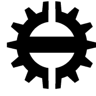


TUOMAS VÄLIMÄKI

AUTOMAATTISEN TESTAUSJÄRJESTELMÄN KEHITYS

Diplomityö

Tarkastaja: professori Kai Koskimies
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston kokouksessa
5. lokakuuta 2011

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
VÄLIMÄKI, TUOMAS: Automaattisen testausjärjestelmän kehitys
Diplomityö, 58 sivua
Joulukuu 2011
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Kai Koskimies
Avainsanat: automaattinen testaus, sovelluskehys, ohjelmistoarkkitehtuuri, täsmäkieli

Aluksi tässä työssä toteutetaan Insta DefSec Oy:ssä erästä projektia varten suunniteltu automaattinen testausjärjestelmä. Testausjärjestelmästä pyritään tunnistamaan sen ongelmat pääasiassa sen toteutuksen ja käytön aikana kerätyn kokemuksen perusteella. Näitä ongelmia ratkotaan laatimalla kolmiosainen kehityssuunnitelma. Lopuksi kehityssuunnitelma arvioidaan esimerkkien avulla.

Testausjärjestelmään kuuluu laiteympäristö, laiteympäristöä ohjaavat sovellukset ja testien tekemiseen tarkoitettu sovelluskehys. Testausjärjestelmä on kehitetty yhtä tuotetta varten, ja tätä on käytetty hyväksi järjestelmän suunnittelussa esimerkiksi yksinkertaistamalla laiteympäristöä ja sen hallintaa.

Sovelluskehys on moduulipohjainen, mutta se käyttää myös datapohjaisten testaussovelluskehysten periaatetta, jossa testidata erotellaan testilogiikasta. Sen arkkitehtuuri on jaettu kolmeen osaan: datankäsittelyyn, yhteyksiin ja raportointiin. Datankäsittely tarjoaa testidatan määrittelyn ja vertailun. Yhteydet tarjoavat testeille yhteydet testausjestelmän osiin ja testattavan tuotteen rajapintoihin. Raportointi tarjoaa keinot testin kulun ja tulosten raportointiin.

Ensimmäinen osa kehityssuunnitelmasta kohdentuu testidatan määrittelyyn ja käsittelyyn. Näitä osia parannetaan helpottamalla testidatan konfigurointia ja ylläpidettävyyttä siirtämällä datan määritykset XML:stä Scala-ohjelmointikielelle.

Toinen osa kehityssuunnitelmasta liittyy sovelluskehyksen tarjoamien yhteyksien käyttöön. Yhteyksien luomista yksinkertaistetaan, ja niiden tarjoama rajapinta suunnitellaan uudelleen. Rajapinnan uudelleensuunnittelulla pyritään helpottamaan eri tyyppisten yhteyksien käyttöä tarjoamalla niiden käyttömallia paremmin vastaavat rajapinnat.

Kolmantena kehitysideana testaussovelluskehykseen lisätään testiasiakasluokka, TestClient. TestClient suunnitellaan tarjoamaan valmista testilogiikkaa, jota voidaan käyttää rakennuspalikoina testeissä. Tällä pyritään yksinkertaistamaan testejä ja vähentämään koodin kopioimista. TestClient toteutetaan Scalalla, ja sen rajapinta suunnitellaan tarjoamaan luonnollista kieltä mukailevan syntaksin, jolla pyritään parantamaan testien luettavuutta.

Kehityssuunnitelmia arvioitiin kolmen esimerkin avulla, joista tehtiin toteutukset olemassa olevalla sovelluskehyksellä ja kehityssuunnitelmat toteuttavalla prototyyppisovelluskehyksellä. Tehdyn arvioinnin perusteella kehitysideoiden todettiin parantavan testien toteuttamisen, ylläpitämisen ja tarkastamisen tuottavuutta.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
VÄLIMÄKI TUOMAS: Development of an Automated Testing System
Master of Science Thesis, 58 pages
December 2011
Major: Software Engineering
Examiner: Professor Kai Koskimies
Keywords: automated testing, software framework, software architecture, domain-specific language

At first in this thesis, an automated software testing system designed for a project by Insta DefSec Oy is implemented. The testing system's problems are identified mostly by using the experience gathered from developing and using the system. These problems are addressed by creating a three-part development plan. Lastly the development plan is evaluated with the help of examples.

The testing system consists of its hardware environment, the software, which controls the hardware and a framework for writing tests. The testing system is developed for a single product and this is taken advantage of in the design of the testing system.

The testing framework is modularity-based, but also uses a principle from data-driven frameworks, where the test data is separated from the test logic. Its architecture is divided to three parts: data handling, connections and reporting. Data handling provides test data definitions and comparisons. Connections provide ways to connect to the testing environment and the system under test. Reporting provides means to report the test flow and results.

The first part of the development plan is about test data handling and definitions. These are improved by moving the test data definitions from XML to Scala to improve data configuration and maintainability.

The second part of the development plan focuses on the connections offered by the framework. Connection creation is simplified and the interface is redesigned. The goal of interface redesign is to make using different types of connections easier by offering more specialized interfaces for connections.

For third development effort, a test client is added to the framework. Test client is designed to offer ready to use test logic, which can be used as building blocks for the final tests. The goal is to simplify tests and reduce code copying. Test client is implemented with Scala and it is designed to offer an interface resembling natural language in hope of improving the readability of the tests.

The development plan was evaluated using three examples, which were implemented using both the existing framework and a prototype framework implementing the development plan. Based on the evaluation, the development plan was observed to increase the productivity of implementing, maintaining and inspecting the tests.

ALKUSANAT

Olen tehnyt tämän diplomityön Insta DefSec Oy:ssä, jossa olen työskennellyt ohjelmistosuunnittelijana. Haluan kiittää asiantuntevista neuvoista ja kommenteista sekä työn tarkastajaa professori Kai Koskimiestä Tampereen teknillisestä yliopistosta että työn ohjaajaa Antti Koivistoa Insta DefSec Oy:stä.

Lisäksi kiitän projektin tiimiä, jonka jäsenenä tein työhön yhtenä osana kuulunutta testausjärjestelmän toteutusta. Oikoluvusta kiitos kuuluu äidilleni Tuija Hillmannille ja sedälleni Antti Välimäelle. Lisäksi haluan kiittää avusta myös kaikkia muita työn suunnittelua ja valmistumista edesauttaneita.

Tampereella 22.12.2011,

____

Tuomas Välimäki

SISÄLLYS

1 Johdanto ... 1
2 Testaus ohjelmistokehityksessä ... 2
2.1 Testauksen rooli ... 2
2.2 Testauksen vaiheet, tasot ja automatisointi ... 3
2.2.1 Yksikkötestaus ... 4
2.2.2 Integrointitestaus ... 4
2.2.3 Järjestelmätestaus ja hyväksymistestaus ... 5
2.2.4 Regressiotestaus ... 5
2.2.5 Jatkuva integrointi ... 5
2.3 Testauksen apuvälineet ... 6
2.3.1 Integroitu kehitysympäristö ... 6
2.3.2 Sovelluskehykset ... 7
2.3.3 Testausautomaatiojärjestelmät ... 8
3 Testausjärjestelmä ... 10
3.1 Testattava tuote ... 10
3.2 Tavoitteet ... 11
3.3 Ympäristö ja komponentit ... 12
3.4 Jatkuva integrointi testausjärjestelmässä ... 14
3.5 Testaussovelluskehys ... 14
3.5.1 Suunnitteluperusteet ... 14
3.5.2 Arkkitehtuuri ... 15
3.6 Testausjärjestelmän käyttö ... 23
3.6.1 Vaatimukset ja suunniteltu testi ... 23
3.6.2 Testin toteutus ... 23
3.6.3 Testin ajaminen ja tulosten tarkastaminen ... 27
4 Testausjärjestelmän kehityssuunnitelma ... 29
4.1 Toteutuskielen valinta ... 29
4.2 Datankäsittely ... 30
4.2.1 Analyysi ... 30
4.2.2 Ratkaisu ... 32
4.3 Yhteydet ... 35
4.3.1 Analyysi ... 35
4.3.2 Ratkaisu ... 36
4.4 Sovelluskehyksen käyttö ... 38
4.4.1 Analyysi ... 38
4.4.2 Ratkaisu ... 39
4.5 Kehityssuunnitelmien arviointi ... 47
4.5.1 Datankäsittelyn ja yhteyksien esimerkki ... 47
4.5.2 Ensimmäinen testilogiikan toteutuksen esimerkki ... 49

4.5.3 Toinen testilogiikan toteutuksen esimerkki 51
4.5.4 Tulokset 52
5 Yhteenveto 54
Lähteet 56

TERMIT JA NIIDEN MÄÄRITELMÄT

CI	**Continuous Integration, jatkuva integrointi** Muutosten jatkuva integrointi olemassa olevaan järjestelmään.
DI	**Dependency Injection, riippuvuuksien injektointi** Yksi tapa toteuttaa IoC. Luokka määrittää omat riippuvuutensa, jotka joku muu ohjelman osa antaa luokalle.
DSL	**Domain-Specific Language, täsmäkieli** Ohjelmointikieli, joka on suunniteltu tietylle sovellusalueelle.
HTTPS	**Hypertext Transfer Protocol Secure** HTTP-protokollan ja SSL/TLS-protokollan yhdistelmä, jota käytetään tiedon suojattuun siirtämiseen.
IDE	**Integrated Development Environment** Työkalu, jolla tuotetaan ohjelmakoodia ja joka tarjoaa apuvälineitä ohjelmointiin.
IoC	**Inversion of Control** Abstrakti käsite, jossa ohjelman kontrollivuo luovutetaan jonkun muun ohjelman osan haltuun.
JMS	**Java Message Service** Ohjelmointirajapinta viestien välittämiseen.
JVM	**Java Virtual Machine** Ohjelma, jolla ajetaan Java-tavukoodia.
LDAP	**Lightweight Directory Access Protocol** Hakemistopalvelujen käyttöön tarkoitettu verkkoprotokolla.
LLOC	**Logical Lines of Code** Lähdekoodin lausekkeiden lukumäärä.
SOAP	**Simple Object Access Protocol** Tietoliikenneprotokolla, joka mahdollistaa etäkutsut XML-kielen avulla.

TDD **Test-Driven Development**
Kehitysprosessi, jossa yksikkötesti kirjoitetaan ennen vastaavaa toteutusta.

UDDI **Universal Description Discovery and Integration**
Alustariippumaton ja avoin palvelurekisteristandardi.

URL **Uniform Resource Locator**
Merkkijono, joka määrittää viitteen Internet-resurssiin.

XML **eXtensible Markup Language**
Rakenteellinen merkintäkieli, jota käytetään sekä tiedon välittämiseen, että dokumenttien kuvaamiseen.

XPath **XML Path Language**
Kieli, jolla haetaan dataa XML-dokumentista.

1 JOHDANTO

Ohjelmistoja kehitetään enemmän kuin koskaan ja ohjelmiston laadusta on tullut yhä merkittävämpi tekijä ohjelmistoprojektien onnistumisessa. Ohjelmistotestauksella voidaan sekä mitata että parantaa ohjelmiston laatua, joten sen merkittävyys on kasvanut laatuvaatimusten mukana.

Testaus on perinteisesti ollut manuaalista, ihmisten tekemää työtä. Iso osa ohjelmistotestauksesta on kuitenkin itseään toistavaa työtä, jossa ihmisen mukautuvuus ei pääse oikeuksiinsa. Tämä osa testauksesta on hyvä kohde testauksen automatisoinnille. Testauksen automatisoinnissa osa, tai jopa kaikki testaustyö automatisoidaan suoritettavaksi tietokoneella. Automatisoinnilla pyritään nostamaan ohjelmiston laatua tekemällä enemmän testausta tai laskemaan projektin kustannuksia vähentämällä testaukseen kuluvia resursseja.

Testausautomaatio ei ole vaihtoehto jokaiseen projektiin, riippuen projektin luonteesta ja vaatimuksista. Tietyntyyppiset ohjelmistot soveltuvat paremmin testauksen automatisointiin kuin toiset, esimerkiksi käyttöliittymätestaus on erittäin hankala automatisoida kokonaisvaltaisesti. Testausautomaation toteuttaminen on joka tapauksessa haastavaa, riippumatta testattavasta sovelluksesta.

Testauksen automatisointiin on olemassa valmiita työkaluja ja sovelluskehyksiä, mutta harvoin valmiillakaan työkaluilla saadaan toimiva testausautomaatiojärjestelmä ilman panostuksia sen kehittämiseen [1]. Testausautomaatiojärjestelmään kuuluu usein automaattisesti ajettavan testikoodin lisäksi myös automaattinen testausympäristö.

Tässä diplomityössä toteutetaan Insta DefSec Oy:n erään projektin lopputuotetta varten kehitetty testausautomaatiojärjestelmä ja tehdään sille kehityssuunnitelma. Järjestelmän toteutus tehdään projektitiimin kanssa valmiin suunnitelman pohjalta. Testausjärjestelmä on alusta asti itse kehitetty pelkästään projektin lopputuotetta varten. Tähän ratkaisuun päädyttiin monesta syystä: lopputuotteen ympäristön haastavuus, valmiiden testausjärjestelmien käyttöönoton vaikeus sekä riippumattomuus muista tahoista.

Testausjärjestelmään kuuluu laiteympäristö, laiteympäristöä ohjaavat sovellukset, sekä testien tekemiseen tarkoitettu sovelluskehys. Järjestelmän laiteympäristö ja sitä ohjaavat sovellukset esitellään pintapuolisesti, mutta sovelluskehystä käydään tarkemmin läpi arkkitehtuurista käyttöön asti. Lopuksi sovelluskehykselle tehdään kehityssuunnitelma, jonka tavoitteena on parantaa testien luettavuutta ja ylläpitoa, sekä helpottaa testien toteuttajien työtä.

2 TESTAUS OHJELMISTOKEHITYKSESSÄ

Tässä luvussa kerrotaan ohjelmistojen testauksesta ja testauksen automatisoinnista. Myös testauksen apuvälineitä esitellään niiltä osin, mitä on käytetty toteutetun testausjärjestelmän puitteissa.

2.1 Testauksen rooli

Hyvä tapa tutustua testaukseen ja sen rooliin ohjelmistokehityksessä on tarkastella sanan testaus määrittelyä. Taulukko 1 esittelee testauksen määrityksiä eri vuosina eri henkilöiden tekeminä.

Taulukko 1 Testauksen eri määrityksiä, vapaasti käännetty lähteestä [2]

Vuosi	Määritys
1979	Testaus on ohjelman ajamista tavoitteena löytää virheitä.
1983	Testaus on mitä tahansa työtä, jolla arvioidaan ohjelman ominaisuutta. Testaus on ohjelman laadun mittaamista.
2002	Testaus on prosessi, jossa suunnitellaan, käytetään ja ylläpidetään testausjärjestelmää testattavan ohjelman laadun mittaamiseksi ja parantamiseksi.

Vuonna 1979 testausta kuvailtiin prosessiksi, jossa yritetään etsiä ohjelmasta virheitä. Määritelmän kirjoittamisen aikana se oli pätevä ja kuvailee hyvin sen ajan näkemystä testauksesta. Määritelmä on edelleen osittain pätevä, koska virheiden etsiminen on osa testausta, mutta nykyään testaus käsitetään paljon laajempana toimintana.

Vuonna 1983 testausta kuvailtiin työksi, jolla arvioidaan ohjelman tai järjestelmän ominaisuutta sekä mitataan ohjelman laatua. Tämä määritelmä on jo paljon lähempänä nykypäivän käsitystä testauksesta, koska laadun mittaaminen on otettu mukaan määritelmään.

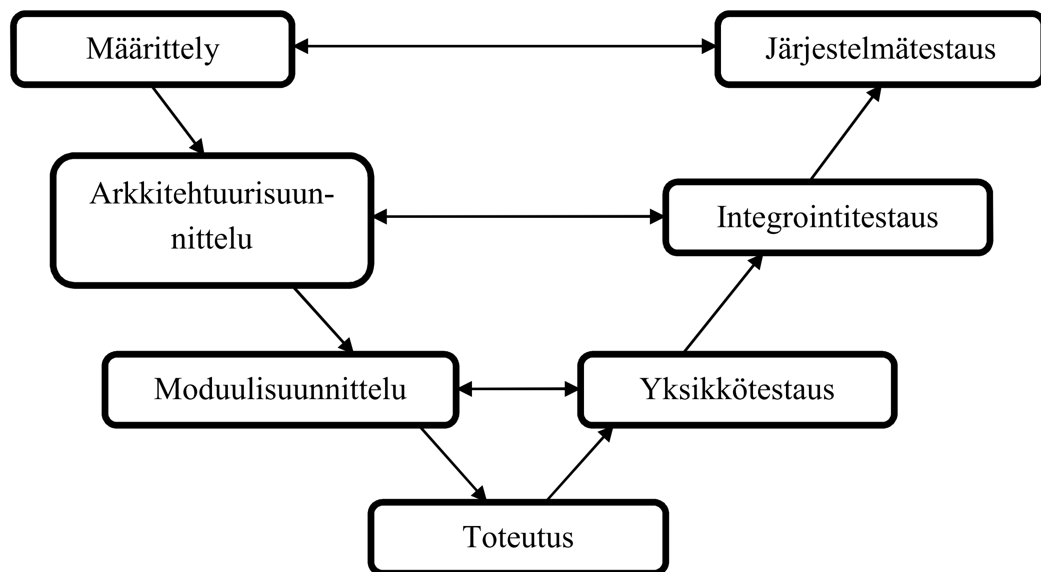
Vuonna 2002 testausta kuvailtiin prosessiksi, joka tähtää testattavan ohjelman laadun mittaamiseen ja parantamiseen. Käytännössä ero aiempiin määritelmiin on se, että testauksella myös parannetaan ohjelman laatua. Laadun parantaminen testauksen avulla onnistuu, jos testaus suoritetaan tarpeeksi aikaisessa vaiheessa tuotteen elinkaaren aikana.

Vaikka ohjelmien monimutkaisuus on kasvanut huimasti viime vuosikymmenien aikana, ovat ohjelmistokehityksen parannukset olleet parhaimmillaan keskinkertaisia. Ohjelmistokehitys on pysynyt pääasiassa työläänä käsityönä, joten ihmisten rajoitteet

rajoittavat myös ohjelmistokehitystä, joka on näin ollen altis inhimillisille virheille. Luotettavan ohjelman tärkeä osa on luottamus siihen, että ohjelma täyttää sille asetetut toiminnalliset ja ei-toiminnalliset vaatimukset. Tämä luottamus saavutetaan testauksella, ja sen osuus koko projektin resurssien käytöstä voi olla jopa 50-75%. [3]

2.2 Testauksen vaiheet, tasot ja automatisointi

Testaus on monivaiheinen prosessi, jota on helppo kuvata perinteisen ohjelmistotuotannon V-mallin avulla. Kuvassa 1 on esitelty V-malli, jossa ohjelmistokehitys on jaettu neljään eri tasoon.



Kuva 1 V-malli [6]

V-mallissa tarkoituksena on suunnitella testaus jokaista ohjelmistokehityksen suunnittelutasoa vasten. Järjestelmätestaus suunnitellaan määrittelyn perusteella, integrointitestaus arkkitehtuurisuunnittelun perusteella ja yksikkötestaus moduulisuunnittelun perusteella. Testaus suoritetaan päinvastaisessa järjestyksessä kuin testien suunnittelu, ja testauksen tuloksia verrataan niitä vastaaviin dokumentteihin. Joissakin V-mallin versioissa on korkeimpana tasona vielä vaatimukset ja sitä vastaava hyväksyntätestaus.

Tehtäessä ohjelmistokehitystä V-mallilla pyritään testausta suunniteltaessa huomaamaan testisuunnittelun pohjana käytetyssä dokumentaatiossa olevat virheet. Tätä kutsutaan ennaltaehkäiseväksi testaukseksi [2]. Koska testaus suunnitellaan ennen toteutusta, on virheet paljon helpompi ja halvempi korjata [4].

Nykyään V-mallia ei juurikaan käytetä ohjelmistokehityksessä, vaan projektit ovat suurilta osin siirtyneet käyttämään ketteriä menetelmiä. Vuonna 2008 erään kyselyn [5] mukaan kyselyyn vastanneista yrityksistä 69% käytti ketteriä menetelmiä ohjelmistoprojekteissaan. Ketterät menetelmät ovat joukko menetelmiä, jotka painottuvat iteratiiviseen kehitykseen ja nopeaan muutoksiin reagointiin. V-malli on

kuitenkin edelleen pätevä esimerkki havainnollistamaan ohjelmistokehityksen eri tasoja ja niiden testausta. Myös ketterissä menetelmissä testauksen suunnittelu ja tulosten tarkastaminen perustuvat dokumentointiin, joka on samalla tasolla kuin kyseessä oleva testaus.

2.2.1 Yksikkötestaus

Yksikkötestauksessa testataan yksittäisiä tai muutamia luokkia kerrallaan. Yksikkötestauksella pyritään löytämään virheitä mahdollisimman aikaisessa vaiheessa ja paikantamaan virhe tiettyyn luokkaan. Usein yksikkötesteissä pyritään testaamaan yhtä luokkaa kerrallaan, jotta testien tekeminen ja virheiden paikantaminen olisi mahdollisimman helppoa. Yksikkötestaus on luonteeltaan lasilaatikkotestausta, ja yksikkötestit tekee yleensä testattavan luokan toteuttaja. Lasilaatikkotestaus tarkoittaa testausta, jossa käytetään hyväksi tietoa ohjelman sisäisestä toteutuksesta [6].

Testattavan luokan eriyttäminen toteutetaan korvaamalla sen riippuvuudet tyngillä (stub) tai luonnostelmilla (mock). Tyngät ja luonnostelmat simuloivat korvattavaa luokkaa. Ero tynkien ja luonnostelmien välillä on se, että luonnostelmien toteutus sisältää oletuksia testin suorituksesta, jolloin ne itsessään ovat osa testin onnistumiskriteerejä [7].

Yksikkötestit toteutetaan usein testipetien (test bed) avulla. Testipeti antaa yksikkötestille ajoympäristön, jolloin tyngät ja luonnostelmat voivat olla osa testipetiä. Testipetiin kuuluu myös testiajuri, joka mahdollistaa testattavan luokan käyttämisen ja tulosten tarkastelun. [6]

Testipetien toteuttamiseen käytetään usein apuna yksikkötestaukseen tarkoitettuja sovelluskehyksiä. Yksikkötestisovelluskehykset ovat automatisoituja, joten yksikkötestien ajaminen on helppoa. Yksikkötestejä ajetaankin kehityksen yhteydessä usein, joten myös niiden suorituksen pitäisi olla nopeaa. Suorituksen nopeus korostuu yksikkötestiläheisissä kehitystavoissa, kuten TDD:ssä (Test-Driven Development). TDD:n perusidea on, että yksikkötesti kirjoitetaan ennen vastaavaa toteutusta [8]. Tällöin testi ohjaa toteutuksen tekoa ja toimii myös dokumentaationa toteutukselle.

2.2.2 Integrointitestaus

Integrointitestauksessa testataan luokkien tai moduulien yhteistoimintaa. Integrointitestauksen päätavoitteena on rajapintojen toimivuuden todentaminen [6]. Integrointitestit ovat usein verrattavissa yksikkötesteihin, ja integrointitestipetien pohja perustuu usein samoihin yksikkötestisovelluskehyksiin kuin yksikkötestipeditkin. Integrointitestit ajetaan yleensä yksikkötestien kanssa, mutta joskus on tarpeen erottaa yksikkötestien ja integrointitestien ajaminen toisistaan integrointitestien hitauden vuoksi.

Integrointi voi edetä kokoavasti (bottom up) tai jäsentävästi (top down). Kokoavassa integroinnissa lähdetään liikkeelle alimman tason luokista ja moduuleista ja

siirrytään vaiheittain ylemmälle tasolle. Jäsentävässä integroinnissa suunta on päinvastainen, eli liikkeelle lähdetään ylimmältä tasolta. [6]

2.2.3 Järjestelmätestaus ja hyväksymistestaus

Järjestelmätestauksessa testataan kokonaista järjestelmää ympäristössä, joka vastaa mahdollisimman tarkasti aitoa käyttöympäristöä. Järjestelmätestaus pohjautuu läheisesti järjestelmän vaatimuksiin eli järjestelmätestit on laadittu määrittelydokumentaation perusteella. Järjestelmätestauksessa testataan myös järjestelmän ei-toiminnalliset ominaisuudet kuten suorituskyky, käytettävyys ja niin edelleen. Järjestelmätestit ovat luonteeltaan mustalaatikkotestausta ja ne suorittaa yleensä erilliset testaajat. Mustalaatikkotestauksessa testitapaukset valitaan ohjelman spesifikaation perusteella tutustumatta ohjelman toteutukseen. [6]

Hyväksymistestit ovat järjestelmätestejä, joiden perusteella asiakas hyväksyy järjestelmän toimituksen. Hyväksymistesteihin ei välttämättä kuulu kaikkia järjestelmätestejä, vaan vain osajoukko niistä.

2.2.4 Regressiotestaus

Kun järjestelmään tehdään muutoksia, esimerkiksi virheenkorjausta, voi muutokset aiheuttaa uusia virheitä muualle järjestelmään. Tämän takia kaikki testit on ajettava uudelleen. Tälläistä uudelleentestausta kutsutaan regressiotestaukseksi, ja sitä tapahtuu kaikilla testauksen tasoilla, yksikkötestauksesta järjestelmätestaukseen. [6]

Regressiotestaus on erittäin kallista manuaalisesti tehtynä, etenkin jos järjestelmästä on toimitettu asiakkaille useita eri versioita. Automaattisella testauksella regressiotestaus kuitenkin helpottuu oleellisesti, koska testit voidaan ajaa helposti uudelleen.

2.2.5 Jatkuva integrointi

Jatkuvalla integroinnilla (Continuous Integration, CI) tarkoitetaan muutosten nopeaa integrointia olemassa olevaan järjestelmään. Käytännössä CI toteutetaan erillisellä palvelimella, joka on varattu ainoastaan jatkuvaan integrointiin. Kehittäjän halutessa integroida muutokset järjestelmään, hän hakee uusimmat koodit omalle koneelleen ja korjaa järjestelmää ja sen testejä kunnes kaikki testit menevät läpi. Tämän jälkeen kehittäjä tallentaa muutokset versionhallintaan. CI-palvelin rakentaa järjestelmän uudelleen ottaen mukaan muuttuneen koodin, ajaa automaattisesti testit ja raportoi tuloksista. [9]

CI-palvelimella muutosten yhteydessä ajettavat testit riippuvat kunkin projektin käytännöistä, mutta pääasiassa ajetaan vähintään kaikki yksikkötestit. Jos muutkin testitasot on automaatisoitu, voidaan nekin ajaa muutosten yhteydessä tai esimerkiksi kerran päivässä. Idea jatkuvassa integroinnissa on se, että mahdolliset muutosten luomat virheet huomattaisiin mahdollisimman aikaisessa vaiheessa, jolloin niiden paikantaminen ja korjaaminen on helppoa.

2.3 Testauksen apuvälineet

Testauksen toteuttamisessa voidaan hyödyntää erilaisia apuvälineitä. Toteutettu testausjärjestelmä on tehty Javalla, joten seuraavaksi keskitytään Java-spesifeihin apuvälineisiin.

2.3.1 Integroitu kehitysympäristö

Integroidulla kehitysympäristöllä (IDE, Integrated Development Environment) tarkoitetaan työkalua, jolla tuotetaan ohjelmakoodia ja joka tarjoaa apuvälineitä ohjelmointiin. IDE on tärkeä ohjelmistokehityksen tuottavuuden tekijä [10]. Kehitysympäristö vaikuttaa toteutuksen tekemisen tuottavuuteen, mutta se voi vaikuttaa myös testauksen tuottavuuteen. Varsinkin yksikkötestaus on hyvin samankaltaista toteutuksen ohjelmoinnin kanssa, joten sille pätee samat integroidun kehitysympäristön antamat edut. Joskus myös automaattiset testit voidaan toteuttaa integroidussa kehitysympäristössä riippuen käytettävästä testausjärjestelmästä, esimerkiksi moduulipohjaisissa testausjärjestelmissä testien toteuttaminen ei juurikaan poikkea tuotteen toteutuksen tekemisestä.

IDE:t tarjoavat lukuisia ohjelmistokehitystä helpottavia ominaisuuksia. Merkittävimpiä ominaisuuksia ovat automaattinen ohjelmiston rakentaminen ja yksikkötestien ajaminen. Automaattinen ohjelmiston rakentaminen tarkoittaa, että IDE kääntää ohjelmakoodin ja tekee muut tarvittavat toimenpiteet, jotta ohjelma on ajettavissa. Yleensä, varsinkin kun Java on kyseessä, automaattinen rakentaminen tapahtuu joka kerta kun tiedosto tallennetaan. Automaattinen kääntäminen voi tapahtua jopa useammin. Tällöin kehittäjä näkee välittömästi käännösaikana havaittavat virheet ja voi korjata ne heti niiden ilmestyessä.

Tuki yksikkötestaukselle on usein IDE:jen perustoiminnallisuutta. Tuella tarkoitetaan yksikkötestien ajamista ja tulosten tarkastelua. Kehittäjä voi napin painalluksella ajaa yksikkötestit, joka on tärkeä ominaisuus varsinkin TDD:tä käytettäessä. IDE voi tukea myös kehittyneempiä ominaisuuksia, kuten hyppäämisen yksikkötestin kohtaan, joka ei mennyt läpi.

IDE:t tarjoavat myös paljon muita tuottavuutta parantavia ominaisuuksia, kuten syntaksikorostuksen, ennakoivan tekstinsyötön, koodin ulkoasun automaattisen muotoilun, koodianalyysityökalujen ajamisen ja ohjelman ajamisen virheidenetsintätilassa (debugging). Ennakoiva tekstinsyöttö on hyödyllinen ohjelmoijalle varsinkin kun käytetään useita kirjastoja, jolloin kaikkia funktio- tai luokkanimiä on hankala muistaa. Ohjelman ajamisella virheidenetsintätilassa tarkoitetaan ohjelman ajamista ympäristössä, jossa ohjelman suoritusta voidaan hallita ja tarkkailla virheiden etsimisen helpottamiseksi.

Suosituimpia Javan kanssa käytettäviä IDE:jä ovat Eclipse [11], Netbeans [12] ja IDEA [13]. Tässä projektissa on käytössä Eclipse, joka on Eclipse Foundationin kehittämä ja avointa lähdekoodia. Eclipsen merkittävin etu muihin kehitysympäristöihin verrattuna on sen suuri liitännäisten (plugin) määrä. Liitännäisillä voidaan laajentaa tai

muokata IDE:n toiminnallisuutta, jolloin siitä saadaan projektin omat tarpeet täyttävä kehitysympäristö.

2.3.2 Sovelluskehykset

Koskimiehen ja Mikkosen mukaan olioperustaiset ohjelmistokehykset ovat luokka-, komponentti- ja/tai rajapintakokoelmia, jotka toteuttavat jonkin ohjelmistojoukon yhteisen arkkitehtuurin ja perustoiminnallisuuden [14]. Sovelluskehyksien ero kirjastoihin verrattuna on niiden käänteisellä käyttötavalla [15]. Kirjastoja käytetään kertomalla kirjastolle mitä tehdään, jolloin ohjelman kontrolli säilyy kirjaston käyttäjällä. Sovelluskehyksiä taas käytetään käänteisesti, eli sovelluskehys kutsuu sen asiakkaan koodia, joten kontrolli on sovelluskehyksellä. Tätä tapaa kutsutaan IoC:ksi (Inversion of Control).

Tässä projektissa testausjärjestelmän sovelluskehys on rakennettu muiden sovelluskehysten päälle. Käytetyt sovelluskehykset ovat Spring ja JUnit, joista JUnit esitellään seuraavaksi tarkemmin.

JUnit

JUnit on yksinkertainen, avoimen lähdekoodin sovelluskehys, jolla voidaan kirjoittaa ja ajaa toistettavia testejä [16]. Se on osa xUnit-sovelluskehysperhettä, jonka sovelluskehykset tarjoavat samankaltaiset toiminnallisuudet eri ohjelmointikielille.

JUnitin ominaisuuksiin kuuluu muun muassa väitteiden (assertion) tekeminen testin odotettujen tuloksien tarkastamista varten, testipetien tekeminen testikoodin uudelleenkäyttöä varten ja testiajureita testien ajamista varten. JUnitilla kirjoitettu yksinkertainen yksikkötesti ja testipeti on esitelty kuvassa 2.

```
 8  public class ExampleJUnitTest {
 9
10      private Multiplier multiplier;
11
12      @Before
13      public void setUp() {
14          multiplier = new Multiplier(5);
15      }
16
17      @Test
18      public void testMultiply() {
19          final int result = multiplier.multiply(6);
20          Assert.assertEquals("Should be multiplied by 5", 30, result);
21      }
22  }
```

Kuva 2 JUnit-esimerkkitesti

Testifunktio määritellään Test-annotaatiolla (@Test), jonka JUnitin testiajuri osaa tulkita ajettavaksi testiksi. Javan annotaatiot ovat lähdekoodiin liitettävää metadataa, joita esimerkiksi kääntäjä voi käyttää tekemään tarkastuksia koodin oikeellisuudesta. Väitteitä taas tehdään käyttämällä Assert-luokan funktioita. SetUp-funktio muodostaa

testipedin, joka on annotoitu Before-annotaatiolla (@Before), jolloin funktio ajetaan ennen jokaista testiä.

Toteutetussa testaussovelluskehyksessä JUnitia on käytetty testiajurina ja testipedin osana. JUnit valittiin käytettäväksi pääasiassa sen yksinkertaisuuden vuoksi, mutta osasyynä oli myös se, että JUnit oli entuudestaan tuttu tekijöille.

2.3.3 Testausautomaatiojärjestelmät

Testausautomaatiojärjestelmillä tarkoitetaan pääsääntöisesti järjestelmä- ja hyväksyntätestausjärjestelmiä. Ne tarjoavat apua vähintään testipetien toteutukseen sovelluskehyksen kautta, mutta voivat tarjota myös kokonaisen pohjan laitteidenhallintaan asti. Testausjärjestelmissä testit perustuvat yleensä tiettyyn periaatteeseen tai malliin, joka määrää miten testi määritellään. Esimerkkeinä malleista ovat muun muassa moduulipohjainen, datapohjainen ja avainsanapohjainen malli.

Moduulipohjaisella sovelluskehyksellä tarkoitetaan kirjastoa, joka tarjoaa testien käyttämää toiminnallisuutta. Yksinkertaisissa testeissä testi kommunikoi suoraan testattavan järjestelmän kanssa, mutta testien monimutkaistuessa huomataan, että testeihin tulee kopioitua koodia. Tämä koodi sijoitetaan sovelluskehykseen, jolloin sitä voidaan uudelleenkäyttää helpommin. [17, 18]

Datapohjaista sovelluskehystä käyttävässä järjestelmässä testin data luetaan erillisestä tiedostosta. Tavoitteena on, että samalla testilogiikalla voidaan suorittaa useita testejä antamalla sille eri data. Yksi hyöty on myös, että uuden testin kirjoittajan ei välttämättä tarvitse osata ohjelmoida testilogiikkaa, vain testin datan määrittely riittää. Myös ylläpidettävyys paranee datapohjaisissa testijärjestelmässä, koska muutosten yhteydessä voidaan muutostyöt jakaa eri henkilöille. [17, 18, 19]

Datapohjaisen sovelluskehyksen suurin ongelma on se, että uutta testilogiikkaa on edelleen hankala tehdä. Tätä ongelmaa yrittävät ratkaista avainsanapohjaiset sovelluskehykset. Näissä sovelluskehyksissä uusia testejä voidaan luoda käyttämällä avainsanoja, jotka kertovat mitä testidatalla tehdään. Avainsanat ovat pieniä, valmiiksi määriteltyjä toiminnallisuuksia, joille voidaan antaa parametreja. Ero moduulipohjaiseen sovelluskehykseen on se, että testien luominen on tehty helpoksi rajoittamalla testilogiikan kirjoittaminen avainsanojen käyttöön. Idea avainsanojen käytössä on se, että testin toteuttajan ei tarvitse olla ohjelmoija luodakseen uutta testilogiikkaa. [18, 19]

Järjestelmätestauksen automatisointi on usein erittäin hankalaa. Yleensä testaustyökaluja tai sovelluskehyksiä esitellään yksinkertaisten esimerkkien kautta, joissa todetaan, että automaattiset testit suorittavat joukon peräkkäisiä toimintoja ilman ihmisen väliintuloa. Esitelmissä mainitaan myös, että koska testejä on usein tarpeen ajaa monta kertaa, automaattisilla testeillä tehdään säästöä jo muutaman testiajokerran jälkeen. Totuus on kuitenkin melko kaukana näistä olettamuksista. Testit ovat harvoin yhtä yksinkertaisia kuin esitettiin, usein ne ovat enemmänkin joukko interaktioita, jotka riippuvat saaduista vastauksista. Automaattisen testausjärjestelmän tekeminen tai käyttöönottaminen monimutkaisille testeille on erittäin hankalaa. Ongelmallista on

myös arvioida, tuoko testauksen automatisointi säästöjä manuaaliseen testaukseen verrattuna, koska automaattinen ja manuaalinen testaus ovat kaksi eri prosessia. [20]

3 TESTAUSJÄRJESTELMÄ

Tässä luvussa esitellään toteutettu testausjärjestelmä, joka on kehitetty yhtä tuotetta varten. Ensin esitellään testattava tuote, sitten testausjärjestelmästä kerrotaan sille asetetut tavoitteet. Tämän jälkeen tutustutaan järjestelmän ympäristöön ja komponentteihin.

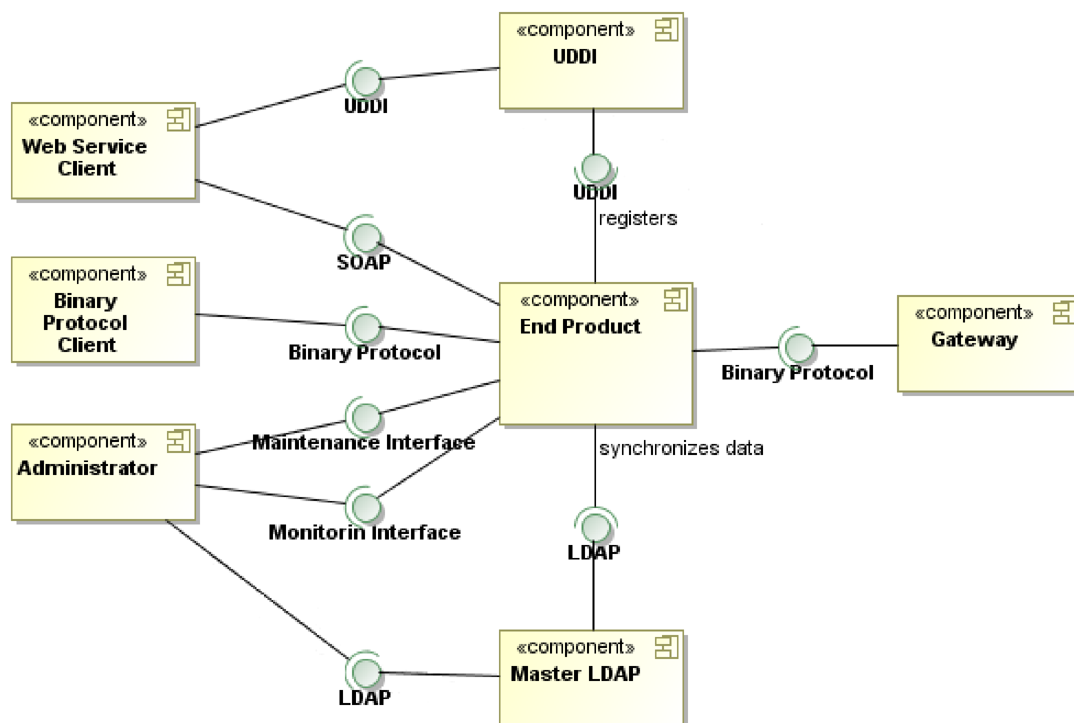
3.1 Testattava tuote

Testattava tuote eli projektin lopputuote (EP, End Product) on palvelin, joka toimii välikätenä asiakasohjelmien ja varsinaisen toiminnallisuuden tarjoavan vanhemman yhdyskäytävän (GW, Gateway) välillä. EP:n tehtäviin kuuluu asiakkaiden lähettämien komentojen suorittaminen tai välittäminen eteenpäin GW:lle, GW:n lähettämien viestien välittäminen asiakkaille ja sellaisen lisätoiminnallisuuden tarjoaminen asiakkaille mitä GW ei jo tarjoa. Kuva 3 esittää EP:n rajapinnat ja sen ympäristön.

Asiakasrajapintaan on toteutettu SOAP-rajapinta johon tulevat viestit on muunnettava GW:n ymmärtämään binäärimuotoon (BP, Binary Protocol). SOAP (Simple Object Access Protocol) on tietoliikenneprotokolla, joka mahdollistaa etäkutsut XML-kielen avulla [21]. XML on rakenteellinen merkintäkieli, jota käytetään sekä tiedon välittämiseen, että dokumenttien kuvaamiseen [22]. Asiakkaat voivat myös vaihtoehtoisesti käyttää suoraan binäärimuotoisia komentoja, jolloin viesteille ei tarvitse tehdä muunnosta. SOAP- ja BP –rajapinnat kulkevat HTTPS-protokollan päällä. Kommunikointi EP:n ja GW:n välillä tapahtuu ainoastaan binäärimuotoisina viesteinä. EP tarjoaa myös huoltorajapinnan, josta voidaan sammuttaa tai käynnistää palvelin uudelleen, sekä valvontarajapinnan, josta laitteiston tilaa voidaan seurata.

Edellä mainittuun SOAP-rajapintaan kuuluu myös yhteys UDDI-rekisteriin. UDDI (Universal Description Discovery and Integration) on palvelurekisteri, josta palvelun käyttäjä saa kaiken tarvitsemansa tiedon palvelun käyttämiseksi [23]. UDDI-rekisteristä asiakas saa muun muassa SOAP-palvelun rajapintamäärityksen.

EP:ssä on myös LDAP-palvelin, joka kommunikoi toisen LDAP-palvelimen kanssa. LDAP (Lightweight Directory Access Protocol) on hakemistopalvelujen käyttöön tarkoitettu verkkoprotokolla [24]. Data on LDAP:ssa avain-arvo-pareina ja ne ovat yleensä järjestetty puumaiseen rakenteeseen. LDAP:n avulla hoidetaan asiakkaiden käyttöoikeuksien hallinta EP:ssä.



Kuva 3 Lopputuotteen rajapinnat ja ympäristö

3.2 Tavoitteet

Päätavoitteena testausjärjestelmälle on testien automatisointi. Testien automatisointi tarjoaa muun muassa seuraavia etuja käsin ajettuihin testeihin verrattaessa: luotettavuus, toistettavuus, uudelleenkäytettävyys ja nopeus [25].

Automaattiset testit suorittavat täsmälleen samat operaatiot joka ajokerralla. Tämä poistaa inhimilliset virheet testejä ajettaessa. Automaatio myös säästää työaikaa, koska testien ajamiseen ei tarvita ihmistä. Tällöin testejä voidaan myös ajaa useammin, jolloin uudet ohjelmistovirheet huomataan nopeammin.

Toistettavuudesta saadaan se hyöty, että testejä voidaan ajaa peräjälkeen, jolloin testien tulokset pitäisi olla joka ajokerralla samat. Jos tulokset vaihtelevat, on virhe joko ohjelmistossa tai testausatomaatiojärjestelmässä.

Uudelleenkäytettävyydellä tarkoitetaan, että testejä voidaan ajaa uudelleen ohjelmiston eri versioilla. Tällöin uusien versioiden ohjelmistovirheet huomataan mahdollisimman nopeasti. Testejä on joskus muutettava eri ohjelmiston versioiden välillä, mutta oletusarvoisesti testit pitäisi olla suunniteltu niin, ettei jokainen pieni muutos aiheuta usean testin muokkausta.

Testien automaattinen ajaminen myös nopeuttaa testien suoritusta, koska ihminen ei ole hidastamassa testien vaatimien toimintojen suoritusta. Testien suuresta määrästä johtuen nopeus on ensiarvoinen ominaisuus. Automaattisella testausjärjestelmälläkin ajettuna kaikkien testien ajaminen voi kestää tunteja. Tällöin testien ajaminen käsin olisi lähes mahdotonta projektille myönnettyjen resurssien puitteissa.

Testausjärjestelmää käytetään myös hyväksyntätesteissä. Testit eivät ole ainoastaan syöte-vaste tyylisiä, yksinkertaisia, toisistaan irrallisia tapahtumia. Testien monimutkaisuuden takia yhdeksi tavoitteeksi asetettiin testien toteuttamisen joustavuus. Testin kirjoittajalla täytyy olla täysi kontrolli testin suorituksesta, eikä testausjärjestelmä saa rajoittaa testin kulkua johonkin tiettyyn muottiin.

Testien toteuttamisen pitäisi olla yksinkertaista, koska testejä on suuri määrä. Epäoleelliset yksityiskohdat pitää piilottaa testin toteuttajalta mahdollisimman hyvin, jotta testin toteuttaja voi keskittyä vain testilogiikkaan.

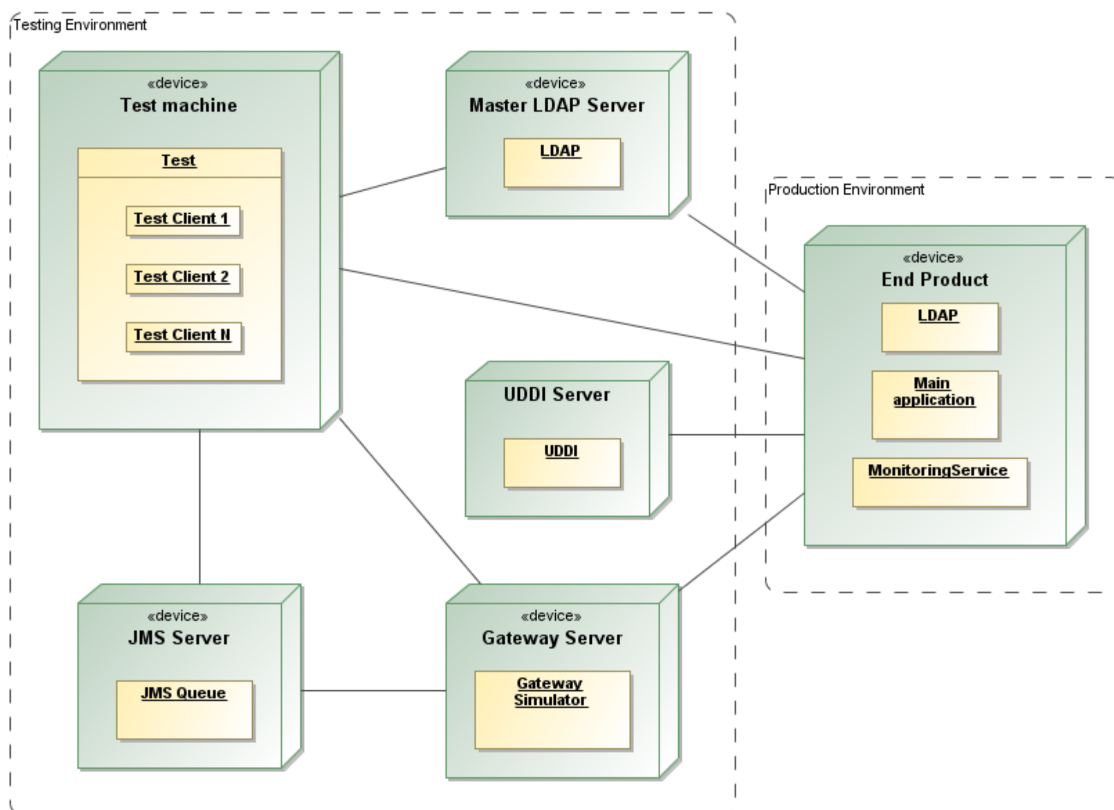
Järjestelmän luotettavuus on tärkeä tavoite, koska testausjärjestelmää käytetään myös hyväksyntätesteissä. Lopputuote voidaan hyväksyä vain, jos testien luotettavuus on korkealla tasolla. Luotettavuuteen kuuluu suurena osana testien tulosten tarkastaminen. On siis voitava selvittää, mitä toimintoja testi suoritti ja mitkä olivat näiden toimintojen tulokset.

3.3 Ympäristö ja komponentit

Testausautomaatiojärjestelmä voidaan jakaa kahteen ympäristöön: testausympäristöön ja lopputuoteympäristöön. Lopputuoteympäristöön kuuluu ainoastaan EP ja sen komponentit eli itse testattava tuote. Lopputuoteympäristö on luonteeltaan musta laatikko testin näkökulmasta. Testi ei pääse käsiksi lopputuoteympäristön komponentteihin, muuten kuin kuvassa 3 esiteltyjen rajapintojen kautta.

Testausympäristöön kuuluu EP:n kanssa keskustelevat ja muut testausjärjestelmän komponentit, ja se rakentuu lopputuoteympäristön ympärille. Testausympäristö on luonteeltaan lasilaatikko testin näkökulmasta. Tavoitteena on luoda oikean käyttöympäristön kaltainen ympäristö, joka on hallittavissa mahdollisimman hyvin, jolloin testit voivat määrittää oman ajoympäristönsä. Kuva 4 esittää testausjärjestelmän ympäristöt ja komponentit.

Lopputuoteympäristön komponentit vastaavat oikeassa ympäristössä ajettavia komponentteja. Testausympäristön komponentit eivät kuulu toimitettavaan tuotteeseen ja näin ollen ne voidaan korvata tyngillä tai luonnostelmilla. Näin on tehty vain GW:n osalta, joka on korvattu ohjattavalla simulaattorilla. Tämä mahdollistaa joustavamman testauksen EP:n ja GW:n väliselle rajapinnalle.



Kuva 4 Testiautomaatiojärjestelmän ympäristö ja komponentit

Testausympäristöön kuuluu testikone (Test machine), LDAP-palvelin (Master LDAP Server), JMS-palvelin (JMS Server), UDDI-palvelin (UDDI Server) ja GW-palvelin (Gateway Server). Testikoneessa ajetaan testiä (Test), joka ohjaa koko testausärjestelmää ja testin kulkua. Riippuen testistä siihen voi kuulua yksi tai useampi testiasiakas. Monen testiasiakkaan tapauksessa kaikkia asiakkaita ajetaan samalla koneella.

LDAP-palvelimen toiminta testeissä on yksinkertaista, joten ei ole nähty tarpeelliseksi korvata sitä tyngällä tai luonnostelmalla. Testausympäristön LDAP-palvelin vastaa oikeassa ympäristössä käytettävää palvelinta sillä erotuksella, että testillä on oikeudet muuttaa kaikkea LDAP-palvelimen dataa. LDAP-palvelimen tietojen käsittely testin aikana on sen avain-arvoparien muuttamista. Nämä muutokset synkronoidaan EP:n LDAP-palvelimelle, jota EP:n liiketoimintalogiikka käyttää.

GW-simulaattori keskustelee EP:n kanssa kuten oikea GW. Testi pystyy ohjaamaan GW:tä suoraan GW:seen rakennetun SOAP-rajapinnan avulla. Suoraa yhteyttä käytetään suurimmaksi osaksi kertomaan GW:lle, miten sen pitäisi vastata viesteihin, jotka tulevat EP:ltä. Testin ja GW:n välisellä yhteydellä on myös muita käyttötarkoituksia, kuten eri tilojen asettaminen päälle GW:ssä.

EP:ltä GW:lle menevien viestien välityksessä testikoneelle on käytössä JMS-palvelin. JMS (Java Message Service) on ohjelmointirajapinta viestien välittämiseen [26]. Saadessaan viestin GW laittaa sen JMS-jonoon, johon viestit kerääntyvät. Testi voi halutessaan hakea viestit jonosta ja tarkastaa ne.

3.4 Jatkuva integrointi testausjärjestelmässä

Tässä projektissa jatkuvan integroinnin ympäristöä (CI, Continuous Integration) käytetään orkesteroimaan testausta muiden tehtävien lisäksi. Jatkuva integrointi tarkoittaa sitä, että koodimuutosten jälkeen muutokset integroidaan automaattisesti olemassa olevaan koodimassaan. Integrointiin kuuluu muun muassa koodin kääntäminen ja testaus. Käytetty CI-ohjelmisto on Hudson, joka on vapaata lähdekoodia ja helppokäyttöinen [27].

Tavallisessa kehityksessä Hudson tarkkailee versionhallintaa ja huomatessaan muutoksen, kääntää ja ajaa lopputuotteen yksikkötestit automaattisesti. Onnistuneiden yksikkötestien jälkeen Hudson rakentaa lopputuotteelle valmiin jakelupaketin.

Hudsonilla on suuri rooli testausjärjestelmän käytössä. Testien ja GW-simulaattorin kääntämisen lisäksi Hudson ajaa kaikki testausjärjestelmän testit. Kuva 4 esitetty testikone on itse asiassa Hudson-palvelin. Hudsonin näkökulmasta testien ajamiseen kuuluu seuraavat vaiheet: Testikoodin kääntäminen, lopputuotteen jakelun levittäminen lopputuoteympäristöihin, GW-simulaattorin kääntäminen ja levittäminen GW-palvelimille, sekä testien ajaminen ja niiden tuloksien kerääminen. Seuraavaksi tarkennetaan mitä eri vaiheet tarkoittavat.

Hudson kääntää ja testaa testausautomaatiojärjestelmän sovelluskehyksen ja sen varaan rakennetut testit. Tätä koodia ei tarvitse paketoida erikseen, koska koodi suoritetaan Hudsonilla.

Lopputuotteen jakelulla lopputuoteympäristöihin tarkoitetaan sitä, että lopputuotteen paketti kopioidaan ja asennetaan EP-koneille. EP-koneita on useampi, koska ne konfiguroidaan testejä varten eri tavoilla. Myös GW-simulaattori käännetään ja levitetään GW-koneille. GW-koneita on pääsääntöisesti yksi yhtä EP-konetta kohden, koska GW on tiukasti sidoksissa EP:n tilaan. Tällöin testejä on mahdollista ajaa useampia samalla kertaa eri ympäristöissä.

Testien ajamisella tarkoitetaan kaikkien testien suorittamista Hudson-koneella. Ajettava testi määrittää itse, mitä koneita vasten testi ajetaan. Testien tulokset kerätään talteen ja niitä voidaan tarkastella suoraan Hudsonin web-palvelusta. Tulokset arkistoidaan, joten myös vanhempia testituloksia on mahdollista tarkastella.

3.5 Testaussovelluskehys

Testausjärjestelmän testit käyttävät toteutuksissaan niille rakennettua sovelluskehystä. Tässä luvussa esitellään sovelluskehyksen suunnitteluperusteet ja arkkitehtuuri.

3.5.1 Suunnitteluperusteet

Sovelluskehyksen suunnitteluperusteet pohjautuvat luonnollisesti testausjärjestelmän tavoitteisiin. Päätavoitteena on, että testit ovat automatisoituja. Sovelluskehyksen kannalta tämä vaatimus kiteytyy siihen, että sovelluskehyksen komponentit eivät saa käyttää käyttäjän syötettä vaativaa koodia.

Testien toteuttamisen joustavuus yhdessä testien helpon toteuttamisen kanssa on hankala vaatimus, koska ne ovat osaksi ristiriidassa keskenään. Vaihtoehtoina on joko suosia toista vaatimusta toisen kustannuksella, tai rakentaa sellainen sovelluskehys, jossa kaikkien testien kaikki eri toiminnot on otettu huomioon. Koska jälkimmäinen vaihtoehto on hyvin haastava toteuttaa, päätettiin sovelluskehys suunnitella ensimmäisen vaihtoehdon mukaisesti.

Jotta joustava testien toteutus olisi mahdollista, täytyy sovelluskehys rakentaa pienistä komponenteista, joita testin toteuttaja voi käyttää haluamallaan tavalla. Komponentit täytyy myös rakentaa laajennettaviksi, jotta testin toteuttamisen aikana mahdollisesti esiin nousevat uudet toiminnalliset vaatimukset saadaan toteutettua mahdollisimman nopeasti sovelluskehykseen.

Luotettavuus sovelluskehyksen kannalta tarkoittaa, että sovelluskehyksessä ei ole ohjelmistovirheitä. Ohjelmistovirheiden määrää voidaan pienentää hyvin tunnetuilla ohjelmistotekniikan menetelmillä, joista testaus tärkeä osa. Sovelluskehys on siis testattava, vaikka se ei ole lopputuotteessa ajettavaa koodia.

Sovelluskehyksen testauksen helpottamiseksi sen komponentit on suunniteltava testausta silmällä pitäen. Testattavuuden suuri aputekijä on vastuiden jakaminen (Separation of Concerns) [28]. Käytännössä tämä tarkoittaa, että luokilla on vain yksi, tarkasti rajattu tehtävä. Monimutkaisempi toiminnallisuus saadaan aikaan käyttämällä näitä luokkia yhdessä. Jos luokka käyttää toisia luokkia, sen tarvitsee vain tietää *mitä* nämä luokat tekevät, ei sitä, *miten* ne sen tekevät. Tämän takia luokat on hyvä piilottaa rajapintojen taakse, jolloin luokka näkee vain toisen luokan rajapinnan, ei sen toteutusta. Rajapintojen avulla on mahdollista suorittaa riippuvuuksien injektio (DI, Dependency Injection), jolloin luokalle annetaan sen riippuvuudet. Tällä tavalla luokkaa testattaessa voidaan sille antaa rajapinnan toteuttava tynkä tai luonnostelma.

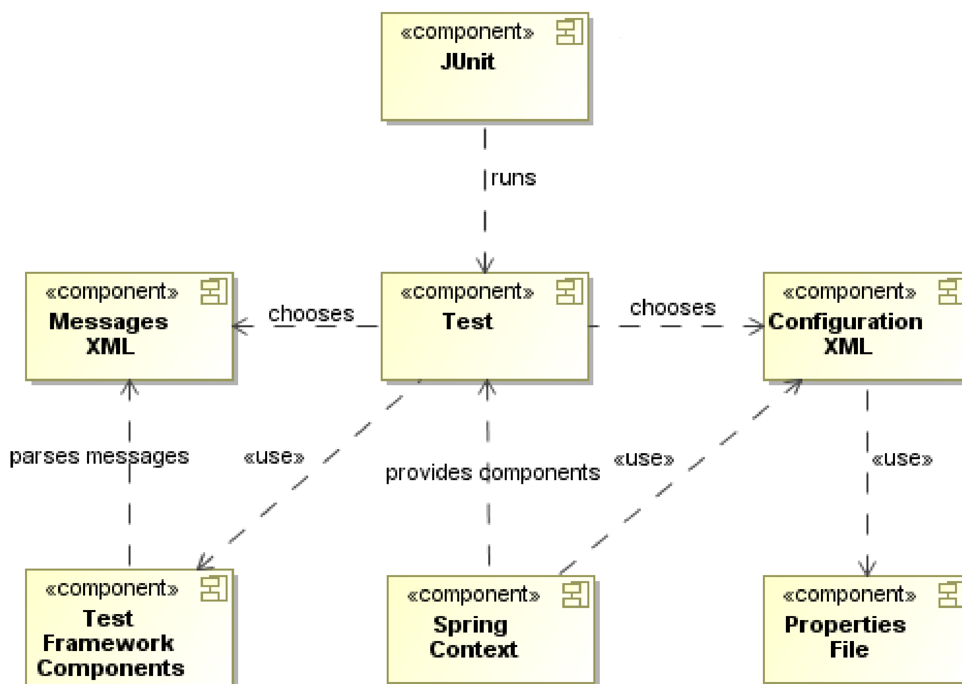
3.5.2 Arkkitehtuuri

Sovelluskehyksen arkkitehtuurissa konfiguroinnin olennaisena osana on Spring Framework –sovelluskehys. Spring Framework on sovelluskehys, joka koostuu useista moduuleista. Nämä moduulit rakentuvat Core-moduulin päälle. Tärkein Core-moduulin tehtävä on tarjota sovellukselle IoC-säiliö, jolla voidaan konfiguroida sovelluksen komponentit ja hallita komponenttien elinkaarta. Spring Coren päätavoitteet ovat, että sovelluskehystä on helppo käyttää, asiakkaan koodi ei riipu Spring Coresta ja Spring Core ei kilpaile muiden sovelluskehysten kanssa [29].

Spring Corea käyttävä sovellus voidaan konfiguroida kahdella eri tavalla, joko Java-koodissa tai perinteisemmin XML-konfigurointitiedostossa. Konfiguroinnissa määritellään sovelluksen komponentit ja niiden riippuvuudet. Riippuvuuksien injektointi on yksi tapa toteuttaa IoC. Konfiguroinnissa määritellään myös komponenttien elinkaari, eli koska komponentti luodaan ja koska se poistuu käytöstä. Spring Corea käyttävän ohjelman liiketoimintalogiikan toteuttava koodi ei usein riipu Spring Coresta sen käyttämän konfigurointitavan ansiosta.

Koska Spring Framework on rakennettu modulaarisesti, toimii se helposti muiden sovelluskehysten kanssa. Spring Core on kevyt ja huomaamaton sovelluskehys muiden sovelluskehysten kannalta, koska asiakkaan koodi on mahdollista rakentaa niin, että se ei ole riippuvainen Spring Coresta. Muut Spring Frameworkin moduulit eivät välttämättä ole yhtä huomaamattomia ja tämä on tärkein syy modulaarisuudelle. Testaussovelluskehyksessä on käytetty ainoastaan Spring Core –moduulia testien konfigurointiin

Testien ajamisen automatisoiminen hoidetaan JUnit-sovelluskehyksen avulla. JUnit tarjoaa myös testiraporttien generoinnin. Sovelluskehyksen komponenttien luonti on eriytetty testistä, komponenttien luonti ja konfigurointi tapahtuu Springin avulla erillisillä XML- ja property-tiedostoilla. Kuva 5 esittää yksittäisen testin ajoympäristön.



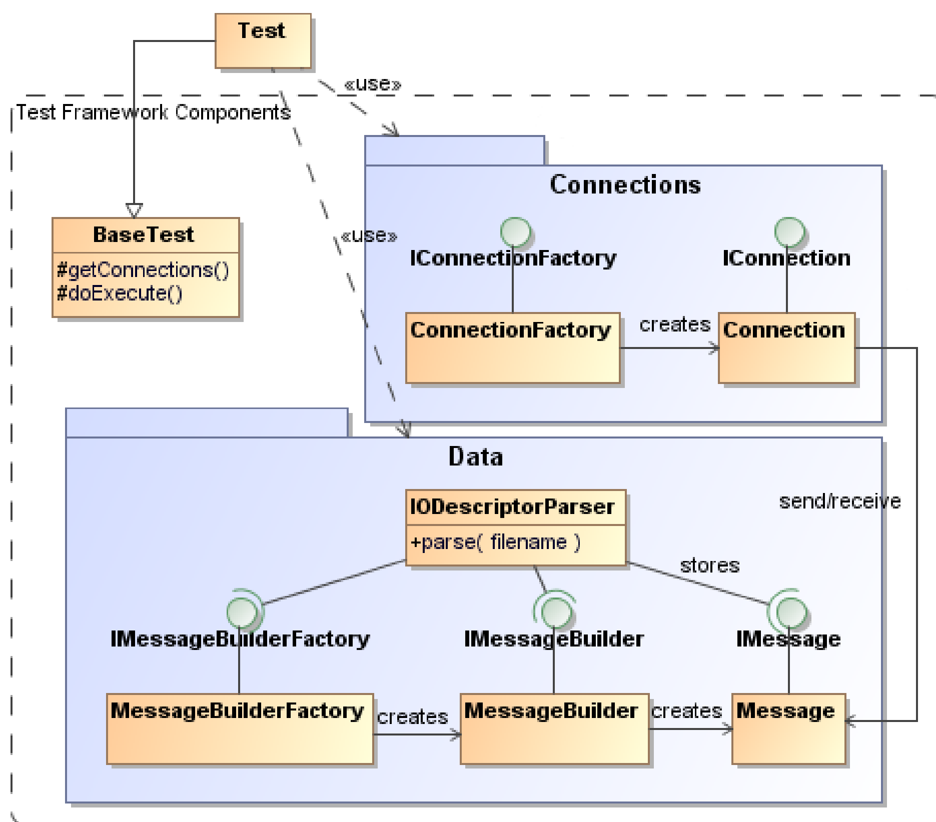
Kuva 5 Yksittäisen testin ajoympäristön arkkitehtuuri

Tavoitteena on ollut eriyttää testin resurssit ja komponenttien konfigurointi itse testin logiikasta. Testiluokassa (Test) määritellään testin logiikka. Testiluokan käyttämät sovelluskehyksen komponentit (Test Framework Components) määritellään Springin XML –konfigurointitiedostossa (Configuration XML). Konfigurointitiedosto taas käyttää Javassa yleisesti käytettyä property-tiedostoa (Properties File) lukemaan yksittäisiä arvoja komponenttien konfigurointiin, kuten testiympäristössä olevien koneiden IP-osoitteet.

Testin käyttämät viestit määritellään erillisessä XML-tiedostossa (Messages XML), jonka sovelluskehys osaa jäsentää testin käyttöön. Testi valitsee itse sekä konfigurointitiedoston että viestitiedoston. Tällä järjestelyllä on pyritty vähentämään muutostarpeita itse testin logiikkaan, esimerkiksi testiympäristön muutokset pitäisi pystyä hoitamaan konfigurointitiedostoa muuttamalla. Tämä malli noudattaa

datapohjaisen testausmallin periaatetta, jossa kaikki mikä on mahdollisesti muuttuvaa, eriytetään testin logiikasta.

Sovelluskehyksen arkkitehtuuri koostuu kolmesta osasta. Kuva 6 esittää nämä osat korkealta tasolta. BaseTest on kantaluokka kaikille testeille ja tarjoaa testin raportoinnin ja muut testien ajamisen kannalta yhteiset osat. Datankäsittely (Data) tarjoaa viestien jäsentämisen ja vertailun. Yhteydet (Connections) tarjoaa yhteydet testausjärjestelmän osiin.



Kuva 6 Sovelluskehyksen arkkitehtuuri

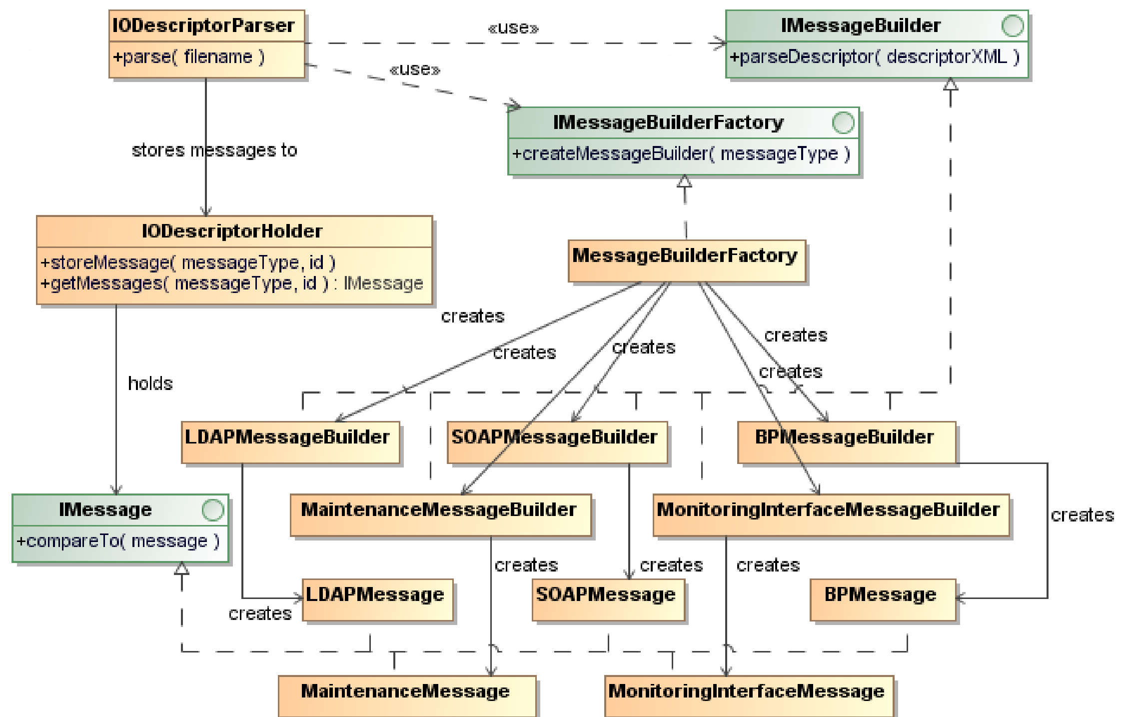
3.5.2.1 Datankäsittely

Datankäsittely tapahtuu pääasiassa IODescriptorParserin kautta. Sen rajapinta muodostuu yhdestä funktiosta: parse. Parse-funktio jäsentää sille annetun tiedoston viesteiksi ja palauttaa viestit IODescriptorHolder-oliossa. IODescriptorHolderista viestit voidaan hakea niiden viestityypin ja niille annetun id:n perusteella. Kuva 7 esittää datankäsittelyn arkkitehtuurin.

IODescriptorParserin sisäinen toiminta perustuu IMessageBuilderFactoryn ja sen tuottamien IMessageBuilderien käyttöön. IMessageBuilder toteutuksia on yhtä monta kuin eri viestityyppejä, yksi MessageBuilder tuottaa yhdenlaisia viestejä.

IODescriptorParserin parse-funktion suoritus kulkee kuvan 8 mukaisesti. Ensin IODescriptorParser jäsentää annetun XML-tiedoston Javan standardikirjaston avulla java.w3c.dom.Document-olioksi. IODescriptorHolder luodaan säilyttämään jäsennetyt viestit. Jäsennetystä XML-tiedostosta käydään läpi kaikki iodescriptor-elementit ja luodaan MessageBuilder riippuen iodescriptor-elementin viestityypistä.

MessageBuilderille annetaan iodescriptor-elementti ja MessageBuilder jäsentää kaikki viestit kyseisestä elementistä. Viestit palautetaan IODescriptorParserille, joka säilöö ne IODescriptorHolderiin. Lopuksi IODescriptorHolder palautetaan parse-funktion kutsujalle, joka saa haettua viestit IODescriptorHolderista.



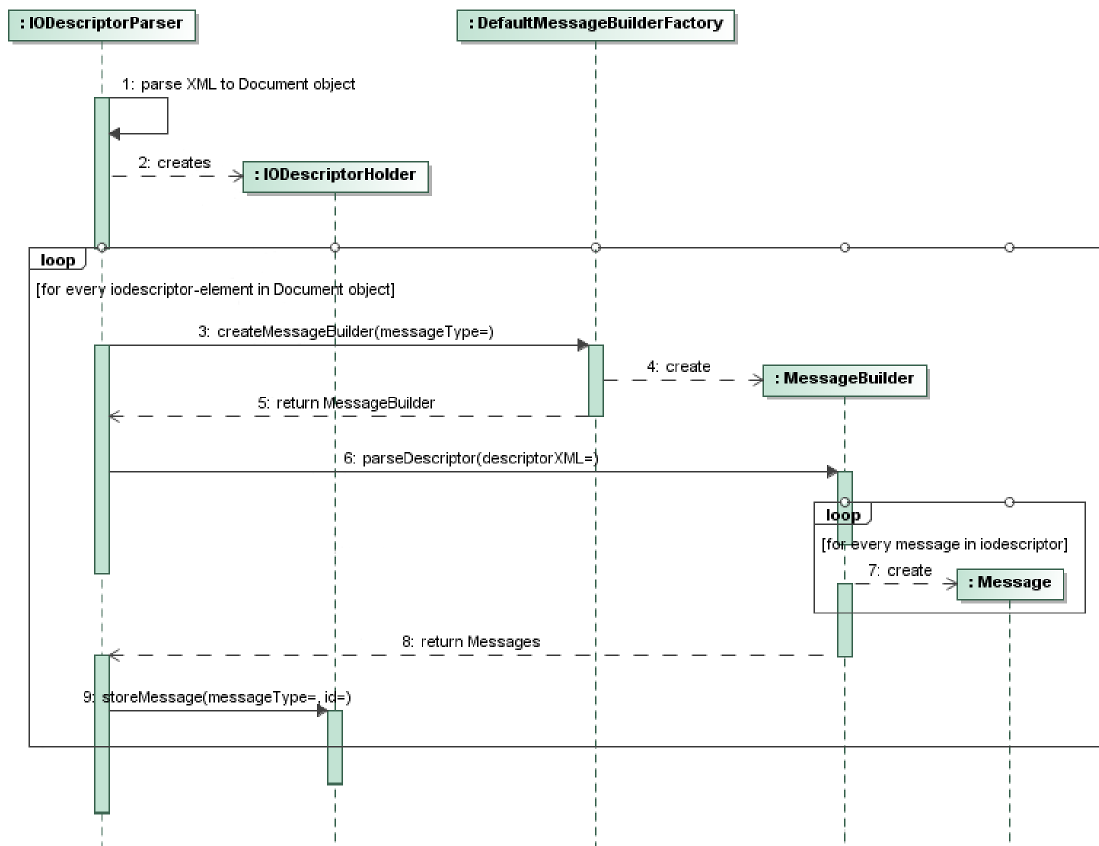
Kuva 7 IODescriptorParserin sisäinen toteutus

IODescriptorParserin jäsentämät viestejä määrittävät XML-tiedostot eli IO-kuvaajat (IO-descriptor) noudattavat kaikki samaa rakennetta: ne koostuvat iodescriptor-elementeistä. Yksi iodescriptor-elementti pitää sisällään input- ja output –osiot. Näihin osioihin määritellään yksi tai useampi viesti. Ideana on määritellä input-osioon testin lähettämät viestit ja output-osioon oletetut EP:n palauttamat viestit.

Iodescriptor-elementeillä on myös type-attribuutti, joka määrittää viestien yhteystyypin. Alkuperäisenä ideana on ollut, että vain yksi iodescriptor-elementti määritellään yhtä yhteystyyppiä varten. Kuitenkin sovelluskehystä tehtäessä jo aikaisessa vaiheessa huomattiin, että valittu ratkaisu ei käytännössä toiminut testien ollessa laajoja. Tämän takia sallittiin useampi samantyyppinen iodescriptor-elementti yhdessä IO-kuvaajassa.

Viestien määrityksellä erillisiin tiedostoihin on kyse yhdestä datapohjaisen-testauksen periaatteesta. Viestien sisältö, eli data, on erillään testin logiikasta. Tällöin jos toinen muuttuu, ei välttämättä tarvitse muuttaa toista, mikä parantaa testien ylläpidettävyyttä. Myös testilogiikan uudelleenkäyttö onnistuu paremmin, kun data on erotettu logiikasta. Testijärjestelmää ei ole kuitenkaan suunniteltu silmälläpitäen logiikan uudelleenkäyttöä, vaikka se mahdollista onkin. Sovelluskehys ei sellaisenaan tarjoa tukea logiikan uudelleenkäytölle, mutta sovelluskehyksen varaan on mahdollista rakentaa uudelleenkäytettäviä komponentteja testien toteuttamisen helpottamiseksi.

Erillisillä data-tiedostoilla on myös pyritty helpottamaan testien hyväksymistä ja katselmointia. Viestien sisältö pystytään tarkastamaan katsomatta testin suorittavaa koodia, mikä helpottaa viestien tarkastamista.



Kuva 8 IODescriptorParserin parse-funktion suoritus

IO-kuvaajien XML-skeema

IO-kuvaajien rakenne on määritelty XML Schema –kielellä. XML Schema valittiin määrittelykieleksi sen yksinkertaisuuden ja helppokäyttöisyyden takia. Vaikka XML Schema ei ole riittävän ilmaisuvoimainen kaikkien sääntöjen määrittelemiseen IO-kuvaajissa, on se tarpeeksi kattava, jotta ilmaisuvoiman heikkoudesta ei muodostu ongelmaa. Ne säännöt, joita XML Schema -kielellä ei voida tarkastaa, tarkastetaan koodissa.

XML Schema on tuettuna valmiiksi Javan standardikirjastoissa, joten sen käyttö XML-tiedostojen jäsentämisen yhteydessä tarkastamaan tiedostojen oikeellisuus on helppoa. Kuvassa 9 esitetään IO-kuvaaja esimerkkinä.

```
 1 <iodescriptor id="descriptorId" type="binaryProtocol">
 2     <inputs>
 3         <bpInput>
 4             ...
 5         </bpInput>
 6         <bpInput>
 7             ...
 8         </bpInput>
 9     </inputs>
10     <outputs>
11         <bpOutput>
12             ...
13         </bpOutput>
14         <bpOutput>
15             ...
16         </bpOutput>
17     </outputs>
18 </iodescriptor>
```

Kuva 9 IO-kuvaaja esimerkkitiedosto

IO-kuvaaja koostuu iodescriptor-elementeistä, jotka kuvaavat loogisesti yhden viestienvaihtosekvenssin. Iodescriptor-elementti taas koostuu inputs- ja outputs-elementeistä, joihin määritellään viestit. Inputs-elementtiin määritellään syötteet ja outputs-elementtiin määritellään odotetut vasteet. Tilanne voi myös olla toisinpäin riippuen yhteyden tyypistä, esimerkiksi GW:ltä lähtevät viestit määritellään outputs-elementtiin.

Inputs- ja outputs-elementtien viestit eroavat niiden käyttötarkoituksen takia toisistaan. Inputs-viestejä käytetään sellaisenaan syötteinä, kun taas outputs-viestejä käytetään vain vertailemaan niitä vastaanotettuihin viesteihin. Käytännössä inputs- ja outputs-viestien määritys on usein samankaltaista, jos viestejä vertaillaan yksi-yhteen-periaatteella. Joskus on kuitenkin tarpeellista verrata viestejä joustavammin, esimerkiksi niin, että viestin tietty kokonaislukuarvo saa olla maksimissaan 5. Tällöin vertailuviesti määritellään eri tavalla riippuen viestin tyypistä.

Viestien määrityksen rakenne riippuu paljon viestin tyypistä. SOAP-viesteillä määritys on helppoa, koska SOAP-viestit ovat jo valmiiksi XML:ää. Binary Protocol –viesteillä määritys on taas totetettu avain-arvo pareilla, jotka kuvautuvat suoraan Java-koodin oliorakenteiksi. XML-rakenne viestien vertailuille riippuu myös paljon viestien tyypistä. Yhtenäistä rakennetta viestien vertailuille ei ole olemassa.

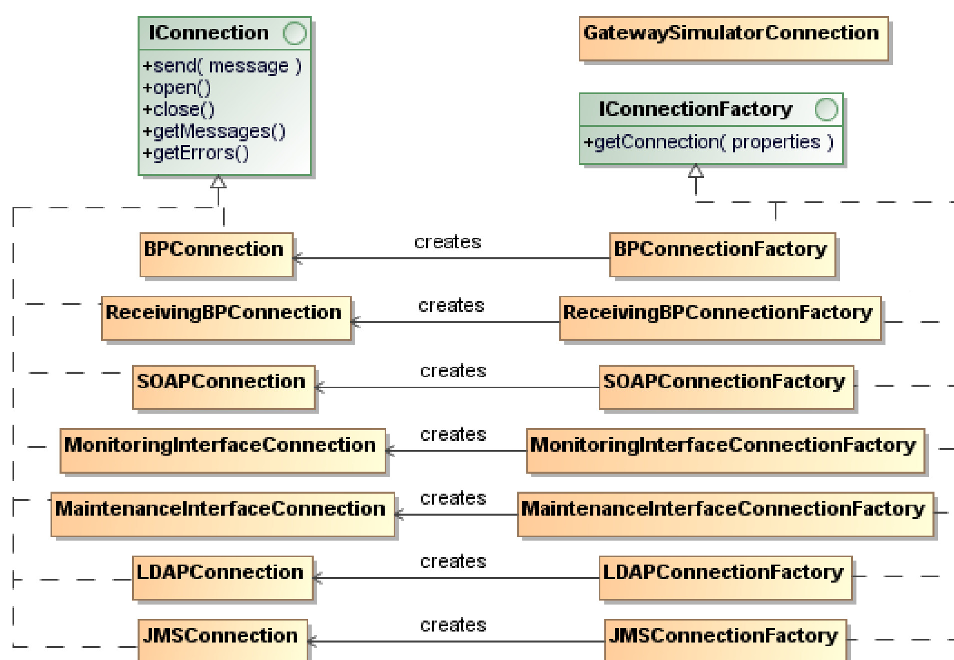
IO-kuvaajien uudelleenkäyttö

Testien suunnittelun aikana huomattiin, että osaa viesteistä käytettiin useammin kuin kerran testeissä. Jotta vältettäisiin viestimääritysten kopiointi, päätettiin toteuttaa IO-kuvaajien uudelleenkäyttömahdollisuus. Uudelleenkäyttö toimii XInclude-toiminnallisuuden avulla, joka on yleiskäyttöinen tapa koostaa XML-dokumentteja pienemmistä dokumenteista. Javan standardikirjastot tukevat suoraan XInclude-toiminnallisuutta, joten tämän toteutus oli suoraviivaista.

Toteutuksen aikana huomattiin myös, että osaa viesteistä käytettiin huomattavan monta kertaa, lähes joka testissä. Koska osa näistä viesteistä on sisällöltään riippuvaisia testin tai testattavan tuotteen tilasta, ei pelkkä XML-dokumenttien koostaminen ole riittävä keino välttämään kopiointia. Tämän takia toteutettiin yleisimmille viesteille helppo tapa määrittää ne IO-kuvaajiin. Yleisimmillä viesteillä on tietty elementti, johon määritellään viestin muuttuva sisältö atribuutteina, tällöin koko viestin määritys on tehtävissä yhdellä rivillä. Näitä viestejä kutsutaan oletusviesteiksi.

3.5.2.2 Yhteydet

Yhteyksien keskeisimmät rajapinnat ovat IConnection ja IConnectionFactory. IConnectien rajapinta kuvaa yhtä yhteyttä, esimerkiksi loppukäyttäjän SOAP-yhteyttä EP:lle. IConnectionFactory määrittää rajapinnan yhteyksien luonnille. Kuva 10 esittää yhteyksien luokkakaavion.



Kuva 10 Yhteydet-luokkakaavio

IConnection-rajapinta määrittää funktiot yhteyden avaamiselle ja sulkemiselle, viestin lähettämiselle sekä vastaanotettujen viestien tai tapahtuneiden yhteysvirheiden palauttamiselle.

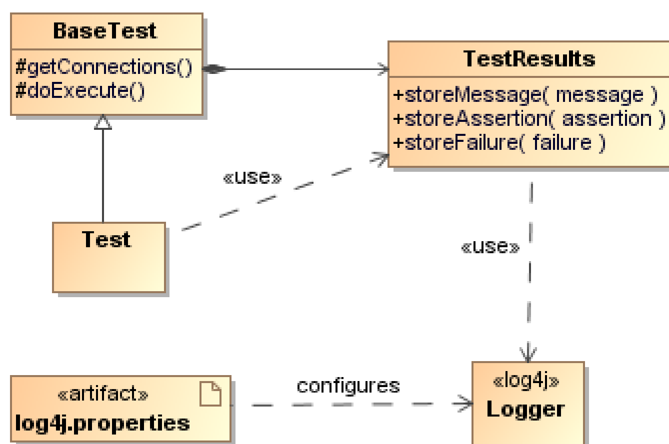
Jokaiselle yhteydelle on yksi toteutus, useampi yhteys kuitenkin käsittelee saman tyyppisiä viestejä. Esimerkiksi BPConnection on tarkoitettu EP:lle lähetettäville komennoille ja niiden vastauksille, kun taas ReceivingBPConnection on tarkoitettu EP:ltä tuleville informaatio- ja dataviesteille.

IConnectionFactory määrittää funktion yhteyden luomiselle. Funktiolle annetaan parametrina Properties-tyyppinen olio, johon testin toteuttaja määrittää luotavan yhteyden asetukset.

GatewaySimulatorConnection on erillinen apuluokka ohjaamaan GW:tä. Koska testeissä on käytössä vain yksi GW simulaattori ja sen yhteydet EP:lle ovat vakiot, ei GatewaySimulatorConnectionille tarvita erillistä tehdasta, vaan se toimii Singleton-suunnittelumallilla [30]. Singleton-suunnittelumallissa oliosta on olemasa vain yksi instanssi johon pääsee käsiksi mistä tahansa ohjelman osasta.

3.5.2.3 Raportointi

Raportointiin kuuluu testin etenemisen ilmoittaminen, tulosten tarkastaminen ja tulosten raportointi. Kuva 11 esittää raportointiin kuuluvat luokat.



Kuva 11 Raportoinnin luokkakaavio

BaseTest on kantaluokka kaikille testeille. Se toteuttaa kaikille testeille yhteiset asiat, kuten ennen testiä EP:n käynnistämisen uudelleen ja testin jälkeen tulosten raportoinnin JUnitin kautta. BaseTestistä periytyvän testin on toteutettava getConnections- ja doExecute-funktiot. Funktio getConnectionsin on palautettava käytetyt yhteydet, jolloin BaseTest voi sulkea ne testin jälkeen ja tarkastaa ne virheiden varalta. Funktio doExecuteen toteutetaan itse testin suoritus.

Testin eteneminen raportoidaan TestResults-luokan storeMessage-funktiolla, jolle annetaan vapaavalintainen ilmoitus. Testin tulosten tarkastamisessa käytetään storeAssertion- ja storeFailure-funktioita. Epäonnistuneen vertailun tai storeFailuren tapauksessa testin lokissa näytetään ilmoitus epäonnistumisesta, mutta testin suoritusta ei keskeytetä. Vertailun tulos tallennetaan tietorakenteeseen tarkastettavaksi testin suorituksen jälkeen.

Koska testien suoritus ei keskeydy, eivät testit toimi automaattisesti niin sanotulla fail-fast-periaatteella. Fail-fast periaatteella testien yhteydessä tarkoitetaan tapaa keskeyttää testin suoritus välittömästi ensimmäisen virheen sattuessa, koska myöhemmät testin vaiheet usein riippuvat aiemmasta jo epäonnistuneesta kohdasta. Syy fail-fast-periaatteen käyttämättömyyteen on pitkäkestoiset testit, joiden vaiheet eivät riipu edellisistä vaiheista. Tällöin testi voidaan ajaa loppuun ja kerätä kaikki epäonnistuneet vertailut, eikä vain ensimmäistä. Tästä syystä fail-fast-periaatetta ei

pakotettu testeihin, vaan testin toteuttaja voi rakentaa testiin mekanismeja lopettamaan testin suoritus ajoissa, jos se testin kannalta on järkevää.

Suorituksen eteneminen ja epäonnistuneet vertailut tallennetaan lokiin log4j-kirjaston tarjoaman Loggerin avulla. Loggerin konfigurointi on erillisessä log4j.properties-tiedostossa. Tavallinen konfigurointi tallentaa lokin tiedostoon aikaleimoilla varustettuna ja tulostaa lokin konsoliin.

3.6 Testausjärjestelmän käyttö

Tässä luvussa esitellään testausjärjestelmän käyttö esimerkkitestin avulla. Aluksi lähdetään liikkeelle lopputuotteelle asetetuista vaatimuksista ja vaatimuksien perusteella suunnitellusta testistä. Testille luodaan toteutus käyttäen testaussovelluskehystä. Lopuksi testi ajetaan testausjärjestelmässä ja tarkastetaan tulokset. Testin ajosta esitetään sekä onnistunut että epäonnistunut ajo.

3.6.1 Vaatimukset ja suunniteltu testi

Esimerkkitestiin liittyviä vaatimuksia on kaksi:

1. Lopputuotteen on tuettava kahta yhtäaikaista asiakasyhteyttä.
2. Kahden yhtäaikaisen yhteyden jälkeen tulevat yhteyspyynnöt on hylättävä syyllä: ”Maximum number of connections reached”.

Näiden vaatimusten perusteella on laadittu testi:

1. Muodosta yhteys onnistuneesti ensimmäisellä asiakkaalla
2. Muodosta yhteys onnistuneesti toisella asiakkaalla
3. Yritä muodostaa yhteys kolmannella asiakkaalla. EP vastaa: ”Maximum number of connections reached”
4. Katkaise ensimmäisen asiakkaan yhteys.
5. Muodosta yhteys onnistuneesti kolmannella asiakkaalla.

3.6.2 Testin toteutus

Testin esimerkkitoteutus on esitetty kuvassa 12.

```
 9⊖ @Override
10 protected void doExecute(String descriptorFilename, BeanFactory context, TestResults results) {
11
12     IODescriptorParser parser = context.getBean("IODescriptorParser", IODescriptorParser.class);
13     IODescriptorHolder ioHolder = parser.parse(descriptorFilename);
14
15     SOAPConnectionFactory soapFactory1 = context.getBean("SOAPConnectionFactory_client1", SOAPConnectionFactory.class);
16     SOAPConnectionFactory soapFactory2 = context.getBean("SOAPConnectionFactory_client2", SOAPConnectionFactory.class);
17     SOAPConnectionFactory soapFactory3 = context.getBean("SOAPConnectionFactory_client3", SOAPConnectionFactory.class);
18     Properties connectionProperties = createConnectionProperties(443, "path/to/soap/interface");
19     SOAPConnection soap1 = soapFactory1.getConnection(connectionProperties);
20     SOAPConnection soap2 = soapFactory2.getConnection(connectionProperties);
21     SOAPConnection soap3 = soapFactory3.getConnection(connectionProperties);
22     connections.add(soap1);
23     connections.add(soap2);
24     connections.add(soap3);
25     soap1.open();
26     soap2.open();
27     soap3.open();
28
29     results.storeMessage("ConnectRequest client 1");
30     DescriptorUtil.runDescriptor(results, soap1, ioHolder, "ConnectRequest1", SOAPMessage.class);
31     results.storeMessage("ConnectRequest client 2");
32     DescriptorUtil.runDescriptor(results, soap2, ioHolder, "ConnectRequest2", SOAPMessage.class);
33     results.storeMessage("ConnectRequest client 3 rejected (Maximum number of connections reached)");
34     DescriptorUtil.runDescriptor(results, soap3, ioHolder, "ConnectRequest3_Rejected", SOAPMessage.class);
35     results.storeMessage("Disconnect client 1");
36     DescriptorUtil.runDescriptor(results, soap1, ioHolder, "Disconnect1", SOAPMessage.class);
37     results.storeMessage("ConnectRequest client 3 accepted");
38     DescriptorUtil.runDescriptor(results, soap3, ioHolder, "ConnectRequest3_Accepted", SOAPMessage.class);
39 }
```

Kuva 12 Esimerkkitestin toteutus

Testistä näytetään doExecute-funktion toteutus. Kuten aiemmin on kuvattu, doExecute on BaseTest-kantaluokan abstrakti funktio, johon toteutetaan testi. Riveillä 12-13 luodaan IO-kuvaajan jäsennin ja jäsennetään IO-kuvaajan sisältö IODescriptorHolder-olioon.

Riveillä 15-17 haetaan Springin ympäristöstä kolmelle asiakasyhteydelle tehtaat. Tehtaat ovat erillisiä eri asiakasyhteyksille, koska muun muassa HTTPS-yhteyden sertifikaatti määritellään tehtaassa ja joka asiakkaalla on oma sertifikaattinsa. Riveillä 18-27 luodaan ja avataan yhteydet. Yhteyksien luonnissa konfiguroidaan luotava yhteys getConnection-funktiolle annettavalla Properties-olion avain-arvo-pareilla. Konfiguroitavat asiat riippuvat yhteyden tyypistä, esimerkiksi palvelimen portti tai palvelun URL:n polku voivat olla konfiguroitavia asioita yhteyttä luodessa. Yhteydet myös tallennetaan connections-listaan, jotta BaseTest pääsee käsiksi kaikkiin käytettyihin yhteyksiin getConnections-funktion kautta.

Riveillä 29-32 avataan kaksi loogista yhteyttä käyttämällä DescriptorUtil-apuluokkaa. Testin etenemisestä raportoidaan TestResults-luokan storeMessage-funktiolla. DescriptorUtil-apuluokkaan on rakennettu apufunktioita suorittamaan yksinkertaisia viestisekvenssejä. Käytetty runDescriptor-funktio on esitetty kuvassa 13. Rivillä 34 yritetään avata yhteys, mutta yhteyspyyntö hylätään. Rivillä 36 suljetaan ensimmäinen yhteys ja rivillä 38 testataan, että nyt kolmannen asiakkaan yhteyspyyntö hyväksytään.

```java
8      public static <T extends IMessage> void runDescriptor(TestResults results,
9              IConnection<T> connection, IODescriptorHolder ioHolder,
10             String descriptorId, Class<T> messageType) {
11         List<T> inputs = ioHolder.getInputs(descriptorId, messageType);
12         List<T> outputs = ioHolder.getOutputs(descriptorId, messageType);
13         checkEqualsAmountOfInputsAndOutputs(results, inputs, outputs);
14
15         for (T input : inputs) {
16             connection.send(input);
17         }
18         results.storeAssertion("There should be no connection errors", 0,
19                 connection.getErrors().size());
20         results.storeAssertion(
21                 "There should be equals amount of responses and expected responses",
22                 outputs.size(), connection.getMessages().size());
23
24         for (int i=0; i<Math.min(outputs.size(), connection.getMessages().size()); i++) {
25             ComparisonResult result = outputs.get(i).compareTo(connection.getMessages().get(i));
26             result.storeAssertions(results);
27         }
28     }
```

Kuva 13 DescriptorUtil.runDescriptor-funktion toteutus

DescriptorUtil.runDescriptor-funktio lähettää descriptorId:ssä määritellyt syöteviestit ja vertailee saatuja vastauksia samassa IO-kuvaajassa määriteltyihin vasteviesteihin. Riveillä 11-13 haetaan jäsennetyt viestit descriptorId:n perusteella ja tarkastetaan, että syötteitä ja vasteita on yhtä monta kappaletta. Riveillä 15-17 syöteviestit lähetetään. Riveillä 18-22 tarkastetaan, että viestien lähetyksessä ei tapahtunut virheitä ja saatuja vastauksia on yhtä monta kuin odotettiin saatavan. Riveillä 24-27 verrataan vastaanotettuja viestejä IO-kuvaajassa määritettyihin viesteihin ja tallennetaan tulokset TestResults-olioon. Osa testin IO-kuvaajasta on esitetty kuvassa 14.

```xml
1  <iodescriptor id="ConnectRequest3_Rejected" type="SOAP">
2      <inputs>
3          <soapInput>
4              <ConnectRequest>
5                  <DataAddress>10.10.10.10</DataAddress>
6                  <DataPort>50000</DataPort>
7              </ConnectRequest>
8          </soapInput>
9      </inputs>
10     <outputs>
11         <soapOutput>
12             <ConnectResponse>Maximum number of connections reached</ConnectResponse>
13         </soapOutput>
14     </outputs>
15 </iodescriptor>
16 <iodescriptor id="ConnectRequest3_Accepted" type="SOAP">
17     <inputs>
18         <soapInput>
19             <ConnectRequest>
20                 <DataAddress>10.10.10.10</DataAddress>
21                 <DataPort>50000</DataPort>
22             </ConnectRequest>
23         </soapInput>
24     </inputs>
25     <outputs>
26         <soapOutput>
27             <ConnectResponse>Ok</ConnectResponse>
28         </soapOutput>
29     </outputs>
30 </iodescriptor>
```

Kuva 14 Osa esimerkkitestin IO-kuvaajasta

IO-kuvaajassa on esitetty kolmannen asiakkaan epäonnistuva ja onnistuva yhteydenluontisekvenssi. Kummatkin sekvenssit koostuvat yhdestä iodescriptor-elementistä, joissa kummassakin on vain yksi syöteviesti. Syöteviestit ovat kummassakin sekvenssissä samat, mutta odotetut vastaukset eroavat. Kuva 12 mukaisesti DescriptorUtil.runDescriptor-funktiolle annetaan iodescriptor-elementin id-attribuutin arvo. Tällöin funktio lähettää IO-kuvaajiin määritetyt syöteviestit ja vertailee saatuja vastauksia IO-kuvaajiin määritettyihin vasteviesteihin.

Automaatiotestijärjestelmän ajurina toimii JUnit, joten testille on kirjoitettava JUnit testi. JUnit testin toteutus on esitetty kuvassa 15.

```
1  @RunWith(SpringJUnit4ClassRunner.class)
2  @ContextConfiguration(locations={"config1-components.xml"})
3  public class ConnectionTest {
4
5      @Autowired
6      private ApplicationContext context;
7
8      @Test
9      public void testMaximumConnections() {
10         MaximumConnections test = new MaximumConnections("MaximumConnections.xml", context);
11         test.execute();
12     }
13
14     ... Other tests ...
15 }
```

Kuva 15 Esimerkkitestin JUnit-toteutus

Testin Spring-ympäristön konfigurointi tehdään ContextConfiguration-annotaatiolla. Tiedosto config1-components.xml on Spring-konfigurointitiedosto, johon on määritelty testien tarvitsemat komponentit. Esimerkkitesti tarvitsee Spring-ympäristöstä seuraavat komponentit: IODescriptorParser, SOAPConnectionFactory_client1, SOAPConnectionFactory_client2 ja SOAPConnectionFactory_client3. Kuvassa 16 on esitetty komponentin SOAPConnectionFactory_client1 määritys Spring-konfiguraatiotiedostossa.

```
1 <bean id="SOAPConnectionFactory_client1" class="package.of.the.class.SOAPConnectionFactory">
2     <constructor-arg value="${ep.host}" />
3     <constructor-arg value="${ep.port}" />
4     <constructor-arg value="client1-certificate.jks" />
5     <constructor-arg value="passwordForCertificate" />
6 </bean>
```

Kuva 16 Osa Spring-konfiguraatiota

SOAPConnectionFactory ottaa rakentajassaan palvelimen osoitteen, portin, asiakkaan sertifikaatin ja sertifikaatin salasanan. Palvelimen portti ja osoite on määritelty erillisessä properties-tiedostossa, joten ne ovat Spring-konfiguraatiossa määritelty erityisellä notaatiolla. Spring konfiguraation ${ep.host} ja ${ep.port} korvataan properties-tiedostossa olevilla arvoilla ennen niiden antamista SOAPConnectionFactoryn rakentajalle.

ConnectionTest-luokan context-jäsenmuuttuja on Spring-ympäristö, jonka avulla testi hakee tarvitsemansa komponentit. Jäsenmuuttuja konfiguroidaan Autowired-

annotaatiolla, jolloin Spring asettaa siihen automaattisesti viitteen käyttämällä Javan Reflection-ominaisuuksia.

Yksittäinen testi määritellään funktioon, joka on merkitty Test-annotaatiolla. Koska testi on toteutettu erilliseen luokkaan, funktiossa yksinkertaisesti luodaan testiluokka ja ajetaan sen execute-funktio.

3.6.3 Testin ajaminen ja tulosten tarkastaminen

Testiautomaatiojärjestelmän testit ajetaan Hudsonilla, johon on määritelty erillinen tehtävä testien ajamiselle. Testien ajo käynnistetään Hudsonin web-liittymästä painamalla tehtävän käynnistävää nappulaa. Konsoli-ikkunaa voidaan seurata suoraan Hudsonin web-käyttöliittymästä. Lokitiedosto tallentuu Hudson-tehtävän työympäristöön, joka on myös selattavissa suoraan web-käyttöliittymästä

Testien tulokset voidaan tarkastaa myös web-käyttöliittymästä. JUnitin koostama raportti julkaistaan testien ajamisen jälkeen. Raportista selviää mm. mitkä testit ajettiin, mitkä testiajot epäonnistuivat ja testiajojen konsolilokit.

Normaalisti testien lokitus on konfiguroitu tallentumaan tiedostoon sekä tulostumaan konsoliin. Tiedostoon tallentuvaan lokiin voidaan kuitenkin kerätä yksityiskohtaisempaa tietoa testin suorituksesta kuin konsolilokiin, jolloin tiedostolokia voidaan käyttää selvittäessä testiajon epäonnistumista. Kuva 17 esittää konsolilokin testin ajosta.

```
INFO 2011-09-04 15:52:19:234 [main] Restarting EP
INFO 2011-09-04 15:52:20:234 [main] ### MaximumConnections ###
INFO 2011-09-04 15:52:20:244 [main] ConnectRequest client 1
INFO 2011-09-04 15:52:20:254 [main] ConnectRequest client 2
INFO 2011-09-04 15:52:20:264 [main] ConnectRequest client 3 rejected (Maximum number of
connections reached)
INFO 2011-09-04 15:52:20:274 [main] Disconnect client 1
INFO 2011-09-04 15:52:20:284 [main] ConnectRequest client 3 accepted

INFO 2011-09-04 15:55:31:567 [main] Restarting
INFO 2011-09-04 15:55:32:567 [main] ### MaximumConnections ###
INFO 2011-09-04 15:55:32:577 [main] ConnectRequest client 1
INFO 2011-09-04 15:55:32:587 [main] ConnectRequest client 2
INFO 2011-09-04 15:55:32:588 [main] ConnectRequest client 3 rejected (Maximum number of
connections reached)
ERROR 2011-09-04 15:55:32:588 [main] Comparing ConnectRequest3_Rejected message 1:
Expected "Maximum number of connections reached", actual "Ok"
INFO 2011-09-04 15:55:32:598 [main] Disconnect client 1
INFO 2011-09-04 15:55:32:608 [main] ConnectRequest client 3 accepted
ERROR 2011-09-04 15:55:32:608 [main] Comparing ConnectRequest3_Accepted message 1:
Expected "Ok", actual "Already connected"
```

Kuva 17 Testiloki

Esimerkin vuoksi testi on ajettu kaksi kertaa. Ensimmäisellä kerralla testi on mennyt onnistuneesti läpi, ja toisella kerralla testiajo on epäonnistunut. Ensimmäisellä rivillä käynnistetään EP uudestaan, uudelleenkäynnistämisen hoitaa BaseTest-kantaluokka. Toisella rivillä tulostetaan testin nimi, ja tämän jälkeen tulostetaan testin vaiheet suorituksen edetessä.

Toinen testiajo on epäonnistunut, jonka huomaa parhaiten rivin aloittavasta ERROR-sanasta. ERROR-rivit kertovat miten testi epäonnistui. Esimerkkiajossa kolmas asiakas yrittää ottaa yhteyttä, jolloin EP:n pitäisi vastata ”Maximum number of

connections reached”, mutta saatu vastaus kertoo ”Ok”. Tämä tarkoittaa, että asiakas sai virheellisesti luotua yhteyden. Myöskään toinen kolmannen asiakkaan yhteydenluontipyyntö ei mene testin mukaisesti, koska yhteys on jo virheellisesti auki. Tässä tapauksessa EP vastaa yhteyspyyntöön viestillä: ”Already connected”.

4 TESTAUSJÄRJESTELMÄN KEHITYSSUUNNITELMA

Tässä luvussa analysoidaan testausjärjestelmän sovelluskehystä ja esitetään ratkaisuja löydettyihin ongelmiin. Testausjärjestelmässä on kehitettävää myös muilla kuin sovelluskehyksen saralla. Työhön annettujen resurssien rajallisuuden takia, tässä työssä on keskitytty sovelluskehyksen kehittämiseen. Ensin perustellaan kehityssuunnitelmien toteutuskielen valinta, sitten esitellään kolme kehityssuunnitelmaa, kukin omassa kohdassaan.

4.1 Toteutuskielen valinta

Testaussovelluskehystä käyttävät testit toteutetaan Javalla. Testeille ei luonnollisesti kirjoiteta testejä, joten Java on hyvä toteutuskieli niille, koska se on sekä vahvasti, että staattisesti tyypitetty kieli. Vahvalla tyypityksellä tarkoitetaan sitä, että muuttujat ja arvot ovat sidottuja tiettyyn tyyppiin [31]. Staattisella tyypityksellä tarkoitetaan sitä, että tyyppitarkastus tehdään käännösaikana [31]. Kummatkin tyypityksen piirteet auttavat toteuttajaa välttämään ohjelmointivirheitä kääntäjän avulla, mikä on tärkeää testaamattomassa koodissa.

Osa myöhemmin esiteltävistä kehityssuunnitelmista vaatii tai ainakin painostaa toteutuskielen vaihtamiseen. Yhtenä suurena vaikeutena Javassa on sen työläs syntaksi, joka osaksi johtuu Javan tyypityksen ominaisuuksista. Toteutuskieleksi on valittava kieli, joka on sekä kevyt syntaksiltaan, että vahvasti ja staattisesti tyypitetty. Toteutuskielen valintaa rajoittaa myös se, että sovelluskehys on Javaa, joten toteutuskielen täytyy olla helposti käytettävissä Java-koodin kanssa.

Nämä kriteerit täyttää Scala [32], joka on sekä vahvasti että staattisesti tyypitetty kieli ja jossa on kevyt syntaksi. Kevyt syntaksi johtuu osaksi Scalan suunnitteluratkaisujen ansiosta, osaksi sen kääntäjän ominaisuuksien ansiosta. Scalan kääntäjä osaa esimerkiksi päätellä muuttujien tyypit niihin sijoitettavien arvojen tyyppien perusteella, jolloin muuttujan tyyppiä ei tarvitse eksplisiittisesti määritellä. Kuten Java, Scala on JVM:ssä (Java Virtual Machine) ajettava kieli, joten sitä on helppo käyttää Javan kanssa. Seuraavaksi esitellään kehityssuunnitelmissa käytetyt Scalan erityisominaisuudet, joita Javassa ei ole.

Scalassa kaikki kielen alkiot ovat olioita, myös funktiot. Tällöin esimerkiksi funktioita voidaan antaa parametreina toisille funktioille, mikä helpottaa vastakutsujen (callback) tekemistä. Vastakutsulla tarkoitetaan koodiviittausta, joka annetaan parametrina toiselle koodille, joka suorittaa viittauksen osoittaman koodin. Javassa

vastakutsujen tekeminen voidaan toteuttaa Komento-suunnittelumallilla [30]. Komento-suunnittelumallissa funktiokutsu kääritään olioon, jolla on kutsuttava funktio. Usein tämä suunnittelumalli toteutetaan Javassa anonyymeillä luokilla [33], mikä on työlästä Javan syntaksilla.

Scala sallii funktioiden ja luokkien niminä lähdes mitä tahansa merkkejä. Scalassa on myös vaihtoehtoinen funktioiden kutsumistapa, jossa ei käytetä pisteitä eikä sulkuja parametrien ympärillä. Tämä kutsumistapa toimii vain funktioissa, joilla on yksi parametri, mutta yhdistettynä funktioiden vapaaseen nimeämiseen, voidaan Scalassa toteuttaa operaattoreiden määrittelyä. Itse asiassa Scalassa operaattorit ovatkin vain tavallisia funktioita.

Scalassa voidaan olioita muuntaa toisiksi automaattisesti implisiittisten funktioiden avulla eli Scalan termein näkymien avulla. Funktio joka ottaa parametrikseen olion tyyppiä A ja palauttaa olion tyyppiä B voidaan merkitä implisiittiseksi, jolloin sitä kutsutaan automaattisesti, jos on tarpeellista tehdä muunnos tyypistä A tyyppiin B. Kääntäjä lisää funktiokutsun automaattisesti, jos implisiittinen funktio on näkyvillä koodilohkossa.

Implisiittisten funktioiden, ”operaattoreiden” määrityksen ja Scalan kevyen syntaksin ansiosta sisäisten täsmäkielien (DSL, Domain-Specific Language) määritteleminen on mahdollista. Täsmäkielellä tarkoitetaan kieltä, joka on suunniteltu tietylle sovellusalueelle. Sisäisellä täsmäkielellä tarkoitetaan täsmäkieltä, joka on toteutettu kirjastona yleiskäyttöisemmälle kielelle. Täsmäkielillä saadaan parannettua tuottavuutta, luotettavuutta ja ylläpidettävyyttä [34].

Scalan tekijöiden mielestä XML on niin tärkeä osa nykypäivän ohjelmia, että Scala tukee natiivisti XML:ää. Scala-koodin joukkoon voidaan kirjoittaa XML-elementtejä, jotka kääntäjä muuntaa automaattisesti Scalan XML-kirjaston elementtiolioiksi.

4.2 Datankäsittely

Sovelluskehyksen datankäsittely esitettiin kohdassa 3.5.2.1. Datan erottamisella testilogiikasta, eli IO-kuvaajien käytöllä on saavutettu useita etuja, kuten parempi ylläpidettävyys ja testidatan tarkastelun helppous. Käytännön toteutuksessa on kuitenkin lukuisia puutteita, jotka hankaloittavat datankäsittelyä. Ensimmäisenä kehityssuunnitelmana on datankäsittelyn parantaminen.

4.2.1 Analyysi

Viestit IO-kuvaajissa on ryhmitelty input- ja output-osiin, joissa kummassakin voi olla useampia viestejä. Tällöin input-osan ensimmäisen viestin vastaus on output-osan ensimmäinen viesti jne. Ongelma syntyy jos johonkin viestiin ei odoteta vastausta, jolloin useamman viestin sekvensseissä viesti-vastausparit on merkittävä erikseen esimerkiksi attribuuteilla tai kommenteilla. Viestirakenteen ongelmallisuus ilmenee myös suurissa IO-kuvaajissa, jotka eivät mahdu yhdelle ruudulle, jolloin viesti-vastauspareja on hankala hahmottaa.

IO-kuvaajissa yleensä input-osa määrittää lähetettävät viestit ja output-osa määrittää odotetut vastaukset, mutta IO-kuvaajasta riippuen osat voivat olla myös toisinpäin. Tavallisessa tapauksessa vertailu on yksi-yhteen-tyyppistä, jolloin vertailuviesti näyttää identtiseltä oikeaan viestiin verrattuna, tällöin on hankala huomata onko viesti lähetettävä viesti vai vertailuviesti.

Tällä hetkellä IO-kuvaajiin ei ole mahdollista määrittää odotettuja virheitä tai virhekoodeja, jotka tapahtuvat alemmilla protokollatasoilla. Esimerkiksi HTTP-statuskoodia ei voida määrittää IO-kuvaajaan, jolloin ei voida IO-kuvaajassa määrittää esimerkiksi viesti-vastausparia, jossa vastauksena saadaan HTTP-viesti statuskoodilla 404. Vastauksen vertailu on toteutettava testilogiikkaan, jolloin se ei näy IO-kuvaajassa.

IO-kuvaajien määritys XML:nä on yhteneväisyyden kannalta toimiva ratkaisu. Kaikki viestit pyritään määrittelemään XML:ssä, eikä ole esimerkiksi tarvetta määritellä osaa viesteistä koodissa. XML:n käyttö aiheuttaa kuitenkin lukuisia ongelmia.

XML-viestimääritykset ovat hankalia ylläpitää viestien sisällön muuttuessa. Esimerkiksi BP-viestit jäsennetään oliomalliin avain-arvopareista, mutta koska avain-arvoparit ovat XML:ssä tekstinä, muutokset oliomalliin huomataan vasta ajon aikana.

XML-määritykset ovat myös paljon tilaa vieviä, varsinkin pieniin viestisekvensseihin tulee paljon XML-elementtejä, jotka eivät määrittele varsinaista viestien sisältöä. Yhteen testiin voi kuulua kymmeniä tai jopa satoja viestejä, jolloin IO-kuvaajat kasvavat pahimmillaan yli kymmenentuhannen rivin pituisiksi.

XML aiheuttaa myös suorituskykyongelmia suurilla IO-kuvaajilla. Jäsentäminen on hidasta ja turhaa jäsentämistä tehdään uudelleenkäytettävien IO-kuvaajien takia, koska niiden kaikkia viestejä ei usein käytetä. IO-kuvaajan jäsentämiseen kuluu tavallisesti aikaa sekunnista kymmeniin sekunteihin riippuen IO-kuvaajan koosta ja käytetystä testikoneesta. Ongelma on suurin testin kehitysvaiheessa, jolloin on tarve ajaa testiä aina muutosten jälkeen.

XML myös rajoittaa viestien konfigurointia ja uudelleenkäyttöä. XML-tiedostoon ei voida määrittää tuntematonta arvoa eli niin kutsuttua placeholder-arvoa, jonka lopullisen arvon määrittää esimerkiksi testilogiikka tai konfiguraatiotiedosto. Tämä toiminnallisuus on rajoitetusti toteutettu IODescriptorParseriin, mutta kaikki tuntemattomat arvot täytyy olla tiedossa ennen XML:n parsintaa. Esimerkiksi testitapaus, jossa odotettu arvo riippuu testin aikana saadusta satunnaisesta arvosta, on mahdotonta toteuttaa niin, että odotettu viesti on määritelty XML:ään. Oletusviestimäärityksillä on helppoa määritellä usein käytettyjä viestejä, mutta oletusviestituen toteuttaminen viestityypille on työlästä, koska toteutus on tehtävä niin XML-skeemaan kuin viestin jäsentäjään.

IODescriptorParserin ja IODescriptorHolderin käyttö on virhealtista. Ennen IO-kuvaajan jäsentämistä, IODescriptorParserille annetaan tuntemattomat arvot avain-arvopareina, mikä on virhealtista, koska missään ei eksplisiittisesti kerrota mitkä arvot IODescriptorParserille täytyy antaa. Myös viestien hakeminen IODescriptorHolderista on virhealtista, koska viestit haetaan merkkijonotyyppisen id:n perusteella. Kirjoitusvirheet tai muutokset id:n eivät tule käännösaikana esille.

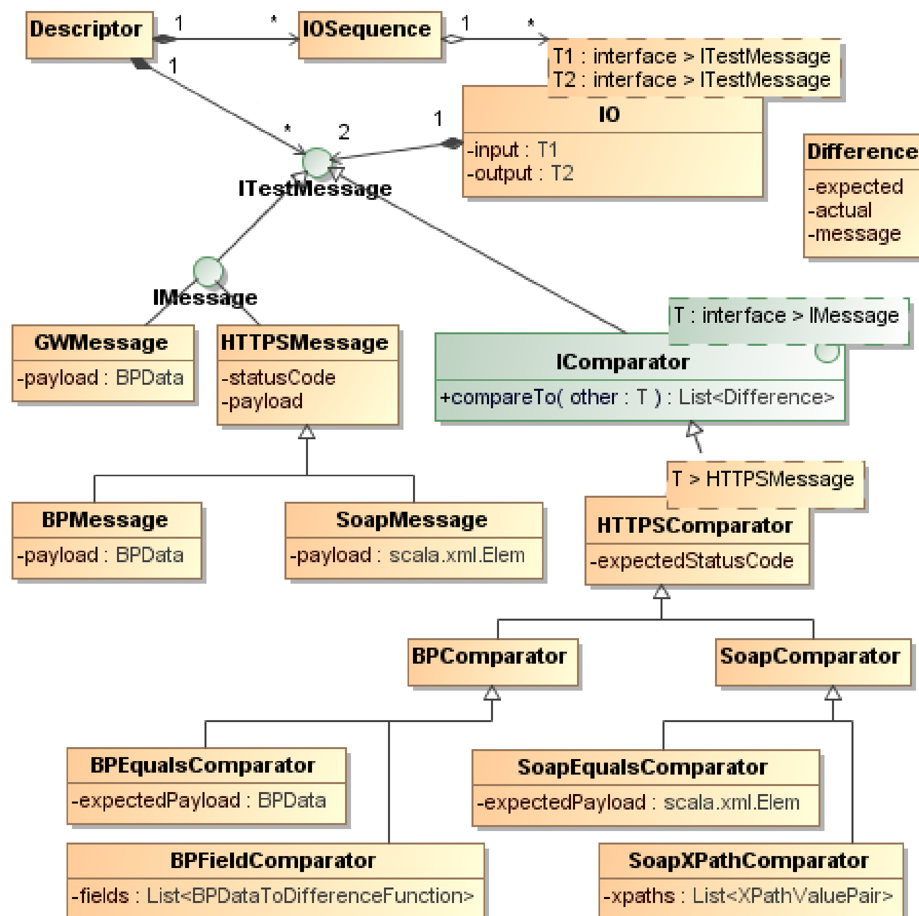
Saman viestin käyttäminen yhtäaikaisesti useammassa paikassa on vaarallista, koska IODescriptorHolderista saatava viesti on aina samaa instanssia ja viestejä pystytään muuttamaan niiden luonnin jälkeen. Esimerkiksi BPConnectionin automaattisesti tekemät muutokset viestin otsikkotietoihin voivat kantautua väärään viestinlähetystapahtumaan.

4.2.2 Ratkaisu

Suuri määrä ongelmia aiheutuu XML:n käytöstä IO-kuvaajissa. Yksinkertainen ratkaisu ongelmiin on siirtää IO-kuvaajat koodiin. IO-kuvaajien siirto koodiin mahdollistaa joustavamman viestien määrityksen esimerkiksi parametrien avulla, parantaa suorituskykyä ja helpottaa viestien ylläpitoa käännösaikaisten virheilmoitusten avulla.

IO-kuvaajien siirtäminen koodin ei ole kuitenkaan helppoa, koska kuvaajien täytyy olla helposti luettavia. Toteutus tehdään Scalalla ja kuvaajien määritykseen tehdään sisäinen täsmäkieli. SOAP-viestien määritykset tehdään Scalan XML-tuella.

Muutos XML:stä Scalaan vaatii muutoksia sovelluskehyksen arkkitehtuuriin datankäsittelyn osalta. Pelkän kielimuutoksen lisäksi uudessa arkkitehtuurissa on otettu huomioon myös muita esitettyjä ongelmia. Kuva 18 esittää uuden arkkitehtuurin datankäsittelylle ja esittelee muutamia toteuttavia luokkia esimerkin vuoksi.



Kuva 18 Uusi arkkitehtuuri datankäsittelylle

IO-kuvaaja (Descriptor) koostuu viestisekvensseistä (IOSequence), viesteistä (IMessage) ja viestivertailijoista (IComparator). Viestisekvenssit koostuvat listasta IO-olioita, jotka pitävät sisällään syötteen ja vasteen. Syöte ja vaste voivat olla joko viestejä tai viestivertailijoita. Viestisekvenssit ja yksittäiset viestit tai viestilistat määritellään IO-kuvaajaan tavallisina funktioina, jotka palauttavat uuden instanssin viestiresurssista. Viestit haetaan funktioiden kautta, jolloin funktion nimen muutos huomataan sitä käyttävässä koodissa käännösvirheenä. Koska funktiot luovat kutsuttaessa aina uuden instanssin resurssista, voidaan samoja viestejä käyttää turvallisesti yhtä aikaa.

IMessage-toteutuksissa on otettu huomioon sovellustason protokollat, koska testilogiikka voi ottaa niihin kantaa. Esimerkiksi BPMessage ja SOAPMessage periytyvät kummatkin HTTPSMessagesta. Tällöin voidaan viestin vertailussa ottaa huomioon HTTPS tason virheet.

IComparatoriin on siirretty vanhassa arkkitehtuurissa IMessagessa ollut compareTo-funktio. Funktiolla vertaillaan viestejä, ja se palauttaa löydetyt eroavaisuudet (Difference). IComparatorin toteuttavat luokat noudattavat vastaavaa rakennetta kuin IMessagen toteuttavat luokat. IComparatoreita eli viestivertailijoita käytetään nimensä mukaisesti vain vertailemaan viestejä. IMessageiden ja IComparatoreiden erotuksella tehdään selkeäksi se, mitä viestejä käytetään vertailuun ja mitkä viestit on tarkoitus lähettää testin aikana. IComparatoreita voidaan toteuttaa testin tarpeiden mukaan. Kuvassa esimerkkeinä on muun muassa SoapEqualsComparator ja SoapXPathComparator. SoapEqualsComparator on ajateltu vertaavan viestiä yksi-yhteen periaatteella. SoapXPathComparatorin on ajateltu määrittävän listan XPath-hakuja ja niiden odotettuja tuloksia, jolloin viesteistä voidaan vertailla vain halutut osat. XPath on kieli, jolla haetaan dataa XML:stä [35].

Uuden arkkitehtuurin mukainen Scalalla toteutettu IO-kuvaaja on esitelty kuvassa 19. Tämä IO-kuvaaja vastaa kuvassa 9 esiteltyä XML-pohjaista IO-kuvaajaa.

```
5   def descriptorId = new IOSequence[BPMessage, BPComparator](
6     BPMessage(
7       BPData() // Empty data
8     )
9     -> BPEqualsComparator(
10      BPData() // Empty data
11    ),
12
13    BPMessage(
14      BPData() // Empty data
15    )
16    -> BPEqualsComparator(
17      BPData() // Empty data
18    )
19  )
```

Kuva 19 Uuden arkkitehtuurin mukainen IO-kuvaaja

Suurin rakenteellinen muutos vanhaan verrattuna on viestisekvenssin muuttaminen syöte-vastepareihin. Syötteen ja vasteen välissä on ”->”-merkkijono, joka kuvaa syötteen ja vasteen välistä suhdetta. Output-viestit ovat esimerkissä viestivertailijoita ja

niiden eron lähetettäviin viesteihin huomaa selkeästi. Viestisekvenssin luonnissa määritellään syötteen ja vasteen tyypit tyyppiparametreissa, tällöin ei voida sekoittaa eri tyyppisiä viestejä ja vertailijoita keskenään. IO-kuvaajan määrittelystä on myös karsiutunut turhia rivejä pois.

Viestien hankala ja rajoittunut konfigurointi oli vanhoissa IO-kuvaajissa suurin ongelma. Koska uusi IO-kuvaaja on tavallista Scala-koodia, voidaan funktioille tai koko IO-kuvaajan luokalle antaa parametreja viestien konfigurointiin. Kuva 20 on kuvan 14 mukainen IO-kuvaaja toteutettuna Scalalla, jossa viesteihin on lisätty konfiguroitavuutta parametrien avulla.

```
22 class ConnectionTestDescriptor(config: Config) {
23
24   def ConnectRequest3_Rejected(port: Int) = new IOSequence[SoapMessage, SoapComparator](
25     SoapMessage(
26       <ConnectRequest>
27         <DataAddress>{ config.clientAddress }</DataAddress>
28         <DataPort>{ port }</DataPort>
29       </ConnectRequest>
30     )
31     -> SoapEqualsComparator(
32       <ConnectResponse>Maximum number of connections reached</ConnectResponse>
33     )
34   )
35
36   def ConnectRequest3_Accepted(port: Int) = new IOSequence[SoapMessage, SoapComparator](
37     SoapMessage(
38       <ConnectRequest>
39         <DataAddress>{ config.clientAddress }</DataAddress>
40         <DataPort>{ port }</DataPort>
41       </ConnectRequest>
42     )
43     -> SoapEqualsComparator(
44       <ConnectResponse>Ok</ConnectResponse>
45     )
46   )
47 }
```

Kuva 20 Uusi esimerkkintestin IO-descriptor

Kuvan IO-kuvaajassa näytetään myös, kuinka XML-tyyppisiä viestejä määritellään käyttämällä Scalaan sisäänrakennettua tukea XML:lle. ConnectionTestDescriptorille annetaan sen rakentajassa config-olio, jota voidaan käyttää suoraan viestien määrityksessä, esimerkiksi DataAddress-elementin sisältö saadaan config-oliosta.

Viestisekvenssifunktioille on myös määritelty parametri, johon viestisekvenssin käyttäjä voi määrittää DataPort-elementin arvon. Tällöin arvo voidaan asettaa testin ajon aikana dynaamisesti. Vanhassa arkkitehtuurissa tämä ei ollut mahdollista, vaan testin ajon aikana selviävät arvot on täytynyt asettaa testilogiikassa, mikä on hankalaa testin toteuttajan ja tarkastajan näkökulmasta.

Oletusviestitoiminnallisuus on helppo toteuttaa mille tahansa viestille luomalla funktio, joka ottaa parametreina konfiguroitavat arvot ja palauttaa valmiin viestin. Esimerkiksi ConnectRequest-viestille on helppo luoda oletusviestifunktio, joka ottaa parametreina DataAddress- ja DataPort-elementtien arvot ja palauttaa valmiin viestin.

SOAP-viestien määritys on edelleen muutosherkkää, koska viestit määritellään pelkkänä XML:nä. Mahdollinen ratkaisu ongelmaan on käyttää hyödyksi Scalan XML-elementtien sijaan datan sitomista (data binding). XML -datan sitomisella tarkoitetaan, sitä kun XML-skeeman perusteella luodaan oliomalli, joka vastaa käytettyjä XML-rakenteita. Oliomalli voidaan muuntaa helposti XML:ksi ja toisinpäin. IO-kuvaajissa voitaisiin käyttää oliomallia XML:n sijaan, kuten BP-viestien tapauksessa jo tehdään. Oliomallia käytettäessä huomataan siihen tulevat muutokset jo käännösvaiheessa. Ratkaisu on kompromissi luettavuuden ja ylläpidettävyyden välillä, koska XML-muotoiset viestit ovat luettavampia.

4.3 Yhteydet

Sovelluskehyksen yhteydet esiteltiin kohdassa 3.5.2.2. Olemassa olevat yhteydet ja niiden tarjoama toiminnallisuus on riittävä toteuttamaan suunnitellut testit, mutta ongelmia on sekä yhteyksien luonnissa että niiden käytössä. Toisena kehityssuunnitelmana on yhteyksien luonnin ja käytön uudistaminen.

4.3.1 Analyysi

Nykyinen yhteyksien arkkitehtuuri, joka on esitetty kuvassa 10, on hyvin yksinkertainen. Ideana on ollut, että testilogiikka käyttää yhteyksiä IConnection- ja IConnectionFactory-rajapintojen kautta. Kaikki yhteydet toteuttavat saman rajapinnan joten ne ovat yhdenmukaisia käyttää.

Suurin ongelma ratkaisussa on se, että esimerkiksi syöte-vastaus-tyyppinen yhteys ei ole käyttömalliltaan samanlainen kuin viestejä vastaanottava yhteys. Tämä on johtanut siihen, että IConnection-rajapintaa on yleistetty pisteeseen, jossa rajapinnan käyttö ei ole intuitiivista millään yhteystyypillä, eli rajapintaan on sovellettu One Size Fits All –antisuunnittelumallia [36].

Yhteyksien luominen on myös hankalaa kahdesta syystä: erityyppiset yhteystehtaat ja yhteyksien konfiguroinnin vastuiden sekava jakaminen. Jokaiselle yhteystyypille on oma tehdas jolloin erityyppisen yhteyden luomiseksi on myös haettava eri tehdas Spring-ympäristöstä. Usein tehtailla luodaan vain yksi yhteys, joten yhteyden luominen on vaivalloista.

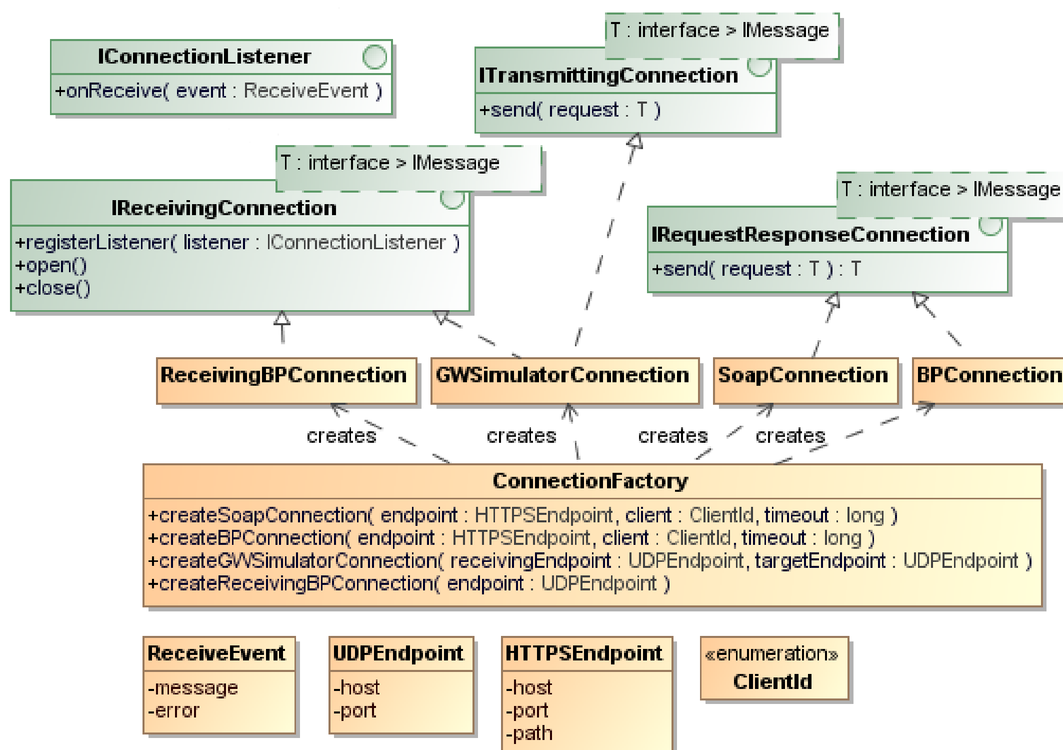
Yhteyksien asetusten konfigurointi tapahtuu sekä tehtaissa että yhteyksiä luodessa. Tehtaiden vastuulla ovat asetukset, jotka ovat testausjärjestelmän ympäristössä vakioita esimerkiksi IP-osoitteet ja portit. Myös asiakkaiden sertifikaatit määritellään tehtaisiin. Yhteyttä luodessa konfiguroidaan yhteyskohtaiset asetukset ja asetukset joita on pystyttävä testaamaan. Tämän ratkaisun etuna on se, että testilogiikan ei tarvitse tietää esimerkiksi, mikä on testattavan EP:n IP-osoite tai SOAP-rajapinnan portti. Toisaalta varsinkin sertifikaattien määritteleminen tehtaisiin hankaloittaa tehtaiden käyttöä, koska joka asiakkaalle on oltava omat tehtaansa määriteltynä. Myöskin testeille, jotka testaavat esimerkiksi porttien konfiguroitavuutta, on määritettävä Spring-ympäristöön omat tehtaansa, joita ei käytetä muissa testeissä.

Yhteyksiä luodessa ne konfiguroidaan antamalla getConnection-funktiolle Properties-tyyppinen olio, missä on avain-arvo-pareilla määritettynä asetukset. Erityyppisillä yhteyksillä vaaditut asetukset vaihtelevat, joten vaaditut asetukset on käytävä tarkistamassa yhteyden dokumentaatiosta. Properties-olion käyttö parametrien välittämiseen on huono ratkaisu, koska virheitä ei huomata käännösaikana, ja dokumentaation selaileminen on ylimääräistä työtä verrattuna siihen, että vaaditut asetukset olisivat näkyvissä funktion määrittelyssä parametreina.

Järjestelmässä GW-simulaattorin käyttö tapahtuu sekä GWSimulatorConnectionin, että JMSConnectionin kautta. GWSimulatorConnectionilla kerrotaan, mitä GW vastaa viesteihin ja GW-simulaattori laittaa saamansa viestit JMS-jonoon. Loogisesti GW:n vastaanottamat ja lähettämät viestit kuuluvat yhden yhteyden vastuulle, joten JMSConnectionin käyttö monimutkaistaa tilannetta.

4.3.2 Ratkaisu

Ongelmien ratkaisemiseksi yhteyksien arkkitehtuuri on suunniteltu uudelleen. Ratkaisussa on pyritty ottamaan huomioon kaikki havaitut ongelmat. Uudistettu yhteyksien arkkitehtuuri on esitetty kuvassa 21. Yhteyksien kaikkia toteutuksia ei ole esitetty kuvassa tilan säästämiseksi. Puuttuvat toteutukset ovat MaintenanceConnection, MonitoringConnection ja LDAPConnection.



Kuva 21 Uusi arkkitehtuuri yhteyksille

IConnection-rajapinta on jaettu kolmeen osaan: IReceivingConnectioniin, ITransmittingConnectioniin ja IRequestResponseConnectioniin. Ratkaisulla on pyritty

yksinkertaistamaan yhteyksien käyttöä, koska rajapinnat vastaavat paremmin yhteyksien käyttömalleja.

IReceivingConnectionilla otetaan vastaan viestejä. Koska viestien vastaanottaja ei tarkalleen tiedä, koska viesti saapuu, yhteys kertoo viestin saapumisesta eli rajapinta toimii Tarkkailija-suunnitelumallin (Observer) [30] mukaisesti. Yhteyden käyttäjä rekisteröi IConnectionListener-kuuntelijan yhteydelle. Yhteys taas ilmoittaa saapuneesta viestistä IConnectionListener.onReceive-funktion ja ReceiveEventin avulla. ReceiveEvent paketoi viestin ja mahdollisen virheen samaan olioon. Tällöin viestin vastaanottaja voi helposti reagoida sekä viesteihin, että tapahtuneisiin virheisiin viestin vastaanotossa.

ITransmittingConnectionilla lähetetään viestejä joihin ei odoteta suoraa vastausta. Rajapinta on tästä syystä hyvin yksinkertainen, se muodostuu send-funktiosta, jolle annetaan parametrina lähetettävä viesti. Huomionarvoisesti GWSimulatorConnection toteuttaa sekä IReceivingConnectionin, että ITransmittingConnectionin. Mahdollinen käyttötapaus GWSimulatorConnectionille, mikä vaatii kummankin rajapinnan toteuttamisen, on viestin vastaanotto ja siihen vastaaminen. GWSimulatorConnectioniin on yhdistetty JMSConnectionin toiminnallisuus.

IRequestResponseConnection on käyttömalliltaan syöte-vaste-tyyppinen. Ero ITransmittingConnectioniin on se, että send-funktio palauttaa saadun vastauksen.

Yhteyksien luonti on keskitetty ConnectionFactoryyn. Kaikki yhteyden vaatimat asetukset annetaan yhteyden luontivaiheessa. Asetuksissa pääosassa on endpoint-käsite, mikä määrittää yhteyden kommunikaatiopisteen, eli mihin yhteys lähettää viestit tai mistä yhteys vastaanottaa viestit. Ideana uusituilla asetuksilla on määrittää valmiit yleisesti käytetyt endpoint-oliot Spring-ympäristöön, jolloin testin ei tarvitse määritellä niitä itse. Testi voi kuitenkin määritellä omia testikohtaisia endpointteja, esimerkiksi testatakseen asetuksien muutoksia.

ClientId-enumeraatio määrittelee testeissä käytettävät asiakkaat ja yhdistää ne sertifikaatteihin. ClientId:n käytöllä abstrahoidaan sertifikaattitiedostot pois testilogiikasta, jolloin testit eivät edelleenkään ole suoraan riippuvaisia asiakkaiden sertifikaattitiedostoista. Testin kirjoittajan täytyy valita asiakasyhteyttä luodessa asiakkaan ClientId, jolloin yhteys osataan luoda käyttämään oikeaa sertifikaattia.

Kuva 22 esittää yhteyksien luontia uudella arkkitehtuurilla esimerkkitestin mukaisesti (katso kuva 12). Aiempaan verrattuna yhteyksien luonti on yksinkertaisempaa, koska kaikki yhteydet hyödyntävät samaa endpointtia ja yhteydet luodaan samalla tehtaalla.

```
19        HTTPSEndpoint endpoint = context.getBean("SoapEndpoint", HTTPSEndpoint.class);
20        SoapConnection soap1 = connectionFactory.createSoapConnection(endpoint, ClientId.CLIENT1, 1000);
21        SoapConnection soap2 = connectionFactory.createSoapConnection(endpoint, ClientId.CLIENT2, 1000);
22        SoapConnection soap3 = connectionFactory.createSoapConnection(endpoint, ClientId.CLIENT3, 1000);
```

Kuva 22 Yhteyksien luonti uudella arkkitehtuurilla

Kuva 23 esittää DescriptorUtil.runDescriptor-funktion toteutuksen uudella arkkitehtuurilla. Toteutus on yksinkertaisempi IRequestResponseConnectionin ja

uusittujen IO-kuvaajien ansiosta. Virheet on helpompi yhdistää lähetettyyn viestiin käytetyn koodirakenteen ansiosta. Tarkastukset ovat loogisesti siirtyneet vertailijoiden vastuulle, kuten puuttuvan vastauksen tai yhteysvirheiden tarkastaminen.

```
10⊖     public static <T extends IMessage> void runDescriptor(TestResults results,
11              IRequestResponseConnection<T> connection, IOSequence<T, IComparator<T>> sequence) {
12
13          for (IO<T, IComparator<T>> io : sequence.getIOs()) {
14              T response = connection.send(io.getInput());
15
16              List<Difference> differences = io.getOutput().compareTo(response);
17              results.storeDifferences(differences);
18          }
19      }
20  }
```

Kuva 23 Descriptor.runDescriptor-funktion toteutus uudella arkkitehtuurilla

4.4 Sovelluskehyksen käyttö

Sovelluskehyksen käyttöä esiteltiin esimerkkitestin kautta kohdassa 3.6. Esimerkkitestissä nähtiin yhteyksien luominen ja yksinkertaista testilogiikkaa DescriptorUtil.runDescriptor-funktion kautta. Kolmantena kehityssuunnitelmana on testilogiikan toteuttamisen yksinkertaistaminen.

4.4.1 Analyysi

Sovelluskehys on rakennettu hyvin matalalle tasolle, testilogiikka muodostuu pääasiassa yhteyksien käytöstä. Sovelluskehys on yleiskäyttöinen siinä mielessä, että se ei tiedä EP:n liiketoimintalogiikasta juuri mitään. Suunnittelun kannalta ratkaisu on looginen, koska luokilla on tiukasti rajattu toiminnallisuus. SOAP-yhteyden ei kuulukaan tietää EP:n liiketoimintalogiikasta mitään, kunhan se pystyy lähettämään ja vastaanottamaan SOAP-viestejä.

Sovelluskehyksen käytön eli testien toteuttamisen kannalta sovelluskehyksen painotus matalan tason komponentteihin ja yleiskäyttöisyyteen on epäkäytännöllinen. Testin kirjoittajan vastuulle jää kaikesta liiketoimintalogiikasta huolehtiminen. Esimerkiksi monessa testissä on tarpeen odottaa, että tietty viesti vastaanotetaan tietyllä yhteydellä. Sovelluskehys ei tarjoa logiikkaa viestin odotukseen ja virhetilanteiden käsittelyyn, joten testeihin on kirjoitettu lähes sama koodi useaan kertaan. Myös monessa testissä toistuu sama toiminnallisuus, vaikka se ei ole oleellinen osa testiä. Esimerkiksi datan tilaaminen asiakkaalle on usein toistuva asia, mutta datan tilaamista testaa varsinaisesti vain pari testiä. Muut testit käyttävät datan tilaamista testaamaan jotain muuta liiketoimintalogiikkaa. Sovelluskehys voisi auttaa usein toistuvien toiminnallisuuksien toteutuksessa, kuten viestin odotus ja datan tilaaminen.

Toinen ongelma on testien huono luettavuus. Yhtenä testijärjestelmän tavoitteena on testien korkea laatu. Luettavuus on yksi osa laatukriteerejä, ja on tärkeää esimerkiksi tarkastusten kannalta, että testikoodi on luettavaa. Sovelluskehyksen matalan tason komponentit eivät ole hyviä luettavuuden kannalta, koska yksi testilogiikan looginen

operaatio voi muodostua koodissa monesta sovelluskehyksen matalan tason operaatiosta. Tällöin testilogiikan tarkoitusta on hankala lukea koodista, koska loogiset operaatiot peittyvät matalan tason operaatioiden alle. Ongelmaa voi kuitenkin pienentää käyttämällä hyviä ohjelmointitapoja, esimerkiksi nimeämällä funktiot ja parametrit kuvaavasti.

4.4.2 Ratkaisu

Pääosana ratkaisua on suunnitella sovelluskehykseen EP:n asiakasta simuloiva luokka (TestClient). Luokan tarkoituksena on tarjota testin kirjoittajalle yksi korkeamman abstraktiotason rajapinta, jolla voidaan suorittaa suurin osa testien vaatimista toiminnallisuuksista. TestClient piilottaa testin kirjoittajalta yhteyksien käytön, sekä tarjoaa usein käytetyille toiminnallisuuksille valmiit funktiot. Testien toteuttamisen tuottavuus nousee käyttämällä TestClientin korkean abstraktiotason rajapintaa [37]. Tapauksissa joissa TestClientin rajapinta ei ole tarpeeksi joustava testien toteuttamiseen, voidaan edelleen käyttää suoraan yhteyksiä.

Toinen osa ratkaisua on toteuttaa TestClient Scalalla. TestClientin rajapinnan takia on oleellista, että Scala tarjoaa mahdollisuuden käyttää funktioita funktioiden parametreina. Koska luokan rajapintaan tulee Scalan ominaisuuksia, joita Javassa ei ole, myös TestClienttiä käyttävät testit on toteutettava Scalalla. Tavoitteena on luoda TestClientin käyttöön täsmäkieli, joka mukailee luonnollista kieltä.

Seuraavaksi esitellään TestClientin toiminnallisuus. Toiminnallisuus ei koostu pelkästään yksinkertaisista funktioista, joten toiminnallisuus esitetään operaatioina. Operaatioilla tarkoitetaan funktiokutsujen sarjaa, joka toteuttaa tietyn toiminnallisuuden. Esittelyissä parametrit on merkitty <- ja >-merkkien sisään. Ei-pakolliset elementit on merkitty {- ja }-merkkien sisään. Tyyppikuvauksissa paluuarvona voi olla Unit, joka on Scalan vastine Javan void-avainsanalle.

Taulukko 2 esittää send-operaation määrityksen. Send-operaatiolla lähetetään viestejä ja tarkastetaan vastaukset. Operaatio suoritetaan ketjuttamalla funktiokutsuja, kuten kaikki TestClientin operaatiot. Käytännössä ketjutus tapahtuu niin, että funktio palauttaa olion, jolla on jäsenfunktionaan seuraavaksi kutsuttava funktio. Ketjun keskellä olevat funktiot ottavat talteen operaatiokutsun parametrit ja viimeinen funktiokutsu suorittaa operaation. Operaatioissa käytetään uudistettua yhteyksien arkkitehtuuria hyväksi endpoint-parametrien kautta. Endpoint-parametrit saadaan joko TestClientilta itseltään tai Spring-ympäristöstä.

Taulukko 2 send-operaatio

Operaatio	send <msg> {to <endpoint>} expect <comparator>	
Kuvaus	Lähettää viestin EP:lle ja olettaa saavansa tietyn vastauksen. Jos vastaus ei ole oletettu, testi epäonnistuu.	
Paluuarvo	Vastaanotettu viesti, tyyppi sama kuin msg-parametrilla.	
Parametri	**Tyypit**	**Tarkoitus ja lisätiedot**
msg	SOAPMessage, BPMessage, LDAPMessage, MaintenanceMessage, MonitoringMessage	Lähetettävä viesti.
endpoint	HTTPSEndpoint, UDPEndpoint, LDAPEndpoint	Minne viesti lähetetään. TestClient valitsee yhteyden endpointin perusteella. Endpointin tyyppi on sidoksissa lähetettävän viestin tyyppiin. Oletusarvo määräytyy lähetettävän viestin tyypin perusteella.
comparator	IComparator	Oletettu vastaus. Comparatorin tyyppiparametri on sidoksissa lähetettävän viestin tyyppiin. Esimerkiksi SOAPMessagen tapauksessa comparatorin tyyppi on IComparator[SOAPMessage]

Kuva 24 esittää esimerkkejä send-operaation käytöstä. Operaatiokutsussa huomionarvoista on, että funktiokutsuista puuttuvat pisteet ja sulut, operaatiokutsuissa käytetäänkin aiemmin esiteltyä Scalan funktioiden vaihtoehtoista kutsumistapaa. Rivit 18 ja 19 suorittavat täysin saman operaation vaihtoehtoisilla kutsumistavoilla. Pisteetön ja suluton muoto on sen luettavuuden vuoksi suositeltava vaihtoehto.

```
18    client send msg expect expectedMsg
19    client.send(msg).expect(expectedMsg) // Same as above
20    val response = client send msg to httpsEndpoint expect expectedMsg
```

Kuva 24 send-operaation esimerkkejä

Taulukko 3 esittää gwSend-operaation määrityksen. Operaatio on lähdes sama kuin ITransmittingConnectionin send-funktio, jota TestClient käyttää toteutuksessaan. GwSend-operaatiolla lähetetään viestejä GW:ltä EP:lle.

Taulukko 3 gwSend-operaatio

Operaatio	gwSend <msg>	
Kuvaus	Lähettää viestin GW:ltä EP:lle. Joka TestClientillä on määritetty yhteys GW:lle, jota käytetään viestin lähettämiseen.	
Paluuarvo	Unit	
Parametri	**Tyypit**	**Tarkoitus ja lisätiedot**
msg	BPMessage	Lähetettävä viesti.

Taulukko 4 esittää execute-operaation määrityksen. Operaatiolla voidaan suorittaa kokonaisia viesti-vastaus-sekvenssejä. Operaatio vastaa DescriptorUtil.runDescriptor-funktiota (katso kuva 13).

Taulukko 4 execute-operaatio

Operaatio	execute <sequence> {to <endpoint>}	
Kuvaus	Lähettää kaikki viesti-vastaus-sekvenssissä olevat viestit EP:lle ja odottaa saavansa sekvenssissä määritellyt vastaukset. Jos kaikki vastaukset eivät ole oletetut, testi epäonnistuu.	
Paluuarvo	Vastaanotetut viestit. Tyyppi List[A], jossa A sama tyyppi kuin sequence-parametrin tyyppiparametri.	
Parametri	**Tyypit**	**Tarkoitus ja lisätiedot**
sequence	IOSequence[SOAPMessage] IOSequence[BPMessage] IOSequence[LDAPMessage] IOSequence[MaintenanceMessage] IOSequence[MonitoringMessage]	Suoritettava viesti-vastaus-sekvenssi. Viesti-vastaus-sekvenssi sisältää yhden tain useamman tietyssä järjestyksessä olevan syöte-vaste-parin. Katso kuva 19.
endpoint	HTTPSEndpoint, UDPEndpoint, LDAPEndpoint	Minne viesti lähetetään. Oletusarvo määräytyy lähetettävän viestin tyypin perusteella.

Taulukko 5 esittää react-operaation määrityksen. React-operaatiolla voidaan TestClient asettaa reagoimaan saapuviin viesteihin halutulla tavalla. Operaatiossa voidaan määrittää ohitettavat viestit ja viestien odotuksen maksimikesto.

Taulukko 5 react-operaatio

Operaatio	react <times> to <comparator> {ignoring <ignored>} {waiting <duration>} on <endpoint> by <action>	
Kuvaus	Asettaa TestClientin tilaan, jossa se reagoi saapuvaan viestiin. Kutsuminen ei odota reagoinnin tapahtumista vaan palaa heti.	
Paluuarvo	React. Reactilla voi lopettaa viesteihin reagoimisen cancel-funktiolla tai odottaa reagointia await-funktiolla. Funktio await palauttaa ReactResult-olion, josta voidaan hakea vastaanotetut viestit. ReactResult-olio myös kertoo onnistuiko reagointi annetussa aikamääreessä.	
Parametri	**Tyypit**	**Tarkoitus ja lisätiedot**
times	Int	Kuinka monta kertaa reagoidaan.
comparator	IComparator[BPMessage]	Mihin viestiin reagoidaan.
ignored	IComparator[BPMessage]	Mihin viesteihin ei reagoida. Tarkastetaan ennen comparatoria. Oletusarvona ignored ei vastaa mitään viestiä, joten reagoidaan kaikkiin viesteihin.
duration	Long	Kuinka monta millisekuntia odotetaan viestiä. Jos viesti ei saavu ajoissa ReactResult-olio kertoo reagoinnin epäonnistuneen. Oletusarvona odotetaan ikuisesti (mikä tahansa negatiivinen kokonaisluku).
endpoint	UDPEndpoint	Mihin viestiä odotetaan. TestClient valitsee yhteyden endpointin perusteella.
action	ReactEvent => Unit (funktio, joka ottaa parametrikseen ReactEventin ja palauttaa Unitin)	Funktio suoritetaan kun odotettu viesti saapuu. Funktio saa parametrina ReactEvent-olion, jossa on saapunut viesti. ReactEventillä on cancelReact-funktio, jolla reagointi voidaan lopettaa aikaisemmin tapauksessa jossa odotetaan useampaa viestiä.

Kuva 25 esittää esimerkkejä react-operaation käytöstä. Riveillä 27-29 on yksinkertainen operaatiokutsu. Huomionarvoisesti times-parametrina on käytetty once-sanaa. Once on yksinkertaisesti Int-tyyppinen olio arvolla 1.

Reagoinnissa suoritettava funktio määritellään aaltosulkeisiin useammalle riville. Scalan funktiokutsussa tavalliset sulut voidaan korvata aaltosulkeilla jos funktio ottaa vain yhden parametrin. Määritetty funktio annetaan by-funktiolle parametrina ja funktio on määritelty funktioliteraalina käyttämällä =>-symbolia.

Rivillä 30 on käytetty times parametrina 5.times-merkintää ja duration-parametrina 2.5.mins-merkintää. Kummassakin tapauksessa käytetään Martin Oderskyn kehittämää Pimp My Library –tekniikkaa. [38]. Tekniikalla voidaan lisätä olemassa oleviin luokkiin

uutta toiminnallisuutta implisiittisten funktioiden avulla. Esimerkiksi duration-parametrin tapauksessa arvo 2.5 on Double-tyyppinen olio. Double-olio muunnetaan implisiittisellä funktiolla toiseksi olioksi, jolla on mins-funktio. Mins-funktio taas palauttaa waiting-funktion vaatiman Long-tyyppisen arvon, tässä tapauksessa 150000.

Riveillä 38-41 on esimerkki react-operaation tulosten odottamisesta. Tuloksia odotetaan await-funktiolla, joka palauttaa ReactResult-olion. Esimerkissä ReactResultista haetaan vastaanotetut viestit.

Riveillä 43-48 on esimerkki reagoinnin lopettamisesta React-olion kautta. Operaatio palauttaa React-olion, jolla on funktio cancel. Funktiolla cancel voidaan lopettaa viesteihin reagointi ennenaikaisesti.

```
27  client react once to expectedMsg on udpEndpoint by { event =>
28    println(event.message)
29  }
30  client react 5.times to expectedMsg ignoring ignoredMsg waiting 2.5.mins on udpEndpoint by { event =>
31    println(event.message)
32  }
33  client react indefinitely to expectedMsg on udpEndpoint by { event =>
34    if (something()) {
35      event.cancelReact()
36    }
37  }
38  val reactResult = client react 3.times to expectedMsg on udpEndpoint by { event =>
39    // Do something
40  } await()
41  println(reactResult.results) // reactResults.results is of type List[BPMessage]
42
43  val react = client react indefinitely to expectedMsg on udpEndpoint by { event =>
44    println(event.message)
45  }
46  if (something()) {
47    react.cancel()
48  }
```

Kuva 25 react-operaation esimerkkejä

Taulukko 6 esittää expect-operaation määrityksen. Operaatiolla odotetaan saapuvia viestejä ja tarkastetaan, että viestit olivat odotuksien mukaisia. Expect-operaatio perustuu react-operaatioon, eli expect-operaatio voidaan suorittaa react-operaation avulla. Koska expect-operaation toiminnallisus on usein käytetty, on käytännöllistä tehdä sille oikotie.

Taulukko 6 expect-operaatio

Operaatio	expect <times> of <comparator> {ignoring <ignored>} {waiting <duration>} on <endpoint>	
Kuvaus	Asettaa TestClientin tilaan, jossa se odottaa seuraavan viestin olevan määritellyn mukainen. Jos viesti ei ole odotettu, testi epäonnistuu.	
Paluuarvo	Expect. Expectillä voi lopettaa viestin odotuksen cancel()-funktiolla tai odottaa viestiä await-funktiolla. Funktio await palauttaa vastaanotetut viestit.	
Parametri	**Tyypit**	**Tarkoitus ja lisätiedot**
times	Int	Kuinka montaa viestiä odotetaan.
comparator	IComparator[BPMessage]	Mitä viestiä odotetaan. Jos viesti ei ole odotettu, testi epäonnistuu.
ignored	IComparator[BPMessage]	Mitkä viestit sivuutetaan. Tarkastetaan ennen comparatoria. Oletusarvona ignored ei vastaa mitään viestiä, joten kaikki viestit tarkastetaan.
duration	Long	Kuinka monta millisekuntia odotetaan viestiä. Jos viesti ei saavu ajoissa, testi epäonnistuu.
endpoint	UDPEndpoint	Mihin viestiä odotetaan. TestClient valitsee yhteyden endpointin perusteella.

Kuva 26 esittää esimerkin expect-operaation käytöstä. Kuten esimerkistä näkee, expect-operaatio muistuttaa react-operaatiota, mutta reagoinnille ei tarvitse antaa itse funktiota, vaan reagointifunktio asetetaan automaattisesti tyhjäksi.

```
64    client expect 5 of expectedMsg ignoring ignoredMsg waiting 5.s on udpEndpoint
65    val receivedMessages = client expect one of expectedMsg on udpEndpoint await
```

Kuva 26 expect-operaation esimerkkejä

Taulukko 7 esittää sendThrough-operaation SOAP-version määrityksen. Tässä versiossa TestClient lähettää EP:lle SOAP-viestin, EP lähettää GW:lle BP-viestin, GW vastaa EP:lle BP-viestillä ja EP vastaa TestClientille SOAP-viestillä. Operaatiossa SOAP- ja BP-viestit vastaavat semanttisesti toisiaan, koska EP muuntaa SOAP-viestit BP viesteiksi keskustellessaan GW:n kanssa ja toisin päin keskustellessaan TestClientin kanssa. Operaatio voidaan toteuttaa myös yhdistelemällä send, expect ja gwSend –operaatioita. SendThrough-operaatio on kuitenkin niin usein käytetty, että on kätevää tehdä sille oikotie.

Taulukko 7 sendThrough-operaatio (SOAP)

Operaatio	sendThrough <msg1> {to <endpoint>} gwExpects <comparator1> returns <msg2> expect <comparator2>	
Kuvaus	Yleinen tapaus jossa EP:lle lähetettävä viesti ohjatuu sellaisenaan GW:lle. GW odottaa saavansa lähetetyn viestin ja vastaa määrätyllä viestillä. TestClient odottaa saavansa GW:n vastaamaan viestin vastauksena TestClientin lähettämään viestiin. Jos TestClientin tai GW:n vastaanottamat viestit eivät ole oletetut, testi epäonnistuu.	
Paluuarvo	Vastaanotettu viesti, tyyppi sama kuin msg1-parametrilla.	
Parametri	**Tyypit**	**Tarkoitus ja lisätiedot**
msg1	SoapMessage	Lähetettävä viesti.
endpoint	HTTPSEndpoint	Minne viesti lähetetään. TestClient valitsee yhteyden endpointin perusteella.
comparator1	IComparator[BPMessage]	Oletettu viesti jonka GW vastaanottaa. Jos viesti ei ole oletettu, testi epäonnistuu.
msg2	BPMessage	GW:n vastaus EP:lle.
comparator2	IComparator[SOAPMessage]	Oletettu viesti jonka TestClient vastaanottaa. Jos viesti ei ole oletettu, testi epäonnistuu.

Taulukko 8 esittää sendThrough-operaation BP-version määrityksen. Tässä versiossa ei ole tarvetta määrittää GW:n käsittelemiä viestejä. EP päästää viestit sellaisenaan läpi GW:lle, jolloin viestit ovat samat kuin TestClientin käsittelemät viestit.

Taulukko 8 sendThrough-operaatio (BinaryProtocol)

Operaatio	sendThrough <msg1> {to <endpoint>} expect <msg2>	
Kuvaus	Yleinen tapaus jossa EP:lle lähetettävä viesti ohjataan sellaisenaan GW:lle. GW odottaa saavansa lähetetyn viestin ja vastaa määrätyllä viestillä. TestClient odottaa saavansa GW:n vastaamaan viestin vastauksena TestClientin lähettämään viestiin. Jos TestClientin tai GW:n vastaanottamat viestit eivät ole oletetut, testi epäonnistuu.	
Paluuarvo	Vastaanotettu viesti, tyyppi sama kuin msg1-parametrilla.	
Parametri	**Tyypit**	**Tarkoitus ja lisätiedot**
msg1	BPMessage	Lähetettävä viesti. GW odottaa saavansa saman viestin EP:ltä. GW:n viestin vertailuu käytetään BPEqualsComparatoria, joka muodostetaan tästä parametrista. Jos viesti ei ole oletettu, testi epäonnistuu.
endpoint	HTTPSEndpoint	Minne viesti lähetetään. TestClient valitsee yhteyden endpointin perusteella.
msg2	BPMessage	Oletettu viesti jonka TestClient vastaanottaa. Vertailuun käytetään BPEqualsComparatoria, joka muodostetaan tästä parametrista. Myös GW:n vastaus EP:lle. Jos viesti ei ole oletettu, testi epäonnistuu.

Esiteltyjen operaatioiden lisäksi TestClientiin voidaan toteuttaa muuta yleisesti käytettyä toiminnallisuutta, kuten yhteyden avaaminen ja sulkeminen sekä datan tilaaminen.

TestClientissä on kaksi mahdollista ongelmaa. Ensimmäinen ongelma liittyy operaatioiden suorittamiseen: Operaatio voidaan jättää suorittamatta, jos testin kirjoittaja unohtaa määrittää viimeisen pakollisen parametrin. Kuva 27 esittää ongelman.

```
60      client react once to expectedMsg // Does not do anything, missing by <action>
```

Kuva 27 Keskeneräinen operaatiokutsu

Kuvassa operaation kutsuja on unohtanut antaa viimeisen action-parametrin. Viimeisen parametrin antaminen operaation kutsussa samalla myös suorittaa operaation. Kääntäjä ei varoita ongelmasta, koska se on syntaksiltaan korrektia Scala-koodia. Jos ongelma näyttää olevan suuri, mahdollinen ratkaisu on lisätä erillinen suoritusfunktio operaatioiden loppuun. Esimerkiksi funktio ”done” voitaisiin lisätä joka operaation loppuun, jolloin olisi helpompi huomata milloin operaatio jää suorittamatta.

Toinen ongelma liittyy Scalan käyttöön. Scalaa ei välttämättä haluta käyttää testilogiikan toteutukseen, ja siinä tapauksessa TestClientiin on tehtävä Java-rajapinta. Ainoa TestClientin rajapinnassa käytetty Scalan ominaisuus, jota ei voida suoraan

toteuttaa Javalla, on funktion antaminen parametrina. Funktioparametrit voidaan korvata anonyymeillä luokilla käyttäen Komento-suunnittelumallia.

4.5 Kehityssuunnitelmien arviointi

Arviointia tehdään kolmen esimerkin avulla. Mahdollisimman kattavan, mutta tiiviin arvioinnin saavuttamiseksi esimerkit eivät ole kokonaisia testejä vaan osia testeistä, jotka on valittu niiden kattaman usein käytetyn toiminnallisuuden perusteella. Arviointi tehdään vertailun avulla niin, että ensin esitellään esimerkki vanhalla sovelluskehyksellä toteutettuna ja sen jälkeen uudella sovelluskehyksellä toteutettuna. Vanhalla sovelluskehyksellä tarkoitetaan nykyistä testaussovelluskehystä, ja uudella sovelluskehyksellä tarkoitetaan testaussovelluskehystä, johon on toteutettu esitetyt kehityssuunnitelmat.

4.5.1 Datankäsittelyn ja yhteyksien esimerkki

Ensimmäinen esimerkki keskittyy datankäsittelyyn ja yhteyksiin. Esimerkin tavoitteena on alustaa testin tarvitsemat oliot, pyytää kuvitteellisen laskurin (counter) arvoa EP:ltä, suorittaa muuta testausta ja lopuksi tarkastaa, että laskurin arvo on kasvanut yhdellä. Kuva 28 esittää ensimmäisen esimerkin vanhalla sovelluskehyksellä toteutettuna.

```
16  IODescriptorParser parser = context.getBean("IODescriptorParser", IODescriptorParser.class);
17  IODescriptorHolder holder = parser.parse(descriptorFilename);
18
19  BPConnectionFactory bpFactory = context.getBean("SOAPConnectionFactory_client1", BPConnectionFactory.class);
20  Properties connectionProperties = createConnectionProperties(443, "path/to/asterix/interface");
21  BPConnection bpConnection = bpFactory.getConnection(connectionProperties);
22  connections.add(bpConnection);
23  bpConnection.open();
24
25  BPMessage request = holder.getInputs("CounterRequest", BPMessage.class).get(0);
26  bpConnection.send(request);
27  BPMessage responseBefore = bpConnection.getMessages().get(0);
28  int counterBefore = getCounter(responseBefore);
29
30  doTesting();
31
32  bpConnection.send(request);
33  BPMessage responseAfter = bpConnection.getMessages().get(0);
34  int counterAfter = getCounter(responseAfter);
35  results.storeAssertion("counter should be increased by one", counterBefore + 1, counterAfter);
```

Kuva 28 Esimerkki 1, vanha sovelluskehys

Riveillä 16-17 jäsennetään IO-kuvaaja ja riveillä 19-23 luodaan ja avataan yhteys EP:lle. Riveillä 25-28 haetaan lähetettävä viesti IO-kuvaajasta, lähetetään se EP:lle, luetaan vastaus ja tallennetaan laskurin arvo. Rivin 30 doTesting-funktiokutsun esittää muuta testausta, joka aiheuttaa laskurin kasvamisen. Riveillä 32-35 haetaan laskurin arvo uudelleen ja vertaillaan sitä vanhaan arvoon. Vertailu on tehtävä testilogiikassa, koska IO-kuvaajien ajonaikainen konfiguroiminen ei ole mahdollista. Esimerkin IO-kuvaajaa on esitetty kuvassa 29.

```
2⊖<test>
3⊖    <iodescriptor id="CounterRequest" type="binaryProtocol">
4⊖        <inputs>
5             <bpInput name="CounterRequest" />
6         </inputs>
7⊖        <outputs>
8             <!-- Checked in code -->
9         </outputs>
10    </iodescriptor>
11 </test>
```

Kuva 29 Esimerkki 1 IO-kuvaaja, vanha sovelluskehys

IO-kuvaajassa määritellään lähetettävä viesti, mutta vastausta ei määritellä, koska sitä ei voida tarkastaa kuvaajassa. Kuva 30 esittää ensimmäisen esimerkin uudella sovelluskehyksellä toteutettuna.

```
25      val descriptor = TestDescriptor
26      val client = context.getBean("TestClient1", classOf[TestClient])
27
28      val response :: Nil = client execute descriptor.counterBefore
29      val counterBefore = getCounter(response)
30      doTesting()
31      client execute descriptor.counterAfter(counterBefore)
```

Kuva 30 Esimerkki 1, uusi sovelluskehys

IO-kuvaajan jäsentäminen on karsiutunut kokonaan pois ja yhteyksien luominen sekä avaaminen on kiteytynyt TestClientin hakemiseen Springin ympäristöstä. Riveillä 28-29 haetaan lähetettävä viesti IO-kuvaajasta, lähetetään se EP:lle, luetaan vastaus ja tallennetaan laskurin arvo. Rivillä 31 kysytään laskurin arvoa uudelleen, tällä kertaa IO-kuvaajassa oleva vertailija konfiguroidaan ajonaikaisesti antamalla vanha laskurin arvo parametrina. IO-kuvaaja on esitetty kuvassa 31.

```
34⊖ object TestDescriptor {
35⊖   def counterBefore = new IOSequence(
36      BPMessage(dataFor[CounterRequest])
37      -> BPTypeComparator[CounterResponse]()
38    )
39
40⊖   def counterAfter(counterBefore: Int) = new IOSequence(
41      BPMessage(dataFor[CounterRequest])
42      -> BPFieldComparator[CounterResponse](List(
43        data => ("counter should be increased by one", counterBefore + 1 == data.counter)
44      ))
45    )
46  }
```

Kuva 31 Esimerkki 1 IO-kuvaaja, uusi sovelluskehys

IO-kuvaajan counterBefore-funktio määrittää viestisekvenssin, jossa on yksi lähetettävä viesti. Vastaanotettua viestiä vertaillaan ainoastaan tyypin perusteella. Kuvaajan counterAfter-funktio määrittää konfiguroitavan viestisekvenssin. Vertailijalle täytyy antaa aiempi laskurin arvo, jotta vertailija osaa tarkastaa, että arvo on kasvanut yhdellä. Laskurin tarkastus tehdään IO-kuvaajassa testilogiikan sijaan, mikä parantaa IO-kuvaajan ymmärrettävyyttä.

Datankäsittelyn yhtenä ongelmana oli muutosherkkyys. Vanhan sovelluskehyksen IO-kuvaajissa viestimääritykset ovat XML:ssä tekstinä, joten esimerkiksi IDE:n tekemä kääntäminen ei huomaa oliomalleihin tulleita muutoksia. Myöskin testilogiikassa käytettävät viestisekvenssien nimimuutokset jäävät huomaamatta käännösaikana. Uudessa sovelluskehyksessä ongelma on ratkaisu, koska sekä IO-kuvaaja että nimiviitteet ovat koodia. Kuva 32 esittää tapauksen, jossa viestin oliomallia on muutettu.

```
41  def counterAfter(counterBefore: Int) = new IOSequence(
42    BPMessage(dataFor[CounterRequest])
43    -> BPFieldComparator[CounterResponse](List(
44      data => ("counter should be increased by one", counterBefore + 1 == data.counter)
45    ))
46  )
```

Kuva 32 Esimerkki 1, uusi sovelluskehys, oliomallin muutos

CounterResponse-luokalta on poistettu counter-jäsenmuuttuja. Virhe ilmenee IO-kuvaajassa, joka käyttää viestin oliomallia. IDE ilmoittaa virheestä käännösaikana eli nykyaikaisessa IDE:ssä usein automaattisesti heti tiedoston tallentamisen jälkeen. Kuva 33 esittää tapauksen, jossa IO-kuvaajassa viestisekvenssin nimeä on muutettu.

```
28  val response :: Nil = client execute descriptor.counterBefore
29  val counterBefore = getCounter(response)
30  doTesting()
31  client execute descriptor.counterAfter(5)
```

Kuva 33 Esimerkki 1, uusi sovelluskehys, IO-kuvaajan muutos

Käytetyllä IO-kuvaajalla ei enää ole counterBefore-funktiota, joten IDE ilmoittaa virheestä automaattisesti. Vanhassa sovelluskehyksessä virhe huomattaisiin testin ajon aikana, mikä on ongelmallista testien hitauden takia.

4.5.2 Ensimmäinen testilogiikan toteutuksen esimerkki

Toinen esimerkki keskittyy testilogiikan toteuttamiseen. Esimerkin tavoitteena on asettaa testiasiakas reagoimaan tiettyihin EP:ltä saapuviin viesteihin lähettämällä viesti EP:lle ja tarkastamalla saatu vastaus, tehdä muuta testausta ja lopuksi tarkastaa, että EP:ltä saapui ainakin yksi viesti. Kuva 34 esittää esimerkin vanhalla sovelluskehyksellä toteutettuna.

```
62  final BPMessage msgToReact = holder.getOutputs("msgToReact", BPMessage.class).get(0);
63  final AtomicBoolean react = new AtomicBoolean(true);
64  final AtomicBoolean reacted = new AtomicBoolean(false);
65  ExecutorService executor = Executors.newSingleThreadExecutor();
66  executor.execute(new Runnable() {
67      public void run() {
68          while (react.get()) {
69              sleep(500);
70
71              List<BPMessage> messages = receivingConnection.getMessages();
72              for (BPMessage message: messages) {
73                  if (msgToReact.compareTo(message).isEqual()) {
74                      DescriptorUtil.runDescriptor(results, bpConnection, holder, "msgToSend", BPMessage.class);
75                      reacted.set(true);
76                  }
77              }
78          }
79      }
80  });
81
82  doTesting();
83
84  react.set(false);
85  executor.shutdown();
86  results.storeAssertion("should have reacted at least once", reacted.get());
```

Kuva 34 Esimerkki 2, vanha sovelluskehys

Riveillä 62-80 toteutetaan reagointi jättämällä toinen säie odottamaan viestejä. Odotus tehdään silmukassa, jossa jokaisella silmukan kierroksella tarkastetaan saapuneet viestit, vertaillaan viestejä reagoitavaan viestiin ja lähetetään EP:lle viesti, jos saapunut viesti vastasi reagoitavaa viestiä. Viestin lähetyksessä käytetään kuvassa 13 esiteltyä DescriptorUtil.runDescriptor-funktiota. Silmukan käyttö viestien odotuksessa on pakollista, koska yhteyden rajapinta ei tue muuta tapaa.

Alkuperäinen säie jatkaa testin suoritusta, kunnes se lopulta rivillä 84 ilmoittaa toiselle säikeelle, että reagoinnin voi lopettaa. Lopuksi rivillä 86 tarkastetaan, että reagoitiin vähintään yhteen viestiin. Kuva 35 esittää esimerkin uudella sovelluskehyksellä toteutettuna.

```
80      val msgToReact = descriptor.msgToReact
81      val react = client react indefinitely to msgToReact on endpoint by { event =>
82        client send descriptor.msgToSend expect descriptor.responseToExpect
83      }
84
85      doTesting()
86
87      react.cancel()
88      val messages = react.await().results
89      results.storeAssertion("should have reacted at least once", messages.size >= 1)
```

Kuva 35 Eimerkki 2, uusi sovelluskehys

Riveillä 80-83 asetetaan TestClient reagoimaan tiettyyn viestiin, reagoiminen tapahtuu automaattisesti toisessa säikeessä. TestClientin react-operaatiolle annetaan funktio, joka suoritetaan aina, kun reagoitava viesti on vastaanotettu. Rivillä 87 reagoiminen lopetetaan ja riveillä 88-89 tarkastetaan, että reagoitiin vähintään yhteen viestiin.

4.5.3 Toinen testilogiikan toteutuksen esimerkki

Kolmas esimerkki keskittyy myös testilogiikan toteuttamiseen. Esimerkin tavoitteena on ensin lähettää viesti EP:lle, tarkastaa vastaus ja sitten odottaa kolmea viestiä EP:ltä 30 sekunnin kuluessa. Testi epäonnistuu, jos saapuva viesti ei ole oletettu, eikä se ole ohitettavien viestien joukossa. Lopuksi tarkastetaan, että viestejä saapui vähintään kolme. Kuva 36 esittää esimerkin vanhalla sovelluskehyksellä toteutettuna.

```
94   DescriptorUtil.runDescriptor(results, bpConnection, holder, "msgToSend", BPMessage.class);
95
96   BPMessage msgToExpect = holder.getOutputs("msgToExpect", BPMessage.class).get(0);
97   BPMessage msgToIgnore = holder.getOutputs("msgToIgnore", BPMessage.class).get(0);
98   int expectedMessagesReceived = 0;
99   long startTime = System.currentTimeMillis();
100  while (System.currentTimeMillis() - startTime < 30000 && expectedMessagesReceived < 3) {
101      sleep(500);
102
103      List<BPMessage> messages = receivingConnection.getMessages();
104      for (BPMessage message: messages) {
105          if (!msgToIgnore.compareTo(message).isEqual()) {
106              if (msgToExpect.compareTo(message).isEqual()) {
107                  expectedMessagesReceived++;
108              } else {
109                  results.storeFailure("Received unexpected message " + message);
110              }
111          }
112      }
113  }
114
115  results.storeAssertion("Should have received 3 messages", expectedMessagesReceived >= 3);
```

Kuva 36 Esimerkki 3, vanha sovelluskehys

Ensimmäisen viestin lähetykseen ja vastauksen tarkastamiseen käytetään tuttua Descriptor.runDescriptor-funktiota rivillä 94. Viestien odotus tapahtuu riveillä 96-113 silmukassa, jossa lopetusehtona on 30 sekunnin aikarajan ylitys tai vähintään kolmen viestin saapuminen. Lopuksi rivillä 115 tarkastetaan, että viestejä saapui vähintään kolme. Kuva 37 esittää esimerkin uudella sovelluskehyksellä toteutettuna.

```
105      client send descriptor.msgToSend expect descriptor.responseToExpect
106
107      val msgToExpect = descriptor.msgToExpect
108      val msgToIgnore = descriptor.msgToIgnore
109      client expect 3 of msgToExpect ignoring msgToIgnore waiting 30.s on endpoint await()
```

Kuva 37 Esimerkki 3, uusi sovelluskehys

Ensimmäisen viestin lähetykseen ja vastauksen tarkastamiseen käytetään TestClientin send-operaatiota. Kolmen viestin odotukseen ja tarkastamiseen käytetään TestClientin expect-operaatiota. Expect-operaatiokutsun lopussa kutsutaan await-funktiota, koska expect-operaation suoritus siirtyy omaan säikeeseensä. Await-funktio odottaa operaation valmistumista.

4.5.4 Tulokset

Ensimmäisessä esimerkissä näytetty IO-kuvaajan jäsentäminen ja yhteyksien luominen on yksinkertaistunut uudessa arkkitehtuurissa huomattavasti. IO-kuvaajan jäsentämistä ei tarvitse enää tehdä ja TestClient konfiguroidaan Spring-ympäristön asetuksissa, jolloin yhteyksiin ei tarvitse suoraan koskea testissä. Tämä parantaa sekä testien luettavuutta että testien toteuttamisen tuottavuutta.

Ensimmäisessä esimerkissä näytettiin myös parannettua IO-kuvaajien konfiguroitavuutta. Konfigurointi tapahtuu yksinkertaisesti antamalla IO-kuvaajan funktioille parametreja. Vanhassa sovelluskehyksessä ajonaikainen viestien konfigurointi oli mahdotonta, mutta uudessa sovelluskehyksessä kaikki konfigurointi tapahtuu ajon aikana. Ajonaikainen konfigurointi mahdollistaa kaikenlaisten vertailuiden tekemisen IO-kuvaajien vertailijoissa. Tämä parantaa IO-kuvaajien ymmärrettävyyttä ja yhtenäistää vertailujen tekemistä.

Kolmas ensimmäisessä esimerkissä esitetty parannus liittyy testien muutosherkkyyden pienentämiseen. IO-kuvaajien siirto koodiin parantaa oleellisesti testien ylläpidettävyyttä, koska muutokset viestien oliomalleihin ja kuvaajien nimiin huomataan käännösaikana. Uuden sovelluskehyksen muutos parantaa oleellisesti testien ylläpidettävyyttä.

Toisessa ja kolmannessa esimerkissä keskityttiin testilogiikan toteuttamiseen. Testilogiikan paremmuutta on hankala arvioida, mutta esimerkiksi luettavuutta ja ylläpidettävyyttä voidaan yrittää arvioida lausekkeiden lukumäärän (LLOC, Logical Lines of Code) perusteella. Toinen yleisesti käytetty mitta on Thomas J. McCaben kehittämä syklomaattinen kompleksisuus [39]. Syklomaattinen kompleksisuus lasketaan käyttäen ohjelman kontrollivuoverkkoa, joka on suunnattu verkko, jossa solmut ovat peruslohkoja ja reunat siirtymiä lohkojen välillä. Syklomaattinen numero on kontrollivuoverkon teiden lukumäärä alkusolmusta loppusolmuun, joka vähintään tarvitaan, jotta jokainen solmu ja reuna tulee käytyä läpi ainakin kerran.

Syklomaattista kompleksisuutta vastaan on kuitenkin esitetty kritiikkiä [40] sen hyödyllisyydestä. On myös olemassa tutkimuksia [41, 42], joista selviää, että syklomaattisen kompleksisuuden ja lausekkeiden lukumäärän välillä on vahva korrelaatio. Korrelaatio tarkoittaa tässä tapauksessa sitä, että jos syklomaattinen kompleksisuus on suuri, niin myös lausekkeiden lukumäärä on suuri. Esitetyn kritiikin ja tutkimusten perusteella teemme vertailun ainoastaan lausekkeiden lukumäärän perusteella. Taulukossa 9 on esitelty esimerkkitoteutuksien lausekkeiden lukumäärät.

Taulukko 9 Esimerkkien LLOC toteutuksittain

	LLOC		
Esimerkki	**Vanha**	**Uusi**	**Erotus**
Toinen	16	7	9
Kolmas	14	4	10

Uudella sovelluskehyksellä toteutetut esimerkit ovat LLOC:ltaan yli 50% lyhyempiä kuin vanhalla sovelluskehyksellä toteutetut esimerkit. Parannus on vieläkin suurempi, jos vertaillaan fyysisiä rivejä, koska Scalan syntaksi auttaa vähentämään niin kutsutta pohjakoodia (boilerplate) eli koodia, jota on käytetty tai voidaan käyttää sellaisenaan uudessa kontekstissa. Vertailun perusteella voidaan sanoa, että uusi sovelluskehys parantaa testien toteuttamisen, ylläpitämisen ja tarkastamisen tuottavuutta huomattavasti.

5 YHTEENVETO

Tähän diplomityöhön kuului testausautomaatiojärjestelmän toteuttaminen annetun suunnitelman pohjalta ja kehityssuunnitelman laatiminen toteutetulle järjestelmälle. Aluksi selvitettiin testauksen roolia ja vaiheita ohjelmistokehityksessä sekä tarkasteltiin testausta sen automatisoinnin näkökulmasta. Järjestelmätestauksen osalta tutkittiin myös testaustyökalujen ja sovelluskehysten periaatteita. Testauksen apuvälineitä selvitettiin etenkin sovelluskehyksen toteuttamisen ja testien kirjoittamisen kannalta.

Toteutettu testausjärjestelmä koostuu sekä fyysisestä ympäristöstä eli laitteistosta, niitä ohjaavista tukisovelluksista ja testaussovelluskehyksestä. Kehityssuunnitelma päätettiin rajata sovelluskehyksen kehittämiseen, koska koko testausjärjestelmän kehittäminen oli liian suuri työ tämän diplomityön puitteissa. Sovelluskehyksen arkkitehtuuri koostuu kolmesta erillisestä osasta: datankäsittely, yhteydet ja raportointi. Järjestelmän toteutuksen ja käytön aikana kerätyn kokemuksen perusteella sovelluskehyksen sopivimmiksi kehityskohteiksi valikoitui datankäsittely- ja yhteydet-komponentit.

Datankäsittely-komponentista löydettiin useita kehitettäviä alueita esimerkiksi viesti-vastaus-rakenne, viestien konfiguroitavuus ja XML-tiedostojen heikko ylläpidettävyys. Kuitenkin ratkaisu, jossa data eroteltiin logiikasta, oli osoittautunut toimivaksi, joten sama ratkaisu säilytettiin uuteen arkkitehtuuriin. IO-kuvaajien rakennetta muutettiin loogisemmaksi, ja niiden toteutuskieli vaihdettiin XML:stä Scalaan. Scalan käyttöönotto mahdollisti myös viestien paremman konfiguroitavuuden, joka oli yksi kehityksen kohteista.

Yhteydet-komponentista havaittiin ongelmia yhteysrajapinnan käytössä. Rajapinta ei ollut intuitiivinen käyttää millään yhteystyypillä, koska rajapinnan suunnittelussa oltiin päädytty kompromissiin, jossa kaikki yhteydet toteuttivat saman rajapinnan. Tämä suunnitteluratkaisu osoittautui huonoksi, joten yhteyksien rajapinnat suunniteltiin uudelleen vastaamaan paremmin yhteyksien käyttömalleja. Myös yhteyksien luominen ja konfigurointi oli työlästä, mikä ratkaistiin yhdistämällä yhteystehtaat ja jakamalla konfigurointivastuut uudelleen.

Kolmantena kehitysideana suunniteltiin korkeamman abstraktiotason testiasiakasluokka, TestClient. Sovelluskehyksen tarjoamat yhteydet olivat matalan tason luokkia, jotka eivät juurikaan tarjonneet apua testilogiikan toteuttamiseen. TestClientillä pyrittiin piilottamaan yhteyksien käyttö ja tarjoamaan valmista testilogiikkaa, joka toimisi rakennuspalikoina varsinaisille testeille. TestClientin rajapinta suunniteltiin luonnollista kieltä mukailevaksi, koska tavoite oli tehdä

testikoodista luettavampaa. Rajapinta toteutettiin Scalalla, koska se mahdollisti kauniin rajapinnan tekemisen, mutta Scala on tarvittaessa korvattavissa Javalla pienellä työllä.

Alkuperäisessä testaussovelluskehyksessä havaittiin lukuisa määrä ongelmia, joihin kaikkiin löydettiin tyydyttävä ratkaisu. Kehityssuunnitelmia ei ole vielä toteutettu, mutta jos ne toteutetaan ja otetaan käyttöön, tulee se tehdyn arvioinnin perusteella parantamaan huomattavasti testien toteuttamisen, ylläpitämisen ja tarkastamisen tuottavuutta.

LÄHTEET

[1] C. Nagle, ”Test Automation Frameworks,” SAS Institute, [WWW]. Saatavissa: http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm. [Haettu 15.12.2011].

[2] R. D. Craig ja S. P. Jaskiel, Systematic Software Testing, Artech House, 2002.

[3] B. Hailpern ja P. Santhanam, ”Software debugging, testing, and verification,” *IBM Systems Journal,* osa/vuosik. 41, nro 1, pp. 4-12, 2002.

[4] RTI Health, Social, and Economics Research, ”The Economic Impacts of Inadequate Infrastructure for Software Testing,” Research Triangle Park, North Carolina, 2002.

[5] S. W. Ambler, ”Has Agile Peaked?,” Dr. Dobb's, 7.5.2008. [WWW]. Saatavissa: http://drdobbs.com/architecture-and-design/207600615. [Haettu 10.11.2011].

[6] I. Haikala ja J. Märijärvi, Ohjelmistotuotanto, Helsinki: Talentum, 2006.

[7] M. Fowler, ”Mocks Aren't Stubs,” 2.1.2007. [WWW]. Saatavissa: http://martinfowler.com/articles/mocksArentStubs.html. [Haettu 29.10.2011].

[8] K. Beck, Test-Driven Development, by example, Addison-Wesley Professional, 2003.

[9] M. Fowler, ”Continuous Integration,” 1.5.2006. [WWW]. Saatavissa: http://www.martinfowler.com/articles/continuousIntegration.html. [Haettu 30.10.2011].

[10] K. Maxwell, L. V. Wassenhove ja S. Dutta, ”Performance Evaluation of General and Company Specific Models in Software Development Effort Estimation,” *Management Science,* osa/vuosik. 45, nro 6, pp. 787-803, 1999.

[11] The Eclipse Foundation, ”Eclipse,” 2011. [WWW]. Saatavissa: http://www.eclipse.org/. [Haettu 12.11.2011].

[12] Oracle Corporation, ”Netbeans,” 2011. [WWW]. Saatavissa: http://netbeans.org/. [Haettu 12.11.2011].

[13] JetBrains, ”IntelliJ IDEA,” [WWW]. Saatavissa: http://www.jetbrains.com/idea/. [Haettu 12.11.2011].

[14] K. Koskimies ja T. Mikkonen, Ohjelmistoarkkitehtuurit, Helsinki: Talentum Media Oy, 2005.

[15] M. Fowler, ”Inversion Of Control,” 26.6.2005. [WWW]. Saatavissa: http://martinfowler.com/bliki/InversionOfControl.html. [Haettu 12.11.2011].

[16] M. Clark, ”JUnit FAQ,” 20.2.2006. [WWW]. Saatavissa: http://junit.sourceforge.net/doc/faq/faq.htm. [Haettu 12.11.2011].

[17] C. Kaner, ”Improving the Maintainability of Automated Test Suites,” *International Software Quality Conference*, San Francisco, 1997.

[18] M. Fewster ja D. Graham, Software Test Automation, Addison-Wesley Professional, 1999.

[19] C. Kaner, J. Bach ja B. Pettichord, Lessons Learned in Software Testing: A Context-Driven Approach, Wiley, John & Sons, Incorporated, 2011.

[20] J. Bach, ”Test Automation Snake Oil v2.1,” 13.6.1999. [WWW]. Saatavissa: http://www.satisfice.com/articles/test_automation_snake_oil.pdf. [Haettu 30.10.2011].

[21] World Wide Web Consortium, ”SOAP Version 1.2 Part 1,” 27.4.2007. [WWW]. Saatavissa: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. [Haettu 16.10.2011].

[22] World Wide Web Consortium, ”Extensible Markup Language (XML) 1.1 (Second Edition),” 16.8.2006. [WWW]. Saatavissa: http://www.w3.org/TR/2006/REC-xml11-20060816/. [Haettu 16.10.2011].

[23] OASIS, ”UDDI Version 3.0.2,” 19.10.2004. [WWW]. Saatavissa: http://uddi.org/pubs/uddi-v3.0.2-20041019.htm. [Haettu 16.10.2011].

[24] Internet Engineering Task Force, ”RFC 4510 - Lightweigth Directory Access Protocol (LDAP): Technical Specification Roadmap,” 6.2006. [WWW]. Saatavissa: http://tools.ietf.org/html/rfc4510. [Haettu 16.10.2011].

[25] H. Q. Nguyen, M. Hackett ja B. K. Whitlock, Happy About Global Software Test Automation: A Discussion of Software Testing for Executives, Happy About, 2006.

[26] ”Java Message Service,” 12.4.2002. [WWW]. Saatavissa: http://download.oracle.com/otn-pub/jcp/7195-jms-1.1-fr-spec-oth-JSpec/jms-1_1-fr-spec.pdf. [Haettu 16.10.2011].

[27] ”Hudson Continuous Integration,” [WWW]. Saatavissa: http://hudson-ci.org/. [Haettu 16.10.2011].

[28] J. M. Voas ja K. W. Miller, ”Software testability: the new verification,” *Software, IEEE,* osa/vuosik. 12, nro 3, pp. 17-28, 1995.

[29] SpringSource, ”Spring Core,” 2011. [WWW]. Saatavissa: http://www.springsource.org/spring-core. [Haettu 12.11.2011].

[30] E. Gamma, R. Helm, R. Johnson ja J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[31] P. Pillai, ”Introduction to Static and Dynamic Typing,” Sitepoint, 18.6.2004. [WWW]. Saatavissa: http://www.sitepoint.com/typing-versus-dynamic-typing/. [Haettu 3.12.2011].

[32] M. Odersky, ”The Scala Language Specification Version 2.9,” 24.5.2011. [WWW]. Saatavissa: http://www.scala-lang.org/docu/files/ScalaReference.pdf. [Haettu 14.10.2011].

[33] J. Gosling, B. Joy, G. Steele ja G. Bracha, ”The Java™ Language Specification, Third Edition,” 1.2005. [WWW]. Saatavissa: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf. [Haettu 28.10.2011].

[34] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith ja L. Walton, ”A Software Engineering Experiment in Software Component Generation,” *International Conference on Software Engineering*, Washington, DC, 1996.

[35] World Wide Web Consortium, ”XML Path Language (XPath) 1.0,” 16.11.1999. [WWW]. Saatavissa: http://www.w3.org/TR/1999/REC-xpath-19991116/. [Haettu 16.12.2011].

[36] J. Long, ”Software Reuse Antipatterns,” *Software Engineering Notes,* osa/vuosik. 26, nro 4, pp. 76-68, 2001.

[37] F. Brooks, The Mythical Man-Month, Boston: Addison-Wesley, 1975.

[38] M. Odersky, ”Pimp My Library,” 9.10.2006. [WWW]. Saatavissa: http://www.artima.com/weblogs/viewpost.jsp?thread=179766. [Haettu 9.10.2011].

[39] T. J. McCabe, ”A Complexity Measure,” *IEEE Transactions on Software Engineering,* osa/vuosik. 2, nro 4, pp. 308-320, 1976.

[40] M. Shepperd, ”A critique of cyclomatic complexity as a software metric,” *Software Engineering Journal,* osa/vuosik. 3, nro 2, pp. 30-36, 1988.

[41] V. R. Basili ja H. D. Hutchens, ”An Empirical Study of a Syntactic Complexity Family,” *IEEE Transactions on Software Engineering,* osa/vuosik. 9, nro 6, pp. 664-672, 1983.

[42] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst ja T. Love, ”Measuring the Psychological Complexity of Software Maintenance Tasks with Halstead and McCabe Metrics,” *IEEE Transactions on Software Engineering,* osa/vuosik. 5, nro 2, pp. 96-104, 1979.