementti ja

sisältösolujen juurielementiksi *tbody*-elementti. XSLT-koodin tekeminen oli tämän lisäyksen myötä hyvin suoraviivainen taulukkojen osalta. Tämän lisäksi verifiointiarvoille määritettiin lisää parametreja, joiden perusteella mittausarvojen hakukysely olisi yksikäsitteinen. Mittaustyyppi on yksi olennaisimmista parametreista.

Mittaustuloksen hakukyselyä ei toteutettu osaksi XSLT-muunnostiedostoa, koska tämä olisi ollut varsin haastavaa ja aikaa vievää. Toteutuksessa menetellään siten, että spesifikaation XML:stä poimitaan mittaustuloksen hakukyselyn muodostamista varten tarvittavat parametrit uuden elementin attribuuteiksi. Nämä attribuutit hyödynnetään mittaustuloksen hakukyselyn muodostamisessa ohjelmallisesti. Ohjelmallisesti muodostetaan mitaustuloksen hakukysely, joka liitetään osaksi luotua raporttimallipohjaa.

Näin aikaansaatiin raporttimallipohja, jota integroitiin raporttipohjaan. Muodostetun raporttipohjan avulla luotiin raportti olemassa olevista mittaustuloksista.

Toteutus olisi ollut mahdollista tehdä joko spesfikaatio- tai raporttikomponenttiin. Komponenttien vastuunjaon perusteella raporttikomponentin avulla luodaan raportteja ja raporttipohjia. Näin ollen toiminnallisuus, joka muuntaa spesifikaation raporttipohjaksi, päädyttiin toteuttamaan raporttikomponenttiin. Näin spesifikaatiokomponentin vastuuksi ei tule raporttipohjan muodostaminen, joka olisi rikkonut komponenttien vastuunjaon periaatteen.

Käyttäjälle annetaan kaksi vaihtoehtoa muodostaa raporttimallipohja: ensimmäinen vaihtoehto on nykyinen toiminnallisuus eli luodaan tyhjä raporttimallipohja, ja toisen vaihtoehdon avulla käyttäjä voi muodostaa raporttimallipohjan spesifikaatiosta. Muunnoksen jälkeen tulos tulee näkyviin raporttimallipohjan tekstieditoriin, jossa käyttäjä voi tehdä tarpeelliset muutokset ja tallentaa lopputuloksen tietokantaan. Tämän jälkeen käyttäjä voi luoda raporttipohjan, johon integroidaan spesifikaatiosta luotu raporttimallipohja ja muut tarpeelliset raporttimallipohjat, jotka halutaan raporttiin.

9.2 Toteutetun ratkaisuvaihtoehdon arviointi

9.2.1 Arvioinnin suoritus

Ratkaisuvaihtoehdon toteutuksen tavoitteena oli säästää työaikaa, joka kuluu raporttimallipohjan luomiseen spesifikaatiosta. Työajan säästymistä arvioitiin tekemällä uudelleen tässä työssä esitelty esimerkkispesifikaatio (kuva 7.3).

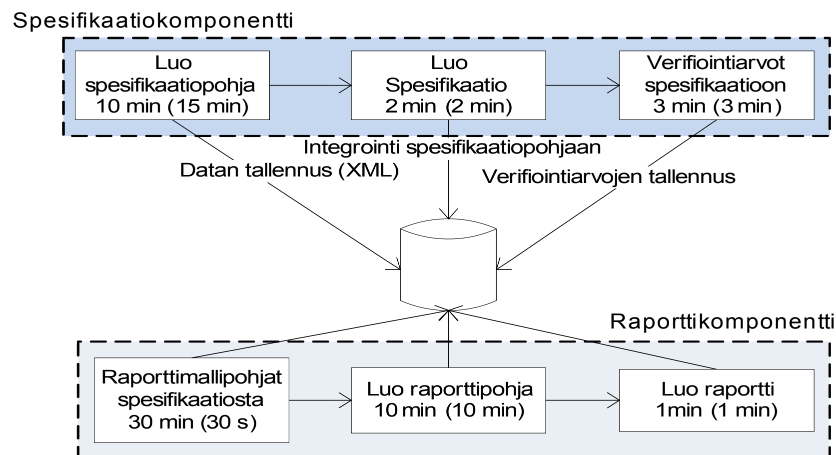
Arvioinnissa käytetään todellista käyttötilannetta kuvaavaa skenaariota. Tietyn valmistajan kaiuttimien akustisille ominaisuuksille on suoritettu mittauksia ja mittaustulokset ovat tietokannassa. Olemassa olevista mittaustuloksista halutaan tuottaa raportti verifiointiarvoineen. Skenaarion toteutumiseksi on luotava spesifikaatio, joka sisältää verifiointiarvot. Spesifikaatiota varten on ensin luotava spesifikaatiopohja, joka kuvaa verifiointidokumentin rakennetta. Tämän jälkeen on luotava raporttimallipohja, joka vastaa verifiointidokumentin rakennetta ja sisältää lisäksi mittaustulosten hakukyselyt. Sitten

muodostetaan raporttipohja, johon integroidaan raporttimallipohjat. Lopuksi tuotetaan raportti mittaus- ja verifiointiarvoineen.

Skenaariolla on tarkoitus mitata työaikaa, joka kuluu eri työvaiheisiin. Työaikaa mitattiin sekä nykyisellä arkkitehtuurilla että parannetulla arkkitehtuurilla, joka sisältää toteutetun ratkaisuvaihtoehdon. Työaika mitattiin tarkkailemalla tehtävän suorituksen aloitus- ja lopetusaikaa. Näin saatiin laskettua aika, joka kului tehtävän suorittamiseen. Kukin tehtävä suoritettiin ympäristöllä, joka vastasi täysin projektin kehittämiseen käytettävää ympäristöä. Näin ollen olosuhteet vastasivat todellista tilannetta. Suoritin jokaisen vaiheen itse, jotta eri vaiheisiin käytetyt työajat olisivat keskenään vertailtavissa. Likimain samoihin tuloksiin päästäisiin vaikka työvaiheet suorittaisi joku muu projektin työntekijöistä. Pieniä eroja kuitenkin voi syntyä työajoissa johtuen siitä, kuinka paljon on ollut tekemisissä XSLT- ja XML-tekniikoiden kanssa.

9.2.2 Arvioinnin tulokset

Kuvassa 9.2 on esitetty suoritetun arvioinnin eri vaiheet. Kuvassa 9.2 olevat vaiheet on suoritettu spesifikaatio- ja raporttikomponentilla. Kuvan yläosan vaiheet on suoritettu spesifikaatiokomponentilla ja alaosan vaiheet raporttikomponentilla.



Kuva 9.2. *Eri vaiheisiin kulunut työaika nykyisellä ja parannetulla arkkitehtuurilla.*

Kuvan 9.2 eri vaiheisiin on merkitty kaksi aikamäärettä. Molemmat kuvaavat vaiheen suorittamiseen kulunutta työaikaa. Ensimmäinen on vaiheen suorittamiseen kulunut työaika nykyisellä arkkitehtuurilla ja toinen, sulkeissa oleva, parannetulla arkkitehtuurilla. Tarkastelemalla työaikoja huomataan, että ainoastaan kahdessa työvaiheessa tapahtuu muutoksia: `Luo spesifikaatiopohja` ja `Raporttimallipohjat spesifikaatiosta`. Ratkaisuvaihtoehdon toteutuksen myötä spesifikaatiopohjaa luotaessa siihen on määritettävä lisää parametreja, jotta mittaustuloksen hakukysely olisi yksikäsitteinen. Lisätyöaika on kuitenkin suhteellisen pieni johtuen siitä, että parametrit voidaan määritellä visuaalisen käyttöliittymän avulla. Spesifikaation luominen ja veri-

fiointiarvojen kirjaaminen pysyy ennallaan. Suurin hyöty saavutetaan, kun on luotava spesifikaation rakennetta vastaavat raporttimallipohjat ja sisällytettävä näihin mittaustulosten hakukyselyt. Tämä tehdään toteutetussa ratkaisuvaihtoehdossa automatisoidusti, joten työaika säästyy huomattavasti. Loput työvaiheista pysyvät ennallaan eli luodaan raporttipohja ja integroidaan siihen raporttimallipohjat sekä tuotetaan raportti raporttipohjan avulla.

Raporttimallipohjien luomisessa säästettävä työaika on riippuvainen spesifikaation pituudesta. Yhden raporttimallipohjan luomiseen menee arviolta noin puoli tuntia. Näin ollen jos spesifikaatio sisältää kymmenen ominaisuutta, joista on luotava raporttimallipohjat, niin niiden luomiseen kuluu viisi tuntia nykyisellä arkkitehtuurilla. Parannetulla arkkitehtuurilla aikaa kuluisi vain viisi minuuttia.

Ratkaisuvaihtoehtojen suunnittelun tavoitteena oli hyödyntää spesifikaation tietoa raporttien ja raporttipohjien muodostamiseen. Tämä tavoite toteutui. Sellainen ratkaisuvaihtoehto on riittävä, joka sekä säästää työaikaa että nopeuttaa raporttien luomisprosessia.

Raporttimallipohjan luomisesta ratkaisuvaihtoehdon kaksi mukaisesti on hyötynä ajan säästymisen lisäksi se, että käyttäjä voi tehdä muutoksia luotuun pohjaan. Mittaustulokset saatetaan esittää eri muodossa riippuen mitattavasta ominaisuudesta. Esimerkiksi jokin mittaustulos esitetään yhden desimaalin tarkkuudella, kun taas toinen neljän desimaalin tarkkuudella tai mittaustulos voidaan esittää prosentuaalisesti. Mittaustuloksen formaatti voidaan määrittää osana spesifikaation verifiointiarvoa, mutta se voidaan tarpeen tullen muuttaa suoraan raporttimallipohjaan. Tässä ratkaisuvaihtoehdossa käyttäjällä on mahdollisuus tehdä muutoksia, kun taas muut ratkaisuvaihtoehdot eivät tarjoa kyseistä ominaisuutta.

Aika, joka kului toteutukseen, oli varsin pieni saataviin hyötyihin nähden. Toteutuksessa kuitenkin tulee vielä parantaa virheiden käsittelyä, koska siihen ei vielä kiinnitetty huomiota, vaan tarkoituksena oli tuottaa aluksi prototyyppi.

Toteutuksen huonona puolena on mittaustuloksen hakukyselyn muodostamislogiikka. Tämä logiikka on herkkä muutoksille, jotka voidaan tehdä spesifikaation XML:ään. Spesifikaation XML:ään voidaan lisätä uusia parametreja verifiointiarvoille. Nämä parametrit on käsiteltävä myöskin ohjelmakoodissa siltä osin kuin ne vaikuttavat mittaustuloksen hakukyselyyn. Mikäli verifiointiarvosta poistetaan jokin parametri, tämä ei vaadi suuria muutoksia ohjelmakoodissa. Tässä tapauksessa tulee ainoastaan poistaa ohjelmakoodissa oleva ylimääräinen parametrin käsittely.

Raporttipohjan generoiminen raporttimallipohjista on edelleen manuaalista eli käyttäjä luo uuden raporttipohjan, johon integroi tarvittavat raporttimallipohjat. Tämä olisi voitu tehdä myöskin automatisoidusti, mutta se ei olisi helpottanut työtä kovin paljon, koska raporttipohjaan useimmiten joudutaan lisäämään muitakin raporttimallipohjia. Raporttipohjan luonti on hyvin suoraviivaista ja nopeaa, joten automaattisen toiminnal-

lisuuden toteutuksella ei oltaisi saavutettu suurta hyötyä verrattuna tarvittavaan työmäärään.

9.3 Vertailu ratkaisuvaihtoehtoon kolme

Ratkaisuvaihtoehdon kaksi prototyypin tekemiseen kului kolme työpäivää. Virheiden käsittelyn ja mittaustuloksen hakulogiikan parantamiseen menisi arviolta vielä toiset kolme työpäivää. Lisäksi spesifikaatiopohjan XML:ään tarvitaan ohjelmallisia muutoksia rakenteen ja validoinnin osalta. Tähän menisi arviolta kaksi työpäivää. Näin ollen ratkaisuvaihtoehdon toteutukseen kuluisi yhteensä kahdeksan työpäivää. Tämä on suhteellisen pieni työmäärä verrattuna saataviin hyötyihin.

Mikäli tämän jälkeen toteutettaisiin ratkaisuvaihtoehto kolme, jo toteutetun ratkaisuvaihtoehdon osia olisi mahdollista käyttää uudelleen. Ratkaisuvaihtoehdon kolme toteutus vaatisi suunnilleen seitsemän työpäivää. Työmäärän arvioinnissa on otettu huomioon osien uudelleenkäytettävyys. Toteutuksen aikana tehtäisiin tarvittavat tietomallimuutokset, niistä johtuvat korjaukset, spesifikaation osien integroiminen raporttipohjaan ja raportin tuottamiseen liittyvät toimenpiteet. Ratkaisuvaihtoehdon kolme vaativin osuus on muutosten tekeminen tietomalliin ja näistä muutoksista aiheutuvien ongelmien korjaus. Muu osuus on suhteellisen suoraviivainen, kuten ratkaisuvaihtoehdossa kaksi.

Tällä hetkellä ratkaisuvaihtoehdon kaksi prototyyppi on kohtalainen. Mikäli sitä kehitettäisiin vielä hiukan, se olisi riittävä raportin tuottamisen kannalta. Toteutuksen riittävyyden lisäksi vältytään tietomallin muutoksilta ja niiden aiheuttamilta korjaustoimenpiteiltä. Tämän lisäksi ratkaisuvaihtoehdon kaksi myötä on mahdollista tehdä muutoksia raporttimallipohjaan ilman, että muutetaan spesifikaatiota. Tämä on varsin kätevää mukautettujen raporttien luomisessa, jolloin tietyn spesifikaation rakenteesta voidaan tuottaa raporttimallipohja ja karsia siitä tarpeettomat osat pois.

Yhteenvetona voidaan todeta, että toteutettu ratkaisuvaihtoehto on riittävä raportin muodostamisen kannalta eikä välttämättä ratkaisuvaihtoehtoa kolme tarvitse toteuttaa. Jos tietomalliin tehdään tulevaisuudessa muutoksia, jotka puoltaisivat ratkaisuvaihtoehtoa kolme, se olisi toteuttavissa suhteellisen pienellä työmäärällä.

10 YHTEENVETO

Tässä työssä tehtyjen havaintojen perusteella takaisinmallinnustyökalut ovat hyviä apuvälineitä ohjelmiston rakenteen ymmärtämisessä. Kuitenkaan niitä ei voida käyttää yksinään johtuen siitä, että niiden avulla ei saada kaikkea tietoa ohjelmasta. Lisäksi tarvitaan tietoja, joita saadaan ainoastaan tutkimalla ja analysoimalla ohjelmakoodia eli käyttäen staattista takaisinmallinnusta. Näin takaisinmallinnustyökalun avulla saadusta rakenteesta voidaan poistaa ylimääräiset ja turhat sidokset luokkien väliltä tai lisätä uusia sidoksia tai elementtejä.

Ohjelmakoodin tutkiminen on aina hankalaa varsinkin, kun se on toisen kirjoittamaa. Analysoinnissa auttavat ohjelmiston järkevä rakenne ja siinä sovelletut koodauskäytännöt. Mittausohjelmiston kannalta koodin tutkiminen oli varsin vaivaton johtuen selkeästä rakenteesta ja vastuunjaosta luokkien välillä.

Analysoidun komponentin selkeän rakenteen ja luokkien nimien perusteella luokkakaavio oli helppo järjestää siten, että siitä huomasi, mitä arkkitehtuurityyliä on käytetty. Spesifikaatiokomponentissa on käytetty MVC-arkkitehtuurityyliä. Tämän näkee lisäksi ohjelmakoodista, jossa luokkien vastuut on jaettu siten, että ne vastaavat miltei suoraan malli-näkymä-ohjaimen vastuita. Spesifikaatiokomponentissa on sovellettu MVC-arkkitehtuurityylin lisäksi Tila-, Tehdasmetodi- ja Tarkkailija-suunnittelumallia. Suunnittelumallien käyttö saattoi olla tietoinen tai tiedostamaton, mutta niitä arkkitehtuurissa on kuitenkin hyödynnetty. Tarkkailija-suunnittelumalli esiintyy muutenkin yleisesti MVC-arkkitehtuurityylissä.

Mittausohjelmisto koostuu viidestä komponentista, jotka toimivat täysin toisistaan riippumattomasti. Mittauskomponentti käyttää mittaustulosten raportoimiseen raporttikomponenttia. Niiden välinen yhteys on kuitenkin hyvin löyhä. Raporttikomponentissa tehdyt muutokset eivät vaadi suuria muutoksia mittauskomponenttiin.

Spesifikaatiokomponentin tietomallista löydettiin muutamia korjauskohteita, joihin on esitetty ratkaisuvaihtoehtoja. Tekemällä muutokset spesifikaatiokomponentin tietomalliin ja ohjelmaan voidaan saavuttaa huomattavia etuja sekä datan ylläpitämisessä että muutostenhallinnassa. Lisäksi työtä saadaan tehostettua raportin muodostamisen osalta, jos se vastaa täsmälleen spesifikaatiota. Tähän ongelmaan on tuotettu hyvin perusteltuja ratkaisuvaihtoehtoja, joita voidaan soveltaa joko tämänhetkisen tietomallin tai parannellun tietomallin yhteydessä. Näiden ratkaisuvaihtoehtojen avulla spesifikaation tietoa voidaan hyödyntää raporttien ja raporttipohjien muodostamiseen. Esitellyt ratkaisuvaihtoehdot ovat tämän työn keskeisimpiä tuloksia, joten niiden kuvausten yhteydessä on huomioitu niiden tuomat hyödyt ja haitat. Näistä ratkaisuvaihtoehdoista paranneltuun

tietomalliin soveltuva ratkaisuvaihtoehto on suositeltavin. Siitä on kerrottu tarkemmin kohdassa 8.4. Tässä työssä kuitenkin toteutettiin ratkaisuvaihtoehto kaksi (kohta 8.3), joka muuntaa spesifikaation raporttipohjaksi. Syynä valintaan oli ratkaisuvaihtoehdon pienempi työmäärä, koska se ei vaadi muutoksia tietomalliin. Toteutuksesta ja sen arvioinnista on kerrottu tarkemmin luvussa 9.

LÄHTEET

[1] Altova UModel. [WWW]. [Viitattu 09.06.2012]. Saatavissa: http://www.altova.com/umodel.html.

[2] Begg C. E., Conolly T. M. 2002. Database Systems: A Practical Approach to Design, Implementation, and Management. 3. painos. USA, Pearson Education Limited. 1236 s.

[3] Bray T., Maler E., Paoli J., Sperberg-McQueen C. M., Yergeau F. 2008. Extensible Markup Language (XML) 1.0. WC3. [WWW]. [Viitattu 02.06.2012]. Saatavissa: http://www.w3.org/TR/REC-xml/.

[4] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. 1996. Pattern-Oriented Software Architecture: A System of Patterns. Englanti, Wiley. 449 s.

[5] Canfora G., Penta M. D. New Frontiers of Reverse Engineering. International Conference on Software Engineering, Minneapolis, Minnesota, USA, 23.-25. Toukokuuta 2007. USA, 2007, IEEE Computer Society Press. S. 326-341. [WWW]. [Viitattu 03.06.2012]. Saatavissa: http://dl.acm.org/citation.cfm?id=1254728.

[6] Chikofsky E. J., Cross J. H. Reverse engineering and design recovery: a taxonomy. IEEE Computer Society Press 7(1990)1, s. 13-17. [WWW]. [Viitattu 03.06.2012]. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=43044.

[7] Gamma E., Helm R., Johnson R., Vlissides J. 1994. Desing Patterns: Elements of Reusable Object-Oriented Software. USA, Addison-Wesley. 395 s.

[8] Harsu M. 2003. Ohjelmien ylläpito ja uudistaminen. Helsinki, Talentum Media Oy. 292 s.

[9] Herman I. 2005. Overview of XSLT and XPath. W3C. [WWW]. [Viitattu 12.08.2012]. Saatavissa: http://www.w3.org/Consortium/Offices/Presentations/XSLT_XPATH.

[10] Jahnke J. H., Mûller H. A., Smith D. B., Strorey M-A., Tilley S. R., Wong K. Reverse Engineering: a roadmap. International Conference on Software Engineering, Ireland, 04.-11. Kesäkuuta 2000. New York, NY, USA, 2000, ACM. S. 47-60. [WWW]. [Viitattu 03.06.2012]. Saatavissa: http://dl.acm.org/citation.cfm?id=336526.

[11] Karhunen M. 2003. Käänteistekniikka ohjelmistojen uudistamisen apuvälineenä. Pro gradu -tutkielma. Kuopion yliopisto, Tietojenkäsittelytieteen laitos. 92 s. [WWW]. [Viitattu 03.06.2012]. Saatavissa: http://cs.uef.fi/uku/tutkimus/Teho/MiianGradu.pdf.

[12] Koskimies K., Mikkonen T. 2005. Ohjelmistoarkkitehtuurit. Helsinki, Talentum Media Oy. 250 s.

[13] Merson, P. 2009. Data Model as an Architectural View. USA, Carnegie Mellon University. 35 s. [WWW]. [Viitattu 19.04.2012]. Saatavissa: http://www.sei.cmu.edu/reports/09tn024.pdf.

[14] Metamill Software. [WWW]. [Viitattu 09.06.2012]. Saatavissa: http://www.metamill.com/product.html.

[15] Nurminen M. 2011. XSLT ohjelmointikielenä. Seminaarityö. 2011. Joensuun yliopisto. 18 s. [WWW]. [Viitattu 03.06.2012]. Saatavissa: http://www.mit.jyu.fi/opiskelu/seminaarit/tiesem2011/nurminen11xslt.pdf.

[16] Quin L. E. R. 2010. XML Schema. W3C. [WWW]. [Viitattu 02.06.2012]. Saatavissa: http://www.w3.org/standards/xml/schema.

[17] Salminen A. 2005. Metatiedot organisaatioiden sisällönhallinnassa. Helsinki. 10 s. [WWW]. [Viitattu 27.04.2012]. Saatavissa: http://users.jyu.fi/~airi/papers/Metatietoartikkeli-2005.pdf.

[18] Sparx Systems: Enterprise Architect. [WWW]. [Viitattu 09.06.2012]. Saatavissa: http://www.sparxsystems.com.au/.

[19] StarUML: The Open Source UML/MDA Platform. [WWW]. [Viitattu 09.06.2012]. Saatavissa: http://staruml.sourceforge.net/en/about.php.

[20] Simmison G. C., Witt G. C. 2005. Data Modeling Essentials. 3. painos. USA, Morgan Kaufmann Publishers. 532 s.

[21] Suhonen J. 2000. Metadata älykkäässä oppimisympäristössä. Pro gradu -tutkielma. Joensuun yliopisto, Tietojenkäsittelytiede. 105 s. [WWW]. [Viitattu 27.04.2012]. Saatavissa: ftp://cs.joensuu.fi/pub/Theses/2000_MSc_Suhonen_Jarkko.pdf.

[22] Understanding Metadata: a general introduction to metadata. Bethesda, USA, 2001, National Information Standards Organization Press. 20 s. [WWW]. [Viitattu 27.04.2012]. Saatavissa: http://www.niso.org/publications/press/UnderstandingMetadata.pdf.

LIITE 1: XSLT-MUUNNOSSÄÄNNÖT

```
<!--<?xml version="1.0" encoding="ISO-8859-1"?>-->
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="title"></xsl:template>

  <xsl:template match="section">
    <section>
      <xsl:attribute name="title"><xsl:value-of select="@title"/></xsl:attribute>
      <xsl:apply-templates select="subsection"/>
        <xsl:apply-templates select="p"/>
        <xsl:apply-templates select="table"/>
    </section>
  </xsl:template>

  <xsl:template match="subsection">
    <subsection>
      <xsl:attribute name="title"><xsl:value-of select="@title"/></xsl:attribute>
      <xsl:apply-templates select="subsubsection"/>
      <xsl:apply-templates select="p"/>
      <xsl:apply-templates select="table"/>
    </subsection>
  </xsl:template>

  <xsl:template match="subsubsection">
    <subsubsection>
      <xsl:attribute name="title"><xsl:value-of select="@title"/></xsl:attribute>
      <xsl:apply-templates select="p"/>
        <xsl:apply-templates select="table"/>
    </subsubsection>
  </xsl:template>

  <xsl:template match="p">
    <p>
      <xsl:value-of select="."/>
    </p>
  </xsl:template>

  <xsl:template match="table">
    <table layout="fixed" numbering="no">
      <xsl:apply-templates select="thead"/>
      <xsl:apply-templates select="tbody"/>
    </table>
    <xsl:apply-templates select="caption"/>
  </xsl:template>

  <xsl:template match="caption">
    <caption>
      <xsl:value-of select="."/>
    </caption>
  </xsl:template>

  <xsl:template match="specitem">
    <xsl:value-of select="/value/@value"/>
  </xsl:template>
```

```
  <xsl:template match="thead">
    <xsl:variable name="maxRowSpan" select="count(tr)"/>
    <thead>
      <xsl:for-each select="tr">
        <tr>
          <xsl:for-each select="th">
            <th>
              <xsl:attribute name="rowspan">
                <xsl:value-of select="@rowspan"/>
              </xsl:attribute>
              <xsl:attribute name="colspan">
                <xsl:value-of select="@colspan"/>
              </xsl:attribute>
              <xsl:value-of select="." />
            </th>
          </xsl:for-each>
          <xsl:if test="position() = 1">
            <th>
              <xsl:attribute name="rowspan">
                <xsl:value-of select="$maxRowSpan"/>
              </xsl:attribute>
              Measurement value
            </th>
          </xsl:if>
        </tr>
      </xsl:for-each>
    </thead>
  </xsl:template>

  <xsl:template match="tbody">
    <tbody>
      <xsl:for-each select="tr">
        <xsl:variable name="currentTr" select="current()"/>
        <tr>
          <xsl:variable name="maxColSpan" select="count(td)"/>
          <xsl:for-each select="td">
            <td>
              <xsl:attribute name="rowspan">
                <xsl:value-of select="@rowspan"/>
              </xsl:attribute>
              <xsl:attribute name="colspan">
                <xsl:value-of select="@colspan"/>
              </xsl:attribute>
              <xsl:choose>
                <xsl:when test="specitem">
                  <xsl:apply-templates select="specitem"/>
                </xsl:when>
                <xsl:otherwise>
                  <xsl:value-of select="." />
                </xsl:otherwise>
              </xsl:choose>
            </td>
          </xsl:for-each>
          <td>
            <specitem>
              <xsl:attribute name="name">
                <xsl:value-of select="$currentTr/td/specitem/@name"/>
              </xsl:attribute>
              <xsl:attribute name="parameters">
                <xsl:for-each select="$currentTr/td/specitem">
                  <xsl:if test="position() = 1">
                    <xsl:for-each select="specParameterValue">
                      <xsl:value-of select="@type"/>=<xsl:value-of select="@value"/>
                      <xsl:if test="not(position()=last())">
                        <xsl:text>,</xsl:text>
                      </xsl:if>
                    </xsl:for-each>
                  </xsl:if>
                </xsl:for-each>
              </xsl:attribute>
            </specitem>
          </td>
        </tr>
      </xsl:for-each>
    </tbody>
  </xsl:template>
</xsl:stylesheet>
```