Masi Pulkkinen

Automatisoitu yksikkötestaus NUnit-testausympäristössä

Tietotekniikan
kandidaatintutkielma
25. helmikuuta 2009

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Masi Pulkkinen
Yhteystiedot: masi.pulkkinen@jyu.fi
Työn nimi: Automatisoitu yksikkötestaus NUnit-testausympäristössä
Title in English: Automated Unit Testing in NUnit Testing Environment
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 37
Tiivistelmä: Tutkielmassa perehdytään ohjelmisto- ja testausprosessiin, testaustasoista erityisesti yksikkötestaukseen sekä testauksen automatisointiin. Soveltavissa esimerkeissä tarkastellaan yksikkötestauksen automatisointia NUnit-testausympäristössä.
English abstract: The thesis examines software and test processes, unit testing and how the tests can be automated. NUnit testing environment is used in the examples to demonstrate the process of writing automated unit tests.
Avainsanat: yksikkötestaus, NUnit, C#, ohjelmistotestaus, yksikkötestauksen automatisointi, testausprosessi, testausympäristö, testitapaus.
Keywords: unit test, NUnit, C#, software testing, automated testing, automated unit testing, test process, test environment, test case.

Sisältö

1 Johdanto

Ohjelmistojen testaus on olennainen osa ohjelmistokehitystä. Uudelleenkäytettävien moduulien kehittämisen ja kerrosarkkitehtuurien myötä testauksessa on korostunut virheiden etsimisen ohella ohjelmiston laadunvarmistus ja lähdekoodiin tehtävien muutoksien hallinta.

Ohjelmistojen testausta voidaan käsitellä eri tasoilla sen mukaan, mitä osaa tai kokonaisuutta siitä testataan. Yksikkötestaus kohdistuu ohjelmiston pienimpiin osiin ja sen avulla ohjelmiston yksittäisten toiminnallisuuksien oikea ja virheetön toiminta voidaan varmistaa. Testausta joudutaan usein suorittamaan uudelleen kehityksen yhteydessä, jolloin puhutaan regressiotestauksesta. Testitapausten automatisointi nopeuttaa testien uudelleensuorittamista.

NUnit on .NET-ohjelmointiin kehitetty testausympäristö, jolla voi suorittaa ohjelmoituja testitapauksia. Sillä on siten mahdollista testata kielillä C#, J#, VisualBasic.NET ja C++ toteutettuja ohjelmia. Tutkielmassa keskitytään C#-kielellä kirjoitettujen ohjelmaosien yksikkötestaamiseen. NUnit tarjoaa C#-kielelle apukirjaston testitapausten ohjelmointiin sekä ajoympäristön testitapausten suorittamiseen ja raportointiin.

Tutkielma perehdyttää lukijan testauksen merkitykseen, tavoitteisiin ja prosessiin yleisellä tasolla. Lisäksi käsitellään yksikkötestausta, testauksen toteuttamista osana ohjelmistoprosessia ja yksikkötestauksen automatisointia NUnit-testausympäristön avulla. Lukijalta odotetaan perustietämystä olio-ohjelmoinnista ja ohjelmistotuotannosta.

Tutkielman toisessa luvussa tarkastellaan testausprosessia osana ohjelmistotuotantoa ja ohjelmistoprosessia. Luvussa 3 perehdytään testausprosessin vaiheisiin ja testauksen tuloksiin. Luvussa 4 käsitellään testauksen tasoja ja luvussa 5 tarkastellaan yksikkötestausta olio-ohjelmoinnissa. Kuudennessa luvussa perehdytään NUnit-testausympäristöön ja seitsemännessä luvussa tarkastellaan yksikkötestitapauksen ohjelmointia NUnit-apukirjaston avulla. Luvussa 8 tutkielman havaintojen ohella pohditaan NUnit-testausympäristön soveltuvuutta yksikkötestaukseen.

2 Testausprosessi osana ohjelmistotuotantoa

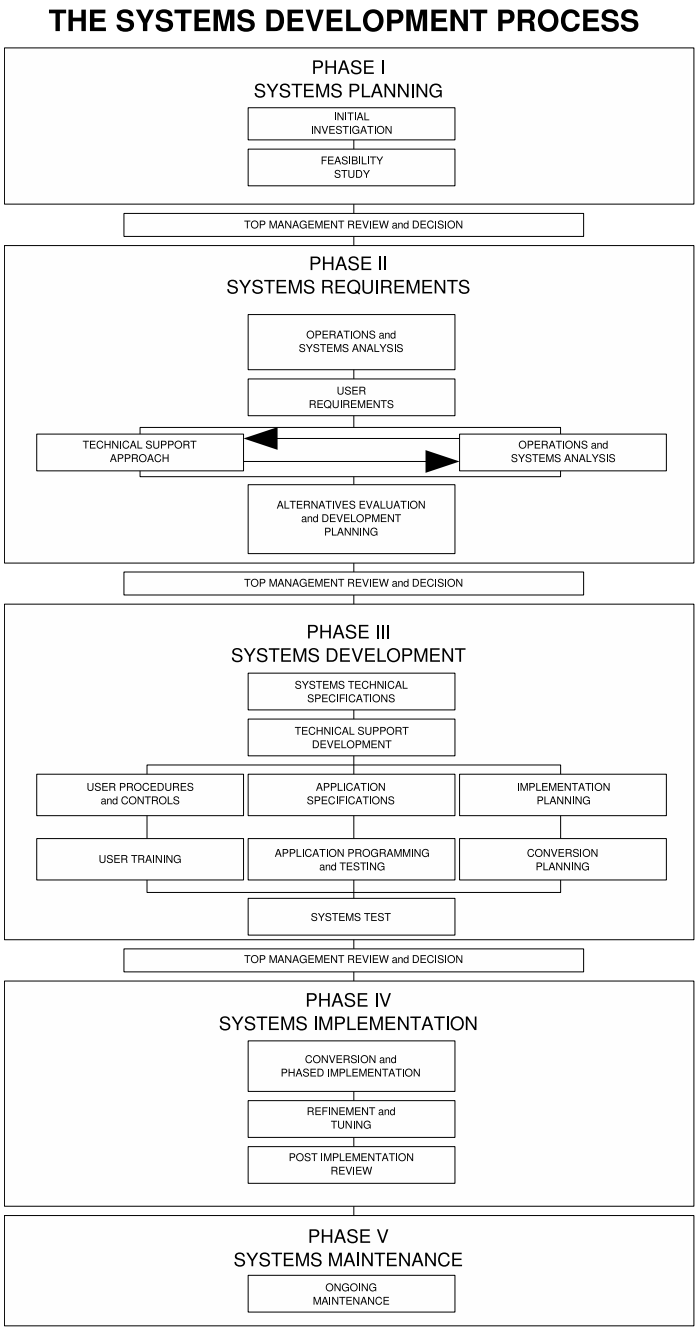
Pressman [10, s. 21] kuvaa ohjelmistoprosessin joukoksi ennalta määriteltyjä tehtäviä ja tuloksia, jotka ovat yhteisiä kaikille ohjelmistoprojekteille koosta tai kompleksisuudesta riippumatta. Pressmanin mukaan ohjelmistoprosessin tarkoitus on olla ohjelmiston toteutuksen toimintamalli, joka mahdollistaa tehokkaan ja ennalta suunnitellun ohjelmiston kehityksen. Luvussa tarkastellaan ohjelmistoprosessia yleisesti ja testausprosessia ohjelmistoprosessin osana. Esimerkkinä prosessimallista tarkastellaan Atkinsin, Biggsin ja Birksin [1, s. 160] prosessimallia, joka on melko perinteinen ohjelmistoprosessin malli.

2.1 Ohjelmistotuotannon prosessimalli

IEEE:n sanastossa [6, s. 57] **prosessi** (engl. *process*) määritellään sarjaksi vaiheita (engl. *phase*), joiden suorittamisella saavutetaan haluttu päämäärä. Ohjelmistotuotannon prosessin määritelmässä [6, s. 67] **ohjelmistoprosessin** (engl. *software development process*) vaiheiksi on katsottu kuuluvan ainakin vaatimusten määrittely (engl. *requirements phase*), ohjelmiston suunnittelu (engl. *design phase*), toteutus (engl. *implementation phase*), testaus (engl. *test phase*) ja käyttöönotto (engl. *installation and checkout phase*).

Erilaisten ohjelmistoprojektien tarpeisiin on kehitetty lukuisia **prosessimalleja** (engl. *process model*), joiden tarkoitus on määritellä projektin tarpeisiin sopiva tapa vaiheistaa ohjelmistokehitys. Eri prosessimalleissa IEEE:n määrittelemät ohjelmistoprosessin vaiheet voidaan suorittaa erilaisissa järjestyksissä ja prosessimallista riippuen vaiheita voidaan kuvata erilaisilla tarkkuuksilla. Esimerkiksi inkrementaalisissa prosessimalleissa vaiheita suoritetaan useaan kertaan ohjelmistokehityksen edetessä.

Atkinsin, Biggsin ja Birksin [1, s. 160] prosessimallissa IEEE:n määrittelemät ohjelmistoprosessin vaiheet ovat esitutkimus (engl. *systems planning*), vaatimusmäärittely (engl. *systems requirements*), toteutus (engl. *systems development*), käyttöönotto (engl. *systems implementation*) ja ylläpito (engl. *systems maintenance*). Jokaisessa vaiheessa on joukko tehtäviä (engl. *task*). Vaikka prosessimallin vaiheet eivät nimien perustella täysin vastaa IEEE:n määrittelyä [6, s. 67], mallin vaiheiden tehtävät sisältävät määrittelyssä kuvatut asiat. Kuvassa 1 on kaavio prosessin vaiheista ja vaiheisiin kuuluvista tehtävistä.



Kuva 1: Atkinsin, Biggsin ja Birksin [1, s. 160] kuvaama ohjelmistokehityksen prosessimalli.

2.2 Ohjelmistotestaus osana ohjelmistotuotannon prosessia

Kuten luvussa 2.1 kuvatusta IEEE:n määrityksestä ilmenee, olennainen osa ohjelmiston toteutusta ohjelmistoprojektissa on testaus. Kuvan 1 prosessimallissa ohjelmistotestaus on mukana useassa tehtävässä. Yleisesti **testauksella** tarkoitetaan toimenpiteitä, joilla pyritään varmistamaan, todentamaan tai arvioimaan ohjelmiston tai sen osan määrittelyn mukainen ja virheetön toiminta.

Atkinsin, Biggsin ja Birksin mallissa [1, s. 160] testaukseen liittyviä tehtäviä esiintyy lähinnä järjestelmän toteutusvaiheessa. Kuvassa 1 kaikkia testaukseen liittyviä tehtäviä ei kuitenkaan ole esitetty, sillä kuvaus on yleisellä tasolla.

Prosessimallien tarkoitus on esittää yleisesti koko ohjelmistotuotantoon liittyvä prosessi. Testausta voidaan kuitenkin tarkastella myös omana prosessinaan erilaisine tehtävineen ja vaiheineen. Tällöin puhutaan luvussa 3 käsiteltävästä testausprosessista.

2.3 Ohjelmistotestauksen V-malli

Tersa on kuvannut tutkielmassaan [14, s. 18–21] ohjelmistotestauksen V-mallia. Malli kuvaa **testausprosessin** vaiheiden ja tulosten suhdetta ohjelmistoprosessin vaiheisiin ja tuloksiin. Myös Pressman [10, s. 468–470] yhdistää vaiheet keskenään osin samalla tavalla omassa spiraalimallissaan.

V-malli lähtee liikkeelle siitä, että ohjelmistoprosessissa on aina jollain tasolla vaiheet liittyen

1. vaatimusten kartoitukseen,
2. määrittelyyn,
3. arkkitehtuurin suunnitteluun ja
4. yksityiskohtaiseen suunnitteluun.

Tersan mukaan näitä ohjelmistoprosessin vaiheita vastaavat **testauksen vaiheet** ovat

1. hyväksymistestaus,
2. järjestelmätestaus,
3. integraatiotestaus ja
4. yksikkötestaus.

Yksityiskohtaisempi suunnitelma vastaa siis testaamisessa ohjelmiston yksityiskohtaisempaa testaamista. Ohjelmisto- ja testausprosessien yhdistävänä tekijänä on ohjelmistoprosessin vaiheissa laaditut suunnitelmat, ja testausprosessissa kyseessä olevia suunnitelmia vastaavat testaustoimenpiteet. Kulminaatiopisteenä mallissa on ohjelmointi, jonka jälkeen suunnittelusta siirrytään testausprosessin osalta testauksen toteutukseen.

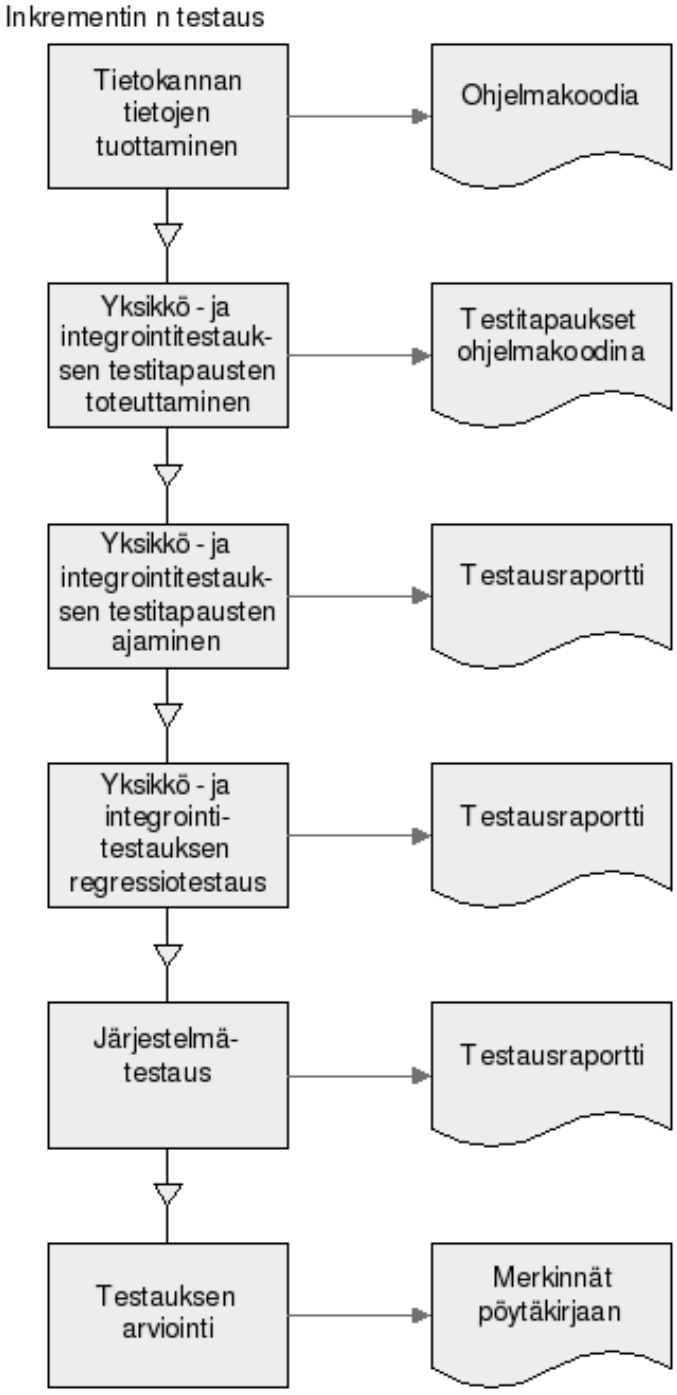
3 Ohjelmistotestauksen tehtäviä ja tuloksia

Yleisesti **testauksen tavoitteena** on ohjelman määritysten mukaisen toiminnallisuuden varmistaminen. National Institute of Standards & Technology listaa raportissaan [11, s. 1–11] puutteellisen testauksen mahdollisiksi seurauksiksi muun muassa ohjelmiston virheiden määrän kasvun, ohjelmiston kehityksen kustannusten kasvun ja ohjelmiston julkaisun viivästymisen. Ohjelmiston testausta on mahdollista mitata esimerkiksi testien lähdekoodin kattavuudella (engl. *code coverage*) kuvaten sitä, kuinka paljon lähdekoodista saadaan katettua testitapauksilla. Luvussa kuvataan testauksen tavoitteita ja hyötyjä sekä testausprosessiin yleisesti kuuluvia vaiheita ja tehtäviä. Luku perustuu pääasiassa lähteisiin [5, s. 9–15] ja [4, s. 3–15].

3.1 Testauksen suunnittelu

Yksi ohjelmistoprojektin tärkeä osa on dokumentointi. IEEE on määritellyt standardissa 829-1998 [4, s. 3–15] **testausprosessin** tuloksiksi muun muassa testaussuunnitelman (engl. *test plan*), testitapauksen (engl. *test case*), testitapausten raporttien (engl. *test log*) ja testausraportin (engl. *test summary report*) sisällöt. Standardin [4, s. 3–6] mukaan **testaussuunnitelmassa** määritellään testausympäristö, testauksen dokumentaatio, ohjelmiston testattavat osat ja ominaisuudet, testien hyväksymisen kriteerit, testaamiseen liittyvät roolit ja vastuuhenkilöt, aikataulu, testausprosessiin ja testaukseen liittyvät riskit sekä riskienhallinta.

Testaussuunnitelma laaditaan yleensä ohjelmistoprosessin suunnitteluvaiheessa, ja siihen kuuluu myös **testausprosessin kuvaus**. Prosessin kuvauksessa määritellään suoritettavat vaiheet ja tehtävät, mutta niitä ei yleensä sidota kalenteriaikaan. Prosessin kuvauksessa voidaan kuvata myös prosessin vaiheen tuloksia. Kuvassa 2 on esimerkki inkrementaalisesta testausprosessin kuvauksesta.



Kuva 2: Sitrus-projektin inkrementin testausprosessi [8, Liite 2].

3.2 Testausympäristön rakentaminen

Testausympäristöön kuuluvien laitteistojen ja ohjelmistojen tarve on yleensä kartoitettu ohjelmistoprosessin suunnitteluvaiheessa ja kirjattu testaussuunnitelmaan [4, s. 5]. Testausympäristöön voi kuulua kehitettävästä ohjelmistosta riippuen testauspalvelin, testitietokanta sekä muita ohjelmiston testaamisessa tarvittavia ohjelmistoja ja laitteistoja. Testausympäristön olennaisin osa on ajoympäristö, joka on testitapaukset suorittava ja testauskerran tulokset raportoiva ohjelmisto. Liitteessä 1 on kuvattu NUnit-testausympäristön rakentaminen ja integroiminen Microsoft Visual Studio 2005 -sovelluskehitysympäristöön.

3.3 Testauksen suorittaminen

IEEE:n sanasto [6, s. 74–75] määrittelee **testitapauksen** joukoksi syötteitä ja palautteita, joilla testataan yksittäistä toiminnallisuutta. Tutkielmassa testitapauksella tarkoitetaan laajemmin luokan metodin, olion tilan, olion tilasiirtymän tai vastaavan toiminnallisen osan testaamista, mihin voi kuulua useita yksittäisiä syötejoukkoja ja palautteiden tarkistamisia. Yhteen **testauskertaan** voi kuulua useita testitapauksia, jotka suoritetaan testausprosessin mukaisesti samassa vaiheessa.

IEEE on laatinut standardin [5, s. 9] testitapauksen suorittamiseen. **Testitapauksen suorittaminen** on riippuvainen käytetyistä tekniikoista ja testausympäristöstä. Yleisellä tasolla testaus suoritetaan testausympäristössä testitapauksessa määritellyillä syötteillä. Testitapauksen suorittamisesta syntyy yleensä palaute, jota verrataan sille määriteltyyn odotettuun palautteeseen. Jos saatu palaute ja odotettu palaute ovat samat sekä ne täyttävät testaussuunnitelmissa määritellyt muut hyväksymiskriteerit, ohjelmisto on läpäissyt testin. Jos tulos poikkeaa odotetusta, ohjelmisto ei ole läpäissyt testiä, ja testin tulos täytyy analysoida. Virheellinen palaute voi johtua virheestä testitapauksessa, testauksen suorittamisessa, testausympäristössä, ohjelmistossa tai määrittelyssä. Suoritetusta testauskerrasta tulisi laatia raportti. Luvussa 3.4 on kuvattu testaukseen liittyvää raportointia.

Jos testaus tai osa siitä suoritetaan ohjelmallisesti, testitapausten suorittamiseen voi kuulua myös testitapausten ohjelmointi. Testitapausten ohjelmointia NUnit-testausympäristöön on kuvattu luvussa 7.

3.4 Testauksen raportointi

Testaussuunnitelmassa määritellään testauksen tuloksena syntyvät raportit. Suoritetusta testitapauksesta laaditaan testitapauksen raportti ja siinä tulisi mainita ainakin

suoritettu testitapaus, testattavien ohjelmanosien versiot, testausympäristö, suorittaja, ajankohta ja tulos. Tuloksessa tulisi mainita palautteet ja mahdolliset viestit tai poikkeukset sekä läpäistiinkö testi vai ei. Läpäisemättömästä testistä voidaan laatia yksityiskohtaisempi **virheraportti** (engl. *test incident report*), jossa kuvataan virhettä, siihen johtaneita tapahtumia ja virheen vaikutuksia [5, s. 13–14].

Tuloksena kustakin testauskerrasta tulisi laatia testausraportti, jossa mainitaan ainakin testauksen kohde, suorittaja ja suoritetut testitapaukset sekä yhteenveto testitapauksien tuloksista [5, s. 11–12]. Näiden lisäksi testauksesta tulisi laatia koko testausta koostava testausraportti [5, s. 14–15].

3.5 Testauksen automatisointi

Automatisoidulla testauksella tarkoitetaan ohjelmallisesti ajettavaa testiä. Automatisoinnin tarkoituksena on kertaalleen kirjoitetun testitapauksen uudelleenkäyttö, joten se mahdollistaa testitapauksen suorittamisen useaan kertaan ohjelmistokehityksen aikana ilman testitapausten manuaalista suorittamista. Tällöin uudelleentestaukseen käytetty työmäärä vähenee verrattuna automatisoimattomaan, manuaaliseen uudelleentestaamiseen.

Automatisoinnin yhteydessä puhutaan Youngin [15, s. 6] mukaan usein investoinnin kannattavuudesta eli ROI-arvosta (engl. *Return on Investment)*. Tällä tarkoitetaan testauksen automatisointiin käytetyn työmäärän suhdetta manuaaliseen testaukseen. ROI-arvo suurenee eli investoinnin hyöty kasvaa, kun luvussa 4.4 käsitellyn regressiotestausten testauskertojen määrä kasvaa ohjelmistoprojektissa.

Testauksen automatisointiin käytetään testausympäristöä, joka voi olla esimerkiksi luvussa 6 esitettävä NUnit-testausympäristö. Laajasti ajateltuna **testausympäristö**llä tarkoitetaan laitteistoa ja ohjelmistoa, jossa testitapaukset suoritetaan. Tällöin käytössä voi olla muun muassa pelkästään testaamiseen tarkoitettu testauspalvelin ja testitietokanta. Tutkielmassa testausympäristöllä kuitenkin tarkoitetaan kehittäjän koneella olevaa ajoympäristöä, jossa testitapaukset suoritetaan.

Yleensä testausympäristön valinta joudutaan tekemään testattavan ohjelmiston ohjelmointikielen mukaisesti, sillä testitapaukset kirjoitetaan yleensä samalla ohjelmointikielellä kuin testattava ohjelma. Poikkeuksena esimerkiksi testattaessa avoimia rajapintoja voidaan käyttää mitä tahansa kyseistä rajapintaa tukevaa tekniikkaa. Web Services on yksi tällainen avoin rajapintatekniikka, jota on mahdollista testata hyvin useilla eri testausympäristöillä.

4 Testauksen tasot

Testauksen tasoilla tarkoitetaan ohjelmakoodin rakenteen mukaisesti jaettuja testauksen tasoja. Atkinsin, Biggsin ja Birksin [1, s. 160] prosessimallin mukaisesti testaus voidaan jakaa seuraaviin tasoihin:

- yksikkötestaus,
- integraatiotestaus ja
- järjestelmätestaus.

Näiden lisäksi prosessimalleissa on usein määritelty regressiotestaus yhdeksi tasoksi. Luvussa perehdytään eri testaustasojen määrityksiin pääasiassa IEEE:n määritysten [6, s. 9, 41 ja 79] pohjalta.

4.1 Yksikkötestaus

IEEE:n määrityksen [6, s. 79] mukaan yksikkötestauksella (engl. *unit testing)* tarkoitetaan yksittäisten ohjelman osien testausta. Ohjelman osalla voidaan tarkoittaa yksittäistä funktiota tai luokkaa.

Runesonin [12, s. 28] mukaan ohjelmistokehityksessä yksikkötestauksen suorittaa yleensä kyseisen yksikön eli ohjelman osan toteuttaja. Tästä voi kuitenkin seurata se, ettei hän havaitse oman koodinsa virheitä, jolloin ne voivat jäädä testaamatta. Tutkimuksessa [12, s. 25–26] kehotetaan kehittäjiä yksikkötestaamaan osia ristiin, jolloin kehittäjän omaan koodiin jäävät virheet havaitaan paremmin.

Testitapauksen kirjoittaminen voidaan tehdä yksikön toteutuksen jälkeen tai ennen yksikön toteutusta. Jälkimmäistä tapaa kutsutaan testauslähtöiseksi ohjelmoinniksi, koska yksikön odotetun toiminnan mukainen testitapaus kirjoitetaan ennen testattavaa ohjelmakoodia.

4.2 Integraatiotestaus

Nimensä mukaisesti integraatiotestauksella tarkoitetaan ohjelman osien välisten integraatioiden eli vuorovaikutusten testaamista. Vuorovaikutukset liittyvät erilaisiin rajapintoihin tai riippuvuuksiin, joita voivat olla esimerkiksi luokkien riippuvuussuhteet, ohjelman osien eli modulien rajapinnat tai ohjelmiston ja laitteiston väliset rajapinnat [6, s. 41] .

Kerrosarkkitehtuureissa integraatiotestauksella voidaan tarkoittaa niin sanottujen liiketoimintaluokkien testaamista. Liiketoimintaluokat vastaavat luokkien integraatiosta, mutta luokkia voidaan testata luvussa 5 esitellyn yksikkötestauksen tapaan, jolloin liiketoimintaluokkaa testattaessa testataan myös luokkien integraatiota.

4.3 Järjestelmätestaus

Pressman [10, s. 483] kuvaa järjestelmätestausta koko kehitettävän järjestelmän toiminnan varmistamiseksi. Järjestelmätestauksessa ei keskitytä tiettyihin ohjelman sisäisiin osiin, vaan testauksen tavoite on löytää mahdollisia virheitä, jotka ilmenevät sen todellisessa käyttöympäristössä ja käyttötapauksissa. Pressmanin [10, s. 459] mukaan järjestelmätestauksen merkitys korostuu reaaliaikajärjestelmissä, jolloin testauksessa täytyy ottaa huomioon mm. koko järjestelmän suorituskyky, poikkeuskäsittely sekä laitteiston ja ohjelmiston odotettu yhteistyö.

Järjestelmätestauksen testitapaukset eivät ole syötteiden ja palautteiden pareja, vaan pikemminkin kuvauksia ohjelmiston käytöstä ja odotetusta käyttäytymisestä. Tällaisten testitapausten automatisointi on melko uusi tutkimuskohde, mutta nykyään on jo käytössä työkaluja käyttöliittymien ja järjestelmien testaamiseen. Esimerkiksi ASP.NET -tekniikalla toteutettujen käyttöliittymien testaamiseen on kehitetty NUnitAsp-lisäosa NUnit-ympäristöön, jolla on Shoren [13] mukaan mahdollista ohjelmallisesti testata käyttöliittymän toiminnallisuutta.

4.4 Regressiotestaus

IEEE:n sanasto [6, s. 9] määrittelee regressiotestauksen muokatun lähdekoodin uudelleentestaukseksi. Sillä varmistetaan, että muokattu koodi toimii halutulla tavalla, eikä ole rikkonut ohjelman muita toiminnallisuuksia. Regressiotestauksessa siis suoritetaan uudelleen yksikkö-, integrointi- ja järjestelmätestitapauksia.

Pressmanin [10, s. 480] mukaan integraatiotestauksen edetessä ja korjattujen ohjelmaosien lisääntyessä voi uudelleentestauksien määrä kasvaa suuresti. Tämän vuoksi Pressman ehdottaa, että regressiotestauksessa testattavat testitapaukset tulisi määritellä koskemaan vain niitä testitapauksia, joihin korjaukset ovat voineet vaikuttaa. Testauksen automatisointi kuitenkin mahdollistaa kaikkien testitapausten suorittamisen melko pienellä vaivalla, jolloin ajettavien testitapausten määrittelyyn kuluva aika voidaan säästää suorittamalla kaikki ohjelmallisesti automatisoidut testit aina tarvittaessa.

5 Yksikkötestaus olio-ohjelmoinnissa

Yksikkötestaus on perinteisesti ajateltu proseduraalisissa kielissä yksittäisen aliohjelman testaamiseksi. Chen, Gao, Hsia ja Toyoshima [2, s. 2] toteavat, että olio-ohjelmoinnissa yksittäisen toiminnallisuuden testaaminen voi olla vaikeampaa, sillä luokasta voidaan tehdä erilaisia instansseja eli olion ilmentymiä, joiden toiminnallisuus voi vaihdella sen tilasta riippuen. Tällaisen luokan testaamisessa voidaan käyttää luvussa 5.1 esiteltyä olion tilaan perustuvaa testaamista. Vastaavaa ongelmaa ei ole olion staattisilla metodeilla, sillä ne eivät ole riippuvaisia yksittäisen olion tilasta.

Chen, Gao, Hsia ja Toyoshima [2, s. 1–5] tarkentavat yksikkötestauksen merkityksen **olio-ohjelmoinnin** osalta tarkoittavan myös luokkien rajapintojen, vuorovaikutuksen ja yhdistettyjen ominaisuuksien testaamista, koska luokkien toimintaa voidaan testata vasta luokan instanssin eli olion toimiessa muiden luokkien olioiden vuorovaikutuksessa. Tällöin testaaminen käsittää yksikkötestauksen lisäksi integraatiotestauksen tavoitteita. Runesonin tutkimuksen [12] mukaan joissakin tapauksissa yksikkötestausta käytetään myös ohjelman osien hyväksymiskäytänteenä.

5.1 Olion tilojen testaaminen

Olion tilasta on teknisten määrittelyjen tai koodin tarkastelun myötä mahdollista löytää sallittuja ja virheellisiä olion tiloja. Näitä tiloja voidaan testitapauksilla testata ja siten tarkastella olion toimivuutta.

Esimerkkinä olion tilasta voidaan tarkastella olioksi mallinnettua ovea, joka voi olla `auki` tai `kiinni`, ja kiinni ollessaan se voi olla joko `lukossa` tai `lukitsematta`. Oven tiloja voi muokata neljällä toiminnolla: `avaa ovi`, `sulje ovi`, `lukitse` ja `avaa lukko`. Tiloista lukko `lukossa` ja ovi `auki` ovat toisensa poissulkevia, eli ne eivät voi olla yhtäaikaa voimassa. Kyseessä on siis olion virheellinen tila, johon sen ei pitäisi koskaan joutua.

Esimerkistä voidaan johtaa tilojen ja tilasiirtymien taulukko 1 sekä virheellisten tilojen taulukko 2. Taulukoissa on määritelty mallinnetun oven tilat, mutta ne eivät vielä ota kantaa toteutukseen. Liitteessä 2 on toteutettu määrityksiä vastaava luokka ja luvussa 7 kuvataan luokan yksikkötestien ohjelmointi.

Oven tila ennen	Toiminto	Odotettu tulos	Oven tila jälkeen
`kiinni, lukitsematta`	`avaa ovi`	Ovi aukeaa	`auki, lukitsematta`
`kiinni, lukitsematta`	`lukitse`	Ovi lukittuu	`kiinni, lukittu`
`kiinni, lukittu`	`avaa lukko`	Lukko aukeaa	`kiinni, lukitsematta`
`auki, lukitsematta`	`sulje ovi`	Ovi sulkeutuu	`kiinni, lukitsematta`

Taulukko 1: Oven sallitut tilat ja tilasiirtymät.

Oven tila ennen	Toiminto	Odotettu tulos	Oven tila jälkeen
`kiinni, lukitsematta`	`sulje ovi`	virhe	muuttumaton
`kiinni, lukitsematta`	`avaa lukko`	virhe	muuttumaton
`kiinni, lukittu`	`avaa ovi`	virhe	muuttumaton
`kiinni, lukittu`	`sulje ovi`	virhe	muuttumaton
`kiinni, lukittu`	`lukitse`	virhe	muuttumaton
`auki, lukitsematta`	`avaa ovi`	virhe	muuttumaton
`auki, lukitsematta`	`lukitse`	virhe	muuttumaton
`auki, lukitsematta`	`avaa lukko`	virhe	muuttumaton

Taulukko 2: Oven tilat ja virheelliset tilasiirtymät.

5.2 Laadullisten vaatimusten huomioiminen

Taulukot 1 ja 2 on laadittu oviolion toiminnallisista vaatimuksista. Tämän lisäksi testitapausten suunnittelemisessa ja laatimisessa täytyy ottaa huomioon vaatimusmäärittelyssä ja muissa määrittelydokumenteissa esitetyt laadulliset vaatimukset. Esimerkin tapauksessa tietoturvan osalta on määritelty, että oven toimintojen `lukitse` ja `avaa lukko` yhteydessä on käytettävä oliota luotaessa määriteltyä avainta. Kyseisiä metodeja vastaavissa testitapauksissa tulisi siten ottaa huomioon taulukon 3 mukaisesti oikean ja väärän avaimen käyttö lukittaessa ja avattaessa lukkoa.

Taulukoiden 1, 2 ja 3 jokainen rivi on käytännössä kuvaus yksittäisestä testistä. Käytänteistä riippuen jokaisesta tilasiirtymästä voidaan tehdä sitä vastaava testitapaus, tai esimerkiksi jokaista metodia kohden tehdään yksi testitapaus, jossa testataan kaikki metodia koskevat toiminnot.

Oven tila ennen	Toiminto	Avain	Odotettu tulos	Oven tila jälkeen
`kiinni`, `lukitsematta`	`lukitse`	Oikea	Ovi lukittuu	`kiinni`, `lukittu`
`kiinni`, `lukitsematta`	`lukitse`	Väärä	virhe	muuttumaton
`kiinni`, `lukittu`	`avaa lukko`	Oikea	Lukko avautuu	`kiinni`, `lukitsematta`
`kiinni`, `lukittu`	`avaa lukko`	Väärä	virhe	muuttumaton

Taulukko 3: Oven tarkennetut toimintojen `lukitse` ja `avaa lukko` tilasiirtymät.

5.3 Yksikkötestauksen automatisoinnin hyödyt

Automatisoitua testausta voidaan käyttää paitsi **virheiden etsintään** myös apuna ohjelmakoodiin tehtävien muutosten ja niin sanotun **refaktoroinnin** eli ohjelman sisäisen toteutuksen muokkaamisen yhteydessä. Hyöty korostuu erityisesti kantaluokissa, jolloin perittyjen luokkien toiminta voidaan varmistaa suorittamalla automatisoidut testit. Edellä kuvattuja hyötyjä on raportoitu esimerkiksi Sitrus-projektin projektiraportissa [7, s. 35].

Automatisoidun testauksen **testitapausten** raportointi tapahtuu useimmiten automaattisesti testausympäristön työkaluilla. Esimerkiksi NUnit-testausympäristö antaa testauksen tulokset XML-muotoisena. Liitteessä 3 on kuvattu vaiheet, joilla tulokset voidaan muokata visuaaliseen muotoon HTML-sivuksi.

6 NUnit-testausympäristö

NUnit-testausympäristö koostuu ajoympäristöstä ja apukirjastosta. Ajoympäristöstä on sekä graafinen käyttöliittymä että konsolissa ajettava sovellus. Apukirjasto helpottaa testitapausten ohjelmointia. Luku perustuu pääosin NUnit-dokumentaatioon [9].

6.1 NUnit.framework-apukirjasto

Testiluokkien ja -tapausten laatimisessa käytetään usein hierarkista lähestymistä siten, että yksi **testiluokka** testaa yhtä luokkaa ja kutakin luokan tilasiirtymää kohden laaditaan yksi **testitapaus**.

Apukirjasto `NUnit.framework` tarjoaa testiluokkien ohjelmoijalle sovelluskehyksen, joka helpottaa testiluokkien ja testitapausten kirjoittamista. Se tarjoaa notaation metodien attribuuttien määrittelyyn sekä `Assert`-luokan vertailujen ja testitapausten hallintaan. NUnit-apukirjasto otetaan käyttöön testiluokassa lisäämällä `NUnit.framework`-kirjasto testiluokan käyttöön komennolla `using NUnit.Framework;` testiluokan lähdekoodin alussa.

6.2 Testiluokan määrittely

Ohjelmoitava testiluokka määritellään testausympäristössä ajettavaksi testiluokaksi asettamalla attribuutti `TestFixture` notaatiolla

```
[TestFixture(Description="Testiluokan kuvaus")]
```

ennen luokan määrittelyä. `Description`-teksti kirjoitetaan osaksi ajossa muodostettavaa raporttia, joten testausraporttien kuvaustekstit voidaan kirjoittaa osana testiluokan ja testitapausten ohjelmointia lähdekoodiin.

Testiluokan testaavat metodit eli testitapaukset määritellään `[Test]`-notaatiolla, jolle voidaan antaa myös `Description`-kuvaus samalla tavalla kuin testiluokan määrittelyssä. **Testitapauksesta odotettu poikkeus** on mahdollista määritellä myös notaatiolla

```
[ExpectedException(typeof(OdotetunPoikkeuksenTyyppi))]
```

ennen testaavaa metodia. Tässä tapauksessa testitapaus suoritetaan onnistuneesti, jos määritelty poikkeus heitetään metodista.

Testiluokalle on mahdollista kirjoittaa konstruktorin tyyppinen metodi, jolla voidaan alustaa testitapausten yhteisiä muuttujia tai olioita. Tällainen metodi merki-

tään [SetUp]-notaatiolla. Vastaavasti voidaan kirjoittaa destruktorin tyyppinen metodi [TearDown]-notaatiolla, jolla voidaan hallita testauksen suorituksen jälkeen tehtäviä toimintoja, esimerkiksi testitietokannan palauttamiseksi oletustilaan.

Edellä kuvatuilla notaatiolla saadaan **määriteltyä testiluokan rakenne** esimerkiksi seuraavasti:

```
[TextFixture(Description="Testiluokan kuvaus")]
public class Testitapaus {
   [SetUp]
   puclic void setUp() { .. }
   [TearDown]
   puclic void tearDown() { .. }
   [Test(Description="Testitapauksen kuvaus")]
   [ExpectedException(typeof(OdotetunPoikkeuksenTyyppi))]
   public void expectAnException(){ .. }
}
```

Test-notaatiolla merkittyjä testimetodeja voi olla useita.

6.3 Vertailu ja tarkistaminen Assert-luokan avulla

Assert-luokka mahdollistaa **odotusarvon ja paluu- tai väliarvon vertailemista** keskenään. Esimerkiksi liitteessä 2 toteutettua oviolion tilaa voidaan verrata metodilla

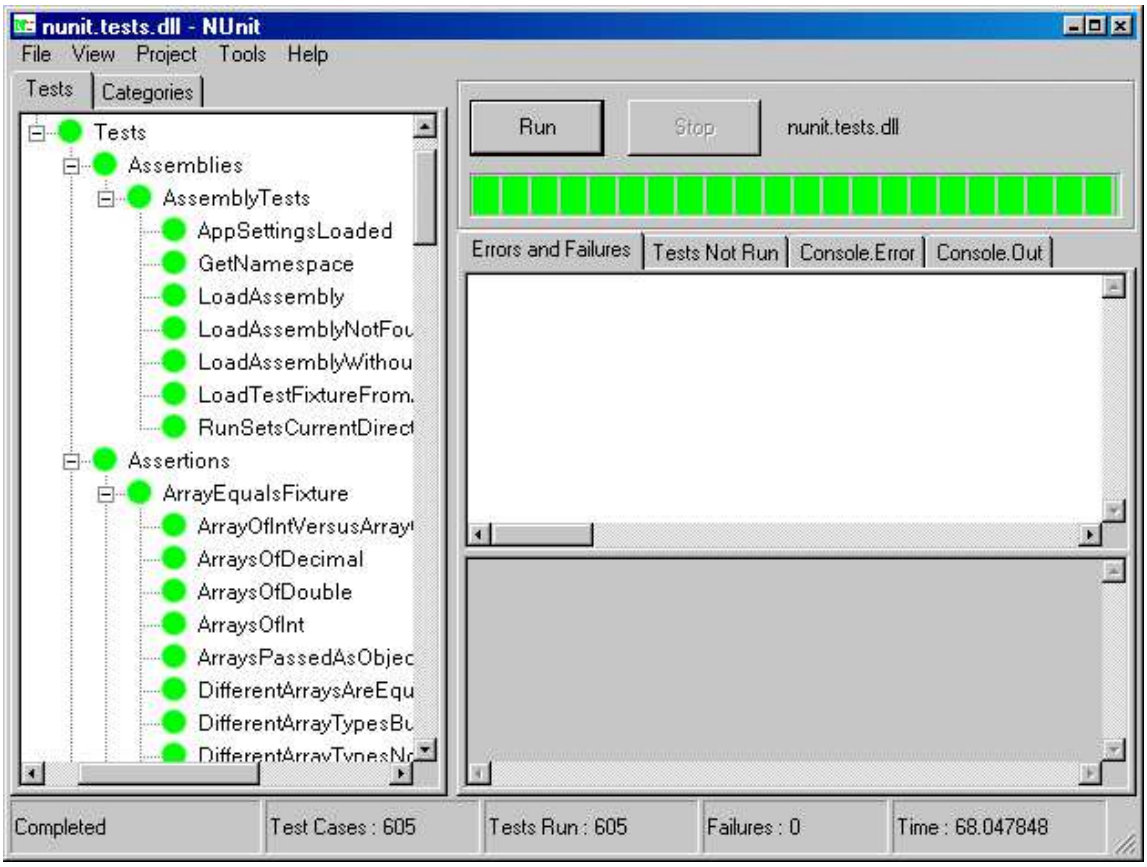
```
Door door = new Door("avain")
Assert.AreEqual(door.isLocked(), true)
```

Metodi `isLocked()` palauttaa boolean-arvon `true` oven ollessa `lukittu` ja `false` oven ollessa `lukitsematta`. Vertailumetodin `Assert.AreEqual` ensimmäinen parametri on olion testattava metodi tai attribuutti ja toinen parametri odotettu tulos.

Assert-luokassa on kaikille perustietotyypeille ja olioille vertailumetodit. Näiden lisäksi Assert-luokassa on tyhjän eli null-viitteen testaamiseen Assert.isNull-metodi. Jos Assert-luokan vertailu palauttaa epätoden, testi katsotaan epäonnistuneeksi, ja Assert-luokka kutsuu Assert.Fail-metodia. Käyttäjä voi myös itse kutsua metodia Assert.Fail, mikä mahdollistaa omien tarkistusten laatimisen.

6.4 NUnit-ajoympäristö

NUnit-ajoympäristön graafinen käyttöliittymä on helppokäyttöinen ympäristö laadittujen testitapausten suorittamiseen ja tulosten tarkasteluun. Kuvassa 3 esitetään ajoympäristön käyttöliittymä. Kuvassa vasemmalla on projektihierarkiaikkuna ja oikealla testitapausten tulosteet. Ajoympäristöön avataan **testiprojektina** tiedosto, jos-



Kuva 3: NUnit-ajoympäristön käyttöliittymä [9].

sa on viittaus muun muassa projektin testiluokkiin. Käyttöliittymän *Run*-painikkeesta painamalla **ajoympäristö suorittaa projektin testiluokkien testitapaukset**. Käyttöliittymän projektihierarkiaikkunan puuvalikosta voidaan valita yksittäinen testiluokka tai testitapaus ajettavaksi. Testitapausten uudelleenajo tapahtuu myös *Run*-painikkeesta. Puuvalikko ilmoittaa ajettujen **testitapausten tulokset** värikoodauksella seuraavasti:

Harmaa tarkoittaa, ettei testitapausta ole vielä suoritettu.

Vihreä tarkoittaa onnistuneesti suoritettua testitapausta.

Keltainen tarkoittaa ohitettua testitapausta.

Punainen tarkoittaa epäonnistunutta testitapausta.

Lisäksi ajoympäristö laatii **yhteenvedon ajetuista testitapauksista** kertoen testitapausten kokonaismäärän, ajettujen, epäonnistuneiden tai ohitettujen testitapausten määrän sekä testaamiseen kuluneen ajan. Ajonaikaiset virheet ja odottamattomat poikkeukset tulostuvat ajoympäristön *Errors and Failures* -välilehdelle. Ajoympäristö antaa testitapauksien tulokset myös XML-muotoisena, jota on mahdollista muokata esimerkiksi HTML-muotoon liitteessä 3 kuvatulla tavalla.

Ajoympäristöä on mahdollista käyttää myös **virheenjäljityksen** tavoin, sillä metodilla `Console.WriteLine` on mahdollista tulostaa esimerkiksi olion tilatietoja testattavasta lähdekoodista tai testiluokan lähdekoodista ajoympäristön *Console Out* -välilehdelle. Tulostettujen tilatietojen perusteella on mahdollista jäljittää lähdekoodin toimintaa. `Console.WriteLine`-metodia kutsutaan esimerkiksi luokan lähdekoodista seuraavasti:

```
Console.WriteLine("Olion tila: "+ this.state);
```

jolloin lähdekoodia ajettaessa ajoympäristön *Console Out* -välilehdelle tulee tekstinä olion `state`-muuttujan arvo.

7 NUnit-testitapauksen toteutus

Luvussa laaditaan luvussa 5.1 esimerkkinä käytetyn oviolion testitapaukset C#-kielellä NUnit-apukirjastoa käyttäen. Luvussa 5.2 on esitelty yleisesti oviolion testaamista ja määritelty olion tilasiirtymät. Liitteessä 2 on kuvattu toteutus testattavasta luokasta C#-kielellä.

7.1 Testiluokan alustus

Ovioliota luotaessa konstruktorissa annetaan parametrina avainmerkkijono, jolla oven lukko on mahdollista lukita tai avata. **Oviolion** metodit ovat seuraavat:

`openDoor()`	avaa lukitsemattoman oven.
`closeDoor()`	sulkee avonaisen oven.
`lockDoor(String key)`	lukitsee suljetun oven avaimella `key`.
`unlockDoor(String key)`	avaa lukon avaimella `key`.

Oviluokan yksikkötestaamisen toteutus aloitetaan kirjoittamalla testiluokka `DoorUnitTest.cs`. Testiluokan toteutuksessa tarvitaan luvussa 6.1 kuvattua `NUnit.Framework`-apukirjastoa, joka saadaan luokan käyttöön luokan alkuun sijoitettavalla komennolla

```
using NUnit.Framework;
```

Ennen testiluokan määrittelyä, testiluokan lähdekoodiin lisätään NUnit-notaation mukaisella `TextFixture`-attribuutilla yksikkötestauksen lyhyt kuvaus seuraavasti:

```
[TextFixture(Description="Yksikkötestaus oviolion
  toiminnallisuuden testaamiseen.")]
public class DoorUnitTest
{
```

Testitapauksia varten luodaan oviluokasta instanssi eli yksi olio testattavaksi sekä testitapauksissa hyödynnettävät muuttujat `correctKey` ja `wrongKey`, joilla voidaan testata oven lukon toimintaa. Nämä luodaan luokan alussa ja alustetaan `SetUp`-metodissa seuraavasti:

```
public Tester.Door door;
public String correctKey
public String wrongKey

[SetUp]
public void TestSetUp(){
  correctKey = "correct test key";
  wrongKey = "incorrect test key";
```

```
        door = new Door(correctKey);
        Assert.isNotNull(door);
    }
```

Toinen vaihtoehto olisi luoda oma oviolio ja avaimet jokaisessa testitapauksessa, jolloin ne eivät olisi riippuvaisia toisistaan. Testitapauksissa voidaan kuitenkin edetä vaihe vaiheelta eteenpäin oven testaamisessa käyttäen samaa instanssia. Jo `SetUp`-metodissa voidaan tehdä tarkistuksia, kuten edellä on testattu, ettei oviolio ole `null`.

7.2 Oviolion avaamisen ja sulkemisen sallitut tilasiirtymät

Ensimmäisessä testitapauksessa testataan **oven avaamista ja sulkemista** eli luvun 5.1 taulukon 1 ensimmäisen ja neljännen rivin tilasiirtymiä. Riveillä

```
[Test(Description="Testitapaus oven avaamiseen ja sulkemiseen.")]
public void TestOpenAndCloseDoor(){
```

oletetaan oviolion olevan oletuksena kiinni ja lukitsematta. Se voidaan kuitenkin tarkistaa seuraavasti:

```
if (door.getState() != Door.State.Closed ||
   door.isLocked())
  Assert.Fail("Testi epäonnistui. Olio ei ole oletustilassa.");
```

Jos oviolio ei ole oletustilassa, testi tulkitaan epäonnistuneeksi.

Testitapauksen ohjelmointia voidaan jatkaa **avaamalla ovi ja tarkistamalla oviolion tila**. Jos tarkastetaan oviluokan toteutusta, `openDoor`-metodi palauttaa boolean-arvon `true` oven avaamisen onnistuessa ja `false` oven avaamisen epäonnistuessa. Tätä voidaan käyttää testitapauksessa hyödyksi seuraavasti:

```
Assert.isTrue(door.openDoor());
Assert.AreEqual(door.getState(), Door.State.Open);
```

Edellä siis avattiin ovi, varmistettiin paluuarvon olevan tosi ja oven tilan olevan avoin. Luvussa 7.3 jatketaan testitapauksen toteutusta virhetilanteiden osalta.

7.3 Oviolion avaamisen ja sulkemisen virhetilanteet

Luvun 7.2 testitapauksessa avattiin ovi ja varmitettiin, että olion tila on auki. Mikäli testaussuunnitelmassa ei ole toisin sovittu, samassa testitapauksessa voidaan toteuttaa myös niitä poikkeustilojen käsittelyjä, jotka testin aikana oleviin olion tiloihin liittyvät. Mikäli poikkeustilanteita käsitellään, tulisi se dokumentoida myös testitapauksen kuvauksessa.

Testataan ne oven **virhetilanteet**, joita voi syntyä **oven ollessa avoinna**. Nämä virhetilanteet on kuvattu luvun 5.1 taulukossa 2, josta testataan alkutilan `auki`,

`lukitsematta` omaavat tilat eli kolme viimeistä riviä seuraavasti:

```
Assert.isFalse(door.openDoor());
Assert.AreEqual(door.getState(), Door.State.Open);
Assert.isFalse(door.lockDoor(correctKey));
Assert.AreEqual(door.getState(), Door.State.Open);
Assert.isFalse(door.unlockDoor(correctKey));
Assert.AreEqual(door.getState(), Door.State.Open);
```

Kun kaikki oven avaamiseen liittyvät poikkeustilat ja tilasiirtymät on testattu, voidaan testata vielä **oven sulkemista**, joka on luvun 5.1 taulukon 1 neljäs sallittu tilasiirtymä seuraavasti:

```
Assert.isTrue(door.closeDoor());
Assert.AreEqual(door.getState(), Door.State.Closed);
Console.WriteLine("Ovi avattiin ja suljettiin määritysten
 mukaisesti.");
```

Edellä olevien kutsujen jälkeen ovi on oletustilassaan eli suljettuna ja lukitsematta. Testitapauksen lopuksi voidaan tulostaa konsoliin testitapauksen tulos.

7.4 Oviolion lukitsemisen ja lukon avaamisen virhetilanteet

Oviolion toisena testitapauksena voidaan testata oven lukitsemista ja lukon avaamista sekä oikeilla että väärillä avaimilla eli luvun 5.2 taulukon 3 tilasiirtymät. Testitapauksen esittelyyn lisätään kuvaus seuraavasti:

```
[Test(Description="Testitapaus oven lukitsemiseen ja lukon
 avaamiseen sekä oikealla että väärällä avaimella.")]
public void TestLockAndUnlockDoor(){
```

Testitapauksen alussa jälleen tarkastetaan olion olevan oletustilassa eli suljettuna ja lukitsematta seuraavasti:

```
if (door.getState() != Door.State.Closed ||
 door.isLocked())
 Assert.Fail("Testi epäonnistui. Olio ei ole oletustilassa.");
```

Oviluokan lähdekoodia tarkastelemalla nähdään, että metodit `lockDoor` ja `unlockDoor` palauttavat boolean-arvon `true` oven lukitsemisen tai lukon avaamisen onnistuessa ja `false` epäonnistuessa. Oven lukitsemista oikealla avaimella sekä lukon avaamista väärällä avaimella testataan seuraavilla riveillä:

```
Assert.isFalse(door.lockDoor(wrongKey));
Assert.isFalse(door.isLocked());
Assert.isTrue(door.lockDoor(correctKey));
```

```
Assert.isTrue(door.isLocked());
Assert.isFalse(door.unlockDoor(wrongKey));
Assert.isTrue(door.isLocked());
```

Lukitun oven avaaminen ei pitäisi onnistua ennen lukon avaamista oikealla avaimella. Seuraavilla riveillä varmistetaan, ettei ovea saada auki:

```
Assert.isFalse(door.openDoor());
Assert.AreEqual(door.getState(), Tester.Door.State.Closed);
Assert.isTrue(door.isLocked());
```

Lopuksi avaamme oven oikealla avaimella ja tarkistamme, että ovi on oletustilassa suljettuna ja lukitsematta seuraavalla tavalla:

```
Assert.isTrue(door.unlockDoor(correctKey));
Assert.isFalse(door.isLocked());
Assert.AreEqual(door.getState(), Tester.Door.State.Closed);
```

Luvuissa 7.2–7.4 on kuvattu kaksi oviolion testitapausta. Testitapaukset eivät kuitenkaan ole vielä täysin kattavat, sillä ne eivät testaa kaikkia luvun 5.1 taulukon 2 rivejä. Testitapaukset olisi mahdollista myös jakaa pienempiin kokonaisuuksiin siten, että yksi testitapaus testaisi vain yhtä olion tilasiirtymää.

7.5 Testitapauksen raportointi

Luvussa 6.4 todettiin ajoympäristön laativan jokaisesta ajokerrasta XML-muotoisen raportin testitapausten tuloksista. Kun ajoympäristö on suorittanut valitut testitapaukset, XML-muotoinen testitapausten raportti saadaan kuvan 3 valikosta *Tools* komennolla *Save Results as XML*. Tätä raporttia voidaan hyödyntää testauksen raportoinnissa.

Liitteessä 3 on kuvattu vaiheet ajoympäristön raportin muuntamiseen kirjalliseen muotoon. Tällöin raportin laatiminen tapahtuu helposti ainakin niiden tietojen osalta, jotka ovat saatavilla ajoympäristöstä.

8 Yhteenveto

Ohjelmistoprosessin tavoitteena on kehittää tarpeiden ja vaatimusten mukainen sekä virheetön ja toimiva ohjelmisto. Jotta ohjelman oikea toiminta voidaan varmistaa, täytyy se testata. Testausta voidaan tarkastella sekä osana ohjelmistoprosessia että omana testausprosessinaan. Tutkielmassa testausprosessin tasoista tarkasteltiin erityisesti yksikkötestausta.

NUnit on testausympäristö C#-, J#-, VisualBasic.NET- ja C++-kielisten ohjelmien testaamiseen. NUnitin lähdekoodi ja asennuspaketti on Internetissä avoimesti saatavilla. Se on osa niin sanottua xUnit-tuoteperhettä, johon kuuluu yleisimpien ohjelmointikielten testausympäristöt.

NUnit-testausympäristön graafinen ajoympäristö mahdollistaa testitapausten suorittamisen ja tulosten raportoinnin vaivattomasti. Testausympäristön `NUnit.Framework`-apukirjasto helpottaa testitapausten toteutusta notaatioiden ja `Assert`-luokan avulla. NUnit-testausympäristön kaltainen työkalu mahdollistaa yksikkötestaamisen automatisoinnin pienellä työmäärällä ja on siksi varteenotettava osa ohjelmistokehittäjän kehitystyökaluja. Tutkielmassa havainnollistettiin NUnit-testausympäristön käyttää olion sallittujen ja virheellisten tilojen testaamisessa käytännön esimerkkien avulla.

NUnit-testausympäristöstä on kehitetty myös muun muassa ASP-tekniikalla toteutettujen käyttöliittymien testaamiseen tarkoitettu `NUnitASP`-apukirjasto [13]. NUnit-testausympäristön käyttöä on siis mahdollista laajentaa yksikkötestaamisesta koko järjestelmän testaamisen automatisointiin. Tutkielmassa perehdyttiin lähinnä yksikkötestaukseen NUnit-testausympäristössä, joten tutkielmaa voisi jatkaa perehtymällä syvällisemmin integrointi-, järjestelmä- ja regressiotestausvaiheisiin sekä tutkia testausympäristön soveltumista ja hyödyntämistä kyseisissä testaustasoissa.

Lähteet

[1] Atkins William, Biggs Charles and Birks Charles, "Managing the Systems Development Process", Prentice-Hall, Englewood Cliffs, N.J, 1980.

[2] Chen Cris, Gao Jerry, Hsia Pei, Kung David and Toyoshima Yasufumi, "Object-Oriented Software Testing, Some Research and Development", Third IEEE International Symposium, 13-14 November 1998, pages 158-165.

[3] Huikari Juha, Jaaranen Teemu, Korttila Matti, Pirttimäki Ville, Pulkkinen Masi, Puranen Tuukka, Räbinä Hannu ja Vesalainen Timo, "Strutsi- ja Sitrus-projektit, Sovellusraportti", Jyväskylän yliopisto, tietotekniikan laitos, 2007.

[4] IEEE-SA Standard Board, "IEEE Standard for Software Test Documentation, IEEE Std 829-1998", The Institute of Electrical and Electronics Engineers, NY, 1998.

[5] IEEE Standard Board, "IEEE Standard for Software Unit Testing, ANSI/IEEE Std 1008-1987", The Institute of Electrical and Electronics Engineers, NY, 1987.

[6] IEEE Standard Board, "IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990", The Institute of Electrical and Electronics Engineers, New York, 1990.

[7] Pirttimäki Ville, Pulkkinen Masi, Puranen Tuukka ja Vesalainen Timo, "Sitrus-projekti, Projektiraportti", Jyväskylän yliopisto, tietotekniikan laitos, 2007.

[8] Pirttimäki Ville, Pulkkinen Masi, Puranen Tuukka ja Vesalainen Timo, "Sitrus-projekti, Testaussuunnitelma", Jyväskylän yliopisto, tietotekniikan laitos, 2007.

[9] Poole Charlie, "NUnit 2.4.5 Documentation", saatavilla HTML-muodossa osoitteesta `http://www.nunit.org/index.php?p=docHome&r=2.4.5`, viitattu 21.5.2008.

[10] Pressman Roger, "Software Engineering: A Practioner's Approach", McGraw-Hill Publishing Company, London, 2000.

[11] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing, Final Report", saatavilla PDF-muodossa osoitteesta `http://www.nist.gov/director/prog-ofc/report02-3.pdf`, National Institute of Standards & Technology, 2002.

[12] Runeson Per, "A Survey of Unit Testing Practices", IEEE Volume 23, Issue 4, July-Aug. 2006, pages 22-29.

[13] Shore James, "NUnitASP - ASP.NET Unit Testing - Overview", saatavilla HTML-muodossa osoitteesta `http://nunitasp.sourceforge.net/index.html`, viitattu 21.5.2007.

[14] Tersa Tiina, "Testausmenetelmien käyttö sovelluksen systeemitestausvaiheessa", pro gradu -tutkielma, Jyväskylän yliopisto, tietotekniikan laitos, 2002.

[15] Young Dan, "Test Automation: An Architected Approach", saatavilla PDF-muodossa osoitteesta `http://www.stickyminds.com/getfile.asp?ot=XML&id=8336&fn=XDD8336filelistfilename1.pdf`, julkaistu STAREAST-konferenssissa vuonna 2004.

Liite 1: Ohje NUnit-testausympäristön rakentamiseen

NUnit-testausympäristön rakentaminen aloitetaan lataamalla NUnit-testausympäristö (saatavilla osoitteesta `http://www.nunit.org/download.html`, viitattu 3.10.2008) ja asentamalla se ohjeiden [9] mukaisesti. Microsoft Visual Studio 2005 on ohjelmistokehitysympäristö, jolla voidaan kirjoittaa C#-kielisiä ohjelmia. Siitä on saatavilla (osoitteesta `http://www.microsoft.com/express/download/default.aspx`, (viitattu 3.10.2008) ilmainen, karsittu Express-versio. Microsoft Visual Studiossa kirjoitetaan testitapaukset. NUnit tarjoaa sovelluskehyksen (engl. *framework*) testitapauksille sekä alustan testien suorittamiseen ja tulosten raportointiin.

Testausta varten Visual Studiossa lisätään kehitettävään järjestelmään (engl. *solution*) uusi projekti valikon *File* komennolla *New Project...* Projektiksi valitaan luokkakirjasto (engl. *class library*), ja siihen liitetään NUnit-sovelluskehys `nunit.framework` sekä testattava projekti valikon *Project* komennolla *Add reference...* Tämän jälkeen testitapaukset kirjoitetaan projektiin `cs`-tiedostoina.

NUnit-testausympäristön graafinen ajoympäristö käynnistetään työpöydän kuvakkeesta tai käynnistysvalikosta. Ohjelmaan täytyy luoda ensimmäisenä uusi projekti valikon *File* komennolla *New Project....* Jotta testausympäristön toiminta saadaan sujuvaksi, täytyy ohjelman asetuksista vaihtaa Visual Studion tuki päälle seuraavasti:

1. Valikosta *Tools* valitaan komento *Options...*
2. Valitaan puusta *Visual Studio.*
3. Valitaan *Enable Visual Studio Support* päälle.

Viimeiseksi liitetään Visual Studiossa tehty testausprojekti NUnit-ohjelmaan valikon *Project* komennolla *Add VS Project....* Tämän jälkeen testausympäristö on käyttövalmis.

Liite 2: Toteutettu ovi-luokka C#-kielellä

```
 // Masi Pulkkinen, 2008.
// The class is used in modelling a generic door that can be opened
// or closed and locked or unlocked with a key.

using System;
using System.Collections.Generic;
using System.Text;

namespace Tester
{
   public class Door
   {
      public enum State { Open, Closed };
      // The door can be locked or unlocked when it is closed.
      private Boolean locked;
      // The door can be open or closed.
      private State state;
      // The key (string) can be used to lock or unlock the door.
      private String key;
      /// <summary>
      /// When the door is generated, it is closed and unlocked by default.
      /// </summary>
      /// <param name="doorKey»The given key is used to generate the lock
      /// to the door.</param>
      public Door(String doorKey)
      {
        this.key = doorKey;
        this.locked = false;
        this.state = State.Closed;
      }
      /// <summary>
      /// The door is opened.
      /// </summary>
      /// <returns>True if the door is opened, otherwise false.
      /// </returns>
```

```
public Boolean openDoor()
{
  if (this.state == State.Closed && this.locked == false)
  {
   this.state = State.Open;
   return true;
  }
  else return false;
}
/// <summary>
/// The door is closed.
/// </summary>
/// <returns>True if the door is closed, otherwise false.
/// </returns>
public Boolean closeDoor()
{
  if (this.state == State.Open)
  {
   this.state = State.Closed;
   return true;
  }
  else return false;
}
/// <summary>
/// The door is locked with the key.
/// </summary>
/// <param name="doorKey»The key is used to lock the door.</param>
/// <returns>True if the key is correct and the door is locked,
/// otherwise false.</returns>
public Boolean lockDoor(String doorKey)
{
  if (this.state == State.Closed && this.locked == false
  && this.key == doorKey)
  {
   this.locked = true;
   return true;
  }
```

```
        else return false;
      }
      /// <summary>
      /// The door is unlocked with the key.
      /// </summary>
      /// <param name="doorKey»The key is used to unlock the door.</param>
      /// <returns>True if the key is correct and the door is unlocked,
      /// otherwise false.</returns>
      public Boolean unlockDoor(String doorKey)
      {
        if (this.state == State.Closed && this.locked == true
        && this.key == doorKey)
        {
         this.locked = false;
         return true;
        }
        else return false;
      }
      /// <summary>
      /// Method to get the current state of the door.
      /// </summary>
      /// <returns>Current state of the door.</returns>
      public State getState()
      {
        return this.state;
      }
      /// <summary>
      /// Method to resolve if the door is locked or not.
      /// </summary>
      /// <returns>True if the door is locked,
      /// otherwise false.</returns>
      public Boolean isLocked()
      {
        return this.locked;
      }
    }
}
```

Liite 3: NUnitin XML-raportin muuntaminen HTML-muotoon

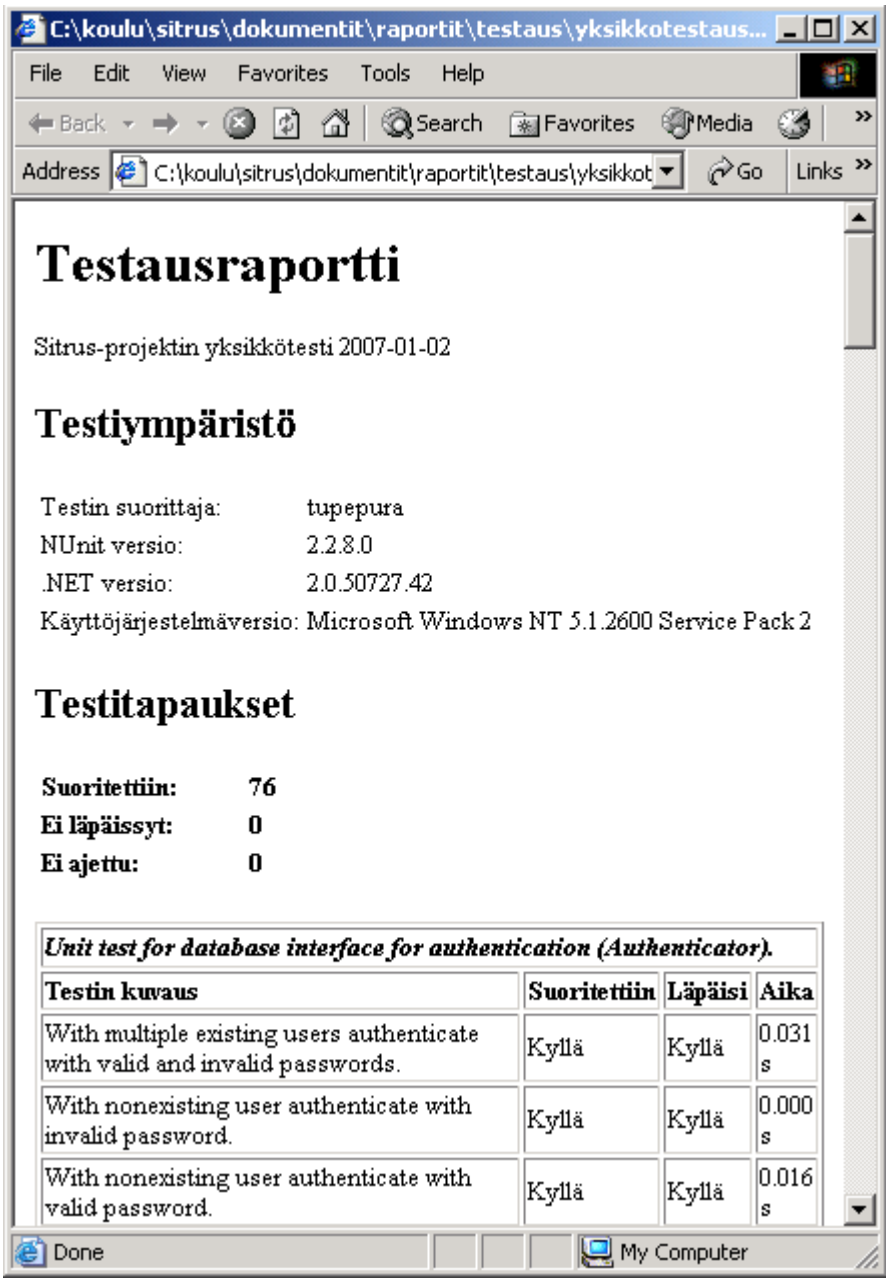
NUnit-ympäristöstä saatava XML-muotoinen testausraportti on muunnettavissa XSL-muuntotiedostolla HTML-muotoiseksi raportiksi. Ohje on muokattu tutkielmaan Strutsi- ja Sitrus-projektien Sovellusraportin [3, s. 90] ohjeen ja XSLT-tiedoston pohjalta. Seuraavassa on vaiheet muunnoksen laatimiseen:

1. Aja NUnit-testitapaukset ajoympäristön graafisessa käyttöliittymässä.
2. Valitse ajon jälkeen *Tools*-valikosta komento *Save Result as XML...*
3. Tallenna tulokset sisältävä XML-tiedosto sopivaan hakemistoon.
4. Kopioi liitteessä 5 oleva teksti `Raportti.xslt`-nimiseksi tiedostoksi samaan hakemistoon kuin juuri tallennettu XML-tiedosto.
5. Lisää XML-tiedoston alkuun rivi

```
<?xml-stylesheet type="text/xsl"href="Raportti.xslt"?>
```

Raportin tarkasteluun tarvittava XML-dokumenttiprosessori (engl. *XML template processor*) löytyy ainakin Internet Explorer 5 -selaimesta. Liitteessä 4 on kuvakaappaus testauskerran raportista, joka on toteutettu edellä kuvatulla tavalla. Raportti ei tosin tällä hetkellä näytä kaikkia epäonnistuneen suorituksen tietoja. Esimerkiksi virheilmoitukset tallennetaan XML-tiedostoon, mutta liitteen 5 tiedosto `Raportti.xslt` ei näitä hyödynnä.

Liite 4: Esimerkki HTML-muotoisesta testausraportista.



Liite 5: Testausraportin XSLT-muunnostiedosto

```
 <xsl:stylesheet version = ’1.0’ xmlns:xsl=
’http://www.w3.org/1999/XSL/Transform’>
<xsl:template match="test-results»
<h1><xsl:text>Testausraportti</xsl:text></h1>
Testausraportti <xsl:value-of select="@date"/>
<h2>Testiympäristö</h2><xsl:apply-templates select="//environment"/>
<h2>Testitapaukset</h2>
<p>
<table border="0»
<tr>
<td width="100» <b>Suoritettiin: </b>
<td> <b> <xsl:value-of select="@total"/> </b> <br/>
</td>
<tr>
<td> <b> Ei läpäissyt: </b>
</td>
<td> <b> <xsl:value-of select="@failures"/> </b> <br/>
</td>
</tr>
<tr>
<td> <b> Ei ajettu: </b>
</td>
<td> <b> <xsl:value-of select="@not-run"/> </b> <br/>
</td>
</tr>
</td>
</tr>
</table></p>
<p> <xsl:for-each select="descendant::test-suite»
<xsl:if test="@description» <p>
<table border="1"width="100%»
<tr>
<td colspan="4» <b> <i> <xsl:value-of select="@description"/> </i> </b>
</td>
</tr>
```

```
<tr>
<td> <b> <xsl:text>Testin kuvaus</xsl:text> </b> </td>
<td> <b> <xsl:text>Suoritettiin</xsl:text> </b> </td>
<td> <b> <xsl:text>Läpäisi</xsl:text> </b> </td>
<td> <b> <xsl:text>Aika</xsl:text> </b> </td>
</tr>
<xsl:for-each select="descendant::test-case»
<tr>
<td width="75%» <xsl:value-of select="@description"/>
</td>
<td width="80»
<xsl:if test="self::node()[@executed='True']»
<xsl:text>Kyllä</xsl:text> </xsl:if>
<xsl:if test="self::node()[@executed='False']»
<xsl:text>Ei</xsl:text> </xsl:if>
</td>
<xsl:if test="self::node()[@executed='True']»
<td width="80» <xsl:if test="self::node()[@success='True']»
<xsl:text>Kyllä</xsl:text> </xsl:if>
<xsl:if test="self::node()[@success='False']»
<xsl:text>Ei</xsl:text> </xsl:if>
</td>
<td width="80» <xsl:value-of select="@time"/> <xsl:text> s</xsl:text>
</td>
</xsl:if>
</tr>
</xsl:for-each>
</table></p>
</xsl:if></xsl:for-each>
</p></xsl:template>
<xsl:template match="environment»
<table border="0»
<tr>
<td> <xsl:text>Testin suorittaja: </xsl:text> </td>
<td> <xsl:value-of select="@user"/> </td>
</tr>
<tr>
```

```
<td> <xsl:text>NUnit versio: </xsl:text> </td>
<td> <xsl:value-of select="@nunit-version"/> </td>
</tr>
<tr>
<td> <xsl:text>.NET versio: </xsl:text> </td>
<td> <xsl:value-of select="@clr-version"/> </td>
</tr>
<tr>
<td> <xsl:text>Käyttöjärjestelmäversio: </xsl:text> </td>
<td> <xsl:value-of select="@os-version"/> </td>
</tr>
</table> </xsl:template>
</xsl:stylesheet>
```