Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka
Osmo Someroja

Opinnäytetyö

Jakeluverkon hallintajärjestelmän testaus

Työn ohjaaja lehtori Erkki Hietalahti
Työn tilaaja Bitwise Oy, ohjaajana tekninen johtaja Heikki Mäenpää
Tampere 5/2010

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma, Ohjelmistotekniikka

Tekijä	Osmo Someroja
Työn nimi	Jakeluverkon hallintajärjestelmän testaus
Sivumäärä	31
Valmistumisaika	toukokuu 2010
Työn ohjaaja	lehtori Erkki Hietalahti
Työn tilaaja	Bitwise Oy, ohjaajana tekninen johtaja Heikki Mäenpää

Tiivistelmä

DHR Finland Oy tarjoaa huoltoasemien kunnossapitopalveluja ja jakeluautomaatiojärjestelmiä. SmartNet on DHR Finlandin öljy-yhtiöille ja asemanomistajille tarjoama jakeluverkon hallintajärjestelmä, jonka ensimmäinen versio julkaistiin kesäkuussa 2006. Alkuperäinen SmartNet-järjestelmä kasvoi lopulta hankalasti ylläpidettäväksi ja laajennettavaksi, joten DHR Finland aloitti elokuussa 2008 yhteistyön Bitwise Oy:n kanssa SmartNet-järjestelmän uudistamiseksi.

Tässä työssä on käsitelty uudistetun SmartNet-järjestelmän testaamista. Työn tarkoituksena on luoda yleiskuva SmartNet-järjestelmän testauksessa käytettyihin menetelmiin ja työkaluihin.

Työssä on esitelty SmartNet-järjestelmän testaamista varten kehitettyjä testisimulaattoreita sekä testaamisen helpottamiseksi käyttöönotettuja työkaluja. Käyttöönotetuista työkaluista on esitelty tarkemmin NUnit- ja Rhino Mocks -ohjelmistokehykset ja Selenium-työkalut. Lisäksi työssä on käyty läpi tarkemmin SmartNet-järjestelmän testaamisen kannalta oleellisia yksikkötestaamisen tekniikoita ja menetelmiä.

Tätä työtä voidaan hyödyntää sekä DHR Finlandin että Bitwisen toimesta tarjoamalla SmartNet-järjestelmän uudistamisprojektissa aloittaville uusille kehittäjille eräänlainen tutustumisopas SmartNet-järjestelmän testaukseen.

Avainsanat testaus, yksikkötestaus, testisimulaattori, riippuvuusinjektio

Tampere University of Applied Sciences, Bachelor's Degree
Department of Computer Systems Engineering, Software Engineering

Writer	Osmo Someroja
Thesis	Testing of a Distribution Network Management System
Pages	31
Graduation time	May 2010
Thesis supervisor	lecturer Erkki Hietalahti
Co-operating company	Bitwise Oy, Technical Director Heikki Mäenpää as supervisor

Abstract

DHR Finland Oy offers service station maintenance services and distribution automation systems. SmartNet is a distribution network management system that DHR Finland offers to oil companies and station managers. The first version of SmarNet was released in June 2006. The original system ultimately grew to be difficult to maintain and extend, so DHR Finland started to co-operate with Bitwise Oy in August 2008 in order to renovate the SmartNet system.

This thesis deals with the testing of the SmartNet system. The main purpose is to present a general picture of the methods, techniques and tools used in the testing.

This thesis describes the test simulators and tools that were developed or selected to ease the testing of the SmartNet system. From the selected tools, NUnit and Rhino Mocks frameworks, and Selenium tools are discussed in more detail. Also unit testing methods and techniques that were relevant to the testing of the SmartNet system are explored further.

This thesis can be used by both DHR Finland and Bitwise to offer a introduction to the testing of the SmartNet system for new developers starting work on the SmartNet renovation project.

Keywords testing, unit testing, test simulator, dependency injection

Sisällysluettelo

1 Johdanto....6

2 SmartNet-järjestelmän kuvaus....8

3 SmartNet-järjestelmän testisimulaattorit....10

3.1 Asemasimulaattori....10

3.2 XML-tapahtumasimulaattori....11

3.3 EMV-simulaattori....12

4 SmartNet-järjestelmän komponenttien testaaminen....13

4.1 Yksikkötestaaminen....13

4.2 Yksikkötestaamisen tekniikat....14

4.2.1 Väärennökset....14

4.2.2 Saumat....15

4.2.3 Riippuvuusinjektio....15

4.2.4 Irrota ja peitä -tekniikka....17

4.3 SmartNet-järjestelmän komponenttien testauksen toteutuminen....20

5 SmartNet-järjestelmän WWW-käyttöliittymän testaaminen....27

5.1 Selenium IDE -työkalu....27

5.2 Selenium RC -työkalu....28

5.3 Selenium-työkalujen käyttö WWW-käyttöliittymän testaamisessa....29

6 Yhteenveto....30

Lähteet....31

Lyhenteiden ja termien luettelo

CUT	Class Under Test. Testattava luokka.
DBMS	Database Management System. Tietokannan hallintajärjestelmä.
DOC	Depended-On Component. Vaadittu komponentti.
EMV	Europay–Visa–Mastercard. Kansainvälinen standardi sirukorteille.
Kylmäasema	Miehittämätön, jakeluautomaatiojärjestelmän avulla toimiva huoltoasema.
MVC	Model–View–Controller. Malli–näkymä–ohjain. Eräs ohjelmisarkkitehtuurimalli.
ORM	Object-Relational Mapping. Olio–relaatio-mallinnus. Tekniikka mallintaa olioita relaatiotietokannan tyypeiksi.
TDD	Test-Driven Development. Testivetoinen kehitys. Eräs ketterä ohjelmistonkehitysmenetelmä.
VPN	Virtual Private Network. Näennäisesti yksityinen verkko. Tapa yhdistää sisäverkkoja julkista verkkoa hyödyntäen.

1 Johdanto

DHR Finland Oy, entiseltä nimeltään Oy Autotank Ab, on huoltoasemien kunnossapitopalveluja ja jakeluautomaatiojärjestelmiä tarjoava yritys. DHR Finland myy brändillä Gilbarco Autotank mm. polttoaineen myynti- ja maksupäätejärjestelmiä.

SmartNet on DHR Finlandin tarjoama jakeluverkon hallintajärjestelmä, jonka avulla DHR Finlandin asiakkaana oleva öljy-yhtiö tai asemanomistaja voi seurata ja hallita jakeluverkkonsa tilaa Internetin kautta. SmartNet on suunniteltu erityisesti kylmäasemia varten, ja sen avulla on esimerkiksi mahdollista seurata asemien tapahtumia, saada vikatilanteissa hälytys sähköpostilla tai tekstiviestillä ja suorittaa etänä huoltotoimenpiteitä.

SmartNetin ensimmäinen versio julkaistiin kesäkuussa 2006. Järjestelmää on sen jälkeen kehitetty jatkuvasti lisäämällä siihen uusia ominaisuuksia asiakkaiden toiveiden ja palautteen pohjalta. Alkuperäinen järjestelmä kasvoi lopulta hankalasti ylläpidettäväksi ja laajennettavaksi. Elokuussa 2008 DHR Finland aloitti yhteistyön Bitwise Oy:n kanssa SmartNet-järjestelmän uudistamiseksi. Tarkoituksena on tuottaa aiempaa monipuolisempi ja helpommin laajennettavissa oleva järjestelmä. Tämän työn valmistumishetkellä uusi SmartNet on versiossa 0.8.0.

Bitwise Oy on vuonna 2003 perustettu ohjelmistotalo, joka tarjoaa asiakkailleen ohjelmistokehitys-, asiantuntija- ja ylläpitopalveluita sekä koulutusta ja mentorointia. Bitwise soveltaa monissa projekteissaan ketteriä ohjelmistonkehitysmenetelmiä ja inkrementaalista kehitysmallia.

Itse olen toiminut SmartNet-järjestelmän uudistamisprojektissa suunnittelijana toukokuusta 2009 lähtien. Erityisinä vastuualueinani määrittelyn, suunnittelun, toteutuksen ja testauksen lisäksi ovat olleet SmartNet-järjestelmän testisimulaattoreiden kehitys sekä erilaisten testaustyökalujen kokeilu, arviointi ja käyttöönotto.

Tämä työ käsittelee uudistetun SmartNet-järjestelmän testaamista. Työn tavoitteena on luoda yleiskuva SmartNetin testauksessa käytettyihin menetelmiin ja työkaluihin sekä tarjota omia kokemuksia niiden käytöstä ja analysoida tehtyjä valintoja ja ratkaisuja.

Luvussa 2 kuvataan uudistetun SmartNet-järjestelmän rakenne ja listataan käytettyjä

teknologioita.

Luvussa 3 esitellään SmartNet-järjestelmän testaamista varten kehitetyt testisimulaattorit ja kuvaillaan niiden toimintaa.

Luvussa 4 käsitellään SmartNet-järjestelmän komponenttien testaamista. Luvussa käsitellään aluksi yksikkötestaamisen teoriaa, menetelmiä ja työkaluja, sen jälkeen näiden käyttöä SmartNetin komponenttien testaamisessa.

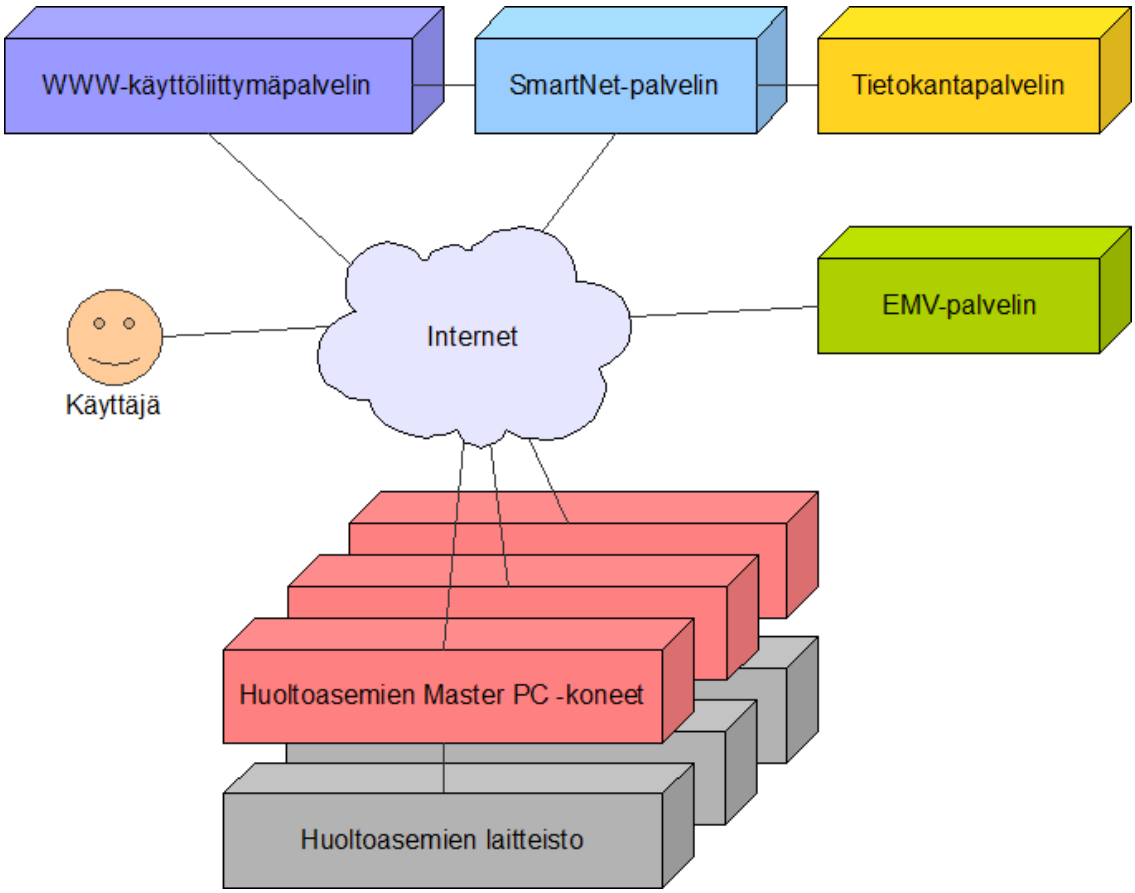
Luvussa 5 käsitellään SmartNet-järjestelmän WWW-käyttöliittymän testaamista. Luvussa esitellään tähän tarkoitukseen suunniteltuja työkaluja ja kerrotaan kokemuksia niiden käytöstä SmartNetin tapauksessa.

Luvussa 6 esitetään yhteenveto ja analyysi SmartNet-järjestelmän testauksen toteutumisesta. Luvussa pohditaan myös vaihtoehtoja tehtyihin ratkaisuihin ja tarkastellaan näiden vaihtoehtojen mielekkyyttä.

Työn kielestä poiketen työssä esiintyvissä ohjelmakoodiesimerkeissä kielenä on englanti, mutta ohjelmakoodi on kommentoitu suomeksi. Ohjelmakoodiesimerkit on esitetty C#-ohjelmointikielellä, jollei toisin ole mainittu. Työn lukijan oletetaan tuntevan olio-ohjelmoinnin ja C#-ohjelmointikielen perusteet.

2 SmartNet-järjestelmän kuvaus

Uudistettu SmartNet-järjestelmä koostuu viidestä osakokonaisuudesta (kuvio 1): SmartNet-palvelimesta, tietokantapalvelimesta, WWW-käyttöliittymäpalvelimesta, EMV (Europay–Visa–Mastercard) -palvelimesta, ja huoltoasemilla olevista Master PC -koneista ja huoltoasemien laitteistosta. Järjestelmän osat ovat näennäisesti yksityisessä verkossa (VPN) keskenään Internetin välityksellä.



Kuvio 1: Uudistetun SmartNet-järjestelmän rakenne

SmartNet-palvelin tarjoaa järjestelmän bisneslogiikan sekä kommunikaatiorajapinnat järjestelmän muille osille. Sen toiminnallisuutta voidaan laajentaa helposti liitännäisillä (plug-in) ja skripteillä. SmartNet-palvelin käyttää tietokantapalvelimella sijaitsevaa tietokantaa.

WWW-käyttöliittymäpalvelin tarjoaa SmartNet-järjestelmän käyttäjille Internet-selaimessa toimivan käyttöliittymän, jonka avulla käyttäjät voivat seurata ja hallita jakeluverkkoaan.

EMV-palvelin huolehtii kommunikoinnista pankkien EMV-järjestelmien kanssa. Sen ohjelmistokokonaisuus on kolmannen osapuolen toteuttama. Aseman Master PC -koneet käyttävät EMV-palvelinta sirukorttien varmentamiseen ja maksuliikennetietojen välittämiseen. SmartNet-palvelin hakee EMV-palvelimelta maksuliikennetietoja näytettäväksi WWW-käyttöliittymässä.

Huoltoasemilla olevat Master PC -koneet sisältävät vanhan SmartNet-järjestelmän osista koostuvan perinteisen järjestelmän (legacy system) ja uuden SmartNet-järjestelmän asemaohjelman. Perinteinen järjestelmä vastaa aseman laitteiston päivittäisestä toiminnasta ja kommunikoi XML-viestein uuden asemaohjelman kanssa. Uusi asemaohjelma käsittelee aseman maksutapahtumia ja ilmoituksia sekä vastaa kommunikaatiosta SmartNet-palvelimen kanssa.

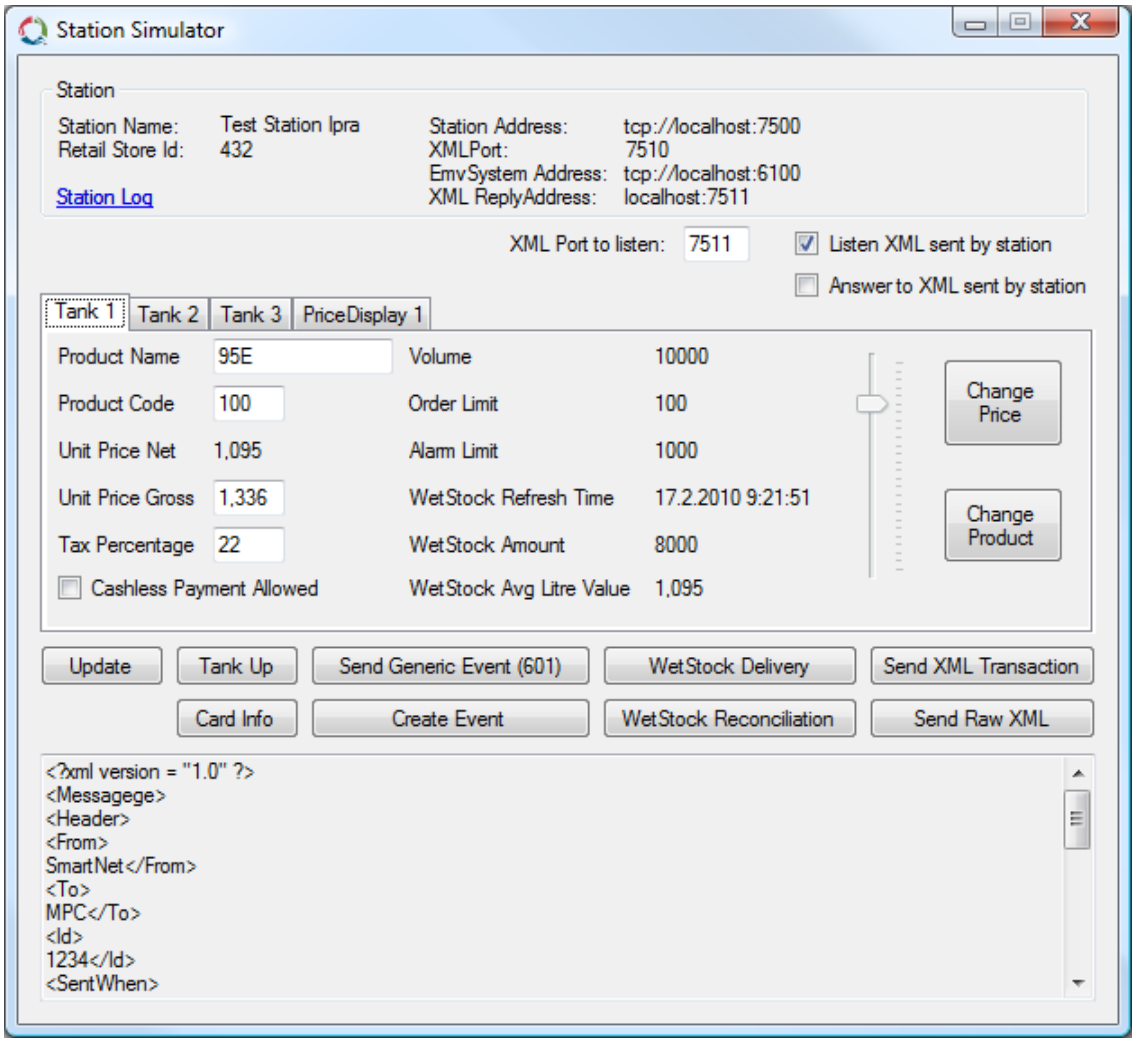
Uusi SmartNet-järjestelmä on toteutettu käyttäen Microsoftin .NET Framework -ohjelmistokomponenttikirjastoa ja C#-ohjelmointikieltä. Tämän lisäksi WWW-käyttöliittymäpalvelimella on käytetty SpringSourcen Spring.NET-ohjelmistokehystä. SmartNet-palvelimen skripteissä käytetään Lua-skriptikieltä. Tietokantapalvelimen tietokannan hallintajärjestelmänä (DBMS) on Microsoftin SQL Server 2005. Tietokantakäsittelyssä olio–relaatio-mallintamiseen (ORM) käytetään NHibernate-ohjelmistokehystä.

3 SmartNet-järjestelmän testisimulaattorit

SmartNet-järjestelmän testaamista varten on kehitetty erilaisia testisimulaattoreita, joiden tarkoitus on korvata jokapäiväisen testaamisen kannalta hankalia järjestelmän osia. Esimerkiksi huoltoasemalla oleva laitteisto voidaan simuloida, jolloin on mahdollista pystyttää tavallista yksinkertaisempia testipalvelimia ja järjestelmän kehittäjät voivat ajaa näennäisesti kokonaista järjestelmää myös omilla työasemillaan.

3.1 Asemasimulaattori

Huoltoaseman ohjelmiston perinteisen järjestelmän sekä laitteiston simulointia varten on kehitetty asemasimulaattori (kuvio 2), jonka avulla on mahdollista kommunikoida uuden SmartNetin asemaohjelman kanssa perinteisen järjestelmän tapaan.



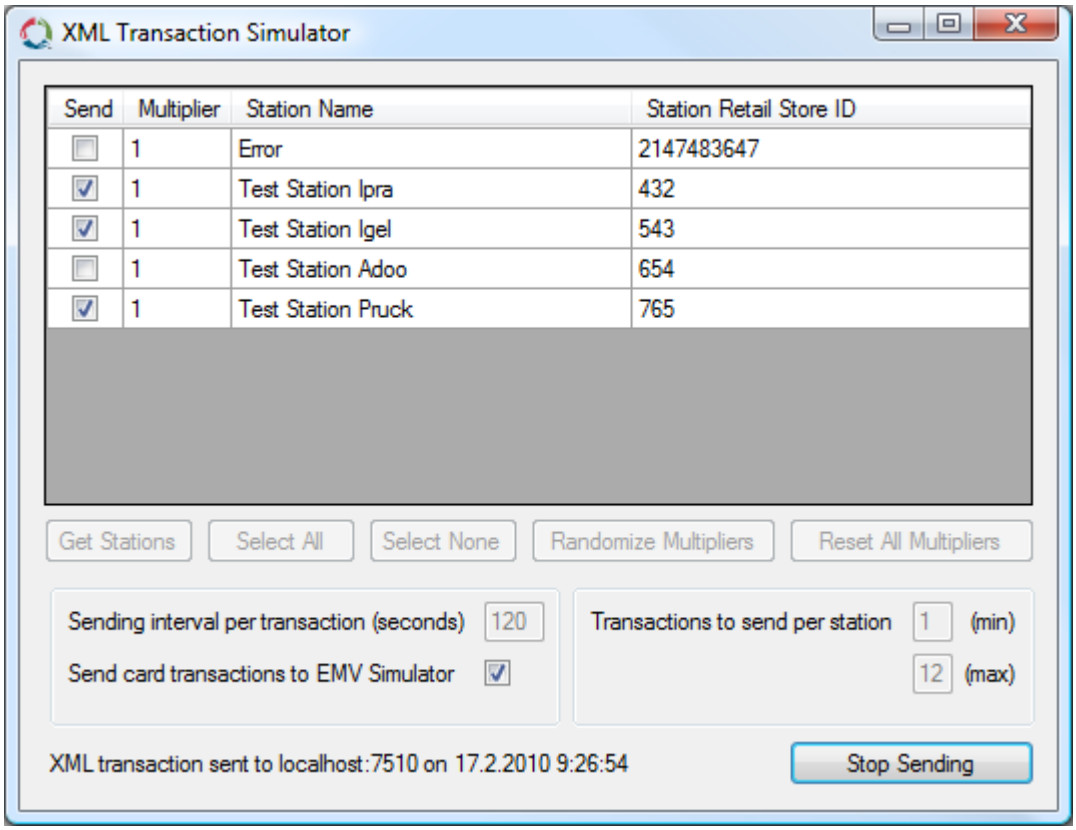
Kuvio 2: Asemasimulaattori

Asemasimulaattorin avulla voidaan lähettää esimerkiksi maksutapahtumia, ilmoituksia ja varaston tietoja. Eräänä mahdollisuutena on myös lähettää käsin kirjoitettua XML-dataa, jonka avulla voidaan testata asemaohjelman kykyä käsitellä virheellistä dataa tai kykyä suojautua erilaisia hyökkäyksiä vastaan.

Asemasimulaattori on mahdollistanut SmartNet-järjestelmän uuden asemaohjelman ja perinteisen järjestelmän välisen kommunikaation kehittämisen ja testaamisen ilman varsinaista huoltoaseman Master PC -konetta ja huoltoasemalla olevaa muuta laitteistoa. Näin on säästetty huomattavasti aikaa, sillä ohjelmointivirheet on mahdollista havaita ennen koko järjestelmän testaamista laitteiston kanssa.

3.2 XML-tapahtumasimulaattori

Eräs tärkeimmistä SmartNetin ominaisuuksista on maksutapahtumien käsittely ja seuranta. Käsittelyalgoritmien ja tietokantahakujen tehokkuuden testaamiseen tarvittiin suuria datamääriä. Tähän ratkaisuna kehitettiin XML-tapahtumasimulaattori (kuvio 3).



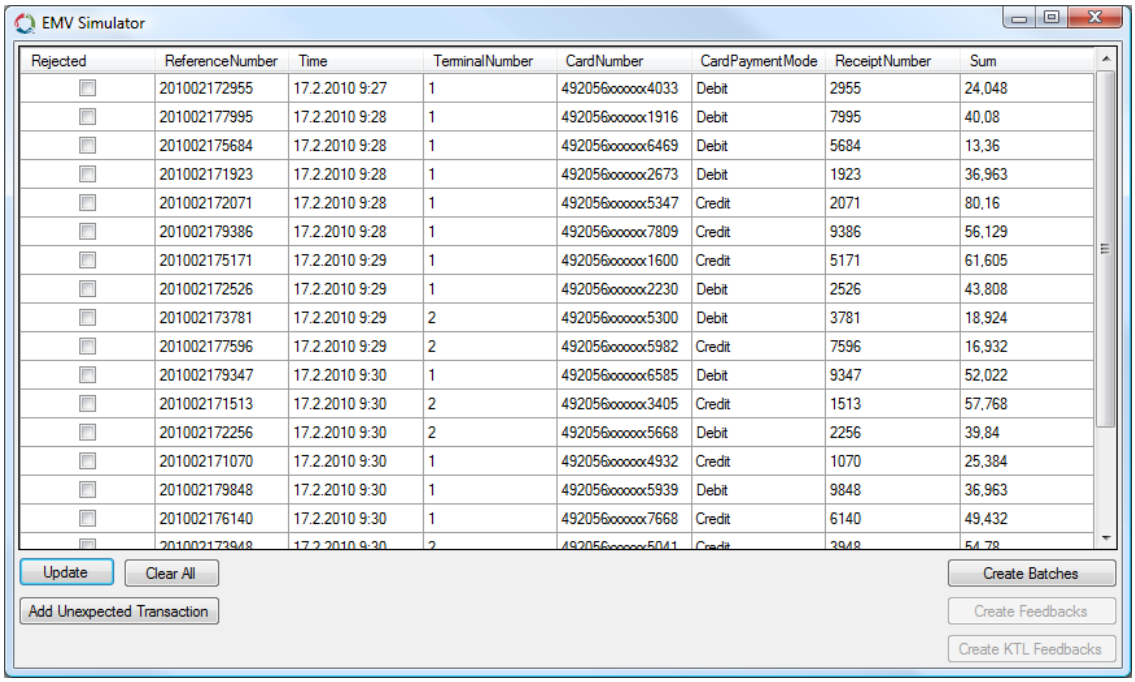
Kuvio 3: XML-tapahtumasimulaattori

Asemasimulaattorilla oli mahdollista lähettää asemaohjelmalle automaattisesti toistuvia maksutapahtumia, mutta useaa asemaohjelmaa testattaessa jokaista asemaohjelmaa kohden oli konfiguroitava oma asemasimulaattori. XML-tapahtumasimulaattorin avulla maksutapahtumien lähettäminen usealle asemaohjelmalle onnistuu helposti. Tämän lisäksi XML-tapahtumasimulaattori generoi maksutapahtuman tiedot satunnaisesti, jolloin saavutetaan suurempi testiaineiston vaihtelu, kuin mitä asemasimulaattoria käyttäen olisi ollut mahdollista.

XML-tapahtumasimulaattoria on käytetty testipalvelimilla erityisesti pitkäkestoisiin kuormitustesteihin, joissa järjestelmää rasitetaan reaalimaailmaa huomattavasti raskaammalla maksuliikenteellä.

3.3 EMV-simulaattori

Maksuliikennetietojen välityksen ja hakemisen testaamista varten kehitettiin EMV-simulaattori (kuvio 4), joka mallintaa EMV-palvelimen toimintaa.



Kuvio 4: EMV-simulaattori

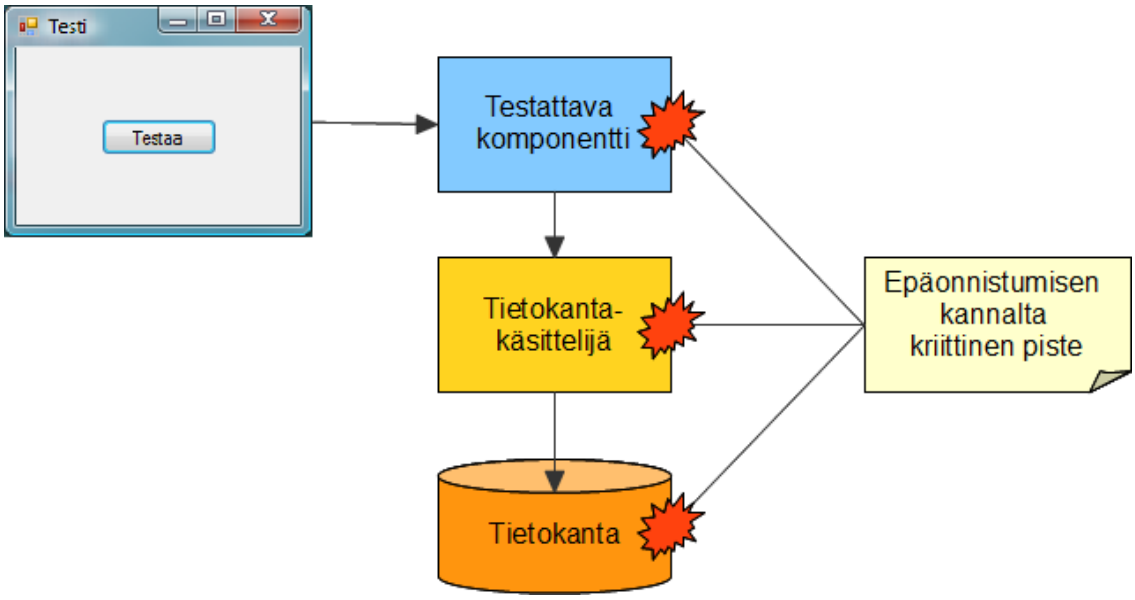
Sekä asema- että XML-tapahtumasimulaattorista on mahdollista lähettää sirukortilla suoritettuja maksutapahtumia asemaohjelman lisäksi myös EMV-simulaattoriin, josta SmartNet-palvelin voi niistä muodostettuja tilityseriä noutaa.

4 SmartNet-järjestelmän komponenttien testaaminen

Tässä luvussa perehdytään aluksi yksikkötestaaminen teoriaan, jonka jälkeen käsitellään SmartNet-järjestelmän testauksen toteutumista.

4.1 Yksikkötestaaminen

Luultavasti yleisin menetelmä, jolla ohjelmoijat testaavat kirjoittamiaan ohjelmiston komponentteja on integraatiotestaus (Osherove 2009). Integraatiotestauksessa useita toisistaan riippuvia ohjelmiston komponentteja testataan yhtä aikaa yhtenä kokonaisuutena. Esimerkiksi kerrosarkkitehtuuria noudattavan sovelluksen jotain komponenttia voidaan testata käyttämällä sitä testausta varten tehdyn käyttöliittymän kautta (kuvio 5).



Kuvio 5: Kerrosarkkitehtuurin testaus käyttöliittymän kautta

Ongelmana kuvion 5 kaltaisessa tilanteessa on se, että mahdollisia epäonnistumisen kannalta kriittisiä pisteitä on monta (Osherove 2009). Integraatiotestin epäonnistuessa testatut komponentit epäonnistuvat kokonaisuutena. Tämä saattaa tehdä mahdollisen vian tai ohjelmointivirheen löytämisestä ja paikallistamisesta erittäin hankalaa. Ratkaisu tähän on komponenttien testaaminen yksikköinä. Tästä käytetään termiä yksikkötestaus.

Tässä työssä yksikkötesteille käytetään seuraavaa määritelmää: Yksikkötesti on täysin automatisoitu, luotettava, luettava ja ylläpidettävä ohjelmakoodi, joka kutsuu testattavaa luokkaa tai metodia ja tarkistaa joitain olettamuksia sen loogisesta toiminnasta. Yksik-

kötesti on helposti kirjoitettavissa ja nopeasti ajettavissa. (Osherove 2009)

Yksikkötestit kirjoitetaan useimmiten yksikkötestaukseen suunniteltua ohjelmistokehystä käyttäen. Tunnetuimmat ohjelmistokehykset kuuluvat xUnit-perheeseen, kuten esimerkiksi Java-ohjelmointikielelle tehty JUnit ja .NET-ohjelmontikielille tehty NUnit, pohjautuen Kent Beckin alkuperäiseen Smalltalk-ohjelmointikielelle tehtyyn toteutukseen SUnit (Fowler). Luvussa 4.3 käsitellään tarkemmin SmartNetin testaamisessa käytettyä NUnit-yksikkötestauskehystä.

4.2 Yksikkötestaamisen tekniikat

Yksikkötestaamiseen liittyy erilaisia tekniikoita, joilla helpotetaan tai mahdollistetaan sovellusten komponenttien testaamista yksikköinä. Tässä työssä keskitytään SmartNet-järjestelmän testaamisen kannalta oleellisiin yksikkötestaamisen tekniikoihin, jotka käydään läpi tämän luvun aliluvuissa. Aliluvut pohjautuvat sekä Gerard Meszaroksen (2007) että Roy Osheroven (2009) kuvauksiin esitellyistä tekniikoista.

Yksikkötestaamisen tekniikoiden kuvauksissa painotetaan olio-ohjelmoinnin sovellutuksia ja kuvauksissa käytetään termejä testattava luokka (CUT) ja vaadittu komponentti (DOC). Vaadittu komponentti on luokka tai komponentti, johon testattavalla luokalla on riippuvuus (dependency).

4.2.1 Väärennökset

Väärennöksillä (fake) voidaan korvata komponentteja, joita ei voida tai haluta käyttää testattaessa. Yleensä väärennöksiä käytetään korvaamaan testattavan luokan riippuvuuksia. Tässä työssä väärennökset jaetaan kahteen eri ryhmään: tynkiin (stub) ja mallinteisiin (mock). Väärennöksistä voidaan käyttää myös nimitystä testisijainen (test double).

Tynkä ja mallinne eroavat toisistaan siten, että tynkää käytetään erilaisten tilanteiden simuloimiseen ja testin kannalta haluttujen arvojen syöttämiseen testattavaan luokkaan. Mallinteen avulla testataan, että testattava luokka käyttää mallinteen esittämää komponenttia oikein. Testi voi epäonnistua mallinteesta tai sen käytöstä riippuvista syistä, mutta ei koskaan tyngän toimesta.

Käytännössä ero tyngän ja mallinteen välillä on häilyvä. Periaatteessa mallinne on

tynkä, joka sisältää logiikkaa. Tästä syystä termejä käytetään joskus ristiin tai yhtä termiä käytetään kattamaan molemmat. Esimerkiksi ainoa vakiintunut väärennöksiin liittyvä suomennos on tynkä, jota käytetään yleensä tarkoittamaan mitä vain väärennöstä. Tarkan erottelun tekee hankalaksi myös se, että alan kirjallisuudessa asioita on jaoteltu eri tavoilla, usein päällekkäisin tai risteävin termein toisiinsa nähden.

Väärennöksiä voidaan ohjelmoida käsin, mutta käsin ohjelmointi on hidasta ja helposti eri tilanteissa uudelleenkäytettäviä väärennöksiä on vaikea tehdä (Osherove 2009). Ratkaisuna tähän tarjolla on väärennösten luontiin tarkoitettuja ohjelmistokehyksiä, joista käytetään nimitystä eristysohjelmistokehys (isolation framework). Tälläisiä ovat esimerkiksi Java-ohjelmointikielelle tarkoitettu jMock sekä .NET-ohjelmointikielille tarkoitetut Moq, Rhino Mocks ja kaupallinen Typemock Isolator. Luvussa 4.3 käsitellään tarkemmin SmartNetin testaamisessa käytettyä Rhino Mocks -eristysohjelmistokehystä.

4.2.2 Saumat

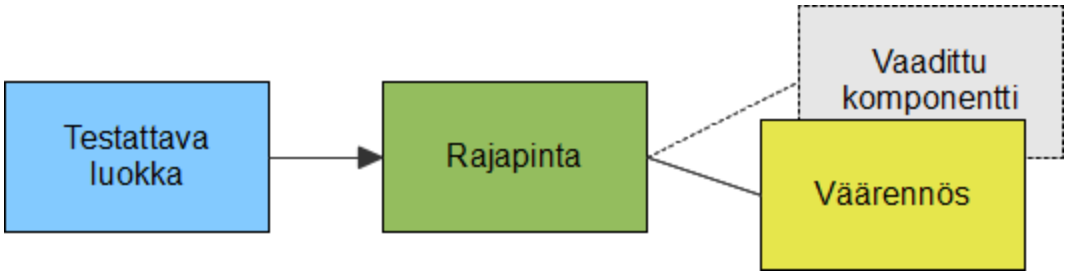
Saumat (seam) ovat kohtia ohjelmakoodissa, joissa ohjelman käyttäytymistä voidaan muuttaa muuttamatta itse ohjelmakoodia (Feathers 2005). Olio-ohjelmoinnissa saumoja ovat esimerkiksi rakentajat ja metodit, joissa parametrina välitetään käytettävä rajapinta, sekä virtuaaliset metodit. Testauksen kannalta mahdollisina saumoina voidaan nähdä myös tehtaat (factory) ja paikalliset tehdasmetodit (local factory method).

Saumojen avulla voidaan esimerkiksi syöttää väärennöksiä testattavaan luokkaan. Tapoja tähän ovat esimerkiksi luvussa 4.2.3 käsiteltävä riippuvuusinjektio ja luvun 4.2.4 Irrota ja peitä -tekniikan käytössä mainittava tehtaan tai paikallisen tehdasmetodin hyödyntäminen.

4.2.3 Riippuvuusinjektio

Riippuvuusinjektio (dependency injection) on tekniikka, jolla testattavan luokan ja sen riippuvuuksien väliset kytkökset (coupling) voidaan hajottaa. Riippuvuusinjektion käytön tuoma muunneltavuus mahdollistaa testattavan luokan kunnollisen yksikkötestaamisen, sillä nyt luokka on mahdollista eristää sen vaatimista komponenteista eli riippuvuuksista.

Käytännössä testattavan luokan eristäminen testaamista varten toteutetaan korvaamalla luokan riippuvuudet erilaisilla väärennöksillä. Jotta väärennös olisi yhteensopiva sitä käyttävän luokan kanssa, toteuttavat se ja vaadittu komponentti yleensä saman rajapinnan, jolloin testattava luokka käyttää väärennöstä ikään kuin se olisi alkuperäinen vaadittu komponentti (kuvio 6).



Kuvio 6: Testattavan luokan vaadittu komponentti on korvattu väärennöksellä

Väärennösten syöttämiseen testattavaan luokkaan riippuvuusinjektiota käyttäen on erilaisia tapoja (koodiesimerkki 1): rakentajainjektiossa (constructor injection) väärennös syötetään testattavaa luokkaa luotaessa, parametri-injektiossa (parameter injection) luokkaa kutsuttaessa ja asetusmetodi- tai ominaisuusinjektiossa (setter injection, property injection) sopivassa vaiheessa luokan luonnin jälkeen ennen kutsumista.

Koodiesimerkki 1: Väärennösten injektointitapoja

```
// Esimerkkiluokka InjectionSample, jolla on riippuvuus IDependency-
// rajapinnan toteuttavaan luokkaan.
public class InjectionSample
{
      private IDependency dependency;

      // Rakentajainjektio
      public InjectionSample(IDependency dependency)
      {
            this.dependency = dependency;
      }
```

```
      // Ominaisuusinjektio
      public IDependency Dependency
      {
            get { return dependency; }
            set { dependency = value; }
      }

      public void Use()
      {
            dependency.Use();
      }

      // Parametri-injektio
      public void Use(IDependency dependency)
      {
            dependency.Use();
      }
}
```

Eri injektointitavoilla on omat vahvuutensa ja heikkoutensa. Rakentajainjektion käyttö viestittää luokan käyttäjille, että rakentajassa välitettävät riippuvuudet ovat pakollisia luokan toiminnan kannalta. Toisaalta, jos luokalla on paljon riippuvuuksia, rakentajien ja rakentajaparametrien määrää kasvaa helposti vaikeasti hallittavaksi.

Rakentajainjektion tuoman kömpelyyden vuoksi yksinkertaisempi ratkaisu onkin käyttää asetusmetodi- tai ominaisuusinjektiota. Tämä kuitenkin samalla viestittää, että tällä tavalla välitettävät riippuvuudet ovat valinnaisia.

Parametri-injektion käyttö viestittää rakentajainjektion tapaan, että välitettävät riippuvuudet ovat nyt pakollisia metodin toiminnan kannalta. Parametri-injektion käyttö on hyödyllistä silloin, kun riippuvuusinjektiota käytetään enemmän muunneltavuuden kuin testattavuuden saavuttamiseksi, ja luokan tarkoitus on tehdä jonkin tietty operaatio useammalle saman rajapinnan toteuttavalle luokalle.

4.2.4 Irrota ja peitä -tekniikka

Irrota ja peitä (extract and override) -tekniikalla voidaan korvata riippuvuusinjektio lisäämällä testattavaan luokkaan saumaksi paikallinen virtuaalinen tehdasmetodi, joka voidaan peittää testattavaa luokkaa laajentavassa luokassa (koodiesimerkki 2). Laajentava luokka tehdään testattavaksi esimerkiksi riippuvuusinjektiota käyttäen. Näin saadaan testattavan luokan riippuvuudet pidettyä suojattuna, ja ne paljastetaan ainoas-

taan testaamista varten luodussa laajennuksessa.

Koodiesimerkki 2: Irrota ja peitä -tekniikan käyttö

```
// Esimerkkiluokka Machine, joka käyttää BevelGear-luokkaa.
// BevelGear-luokan toteutuksesta ei tarvitse välittää.
public class Machine
{
      private BevelGear bevelGear;

      public Machine()
      {
            bevelGear = new BevelGear();
      }

      public void Work()
      {
            if (bevelGear.Spinning)
            {
                  ...
            }
      }
}

// Muokataan esimerkkiluokkaa lisäämällä paikallinen virtuaalinen
// tehdasmetodi.
// BevelGear toteuttaa nyt IGear-rajapinnan.
public class Machine
{
      private IGear gear;

      public Machine()
      {
            gear = GetGear();
      }

      public void Work()
      {
            if (gear.Spinning)
            {
                  ...
            }
      }

      protected virtual IGear GetGear()
      {
            return new BevelGear();
      }
}
```

```
// Laajennetaan Machine-luokasta testattava luokka, ja peitetään
// GetGear-metodi.
// Peittävä metodi palauttaa erikseen ominaisuusinjektiolla
// syötettävän toteutuksen, esimerkiksi testitilanteessa sopivan
// väärennöksen.
public class TestableMachine : Machine
{
      private IGear gear;

      public TestableMachine() : base()
      {
      }

      public IGear Gear
      {
            get { return gear; }
            set { gear = value; }
      }

      protected override IGear GetGear()
      {
            return gear;
      }
}
```

Yksinkertainen sovellutus Irrota ja peitä -tekniikasta on paikallisen virtuaalisen tehdas-metodin sijaan tehdä vaadittua komponenttia käyttävästä metodista virtuaalinen ja testattavaa luokkaa laajentavan luokan peittävässä metodissa palauttaa suoraan testin kannalta haluttu lopputulos (koodiesimerkki 3).

Koodiesimerkki 3: Irrota ja peitä -tekniikan yksinkertainen sovellutus

```
// Koodiesimerkin 2 Machine-luokkaa on nyt muokattu niin, että
// BevelGear-luokan käyttö on siirretty virtuaaliseen metodiin.
public class Machine
{
      private BevelGear bevelGear;

      public Machine()
      {
            bevelGear = new BevelGear();
      }

      public void Work()
      {
            if (IsGearSpinning())
            {
                  ...
```

```
            }
      }

      public virtual bool IsGearSpinning()
      {
            return bevelGear.Spinning;
      }
}

// Nyt Machine-luokasta voidaan laajentaa luokka, joka peittää
// IsGearSpinning-metodin.
// Peittävä metodi palauttaa erikseen asetetun halutun paluuarvon.
public class TestableMachine : Machine
{
      private bool isGearSpinningReturnValue;

      public TestableMachine() : base()
      {
      }

      public bool IsGearSpinningReturnValue
      {
            get { return isGearSpinningReturnValue; }
            set { isGearSpinningReturnValue = value; }
      }

      public override bool IsGearSpinning()
      {
            return isGearSpinningReturnValue;
      }
}
```

Irrota ja peitä -tekniikka on hyödyllinen silloin, kun vaadittu komponentti ei toteuta valmiiksi jotain rajapintaa tai testattava luokka ei tarjoa sopivaa saumaa riippuvuusinjektiota varten eikä ylimääräistä rajapintaa tai riippuvuusinjektion mahdollistavaa saumaa haluta lisätä.

4.3 SmartNet-järjestelmän komponenttien testauksen toteutuminen

SmartNetin komponenttien testaamiseen on käytetty sekä yksikkötestejä että automatisoituja integraatiotestejä. Luonteensa vuoksi integraatiotestit vaativat useiden komponenttien ajossa oloa, eikä kaikkia välttämättä pystytä korvaamaan testattavilla versioilla. Tämä on ratkaistu lisäämällä suurimpien komponenttien asetustiedostoon mahdollisuus

käyttää tätä komponenttia testitilassa, jolloin se ja sen käyttämät muut komponentit saadaan käyttäytymään testaamisen kannalta sopivalla tavalla.

SmartNetin yksikkötestaustyökaluksi on valittu NUnit[1], joka on avoimen lähdekoodin yksikkötestauskehys Microsoftin .NET-ohjelmistokehyksen ohjelmointikielille. NUnit valittiin siksi, että se on kehittäjien suosima ja xUnit-perheeseen kuuluvana sen käyttö on helppo omaksua muita vastaavia työkaluja käyttäneille.

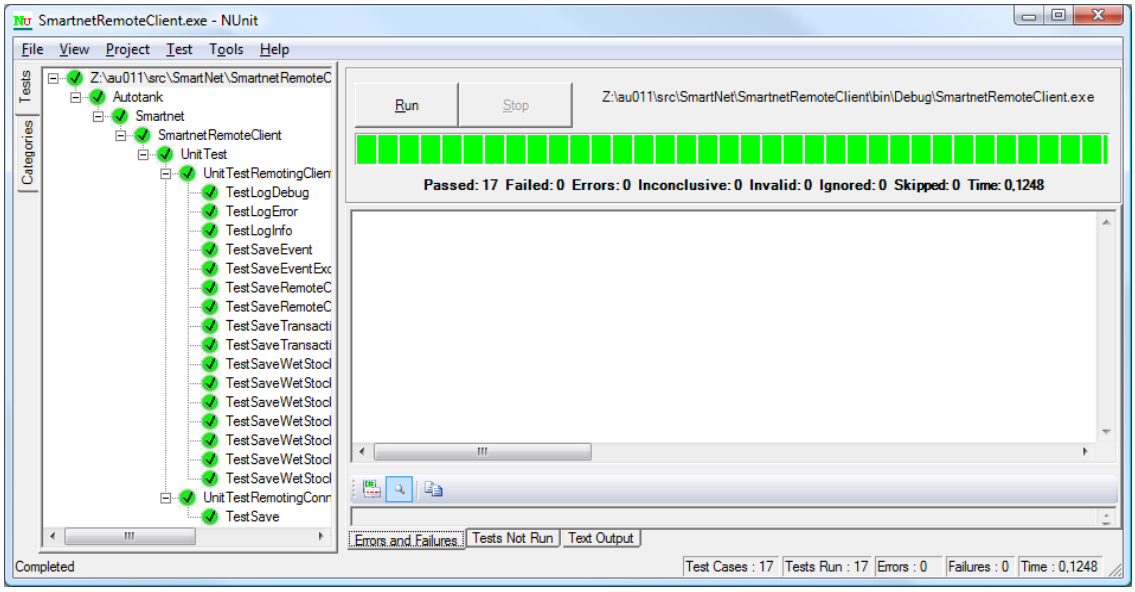
SmartNetin NUnit-testit on kirjoitettu C#-ohjelmointikielellä. Yksi NUnit-testi koostuu useasta eri osasta (koodiesimerkki 4): testin kehyksestä (test fixture), mahdollisista pystytys- ja purkuosioista (setup, teardown) ja itse testeistä. Testien suorittamiseen NUnit tarjoaa selkeän käyttöliittymän (kuvio 7).

Koodiesimerkki 4: NUnit-testin eri osat.

```
using NUnit.Framework;

[TestFixture]
public class ExampleNunitTest // Testin kehyksenä toimiva luokka.
{
      public ExampleNunitTest()
      {
            // Luokan rakentaja, ajetaan kerran testausta aloitettaessa.
      }

      [SetUp]
      public void SetUp()
      {
            // Pystytysosio, ajetaan ennen jokaista yksittäistä testiä.
      }

      [TearDown]
      public void TearDown()
      {
            // Purkuosio, ajetaan jokaisen yksittäisen testin jälkeen.
      }

      [Test]
      public void TestA()
      {
            // Yksittäinen testi.
      }
      [Test]
      public void TestB()
```

1 http://www.nunit.org/

```
        {
                // Yksittäinen testi.
        }

        [Test]
        public void TestC()
        {
                // Yksittäinen testi.
        }
}
```



Kuvio 7: NUnit-yksikkötestauskehyksen käyttöliittymä

Testaamisessa tarvittavien väärennösten luontia helpottamaan on SmartNetin testauksessa otettu käyttöön Rhino Mocks[2] -eristysohjelmistokehys. NUnitin tapaan se on suunniteltu Microsoftin .NET-ohjelmistokehyksen ohjelmointikielille ja sitä käytetään SmartNetin tapauksessa C#-ohjelmointikielellä. Eristysohjelmistokehystä valittaessa Rhino Mocksiin päädyttiin siksi, että se on NUnit-kehyksen tapaan suosittu ja sen käyttöön oli esimerkkimateriaalia saatavilla.

Rhino Mocks mahdollistaa tynkien ja mallinteiden dynaamisen luomisen suoritusaikaisesti. SmartNetin testeissä on käytetty Rhino Mocksin Nauhoita ja toista -mallia (record-and-replay model), jossa nauhoitusvaiheessa käsitellään mallinteita sillä tavalla, miten testattavan luokan oletetaan testitapauksessa niitä käyttävän. Tämän jälkeen toistovaiheessa kutsutaan testattavaa luokkaa ja tarkistetaan olettamuksien toteutuminen.

2 http://www.ayende.com/projects/rhino-mocks.aspx

Koodiesimerkki 5 havainnollistaa Rhino Mocksin käyttöä SmartNetin yksikkötesteissä. Esimerkissä luodaan testin pystytysvaiheessa Rhino Mocksin MockRepository-olio, jonka avulla luodaan SmartNetin ISmartNetLogger- ja IRemotingConnection-rajapinnat toteuttavat väärennökset. ISmartNetLogger-rajapinnasta tarvitaan mallinne, sillä halutaan varmistaa, että testattava luokka kirjoittaa lokitiedostoon oikeat asiat. IRemotingConnection-rajapinnan toteutukseksi riittää tynkä, sillä tämän rajapinnan käytöllä ei ole väliä testin onnistumisen kannalta. Tämän jälkeen tehdään nauhoitus- ja toistovaiheet käyttäen C#-ohjelmointikielen using-lausetta ja MockRepository-olion Record- ja Playback-metodeita. Testattavasta SmartNetin RemotingClient-luokasta on testaamista varten laajennettu TestableRemotingClient-luokka Irrota ja peitä -tekniikkaa hyödyntäen.

Koodiesimerkki 5: Rhino Mocksin Nauhoita ja toista -mallin käyttö

```
using System;
using Autotank.Smartnet.ObjectModel;
using Autotank.Smartnet.SmartNetLogger;
using NUnit.Framework;
using Rhino.Mocks;

namespace Autotank.SmartNet.RemotingClient.UnitTest
{
      [TestFixture]
      public class UnitTestRemotingClient
      {
            private MockRepository mocks;
            private ISmartNetLogger logger;
            private IRemotingConnection connection;
            private RemotingClientSettings remotingClientSettings;

            [SetUp]
            public void SetUp()
            {
                  // Luodaan MockRepository-olio ja luodaan sen avulla
                  // väärennökset RemotingClient-luokan riippuvuuksista.
                  // Väärennökset välitetään RemotingClientSettings-
                  // parametiolion avulla.
                  mocks = new MockRepository();
                  logger = mocks.StrictMock<ISmartNetLogger>();
                  connection = mocks.Stub<IRemotingConnection>();
```

```
            // Muita parametriolion tarvitsemia tietoja, joilla ei
            // ole merkitystä testin kannalta.
            int retailStoreId = 0;
            int xmlListenerPort = 0;
            string xmlSendAddress = String.Empty;
            int xmlSendPort = 0;
            bool stockSurveillanceOn = false;
            int pollingInterval = 0;

            // Luodaan RemotingClientSettings-parametriolio,
            // jonka avulla välitetään halutut parametrit
            // RemotingClient-oliota luotaessa.
            settings = new RemotingClientSettings(retailStoreId,
                  xmlListenerPort, xmlSendAddress, xmlSendPort,
                  stockSurveillanceOn, pollingInterval, logger,
                  connection);
      }

      // Esimerkkitesti, jossa testataan RemotingClient-luokan
      // SaveEvent-metodin toimintaa poikkeustilanteessa.
      // Testissä halutaan varmistua siitä, että RemotingClient
      // kirjoittaa virhetilanteessa oikeat lokimerkinnät ja
      // heittää ApplicationException-tyyppisen poikkeuksen.
      [Test]
      [ExpectedException(typeof(ApplicationException),
            ExpectedMessage = "Could not send event to server.")]
      public void TestSaveEventException()
      {
            // Pyydetään TestObjectFactory-tehtaalta Event-olio
            // testiä varten.
            Event testEvent = TestObjectFactory.GetTestEvent();

            // Aloitetaan nauhoitus. Nauhoitus päättyy
            // automaattisesti using-lauseesta poistuttaessa.
            using (mocks.Record())
            {
                  // Ilmoitusta tallennettaessa RemotingClient
                  // kirjoittaa siitä lokiin.
                  logger.LogInfo("Saving event...");

                  // IRemotingConnection-tyngän Save-metodin
                  // kutsumisen jälkeen pyydetään testikehystä
                  // heittämään poikkeus.
                  remotingConnection.Save(testEvent);
                  LastCall.Throw(new Exception());

                  // Poikkeuksen tapahtuessa RemotingClient
                  // kirjoittaa siitä virhelokiin.
                  logger.LogError("Failed to save event.");
            }
```

```
                // Aloitetaan toisto. Using-lauseesta poistuttaessa
                // tehdään automaattinen vertailu nauhoitettuun
                // käsittelyyn.
                using (mocks.Playback())
                {
                      // Luodaan RemotingClient-olio. RemotingClient-
                      // luokasta on laajennettu testaamista varten
                      // TestableRemotingClient-luokka.
                      TestableRemotingClient client =
                            new TestableRemotingClient(settings);

                      // SaveEvent-kutsun pitäisi käsitellä
                      // ISmartNetLogger-mallinnetta samalla tavalla
                      // kuin nauhoitusvaiheessa.
                      client.SaveEvent(testEvent);
                }
          }

          ...
     }
}
```

Voisi olettaa, että koodiesimerkin 5 kaltaisten testien kirjoittaminen lisäisi projektin työmäärää. Toteutusvaiheen työmäärää todellakin kasvaa testien kirjoittamisen vuoksi, mutta samalla ylläpito- ja mahdollisten lisäkehitysvaiheiden työmäärät alenevat, sillä yksikkötestit suorittamalla on nopeaa ja helppoa nähdä, rikkoivatko tehdyt muutokset järjestelmän.

Uuden SmartNetin modulaarinen rakenne on helpottanut testaamista. Suurin osa komponenttien ja prosessien välisestä kommunikaatiosta tapahtuu erilaisten rajapintojen kautta, joten testauksessa oikeat toteutukset on helppo korvata sopivilla väärennöksillä.

Myös Irrota ja peitä -tekniikka on havaittu hyödylliseksi, sillä erilaisia rajapintoja on jo valmiiksi niin monta, ettei uutta rajapintaa haluta lisätä pelkästään testaamisen vuoksi. Irrota ja peitä -tekniikan avulla on ollut myös mahdollista peittää asioita, jotka eivät olisi mahdollistaneet yksikkötestien olevan toistettavia ja luotettavia, esimerkiksi taustaprosesseja käynnistäviä metodeja tai kutsuja tilakone- tai laskurimaisesti toimiviin staattisiin luokkiin.

Yksi haasteista on ollut tietokantakäsittelijän testaus. Luokkien olio–relaatio-mallinnusten, tietokantaan tallentamisen ja tietokantahakujen testaaminen suoritetaan käyttämällä automatisoituja integraatiotestejä oikean, testaamista varten tarkoitetun, tieto-

kannan kanssa. Testitietokanta tyhjennetään testien välissä ja täytetään uudestaan testi-datalla, joten testit ovat toistettavissa ja toisistaan riippumattomia. Tästä tekniikasta käytetään nimeä tietokantahiekkalaatikko (database sandbox).

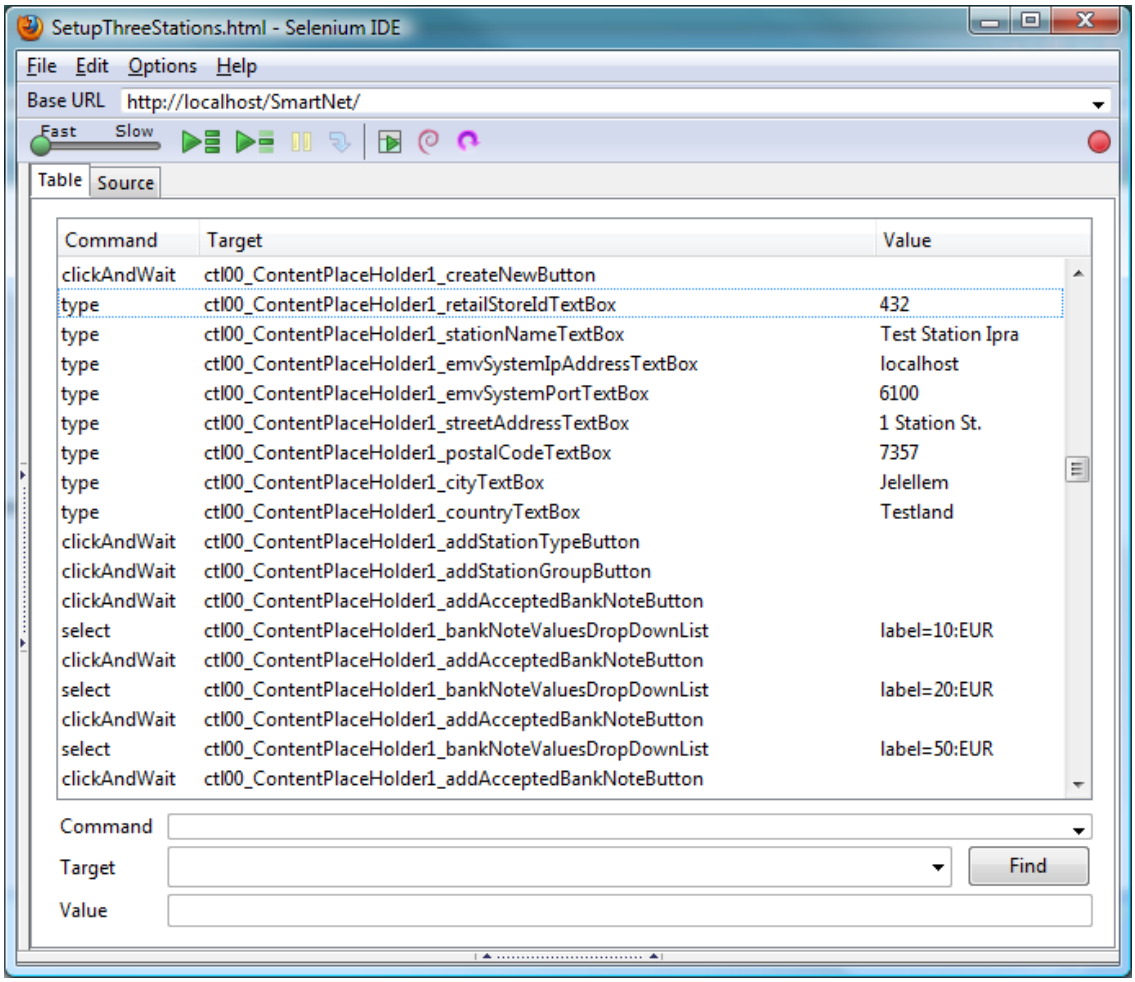
Tulevaisuudessa on tarkoitus nostaa yksikkötestien kattavuutta ja määrää, sekä korvata automatisoituja integraatiotestejä yksikkötesteillä. Lopullinen tavoite on, että kaikki SmartNetin komponentit paitsi tietokantakäsittelijä ovat testattavissa yksikkötesteillä. Tietokantakäsittelijä on ainoa komponentti, jonka yksikkötestaaminen on hyödytöntä, sillä koko tietokantatason mallintaminen yksikkötestejä varten olisi liian työlästä.

5 SmartNet-järjestelmän WWW-käyttöliittymän testaaminen

SmartNetin WWW-käyttöliittymä päädyttiin testaamaan suurilta osin käsin perinteisenä testaustyönä. Testaamista helpottamaan kokeiltiin Selenium-työkaluja[3], jotka on suunniteltu erityisesti verkkosovellusten testaamista varten. Merkittävimmät Selenium-perheen työkaluista ovat Selenium IDE ja Selenium RC.

5.1 Selenium IDE -työkalu

Selenium IDE on työkalu, jolla voidaan kehittää ja suorittaa Selenium-testitapauksia. Toteutukseltaan se on Mozilla Firefox -selaimen liitännäinen. Selenium-testitapaus on muodoltaan HTML-koodia, jonka Selenium IDE näyttää taulukkomuodossa (kuvio 8).



Kuvio 8: Selenium IDE -työkalu

3 http://seleniumhq.org/

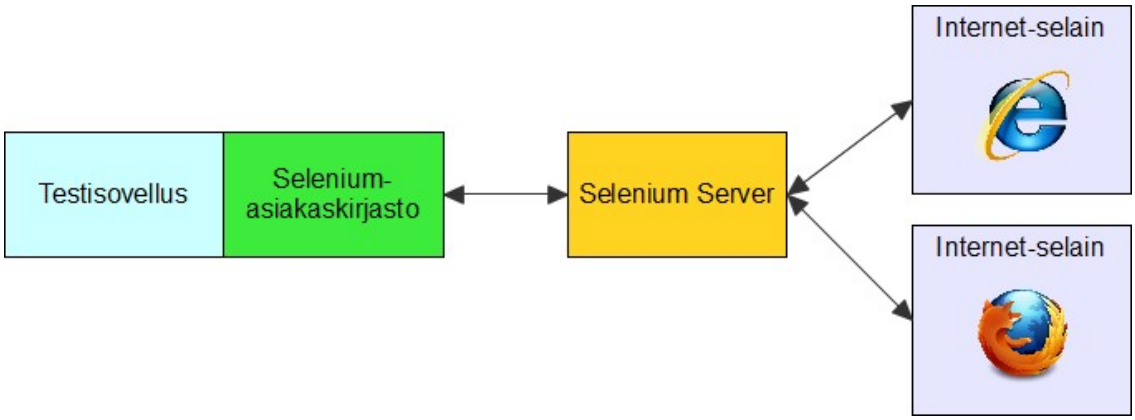
Testitapausten nopeaan kehittämiseen Selenium IDE tarjoaa kätevän nauhoitusominaisuuden, jolla testitapaukseen lisätään suoritettavia komentoja käyttäjän selaimessa suorittamien toimintojen mukaan. Esimerkiksi linkin tai napin painaminen WWW-sivulla generoi vastaavan toiminnon suorittavan click-komennon ja tekstilaatikkoon kirjoittaminen type-komennon.

5.2 Selenium RC -työkalu

Selenium RC on työkalu, joka mahdollistaa monimutkaisempien testien tekemisen kuin mitä Selenium IDE -työkalun avulla on mahdollista tehdä. Selenium IDE ei tarjoa tukea esimerkiksi ehto- ja toistolauseille.

Selenium RC koostuu kahdesta osasta: Selenium Server -ohjelmasta ja asiakaskirjastosta, josta on versioita eri ohjelmointikielille. Selenium RC -työkalun Selenium-testitapauksia voidaan kirjoittaa esimerkiksi Java-, C#-, Python- tai Ruby-ohjelmointikielillä.

Selenium RC toimii ikään kuin kaukosäätimenä Internet-selaimelle. Selenium Server suorittaa eri selainprosesseja ja vastaanottaa asiakaskirjastoa käyttävältä testisovellukselta komentoja, jotka se suorittaa selaimessa, ja palauttaa tuloksen testisovellukselle (kuvio 9).



Kuvio 9: Selenium RC -työkalun toimintaperiaate

5.3 Selenium-työkalujen käyttö WWW-käyttöliittymän testaamisessa

SmartNet-järjestelmän WWW-käyttöliittymän testaamisessa Selenium IDE -työkalua on käytetty lähinnä testaamisen apuna automatisoituun käyttöliittymän kautta tapahtuvaan konfiguraatioon. Tämä voidaan nähdä eräänlaisena WWW-käyttöliittymän automatisoituna regressiotestaamisena.

Selenium RC -työkalu otettiin koekäyttöön, sillä haluttiin tutkia, olisiko sen avulla mahdollista tehdä täysin automatisoituja testejä SmartNetin WWW-käyttöliittymälle. Vaikka syntyneet testit eivät olleetkaan yksikkötestejä termin puhtaassa merkityksessä, käytettiin testien moottorina NUnitia, sillä se tarjosi valmiin alustan, jonka avulla testejä oli helppo ajaa.

Vaikka Selenium RC olisi mahdollistanut testien automatisoinnin viemisen huomattavan pitkälle, nähtiin täysimittaisen testikirjaston tekeminen liikaa resursseja vieväksi. Lopulta päädyttiin suosimaan Selenium IDE -työkalua sen helppokäyttöisyyden vuoksi.

SmartNetin tapauksessa hyvien ja kattavien automatisoitujen testien tekemiseen Selenium-työkaluilla kuluisi sen verran aikaa, että WWW-käyttöliittymän perinteistä testaamista pidettiin tehokkaampana resurssien käyttönä. Vaikka Selenium-työkalujen käyttö mahdollistaisi täysin automatisoitujen testien tekemisen, ei niiden avulla voida testata esimerkiksi käytettävyyttä tai käyttöliitymän ulkoasua.

Selenium IDE -työkalua käytetään SmartNetin tapauksessa nykyään varsinaisen testaamisen sijaan automaattiseen testiympäristöjen pystyttämiseen ja konfigurointiin. Näin on säästetty huomattavasti aikaa, sillä WWW-käyttöliittymän kautta tapahtuvat toistuvat toimenpiteet on saatu automatisoitua.

6 Yhteenveto

Tässä työssä käsiteltiin uudistetun SmartNet-järjestelmän testaamista. Työssä esiteltiin SmartNet-järjestelmän testaamista varten kehitettyjä testisimulaattoreita sekä testaamisen helpottamiseksi käyttöönotettuja työkaluja. Lisäksi työssä esiteltiin SmartNet-järjestelmän testaamisen kannalta oleellisia yksikkötestaamisen tekniikoita ja menetelmiä.

SmartNet-järjestelmän testauksessa käytettiin NUnit- ja Rhino Mocks -ohjelmistokehyksiä. Vaihtoehtona NUnit-kehykselle olisivat olleet esimerksiksi MbUnit tai xUnit.net, mutta NUnit päädyttiin valitsemaan sen suosion ja vakiintuneisuuden vuoksi. Rhino Mocks -kehykselle järkevin vaihtoehto olisi ollut Moq, mutta Moq ei tarjonnut tukea .NET 2.0 -ohjelmistokomponenttikirjastolle. Lähinnä huoltoasemien Master PC -koneista ja muista vaatimuksista johtuvista syistä .NET-kirjaston 3.5-versiota ei voitu ottaa käyttöön.

SmartNet-järjestelmän uudistamisprojektissa työskennellessäni ja tätä työtä tehdessäni opin käyttämään C#-ohjelmointikieltä ja toteuttamaan yksikkötestaamisen tekniikoita ja malleja sen avulla. Opin myös käyttämään useita uusia ohjelmistokehyksiä ja -työkaluja. Tämän työn avulla pystyin laajentamaan tietämystäni yksikkötestaamisesta ja testaamisesta yleensä huomattavasti.

Työn tausta-aineoston avulla tutustuin entistä perusteellisemmin testivetoisen kehityksen filosofiaan, ja testivetoinen kehitys (TDD) olisi voinut olla mielenkiintoinen ja tehokas kehitysmenetelmä projektille. Puhdasoppisessa testivetoisessa kehityksessä tosin järjestelmää suunnittellaan juuri testien avulla, mikä ei tähän projektiin olisi välttämättä täysin sopinut. Eräänlainen ”testit ensin” -filosofia olisi kuitenkin voinut paikoin säästää aikaa esimerkiksi luokkien koheesion ja luokkien välisten kytköksien huomioinnissa.

Ottaen huomioon projektin nopean kehitystahdin ja jatkuvat muutokset SmartNet-järjestelmän testaustilanne on hyvä. Kun projektissa päästään sellaiseen vaiheeseen, että julkaisu- tai ylläpitoversio voidaan saattaa valmiiksi, on mahdollista nostaa testien määrää ja testauksen laatua entisestään.

Lähteet

Feathers, Michael C. 2005. Working Effectively with Legacy Code. New Jersey: Prentice Hall.

Fowler, Martin. Xunit. [online]. [viitattu 23.12.2009]. Saatavilla: http://www.martinfowler.com/bliki/Xunit.html

Meszaros, Gerard 2007. xUnit Test Patterns: Refactoring Test Code. Boston: Addison-Wesley.

Osherove, Roy 2009. The Art Of Unit Testing: with Examples in .NET. Greenwich, Connecticut: Manning Publications Co.