TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Tutkintotyö

Jani Levonen

TESTAUSJÄRJESTELMÄN KEHITTÄMINEN JAVA-POHJAISELLE OHJELMISTOKEHYKSELLE

Työn tilaaja Convia Oy, Managing Director Perttu Korpela
Tampere 2011

TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Tekijä: Jani Levonen
Työn nimi: Testausjärjestelmän kehittäminen Java-pohjaiselle ohjelmistokehykselle
Sivumäärä: 39 sivua + 7 liitesivua
Valmistumisaika: Tampere 2011
Työn tilaaja: Convia Oy, Managing Director Perttu Korpela

TIIVISTELMÄ

Tämän työn tarkoituksena oli perehtyä Convia Oy:n kehittämän Java-pohjaisen ohjelmistokehyksen rakenteeseen ja siihen, miten sen testausprosessia voitaisiin kehittää nyt ja tulevaisuudessa. Ohjelmistokehyksen testaamisen lisäksi työssä perehdyttiin ohjelmistokehystä ohjaavien XML-skriptien rakennemäärittelyyn ja validointiin, sekä testauksen ohjaamaan sovelluskehitykseen.

Toimivan ja monipuolisen ohjelmistokehyksen hyödyntäminen voi osaltaan tehostaa yrityksen sovelluskehitysprosessia, sekä merkittävästi lisätä koodin uudelleenkäytettävyyttä. Tästä syystä ohjelmistokehysten käyttö sovelluskehityksessä on kasvattanut suosiotaan tasaisesti viime aikoina. Jatkuvasti kehittyvän ohjelmistokehyksen toimivuuden varmistaminen vaatii kuitenkin vahvaa panostamista myös itse ohjelmistokehyksen ja sen avulla luotujen sovellusten testaamiseen.

Convian ohjelmistokehys on tarkoitettu lisäarvosovellusten luomiseen PTC:n ohjelmistotuotteisiin. Sitä on hyödynnetty ja hyödynnetään monenlaisissa projekteissa, mikä aiheuttaa omat haasteensa testaukselle. Tästä syystä työhön on sisällytetty myös XML-skriptien testaus- ja validointimahdollisuuksien tutkimista.

Työssä käsitellyt testausmenetelmät todettiin soveltuviksi Convian ohjelmistokehyksen kattavaan testaamiseen, laadun parantamiseen sekä jatkokehityksen ja ylläpidon tehostamiseen.

Avainsanat Java, ohjelmistokehys, testaus, XML, J-Link

TAMPERE UNIVERSITY OF APPLIED SCIENCES
Computer Systems Engineering
Software Engineering

Writer:	Jani Levonen
Thesis:	Developing a Testing System for a Java-Based Application Framework
Pages:	39 pages + 7 appendices
Graduation Time:	Tampere 2011
Co-operating Company:	Convia Oy, Managing Director Perttu Korpela

ABSTRACT

The main purpose of this thesis was to get acquainted with the structure of the Java-based application framework developed by Convia Oy and to consider how the testing process affiliated with the application framework could be improved. In addition to the testing of the application framework, the thesis also concentrates on the use of markup declarations of the XML documents and on their validation by using the document type definition.

Using an application framework can significantly streamline the software development process in a company and also increase the reusability of the code. This is one of the main reasons why the usage of application frameworks has become more popular in the software development industry. However, the use of an application framework also increases the importance of a well-organized testing process, so that the quality of the applications can be verified.

Convia uses its application framework to develop applications which aim at enhancing the usability of the products of PTC. Convia has developed a vast variety of different applications for different customers, which for its part makes it even more important to develop the testing process in the company.

The testing methods introduced in this thesis were found suitable for the testing, quality assurance and maintenance of the application framework and applications of Convia Oy.

Keywords Java, framework, testing, XML, J-Link

ESIPUHE

Tämän opinnäytetyön aihetta ehdotettiin minulle jo syksyllä 2010 työskennellessäni osa-aikaisena ohjelmistosuunnittelijaharjoittelijana Convia Oy:ssä. Työn aloittaminen siirtyi kuitenkin alkukevääseen 2011 asti, sillä opintojen ja työnteon yhteensovittaminen vaati paljon aikaa ja energiaa. Työ valmistui toukokuussa 2011, samoihin aikoihin kun minut palkattiin Conviaan vakituiseksi ohjelmistoasiantuntijaksi.

Haluan kiittää Sami Toivaista aiheeseen liittyvistä hyvistä neuvoista, Convian muuta henkilöstöä hyvästä työilmapiiristä, Erkki Hietalahtea asiantuntevasta ohjauksesta ja opetuksesta, sekä tietysti myös muita Tampereen ammattikorkeakoulun ohjelmistotekniikan opettajia laadukkaasta opetuksesta. Erityiskiitos kuuluu myös puolisolleni Maijalle, jonka vankkumattomalla tuella sain opintoni kunnialla päätökseen.

Tampereella toukokuussa 2011

Jani Levonen

SISÄLLYSLUETTELO

1 JOHDANTO 8
2 CREO ELEMENTS/PRO 3D CAD-OHJELMISTO 10
2.1 Tietokoneavusteinen suunnittelu 11
2.2 J-Link-ohjelmointirajapinta 12
2.2.1 Synkroniset J-Link-sovellukset 12
2.2.2 Asynkroniset J-Link-sovellukset 14
2.2.3 Convian J-Link-lisäarvosovellukset 14
3 OHJELMISTOKEHYS JA SEN TOIMINTA 16
3.1 Convian Java-pohjainen ohjelmistokehys 17
3.1.1 Sovelluksen luominen ohjelmistokehyksen avulla 17
3.1.2 Lisäarvosovellusten käynnistäminen 19
3.2 XML-skriptit 21
3.2.1 XML-skriptien rakenne 21
3.2.2 XML-skriptien tulkkaaminen Javassa 22
4 OHJELMISTON TESTAAMINEN 23
4.1 Testausmenetelmät 24
4.1.1 Mustalaatikkotestaus 24
4.1.2 Lasilaatikkotestaus 25
4.1.3 Regressiotestaus 26
4.2 Yksikkötestaus 27
4.2.1 JUnit-sovelluskehyksen hyödyntäminen 28
4.2.2 Testien ohjelmointi JUnitilla 28
5 XML-SKRIPTIEN RAKENNEMÄÄRITTELY 32
5.1 DTD 32
5.1.1 DTD-määrittelyn rakenne 33
5.1.2 XML-skriptien validointi DTD:n avulla 34

5.2 Rakennemäärittelyn toteuttaminen....35
6 TESTAUKSEN TOTEUTTAMISSUUNNITELMA....36
6.1 Yksikkötestauksen suorittaminen ohjelmistokehyksessä....36
6.2 XML-testiskriptien laatiminen ja ajaminen....36
6.3 TDD:n hyödyntäminen sovelluskehityksessä....37
7 YHTEENVETO....38
LÄHDELUETTELO....39
LIITTEET....40
Liite 1: Esimerkkisovellus....40
Liite 2: JUnit, Assert-luokan metodit....41
Liite 3: Näyte XML-testiskriptistä....45

ERITYISSANASTO JA LYHENTEET

Java	Laitteistoriippumaton oliopohjainen ohjelmointikieli
Ohjelmistokehys	Ohjelmistotuote, joka muodostaa rungon tietokoneohjelmalle
JVM	Java-virtuaalikone
XML	Rakenteellinen kuvaus- ja merkintäkieli
XSLT	Merkintäkieli XML-tiedostojen muunnoksiin
API	Ohjelmointirajapinta
CAD	Tietokoneavusteinen suunnittelu
PDM	Tuotetiedon hallinta
PLM	Tuotteen elinkaaren hallinta
JNI	Java-rajapinta tiedonvälitykseen natiivin koodin kanssa
RPC	Proseduurien etäkutsu
DTD	SMGL- ja XML-kielten rakennemäärittelytapa
TDD	Testivetoinen kehitys
XP	Ketterän ohjelmistokehityksen eräs metodologia
MySQL	SQL-tietokannan hallintajärjestelmä
JDBC	Java-ohjelmointikielen rajapinta tietokannan käsittelyyn
IDE	Integroitu ohjelmointiympäristö

1 JOHDANTO

Työ on tehty toimeksiantona Convia Oy:lle, jonka kehittämän Java-pohjaisen ohjelmistokehyksen kattavaan testaamiseen ei ole aiemmin ollut olemassa erillistä testausjärjestelmää.

Convia Oy on amerikkalaisen Parametric Technology Corporationin (PTC) ohjelmistotuotteiden edustaja Suomessa. Convian edustamat tuotteet jakautuvat kolmeen eri kategoriaan; tuotetiedon tuottamiseen, jakamiseen ja kontrollointiin. Convia edustaa mm. seuraavia PTC:n tuotteita: Creo Elements/Pro (ent. Pro/ENGINEER Wildfire), Windchill PDMLink, Windchill ProductPoint ja Mathcad. Lisäksi yritys kehittää erilaisia maksullisia ja maksuttomia lisäarvosovelluksia PTC:n tuotteisiin. Merkittävin osa Convian omista lisäarvosovelluksista on tarkoitettu Creo Elements/Pro 3D CAD-ohjelmalle ja niiden tarkoitus on entisestään tehostaa sen käyttöä. Näiden lisäarvosovellusten kehittämisessä on hyödynnetty Creo Elements/Pro:sta löytyviä Web.Link- ja J-Link-ohjelmointirajapintoja.

Creo Elements/Pro:n J-Link-ohjelmointirajapinta mahdollistaa erilaisten Java-pohjaisten lisäsovellusten kehittämisen ohjelmaan. Convia Oy on tehnyt näiden lisäsovellusten kehittämisen helpottamiseksi oman Java-ohjelmistokehyksen, jonka avulla monipuolisten lisäarvosovellusten tekeminen on tehokasta ja nopeaa. Lisäarvosovellukset luodaan Javalla, mutta ohjelmistokehystä ohjaillaan XML-tiedostoilla. Ohjelmistokehys tulkkaa sille annetut XML-tiedostot, ja luo niiden perusteella halutunlaisen lisäarvosovelluksen. Koska ohjelmistokehystä kehitetään jatkuvasti eteenpäin, järjestelmän testaamiseen on myös panostettava enenevissä määrin. Tämä työ on tehty osana ohjelmistokehyksen tuotekehitysprosessia.

Työn lähtökohtana on selvittää, mikä olisi paras tapa testata Convian Java-pohjaista ohjelmistokehystä. Koska ohjelmistokehystä kehitetään jatkuvasti eteenpäin, on tärkeää löytää tehokas ja luotettava tapa suorittaa regressiotestaus ohjelmistokehykselle aina muutosten jälkeen. Näin voidaan varmistua siitä, että muutosten jälkeen kaikki ohjelmistokehyksen vanhatkin osat toimivat uusien ohella. Työn tarkoituksena on löytää Convian ohjelmistokehykselle parhaiten sopiva testaustapa ja hahmotella sen pohjalta

toimiva testausjärjestelmä kattavan regressiotestauksen järjestämiseksi. Lisäksi työssä perehdytään XML-tiedostojen validointiin.

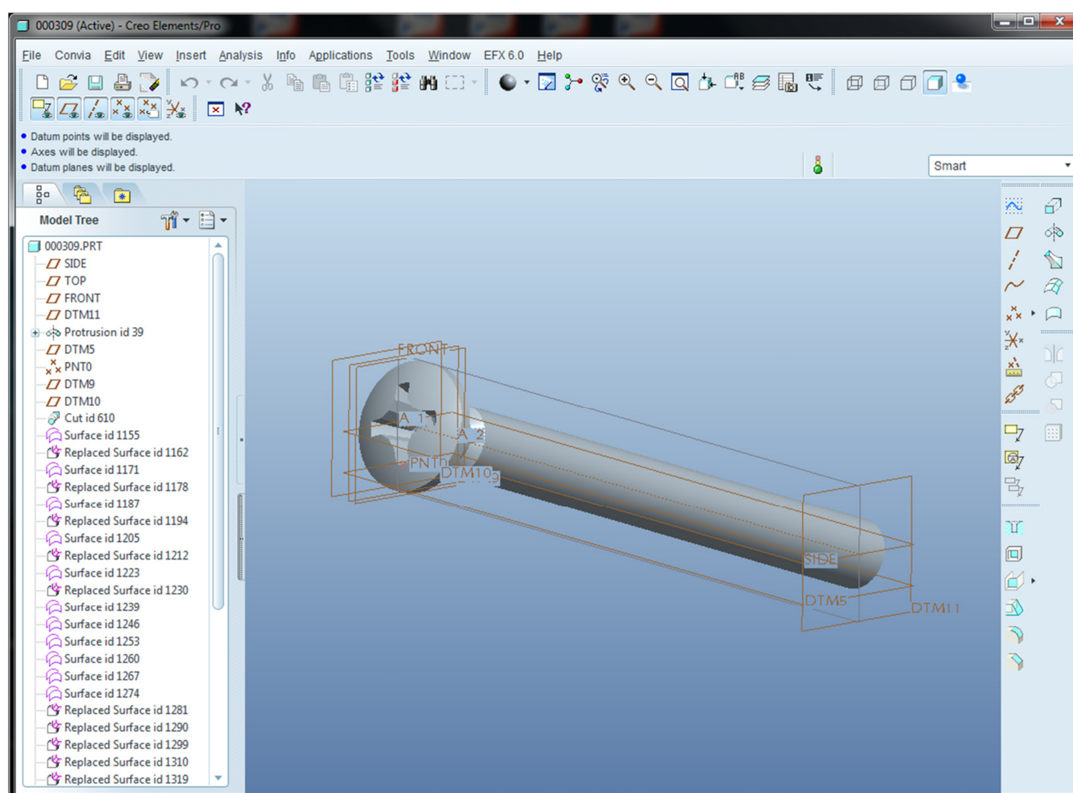
Luvussa 2 käydään läpi PTC:n Creo Elements/Pro 3D CAD-ohjelmistoa ja sen J-Link-ohjelmointirajapintaa. Luvussa 3 kerrotaan, mitä tarkoittaa ohjelmistokehys ja selvennetään, miten Convian kehittämä ohjelmistokehys pääpiirteittäin toimii. Luvussa 4 keskitytään sovellusten testaamiseen yleisellä tasolla ja kuvaillaan eri testausmenetelmiä. Luku 5 sisältää kuvauksen siitä, mitä tarkoittaa XML-tiedostojen rakennemäärittely, ja miten sitä voidaan hyödyntää Convian lisäarvosovellusten laadun parantamisessa.

Luvussa 6 kuvataan lyhyesti, miten testausta lähdetään viemään eteenpäin Convian tuotekehitysprosessissa.

Tässä työssä on pyritty kuvaamaan Convian kehittämän Java-pohjaisen ohjelmistokehyksen ja lisäarvosovellusten toiminnallisuutta, rakennetta ja tekniikoita niin, että lukija saa yleiskäsityksen niiden toiminnasta.

2 CREO ELEMENTS/PRO 3D CAD-OHJELMISTO

Creo Elements/Pro on yksi maailman suosituimmista 3D CAD-suunnitteluohjelmistoista. Ohjelmisto tunnettiin aiemmin nimellä Pro/ENGINEER Wildfire, mutta vuonna 2011 PTC päätti uudistaa ohjelman nimen. Ohjelmisto on assosiatiivinen ja parametrinen, minkä tarkoituksena on omalta osaltaan tehostaa suunnittelutyötä. Creo Elements/Pro:n monipuolisuus ja muokattavuus mahdollistavat sen, että siitä voivat hyötyä niin pienet kuin isotkin omaa tuotekehitystä omaavat yritykset. Oheisessa kuvassa on esimerkki ohjelmiston käyttöliittymästä (kuvio 1).



Kuvio 1: Creo Elements/Pro:n päänäkymä

PTC:n 3D CAD-ohjelmistosta on vuosien mittaan julkaistu useita eri versioita. Viimeinen Pro/ENGINEER nimellä julkaistu ohjelmisto oli versionumeroltaan 5.0 ja tämä numero siirtyi myös Creo Elements/Pro:n versioksi. Convian lisäarvosovellukset on pyritty toteuttamaan siten, että ne toimivat kaikilla yleisesti käytössä olevilla versioilla PTC:n ohjelmistoista. Koska Creo Elements/Pro on tuorein versio PTC:n 3D CAD-ohjelmistosta, tässä työssä keskitytään juuri kyseiseen versioon.

2.1 Tietokoneavusteinen suunnittelu

Tietokoneavusteinen suunnittelu eli CAD (Computer Aided Design) tarkoittaa erityisesti insinöörien ja arkkitehtien harjoittamaa suunnittelutyötä, jossa käytetään apuna tietokonetta. CAD-ohjelmia on ollut markkinoilla jo useita vuosia. Alun perin tietokoneavusteinen suunnittelu keskittyi 2D-kuvien tekemiseen, mutta nykyisin markkinoilta löytyy useita 3D-suunnitteluun keskittyviä ohjelmia. 3D CAD mahdollistaa kolmiulotteisten mallien tekemisen suunniteltavista tuotteista. Näitä malleja on mahdollista vapaasti käännellä ja tarkastella eri suunnista. 3D CAD-ohjelmistoilla on mahdollista suunnitella myös monimutkaisia mekanismeja, suorittaa lujuuslaskelmia yms.

Käytännössä 3D CAD:n avulla on nykyisin mahdollista tehdä toimivia prototyyppejä, mikä omalta osaltaan auttaa yrityksiä säästämään materiaali- ja työkustannuksissa merkittäviä summia. Lisäksi 3D CAD-ohjelmistot nopeuttavat tuotekehitysprosessia huomattavasti.

Monet 3D CAD-ohjelmistoista on myös mahdollista integroida suoraan saatavilla oleviin tuotetiedon hallinta- ja tuotannonohjausjärjestelmiin. Creo Elements/Pro voidaan liittää saumattomasti esimerkiksi PTC:n ja Microsoftin yhdessä kehittämään Windchill ProductPoint 2.0 ratkaisuun. Windchill ProductPoint 2.0 on tuotetiedon hallintaan (PDM) ja tuotteen elinkaaren hallintaan (PLM) tarkoitettu ratkaisu. PDM ja PLM ratkaisut helpottavat esimerkiksi 3D CAD-datan hallintaa merkittävästi, sillä sen hallinnointi esimerkiksi verkkolevyillä on melko työlästä ja jatkuvaa ylläpitoa vaativaa. Kuten 3D CAD-ohjelmistoja, myös erilaisia PDM ja PLM ratkaisuja on markkinoilla useita erilaisia.

Koska eri asiakkailla on erilaisia tarpeita 3D CAD-ohjelmiston suhteen, Creo Elements/Pro sisältää erilaisia ohjelmointirajapintoja, joiden avulla on mahdollista räätälöidä sovelluksesta mahdollisimman hyvin asiakkaan tarpeisiin sopiva. Convia käyttää omissa maksullisissa lisäarvosovelluksissaan Javalle tarkoitettua J-Link-ohjelmointirajapintaa.

2.2 J-Link-ohjelmointirajapinta

J-Link on PTC:n Creo Elements/Pro 3D CAD-ohjelmalle kehitetty ohjelmointirajapinta eli API (Application Programming Interface), jonka avulla ulkoiset Javalla kehitetyt ohjelmat pääsevät käsiksi Creo Elements/Pro:n sisäisiin komponentteihin. J-Link on rakennettu Creo Elements/Pro:n Toolkit API:n päälle. Sen avulla voidaan sekä laajentaa, kustomoida että automatisoida Creo Elements/Pro:n toimintaa. J-Link-rajapintaa käyttävät Java-sovellukset voivat lisäksi muokata Creo Elements/Pro:lla luotuja 3D-malleja.

Creo Elements/Pro:n mukana toimitetaan kattavat ohjeet omien J-Link-sovellusten tekemiseen. Ohjelmiston asennuspaketista löytyy myös paljon J-Link-esimerkkisovelluksia.

J-Link mahdollistaa sekä synkronisten, että asynkronisten sovellusten tekemisen. Molemmissa sovellustyypeissä on omat etunsa ja haittansa. Seuraavassa on kerrottu, kuinka nämä kaksi eri sovellustyyppiä eroavat toisistaan pääpiirteittäin.

2.2.1 Synkroniset J-Link-sovellukset

Synkroniset J-Link sovellukset käynnistetään Creo Elements/Pro:n toimesta ja se myös hallinnoi sovellusta. Mikäli synkroninen J-Link sovellus on käynnissä, se varaa järjestelmän resurssit käyttöönsä, mikä tarkoittaa käytännössä sitä, ettei Creo Elements/Pro:n toimintoja voida käyttää samaan aikaan sovelluksen ollessa käynnissä. Tämä asettaa tiettyjä rajoituksia sovellukselle, eli esimerkiksi avoinna olevaa mallia ei ole mahdollista liikutella tai muokata itse Creo Elements/Pro:ta käyttäen sovelluksen ollessa käynnissä. Ulkoisen J-Link-sovelluksen on silti mahdollista päästä kattavasti käsiksi pääohjelman komponentteihin ja toimintoihin. Tämä tarkoittaa käytännössä sitä, että esimerkiksi avattua 3D-mallia ja sen sisältämiä tietoja voidaan muokata ulkoisen sovelluksen toimesta.

J-Link-sovellusten käynnistäminen Creo Elements/Pro:ssa vaatii ohjelmiston protk.dat-tiedoston muokkaamista. Kyseessä on rekisteritiedosto, jossa määritetään Creo

Elements/Pro:n kanssa käytettävät J-Link-sovellukset. Seuraavassa on esitetty, mitä kyseinen rekisteritiedosto voi sisältää (esimerkki 1).

```
name                    java_app
startup                 java
java_app_class          JavaApp
java_app_start          start
java_app_stop           stop
java_app_classpath      C:\example\java_app.jar
allow_stop              true
delay_start             false
text_dir                C:\temp\text
end
```

Esimerkki 1: Protk.dat-tiedoston sisältö (Parametric Technology Corporation, 2010)

J-link-sovelluksen ajamiseen liittyvät määritykset tehdään siis kyseisessä tiedostossa. Seuraavassa on lyhyesti selitetty kenttien sisällön merkitys.

- **name** = Sovelluksen nimi. Tarvitaan erityisesti silloin, kun sovelluksia on useampia.
- **startup** = Asetus määrittää, millä tavoin Creo Elements/Pro kommunikoi sovelluksen kanssa. J-Link-sovellusten tapauksessa asetuksen arvona tulee olla Java.
- **java_app_class** = Java-sovelluksen pääluokan nimi.
- **java_app_start** = Java-sovelluksen käynnistysmetodin nimi. Käynnistysmetodi tulee löytyä asetetusta sovelluksen pääluokasta.
- **java_app_stop** = Java-sovelluksen pysäytysmetodin nimi. Myös pysäytysmetodi tulee löytyä asetetusta sovelluksen pääluokasta.
- **java_app_classpath** = Java-sovelluksen luokkien ja jar-pakettien sijainti.
- **allow_stop** = Salli tai estä sovelluksen pysäyttäminen (true/false).
- **delay_start** = Salli tai estä sovelluksen käynnistämisen hallinta (true/false).
- **text_dir** = Viesti- ja valikkotekstit sisältävien tiedostojen sijainti.
- **end** = J-Link-sovelluksen kuvauksen lopetus. Samassa tiedostossa voi olla useiden eri J-Link-sovellusten määritykset, joten ne erotetaan toisistaan lopetusmerkinnällä.

Kun oheiset asetukset on tehty jokaista J-Link-sovellusta varten, ne ovat valmiita käytettäväksi Creo Elements/Pro:n kanssa. Sovelluksen käynnistystapaan vaikuttaa se, mitä delay_start-kentän asetukseksi on asetettu. Mikäli asetuksen arvona on true, ohjelma pitää käynnistää Creo Elements/Pro:n valikosta, muuten ohjelma käynnistyy automaattisesti Creo Elements/Pro:n mukana.

Synkroniset J-Link-sovellukset käyttävät oletuksena Creo Elements/Pro:n mukana toimitettavaa Java-virtuaalikonetta (JVM). Käytettävä JVM on kuitenkin mahdollista vaihtaa myös muuhun kuin mukana tulevaan versioon Creo Elements/Pro:n konfiguraatio-optiolla jlink_java_command. Lisäksi jlink_java_command-optiolla voidaan määritellä erilaisia käynnistysparametreja JVM:lle.

2.2.2 Asynkroniset J-Link-sovellukset

Asynkroniset J-Link sovellukset voidaan ajaa itsenäisesti ja niillä on mahdollista käynnistää sekä hallinnoida Creo Elements/Pro:n prosesseja. Tämä mahdollistaa siis käytännössä sen, että Creo Elements/Pro ja J-Link sovellus voivat suorittaa yhtäaikaisia toimintoja. Synkronisista sovelluksista poiketen asynkroniset J-Link sovellukset käyttävät JNI:tä (Java Native Interface) ja RPC-kutsuja Creo Elements/Pro:n ja J-Link sovelluksen väliseen tiedonsiirtoon.

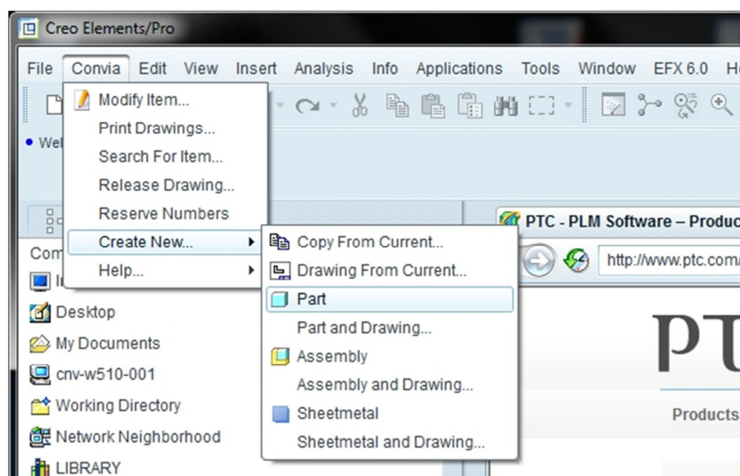
Asynkroniset sovellukset voidaan käynnistää ilman Creo Elements/Pro:ta, mutta johtuen proseduurien etäkutsujen eli RPC-kutsujen käytöstä asynkroniset sovellukset ovat kuitenkin huomattavasti hitaampia kuin synkroniset sovellukset. Asynkroniset J-Link-sovellukset hallitsevat Creo Elements/Pro:ta täysin, joten niiden avulla on mahdollista automatisoida ohjelmiston prosesseja. Tällaisilla sovelluksilla voidaan esimerkiksi toteuttaa automatisoidusti halutunlaiset muutokset suureen määrään 3D-malleja.

2.2.3 Convian J-Link-lisäarvosovellukset

Convian ohjelmistokehyksen avulla on mahdollista toteuttaa sekä synkronisia että asynkronisia sovelluksia. Yrityksen päätuotteet käyttävät synkronista lähestymistapaa, mutta asynkronisiakin sovelluksia on toimitettu asiakkaille. Nämä sovellukset ovat olleet pääasiassa erilaisia datan louhintaan tarkoitettuja lisäarvosovelluksia. Datan

louhimisella tarkoitetaan tässä yhteydessä esimerkiksi sellaista tilannetta, jossa suuresta määrästä erilaisia 3D-malleja on tarve hakea tietoa ja muokata niiden ominaisuuksia. Koska tällaisten toimintojen automatisoinnilla voidaan saavuttaa suuria ajallisia ja kustannuksellisia säästöjä, asynkronisen lähestymistavan käyttö on tullut kyseeseen.

Synkronisten J-Link-sovellusten käynnistäminen onnistuu Creo Elements/Pro:n valikosta. Convian J-Link-lisäarvosovellukset kuitenkin käynnistetään omasta valikostaan, joka lisätään Creo Elements/Pro:n sisälle. Oheisessa kuvassa (kuvio 2) on esimerkki siitä, minkälaisia toimintoja Convia-valikon alta voi lyötyä. Toiminnot vaihtelevat sen mukaan, minkälaisen paketin asiakas on tilannut.



Kuvio 2: Convian lisäarvosovellusvalikko Creo Elements/Pro:ssa

Koska Convian lisäarvosovelluksille luodaan yllä olevan kaltainen valikkonsa Creo Elements/Pro:n sisälle, loppukäyttäjän ei tarvitse huolehtia sovellusten käynnistämisestä ja sammuttamisesta itse. Näin lisäsovellukset integroituvat hyvin yhteen itse pääsovelluksen kanssa.

3 OHJELMISTOKEHYS JA SEN TOIMINTA

Ohjelmistokehys on käsitteenä melko yksiselitteinen ja helposti ymmärrettävä. Sillä tarkoitetaan ohjelmistorunkoa, jota täydentämällä on mahdollista luoda useita erilaisia sovelluksia. Ohjelmistokehyksen perimmäinen tarkoitus on tehostaa ja nopeuttaa sovellusten tekemistä. Koska Convian tavoitteena on toimittaa räätälöityjä lisäarvosovelluksia asiakkaille heidän tarpeidensa mukaan, oman oliopohjaisen ohjelmistokehyksen tekeminen on osoittautunut kannattavaksi.

> *"Olioperustainen ohjelmistokehys (object-oriented framework) on luokka-, komponentti- ja/tai rajapintakokoelma, joka toteuttaa jonkin ohjelmistojoukon yhteisen arkkitehtuurin ja perustoiminnallisuuden." (Koskimies & Mikkonen, 2005, 187.)*

Yllä oleva lainaus kuvaa ohjelmistokehyksiä yleensä ja myös Convian kehittämää oliopohjaista ohjelmistokehystä melko osuvasti. Suurimpina etuina ohjelmistokehyksen käytössä voidaan mainita uusien sovellusten kehittämisen nopeus, sekä lisäksi se, että kehykseen voidaan helposti tehdä uusia ominaisuuksia, jotka on mahdollista ottaa käyttöön myös vanhoissa samaa ohjelmistokehystä käyttävissä sovelluksissa. Lisäksi ohjelmistokehyksen avulla kaikista tehdyistä sovelluksista löytyvät samat perustoiminnot.

Ohjelmistokehyksen käytöstä löytyy yllämainittujen hyötyjen lisäksi myös omat haittapuolensa. Esimerkiksi ohjelmistokehyksestä löytyvät toiminnallisuudet on syytä dokumentoida kattavasti. Mikäli näin ei tehdä, ja ohjelmistokehykseen lisätään uusia ominaisuuksia jatkuvasti, on vaarana, että se paisuu liian suureksi ja hankalaksi käyttää. Muutenkaan ohjelmistokehyksen käyttö ei sovellu kaikkiin mahdollisiin tilanteisiin. Jos kehitettävät sovellukset eroavat suuresti toisistaan, eikä niissä ole samoja toimintoja, saattaa ohjelmistokehyksen käyttäminen jopa hidastaa ja muutenkin hankaloittaa sovelluskehitystä. Kehyksen käyttämistä on siis harkittava tarkkaan tilanteen mukaan.

Convian tapauksessa ohjelmistokehyksen käyttö on kuitenkin osoittautunut erittäin tarpeelliseksi ja sen hyödyt ovat kiistattomat. Convian lisäarvosovellukset toimivat

usein linkkinä 3D-mallien ja erilaisten tietokantojen välillä, joten usein on tarpeen kehittää erilaisia integrointeja järjestelmien välille tiedon siirtämiseksi. Näiden integraatioiden tekemisessä ohjelmistokehyksen käyttäminen on avuksi, sillä usein integraatiosovellukset halutaan ottaa käyttöön nopeasti. Kun sovellusten perusrunko on valmiina, integrointien implementointi onnistuu tarvittaessa hyvinkin nopeasti.

Ohjelmistokehyksen avulla Convia on voinut kehittää muutamia ns. perustuotteita, kuten Pro/PARAM- ja Pro/BOM-lisäarvosovellukset. Nämä sovellukset sisältävät paljon toimintoja, joita jokainen sovelluksen ostanut asiakas tarvitsee. Koska näissä sovelluksissa on käytetty pohjana samaa ohjelmistokehystä, niihin voidaan helposti ja nopeasti lisätä räätälöityjä ominaisuuksia ja toimintoja asiakkaan toiveiden mukaisesti. Ohjelmistokehyksen käyttämisen ansiosta myös sovellusten ylläpito on helppoa, sillä kehykseen tehdyt päivitykset voidaan tehdä kaikkien eri asiakkaiden versioihin ainoastaan korvaamalla ohjelmistokehyksen sisältävä jar-paketti.

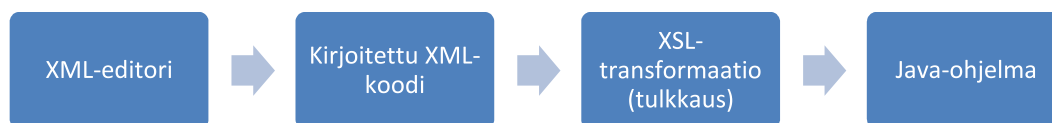
3.1 Convian Java-pohjainen ohjelmistokehys

Convian ohjelmistokehyksen ohjelmointikieleksi on valikoitunut Java muun muassa siitä syystä, että sovelluksissa voidaan hyödyntää Creo Elements/Pro:n J-Link-ohjelmointirajapintaa (kts. kappale 2.2). Lisäksi Java on käyttöjärjestelmäriippumaton kieli, koska sitä ajetaan omassa virtuaalikoneessaan. Tästä syystä sovellukset toimivat hyvin esimerkiksi Windowsin eri versioissa.

Lisäarvosovellusten käyttöliittymät on luotu käyttäen Javan Swing-kirjaston käyttöliittymäelementtejä. Monia näistä elementeistä on modifioitu vastaamaan paremmin sovellusten tarpeita.

3.1.1 Sovelluksen luominen ohjelmistokehyksen avulla

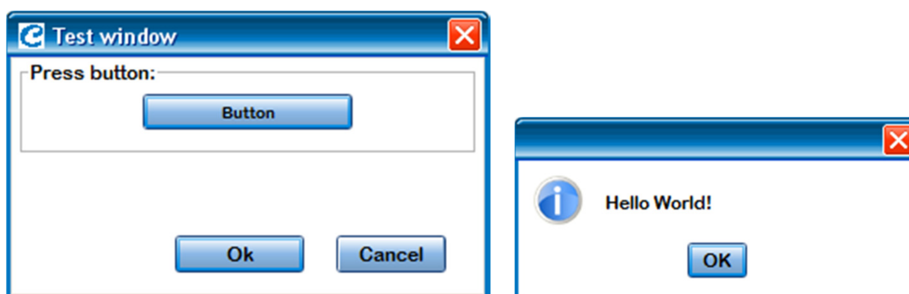
Ohjelmistokehys itsessään ei ole ajokelpoinen sovellus, vaan sillä luodaan halutunlaisia sovelluksia ohjaamalla sen toimintaa XML-tiedostojen avulla. Ohjelmistokehys käyttää XSL-transformaatiota tulkatessaan lukemansa XML-tiedostot. Koko prosessia voidaan kuvata seuraavan kaaviokuvan avulla (kuvio 3).



Kuvio 3: XML-tiedostosta Java-sovellukseksi

XML-koodin kirjoittaminen on melko nopeaa ja helppoa. Koska XML-kieli on rakenteinen, koodista tulee lisäksi selkeää ja helppolukuista. Liitteenä oleva lyhyt esimerkki selventää hieman Convian ohjelmistokehykselle tehdyn XML-koodin rakennetta (liite 1).

Kyseinen esimerkkiohjelma luo alla olevan kuvan (kuvio 4) mukaisen Java-ikkunan, jossa on Button-painike, sekä lisäksi Ok- ja Cancel-painikkeet. Button-painiketta painamalla aukeaa uusi viesti-ikkuna, jossa on teksti ”Hello World!” ja Ok-painike. Esimerkin on tarkoitus havainnollistaa sitä, miten XML-koodi rakentuu, ja lisäksi demonstroida sovellusten luomista Convian ohjelmistokehyksen avulla. XML-koodin rakennetta ja ominaisuuksia kuvataan tarkemmin kappaleessa 3.2.

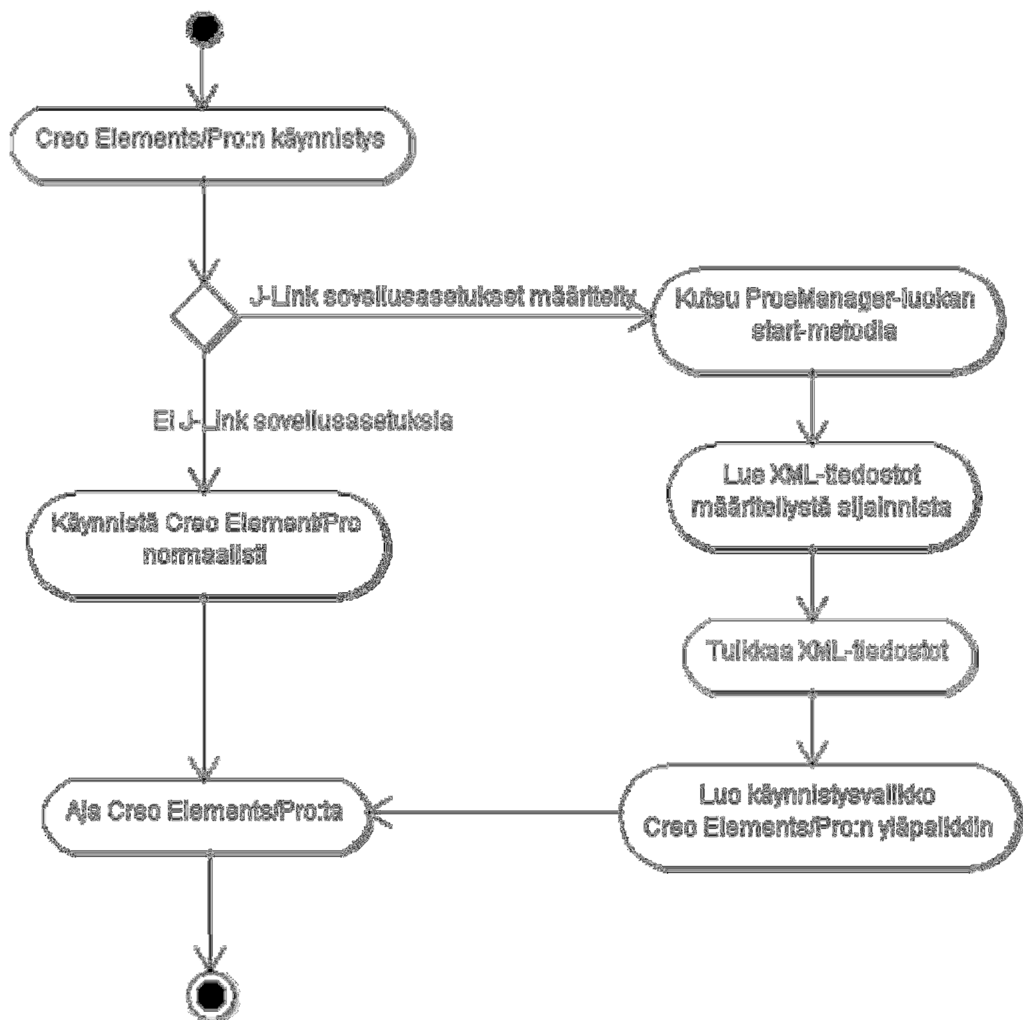


Kuvio 4: ”Hello World”-esimerkkiohjelma

Yllä olevan kaltaisen sovelluksen tekeminen on todella nopeaa Convian ohjelmistokehyksen avulla. Tosin eri komentojen, syntaksin ja toimintojen opetteleminen vie oman aikansa. Koska eri asiakkaille toimitettavat sovellukset vastaavat suurilta osin toisiaan, koodin uudelleenkäytettävyys on omaa luokkaansa. Lisäksi, kun ohjelman ulkoasu on kerran toteutettu, voidaan kehitykseen kuluva työaika kohdentaa huomattavasti tehokkaammin varsinaisten toimintojen hiomiseen.

3.1.2 Lisäarvosovellusten käynnistäminen

Kun tarvittavat asetukset on tehty Creo Elements/Pro:n protk.dat-tiedostoon (kts. kappale 2.2.1), lisäarvosovellukset käynnistyvät pääohjelman rinnalla. Seuraava yksinkertaistettu aktiviteettikaavio selventää Creo Elements/Pro:n käynnistysprosessia sekä ilman lisäarvosovelluksia, että niiden kanssa (kuvio 5).



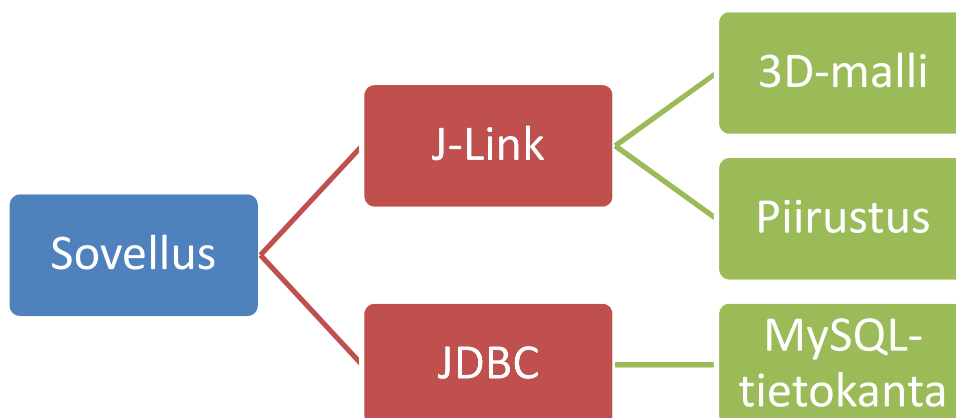
Kuvio 5: Creo Elements/Pro:n käynnistäminen

Convian ohjelmistokehyksen pääluokka on nimeltään ProeManager. Tämä luokka sisältää mm. sovellusten käynnistämiseen ja sammuttamiseen tarvittavat start- ja stop-

metodit. Kun Creo Elements/Pro käynnistetään Convian asetuksilla, se kutsuu ensimmäisenä ProeManager-luokan start-metodia. Kun ohjelmistokehyksen start-metodia kutsutaan, se lukee aluksi määritellystä sijainnista joukon XML-tiedostoja, jotka sitten tulkataan ohjelmistokehyksen toimesta. Tulkattujen XML-tiedostojen pohjalta luodaan varsinaiset sovellukset.

Convian sovelluksia varten rakennetaan Creo Elements/Pro:n yläpalkkiin oma valikkonsa, josta eri sovellukset voidaan avata. Kuten edellä mainittiin, XML-tiedostot on luettu muistiin ohjelman käynnistyessä, mutta sovellusten käyttöliittymät rakennetaan varsinaisesti vasta niiden avaamisen yhteydessä.

Useimpiin Convian lisäarvosovelluksiin liittyy myös tietokantojen käsittelyä. Tarvittavat kyselyt ja datan lataaminen suoritetaan useimmissa tapauksissa jo Creo Elements/Pro:n käynnistymisen yhteydessä sovellusten nopeuden parantamiseksi. Tietokantojen päivittyessä sovellusten käyttämä data voidaan kuitenkin tarvittaessa hakea uudelleen tietokannasta. Seuraava kaavio (kuvio 6) kuvaa tarkemmin sitä, minkälainen arkkitehtuuri sovelluksen, tietokannan ja Creo Elements/Pro:n välillä vallitsee.



Kuvio 6: Sovelluksen, tietokannan ja Creo Elements/Pro:n välinen arkkitehtuuri

3.2 XML-skriptit

XML on rakenteellinen merkintäkieli, jota käytetään tiedon välittämiseen järjestelmien välillä sekä dokumenttien tallennusformaattina. Sen avulla on helppo hahmottaa laajojakin tietomääriä, sille se on rakenteellinen kuvauskieli. Koska XML-kielellä voidaan kuvata tiedon rakennetta ilman ennalta määrättyjä koodeja, sitä voidaan hyödyntää uusien koodien luomisessa. (Wikipedia: XML, 2011.)

Convian lisäarvosovellusten tekemiseen XML-kieli on valikoitunut juuri sen muokattavuuden vuoksi. XML:n avulla on voitu kehittää jatkuvasti kehittyvä ja nopea tapa luoda sovelluksia ja ohjata ohjelmistokehyksen toimintoja.

3.2.1 XML-skriptien rakenne

XML-tiedostojen rakenne on tarkkaan määritelty, minkä johdosta Convian sovellusten rakenne on selkeä ja niiden implementointi nopeaa. Kieli rakentuu erilaisista tageista, joilla ikään kuin merkitään eri elementit XML-dokumentin sisällä. Tagit helpottavat merkittävästi erilaisten käyttöliittymäkomponenttien luomista sovelluksiin. Esimerkiksi Java-ikkunan luominen onnistuu yksinkertaisesti form-tagin avulla (kts. esimerkki 2). Esimerkkitagin sisällä on myös määritelty erilaisia attribuutteja luotavalle käyttöliittymäkomponentille. Näiden attribuuttien avulla voidaan komponentille määritellä monenlaisia ominaisuuksia, kuten leveys, korkeus, nimi tms. Attribuutit vaihtelevat eri komponenttien mukaan.

```
<form
  width="480" height="770"
  name="testi_form" title="Testi"
  resize="false" centered="true"
  onload_action="" action="">
</form>
```

Esimerkki 2: Java-ikkunan luonti käyttäen XML-kieltä ja Convian ohjelmistokehystä

Esimerkiksi form-elementin tapauksessa eri elementtejä voidaan asettaa sisäkkäin. Jos esimerkiksi halutaan luoda kehys (frame) ikkunan sisälle, voidaan kehyksen tagi sijoittaa form-elementin alku- ja lopputagien sisäpuolelle. Tällöin kehys sijoitetaan

ikkunan sisään Javan toimesta sovellusta luotaessa. Tämä lisää huomattavasti koodin luettavuutta ja käyttöliittymien hahmottamista. Erilaisia elementtejä on paljon ja niiden dokumentointiin on panostettava, jotta kaikkien olemassa olevien ominaisuuksien käyttö on mahdollisimman tehokasta.

XML-kieli on erittäin siirrettävää, sekä lisäksi sen yhteiskäytettävyys on suuri etu muihin ohjelmointikieliin verrattuna. Siirrettävyydellä viitataan tässä siihen, että XML-kieli on periaatteessa yksinkertaisesti tekstiä, joten se voidaan siirtää helposti alustalta toiselle. Koska XML noudattaa aina samaa standardia, sitä tulkkaavat ohjelmat tietävät aina, miten tietoa on tulkittava. Yhteiskäytettävyydellä puolestaan viitataan siihen, että XML-kielessä ei ole rajoitettu sen käyttötarkoitusta oikeastaan millään tavalla. Ainoastaan dokumentin rakenne on rajoitettu, sen sisältö puolestaan ei. Tämä on selkeä vahvuus verrattaessa XML:ää muihin kieliin. (McLaughlin, 2002, 19-21.)

3.2.2 XML-skriptien tulkkaaminen Javassa

Jotta XML-tiedostoja voidaan hyödyntää sovelluksissa, tarvitaan XML-parseri. Parsiminen tarkoittaa XML-dokumentin sisältämän datan tulkkaamista raakamuodon perusteella manipuloitavaksi datarakenteeksi. Parseri on olennainen osa XML-tiedostojen käyttöä. (McLaughlin, 2002, 25.)

Convian sovelluksiin liittyvät XML-skriptit tulkataan Java-ohjelmistokehyksen toimesta. Ohjelmistokehys parsii määriteltyjen XML-tiedostojen sisällön ja rakentaa sovelluksen niiden perusteella. Lisäksi joissain lisäarvosovelluksissa XML-dokumentteja käytetään myös datan jatkojalostukseen, esimerkiksi erilaisten raporttien tekemiseen.

4 OHJELMISTON TESTAAMINEN

Testaaminen on erittäin oleellinen osa ohjelmistotuotantoprosessia, sillä se on ainoa tapa varmistaa, että ohjelmoitavat tuotteet ovat laadukkaita ja että ne tyydyttävät asiakkaiden tarpeet riittävän hyvin. Testaaminen on ollut pitkään se ohjelmistotuotannon osa-alue, jota yritykset ovat laiminlyöneet. Viime vuosien aikana on kuitenkin paremmin ymmärretty testaamisen tärkeys, ja siihen on alettu kiinnittää yhä enemmän huomiota. Testausmenetelmiä on kehitetty eteenpäin, jotta ohjelmistojen laatu saataisiin paremmaksi. Tähän on osaltaan vaikuttanut 1990-luvun puolivälissä syntyneiden ketterien menetelmien suosion kasvu. Erityisesti testauksen ohjaama ohjelmistokehitys, eli Test Driven Development (TDD) on vaikuttanut paljon testausmenetelmien kehittymiseen. (Harju & Juslin, 2006, 89.)

Testaamisen laiminlyöntiin vaikuttaa osaltaan ainakin se, ettei kovan kilpailun ja tiukkojen aikataulujen vuoksi yksinkertaisesti jää aikaa toteuttaa riittävän kattavaa ja tehokasta testausta tehdyille sovelluksille. Tästä johtuen valmiisiin ohjelmistotuotteisiin saattaa jäädä toimintaa haittaavia virheitä, joita joudutaan sitten korjaamaan jälkikäteen. Tämä on luonnollisesti tilanne, jota tulisi välttää. Ohjelmistotuotteiden laadun varmistamiseksi olisikin erittäin tärkeää panostaa ohjelmiston kattavaan testaamiseen jo ohjelmaa tehtäessä.

Ohjelmistokehyksen testaaminen on melko ongelmallista, johtuen pitkälti samasta syystä kuin tuoterunkojen testaamisen hankaluus yleensä: kombinaatioita, joiden kanssa kehyksen tulisi toimia, on olemassa lähes loputtomasti. Koska kaikkien erikoistuksien ja muunnelmakombinaatioiden testaaminen on lähes mahdoton tehtävä, ohjelmistokehys testataan yleensä vain muutamien esimerkkierikoistuksien kanssa. Näillä pyritään kattamaan mahdollisimman hyvin koko kehyksen koodi. (Koskimies & Mikkonen, 2005, 211.)

Testaamiseen toteuttamiseen eri ympäristöissä on kehitetty useita erilaisia tapoja, mutta seuraavassa on keskitytty niihin menetelmiin, joita tullaan hyödyntämään Convian sovellusten runkona käytetyn ohjelmistokehyksen testaamisessa.

4.1 Testausmenetelmät

Seuraavassa on kuvailtu testausmenetelmiä, joita tullaan hyödyntämään Convian ohjelmistokehityksessä. Eri testaustapoja yhdistelemällä on mahdollista parantaa yrityksen tuottamien sovellusten laatua ja tehostaa niiden kehitystä. Menetelmiksi on valittu ohjelmistokehyksen funktionaaliseen ja rakenteelliseen testaamiseen parhaiten sopivat testausmenetelmät.

4.1.1 Mustalaatikkotestaus

Mustalaatikkotestaus eli black box-testaus tarkoittaa sellaista testaamista, jossa testattavan koodin implementoinnista ei ole tarkkaa tietoa. Mustalaatikkotestaukseen kuuluu useita erilaisia testaamistapoja, kuten esimerkiksi satunnaisilla arvoilla testaaminen, raja-arvoanalyysi, mallipohjainen testaaminen ja määrittelyyn perustuva testaus. Mustalaatikkotestauksella tutkitaan siis vain ohjelman ulkoista toimintaa, eikä lainkaan sen toteutusta.

Koska mustalaatikkotestaus ei kata sovelluksen toteutuksen, eli lähdekoodin, testaamista, se soveltuu paremmin sovelluksen ulkoisten ominaisuuksien testaamiseen. Tämä voi tarkoittaa esimerkiksi funktionaalista testaamista, jolla pyritään selvittämään, minkälaisia syötteitä sovellus hyväksyy. Mustalaatikkotestauksella voidaan kartoittaa virhetilanteita, jotka syntyvät sovelluksen käytön tuloksena. Sovelluksen toteutuksesta tällaisten virhetilanteiden löytäminen voi olla hankalampaa, joissain tapauksissa jopa mahdotonta.

Convian ohjelmistokehyksen tapauksessa mustalaatikkotestausta voidaan suorittaa tekemällä erityisiä XML-testiskriptejä, joilla voidaan testata käyttöliittymäelementtien ja niihin liittyvien toimintojen toimivuutta ja käyttäytymistä erilaisilla syötteillä. Testiskriptien ajamista varten voidaan laatia erillinen sovellus, jossa on valmiina toteutettuna eri käyttöliittymäelementtejä. Testiskriptien ajamiseen tarkoitettua sovellusta voidaan käyttää joko testaajan toimesta, tai sille voidaan tehdä automatisoituja testausrutiineja.

Mustalaatikkotestaus on funktionaalista testaamista, joten sitä suoritetaan aina myös yksinkertaisesti käyttämällä luotuja sovelluksia. Convian lisäarvosovellusten testausprosessia tulisi viedä eteenpäin panostamalla yhä enenevissä määrin valmiin tuotteen testauttamiseen. Ennen tuotteen toimitusta tulisi siis varmistaa, että toimitettavan sovelluksen kaikki vaaditut toiminnot on toteutettu ja ne toimivat kuten spesifikaatiossa on määritelty. Tämä voidaan toteuttaa helposti antamalla sovellus testattavaksi usealle eri henkilölle yrityksen sisällä. Tuotetta testaavat henkilöt voisivat sitten raportoida tarkasti havainnoistaan ja mahdollisesti havaitsemistaan puutteista tai virheistä.

4.1.2 Lasilaatikkotestaus

Lasilaatikkotestauksessa keskitytään mustalaatikkotestauksesta poiketen sovelluksen toteutuksen testaamiseen. Sitä kutsutaan myös rakenteelliseksi testaamiseksi. Tällöin siis tunnetaan sovelluksen sisäinen rakenne, eli sen lähdekoodi. Lähdekoodin perusteella voidaan suunnitella tarvittavat testit, joilla koodin toimivuus on mahdollista varmistaa.

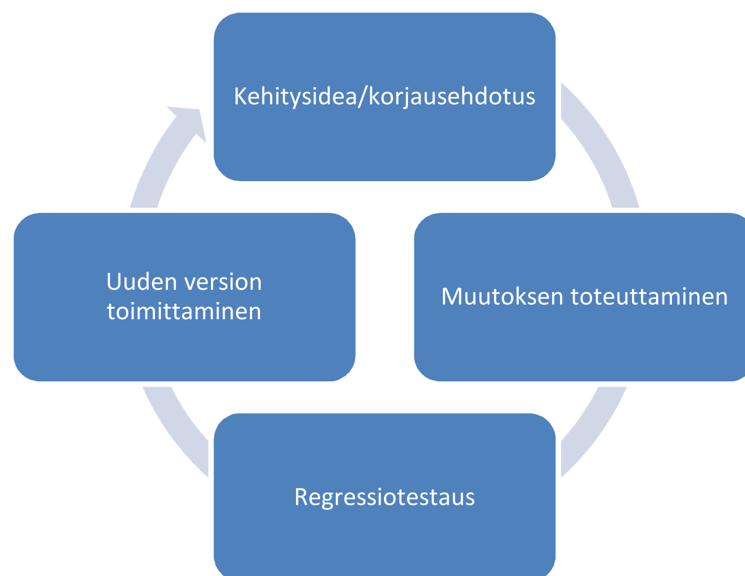
Rakenteellisen testaamisen perusajatuksena on keskittyä koodissa tapahtuvaan tiedon kulkuun ja käskyjen oikeellisuuteen. Sovelluksessa kontrolli vaihtuu ohjelman komponenttien välillä useilla eri tavoilla, kuten funktiokutsuilla, viestinvälityksen avulla sekä erilaisten keskeytyskäskyjen toimesta. Lisäksi erilaiset ehtolauseet vaikuttavat siihen, missä järjestyksessä eri käskyjä toteutetaan sovelluksen ollessa käynnissä. Rakenteellisen testaamisen avulla voidaan varmistua tiedon eheydestä ja käskyjen toimivuudesta. (Naik & Tripathy, 2008, 20-21.)

Rakenteellista- eli lasilaatikkotestausta tulee hyödyntää funktionaalisen testauksen rinnalla. Kumpikaan näistä testaustavoista ei vielä yksinään takaa ohjelman laatua ja oikeanlaista toimivuutta, mutta järkevästi molempia tapoja yhdistellen niistä on merkittävää hyötyä Convian ohjelmistokehyksen ja sillä toteutettujen sovellusten testaamisessa. Lasilaatikkotestauksessa tulee keskittyä ohjelmistokehyksen Java-toteutuksen testaamiseen, eli rakenteen testaamiseen koodista käsin.

4.1.3 Regressiotestaus

Regressiotestauksella tarkoitetaan sellaista testaamista, joka suoritetaan aina, kun sovelluksen, tai esimerkiksi ohjelmistokehyksen, jokin komponentti muuttuu. Regressiotestaamisen tarkoituksena on selvittää, toimiiko sovellus oikein tehtyjen muutosten jälkeenkin. Muutokset saattavat olla esimerkiksi pieniä korjauksia virheellisiin osioihin, tai vastaavasti osajärjestelmien lisäyksiä tai poistoja. Kaikkien muutosten jälkeen on tarpeellista testata kattavasti, etteivät muutokset ole aiheuttaneet konflikteja koodissa.

Regressiotestaus suoritetaan yleensä olemassa olevien yksikkötestien avulla. Koska Convian ohjelmistokehyksellä toteutetaan erilaisia sovelluksia eri asiakkaille ja siihen kehitetään uusia ominaisuuksia jatkuvasti, on tärkeää panostaa enemmän myös ohjelmistokehyksen regressiotestaukseen. Tällöin voidaan välttää tilanteita, jossa esimerkiksi koodiin tehty korjaus tai uusi ominaisuus aiheuttaa toimimattomuutta toisaalla koodissa. Kattavan regressiotestauksen avulla voidaan välttyä myös tilanteelta, jossa asiakkaalle toimitettu ohjelmisto sisältää toimimattomia ominaisuuksia. Seuraava kaavio selventää Convian ohjelmistokehyksellä toteutetun lisäarvosovelluksen mahdollista ylläpitoprosessia (kuvio 7).



Kuvio 7: Lisäarvosovelluksen ylläpitoprosessi

Koska regressiotestaus on osa testauslähtöistä ohjelmistokehitysprosessia, olisi ohjelmistokehityksen mallia hieman muutettava Conviassa. Ohjelmistokehityksen toteutustapaa tulisi viedä vahvemmin testauksen ohjaaman ohjelmistokehityksen suuntaan. Mallin sovittaminen Convian toimintatapoihin onnistuisi hyvin, sillä ohjelmistokehitykselle ei ole tässä vaiheessa vakiintunut mitään tiettyä tapaa. Järjestelmällisempi sovellusten tuotantoprosessi ehkäisisi osaltaan lisäarvosovellusten sisältämiä puutteita tai virheitä.

Ohjelmistokehykselle, eli käytännössä Java-lähdekoodiin, on kehitettävä testiluokat jokaiselle luokalle, jotta regressiotestaus voidaan suorittaa mahdollisimman kattavasti muutosten jälkeen. Alkuvaiheessa olisi keskityttävä siihen, että uusia luokkia tehtäessä niiden testiluokat toteutettaisiin ennen varsinaisen koodin kirjoittamista TDD-mallin mukaisesti. Jo olemassa oleville luokille testiluokat tulisi toteuttaa myös tilaisuuden tullen. Koska regressiotestaus nojaa vahvasti yksikkötestaukseen, kappaleessa 4.2 keskitytään siihen hieman syvällisemmin.

4.2 Yksikkötestaus

Sellaista testausta, joka tapahtuu ohjelman sisäisten rakenteiden tasolla, eli jolla testataan ohjelman pienempien rakenneyksiköiden toimintaa, kutsutaan yksikkötestaukseksi. Yksikkötestauksessa on tärkeää tietää, miten ohjelma rakentuu. Tästä puolestaan aiheutuu se, että yksikkötestaus voidaan mieltää lasilaatikkotestaamiseksi (kts. kappale 4.1.2). Yksikkötestauksessa ohjelman rakenne on siis kokoajan nähtävissä ja sen testaamista suoritetaan sisäisesti. (Harju & Juslin, 2006, 89-90.)

Yksikkötestaus on siis hyvin matalan tason testausta, jossa pyritään varmistumaan koodin toimivuudesta metoditasolla. Tästä johtuen koodin kehittäjä kirjoittaa usein tarvittavat yksikkötestit omalle koodilleen. Yksikkötestausta hyödynnetään erityisesti TDD:ssä, mutta toki myös muunlaisessa ohjelmistokehityksessä. Yksikkötestauksen ominaisimpiin piirteisiin kuuluu se, että yksittäistä metodia varten vaaditaan useita testejä erilaisilla parametrikombinaatioilla. Vain näin on mahdollista varmistaa metodin toimivuus. (Ketterät käytännöt.fi: Yksikkötestaus, 2011.)

4.2.1 JUnit-sovelluskehyksen hyödyntäminen

Javassa yksikkötestaamisen suorittamiseen tarvittavien testeriluokkien tekemiseen on olemassa avoimeen lähdekoodiin perustuva JUnit-sovelluskehys. Kuten Harju ja Juslin (2006) kirjassaan toteavat, JUnitilla on vahva jalansija XP:n mukaisessa ohjelmistokehityksessä. Convian ohjelmistokehyksen kanssa JUnit-testien avulla voitaisiin varmistua siitä, että kehys toimii tarkoitetulla tavalla etenkin muutosten jälkeen. JUnitissa on paljon valmiita testiluokkien ja -tapausten tekemistä helpottavia metodeita ja luokkia.

Koska yksikkötestaamisen toteuttamiseen tarvittavien JUnit-testiluokkien toteuttaminen on aikaa vievää työtä, tulisi niiden toteuttaminen aloittaa ohjelmistokehyksen tärkeimmistä osista. Etenkin ProeManager- ja ProeApi-luokkien testiluokat tulisi toteuttaa ensin, sillä ne muodostavat Convian lisäarvosovellusten rungon. Myös ActionManager-luokan toimintojen kattava testaaminen on ensiarvoisen tärkeää, sillä monet ohjelman perustoiminnoista löytyvät kyseisestä luokasta.

JUnit-sovelluskehyksen käytön aloittaminen Convian ohjelmistokehyksen yksikkötestauksessa toteutuisi helposti, sillä JUnit on helposti integroitavissa Eclipse-ohjelmointiympäristöön. Kyseinen avoimeen lähdekoodiin perustuva IDE on käytössä Convian sovelluskehittäjillä.

4.2.2 Testien ohjelmointi JUnitilla

JUnit-testiluokkien kirjoittamisessa on tärkeää tuntea sen tarjoamat perustoiminnot testitapausten ajamiseen ja raportointiin. Tärkeimmät JUnitin toiminnot ovat:

- testien ajajaluokat
- assert-lauseet
- fixtuurit
- testitapaukset

Seuraavassa esimerkissä (esimerkki 3) on esitetty yksinkertainen Java-luokka, jolle on tarkoitus rakentaa oma JUnit-testiluokkansa. Esimerkkiluokkaa voidaan käyttää

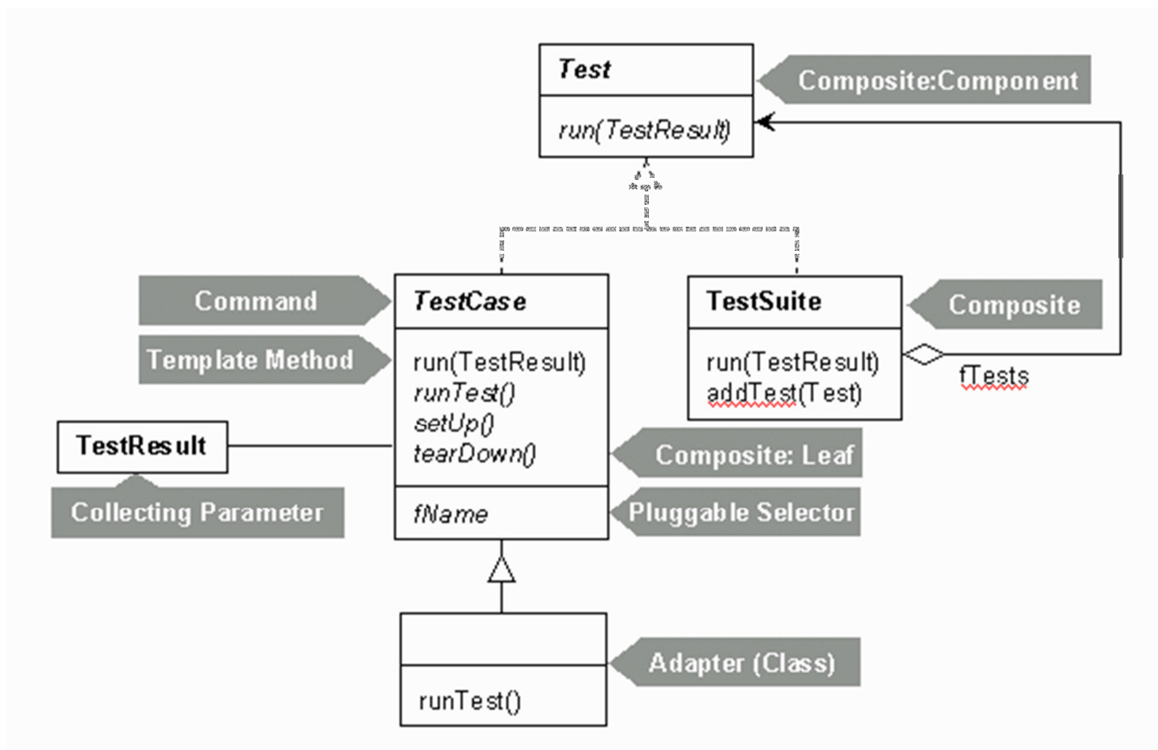
yhteenlaskujen toteuttamiseen. Lisäksi luokasta löytyy vertailumetodi, jolla parametrina annettua lukua voidaan verrata laskettuun summaan.

```
public class Summa {
   private int summa;

   public Summa (int luku1, int luku2) {
      setSumma(luku1, luku2);
   }

   public void setSumma (int luku1) {
      if (luku1 != null && luku2 != null) {
         this.summa = luku1 + luku2;
      } else if (luku1 == null) {
         System.out.println(”Luku 1 on virheellinen.”);
      } else if (luku2 == null) {
         System.out.println(”Luku 2 on virheellinen.”);
      }
   }

   public boolean compareNumbers (int verrattavaLuku) {
      if (summa == verrattavaLuku) {
         return true;
      } else {
         return false;
      }
   }
}
```

Esimerkki 3: Yksinkertainen Java-esimerkkiluokka

JUnit testiluokan rakentaminen lähtee liikkeelle testiluokan rungon luomisesta. Kaikki testiluokat on periytettävä JUnitin TestCase-luokasta. Jotta testattavan luokan toimivuutta voidaan alkaa testaamaan, tulee testiluokkaan tehdä setUp-niminen metodi, joka vastaa testiluokan kaikkien testimetodien yhteisen alkutilan asettamisesta. Alkutila eli fixtuuri määrittää esimerkiksi käytettävät resurssit, joita testimetodit tarvitsevat testien suorittamiseen. Kun testiluokan runko on tehty, voidaan testiluokkaan tehdä varsinaiset testitapaukset, joita voi olla useita. Testit ovat testiluokan metodeita ja niiden nimien tulee alkaa sanalla test. Tällöin JUnit osaa löytää automaattisesti kaikki testiluokan testit. Eri testitapaukset kannattaa tehdä toisistaan riippumattomiksi, jotta ne voidaan suorittaa missä järjestyksessä tahansa. (Harju & Juslin, 2006, 91-92.)

JUnit-testiluokkien rakenteen ymmärtämistä helpottaa oheinen UML-kaavio (kuvio 8). Kaaviosta selviää, miten testiluokat periytyvät TestCase-luokasta ja miten ne ovat kytköksissä itse testiin. JUnit sisältää siis käytännössä neljä erillistä komponenttia, jotka muodostavat testiympäristön.



Kuvio 8: JUnit Patterns Summary (JUnit: A Cook's Tour, 2011)

Kuviossa mainittua tearDown-metodia käytetään JUnit-testaamisessa setUp-metodilla varattujen resurssien vapauttamiseen. Jos siis testien toteuttamista varten olisi tarpeen käyttää esimerkiksi tietokantaa, kyseinen metodi olisi sisällytettävä itse testiluokkaan. Vapautusmetodi ei ole kuitenkaan pakollinen testiluokassa, mikäli mitään vapauttamista vaativia resursseja ei ole käytetty.

Aiemmin esitetyn summa-luokan testaaminen voidaan aloittaa esimerkiksi compareNumbers-metodin testaamisella. Kun testiluokan runko on valmis, on tärkeää muistaa tehdä testiluokalle setUp-metodi resurssien varaamiseen. Summa-luokan testaamisessa on tarpeen luoda ainoastaan yksi olio kyseisestä luokasta. Tämän olion avulla voidaan testata compareNumbers-metodin toimivuutta. Seuraavassa on esitettynä itse testiluokka (esimerkki 4).

```
import junit.framework.Assert;
import junit.framework.TestCase;

public class Summatesti extends TestCase {
   private Summa summa;

   public void setup() {
      summa = new Summa(1,1);
   }

   public void testComparison() {
      Assert.assertTrue ("Arvot eivät täsmää", summa.compareNumbers(2));
   }

   public static void main(String[] args) {
      junit.swingui.TestRunner.run(Summatesti.class);
   }
}
```

Esimerkki 4: Testiluokka summa-luokan testaamiseen

TestComparison-testimetodi testaa summa-luokan vertailumetodin toimivuutta. Testaamisessa käytetään JUnitin Assert-luokkaa, jossa on useita erilaisia assert-lauseita. Esimerkissä käytetty assertTrue-lause koostuu virheilmoituksesta, sekä testattavasta toiminnosta, tässä siis summa-luokan compareNumbers-metodista.

JUnit-testit voidaan ajaa joko käyttäen esimerkistäkin löytyvää main-funktiota, tai ne voidaan käynnistää suoraan komentoriviltä. Convian ohjelmistokehyksen testaamisessa voidaan hyödyntää kumpaa tahansa näistä tavoista. Tärkeintä on tehdä testiluokat jokaiselle luokalle, jotta yksikkötesteillä voidaan kattaa koko järjestelmä. Tällöin ohjelmistokehykselle voidaan ajaa kattava regressiotestaus jokaisen muutoksen jälkeen.

Kun oheinen testiohjelma ajetaan, testataan palauttaako summa-luokan vertailumetodi arvon true, koska luodun summa-olion arvo ja vertailtava arvo ovat samat. Mikäli arvo ei ole true, antaa testiohjelma asetetun virheilmoituksen. Testiluokkaan voitaisiin tehdä myös esimerkiksi testimetodi sille, palauttaako summa-luokan vertailumetodi arvon false lukujen ollessa erisuuret. Tällöin käytettäisiin testaamiseen Assert.assertFalse-lausetta. Liitteessä 2 on listattuna Assert-luokan eri metodeita, joita voidaan hyödyntää testikoodissa. Monet näistä metodeista ovat erittäin hyödyllisiä, kun ohjelmistokehystä testaavia testiluokkia rakennetaan.

5 XML-SKRIPTIEN RAKENNEMÄÄRITTELY

Convian ohjelmistokehystä ja sen toimintaa ohjaillaan XML-tiedostojen avulla. Näiden XML-tiedostojen perusteella sovelluksille rakennetaan käyttöliittymä ja määritellään niiden sisältämät toiminnallisuudet. Tällä ratkaisumallilla on pyritty nopeuttamaan uusien lisäarvosovellusten luomista ja tehostamaan koodin uudelleenkäytettävyyttä.

Koska ohjelmoija on itse vastuussa kirjoittamiensa XML-skriptien rakenteen oikeellisuudesta, on tarpeen kehittää XML-tiedostojen validointia niin, ettei ohjelmoijan itsensä tarvitsisi huolehtia liikaa rakenteesta. Tällöin voidaan lisäarvosovellusten implementointia entisestään tehostaa. Kun aikaa ei kulu virheellisten XML-skriptien läpikäymiseen, voidaan keskittyä itse sovellusten ominaisuuksien hiomiseen ja muuhun testaamiseen.

On tärkeää dokumentoida kaikki XML-koodin eri tagit ja attribuutit, joita voidaan käyttää sovellusten luomiseen ohjelmistokehyksen avulla. Rakennemäärittely puolestaan varmistaa, että syntaksivirheet huomataan jo aikaisessa vaiheessa ennen sovelluksen muun testaamisen aloittamista.

5.1 DTD

DTD on rakennemäärittelytapa, jota käytetään XML- ja SGML-kielten yhteydessä. Sen avulla voidaan määritellä rakenteisen dokumentin elementtien ja attribuuttien sallitut ilmenemismuodot, jolloin määrittelystä muodostuu uusi merkintäkieli. (Wikipedia: DTD, 2011.)

DTD:tä voidaan hyödyntää joko sisällyttämällä se jokaiseen XML-tiedostoon erikseen, tai käyttämällä erillisiä DTD-määrittelydokumentteja, joihin viitataan niitä käyttävien XML-tiedostojen sisällä. Koska uudelleenkäytettävyys on suuressa roolissa Convian lisäarvosovelluksissa, DTD-määritykset kannattaa tehdä erillisiin tiedostoihin, joita käytetään sitten kaikkien XML-tiedostojen yhteydessä. Näin vältytään tilanteelta, jossa uuden XML-elementin määrittely joudutaan tekemään jokaiseen XML-tiedostoon erikseen.

5.1.1 DTD-määrittelyn rakenne

Dokumenttityyppimäärittelyn eli DTD:n rakenne on hyvin selkeä ja eritoten XML-kieltä tuntevalle helposti ymmärrettävä. Rakenteeseen kuuluu useita erilaisia nimettyjä komponentteja, joiden avulla määrittely rakentuu. Kuten Nakhimovsky ja Myers (2002, 212-213) kirjassaan mainitsevat, DTD:tä voidaan pitää ikään kuin XML-kielen kielioppina.

Esimerkissä 5 on aloitettu määrittely !DOCTYPE-komennolla. Eri elementit on vastaavasti määritelty käyttämällä !ELEMENT-komentoja. Elementtien nimet on merkitty määrittelyyn komennon jälkeen. Lisäksi elementeille on määritelty niille sallitun datan tietotyyppi sulkujen sisään.

Yksinkertaisimmillaan DTD voi olla seuraavan esimerkin mukaisesti sisällytettynä itse XML-tiedostoon (esimerkki 5).

```
<?xml version="1.0"?>
<!DOCTYPE note [
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>]>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend</body>
</note>
```

Esimerkki 5: DTD:n käyttö XML-tiedoston sisällä (W3Schools: DTD Tutorial 2011)

DTD-rakennemäärittelyn avulla voidaan siis määritellä, minkälaisia elementtejä XML-dokumenteissa sallitaan ja minkälaisia attribuutteja elementeillä voi olla. Attribuutit voidaan listata myös käyttämällä !ATTLIST-komentoa.

Kuten aiemmin mainittiin, Convian XML-tiedostojen rakennemäärittely on syytä tehdä erillisiin tiedostoihin. Näin voidaan samoja määrityksiä käyttää useiden eri tiedostojen

kesken. Ulkoisiin määrittelyihin tulee viitata XML-tiedostojen alussa käyttämällä DTD:n !DOCTYPE avainsanaa. Jos esimerkiksi halutaan viitata form-elementin rakennemäärittelyyn, tulee tehdä esimerkin 6 mukainen viittaus sitä käyttävän XML-tiedoston alkuun.

```
<!DOCTYPE form SYSTEM ”../dtd/form.dtd”>
```

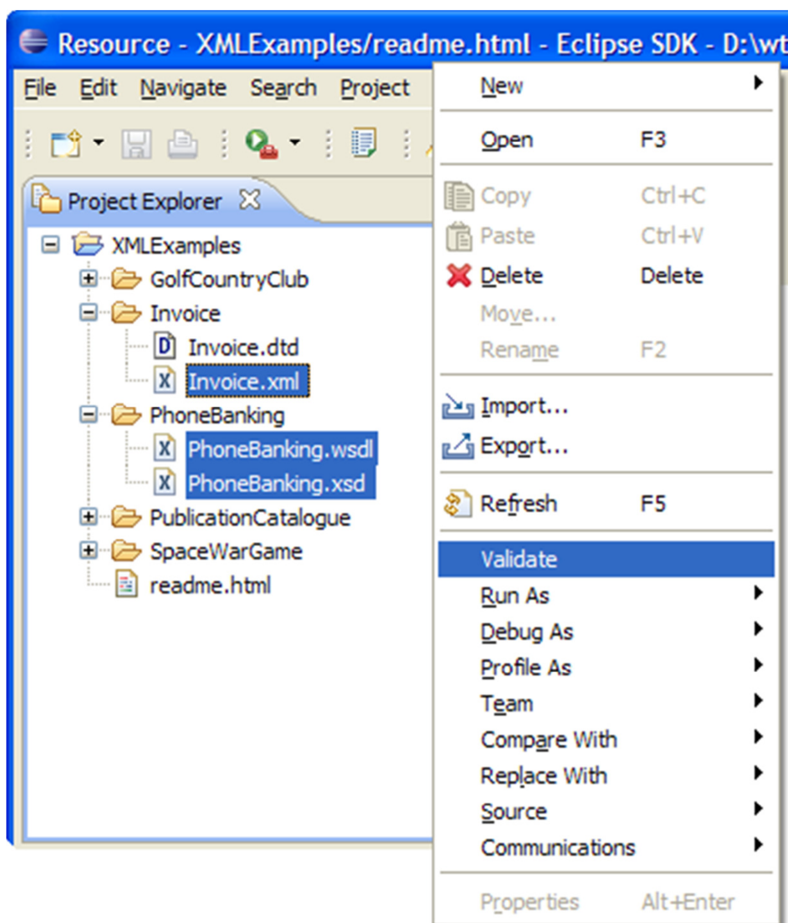
Esimerkki 6: Viittaus ulkoiseen DTD-rakennemäärittelyyn

Viittaukseen merkitään elementti, jota rakennemäärittely koskee, sekä lisäksi DTD-tiedoston sijainti. Avainsanaa SYSTEM käytetään polun määrittämiseen.

5.1.2 XML-skriptien validointi DTD:n avulla

Kun DTD-rakennemäärittelytiedostot on tehty ja niitä käyttävät XML-tiedostot sisältävät tarvittavat viittaukset, voidaan tiedostoille suorittaa validointi. Tämä tarkoittaa käytännössä XML-tiedostojen parsimista siihen sopivalla ohjelmalla. Mikäli tiedostoissa on virheitä, parseri antaa siitä ilmoituksen.

Validointi voidaan suorittaa esimerkiksi Microsoftin Internet Explorerilla. Lisäksi XML-tiedostojen validointi onnistuu erityisesti kyseiseen tehtävään erikoistuneilla internet-sivuilla. Convian tapauksessa paras työkalu validoinnin suorittamiseen on kuitenkin Eclipse, johtuen siitä, että sitä käytetään ohjelmistokehyksen kehityksessä. Eclipsen käyttöliittymän avulla validointi on helppo suorittaa ja mahdolliset virheet raportoidaan kattavasti. Seuraava kuva havainnollistaa validoinnin suorittamista Eclipsessä (kuvio 9).



Kuvio 9: XML-tiedostojen validointi Eclipsessä (Eclipse Validation Tutorial, 2011)

5.2 Rakennemäärittelyn toteuttaminen

Rakennemäärittelyn toteuttamisessa on lähdettävä liikkeelle siitä, että kaikki XML-dokumenteissa käytettävät elementit ja niiden attribuutit on dokumentoitu tarkasti. Tämän jälkeen voidaan kirjoittaa DTD-rakennemäärittelytiedostot jokaiselle elementille, ja tallentaa ne sopivaan paikkaan. Erillisten DTD-tiedostojen ylläpito on huomattavasti helpompaa kuin XML-skripteihin sisällytettyjen määrittelyiden ylläpito.

Valmiiden DTD-tiedostojen avulla validointi tulisi suorittaa aina, kun ohjelmiin tehdään muutoksia. Validoitujen XML-tiedostojen ansiosta voidaan tehokkaasti välttää syntaksivirheitä XML-datan joukossa. Tämä tehostaa omalta osaltaan lisäarvosovellusten tuotekehitysprosessia.

6 TESTAUKSEN TOTEUTTAMISSUUNNITELMA

Convian ohjelmistokehyksen ja sillä toteutettujen lisäarvosovellusten testaaminen on monivaiheinen prosessi, johon sisältyy paljon työtä ja erilaisia vaiheita. Jotta testaus voidaan toteuttaa riittävän kattavasti, tulee aiemmissa kappaleissa kuvattuja menetelmiä käyttää soveltuvin osin tuotteiden laadun parantamiseksi.

Testauksen avulla voidaan lisäksi saavuttaa merkittäviä säästöjä niin kehitykseen kuluvassa ajassa kuin myös kustannuksissa. Seuraavassa on vielä tiivistetysti kuvattu, kuinka testausprosessia olisi lähdettävä viemään eteenpäin.

6.1 Yksikkötestauksen suorittaminen ohjelmistokehyksessä

Ohjelmistokehystä käytetään Convian kaikissa J-Link-pohjaisissa lisäarvosovelluksissa, joten sen toimivuus on varmistettava tarkoin. JUnit-testiluokkien avulla ohjelmistokehyksen toimintaa voidaan testata kattavasti. Testiluokkien tekeminen vaatii paljon aikaa, sillä ohjelmistokehys sisältää huomattavan määrän erilaisia luokkia. Testien kehittämisessä tulisi lähteä liikkeelle sovellusten kannalta tärkeimpien ja eniten tarvittujen luokkien testaamisesta, ja vähitellen rakentaa testiluokat koko järjestelmälle.

Kattavan regressiotestauksen toteuttaminen vaatisi testiluokkien rakentamisen ohjelmistokehyksen kaikille osille. Kun testiluokkien kattavuus on suuri, voidaan regressiotestauksen avulla varmistaa kehyksen toimivuus jokaisen muutoksen jälkeen. Regressiotestaus parantaa tuotteiden laatua merkittävästi, joten yksikkötestaukseen on syytä panostaa.

6.2 XML-testiskriptien laatiminen ja ajaminen

XML-skriptien validoinnin lisäksi lisäarvosovellusten toiminnallisuuden varmistamiseen voidaan käyttää testiskriptejä, jotka käyvät kattavasti läpi kaikki mahdolliset ohjelmistokehyksen toiminnot. XML-testiskriptien laatiminen on jo aloitettu ja niiden tekemiseen tulee panostaa myös jatkossa. Testiskriptien laatimisesta

on hyötynä myös se, että sen ohessa voidaan dokumentoida järjestelmän toiminnallisuudet erittäin kattavasti.

Testiskripteillä on tarkoitus luoda yksinkertaisia sovelluksia, joiden avulla voidaan eri toimintojen ja käyttöliittymäkomponenttien toimivuus varmistaa. Mahdollisuuksien mukaan tulisi harkita myös testien automatisointia, sillä nykyisellään testiskriptien luomia sovelluksia testaamaan tarvitaan erillinen testaaja. Liitteessä 3 on lyhyt esimerkki tähän mennessä laadittujen testiskriptien sisällöstä.

Mustalaatikkotestauksen hyödyllisyyttä ei tietenkään tulisi aliarvioida, joten testiohjelmien toimivuutta ja ominaisuuksia olisi tarpeen testauttaa myös yrityksen muilla työntekijöillä. Täysin automatisoituun testaamisprosessiin ei tulisi siis tyytyä.

6.3 TDD:n hyödyntäminen sovelluskehityksessä

Täyden hyödyn saavuttaminen yksikkötestauksesta vaatii työskentelytapojen kehittämistä monessakin suhteessa. Eritoten TDD:n hyödyntäminen Convian sovelluskehityksessä olisi hyödyksi. Kun ohjelmistokehykseen kehitetään uusia luokkia ja ominaisuuksia, tulisi niiden kehittämisessä lähteä liikkeelle testien tekemisestä. Tällä tavoin mahdollistettaisiin myös kattavan regressiotestauksen suorittaminen jatkossa koko järjestelmälle. Koska testiluokkien laatiminen jälkikäteen on selkeästi hitaampaa, TDD:llä saavutettaisiin merkittävää etua tämän hetkiseen sovelluskehitysmalliin verrattuna.

7 YHTEENVETO

Convia Oy:n kehittämän ohjelmistokehyksen testaaminen on monivaiheinen prosessi. Kehyksen kattava testaaminen vaatii normaalin sovelluskehityksen rinnalla lähes yhtä paljon työtä, mutta on välttämätön osa laadukkaiden lisäarvosovellusten tuotekehitys- ja ylläpitoprosessia. Useita erilaisia testausmenetelmiä ja validointitapoja soveltaen yhdistelemällä saavutetaan suurin hyöty käytettäväksi valituista menetelmistä.

Java-pohjaisen ohjelmistokehyksen testaamisessa voidaan hyödyntää erityisen hyvin yksikkötestausta, eritoten avoimeen lähdekoodiin perustuvaa JUnit-sovelluskehystä. Sen sisältämät testausmetodit ja valmiit luokat helpottavat ja tehostavat ohjelmistokehyksen testaamista merkittävästi. Koska kyse on jatkuvasti kehittyvästä ja uudelleenkäytettävyyteen perustuvasta ohjelmistosta, sen regressiotestaukseen on panostettava yhä enenevissä määrin. Regressiotestauksen avulla muutoksien toteuttaminen on nopeampaa ja turvallisempaa, kun mahdollisesti aiheutuneet virhetilanteet havaitaan jo aikaisessa vaiheessa ennen tuotteiden mustalaatikkotestaamista.

XML-skriptien validoinnilla voidaan myös entisestään tehostaa sovelluskehitystä Conviassa. Sen hyödyt ovat suuret, sillä validoinnin avulla sovelluskehittäjien aikaa ei kulu virheellisen syntaksin läpikäymiseen. Virheet löydetään huomattavasti nopeammin DTD-rakennemäärittelyä hyödyntämällä. Lisäarvoa rakennemäärittelyn laatimiselle antaa se, että myös dokumentointia voidaan suorittaa sen ohella.

Ketterät menetelmät ja testauksen ohjaama sovelluskehitys ovat nykypäivänä paljon käytössä ohjelmistoalalla. Näihin toimintamalleihin perehtymällä voidaan Convian sovelluskehitystä viedä eteenpäin ja varmistaa tuotteiden korkea laatu myös tulevaisuudessa.

LÄHDELUETTELO

Painetut lähteet

Harju, Jukka & Juslin, Jukka 2006. Tuloksellinen Java-ohjelmointi. Helsinki: Edita Publishing Oy.

Koskimies, Kai & Mikkonen, Tommi 2005. Ohjelmistoarkkitehtuurit. Jyväskylä: Talentum Media Oy.

McLaughlin, Brett 2001. Java & XML: Tehokäyttäjän opas (suomennos). O'Reilly and Associates, Inc.

Nachimovsky, Alexander & Myers, Tom 1999. Inside Java ja XML (suomennos). Wrox Press.

Naik, Kshirasagar & Tripathy, Priyadarshi 2008. Software testing and quality assurance. Hoboken: John Wiley & Sons, Inc.

Sähköiset lähteet

Parametric Technology Corporation 2010. Pro/ENGINEER Wildfire 5.0 J-Link User's Guide. [pdf]

Ketterät käytännöt.fi: Yksikkötestaus. [online]. [viitattu 13.4.2011] Saatavissa: http://www.ketteratkaytannot.fi/fi-FI/Kaytannot/Yksikkotestaus/

Wikipedia: DTD. [online]. [viitattu 13.4.2011] Saatavissa: http://fi.wikipedia.org/wiki/DTD

W3Schools: DTD Tutorial. [online]. [viitattu 4.5.2011] Saatavissa: http://www.w3schools.com/dtd/

Wikipedia: XML. [online]. [viitattu 9.5.2011] Saatavissa: http://fi.wikipedia.org/wiki/XML

JUnit: A Cook's Tour. [online]. [viitattu 15.5.2011] Saatavissa: http://junit.sourceforge.net/doc/cookstour/cookstour.htm

Eclipse Validation Tutorial. [online]. [viitattu 16.5.2011] Saatavissa: http://www.eclipse.org/webtools/community/tutorials/XMLValidation/XMLValidation Tutorial.html

LIITTEET

Liite 1: Esimerkkisovellus

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<appdef>
   <task name="show_message">
      <action name="base.ui.msgbox">
         <attribute name="message" value="Hello World!" />
      </action>
   </task>

   <form
      name="window"
      title="Test window"
      resize="false"
      centered="true"
      height="180"
      width="300"
      onload_action=""
      action="" >

      <frame name="hello_frame"
            title="Press button:"
            colspan="1"
            cols="1"
            rowpos="0"
            rowspan="1"
            rows="1">

         <button name="hello_button"
               colpos="0"
               rowpos="0"
               title="Button"
               related_to=""
               action="show_message"
               width="130"
               fill="none"
               align="center" />
      </frame>
   </form>
</appdef>
```

Liite 2: JUnit, Assert-luokan metodit

(Lähde: http://junit.sourceforge.net/javadoc/org/junit/Assert.html)

Class Assert

```
java.lang.Object
  └ org.junit.Assert
```

```
public class Assert
extends java.lang.Object
```

A set of assertion methods useful for writing tests. Only failed assertions are recorded. These methods can be used directly: `Assert.assertEquals(...)`, however, they read better if they are referenced through static import:

```
 import static org.junit.Assert.*;
    ...
    assertEquals(...);
```

See Also:

`AssertionError`

Constructor Summary

`protected`	`Assert()` Protect constructor since it is a static only class

Method Summary

`static void`	`assertArrayEquals(byte[] expecteds, byte[] actuals)` Asserts that two byte arrays are equal.
`static void`	`assertArrayEquals(char[] expecteds, char[] actuals)` Asserts that two char arrays are equal.
`static void`	`assertArrayEquals(int[] expecteds, int[] actuals)` Asserts that two int arrays are equal.
`static void`	`assertArrayEquals(long[] expecteds, long[] actuals)` Asserts that two long arrays are equal.
`static void`	`assertArrayEquals(java.lang.Object[] expecteds, java.lang.Object[] actuals)` Asserts that two object arrays are equal.

static void	assertArrayEquals(short[] expecteds, short[] actuals) Asserts that two short arrays are equal.
static void	assertArrayEquals(java.lang.String message, byte[] expecteds, byte[] actuals) Asserts that two byte arrays are equal.
static void	assertArrayEquals(java.lang.String message, char[] expecteds, char[] actuals) Asserts that two char arrays are equal.
static void	assertArrayEquals(java.lang.String message, int[] expecteds, int[] actuals) Asserts that two int arrays are equal.
static void	assertArrayEquals(java.lang.String message, long[] expecteds, long[] actuals) Asserts that two long arrays are equal.
static void	assertArrayEquals(java.lang.String message, java.lang.Object[] expecteds, java.lang.Object[] actuals) Asserts that two object arrays are equal.
static void	assertArrayEquals(java.lang.String message, short[] expecteds, short[] actuals) Asserts that two short arrays are equal.
static void	assertEquals(double expected, double actual) Deprecated. Use assertEquals(double expected, double actual, double epsilon) instead
static void	assertEquals(double expected, double actual, double delta) Asserts that two doubles or floats are equal to within a positive delta.
static void	assertEquals(long expected, long actual) Asserts that two longs are equal.
static void	assertEquals(java.lang.Object[] expecteds, java.lang.Object[] actuals)

	Deprecated. use assertArrayEquals
`static void`	assertEquals(java.lang.Object expected, java.lang.Object actual) Asserts that two objects are equal.
`static void`	assertEquals(java.lang.String message, double expected, double actual) Deprecated. Use assertEquals(String message, double expected, double actual, double epsilon) instead
`static void`	assertEquals(java.lang.String message, double expected, double actual, double delta) Asserts that two doubles or floats are equal to within a positive delta.
`static void`	assertEquals(java.lang.String message, long expected, long actual) Asserts that two longs are equal.
`static void`	assertEquals(java.lang.String message, java.lang.Object[] expecteds, java.lang.Object[] actuals) Deprecated. use assertArrayEquals
`static void`	assertEquals(java.lang.String message, java.lang.Object expected, java.lang.Object actual) Asserts that two objects are equal.
`static void`	assertFalse(boolean condition) Asserts that a condition is false.
`static void`	assertFalse(java.lang.String message, boolean condition) Asserts that a condition is false.
`static void`	assertNotNull(java.lang.Object object) Asserts that an object isn't null.
`static void`	assertNotNull(java.lang.String message, java.lang.Object object) Asserts that an object isn't null.
`static void`	assertNotSame(java.lang.Object unexpected, java.lang.Object actual) Asserts that two objects do not refer to the same object.

static void	assertNotSame(java.lang.String message, java.lang.Object unexpected, java.lang.Object actual) Asserts that two objects do not refer to the same object.
static void	assertNull(java.lang.Object object) Asserts that an object is null.
static void	assertNull(java.lang.String message, java.lang.Object object) Asserts that an object is null.
static void	assertSame(java.lang.Object expected, java.lang.Object actual) Asserts that two objects refer to the same object.
static void	assertSame(java.lang.String message, java.lang.Object expected, java.lang.Object actual) Asserts that two objects refer to the same object.
static <T> void	assertThat(java.lang.String reason, T actual, org.hamcrest.Matcher<T> matcher) Asserts that actual satisfies the condition specified by matcher.
static <T> void	assertThat(T actual, org.hamcrest.Matcher<T> matcher) Asserts that actual satisfies the condition specified by matcher.
static void	assertTrue(boolean condition) Asserts that a condition is true.
static void	assertTrue(java.lang.String message, boolean condition) Asserts that a condition is true.
static void	fail() Fails a test with no message.
static void	fail(java.lang.String message) Fails a test with the given message.

Liite 3: Näyte XML-testiskriptistä


```
<appdef>
<task name="test_base.file_actions">
   <action name="base.file.copy">
      <attribute name="srcfile" value="testfile.txt" />
      <attribute name="tgtfile" value="testfile2.txt" />
   </action>
   <action name="base.file.exists">
      <attribute name="filename" value="testfile2.txt" />
   </action>
   <action name="base.file.delete">
      <attribute name="filename" value="testfile.txt" />
   </action>
   <action name="base.file.dir">
      <attribute name="path" value="modules" />
   </action>
   <action name="base.file.info">
      <attribute name="filename" value="testfile2.txt" />
   </action>
   <action name="base.file.mkdir">
      <attribute name="path" value="testfolder" />
   </action>
</task>

<form
   name="window"
   title="Action test"
   resize="false"
   centered="true"
   height="600"
   width="600"
   onload_action=""
   action=""
>

<frame name="start_test"  title="BASE-module tests" colspan="1" cols="6"
rowpos="0" rowspan="1" rows="6">

   <label title="Choose a test:" />
   <text name="test_field" colpos="1" rowpos="1" fill="horizontal" width="220"/>
   <button name="start1"
      colpos="1"
      rowpos="2"
      title="base.file TEST"
      related_to=""
      action="test_base.file_actions"
      width="125"
      fill="none"
```

```
        align="left"
    />

    <button name="start2"
        colpos="2"
        rowpos="2"
        title="base.string TEST"
        related_to=""
        action="test_base.string_actions"
        width="125"
        fill="none"
    align="right"
    />

    <button name="start3"
        colpos="1"
        rowpos="3"
        title="base.data TEST"
        related_to=""
        action="test_base.data_actions"
        width="125"
        fill="none"
        align="left"
    />

<button name="start4"
        colpos="2"
        rowpos="3"
        title="base.system TEST"
        related_to=""
        action="test_base.system_actions"
        width="125"
        fill="none"
        align="right"
    />

    <button name="start5"
        colpos="1"
        rowpos="4"
        title="base.ui TEST"
        related_to=""
        action="test_base.ui_actions"
        width="125"
        fill="none"
        align="left"
    />
</frame>
</form>

</appdef>
```