TAMPEREEN TEKNILLINEN YLIOPISTO

KIMMO YLI-ROHDAINEN

TESTIVETOISEN KEHITYKSEN HYÖDYNTÄMINEN SULAUTETUN TIEDONKERUUOHJELMISTON TOTEUTTAMISESSA

Diplomityö

Tarkastaja: professori Seppo Kuikka
Tarkastaja ja aihe hyväksytty
Automaatio-, kone- ja materiaalitekniikan tiedekuntaneuvoston
kokouksessa 5. syyskuuta 2012

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Automaatiotekniikan koulutusohjelma
YLI-ROHDAINEN, KIMMO: Testivetoisen kehityksen hyödyntäminen sulautetun tiedonkeruuohjelmiston toteuttamisessa
Diplomityö, 60 sivua, 0 liitesivua
Lokakuu 2012
Pääaine: Automaation ohjelmistotekniikka
Tarkastaja: professori Seppo Kuikka
Avainsanat: TDD, testivetoinen kehitys, jatkuva integrointi, sulautetut järjestelmät, tiedonkeruuohjelmistot

Testivetoinen kehitys on ohjelmistokehityksessä käytettävä menetelmä, jossa ohjelmoija kirjoittaa aina automaattisen yksikkötestin ennen varsinaisen toiminnallisuuden toteuttamista. Yksikkötestejä luodaan järjestelmän vaatimusten perusteella ja nämä testit ohjaavat ohjelmiston kehitystä. Vaikka testivetoinen kehitys on saavuttanut suosiota työpöytä- ja web-sovellusten kehityksessä, ei sitä ole kuitenkaan sovellettu laajasti sulautettujen järjestelmien kehityksessä. Tämän tutkimuksen tavoitteena on tutkia, miten testivetoista kehitystä voidaan hyödyntää sulautettuja järjestelmiä kehitettäessä.

Tutkimus jakaantuu kahteen vaiheeseen: Ensimmäisessä vaiheessa tutustutaan testivetoiseen kehitykseen kirjallisuustutkimuksen avulla esittelemällä aiheeseen liittyvät käsitteet, teoriat ja menetelmät sekä tutkimalla aiempia kokemuksia ja tutkimuksia menetelmän käytöstä. Toisessa vaiheessa tutkitaan menetelmän käyttöä käytännössä tapaustutkimuksen avulla ja verrataan kokemuksia ensimmäisen osan tuloksiin. Tapaustutkimuksessa tutkitaan menetelmän käyttöä ohjelmistokehitysprojektissa, jossa Bitwise toteuttaa asiakkaalleen Kempille hitsauslaitteeseen liitettävän sulautetun tiedonkeruulaitteen ohjelmiston.

Tutkimuksen tulokset osoittavat, että testivetoisen kehityksen hyödyntäminen sulautettujen ohjelmistojen kehityksessä on sekä mahdollista että hyödyllistä. Menetelmän käytöllä saavutettavat hyödyt liittyvät esimerkiksi koodin ulkoiseen laatuun, koodin sisäiseen laatuun, testaukseen, tuottavuuteen, dokumentointiin sekä psykologisiin ja sosiaalisiin vaikutuksiin. Sulautettuja ohjelmistoja kehitettäessä etuja ovat myös esimerkiksi laitteistoon liittyvien riskien ja kulujen vähentyminen ja koodin parempi siirrettävyys. Menetelmän käyttäminen ei ole silti ongelmatonta, sillä erityisesti sulautettujen ohjelmistojen erityispiirteet tuovat mukanaan omat haasteensa. Suurimpana ongelmana on kuitenkin menetelmän vaativuus, sillä menetelmän hyödyntäminen tehokkaasti vaatii kehittäjältä paljon harjoittelua.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Automation Technology
YLI-ROHDAINEN, KIMMO: Test-Driven Development of an Embedded Data Acquisition Software
Master of Science Thesis, 60 pages, 0 Appendix pages
October 2012
Major: Automation Software Engineering
Examiner: Professor Seppo Kuikka
Keywords: TDD, test-driven development, continuous integration, embedded systems, data acquisition software

Test-driven development is a software development methodology in which a programmer always writes an automatic unit test before implementing the actual functionality. Unit tests are being created based on the system requirements and these tests guide the development of the software. Even though test-driven development has gained popularity among the desktop and web software developers, it has not been widely applied to embedded system development. The goal of this study is to investigate how test-driven development can be utilized in the development of embedded systems.

The study is divided into two phases: In the first phase literature research is used to introduce terminology, theories and methodologies related to test-driven development and to examine previous experiences and studies concerning the use of the methodology. In the second phase the utilization of test-driven development in practice is investigated by means of a case study whose experiences are compared with the results of the first phase. The case study observes the use of the methodology in a software development project in which Bitwise implements the software for an embedded welding data acquisition device for Kemppi.

The results of this study indicate that the utilization of test-driven development in the development of embedded systems is viable and beneficial. The benefits that can be obtained are related to, for example, external code quality, internal code quality, testing, productivity, documentation, and psychological and social effects. When developing embedded software the benefits also include, for instance, reduced hardware related risks and expenses, and better software portability. However, the use of test-driven development is not without problems, as particularly the characteristics of embedded software bring their own challenges. The biggest problem with the methodology is the level of difficulty, as much practice is needed to be effective with it.

ALKUSANAT

Olen tehnyt tämän diplomityön toimiessani ohjelmistosuunnittelijana Bitwise Oy:n palveluksessa. Idea työhön sai alkunsa kiinnostuksesta testivetoiseen kehitykseen ja halusta soveltaa sitä myös sulautettujen sovellusten kehittämiseen. Työssä tutkin testivetoisen kehityksen soveltamista projektissa, jossa toteutettiin Kemppi Oy:lle sulautetun tiedonkeruulaitteen ohjelmisto.

Haluan kiittää sekä Bitwise Oy:tä että Kemppi Oy:tä mahdollisuudesta tämän diplomityön tekemiseen. Haluan erityisesti kiittää työn ohjauksesta Tampereen teknillisen yliopiston professori Seppo Kuikkaa ja Bitwise Oy:n Pauli Ahoa. Erityiskiitokset Kemppi Oy:n Mikko Mönkköselle ja Joni Hiljaselle hyvin sujuneesta yhteistyöstä. Haluan myös kiittää Bitwise Oy:n Tomi Mikkosta mahdollisuudesta työn tekemiseen työaikana.

Tampereella 12.9.2012

Kimmo Yli-Rohdainen

SISÄLLYS

TERMIT JA LYHENTEET

ANSI	American National Standards Institute.
ATDD	Acceptance Test -Driven Development. Hyväksymistestivetoinen kehitys.
BDD	Behaviour-Driven Development. Testivetoiseen kehitykseen perustuva ohjelmistokehitysmenetelmä.
CAN	Controller Area Network. Alunperin ajoneuvoihin suunniteltu väylä.
FAT	File Allocation Table. Microsoftin kehittämä tiedostojärjestelmä.
IEEE	Institute of Electrical and Electronics Engineers.
JTAG	Joint Test Action Group. Yleinen nimitys IEEE:n standardille 1149.1, joka määrittelee liitynnän laitteiston ja ohjelmiston testaamiseksi.
Refaktorointi	Koodin rakenteen ja tyylin parantaminen muuttamatta toiminnallisuutta.
Ristikäännös	Ohjelma käännetään eri alustalla kuin se suoritetaan.
TDD	Test-Driven Development. Testivetoinen kehitys.

1. JOHDANTO

Testivetoinen kehitys (engl. Test-Driven Development, TDD) on suosittu ohjelmistokehityksessä käytettävä menetelmä. Menetelmää käytettäessä ohjelmoija kirjoittaa aina automaattisen yksikkötestin ennen varsinaisen toiminnallisuuden toteuttamista. TDD ei ole kuitenkaan varsinainen testausmenetelmä, vaan kehitysmenetelmä. Yksikkötestejä luodaan järjestelmälle määriteltyjen vaatimusten perusteella ja nämä testit ohjaavat järjestelmän kehitystä. Vaikka TDD on saavuttanut suosiota työpöytä- ja web-sovellusten kehityksessä, ei sitä ole kuitenkaan sovellettu laajasti sulautettujen järjestelmien ohjelmistojen kehityksessä [20]. Tässä työssä tutkitaan TDD:n soveltamista kehitettäessä sulautettujen järjestelmien ohjelmistoja. TDD:n käyttö on erityisen hyödyllistä sulautettuja järjestelmiä kehitettäessä, sillä se mahdollistaa ohjelmiston kehityksen hyvin pitkälle laitteiston kehityksen rinnalla ja helpottaa ohjelmiston ja laitteiston integrointia.

Työssä tutkitaan TDD:n käyttöä ohjelmistokehitysprojektissa, jossa toteutetaan sulautetun tiedonkeruulaitteen ohjelmisto. Ohjelmiston toteuttaa Bitwise Oy, joka on vuonna 2003 perustettu vaativiin ohjelmistoratkaisuihin keskittyvä suomalainen ohjelmistoyritys. Projektin asiakkaana toimii Kemppi Oy, joka on vuonna 1949 perustettu suomalainen kansainvälisesti tunnettu hitsauslaitteiden ja -ratkaisujen tarjoaja.

Tutkimuksen tavoitteet kuvataan kohdassa 1.1. Kohdassa 1.2 asetetaan tutkimusongelma ja jaetaan se pienempiin tutkimuskysymyksiin. Tutkimuksen laajuutta rajataan kohdassa 1.3. Tutkimusmenetelmät ja -vaiheet esitellään kohdassa 1.4. Lopuksi kuvataan työn rakenne kohdassa 1.5.

1.1 Työn tavoitteet

Tämän diplomityön päätavoitteena on laajentaa osaamista ja parantaa sulautettujen sovellusten ohjelmistokehitysprosessia. Tähän pyritään tutkimalla TDD:n soveltamista sulautettujen sovellusten ohjelmistokehitykseen ja keräämällä kokemuksia aiheesta.

1.2 Tutkimusongelma

Tämä diplomityö pyrkii antamaan vastauksen seuraavaan tutkimusongelmaan: miten TDD:n käyttöönotolla voidaan parantaa sulautettujen sovellusten ohjelmistoke-

hitysprosessia.

Tutkimusongelman laajuuden takia jaetaan se useaan pienempään tutkimuskysymykseen. Jotta tutkimusongelmaan voidaan löytää ratkaisu, pitää siis vastata seuraaviin tutkimuskysymyksiin:

1. Mitä ongelmia on TDD:n käyttämisessä sulautettujen sovellusten kehittämisessä?
2. Mitä hyötyjä saadaan TDD:n käyttämisestä sulautettujen sovellusten kehittämisessä?
3. Mitä TDD:n käyttäminen vaatii projektin henkilöstöltä?
4. Mitä TDD:n käyttäminen vaatii sovelluksen arkkitehtuurilta ja suunnitteluperiaatteilta?
5. Minkälaisia työkaluja TDD:n käyttäminen vaatii?
6. Mitkä ovat parhaat käytännöt ja menetelmät käytettäessä TDD:tä?

Näihin tutkimuskysymyksiin palataan työn lopussa ja arvioidaan, miten hyvin niihin on löytynyt vastauksia.

1.3 Tutkimuksen rajaus

Tässä diplomityössä käsitellään TDD:tä myös yleisellä tasolla, mutta keskittyen sulautettujen järjestelmien ohjelmistojen kehittämiseen. Sulautetuilla järjestelmillä tarkoitetaan tässä työssä tiettyyn tarkoitukseen tehtyjä laitteistoja ja niitä ohjaavia ohjelmistoja. Usein laitteiston käyttäjän ei tarvitse olla edes tietoinen laitteiston sisältämästä ohjelmistosta. Tässä työssä tarkastellaan erityisesti C- ja C++-ohjelmointikielillä toteutettuja sulautettuja ohjelmistoja, koska suurin osa sulautetusta ohjelmistokehityksestä tehdään näillä kielillä [20]. Työn soveltavassa vaiheessa tutkitaan TDD:n käyttöä sulautetun tiedonkeruulaitteen ohjelmiston kehittämisessä.

1.4 Tutkimusmenetelmät ja -vaiheet

Tutkimus jakautuu kahteen vaiheeseen. Ensimmäisessä vaiheessa tutustutaan aihepiiriin kirjallisuustutkimuksen avulla. Tarkoituksena on esitellä aiheeseen liittyvät käsitteet, teoriat ja menetelmät. Lisäksi tutkitaan muiden kokemuksia TDD:n käytöstä.

Tutkimuksen toinen vaihe toteutetaan tapaustutkimuksen muodossa. Tässä solveltavassa vaiheessa otetaan TDD käyttöön projektissa, jossa toteutetaan sulautetun tiedonkeruulaitteen ohjelmisto. Soveltavan vaiheen tarkoituksena on arvioida

TDD:n soveltuvuutta sulautettuun ohjelmistokehitysprojektiin ja verrata kokemuksia ensimmäisessä vaiheessa tehtyihin havaintoihin.

1.5 Työn rakenne

Luku 2 esittelee diplomityön aiheeseen liittyvät käsitteet, teoriat ja menetelmät. Lisäksi luvussa perehdytään aiempiin tutkimuksiin ja kokemuksiin aiheesta.

Luku 3 kuvaa varsinaista TDD:n käyttöönottoa ja käyttöä tutkimuksen soveltavassa vaiheessa. Luvussa esitellään tapaustutkimuksessa tutkittu projekti ja toteutettu ohjelmisto.

Luku 4 esittelee soveltavassa osassa käytetyt työkalut. Lisäksi luvussa arvioidaan työkalujen käyttökelpoisuutta.

Luku 5 kuvaa kokemuksia TDD:n käytöstä projektissa ja arvioi TDD:n hyötyjä ja siihen liittyviä ongelmia. Tapaustutkimuksen tuloksia verrataan kirjallisuustutkimuksen tuloksiin.

Luku 6 esittelee ja arvioi tutkimuksen tuloksia. Lisäksi luvussa annetaan jatkotutkimusehdotuksia.

2. TEOREETTINEN TAUSTA

Tässä luvussa tutustutaan työhön liittyviin keskeisiin käsitteisiin, teorioihin ja menetelmiin. Lähteenä on käytetty alan kirjallisuutta ja julkaisuja. Kohdassa 2.1 esitellään ensin yksikkötestaukseen liittyviä käsitteitä ja menetelmiä. Seuraavaksi käsitellään testivetoista kehitystä ensin kohdassa 2.2 yleisellä tasolla ja kohdassa 2.3 erityisesti sulautettujen sovellusten kehittämisen kannalta. Lopuksi esitellään vielä testivetoiseen kehitykseen läheisesti liittyvä jatkuvan integroinnin menetelmä kohdassa 2.4.

2.1 Yksikkötestaus

Testivetoiseen kehitykseen liittyy olennaisena osana yksikkötestaus. TDD:n tapauksessa kyseessä on erityisesti automaattisten yksikkötestien kirjoittaminen ja suorittaminen. Jotta voitaisiin käsitellä TDD:tä, esitellään seuraavaksi muutamia olennaisia yksikkötestaukseen liittyviä käsitteitä.

2.1.1 Yksikkötesti

Yksikkötesti on pieni ja nopeasti suoritettava testi, joka testaa yhtä ohjelmiston yksikköä. Testattava yksikkö voi olla esimerkiksi funktio, moduuli tai luokka. Testi suoritetaan siten, että testattava yksikkö on eristetty normaalista ympäristöstään. Yksikkötestillä voidaan siis varmistaa testattavan yksikön oikea toiminta. Roy Osherove on määritellyt yksikkötestin seuraavasti:

> ”A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.” [36]

Teknisen määritelmän lisäksi on tärkeää kuvata hyvän yksikkötestin ominaisuudet. Huonoista yksikkötesteistä on enemmän haittaa kuin hyötyä, sillä ne hankaloittavat sovelluksen ylläpitoa ja niiden kirjoittamiseen tuhlaantuu aikaa. Osherove onkin kuvannut hyvän yksikkötestin ominaisuuksia. Osheroven mukaan hyvä yksikkötesti

- on täysin automatisoitavissa
- on toistettavissa

- on helppo kirjoittaa
- on käyttökelpoinen tulevaisuudessakin
- on kenen tahansa ajettavissa
- on ajettavissa napin painalluksella
- on nopea
- on helposti luettava
- on helposti ylläpidettävä
- on luotettava
- voidaan ajaa missä tahansa järjestyksessä osana testijoukkoa
- voidaan ajaa muistissa (ei esimerkiksi tietokantojen tai tiedostojen käsittelyä)
- johdonmukaisesti palauttaa aina saman tuloksen (ei satunnaisuutta)
- testaa vain yhtä loogista konseptia (eli esimerkiksi yhtä toiminnallisuutta tai virhetilannetta). [35, 36]

Jotta yksikkötestin lukijan olisi helppo ymmärtää testin tarkoitus, suosittelee Gerard Meszaros jakamaan testitapauksen neljään selvästi erottuvaan vaiheeseen (Four-Phase Test). Testiä ajettaessa nämä vaiheet suoritetaan järjestyksessä peräkkäin. Ensimmäisessä vaiheessa alustetaan testattava komponentti ja testausympäristö testitapauksen vaatimaan alkutilaan. Toisessa vaiheessa testattavaa komponenttia käytetään. Kolmannessa vaiheessa tarkistetaan, että lopputulos on haluttu. Neljännessä eli viimeisessä vaiheessa testiympäristö palautetaan ennen testitapauksen suorittamista olleeseen tilaan. [34]

2.1.2 Yksikkötestikehys

Jotta edellisen kohdan mukaisia hyviä yksikkötestejä voidaan kirjoittaa, tarvitaan jokin tapa ilmaista miten testattavan koodin pitäisi käyttäytyä. Tätä tarkoitusta varten on kehitetty lukuisia yksikkötestikehyksiä. James Grenningin mukaan yksikkötestikehyksen tarkoituksena on tarjota

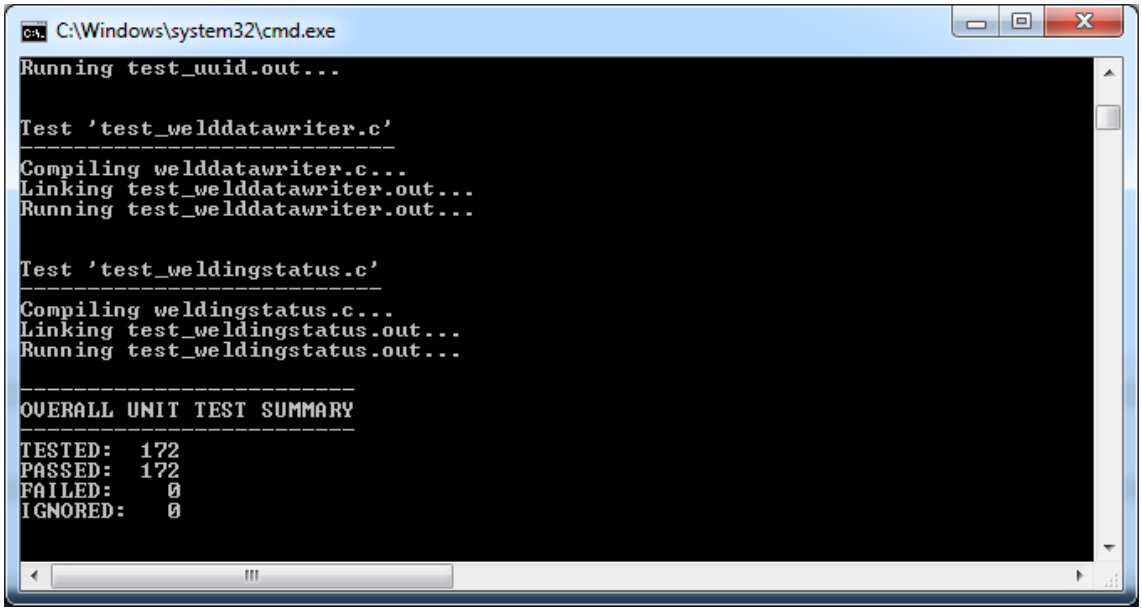
- yhteinen kieli testitapausten kuvaamiseksi
- yhteinen kieli odotettujen tulosten kuvaamiseksi
- pääsy testattavassa koodissa käytetyn ohjelmointikielen ominaisuuksiin

- paikka, jonne kerätä kaikki projektin, järjestelmän tai alijärjestelmän testitapaukset
- mekanismi testitapausten ajamiseen
- tiivis raportti testien onnistumisesta tai epäonnistumisesta
- tarkat raportit kaikista epäonnistuneista testeistä. [21]

Yksikkötestikehyksiä on kehitetty monille eri kielille ja alustoille. Monet näistä yksikkötestikehyksistä kuuluvat xUnit-perheeseen. XUnit-nimitystä käytetään yksikkötestikehyksistä, jotka on mallinnettu JUnit- tai SUnit-yksikkötestikehyksen mukaan. Kaikki xUnit-perheen jäsenet tarjoavat tietyt perusominaisuudet. Niillä voidaan kuvata testitapaus testifunktiona, kuvata testitapauksen odotetut tulokset kutsuilla Assert-funktiohin, koota testitapauksia testijoukoiksi ja ajaa yksi tai useampi testitapaus ja saada raportti tuloksista. [34]

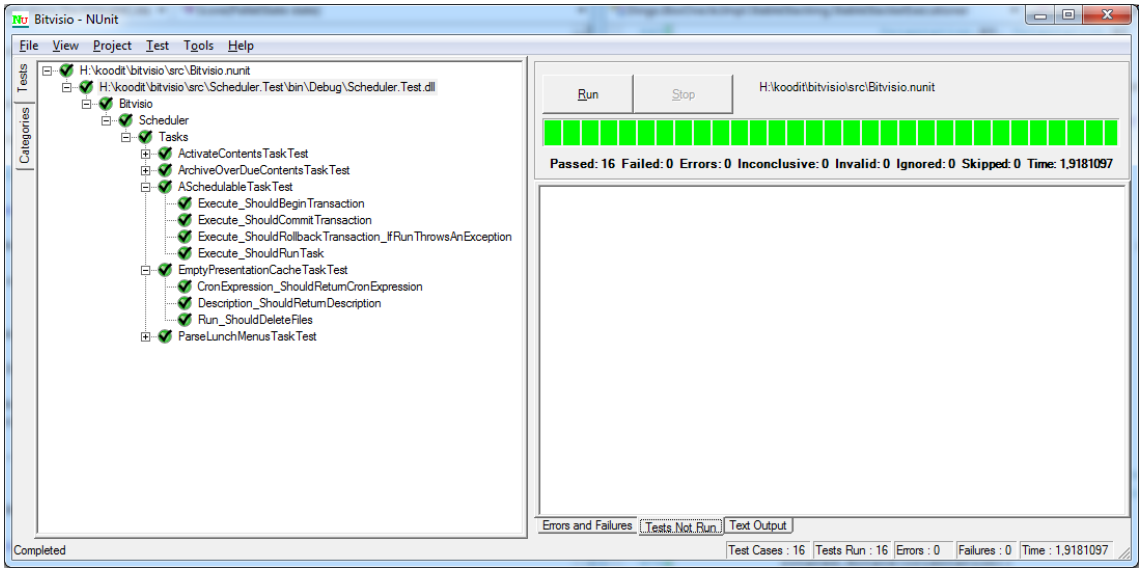
2.1.3 Testiajuri

Testiajuri on ohjelma tai ohjelman osa, joka ajaa testitapauksia. Edellisessä kohdassa kuvattuihin yksikkötestikehyksiin kuuluu yleensä yleiskäyttöinen testiajuri, jolla voidaan ajaa valitut testitapaukset tai joukot. Tällöin jokaiselle testitapaukselle tai testijoukolle ei siis tarvitse kirjoittaa omaa testiajuria. [21, 34]



Kuva 2.1: Unity-yksikkötestikehyksen komentorivipohjainen testiajuri.

Testiajureita on erityyppisiä eri tarkoituksiin. Esimerkiksi komentoriviltä ajettava testiajuri on hyödyllinen, jos testejä halutaan ajaa skriptillä. Kuvassa 2.1 on



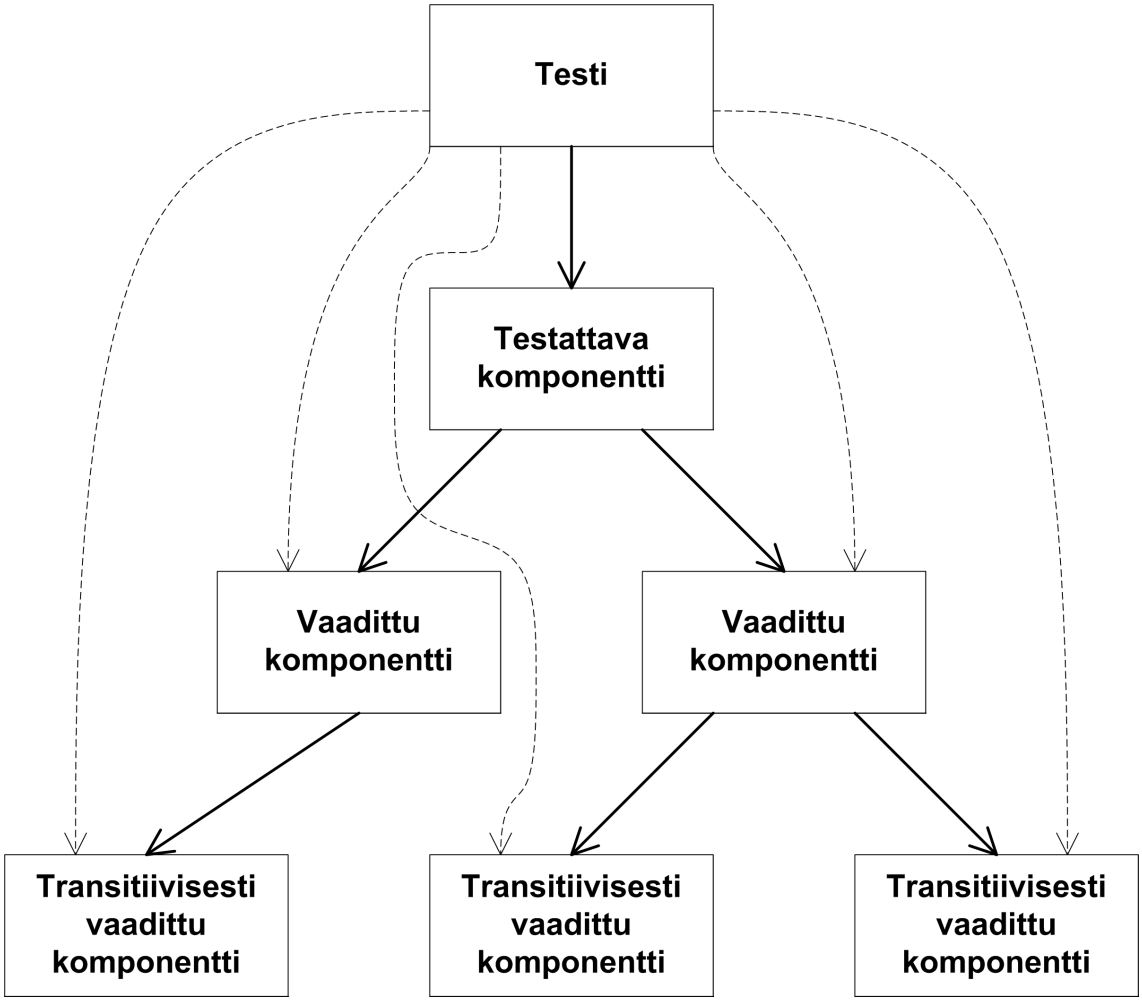
Kuva 2.2: NUnit-yksikkötestikehyksen graafinen testiajuri.

Unity-yksikkötestikehyksen komentorivipohjainen testiajuri. Kuvassa näkyy selkeästi suoritettujen, onnistuneiden ja epäonnistuneiden testien lukumäärät. Graafisella testiajurilla voidaan taas visualisoida testien suorituksen edistymistä ja testien tuloksia. Tyypillisesti onnistuneet testitapaukset kuvataan vihreällä värillä ja epäonnistuneet punaisella. [34] Kuvassa 2.2 on esitetty xUnit-perheeseen kuuluvan NUnit-yksikkötestikehyksen graafinen testiajuri. Kuvan vasemmassa laidassa näkyvät testijoukot ja testitapaukset puumaisena listana. Kuvan oikeassa laidassa näkyvät edistymispalkki ja testien tulokset.

2.1.4 Testisijaiset

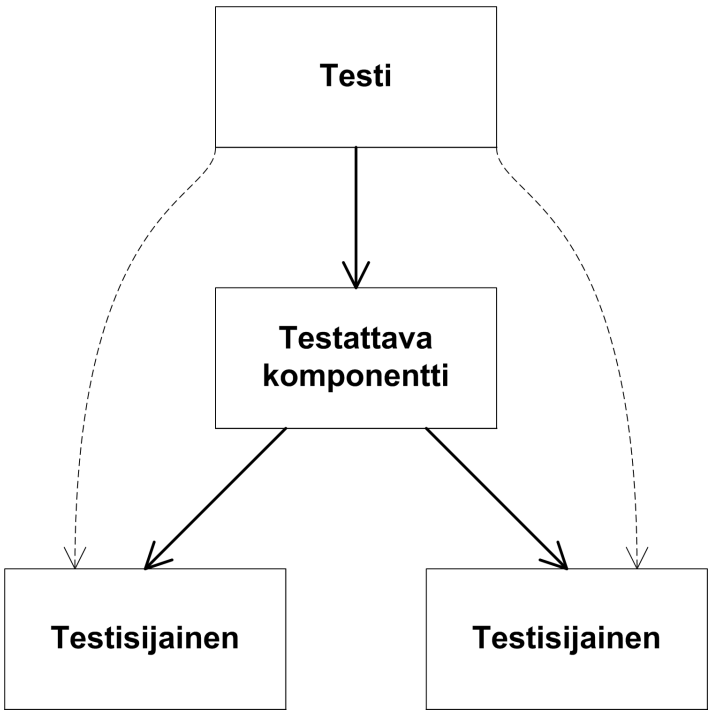
Usein kohdan 2.1.1 mukaisia hyviä yksikkötestejä on vaikea kirjoittaa, koska testattava komponentti riippuu muista komponenteista, joita on hankalaa tai mahdotonta käyttää testiympäristössä. Nämä vaaditut komponentit (engl. Depended-On Component, DOC) saattavat olla vielä toteuttamatta, toimia liian hitaasti yksikkötesteissä käytettäväksi, sisältää haitallisia sivuvaikutuksia tai olla hankalia saada käyttäytymään testitapausten vaatimilla tavoilla. Tällaisissa tilanteissa vaadittu komponentti voidaan testissä korvata testisijaisella (engl. Test Double). Termi testisijainen tulee elokuvateollisuudesta, jossa vaarallisissa kohtauksissa esiintyy päänäyttelijän sijaan sijaisnäyttelijä (engl. stunt double). Testisijainen tarjoaa testattavalle yksikölle saman rajapinnan kuin oikeakin komponentti, mutta sen ei tarvitse käyttäytyä samalla tavalla. Testisijaisen käyttäytyminen voi olla kovakoodattua tai se voidaan konfiguroida testitapauksen alustusvaiheessa. Testisijainen tarjoaa testattavalle komponentille epäsuoria syötteitä paluuarvojen muodossa. Sitä voidaan käyttää myös tarkistamaan testattavan komponentin epäsuoria tuloksia tutkimalla

testisijaiselle annettuja syötteitä. [21, 34]



Kuva 2.3: Testin riippuvuudet.

Kuvassa 2.3 on esitetty erään komponentin testijärjestely ja siinä esiintyvät riippuvuudet, kun testisijaisia ei ole käytetty. Ehjät viivat kuvaavat suoria riippuvuuksia: testikoodi kutsuu testattavan komponentin koodia ja testattava komponentti kutsuu vaadittujen komponenttiensa koodia. Testikoodin pitää alustaa vaaditut komponentit haluttuun tilaan testitapauksen aluksi ja mahdollisesti myös tarkistaa niiden tila testitapauksen lopuksi. Tämän takia testikoodi riippuu testattavan komponentin lisäksi testattavan komponentin vaadituista komponenteista. Nämä riippuvuudet näkyvät kuvassa katkoviivoilla. Myös vaadituilla komponenteilla on riippuvuuksia muihin komponentteihin. James Grenning kutsuu näitä komponentteja transitiivisesti vaadituiksi komponenteiksi. Riippuvuusverkko voi huonossa tapauksessa sisältää lähes kaikki ohjelmiston komponentit. Nämä kaikki riippuvuudet tekevät testeistä hyvin monimutkaisia ja hankalia ylläpitää. Onkin selvää, että nämä riippuvuusketjut pitää pyrkiä katkaisemaan. Tämä voidaan tehdä käyttämällä testisijaisia testattavan komponentin vaadittujen komponenttien tilalla. Tällöin ei



Kuva 2.4: Testin riippuvuuksien hallinta testisijaisilla.

tarvita lainkaan transitiivisesti vaadittuja komponentteja, joten riippuvuusverkosta ja siten myös testeistä tulee paljon yksinkertaisempia. Kuvassa 2.4 on esitetty testijärjestelyn riippuvuudet, kun vaaditut komponentit on korvattu testisijaisilla. [21]

Kaikkia vaadittuja komponentteja ei kuitenkaan tarvitse korvata testisijaisilla. Grenningin mukaan komponentin oikeaa toteutusta kannattaa käyttää aina, jos se on mahdollista. Testisijaisia tulisi käyttää siis vain tarvittaessa. Esimerkiksi linkitettyä listaa ei kannata korvata sijaisella. Grenning antaa muutamia yleisiä syitä testisijaisen käyttämiseen. Näitä ovat laitteistoriippuvuudet, hankalasti tuotettavat syötteet, hitaat vaaditut komponentit, riippuvuus jostain muuttuvasta (esimerkiksi kellonaika), keskeneräiset vaaditut komponentit ja vaikeasti konfiguroitavat riippuvuudet. Usein vain osa testattavan komponentin vaadituista komponenteista korvataan testisijaisilla. Vaadittu komponentti voidaan myös korvata testisijaisella vain joissain testitapauksissa ja käyttää oikeaa toteutusta muissa. [21]

Testisijaisia on monia erityyppisiä erilaisiin käyttötarkoituksiin. Näiden erilaisten variaatioiden nimitykset eivät ole kovin vakiintuneita, vaan niitä käytetään joskus ristiin ja päällekkäin. Myöskin vakiintuneita suomennoksia näille termeille on vain hyvin vähän. Tässä työssä käytetään Gerard Meszarosin määrittelemiä termejä ja niiden suomennoksia. Meszaros on luokitellut ja nimennyt testisijaiset sen mukaan miten tai miksi niitä käytetään. Tynkää (Test Stub) käytetään korvaamaan oikea komponentti, jotta testattavalle komponentille saadaan annettua halutut epäsuorat

syötteet. Nämä epäsuorat syötteet annetaan yleensä tyngän testattavalle komponentille palauttamina paluuarvoina. Näin testattava komponentti saadaan pakotettua sellaisille suorituspoluille, joita se ei muuten suorittaisi. Testivakooja (Test Spy) on kehittyneempi versio tyngästä. Testivakooja voi tyngän toimintojen lisäksi tallentaa sille annettuja syötteitä eli testattavan komponentin epäsuoria tuloksia, joita voidaan verifioida testissä. [16, 34]

Korvikeoliota (Mock Object) käytetään tarkistamaan, että testattava komponentti käyttää vaaditun komponentin rajapintaa oikein. Korvikeolion avulla voidaan esimerkiksi tarkistaa, että testattava komponentti kutsuu rajapinnan funktioita oikeilla parametreilla, oikeaan aikaan ja oikeassa järjestyksessä. Korvikeolioilla voidaan siis tarkistaa testattavan komponentin käyttäytymistä sen sijaan, että tarkistettaisiin testattavan komponentin tilaa testin lopuksi. Väärennös (Fake Object) on luonteeltaan hieman erilainen, sillä testi ei ohjaa eikä tarkkaile sitä. Väärennös toteuttaa yleensä saman toiminnallisuuden kuin oikeakin toteutus tai ainakin osan siitä, mutta yksinkertaisemmalla tavalla. Väärennöstä voidaan käyttää, jos oikea toteutus ei ole vielä valmiina, on liian hidas tai ei ole saatavilla testiympäristössä. [16, 34]

Testisijaisia voidaan toteuttaa kahdella tavalla. Ensimmäinen tapa on kirjoittaa käsin tarvittava testisijainen. Toinen tapa on käyttää jotain ohjelmistokehystä tai työkalua, jolla tarvittava testisijainen voidaan generoida. Jos testisijainen generoidaan, voidaan sijaisen generointi tehdä dynaamisesti tai staattisesti. Dynaaminen generointi tapahtuu ajonaikaisesti, kun taas staattinen generointi tehdään ennen koodin kääntämistä. Joitakin testisijaisia pitää myös konfiguroida, sillä niiden pitää esimerkiksi palauttaa tiettyjä paluuarvoja tai heittää tiettyjä poikkeuksia. Käsin kirjoitetuille sijaisille voidaan paluuarvot kovakoodata suoraan testisijaisen koodiin. Jotta samaa testisijaista voitaisiin käyttää useassa testissä, saattaa olla hyödyllistä tehdä sijaisesta konfiguroitava. Konfiguroitavalle sijaiselle voidaan ajonaikana asettaa halutut paluuarvot esimerkiksi konfigurointirajapinnan kautta. Generoidut sijaiset ovat käytännössä aina konfiguroitavia. [34]

2.2 TDD

TDD:n tavoitteena on ”Clean code that works”, kuten Ron Jeffries on sanonut. TDD:ssä tuohon tavoitteeseen pyritään ohjaamalla ohjelmiston kehitystä automaattisilla testeillä. TDD on siis kehitysmenetelmä eikä testausmenetelmä. Menetelmän idea perustuu kahteen yksinkertaiseen sääntöön:

1. Kirjoita uutta koodia vain, jos automaattinen testi on epäonnistunut.

2. Poista toisteinen koodi (duplicated code). [4]

Nämä säännöt vaikuttavat yksinkertaisilta, mutta niillä on Kent Beckin mukaan suuret vaikutukset ohjelmistojen kehittämiseen. Ensinnäkin ohjelmistoa pitää suunnitella orgaanisesti toimivan koodin tarjotessa palautetta suunnittelupäätöksistä. Freeman ja Pryce puhuvatkin ohjelmistojen kasvattamisesta kirjassaan "Growing Object-Oriented Software, Guided by Tests" [17]. Kehittäjän pitää myös kirjoittaa testit itse, sillä hän ei voi jatkuvasti odottaa, että joku muu saisi uuden testin kirjoitettua. Kehitysympäristön pitää tarjota välitön palaute pieniinkin koodimuutoksiin. Kehittäjän pitää pystyä siis ajamaan testit helposti ja nopeasti. Ohjelmiston pitää koostua monista toiminnallisuudeltaan yhtenäisistä (engl. high cohesion) ja vähän riippuvuuksia sisältävistä (engl. low coupling) komponenteista, jotta testaaminen olisi helppoa. [4]

2.2.1 Kehityssykli

Edellisessä kohdassa esitetyistä säännöistä johtuen TDD on luonteeltaan iteratiivinen ja inkrementaalinen menetelmä. Ohjelmiston kehitys etenee pienissä ja lyhyissä iteraatioissa, joiden kesto on yleensä vain joitain minuutteja. Tälläista iteraatioita kutsutaan usein TDD:n kehityssykliksi tai mikrosykliksi ja koostuu Beckin mukaan seuraavista vaiheista:

1. Lisää uusi testi.
2. Aja kaikki testit ja varmista, että uusi testi ei mene läpi.
3. Tee pieni muutos.
4. Aja kaikki testit ja varmista, että ne menevät läpi.
5. Refaktoroi poistaaksesi toisteisen koodin. [4]

Syklin ensimmäisessä vaiheessa kehittäjä joutuu miettimään miten uuden ominaisuuden pitäisi toimia, jotta osaisi kirjoittaa ominaisuudelle uuden testin. Kehittäjä joutuu myös miettimään minkälaisen rajapinnan kautta ominaisuutta tulisi käyttää. Kehittäjä joutuu siis miettimään uuden ominaisuuden vaatimuksia ennen kuin on kirjoittanut riviäkään toteutusta. Uutta ominaisuutta tulee myös mietittyä ominaisuuden käyttäjän kannalta, eikä toteutuksen kannalta. Tässä vaiheessa tärkeintä on tehdä testistä helposti luettava. [4, 17]

Syklin toisessa vaiheessa ajetaan kaikki testit, jolloin uuden testin pitäisi epäonnistua. Tämä on tärkeä vaihe, koska sillä varmistetaan testikoodin toimivuus. Jos tätä vaihetta ei suoriteta, saattaa uusi testi mennä aina virheen takia läpi, vaikkei testiä vastaavaa toteutusta ole vielä edes kirjoitettu. Epäonnistuneen testin pitäisi myös ilmaista selkeästi epäonnistumisen syy. Tällä varmistetaan, että testi testaa

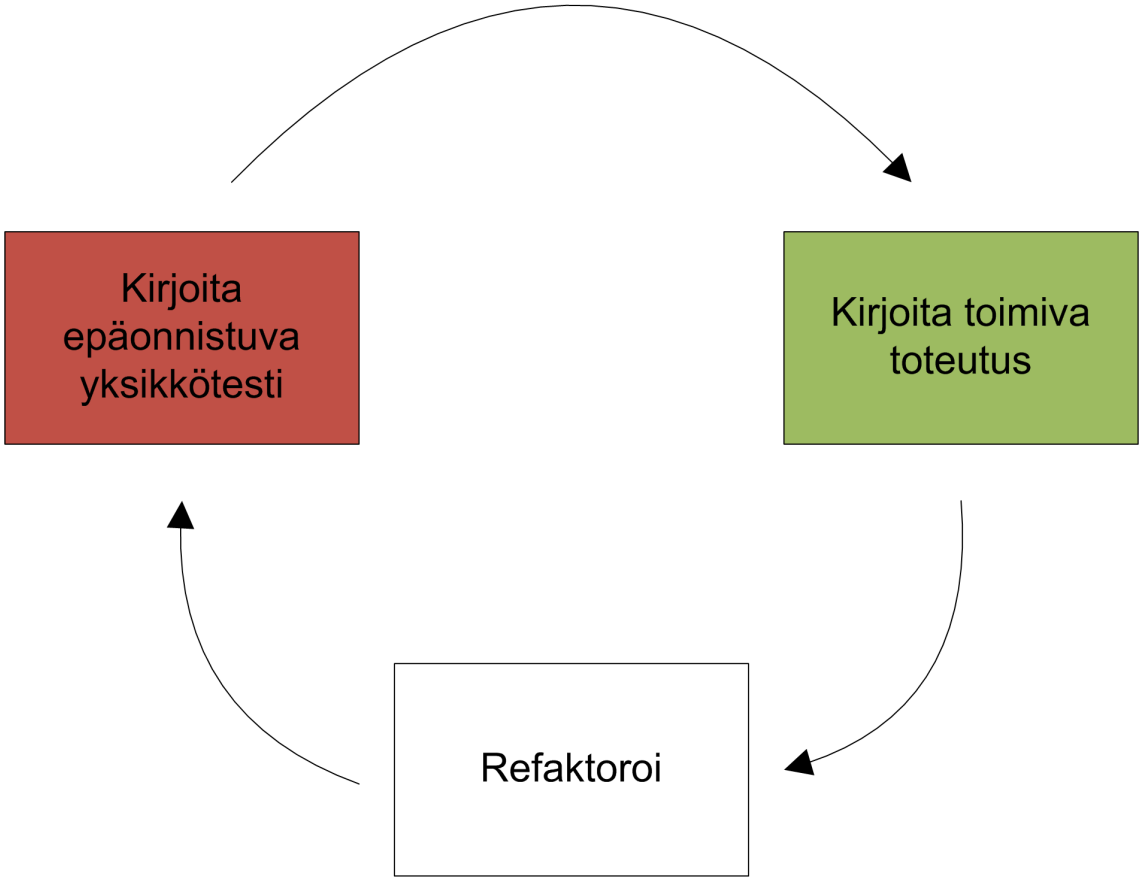
oikeaa asiaa. Toisaalta kuvaavasta virheilmoituksesta on hyötyä tulevaisuudessa, jos testi epäonnistuu joskus myöhemmin. Jos testi ei antaisi selkeää kuvausta epäonnistumisen syystä, olisi virheen jäljittäminen hyvin hankalaa. Helposti luettava testi ja kuvaavat virheilmoitukset helpottavat testin ja toteutuksen tarkoituksen ymmärtämistä ja ovat siten erittäin tärkeitä testikoodin ylläpidettävyyden kannalta. [4, 17]

Syklin kolmannessa vaiheessa tehdään pieni muutos, jolla saadaan uusi testi läpäistyä. Oleellista on saada testi läpäistyä nopeasti. Millään muulla ei ole tässä vaiheessa merkitystä. Koodin siisteydellä, luettavuudella, laajennettavuudella tai robustisuudella ei ole väliä. Jos kehittäjä pystyy helposti ja nopeasti kirjoittamaan hyvän ja siistin toteutuksen, kannattaa se tietysti tehdä. Mutta jos ongelma on hankalampi, kehittäjän tulee kirjoittaa mahdollisimman yksinkertainen toteutus. Tämä voi tarkoittaa esimerkiksi sitä, että funktion paluuarvoksi kovakoodataan testissä vaadittu arvo. Tässä vaiheessa tärkeintä on siis vain saada koodi toimimaan (”that works”). Neljännessä vaiheessa ajetaan kaikki testit ja varmistetaan, että ne menevät läpi. Näin varmistetaan, että uusi ominaisuus toimii eikä vanhojen ominaisuuksien käyttäytyminen ole muuttunut. [4]

Syklin viidennessä ja viimeisessä vaiheessa koodia refaktoroidaan, jotta saadaan toisteinen koodi poistettua. Koodin refaktorointi tarkoittaa koodin sisäisen rakenteen muuttamista siten, ettei koodin ulkoinen käyttäytyminen muutu. Refaktoroinnin tarkoituksena on parantaa koodin laatua tehden siitä helpommin luettavaa ja ylläpidettävää. Refaktorointi suoritetaan tekemällä monia pieniä muutoksia, esimerkiksi jakamalla koodia pienempiin funktioihin, poistamalla toisteisuutta tai nimeämällä funtioita ja muuttujia paremmin. Tässä vaiheessa on tärkeätä huomata, että myös testikoodista pitää poistaa toisteisuus. Testikoodinkin pitää olla helposti ymmärrettävää ja ylläpidettävää, jotta siitä olisi hyötyä myös tulevaisuudessa. Jokaisen pienen muutoksen jälkeen testit ajetaan uudestaan, millä varmistetaan, ettei koodin ulkoinen käyttäytyminen ole muuttunut. Tämän vaiheen tarkoituksena on siis tehdä toimivasta koodista myös siistiä koodia (”clean code”). [15, 17]

Tätä sykliä toistetaan, kunnes halutut ominaisuudet on toteutettu. Sykliä voidaan lyhyesti kuvata sanoin ”Red-Green-Refactor”. Eli ensin kirjoitetaan epäonnistuva yksikkötesti, sitten kirjoitetaan toimiva toteutus ja viimeiseksi refaktoroidaan. Tämä on esitetty kuvassa 2.5. Yhdessä syklissä tehtävien muutosten määrä voi vaihdella kehittäjän ja ratkaistavan ongelman mukaan. Beckin mukaan kannattaa edetä hyvin pienin askelin, jos kehittäjä ei ole vielä kovin kokenut TDD:n käyttäjä tai jos ratkaistava ongelma on hankala. Jos taas kehittäjä on kokenut ja ratkaistava ongelma on helppo, voidaan edetä pidemmillä askeleilla. Etenemisnopeutta kannattaakin säätää jatkuvasti tilanteen mukaan. [4]

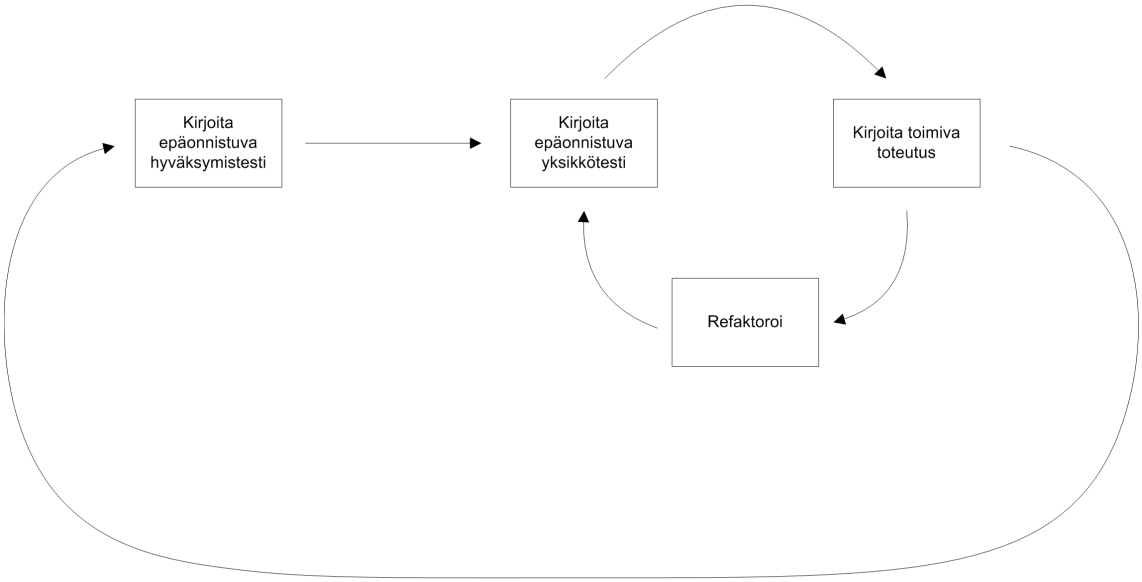
Tyypillisesti tätä sykliä käytettäessä kirjoitetaan vain yksikkötestejä. Tällöin vaarana on, että tuotettua hyvin testattua koodia ei kutsutakaan mistään tai sitä ei



Kuva 2.5: TDD-sykli, ”Red-Green-Refactor”.

pystytä integroimaan muuhun ohjelmistoon. Freeman ja Pryce esittävät ratkaisuksi TDD-prosessin aloittamista kirjoittamalla korkeamman tason hyväksymistestin uudelle ominaisuudelle. Epäonnistuessaan hyväksymistesti osoittaa, ettei ominaisuutta ole vielä toteutettu. Kun hyväksymistesti menee onnistuneesti läpi, on ominaisuus toteutettu. Hyväksymistestit pyritään kirjoittamaan siten, että ne testaavat ohjelmistoa päästä päähän (engl. End-to-End) kutsumatta suoraan ohjelmiston sisäistä koodia. Hyväksymistestin pitäisi siis käyttää vain ohjelmiston ulkoisia rajapintoja kuten käyttöliittymiä tai viestirajapintoja. Hyväksymistesti ohjaa kirjoittamaan vain uutta ominaisuutta varten oikeasti tarvittavaa koodia. Hyväksymistestin läpäisemiseksi sovelletaan edellä esitettyä sykliä yksikkötestitasolla. Tällöin TDD-prosessissa on kaksi sisäkkäistä sykliä. Ulompi sykli mittaa hyväksymistesteillä ohjelmiston kehityksen etenemistä ja osoittaa mitkä ominaisuudet ovat jo toteutettuna. Sisempi sykli ohjaa kehittäjää kirjoittamaan siistiä ja toimivaa koodia yksikkötestien avulla. Nämä syklit on esitetty kuvassa 2.6. Tälläista menetelmää kutsutaan usein hyväksymistestivetoiseksi kehitykseksi (engl. Acceptance Test -Driven Development,

ATDD). [17]



Kuva 2.6: Sisempi ja ulompi TDD-sykli.

2.2.2 Esitettyjä etuja

TDD:n käytön on väitetty tuovan hyvin monenlaisia etuja. Seuraavaksi esitellään näitä väitteitä, joita on kerätty alan lehtijulkaisuista ja kirjoista. Aiheesta on tehty pääasiassa kahdenlaisia tutkimuksia: akateemisia tutkimuksia ja teollisia tutkimuksia. Akateemisissa tutkimuksissa on yleensä vertailtu saman sovelluksen toteuttamista sekä TDD:llä että jollakin toisella menetelmällä. Tutkittavat henkilöt ovat useimmiten olleet opiskelijoita. Teolliset tutkimukset ovat enimmäkseen tapaustutkimuksia, joissa tutkitaan oikean tuotteen toteuttamista asiakkaalle. Taloudellisista syistä johtuen samaa tuotetta ei voida toteuttaa kahdella eri menetelmällä, joten vertailukohtina pitää käyttää muita samankaltaisia projekteja. Väitettyjen etujen voidaan karkeasti jaotella liittyvän koodin ulkoiseen laatuun, koodin sisäiseen laatuun, testaukseen, tuottavuuteen, dokumentointiin sekä psykologisiin ja sosiaalisiin vaikutuksiin.

Koodin ulkoinen laatu kuvaa kuinka hyvin sovellus täyttää asiakkaiden ja käyttäjien tarpeet eli kuinka toimiva, luotettava ja käytettävä sovellus on [17]. Ulkoista laatua mitataan useimmiten löydettyjen virheiden perusteella. Löydettyjen virheiden määrää voidaan tutkia esimerkiksi mustalaatikkotestauksella tai tutkimalla virheraportteja. TDD:n väitetään vähentävän virheiden määrää, koska suurin osa virheistä löydetään jo hyvin aikaisessa vaiheessa [4]. Aiheesta onkin tehty monia tutkimuksia. Vähentyneitä virhemääriä ovat tutkimuksissaan havainneet muun muassa

Maximilien ja Williams IBM:llä [33], Sanchez et al. IBM:llä [39], Bhat ja Nagappan Microsoftilla [5], Yenduri ja Perkins [46], Crispin [11], Martin [32] sekä George ja Williams [19]. Vähentyneet virhemäärät voivat johtua osittain siitä, että yksikkötestaaminen ei ollut kovinkaan kurinalaista tai kattavaa ennen TDD:n käyttöönottoa kuten Maximilienin ja Williamsin tapauksessa [33]. Lui ja Chanon huomasivat myös Kiinassa toteutetussa tutkimuksessaan TDD:n nopeuttavan käyttäjien raportoimien virheiden korjaamista [30]. TDD:tä käytettäessä suurin osa virheistä havaitaan jopa minuuttien sisällä niiden syntymisestä. Virheet ovat myös helposti paikannettavissa ja korjattavissa, koska epäonnistuva yksikkötesti osoittaa virheen. Lisäksi tarve työläälle ja aikaavievälle debuggaukselle vähenee huomattavasti tai poistuu lähes kokonaan. [19, 32]

Koodin sisäinen laatu kuvaa kuinka hyvin sovellus täyttää kehittäjien ja ylläpitäjien tarpeet eli kuinka helppo koodia on ymmärtää ja muuttaa. Sisäinen laatu siis mahdollistaa sovelluksen käyttäytymisen muuttamisen turvallisesti ja ennustettavasti, kun vaatimukset muuttuvat. [17] TDD:n on väitetty parantavan koodin sisäistä laatua usealla tavalla. TDD parantaa komponenttien yhtenäisyyttä eli koheesiota, koska sovellus pitää jakaa pieniin yhtenäisiin komponentteihin, jotta koodi olisi helpommin testattavaa. Komponenteilla pitää myös olla selkeät vastuualueet. [3, 4, 17, 32] Helpomman testattavuuden saavuttamiseksi TDD myös ohjaa vähentämään koodin sisäisiä riippuvuuksia [3, 4, 17, 32]. Komponenttien väliset riippuvuudet tulevat myös eksplisiittisesti esille, eivätkä jää piiloon [17]. Vaikkei TDD pakotakaan kirjoittamaan siistimpää koodia, se mahdollistaa koodin siistimisen turvallisesti. Kattavat yksikkötestit varmistavat, ettei sovelluksen toiminta muutu refaktoroinnin aikana. Ne myös mahdollistavat toiminnallisten muutosten tekemisen turvallisesti. Koodi on siis joustavampaa. [32] TDD:n väitetään pienentävän koodin kompleksisuutta. Koodin kompaktiutta voidaan käyttää yhtenä kompleksisuuden mittana, sillä pienempiä funktioita ja komponentteja on helpompi ymmärtää. Janzen ja Saiedian totesivat tutkimuksessaan, että TDD:tä käyttäneiden funktiot olivat keskimäärin lyhyempiä, komponentit sisälsivät vähemmän funktioita ja komponentit olivat pienempiä [25]. Koodin kompleksisuutta voidaan arvioida myös erilaisilla kompleksisuusmetriikoilla. Muutamat tutkimukset ovatkin todenneet TDD:n pienentävän koodin kompleksisuutta näillä metriikoilla mitattuna [25, 42].

Vaikkei TDD olekaan varsinainen testausmenetelmä, on sen silti väitetty parantavan testausta. Oleellinen osa TDD:tä on testien kirjoittaminen, joten on tietysti luonnollista olettaa, että niitä syntyy enemmän. Monissa tutkimuksissa onkin havaittu kirjoitettujen yksikkötestien määrän olevan suurempi käytettäessä TDD:tä. Tällaisia havaintoja ovat tutkimuksissaan tehneet esimerkiksi Janzen ja Saiedian [24], Yenduri ja Perkins [46] sekä Erdogmus et al. [13]. Koska TDD:tä käytettäessä uutta toteutuskoodia kirjoitetaan vain testin epäonnistuessa, voidaan testikat-

tavuuden olettaa olevan korkea. TDD:n onkin todettu parantavan testikattavuutta monissa tutkimuksissa, kun TDD:llä toteutettua koodia on verrattu perinteisin menetelmin toteutettuun koodiin käyttäen erilaisia testikattavuusmittoja. Tällaisia havaintoja ovat tehneet esimerkiksi Erdogmus [13] sekä Siniaalto ja Abrahamsson [41]. Myös esimerkiksi Martin [32], Bhat ja Nagappan [5], George ja Williams [19] ovat havainneet korkeita testikattavuuksia TDD:tä käytettäessä, vaikkei tuloksia olekaan verrattu perinteisin menetelmin toteutetun ratkaisun kanssa.

Pyrittäessä parantamaan ohjelmistokehitysprosessia on tärkeää huomioida laadun lisäksi tuottavuus. Tuottavuutta on tyypillisesti arvioitu mittaamalla projektiin käytettyä kokonaisaikaa, tuotettujen koodirivien määrää aikayksikköä kohti tai toteutettujen ominaisuuksien määrää [13]. TDD:n on havaittu parantavan ohjelmiston laatua, mutta laatu ei saisi syntyä tuottavuuden kustannuksella. Laadun parantuessa voidaan kuitenkin hyväksyä pieni tuottavuuden heikentyminen, sillä parempi laatu yleensä helpottaa ohjelmiston ylläpitoa. TDD:n vaikutuksista tuottavuuteen on tehty vaihtelevia ja ristiriitaisiakin havaintoja [26]. Erdogmus et al. havaitsivat TDD:n parantavan tuottavuutta, kun toteutettujen ominaisuuksien määrää mitattiin mustalaatikkotesteillä. He uskovat tuottavuusedun olevan seurausta paremmasta ongelman ymmärtämisestä, paremmasta keskittymisestä yhteen asiaan kerrallaan, nopeammasta oppimisesta ja virheiden korjauksen nopeutumisesta. He myös havaitsivat, että suurimman tuottavuushyödyn TDD:n avulla saivat osaavimmat henkilöt. [13] Samankaltaisia havaintoja ovat tehneet myös muun muassa Gupta ja Jalote [22], Yenduri ja Perkins [46] sekä Madeyski ja Szala [31]. Joissain tutkimuksissa taas TDD:n havaittiin pienentävän tuottavuutta, mutta vain vähän. Tällaisia havaintoja ovat tehneet esimerkiksi Maximilien ja Williams [33] sekä Bhat ja Nagappan [5]. TDD:n käytön on myös huomattu parantavan työmääräarvioiden tarkkuutta useissakin tutkimuksissa [7, 30].

TDD:ssä kirjoitetut testit toimivat myös eräänlaisena dokumentaationa, koodiesimerkkeinä. Jokainen testi kuvaa pienen osan koko ohjelmiston toiminnasta. Testit näyttävät esimerkiksi miten tietynlainen olio luodaan tai miten tiettyä funktiota tulee kutsua. Testit ovat dokumentaationa hyvin tarkkoja ja ne pidetään ajan tasalla. Ongelmana on kuitenkin se, että testien pitää olla helposti luettavia ja ymmärrettäviä, jotta niistä olisi hyötyä dokumentaatiotarkoituksiin. Tietysti samat edut voidaan saavuttaa siinäkin tapauksessa, että testit kirjoitetaan perinteisesti vasta toteutuksen jälkeen. TDD:tä käytettäessä testejä ja siten myös koodiesimerkkejä syntyy kuitenkin yleensä enemmän. [32]

TDD:n mahdollisesti suurimmat edut ovat luonteeltaan psykologisia ja sosiaalisia ja siten vaikeasti mitattavissa. Esimerkiksi Beckin mukaan TDD:n käyttäminen poistaa pelkoa, vähentää stressiä ja lisää luottamusta. Beckin mukaan ohjelmointi on myös miellyttävämpää TDD:tä käytettäessä. [4] Myös Jeffries ja Melnik ovat

samaa mieltä TDD:n miellyttävyydestä:

> ”The process might sound tedious in the telling, but the practice is rhythmic, quite pleasant, and productive. The swing from test to code to test occurs as frequently as every five or 10 minutes. It’s been compared to a waltz, to the smooth grace of skating, and to the seemingly effortless movements of a yin-style martial art. As in all these analogous situations, the practitioner is fully engaged and concentrating, while the work just seems to flow.” [26]

2.2.3 BDD

TDD:n ongelmana on pidetty sitä, että kehittäjät keskittyvät helposti liikaa testaamiseen, vaikka TDD:n suurimmat hyödyt ja keskeiset ideat liittyvät aivan muuhun kuin testaukseen. TDD:n juuret testauksessa ja testaukseen liittyvä sanasto ovat todennäköisesti johtaneet tähän tilanteeseen. Toinen suuri ongelma on keskittyminen liikaa koodin yksiköihin kuten perinteisessä yksikkötestauksessa. Tämä yksikkökeskeinen ajattelutapa johtaa helposti siihen, että myös testit jaotellaan tiukasti koodin rakenteen mukaan eikä toimintojen ja käyttäytymisen mukaan. TDD:n rinnastaminen testaukseen voi myös johtaa negatiivisiin reaktioihin ja syihin olla käyttämättä sitä kuten ”ei tätä tarvitse testata”, ”testaaminen on ajanhukkaa”, ”testaamme, kun koodi on valmis” tai ”ei ole aikaa testata”. Näiden ongelmien takia monet TDD:n käyttäjistä eivät saa täyttä hyötyä menetelmän eduista. [1]

Vastauksena näihin epäkohtiin syntyi Behaviour-Driven Development (BDD). Chelimsky et al. kuvaavat BDD:tä seuraavasti:

> ”Behaviour-Driven Development is about implementing an application by describing its behavior from the perspective of its stakeholders.” [8]

BDD pohjautuu siis TDD:n ideoihin, mutta korostaa testien sijaan ohjelmiston käyttäytymisen määrittelyä. BDD pyrkii myös huomioimaan paremmin kaikkien sidosryhmien tarpeet ja toiveet määriteltäessä ohjelmiston haluttua käyttäytymistä. Oleellista on myös huomata että sidosryhmiä on useita, siis muitakin kuin ohjelmiston loppukäyttäjä. BDD lainaakin tekniikoita Domain-Driven Designista (DDD) ja pyrkii määrittelemään ohjelmiston haluttua käyttäytymistä käyttäen sovellusalueen omaa sanastoa. [1, 8]

BDD:n perimmäiset ideat voidaan tiivistää kolmeen periaatteeseen. Ensinnäkin on oleellista tehdä vain se, mikä on oikeasti tarpeen, eikä yhtään enempää (”Enough is enough”). Toiseksi on tärkeää tehdä vain niitä asioita, jotka tuottavat lisäarvoa sidosryhmille tai parantavat mahdollisuuksia tuottaa lisäarvoa (”Deliver stakeholder value”). Viimeiseksi tulee huomata, että samaa ajattelutapaa ja kieltä voidaan käyttää käyttäytymisen kuvaamiseen sekä koodin tasolla että koko ohjelmiston tasolla

("It's all behavior"). BDD:tä voidaan käyttää huomioimalla sen periaatteet testien kirjoittamisessa ja nimeämisessä käyttäen perinteisiä testikehyksiä tai vaihtoehtoisesti käyttäen juuri BDD:tä varten kehitettyjä kehyksiä ja kieliä. [8]

2.2.4 Soveltuvuus

TDD ei välttämättä sovellu hyvin kaikenlaisten sovellusten tai komponenttien kehittämiseen, sillä TDD:n soveltaminen niiden kehittämiseen saattaa olla työlästä tai jopa mahdotonta. Tällaisia hankalia sovellusalueita ovat esimerkiksi käyttöliittymät, hajautetut järjestelmät, tietokannat, reaaliaikajärjestelmät ja turvallisuuskriittiset järjestelmät. Näille sovellusalueille tyypillistä on se, että haluttuja vaatimuksia kuvaavia testejä on hyvin vaikea tai mahdoton kirjoittaa. Näissäkin tapauksissa yleensä suurin osa koodista voidaan kehittää käyttäen TDD:tä ja vain pieni osa koodista kirjoitetaan muilla tavoin. [4]

TDD:tä voidaan kyllä usein soveltaa näissä hankalammissakin tapauksissa, mutta se vaatii enemmän työtä ja sopivia työkaluja. Esimerkiksi Freeman ja Pryce soveltavat TDD:tä Javalla toteutettavan graafisen käyttöliittymän kehittämiseen. Lisäksi he näyttävät miten TDD:tä voidaan soveltaa rinnakkaisuuteen ja tietokantoihin liittyviin ongelmiin. [17] TDD:tä voidaan hyödyntää myös turvallisuuskriittisten järjestelmien kehittämisessä, kunhan huomioidaan turvallisuuskriittisiin järjestelmiin liittyvät erityisvaatimukset. Tärkeässä asemassa on validointi, jonka tarkoituksena on varmistaa, että järjestelmä täyttää sille asetetut vaatimukset. TDD:n avulla syntyneitä testejä voidaan käyttää osana validointia, mutta lisäksi tarvitaan vaara- ja riskianalyysi sekä turvallisuusarvio. [45] Näissä tapauksissa pitää tapauskohtaisesti arvioida ovatko TDD:llä saatavat hyödyt lisätyön arvoisia.

TDD:tä voidaan soveltaa myös reaaliaikaisten ohjausjärjestelmien kehittämiseen. Esimerkiksi Dohmke esittää miten TDD:tä voidaan soveltaa PID-säätimen suunnitteluun. Dohmke käytti PID-säätimen suunnitteluun MATLAB:iin liittyvää graafista mallinnus- ja simulointityökalua Simulinkia. Dohmke loi PID-säätimelle erilaisia testejä, jotka kuvasivat suljetun järjestelmän vaatimuksia. Lisäksi vaatimusten täytyttyä Dohmke käytti suunnittelussa testejä, joilla saatiin vielä optimoitua suljetun järjestelmän käyttäytymistä. [12]

2.3 TDD sulautettujen sovellusten kehittämisessä

Vaikka TDD on kasvattanut suosiotaan esimerkiksi työpöytä- ja web-sovellusten kehityksessä, ei sitä ole kuitenkaan vielä sovellettu laajasti sulautettujen järjestelmien ohjelmistojen kehityksessä [20]. Kohdassa 2.3.1 tutkitaan ongelmia ja haasteita, jotka liittyvät TDD:n soveltamiseen sulautettujen sovellusten kehittämiseen ja pyritään löytämään ratkaisuja niihin. Kohdassa 2.3.2 esitetään laajennettu kehityssykli so-

vellettaessa TDD:tä sulautettujen järjestelmien kehittämiseen. Kohdassa 2.3.3 tutkitaan miten testisijaisia voidaan toteuttaa sulautetuissa järjestelmissä. Kohdassa 2.3.5 tutkitaan TDD:n hyötyjä erityisesti sulautettujen sovellusten kehittämisessä.

2.3.1 Ongelmia ja ratkaisuja

TDD:n käyttö sulautettujen järjestelmien ohjelmistojen kehityksessä on vielä hyvin vähäistä. Tämä on ymmärrettävää, koska aiheeseen liittyy monia haasteita, ongelmia ja ehkä jopa harhaluuloja. Seuraavaksi käydään läpi yleisimpiä ongelmakohtia ja pyritään löytämään ratkaisuja niihin.

Monet ongelmista ja haasteista liittyvät kehitettävän koodin ajamiseen kohdelaitteistossa. Usein laitteiston ja ohjelmiston kehitys etenee rinnakkain, joten kohdelaitteisto on käytettävissä ohjelmiston testaukseen vasta myöhäisessä vaiheessa projektia. Usein laitteisto on myös kallis tai siitä on olemassa vain prototyyppejä, joten jokaiselle kehittäjälle ei ole omaa laitteistoa. Lisäksi ensimmäisissä laitteistoversioissa voi olla virheitä, jotka hankaloittavat testaamista. Nämä ongelmat tietysti hankaloittavat ohjelmiston kehittämistä käyttäen TDD:tä, sillä testejä pitää pystyä ajamaan jatkuvasti kehityksen aikana. Tyypillisesti sulautetun järjestelmän ohjelmisto käännetään kehitysympäristössä ristikääntäjällä ja ladataan kohdelaitteiston muistiin. Tähän voi mennä paljon aikaa, mikä hidastaa kohdassa 2.2.1 esitettyä kehityssykliä huomattavasti. [20, 21]

Jotta nämä ongelmat saataisiin ratkaistua, voidaan sovelluksen koodia ajaa ja testata ensin kehitysympäristössä. Kehittäjä siis ensin kääntää koodin tavallisella PC:n kääntäjällä ja ajaa testit suoraan omalla tietokoneellaan. Vasta myöhemmin koodi käännetään ristikääntäjällä kohdelaitteistolle ja testataan oikealla laitteistolla. Näin kehityssykli nopeutuu riittävästi, jotta TDD:n käyttäminen on mahdollista. Kehitys ja testaus voidaan myös aloittaa jo ilman kohdelaitteistoa. Lisäksi on helpompi erottaa ohjelmistosta ja laitteistosta aiheutuvat virheet, mikä helpottaa ohjelmiston ja laitteiston integrointia. [10, 21, 40]

Edellä kuvattu järjestely ei ole kuitenkaan aivan suoraviivainen, vaan sillä on etujen lisäksi omat ongelmansa ja riskinsä. Ohjelmisto pitää pystyä kääntämään sekä kohdelaitteistolle että PC:lle, yleensä siis kahdella eri kääntäjällä, mikä aiheuttaa tietysti lisätyötä. Kohdeympäristön ja kehitysympäristön väliset eroavaisuudet voivat aiheuttaa ongelmia. Eroavaisuuksia voi olla esimerkiksi kääntäjän tukemissa ohjelmointikielen ominaisuuksissa tai laajennoksissa, kääntäjän tekemissä optimoinneissa, käytettävissä otsikkotiedostoissa, käytettävissä kirjastoissa, käyttöjärjestelmässä, tietotyyppien tavumäärissä ja tavujärjestyksissä sekä muistin varauksessa. Lisäksi toisessa kääntäjässä voi olla tietynlaisia virheitä ja toisessa taas toisenlaisia. Kohdelaitteisto on yleensä myös hyvin erilainen tavalliseen tietokoneeseen verrattuna, sillä siinä on tyyppillisesti erilaisia digitaali- ja analogiatuloja, -lähtöjä, väyliä ja

oheislaitteita. [10, 21, 40]

Kohdeympäristön ja kehitysympäristön eroista sekä laitteistoriippuvuuksista johtuen kaikkea koodia ei pystytä testaamaan kehitysympäristössä. Koodi voidaan siis jakaa ”puhtaaseen” laitteisto- ja alustariippumattomaan koodiin sekä laitteisto- ja alustakohtaiseen koodiin:

> ”This test technique required all the team members to have a clear understanding of the boundary between ”pure” code and hardware-specific code. That, in itself, was good for software design and modularity.” [40]

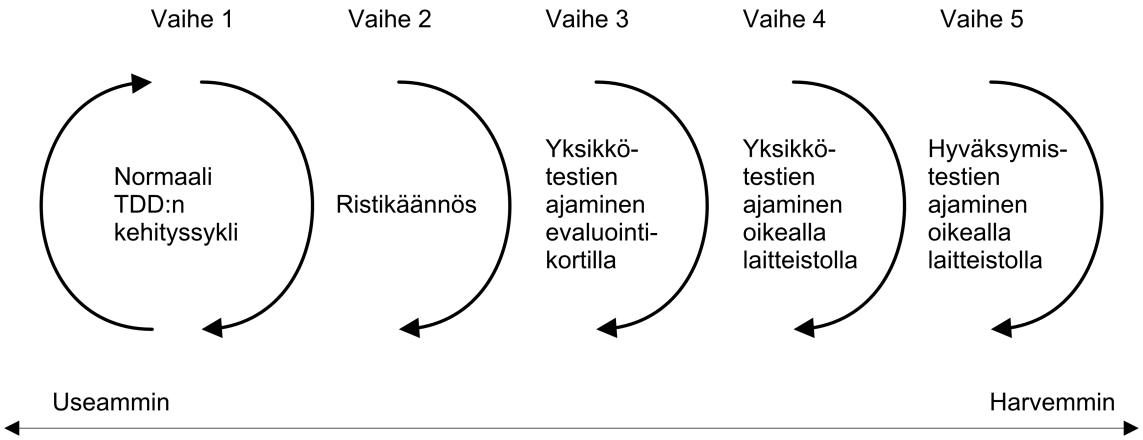
Tämä erottelu on hyvä ohjelmiston laadun ja modulaarisuuden kannalta. Jotta mahdollisimman suuri osa ohjelmistosta voitaisiin kehittää käyttäen TDD:tä, pitää laitteisto- ja alustakohtainen koodi eristää selkeiden rajapintojen taakse. Testeissä laitteisto- ja alustakohtainen koodi voidaan korvata joko kohdassa 2.1.4 esitetyillä testisijaisilla tai kehitysympäristössä toimivilla rajapintojen toteutuksilla. Yleensä vain hyvin pieni osa koodista on oikeasti riippuvainen käytetystä alustasta. [10, 21, 40]

Testejä olisi hyvä ajaa kehitysympäristön lisäksi myös oikealla laitteistolla, mahdollisten eroavaisuuksien ja ongelmien löytämiseksi. Tämä voi kuitenkin joskus olla hankalaa. Kohdelaitteistossa on muistia yleensä hyvin rajallisesti, joten kaikki testit eivät välttämättä mahdu kerralla muistiin. Tällöin testit pitää järjestää niin, että ne voidaan ajaa pienissä erissä. Kohdelaitteiston pitää myös pystyä ilmaisemaan testien tulos jotenkin. Yleensä laitteisto pystyy lähettämään testien tuloksen jotain kommunikointiväylää, kuten sarjaporttia, käyttäen. Joskus tämä ei ole kuitenkaan mahdollista, joten tulos pitää ilmaista esimerkiksi ledeillä. [10, 21]

Sulautettuja järjestelmiä kehitettäessä on usein hyvin monenlaisia järjestelmätason vaatimuksia, joista monet liittyvät sekä laitteistoon että ohjelmistoon. Näiden vaatimusten muuntaminen TDD:n avulla automaattisiksi yksikkötesteiksi voi olla vaikeaa. Tässä apuna voi hyödyntää esimerkiksi kohdassa 2.2.1 esiteltyä hyväksymistestivetoista kehitystä. Tyypillistä on myös vaatimusten muuttuminen hyvin myöhään kehitysprosessin aikana, esimerkiksi ohjelmiston ja laitteston integroinnissa ilmenneiden ongelmien takia. Tällöin pitää pystyä selvittämään mitä testejä pitää muuttaa muuttuneiden vaatimusten takia. Huolellinen vaatimustenhallinta ja muutosten hallinta on siis erittäin tärkeää. Ongelmien löytämistä voidaan myös nopeuttaa hyödyntämällä jatkuvaa integrointia, joka esitellään kohdassa 2.4. [2]

2.3.2 TDD-sykli sulautetuille järjestelmille

Kohdassa 2.3.1 esiteltiin käytäntö yksikkötestien ajamiseksi kehitysympäristössä, jotta TDD:n kehityssykli saadaan riittävän nopeaksi. Tähän käytäntöön huomattiin liittyvän monia riskejä, jotka liittyvät kehitysympäristön ja oikean kohdeym-



Kuva 2.7: Laajennettu kehityssykli sulautetuille järjestelmille.

päristön eroavaisuuksiin. Näiden riskien pienentämiseksi Grenning ehdottaa laajennetun TDD:n kehityssyklin käyttämistä. Tämä sulautettujen järjestelmien kehityssykli koostuu viidestä vaiheesta, joista ensimmäistä vaihetta suoritetaan usein ja viimeistä vaihetta harvemmin. Nämä syklin vaiheet on esitetty kuvassa 2.7. [20, 21]

Syklin ensimmäinen vaihe on kohdassa 2.2.1 esitetty normaali TDD:n kehityssykli. Tätä vaihetta suoritetaan usein ja siihen käytetään eniten aikaa. Suuri osa ohjelmiston koodista kirjoitetaan ja käännetään kehitysympäristössä ja juuri tässä vaiheessa. Testit voidaan ajaa nopeasti ja siten saada myös palautetta nopeasti. Käytettävissä on myös monia hyviä työkaluja kuten debuggereita, profilointityökaluja ja testikattavuustyökaluja. Tässä vaiheessa kirjoitettava koodi on alustariippumatonta. Laitteiston ja ohjelmiston välinen raja tulee selväksi ja näkyviin testeihin. [21]

Kuten kohdassa 2.3.1 huomattiin, kehitysympäristön ja kohdeympäristön eroavaisuudet voivat aiheuttaa ongelmia. Riskien pienentämiseksi syklin toisessa vaiheessa ohjelmiston koodi käännetään kohdeympäristöön käyttäen ristikääntäjää. Tässä vaiheessa huomataan kääntäjien eroavaisuuksista johtuvat ongelmat. Tällaisia ongelmia voivat olla esimerkiksi puuttuvat otsikkotiedostot tai puuttuvat ohjelmointikielen ominaisuudet. Tällä vaiheella voidaan siis varmistaa, että koodissa käytetään vain molemmilla kääntäjillä toimivia ominaisuuksia. Tätä vaihetta ei tarvitse suorittaa yhtä usein kuin ensimmäistä vaihetta. Tämä tulisi kuitenkin suorittaa aina, kun käytetään esimerkiksi jotain uutta kielen ominaisuutta tai otsikkotiedostoa. Tämä vaihe olisi myös hyvä sisällyttää jatkuvaan integrointiin. Jatkuvaan integrointiin palataan kohdassa 2.4. [21]

Vaikka ristikäännös onnistuisikin syklin toisessa vaiheessa, saattaa koodi silti käyttäytyä eri tavoin kehitysympäristössä ja kohdeympäristössä. Eroja ympäristöjen välillä voi olla esimerkiksi tietotyyppien tavumäärissä ja tavujärjestyksissä sekä kirjastojen käyttäytymisessä. Näiden erojen paljastamiseksi ja riskien vähentämisek-

si syklin toisessa vaiheessa ajetaan yksikkötestit evaluointikortilla. Näin huomataan, jos ympäristöjen välillä on ongelmallisia eroavaisuuksia. Myös tämä vaihe olisi hyvä sisällyttää jatkuvaan integrointiin. Jos oikea laitteisto on valmis, luotettava ja kaikkien kehittäjien käytettävissä, ei tätä vaihetta välttämättä tarvita. [21]

Neljännen vaiheen tavoitteet ovat samat kuin edellisessä vaiheessa. Tässä vaiheessa yksikkötestit ajetaan kuitenkin oikealla laitteistolla. Lisäksi tässä vaiheessa voidaan ajaa laitteistoriippuvia testejä, joilla voidaan testata laitteistoriippuvaa koodia, jota ei voida testata aiemmissa vaiheissa. Nämä testit voivat olla esimerkiksi automaattisia tai puoliautomaattisia integrointitestejä. Kuten kohdassa 2.3.1 huomattiin, testien ajaminen oikealla laitteistolla voi joskus olla hankalaa. Esimerkiksi muistin vähyyden takia testejä voidaan joutua ajamaan pienissä erissä. [21]

Syklin viimeisessä vaiheessa varmistetaan, että laitteisto ja ohjelmisto toimivat oikein kokonaisuutena ja toteuttavat halutut ominaisuudet. Tässä vaiheessa suoritetaan automaattiset ja manuaaliset hyväksymistestit. Näitä kaikkia vaiheita ei välttämättä tarvita aina, vaan tätä sykliä kannattaa soveltaa projektin ja projektin tilanteen mukaan. Esimerkiksi kahta viimeistä vaihetta ei voida suorittaa, jos oikeaa laitteistoa ei ole vielä saatavilla. [21]

2.3.3 Testisijaisten toteuttaminen

Kuten kohdassa 2.1.4 huomattiin, testattavan komponentin riippuvuudet pitää usein pystyä korvaamaan testisijaisilla. Oliopohjaisissa kielissä riippuvuuksien korvaaminen testisijaisilla on suhteellisen helppo tehdä ja toteutustapoja on useita. Sulautetuissa sovelluksissa käytetään kuitenkin usein C-ohjelmointikieltä, joka ei tue olio-ohjelmointia ja riippuvuuksien korvaaminen on hankalampaa. C:tä käytettäessä mahdollisia mekanismeja riippuvuuksien korvaamiseksi on vain muutama: linkittäjä, funktio-osoittimet ja esikääntäjä. [21]

Linkittäjällä voidaan korvata kokonainen komponentti (käännösyksikkö) testisijaisella. Oikean toteutuksen sijaan sovellukseen siis linkitetään testisijainen. Tällä tavoin voidaan korvata myös sellaisia komponentteja, joiden rajapintaa ei voida muuttaa. Tällaisia ovat esimerkiksi kolmannen osapuolen kirjastot. Tämä korvaustapa soveltuu hyvin käyttöjärjestelmän, kirjastojen ja laitteistosta riippuvien komponenttien korvaamiseen testisijaisilla. Se on yksinkertainen ja helppo tapa korvata komponentti testisijaisella, mutta ei kovinkaan joustava tapa. Tätä menetelmää käytettäessä pitää usein linkittää eri testijoukkoja erillisiksi sovelluksiksi, jotta jokaiselle testijoukolle saadaan haluttu konfiguraatio oikeita komponentteja ja testisijaisia. [21]

Joissain tilanteissa edellä esitetty tapa korvata komponentteja testisijaisilla ei ole riittävän joustava. Esimerkiksi joskus on tarve korvata oikea komponentti testisijaisella vain osassa testitapauksista tai on tarve korvata vain osa jostain komponen-

tista (käännösyksiköstä). Tällaisissa tilanteissa voidaan käyttää funktio-osoittimia suorien funktiokutsujen sijaan. Normaalisti funktio-osoitin alustetaan osoittamaan funktion oikeaan toteutukseen. Testeissä tämä funktio-osoitin voidaan kuitenkin asettaa osoittamaan testisijaiseen. Ohjelma 2.1 on esimerkki funktio-osoittimen käytöstä. Funktio-osoittimilla voidaan tehdä korvauksia hyvin joustavasti, sillä käytettäviä toteutuksia voidaan vaihtaa ajonaikaisesti. Menetelmän huonoja puolia ovat hieman suurempi muistinkulutus, monimutkaisuus ja koodin huonompi luettavuus. Funktio-osoittimien sijaan kannattaakin useimmissa tapauksissa käyttää linkittäjää, jos se vain on mahdollista. [21]

```
// In header file:

extern int (*DoSomething)(int param);

// In code file:

static int DoSomethingImpl(int param)
{
    /* Implementation */
}

/* Initialize with the real implementation. */
int (*DoSomething)(int param) = DoSomethingImpl;

// In test file:

static int DoSomethingStub(int param)
{
    /* Stub implementation. */
}

/* Replace with a test double. */
DoSomething = DoSomethingStub;
```

Ohjelma 2.1: Toteutuksen korvaaminen testisijaisella funktio-osoitinta käyttäen.

Joissain tilanteissa ei ole mahdollista tehdä korvausta edes linkittäjällä tai funktio-osoittimilla. Esimerkiksi vain tiettyjen standardikirjaston funktioiden korvaaminen testisijaisilla ei onnistu näillä menetelmillä. Näissä tilanteissa voidaan käyttää esikääntäjän makroja, joilla voidaan korvata funktioita hyvin joustavasti. Tätä menetelmää tulisi kuitenkin käyttää vain poikkeuksellisissa tilanteissa ja viimeisenä vaihtoehtona. Yleensä kannattaakin ennemmin luoda korvattavalle koodille uusi rajapinta ja käyttää linkittäjää tai funktio-osoittimia. [21]

2.3.4 Arkkitehtuuri ja suunnitteluperiaatteet

Vaikka TDD:n käyttäminen ohjaakin sovelluksen arkkitehtuuria ja suunnittelua modulaariseen ja testattavaan suuntaan, kannattaa arkkitehtuuria ja suunnitteluperiaatteita miettiä etukäteen ja noudattaa hyväksi havaittuja käytäntöjä. Yksi tärkeimmistä periaatteista on sovelluksen jakaminen moniin pieniin komponentteihin, joilla jokaisella on jokin selkeä vastuualue. C-ohjelmointikieltä käytettäessä kannattaa hyödyntää Liskovin ideoita abstraktista tietotyypistä. Abstrakti tietotyyppi on määritelty epäsuorasti sille suoritettavien operaatioden kautta. Tietotyypin varsinainen tietosisältö on piilotettu abstraktin tietotyypin sisään ja sitä voidaan käsitellä tietotyypin ulkopuolelta vain epäsuorasti julkisen rajapinnan kautta. [21, 29]

C-ohjelmointikielellä komponentin tietosisältö voidaan piilottaa määrittelemällä muuttujat *static*-avainsanalla, jolloin niihin voidaan viitata vain saman käännösyksikön sisällä olevissa funktioissa. Tämä menetelmä toimii, jos komponentti hallitsee kerrallaan vain yhtä tietokokonaisuutta. Jos komponentin pitää hallita useita tietokokonaisuuksia kerrallaan, voidaan tiedon piilottaminen tehdä toisella tavalla. Komponentti alustaa tietotyypin ja palauttaa komponentin käyttäjälle osoittimen siihen. Komponentin käyttäjä välittää saamansa osoittimen aina parametrina käyttäessään komponentin funktioita. Tietotyypin varsinainen sisältö voidaan pitää piilossa komponentin käyttäjältä esittelemällä tietotyyppi etukäteen (engl. forward declaration), jolloin komponentin käyttäjä voi käsitellä osoittimia tietotyyppiin tietämättä sen rakennetta. Tietotyypin rakenne määritellään komponentin kooditiedostossa eikä otsikkotiedostossa, jolloin se on näkyvissä vain komponentin sisäisille funktioille. [21]

Toteutettaessa komponentteja C:llä, tulisi noudattaa muutamia käytäntöjä. Ensinnäkin komponentin julkinen rajapinta tulee määritellä erillisessä otsikkotiedostossa, jolloin rajapinnat on selkeästi määritelty ja komponentit käyttävät toisia komponentteja vain näiden rajapintojen kautta. Otsikkotiedostossa esitellään julkisen rajapinnan funktioiden prototyypit ja määritellään tarvittavat tietotyypit ja vakiot. Funktioiden toteutukset ovat varsinaisessa kooditiedostossa. Testit kirjoitetaan erilliseen tiedostoon, jolloin testikoodi ja toteutuskoodi pysyvät selvästi erillään. Yhdellä komponentilla on tyypillisesti yksi testitiedosto, joskus useampia. [21]

Jotta komponentti olisi helposti testattavissa, se pitää pystyä asettamaan johonkin tunnettuun tilaan. Tämä on välttämätöntä, jotta samalle komponentille voidaan ajaa useita testitapauksia ja alustaa komponentin sisäinen tila aina jokaisen testitapauksen alussa. Tätä varten komponenteille kannattaa tehdä alustusfunktio, joka alustaa komponentin sisäiset muuttujat johonkin järkevään alkutilaan. Joillekin komponenteille kannattaa tehdä myös purkufunktio, joka vapauttaa komponentin varaamat resurssit. C++-kielessä samaan tarkoitukseen voidaan käyttää luokan rakentajaa ja purkajaa. Tietysti komponentti on vielä parempi, jos sillä ei ole lainkaan

sisäistä tilaa. Tällöin alustusfunktiota ei välttämättä tarvita. [21]

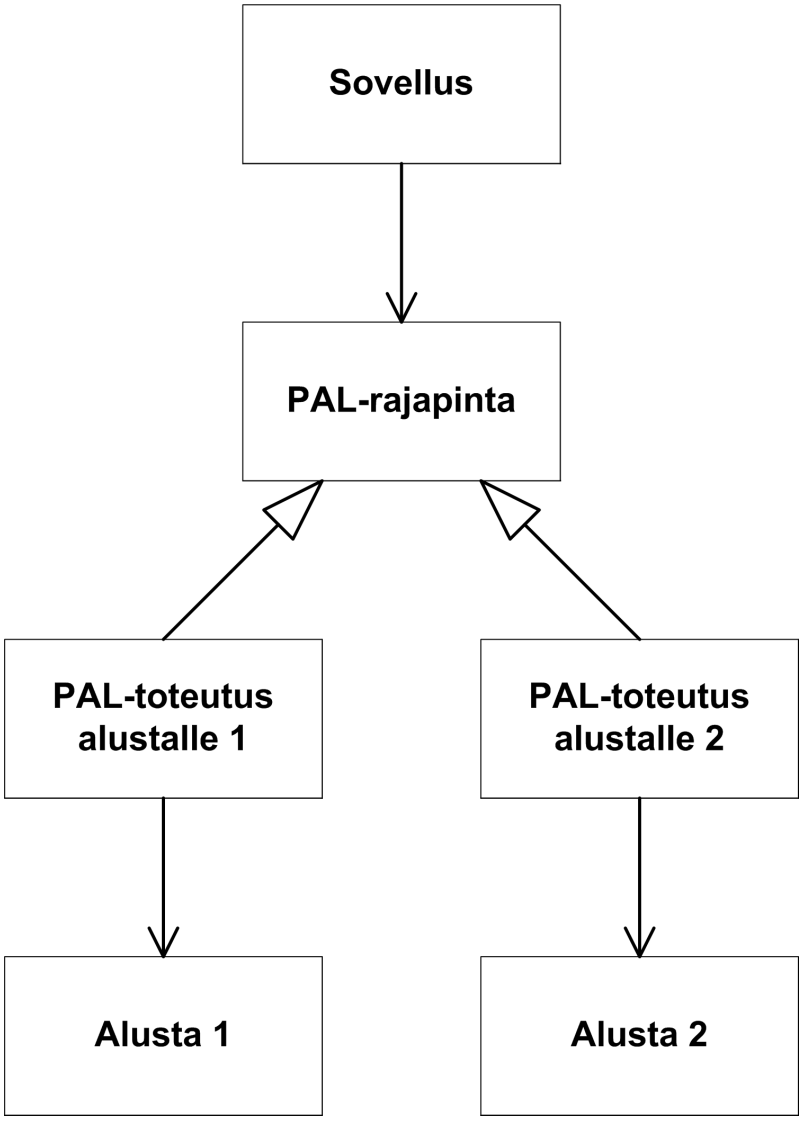
Kuten kohdassa 2.3.1 todettiin, yleensä osa koodista on laitteisto- ja alustariippuvaista ja osa ei. Tällöin on tärkeää eristää laitteisto- ja alustariippuvainen koodi yhteen paikkaan, jotta käytettävää alustaa voidaan helposti vaihtaa. Tällöin kannattaa hyödyntää kuvassa 2.8 esitettyä kerrosarkkitehtuuria, jossa on ylempi sovelluskerros ja alempi abstraktiokerros. Ylemmässä sovelluskerroksessa on vain laitteisto- ja alustariippumatonta koodia, joten sitä voidaan käyttää kaikilla alustoilla. Abstraktiokerros määrittelee rajapinnat laitteisto- ja alustariippuville toiminnoille, joita sovelluskerros sitten käyttää näiden rajapintojen kautta. Tätä abstraktiokerrosta kuvataan usein nimillä Platform Abstraction Layer (PAL), Platform Adaptation Layer (PAL), Hardware Abstraction Layer (HAL) tai Operating System Abstraction Layer (OSAL). Rajapinnoille tehdään erilliset toteutukset kaikille tuetuille alustoille ja oikea toteutus voidaan valita linkitysvaiheessa. Rajapintojen toteutuksessa voidaan usein hyödyntää adapteri-suunnittelumallia, joka muuntaa rajapinnan funktiokutsut jonkin toisen rajapinnan funktiokutsuiksi [18]. Tällä tavoin voidaan käyttää esimerkiksi käyttöjärjestelmän tai erilaisten kirjastojen rajapintoja. [21]

2.3.5 Etuja

Sovellettaessa TDD:tä sulautettujen sovellusten kehittämiseen voidaan tietysti saavuttaa samoja etuja kuin TDD:llä normaalistikin. Näitä etuja on esitelty kohdassa 2.2.2. Näiden lisäksi on muutamia erityisesti sulautettujen sovellusten kehittämiseen liittyviä etuja. [21]

TDD:n avulla voidaan vähentää laitteistoon liittyviä riskejä ja kuluja, sillä koodia voidaan helpommin kehittää ja testata jo ennen kuin laitteisto on valmis. Tyypillisesti laitteisto myös kehittyy ja muuttuu useaan kertaan ohjelmiston kehityksen aikana. TDD helpottaa sopeutumista muutoksiin. TDD myös helpottaa kehittämistä siinä tapauksessa, että käytettävä laitteisto on kallis tai laitteistoja on käytettävissä vain vähän. Koska koodia pystytään testaamaan muullakin kuin oikealla laitteistolla, voidaan ohjelmistovirheet ja laitteistovirheet helpommin erottaa toisistaan, mikä muuten olisi hyvin hankalaa. TDD:tä käytettäessä voidaan myös helpommin simuloida harvinaisia virhetilanteita, joita olisi muuten hyvin hankala testata. [21, 40]

Lisäksi TDD:n avulla voidaan vähentää aikaa, jota käytetään ristikäännökseen, linkkaukseen ja ohjelmiston lataukseen kohdelaitteistoon. Tämä on mahdollista, koska suuri osa virheistä voidaan havaita jo kehitysympäristössä. Pitkiä ristikäännös-, linkkaus- ja lataussyklejä ei siis tarvitse suorittaa yhtä usein. Samasta syystä voidaan vähentää oikealla laitteistolla debuggaukseen käytettävää aikaa huomattavasti. Näistä seurauksena on se, ettei kohdelaitteiston tyypillisesti kalliita kääntäjiä ja työkaluja tarvita yhtä paljon. TDD siis mahdollistaa rahan säästämisen lisenssikuluissa. [21]



Kuva 2.8: Alustan abstraktiokerros.

Koska TDD:tä käytettäessä pitää laitteiston ja ohjelmiston välistä interaktiota mallintaa testeillä, voidaan huomata siihen liittyviä ongelmia jo varhaisessa vaiheessa. Parhaassa tapauksessa ongelmat voidaan huomata jo ennenkuin laitteiston kehitys on valmis, jolloin ongelmat voidaan vielä helposti korjata. Testaamisen helpottamiseksi on myös riippuvuudet laitteistoon ja käyttöjärjestelmään minimoitava ja eristettävä, mikä parantaa ohjelmiston laatua. Ohjelmisto on helpommin siirrettävissä toiselle kohdelaitteistolle, sillä suuri osa koodista on arkkitehtuuri- ja alustariippumatonta ja kattavat yksikkötestit varmistavat ohjelmiston oikean toiminnan uudessa laitteistossa. [21]

2.4 Jatkuva integrointi

Perinteisiä menetelmiä käyttäen saatetaan koko ohjelmisto koostaa ja integroida vasta projektin loppupuolella. Tällöin komponenttien liittämisessä yhteen voi tulla suuriakin ongelmia, jolloin integrointiin kuluu paljon aikaa. Mitä suurempi projekti, sitä enemmän aikaa integrointiin menee. Tähän ongelmaan on ratkaisuksi esitetty jatkuvaa integrointia (engl. Continuous Integration). Jatkuva integrointi on prosessi, jossa koko ohjelmisto koostetaan, integroidaan ja testataan mahdollisimman usein. Yleensä tämä integrointi suoritetaan kerran päivässä tai aina vietäessä muutoksia versionhallintaan. Yleensä tämä prosessi myös automatisoidaan esimerkiksi integrointipalvelinta käyttäen. Jatkuva integrointi sopii myös hyvin yhteen TDD:n ajatusmallien kanssa ja ne tukevat hyvin toisiaan. TDD:n myötä syntyneitä ja muita automaattisia testejä voidaan ajaa osana jatkuvaa integrointia. Lisäksi voidaan tehdä esimerkiksi staattista analyysiä koodille tai generoida dokumentaatiota koodin kommenttien perusteella. [14, 23]

Jotta jatkuvan integroinnin hyödyntäminen olisi mahdollista, pitää noudattaa muutamia käytäntöjä. Ensinnäkin kaikki ohjelmiston kääntämiseen ja koostamiseen tarvittavat tiedostot pitää olla helposti saatavilla, esimerkiksi versionhallinnasta. Lisäksi ohjelmiston kääntäminen ja koostaminen pitää automatisoida, jotta se on suoritettavissa helposti yhdellä komennolla. Tämän prosessin pitäisi olla mahdollisimman nopea, jotta se voidaan suorittaa mahdollisimman usein, mielellään aina vietäessä muutoksia versionhallintaan. [14, 23]

Jatkuvan integroinnin avulla saavutetaan osittain samoja etuja kuin TDD:lläkin. Jatkuvalla integroinnilla pystytään poistamaan integrointiin liittyvää epävarmuutta ja ongelmia, sillä integrointia tehdään jatkuvasti eikä vain projektin lopuksi. Aina on siis helposti nähtävissä, kääntyykö ohjelmisto ja menevätkö automaattiset testit läpi. Monet virheistä tulevat myös nopeammin esille jatkuvan integroinnin ansiosta. Lisäksi ohjelmistosta voidaan nopeasti koostaa versio testattavaksi tai julkaistavaksi ilman pitkää erillistä integrointivaihetta. Jatkuvaa integrointia voidaan myös viedä vielä pidemmälle ottamalla mukaan ohjelmiston jatkuva julkaisu tai käyttöönotto (engl. Continuous Delivery), jolloin ohjelmistosta koostetaan usein uusia versioita julkaistavaksi tai otettavaksi tuotantokäyttöön. Tällöin ohjelmistoon tehdyistä muutoksista saadaan hyvin nopeasti arvokasta palautetta käyttäjiltä. [14, 23]

3. HITSAUSLAITTEESEEN LIITETTÄVÄN TIEDONKERUULAITTEEN OHJELMISTON KEHITTÄMINEN

Tässä luvussa kuvataan tutkimuksen soveltavaa vaihetta eli testivetoisen kehityksen soveltamista oikeassa ohjelmistokehitysprojektissa. Tutkittavassa projektissa toteutettiin Kempille hitsauslaitteeseen liitettävän tiedonkeruulaitteen ohjelmisto. Projektiin liittyvät laitteistot esitellään kohdassa 3.1. Tiedonkeruulaitteen tarkoitus ja toiminta esitellään kohdassa 3.2. Kohdassa 3.3 kuvataan toteutetun ohjelmiston arkkitehtuuria ja suunnitteluperiaatteita. Lopuksi kohdissa 3.4 ja 3.5 kuvataan, miten testivetoista kehitystä ja jatkuvaa integrointia sovellettiin tutkittavassa projektissa.

3.1 Laitteisto

Seuraavaksi esitellään lyhyesti projektiin liittyvät laitteet. Projektissa toteutettu ohjelmisto tehtiin Kempin DataGun-laitteelle, joka esitellään kohdassa 3.1.1. Kohdassa 3.1.2 esitellään hitsauslaitteita, joiden kanssa DataGun-laitetta voidaan käyttää.

3.1.1 DataGun

DataGun on Kempin myymä helppokäyttöinen ohjelmointilaite hitsauslaitteille. DataGunin avulla voidaan suorittaa monenlaisia hitsauslaitteisiin liittyviä ohjelmointi- ja ylläpitotehtäviä. DataGunia käyttäen voidaan esimerkiksi asentaa ja päivittää hitsauslaitteiden laiteohjelmia, varmuuskopioida ja palauttaa hitsauslaitteiden asetuksia ja asentaa hitsausprosesseja sekä materiaalikohtaisia hitsausohjelmia. Asennettavia laiteohjelmien päivityksiä, hitsausprosesseja ja materiaalikohtaisia hitsausohjelmia voidaan ladata Kempin DataStore-verkkokaupasta. [37, 38]

DataGun-ohjelmointilaite on esillä kuvassa 3.1. Laitteen vasemmassa reunassa on nähtävissä USB-liitin (Universal Serial Bus), jolla laite voidaan kytkeä tietokoneeseen. Kun DataGun on kytkettynä tietokoneeseen, voidaan laitteen muistiin ladata esimerkiksi laiteohjelmia tai hitsausohjelmia. Tämän jälkeen DataGun voidaan kytkeä laitteen päässä olevalla liittimellä hitsauslaitteeseen, jolloin ohjelmointilaitteelle ladatut ohjelmat asennetaan automaattisesti hitsauslaitteelle. Toiminnon etenemistä voi seurata DataGunin päällä olevista merkkivaloista.

DataGun on ARM7-arkkitehtuuriin pohjautuvaan prosessoriin perustuva sulautettu laite, joka sisältää 64 kilotavua käyttömuistia ja 512 kilotavua haihtumatonta Flash-muistia. Lisäksi laitteessa on jo aikaisemin mainittu USB-liitin, jonka avulla laite voi kommunikoida tietokoneen kanssa. Hitsauslaitteen kanssa laite pystyy kommunikoimaan CAN-väylän (Controller Area Network) avulla. Käyttöjännitteen laite saa joko tietokoneelta USB:n kautta tai hitsauslaitteelta sen mukaan kumpaan laite on kytketty. Laitteessa on myös muun muassa reaaliaikakellopiiri, merkkivaloja ja SD-muistikortteja (Secure Digital) lukeva muistikortinlukija. Muistikorttia käytetään esimerkiksi laiteohjelmien ja hitsausohjelmien tallentamiseen.

Ohjelmistokehitystä varten laitteesta oli käytettävissä versio, jossa oli liitin JTAG-liityntää (Joint Test Action Group) varten. JTAG on yleinen nimitys IEEE:n (Institute of Electrical and Electronics Engineers) standardille 1149.1, joka määrittelee liitynnän laitteiston ja ohjelmiston testaamiseksi. JTAG-liittimeen voidaan kytkeä debuggeri (In-Circuit Debugger), joka kytketään edelleen tietokoneeseen. Tämä mahdollistaa laitteessa ajettavan ohjelman debuggaamisen eli esimerkiksi prosessorin rekisterien ja muistin sisällön tarkastelun, keskeytyspisteiden asettamisen ja ohjelman ajamisen askeleittain. JTAG-liitynnän kautta voidaan myös kirjoittaa ohjelma ja data laitteen Flash-muistille. [9]

Kuva 3.1: DataGun-ohjelmointilaite Kempin hitsauslaitteille [37].

3.1.2 Hitsauslaitteet

Edellä esiteltyä DataGun-ohjelmointilaitetta voidaan käyttää monien Kempin hitsauslaitteiden kanssa. Yhteensopivia hitsauslaitteita ovat esimerkiksi FastMig KMS ja FastMig Pulse. Molemmat sisältävät elektronisen säädön ja ovat rakenteeltaan

modulaarisia. Modulaarisen rakenteen ansiosta asiakas voi valita juuri itselleen sopivan kokoonpanon. Kokoonpanoon valittavia asioita ovat esimerkiksi virtalähde, langansyöttölaite, langansyöttölaitteen ohjauspaneeli, jäähdytysratkaisu, langansyöttöetäisyys ja etähallintalaitteet. Kuvassa 3.2 on esimerkki FastMig Pulse -hitsauslaitteesta. Kuvassa näkyvät moduulit ovat järjestyksessä alimmasta alkaen FastCool 10 -jäähdytysyksikkö, FastMig Pulse 450 -virtalähde ja FastMig MXF-67 -langansyöttölaite. DataGun-ohjelmointilaite voidaan kytkeä virtalähteen ja langansyöttölaitteen etupaneeleissa näkyviin sinisiin liittimiin. [37, 38]

Projektissa oli testausta varten käytettävissä hitsauslaitteista muokattuja versioita, joilla pystyttiin simuloimaan hitsausta ilman valokaarta. Tämä mahdollisti tiedonkeruulaitteen testaamisen helposti toimistotiloissa. Testauskäyttöön muokatut hitsauslaitteet koostuivat pelkästä langansyöttölaitteesta, johon oli asennettu myös esimerkiksi virtalähteen ohjauselektroniikka. Näihin laitteisiin oli tehty myös muita muutoksia helpottamaan testausta. Niiden etupaneeliin oli esimerkiksi asennettu kytkin, jolla hitsaus voidaan aloittaa ja lopettaa ilman hitsauspistoolia.

3.2 Tiedonkeruulaite

Tutkittavassa projektissa toteutettiin kohdassa 3.1.1 esitellylle DataGun-laitteelle ohjelmisto, joka muuntaa sen tiedonkeruulaitteeksi. Tiedonkeruulaitteen tarkoituksena on tallentaa hitsausprosessista tietoja, joita voidaan hyödyntää monenlaisissa raportointi-, valvonta- ja analyysitehtävissä. Kerättyjen tietojen avulla voidaan seurata esimerkiksi hitsauksen laatua, tuotannon tehokkuutta, hitsauslaitteen kuntoa ja energiankulutusta. Hitsauksen aikana mitattavia ja tallennettavia tietoja ovat esimerkiksi hitsausjännite, hitsausvirta ja langansyöttönopeus. Hitsauksen aikana tallennetaan myös erilaisia hitsauslaitteen asetusparametreja kuten käytetty hitsausprosessi ja langansyöttöetäisyys. Lisäksi hitsauslaitteen kokoonpanosta kerätään joitain tietoja kuten mallinimiä, versionumeroita ja sarjanumeroita. Kaikki kerätyt tiedot tallentuvat laitteen muistikortille ja ne voidaan myöhemmin ottaa talteen kytkemällä laite tietokoneeseen.

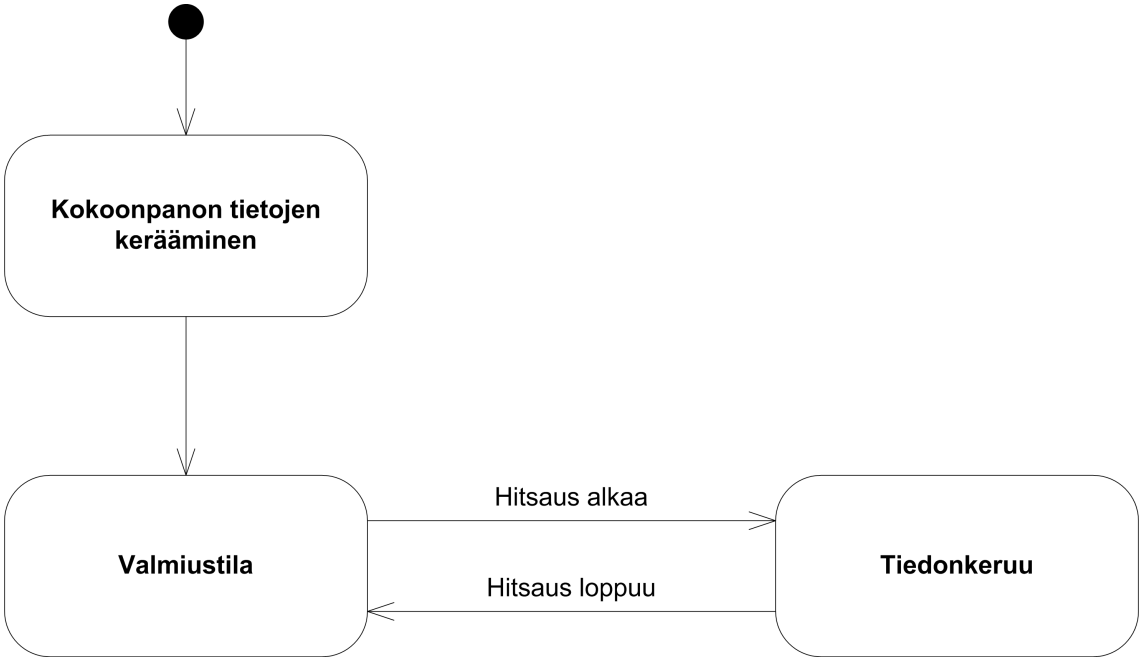
Tiedonkeruulaitteen toiminta on kuvattu tilakaaviona kuvassa 3.3. Tiedonkeruulaitteen toiminta alkaa, kun laite liitetään hitsauslaitteeseen ja hitsauslaitteeseen kytketään virrat päälle. Aluksi tiedonkeruulaite selvittää millaiseen hitsauslaitteeseen se on kytketty ja kerää muistiin tarvittavat tiedot. Tämän jälkeen tiedonkeruulaite siirtyy valmiustilaan, jossa laite tarkkailee hitsauslaitteen tilaa ja odottaa valokaaren syttymistä eli hitsauksen alkamista. Hitsauksen alkaessa laite siirtyy tiedonkeruutilaan, jossa laite tallentaa näytteitä asetusparametreista ja mitattavista suureista 100 millisekunnin välein. Kerätyt tiedot tallennetaan laitteen muistikortille XML-muodossa. Valokaaren sammuessa eli hitsauksen loppuessa laite siirtyy taas takaisin valmiustilaan. Tiedonkeruulaitteen toiminta on täysin automaattista

Kuva 3.2: Kemppi FastMig Pulse -hitsauslaite [38].

eli käyttäjän ei tarvitse tehdä muuta kuin liittää laite hitsauslaitteeseen.

3.3 Ohjelmistoarkkitehtuuri

Tiedonkeruuohjelmisto toteutettiin ANSI-C -kielellä (C89), joka mahdollistaa koodin hyvän siirrettävyyden, sillä ANSI-C -kieli on tuettuna hyvin monissa kääntäjissä. Ohjelmiston suunnittelussa pyrittiin alustariippumattomuuteen mahdollisimman suurelta osin ohjelmistoa. Alustariippuvainen koodi pyrittiin eristämään selkeästi alustariippumattomasta koodista. Nämä suunnitteluperiaatteet mahdollistivat koodin hyvän siirrettävyyden ja yksikkötestien ajamisen helposti kehitysympäristössä, mikä helpotti huomattavasti TDD:n käyttöä. Muita sovellettuja suunnitteluperiaatteita olivat koodausstandardin noudattaminen, ehdollisen kääntämisen välttäminen, komponenttien välisten riippuvuuksien minimointi ja rinnakkaisten säikeiden lukumäärän minimointi. Lisäksi sovellettin kohdassa 2.3.4 esiteltyjä periaatteita.



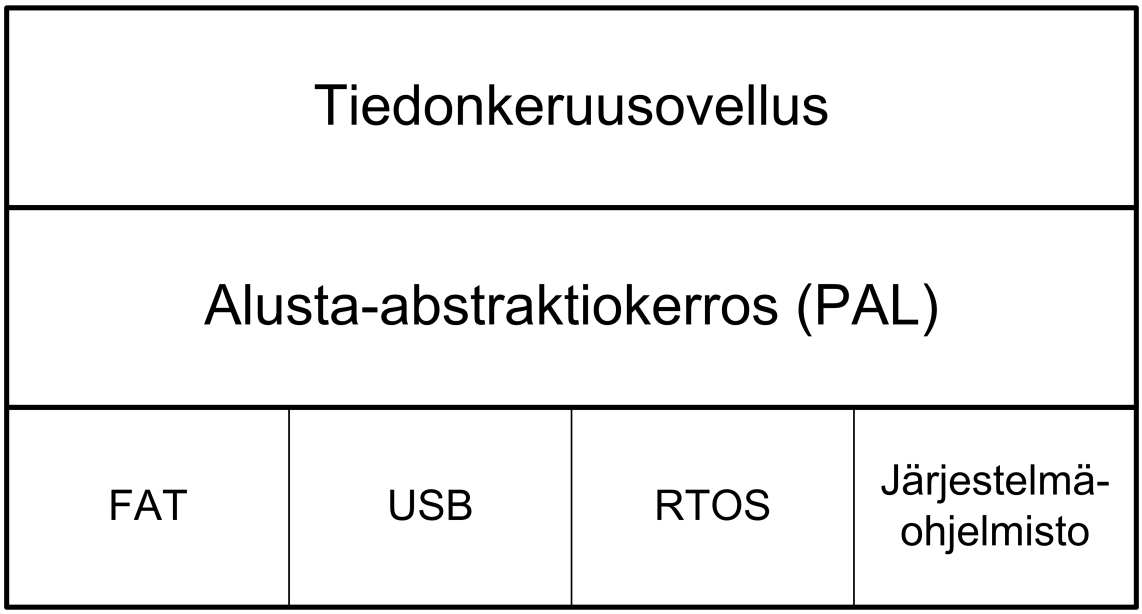
Kuva 3.3: Tilakaavio tiedonkeruulaitteen toiminnasta.

Tiedonkeruuohjelmiston ohjelmistoarkkitehtuurin yleiskuva on esitetty kuvassa 3.4. Tiedonkeruuohjelmiston suunnittelussa hyödynnettiin kerrosarkkitehtuuria, jossa ohjelmiston komponentit jaetaan eri kerroksiin niiden abstraktiotason mukaan. Ylempien kerrosten komponentit hyödyntävät alempien kerrosten tarjoamia palveluita toteuttaessaan omia palveluitaan. Kerrosarkkitehtuuri jakaa ohjelmiston pienempiin ja helpommin hallittaviin osiin sekä helpottaa ohjelmiston rakenteen hahmottamista. Kerrosarkkitehtuuri myös rajoittaa komponenttien välisiä riippuvuuksia, sillä kerroksen komponentit riippuvat vain saman kerroksen ja välittömästi alempana olevan kerroksen komponenteista. [6]

Ohjelmistoarkkitehtuurin ylin kerros sisältää varsinaisen sovelluslogiikan eli komponentit, jotka toteuttavat kohdassa 3.2 esitellyt toiminnot. Tämä kerros sisältää vain alustariippumatonta koodia, joten sitä pystyttiin helposti kehittämään ja testaamaan kehitysympäristössä. Sovelluskerros käyttää toimintojensa toteuttamiseen keskimmäisen kerroksen palveluita. Keskimmäinen kerros on kohdan 2.3.4 mukainen alusta-abstraktiokerros, joka määrittelee rajapinnat alustariippuvaisille toiminnoille. Tämä kerros sisältää rajapintoja esimerkiksi käyttöjärjestelmän palveluiden, tiedostojärjestelmän, CAN-väylän ja muiden laitteistonläheisten toimintojen käyttämistä varten. Monien rajapintojen toteuttamisessa hyödynnetään adapterisuunnittelumallia eli rajapinnan funktiokutsut ohjataan alimman kerroksen komponenteille.

Ohjelmistoarkkitehtuurin alin kerros sisältää monia alustariippuvaisia komponentteja, jotka tarjoavat laitteistonläheisiä palveluita. Näitä komponentteja käyte-

tään alusta-abstraktiokerroksen määrittelemien rajapintojen toteuttamiseen. Kerros sisältää reaaliaikakäyttöjärjestelmän, Kempin yleiskäyttöisen järjestelmäohjelmiston, USB-protokollapinon ja FAT-tiedostojärjestelmätoteutuksen. Reaaliaikakäyttöjärjestelmä huolehtii esimerkiksi suoritettavien säikeiden skeduloinnista ja tarjoaa ajastimiin, viestijonoihin, säikeisiin ja semaforeihin liittyviä palveluja. Kempin yleiskäyttöistä järjestelmäohjelmistoa käytetään pohjana monissa Kempin laiteohjelmistoissa ja se tarjoaa monia palveluja, jotka liittyvät esimerkiksi laitteiden väliseen kommunikointiin, ohjelmistojen päivittämiseen sekä hitsausprosessin ja -parametrien hallintaan ja seurantaan. USB-protokollapino ja FAT-tiedostojärjestelmätoteutus mahdollistavat yhdessä laitteen toimimisen USB-massamuistilaitteena.



Kuva 3.4: Tiedonkeruuohjelmiston arkkitehtuuri.

3.4 TDD:n soveltaminen projektissa

Projektissa ei pyritty tiukasti soveltamaan TDD:tä jokaisessa tilanteessa, vaan sitä hyödynnettiin tilanteen, osaamisen ja kehittäjän halukkuuden mukaan. Projektissa tutkija pyrki kirjoittamaan suurimman osan tuottamastaan koodista TDD:tä käyttäen, mutta aivan kaikkea ei kuitenkaan ollut käytännöllistä tehdä TDD:n avulla. Muut projektissa mukana olleet kehittäjät kokeilivat myös jonkin verran TDD:tä. Lisäksi joillekin komponenteille kirjoitettiin yksikkötestejä vasta toteutuksen jälkeen, mikä mahdollisti TDD:n ja perinteisen yksikkötestauksen vertailun samassa projektissa ja samoilla työkaluilla.

Projektissa hyödynnettiin melko paljon vanhaa koodia, joka on monimutkaisten riippuvuussuhteiden, kääntäjäriippuvaisen koodin ja suurien komponenttien takia hankalasti testattavaa. Vaikka tähän vanhaan koodiin tehtiinkin jonkin verran

muutoksia, ei sille kuitenkaan pääsääntöisesti pyritty kirjoittamaan yksikkötestejä suuren työmäärän takia. Jotta vanhalle koodille olisi pystynyt kirjoittamaan hyviä yksikkötestejä, olisi se vaatinut hyvin paljon refaktorointia. Vanhaan koodiin muutoksia tehtäessä kehitys ja testaus tehtiin ajamalla koodia kohdelaitteessa ja hyödyntämällä debuggeria.

TDD:tä käytettäessä pyrittiin noudattamaan mahdollisimman hyvin luvussa 2 esiteltyjä hyviä menetelmiä ja käytäntöjä. Yksikkötestit pyrittiin kirjoittamaan siten, että ne olisivat kohdan 2.1.1 mukaisia hyviä yksikkötestejä. Tämä oli erittäin tärkeää, jotta yksikkötesteistä olisi hyötyä myös jatkossa eikä niiden ylläpitoon menisi kohtuuttomia työmääriä. Myös kohdassa 2.1.4 esiteltyjä testisijaisia hyödynnettiin paljon. Testisijaisten avulla pystyttiin hyvin korvaamaan vaikeasti testattavat vanhan koodin komponentit, kun kehitettiin uusia komponentteja. Kohdassa 2.2.3 esitellyn BDD:n periaatteita hyödynnettiin myös. Erityisesti testien nimeämisessä hyödynnettiin BDD:n ideoita ja lisäksi testit yritettiin kirjoittaa niin, että ne määrittelevät komponentin toiminnan. Testit pyrittiin nimeämään seuraavassa muodossa: *test_[testattavan funktion nimi]_Should[jokin vaatimus funktiolle]*, eli esimerkiksi *test_SAMPLING_Execute_ShouldStartSamplingTimer_IfWeldingStarts*.

Kuten kohdassa 2.3.1 todettiin, TDD:n soveltaminen ajamalla testejä suoraan kohdelaitteessa voi olla hankalaa. Tämän takia tässäkin projektissa hyödynnettiin kohdassa 2.3.2 esiteltyä kehityssykliä sulautetuille järjestelmille. Tämän syklin kaikkia vaiheita ei kuitenkaan käytetty. Käytännössä ensin kehitettiin komponentteja normaalin TDD:n kehityssyklin mukaisesti ajamalla testejä kehityskoneessa. Apuna tässä vaiheessa käytettiin Unity-yksikkötestikehystä, CMock-ohjelmistokehystä testisijaisten generointiin ja Ceedling-ohjelmistokehystä testien automaattiseen kääntämiseen ja ajamiseen. Nämä työkalut esitellään tarkemmin luvussa 4. Seuraavassa vaiheessa koodi käännettiin ristikääntäjällä, jolloin saatiin selville esimerkiksi kääntäjien ja kirjastojen eroavaisuuksiin liittyviä ongelmia. Viimeisessä vaiheessa testattiin ohjelmistoa kokonaisuutena oikealla laitteistolla. Kohdassa 2.2.1 kuvattua hyväksymistestivetoista kehitystä (ATDD) ei tässä projektissa hyödynnetty, vaan kaikki hyväksymistestit suoritettiin manuaalisesti.

3.5 Jatkuvan integroinnin soveltaminen projektissa

Projektissa hyödynnettiin kohdassa 2.4 esiteltyä jatkuvaa integrointia. Jatkuvan integroinnin prosessi automatisoitiin käyttäen integrointipalvelinta, jossa ajettiin Jenkins-integrointipalvelinohjelmistoa. Jenkins esitellään tarkemmin kohdassa 4.4. Jatkuvan integroinnin mahdollistamiseksi kaikki ohjelmiston kääntämiseen ja koostamiseen vaadittavat tiedostot säilytettiin versionhallinnassa. Säilytettäviä tiedostoja ovat esimerkiksi kaikki kooditiedostot, mukaan lukien testikooditiedostot, ja erilaiset konfiguraatiotiedostot.

Integrointipalvelin suoritti integroinnin automaattisesti aina, kun versionhallintaan vietiin muutoksia. Integrointi suoritettiin siis useaan kertaan päivässä. Integrointipalvelin käänsi yksikkötestit, ajoi ne, tallensi testitulokset, tallensi testikattavuudet ja generoi dokumentaation lähdekoodista. Jos prosessissa tuli esille jokin ongelma, lähetti integrointipalvelin siitä ilmoituksen sähköpostilla kehittäjille. Ilmoitus lähetettiin esimerkiksi käännöksen päättyessä virheeseen tai testien epäonnistuessa. Ristikäännöstä ei pystytty tekemään automaattisesti integrointipalvelimella, sillä kääntäjän lisenssejä ei ollut riittävästi käytettävissä. Myös kääntäjän lisenssin vaatiman USB-avaimen käyttäminen virtuaalipalvelimessa olisi ollut hankalaa. Ristikäännös jouduttiin siis tekemään kehittäjien omilla tietokoneilla manuaalisesti.

4. TYÖKALUT

Kohdassa 2.2.1 esitellyn TDD:n kehityssyklin sujuvaa käyttämistä ja kohdassa 2.1.1 kuvattujen hyvien yksikkötestien kirjoittamista voidaan helpottaa huomattavasti hyvillä ja tilanteeseen sopivilla työkaluilla. Toki TDD:n hyödyntäminen on mahdollista ilman mitään valmiita työkaluja, mutta niiden käyttäminen on erittäin hyödyllistä. Työkalujen asema kehitysprosessissa onkin keskeinen TDD:tä käytettäessä, sillä testien kirjoittaminen ja ajaminen pitää olla mahdollisimman vaivatonta, jotta menetelmästä saadaan paras mahdollinen hyöty. Tässä luvussa esitelläänkin projektissa käytettyjä testivetoiseen kehitykseen ja jatkuvaan integrointiin liittyviä työkaluja. Kaikki esitellyt työkalut ovat saatavilla ilmaiseksi ja pohjautuvat avoimeen lähdekoodiin. Tässä esitellyt työkalut eivät ole välttämättä parhaita vaihtoehtoja käyttötarkoituksiinsa, vaan ne esitellään esimerkkeinä saatavilla olevista työkaluista. Kohdissa 4.1-4.3 esitellään testivetoisen kehityksen mahdollistavat työkalut: Unity, CMock ja Ceedling. Kohdassa 4.4 esitellään jatkuvan integroinnin mahdollistava Jenkins-integrointipalvelinohjelmisto. Kohdassa 4.5 on lyhyt yhteenveto ja arvio käytetyistä työkaluista.

4.1 Unity

Unity on kohdan 2.1.2 mukainen yksikkötestikehys C-ohjelmointikielelle. Se soveltuu hyvin käytettäväksi sulautettujen järjestelmien kehittämiseen keveytensä takia. Unity koostuukin vain yhdestä kooditiedostosta ja muutamasta otsikkotiedostosta. Unity on myös siirrettävää koodia ja suunniteltu erityisesti käytettäväksi sulautetuissa sovelluksissa, joten se on helppo konfiguroida ja kääntää erilaisille prosessoreille. Konfiguroitavia asioita ovat esimerkiksi eri tietotyyppien koot, liukulukujen käyttö, muistin varaus ja testitulosten tulostustapa. [21, 28, 44]

Ohjelma 4.1 on yksinkertainen esimerkki Unity-yksikkötestikehyksellä toteutetusta testikoodista. SetUp-funktiota kutsutaan automaattisesti ennen jokaisen testitapauksen ajamista. TearDown-funktiota kutsutaan vastaavasti jokaisen testitapauksen ajamisen jälkeen. Näillä funktioilla voidaan tehdä alustus- ja purkutehtäviä, kuten resurssien varaamista ja vapauttamista tai testidatan alustamista. Esimerkissä funktiot eivät tee mitään. Test-alkuiset funktiot ovat varsinaisia testitapauksia, joissa kutsutaan testattavaa koodia ja tarkistetaan tulokset. Tulosten tarkistamiseen käytetään Unityn määrittelemiä TEST_ASSERT-alkuisia assertiomakroja. [44]

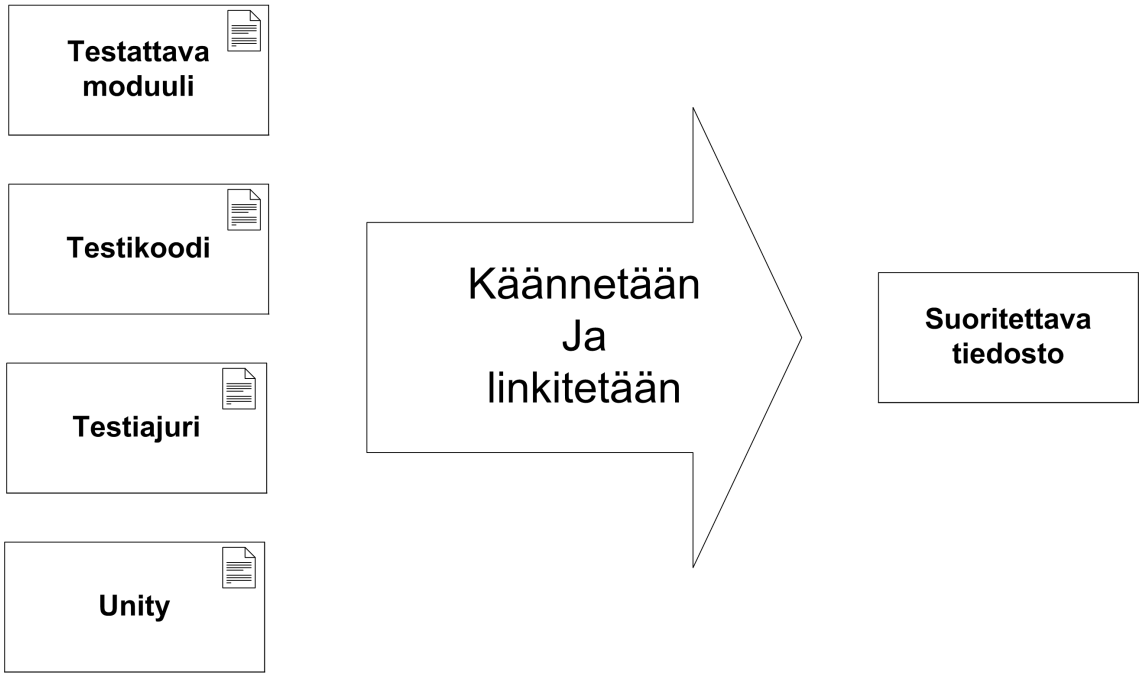
```
#include "Unity.h"
#include "Math.h"

void setUp(void)
{
    /* Setup code. */
}

void tearDown(void)
{
    /* Teardown code. */
}

void test_SquareRoot_ShouldHandleEasyNumbers(void)
{
    TEST_ASSERT_EQUAL_INT(0, SquareRoot(0));
    TEST_ASSERT_EQUAL_INT(1, SquareRoot(1));
}

void test_SquareRoot_ShouldHandleMoreDifficultNumbers(void)
{
    TEST_ASSERT_EQUAL_INT(3, SquareRoot(9));
    TEST_ASSERT_EQUAL_INT(10, SquareRoot(100));
}

void test_SquareRoot_ShouldHandleBigNumbers(void)
{
    TEST_ASSERT_EQUAL_INT(100, SquareRoot(10000));
    TEST_ASSERT_EQUAL_INT(300, SquareRoot(90000));
}
```

Ohjelma 4.1: Esimerkki testitiedostosta Unity-yksikkötestikehystä käyttäen.

Kuvassa 4.1 on esitetty suoritettavan tiedoston koostuminen eri osista Unity-yksikkötestikehystä käytettäessä. Kun testikoodi on kirjoitettu, pitää seuraavaksi luoda testiajurin koodi. Tämä voidaan tehdä automaattisesti Unityn mukana tulevalla skriptillä tai käsin. Tämän jälkeen testattava moduuli, testikoodi, Unity ja testiajuri käännetään jokainen erikseen. Lopuksi käännetyt osat linkitetään yhteen yhdeksi suoritettavaksi tiedostoksi, joka ajaa testit ja tulostaa testitulokset. Käytännössä siis jokaisesta testikooditiedostosta syntyy yksi suoritettava tiedosto. [28]

4.2 CMock

CMock on ohjelmistokehys, jolla voidaan automaattisesti generoida kohdan 2.1.4 mukaisia testisijaisia, erityisesti korvikeolioita. CMock käyttää staattista testisijais-

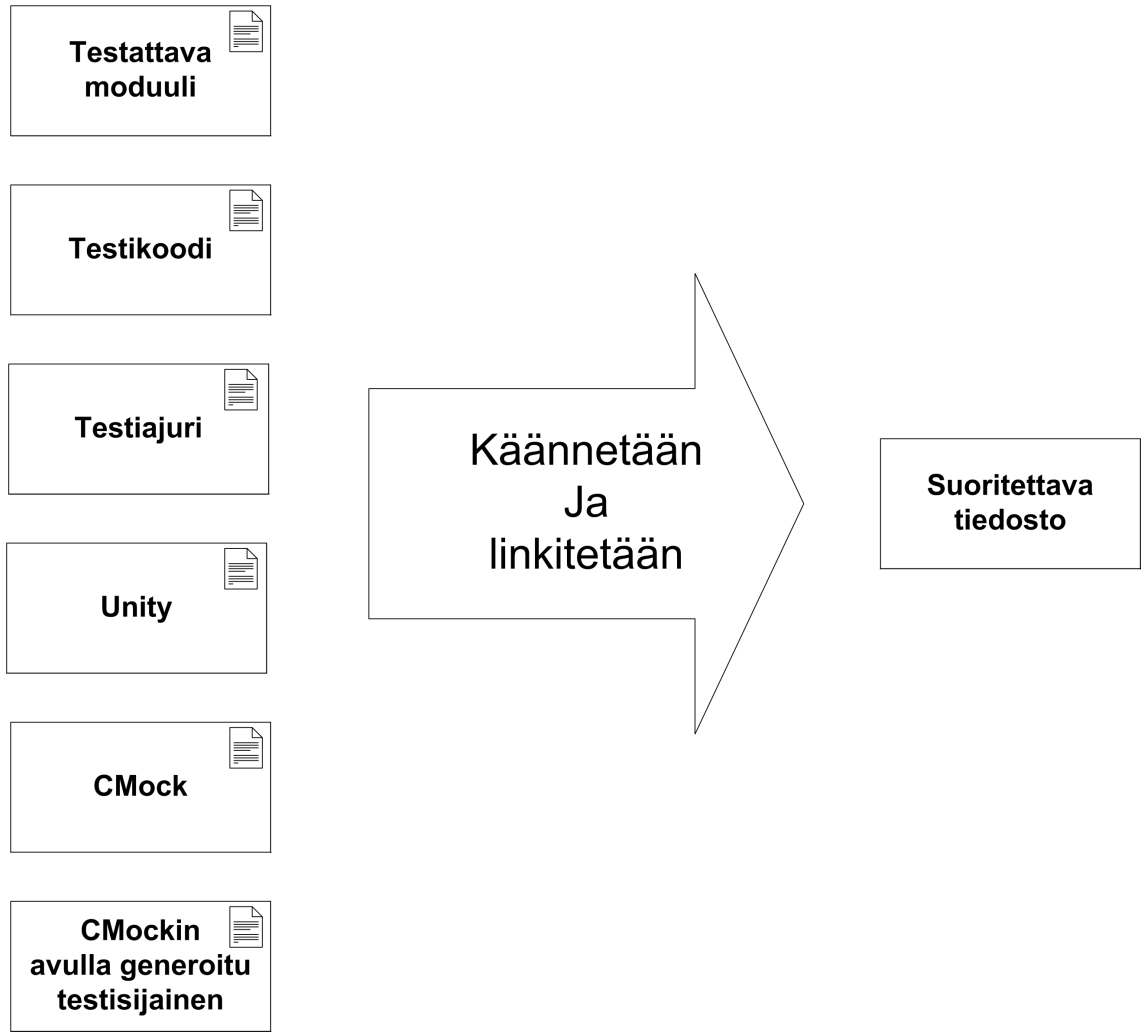


Kuva 4.1: Suoritettavan testin koostuminen Unity-yksikkötestikehystä käytettäessä.

ten generointia eli korvikeoliot generoidaan ennen koodin kääntämistä. Unityn tavoin myös CMock soveltuu hyvin käytettäväksi sulautettujen järjestelmien kehittämisessä, kunhan koodin generointi suoritetaan kehitysympäristössä. CMockin generoima koodi on siirrettävää ja helposti konfiguroitavaa C-koodia, jota voidaan kääntää erilaisille prosessoreille. Konfiguroitavia asioita ovat Unityn tavoin esimerkiksi tietotyyppien koot ja muistinvaraus. [28, 44]

CMock käyttää Ruby-skriptejä, joilla se generoi C-koodia. Ensin skriptit lukevat korvikeoliolla korvattavan moduulin otsikkotiedoston ja etsivät siitä funktioiden prototyyppejä. Nämä funktiot siis muodostavat korvattavan moduulin julkisen rajapinnan. Seuraavaksi skriptit generoivat korvattaville funktioille toteutukset ja funktiot korvikeolion konfigurointiin eli esimerkiksi paluuarvojen ja odotettujen parametrien asettamiseen. Lisäksi skriptit generoivat funktiot, joilla voidaan tarkistaa, että testattava koodi käytti rajapintaa oikein. Tarkistettavat asiat riippuvat siitä, miten korvikeolio konfiguroidaan. Generoitu koodi voidaan lopuksi kääntää ja linkittää oikean moduulin sijaan suoritettavaan tiedostoon. Kuvassa 4.2 on esitetty suoritettavan testin koostuminen eri osista, kun käytetään sekä Unity-yksikkötestikehystä että CMock-ohjelmistokehystä. [28, 44]

Ohjelma 4.2 on esimerkki CMock-ohjelmistokehyksellä luodun korvikeolion konfiguroinnista ja käytöstä yksikkötestauksen apuna. Esimerkissä on käytetty myös Unity-yksikkötestikehystä. Esimerkin alussa sisällytetään CMockin generoima otsikkotiedosto *MockParameters.h*, joka sisältää generoidun korvikeolion konfigurointirajapinnan. Esimerkki sisältää yhden testitapauksen, jonka alussa korvikeolio konfigu-



Kuva 4.2: Suoritettavan testin koostuminen Unity-yksikkötestikehystä ja CMock-ohjelmistokehystä käytettäessä.

roidaan. Korvikeolion konfigurointi tehdään kutsumalla korvikeolion konfigurointirajapinnan funktioita, joilla kuvataan odotetut funktiokutsut ja niiden parametrit. Lisäksi konfiguroidaan odotettujen funktiokutsujen palauttamat paluuarvot. Esimerkiksi funktiokutsu *SetAcceleration_Expect(0)* kertoo korvikeoliolle, että testattavan funktion pitäisi kutsua *SetAcceleration*-funktiota parametrilla 0. Samoin funktiokutsu *SetBrakes_ExpectAndReturn(100, 1)* kertoo, että testattavan funktion pitäisi kutsua *SetBrakes*-funktiota parametrilla 100 ja lisäksi korvikeolion tulee palauttaa paluuarvona 1. Testitapauksen lopuksi kutsutaan testattavaa funktiota *StopVehicle*. Tämän jälkeen korvikeoliolta tarkistetaan automaattisesti, että testattava koodi täytti kaikki asetetut odotukset eli teki vaaditut funktiokutsut. [28, 44]

Edellisessä kohdassa kuvatun korvikeolion konfigurointirajapinnan lisäksi korvikeolion käyttäytymiseen voidaan vaikuttaa CMockin asetuksilla sekä liitännäisillä (engl. plugin), jotka ovat CMockin toimintaa laajentavia ohjelmistokomponentteja. CMockin asetuksilla voidaan vaikuttaa esimerkiksi siihen, pitääkö testattavan koo-

din tehdä odotetut funktiokutsut tietyssä järjestyksessä. CMockin liitännäisistä voidaan esimerkkeinä mainita ignore ja callback. Ignore-liitännäinen lisää korvikeolion konfigurointirajapintaan funktioita, joilla voidaan asettaa odotettuja funktiokutsuja välittämättä parametrien arvoista tai siitä, kuinka monta kertaa funktiota kutsutaan. Ignore-liitännäistä on käytetty ohjelmassa 4.2. Callback-liitännäinen taas lisää korvikeolion konfigurointirajapintaan funktioita, joilla korvattavan rajapinnan funktioita on helppo korvata omilla funktioilla. Näillä omilla korvikefunktioilla voi tehdä juuri haluamansa tarkistukset. Tämä mahdollistaa siis korvikeolion konfiguroinnin vielä normaalia vapaammin ja joustavammin, mutta vaatii enemmän työtä. [44]

```
#include "Unity.h"
#include "VehicleControl.h"

/* Include mock header generated by CMock. */
#include "MockParameters.h"

void setUp(void)
{
}

void tearDown(void)
{
}

void test_StopVehicle_ShouldTurnOffAccelerationAndTurnOnBrakes(void)
{
    /* Set up expected calls to mock object. */

    /* Expect function call SetAcceleration(0) */
    SetAcceleration_Expect(0);

    /* Expect function call SetBrakes(100) which returns value 1. */
    SetBrakes_ExpectAndReturn(100, 1);

    /* Ignore function call SetBrakeLightOn() */
    SetBrakeLightOn_Ignore();

    /* Call function under test. */
    StopVehicle();
}
```

Ohjelma 4.2: Esimerkki testitiedostosta Unity-yksikkötestikehystä ja CMock-ohjelmistokehystä käytettäessä.

4.3 Ceedling

Vaikka edellä esitellyt Unity- ja CMock-ohjelmistokehykset helpottavatkin testausta huomattavasti, vaativat ne silti melko paljon manuaalista työtä. Näiden ohjelmistokehyksien käyttäjän pitää muun muassa generoida tarvittavat korvikeoliot CMockin tarjoamilla skripteillä, kääntää ja linkittää tarvittavat moduulit ja lopuksi ajaa testit, mikä voi olla hyvin työlästä. Lisäksi aina kun jotain kooditiedostoa muutetaan, pitää osa näistä vaiheista suorittaa uudelleen. Jotta TDD:n käyttö olisi mahdollista, pitäisi testit pystyä ajamaan helposti yhdellä komennolla.

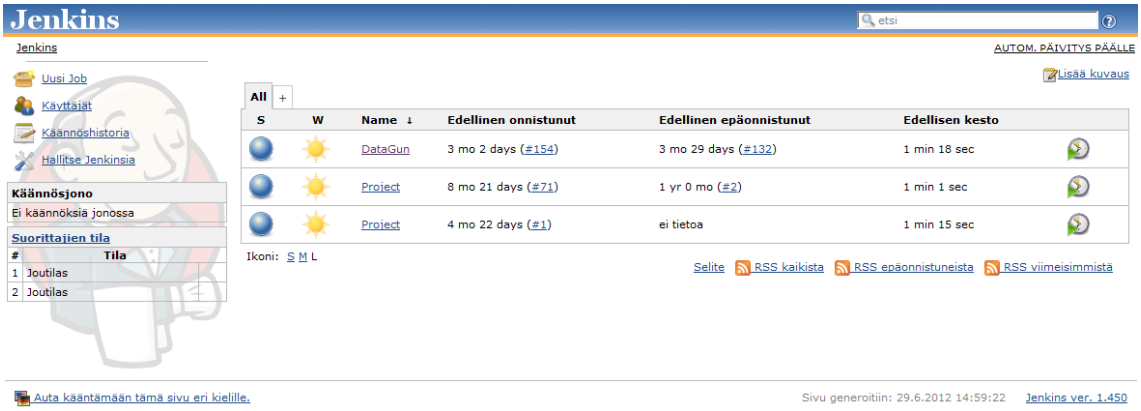
Ratkaisuksi edellä kuvattuun ongelmaan on kehitetty C-ohjelmointikielelle tarkoitettu käännösjärjestelmä Ceedling, joka on suunniteltu erityisesti TDD:n kanssa käytettäväksi. Ceedling on rakennettu Ruby-ohjelmointikieleen perustuvan Rake-käännöstyökalun pohjalle. Ceedling helpottaa Unityn ja CMockin käyttöä automatisoimalla korvikeolioiden generoinnin, tarvittavien moduulien kääntämisen, linkittämisen ja testien ajamisen. Käytännössä Ceedling osaa siis yhdellä komennolla kääntää ja ajaa kaikki projektin testit. Ceedling osaa tiedostojen riippuvuuksien perusteella päätellä, mitkä vaiheet pitää suorittaa uudelleen, kun joitain tiedostoja on muokattu. Ceedling osaa myös tarvittaessa ajaa vain muuttuneet testit. Nämä ominaisuudet mahdollistavat TDD:n vaatiman nopean kehityssyklin. Ceedlingin toimintaa voidaan myös laajentaa CMockin tapaan erilaisilla liitännäisillä, joiden avulla voidaan esimerkiksi tallentaa testitulokset XML-muodossa tiedostoon. [27, 28]

Ceedlingin toiminta perustuu konfiguraatiotiedostoon ja riippuvuuksien jäljittämiseen. Projektin konfiguraatiotiedostoon määritellään esimerkiksi polut lähdekoodihakemistoon ja testikoodihakemistoon, käytettävä kääntäjä ja linkittäjä, Unityn ja CMockin asetukset, käytettävät liitännäiset sekä muita asetuksia. Konfiguraation avulla Ceedling löytää testikooditiedostot ja pystyy selvittämään testien riippuvuudet testikoodin sisällyttämien otsikkotiedostojen perusteella. Jos testikoodissa sisällytetään Mock-alkuinen korvikeolion otsikkotiedosto, osaa Ceedling generoida sen automaattisesti CMockin avulla. Jos taas sisällytettyä otsikkotiedostoa vastaava kooditiedosto löytyy projektin lähdekoodihakemistosta, osaa Ceedling automaattisesti kääntää sen ja linkittää sen lopulliseen suoritettavaan tiedostoon. Ceedlingin käyttäjän ei siis tarvitse tehdä konfiguraation luonnin jälkeen muuta kuin lisätä uusia testejä, sillä Ceedling löytää ne automaattisesti ja osaa kääntää ja ajaa ne yhdellä komennolla. [27, 44]

4.4 Jenkins

Kohdassa 2.4 esitellyn jatkuvan integroinnin toteuttamiseksi kannattaa käyttää apuna jotain valmista työkalua. Eräs tähän tarkoitukseen sopiva työkalu on Jenkins. Jenkins on Java-ohjelmointikielellä toteutettu web-käyttöliittymällä varustettu in-

tegrointipalvelinohjelmisto. Jenkinsin hyviin puoliin kuuluvat esimerkiksi helppokäyttöisyys, joustavuus ja laaja yhteisö. Jenkinsin helppokäyttöisyys perustuu yksinkertaiseen web-käyttöliittymään, jolla ohjelmisto voidaan helposti konfiguroida. Myös Jenkinsin asennus on hyvin helppo ja nopea suorittaa. Joustavuus taas perustuu Jenkinsin lukuisiin liitännäisiin, joilla ohjelmisto voidaan räätälöidä hyvinkin monenlaisiin tarpeisiin sopivaksi. Liitännäiset liittyvät esimerkiksi versionhallintaohjelmistoihin, käännöstyökaluihin, testaukseen tai koodin staattiseen analyysiin. Jenkinsin suuri suosio ja laaja yhteisö varmistavat sen, että liitännäisiä on tarjolla lähes jokaiseen tarpeeseen. [43]

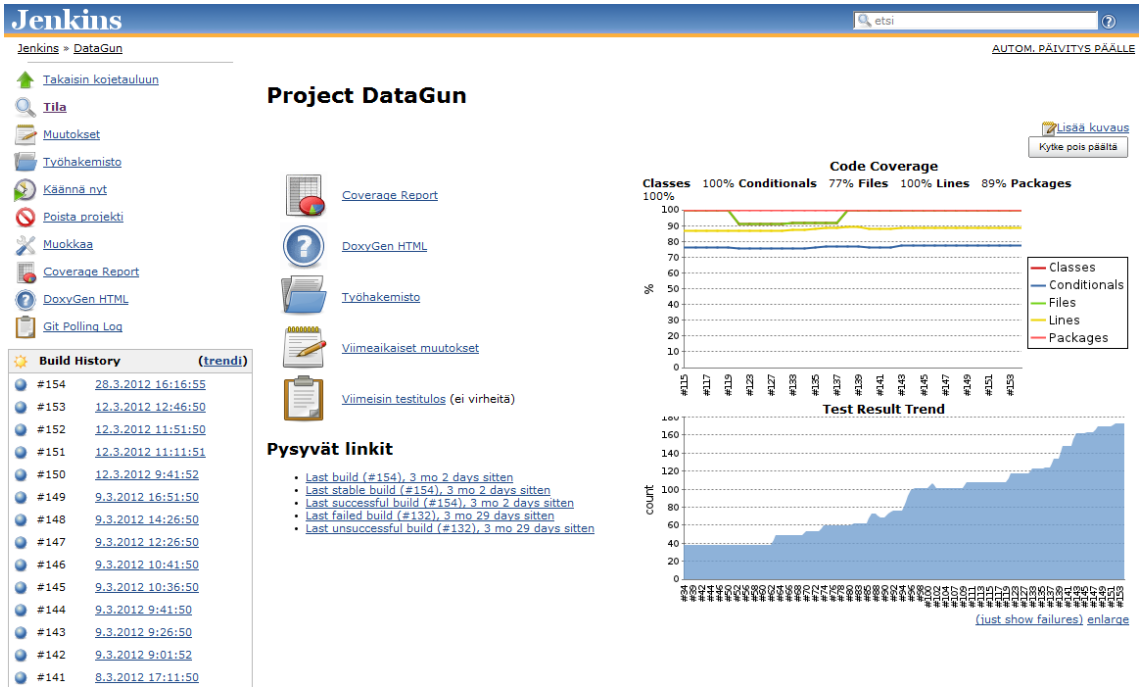


Kuva 4.3: Jenkinsin päänäkymä, jossa on nähtävissä kaikkien töiden tilat.

Jenkinsin päänäkymä on kuvan 4.3 mukainen. Päänäkymässä on nähtävissä kaikkien töiden (Job) tilat. Työt ovat prosesseja, jotka suorittavat tietyt ennalta konfiguroidut vaiheet ohjelmiston koostamiseksi ja testaamiseksi. Tyypillisesti työ sisältää vaiheet, joilla noudetaan koodi versionhallinnasta, käännetään koodi ja ajetaan testejä. Yleensä työt suoritetaan tietyin väliajoin tai kun versionhallintaan on viety muutoksia. Lisäksi töitä voidaan suorittaa manuaalisesti käyttöliittymän kautta.

Päänäkymästä voidaan siirtyä tarkastelemaan yhden työn tietoja. Kuvassa 4.4 on erään työn näkymä. Näkymän vasemmassa alakulmassa on nähtävissä työn suoritushistoria, jossa onnistuneet suoritukset esitetään sinisillä ja epäonnistuneet punaisilla palloilla. Näkymän vasemmassa yläkulmassa sijaitsevat linkit, joilla voidaan hallita työtä tai saada lisätietoa työn suorituksista. Näkymän oikean laidan sisältö riippuu työssä käytettävistä liitännäisistä. Kuvan työssä oikeassa laidassa on näkyvissä testikattavuusraportti ja yksikkötestien määrän kehittyminen.

Kuvan 4.4 suoritushistoriasta voidaan valita jokin työn suorituskerta ja tutkia sen tarkempia tietoja. Kuvassa 4.5 on erään suorituskerran näkymä. Näkymän sisältö riippuu työssä käytetyistä liitännäisistä. Tässä kuvassa näkyvissä ovat suorituskerran perustietojen lisäksi esimerkiksi versionhallintaan viedyt muutokset ja testitulokset. Näkymän vasemman reunan linkeistä pääsee tarkastelemaan suorituskerran tarkempia tietoja, kuten erilaisia lokitietoja ja raportteja. Myös näkymän vasemman



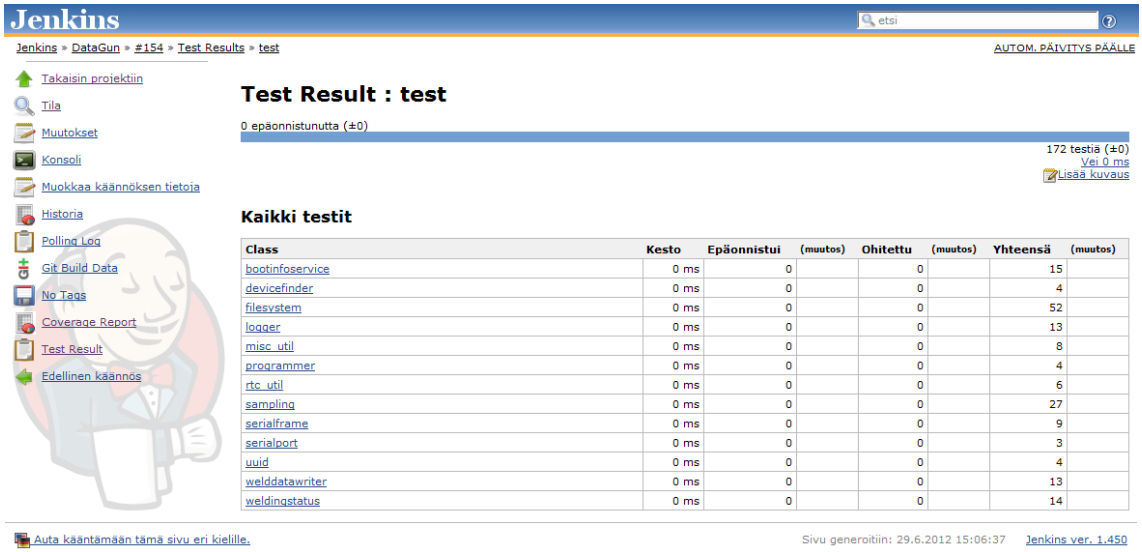
Kuva 4.4: Yhden työn näkymä Jenkinsissä.

reunan sisältö on riippuvainen työssä käytetyistä liitännäisistä. Näkymän vasemmassa reunassa voi olla esimerkiksi linkki yksikkötestiraporttiin, jollainen on nähtävissä kuvassa 4.6.

4.5 Yhteenveto ja arvio työkaluista

TDD:n käytön helpottamiseksi käytettiin siis Unity-yksikkötestikehystä, CMock-ohjelmistokehystä testisijaisten generointiin ja Ceedling-ohjelmistokehystä testien kääntämiseen ja ajamiseen. Unity soveltui tehtäväänsä erinomaisesti keveytensä, yksinkertaisuutensa ja helppokäyttöisyytensä ansiosta. Projektin aikana ei tullut

Kuva 4.5: Yhden suorituskerran näkymä Jenkinsissä.



Kuva 4.6: Jenkinsin yksikkötestiraportti.

esiin juurikaan Unityyn liittyviä ongelmia. Myös CMock toimi soveltui tehtäväänsä pääasiassa hyvin, vaikkei yltänytkään aivan samalle tasolle monipuolisuudessa ja helppokäyttöisyydessä kuin vastaavat työkalut oliopohjaisille ohjelmointikielille. CMockin avulla pystyttiin kuitenkin melko helposti generoimaan, konfiguroimaan ja ylläpitämään yksikkötesteissä tarvittuja testisijaisia. Otsikkotiedostojen analysoinnissa ja testisijaisten generoinnissa ilmeni välillä joitain ongelmia, mutta nekin pystyttiin ratkaisemaan tai kiertämään. CMockin avulla luoduista korvikeolioista tuli helposti liian tiukkoja testeissä tarkistettavien asioiden suhteen, mikä vaikeutti testien ylläpitoa. Tätä ongelmaa kuvataan tarkemmin kohdassa 5.2. Vaikkei Ceedling-ohjelmistokehyksen käyttäminen olisikaan ollut mitenkään välttämätöntä, helpotti se kuitenkin paljon TDD:n käyttöä automatisoimalla testisijaisten generointia, testien kääntämistä ja testien ajamista.

Erityisen hyvää TDD:n soveltamiseen käytetyissä työkaluissa oli niiden saumaton yhteensopivuus. Työkalut on alunperinkin suunniteltu käytettäväksi yhdessä. Työkalut täydensivät toisiaan erittäin hyvin. Hyvää oli myös työkalujen keveys ja soveltuvuus käytettäväksi sulautetun ohjelmiston kehittämiseen. Työkalut onkin kehitetty juuri TDD:n soveltamiseen sulautettujen ohjelmistojen kehittämisessä. Työkalut ovat myös helposti laajennettavissa liitännäisillä, vaikkei valmiita liitännäisiä ollutkaan kovin paljon saatavilla. Työkalut eivät silti olleet ominaisuuksien ja joustavuuden suhteen aivan yhtä hyviä kuin vastaavat työkalut oliopohjaisille kielille. Esimerkiksi graafinen testiajuri olisi ollut hyödyllinen, sillä se olisi mahdollistanut testitulosten tarkistuksen yhdellä silmäyksellä. Nyt testitulokset piti tarkistaa testiajurin tekstimuotoisista tulosteista.

Jatkuvan integroinnin toteuttamiseksi käytettiin Jenkins-integrointipalvelinta, joka soveltui erinomaisesti tehtäväänsä. Jenkinsin konfigurointi ja käyttö oli helppoa,

eikä sen käytössä ollut ongelmia. Jenkinsin käyttöliittymän suomennos on valitettavasti laadultaan heikko. Jenkinsin liitännäisvalikoima on erittäin kattava ja kaikkiin projektin tarpeisiin löytyikin sopiva liitännäinen.

5. KOKEMUKSIA TDD:N KÄYTÖSTÄ

Tässä luvussa käydään läpi kokemuksia TDD:n ja jatkuvan integroinnin soveltamisesta luvussa 3 kuvatussa projektissa. Tämän luvun sisältö perustuu pääasiassa tutkijan omiin kokemuksiin ja mielipiteisiin projektista. Vaikkei muita projektissa mukana olleita kehittäjiä varsinaisesti haastateltukaan, projektin aikana aiheeseen liittyvistä asioista keskusteltiin usein. Näin myös heidän kokemuksensa ovat vaikuttaneet tämän luvun sisältöön. Kohdassa 5.1 kuvataan TDD:n tarjoamia hyötyjä projektissa ja verrataan niitä kohdissa 2.2.2 ja 2.3.5 esitettyihin etuihin. Kohdassa 5.2 kuvataan projektin aikana esiin tulleita ongelmia TDD:n käytössä. Kohdassa 5.3 mietitään, miten TDD:n soveltamista kehitysprosessissa voisi vielä parantaa. Lopuksi kohdassa 5.4 on yhteenveto projektin kokemuksista.

5.1 Hyödyt

TDD:n käytöstä oli projektissa hyötyä monellakin eri osa-alueella. Projektissa havaitut hyödyt olivat pääasiassa yhteneviä kirjallisuudessa esitettyjen ja aiemmissa tutkimuksissa havaittujen hyötyjen kanssa. Monet hyödyistä olivat selkeitä ja niiden vaikutus oli huomattava. Osa hyödyistä taas oli hankalammin havaittavia ja vaikutukseltaan pienempiä.

Helpoimmin huomattavat hyödyt liittyvät koodin ulkoiseen laatuun, virheiden paikantamiseen ja virheiden korjaamiseen. TDD:tä käytettäessä useimmat virheet löydettiin hyvin nopeasti, jopa muutaman minuutin sisällä niiden syntymisestä. Löydetyt virheet oli myös nopea korjata, sillä virheen sijainti oli melko tarkkaan tiedossa. Virheiden löytämisen nopeuden tuoma hyöty vielä korostui sulautettua ohjelmistoa kehitettäessä, sillä tarve ohjelman debuggaamiseen kohdelaitteella väheneni huomattavasti. Monta komponenttia pystyttiinkin kehittämään lähes valmiiksi jo kehitysympäristössä lataamatta sitä välillä kohdelaitteeseen. Kehitys nopeutui, kun ohjelmaa ei tarvinnut ladata kovinkaan usein kohdelaitteelle ja testata manuaalisesti. Mahdollisuus kehittää komponentteja hyvin pitkälle ilman kohdelaitetta oli erityisen hyödyllistä tässä projektissa, sillä projektin kolmella kehittäjällä oli käytettävissään vain kaksi laitetta ja kaksi kääntäjän lisenssiä.

Koodin sisäiseen laatuun liittyviä hyötyjä on vaikeampi arvioida. Projektissa tuotettua koodia arvioimalla voidaan huomata, että komponentit ovat pieniä, niillä on selkeät vastuualueet ja niiden välisiä riippuvuuksia on melko vähän. TDD:n käyttä-

minen myös tuo kaikki komponenttien väliset riippuvuudet selkeästi esiin, sillä testeihin pitää sisällyttää kaikki riippuvuudet, joko oikeat toteutukset tai testisijaiset. Näin koodiin ei jää piilotettuja riippuvuuksia ja kaikilla riippuvuuksilla on selkeä tarkoitus. Koodi on muutenkin hyvälaatuista, sillä esimerkiksi funktiot ovat lyhyitä, muuttujat ja funktiot ovat kuvaavasti nimettyjä ja komponenttien rajapinnat on dokumentoitu kommenteilla. Vaikka projektissa tuotetun koodin sisäinen laatu näyttäisikin olevan hyvä, on vaikea arvoida miltä osin se on TDD:n käytön ansiota. Osa komponenteista toteutettin käyttämättä TDD:tä ja niiden laatu on suunnilleen yhtenevä verrattaessa sitä TDD:llä toteutettujen komponenttien laatuun. Saman laadun voi siis saavuttaa ilman TDD:n käyttöäkin, mutta TDD ohjaa ja osittain jopa pakottaa kirjoittamaan koodia, joka on sisäiseltä laadultaan hyvää ja helposti testattavaa.

Koodin kehittäminen TDD:tä käyttäen pakotti kirjoittamaan siirrettävää koodia, sillä koodin piti toimia sekä kehitysympäristössä että kohdelaitteessa. Laitteistoriippuvainen koodi piti siis eristää selvästi laitteistoriippumattomasta koodista. Tästä voi olla paljon hyötyä myöhemmin, sillä laitteistoriippumatonta osaa koodista on helppo käyttää tulevaisuudessa esimerkiksi jossain toisessa laitteessa. Myös kohdelaitteen ominaisuudet voivat muuttua tulevaisuudessa, jolloin on helppo muuttaa vain laitteistoriippuvaista osaa koodista tarpeen mukaan. Toki tässäkin tapauksessa saman voi saavuttaa myös ilman TDD:n käyttöä, mutta TDD pakottaa kirjoittamaan siirrettävää koodia ja eristämään laitteistoriippuvan koodin.

TDD:n käyttämisen tuloksena syntyi myös kattava kokoelma yksikkötestejä monille komponenteille. Osalle komponenteista yksikkötestit kirjoitettiin tosin vasta toteutuksen jälkeen. TDD:llä toteutettujen komponenttien testikattavuudet olivat hyvin korkeita, monilla komponenteilla lausekattavuus olikin yli 90%. Korkea testikattavuus ei toki yksinään takaa laadukasta testausta. Korkea testikattavuus voidaan myös saavuttaa kirjoittamalla yksikkötestit vasta toteutuksen jälkeen, mutta TDD:tä käytettäessä korkea testikattavuus syntyy luonnostaan. TDD:n käyttö, tai oikeastaan yksikkötestien kirjoittaminen, mahdollistaa myös hankalasti toistettavien virhetilanteiden simuloinnin. Tämä on erityisen hyödyllistä sulautettuja ohjelmistoja kehitettäessä, sillä monia laitteistoon liittyviä virhetilanteita on vaikea saada aikaan muuten.

TDD:n vaikutusta työn tuottavuuteen on hyvin vaikea arvioida, sillä se vaikuttaa tuottavuuteen hyvin monella tavalla. Tuottavuutta on myös erittäin hankala mitata luotettavasti ilman sopivia koejärjestelyjä, joten tällaisessa tapaustutkimuksessa se on käytännössä mahdotonta. TDD huononsi tavallaan tuottavuutta, sillä testien kirjoittamiseen kului melko paljon aikaa. Toisaalta saman verran aikaa saatettaisiin käyttää testien kirjoittamiseen ilman TDD:n käyttöäkin. Kysymys onkin lähinnä siitä, kirjoitetaanko yksikkötestejä ylipäätään. Myös sitä on hyvin vaikea

arvioida, paljonko TDD:n käyttö tai yksikkötestit ylipäätään vähensivät virheiden etsintään ja kohdelaitteella debuggaamiseen käytettyä aikaa. Lisäksi TDD:n käyttö lisää ylläpitoon käytettävää aikaa, sillä myös testikoodia pitää ylläpitää. Toisaalta kattava yksikkötestikokoelma nopeuttaa huomattavasti muutosten tekemistä ja regressiotestausta. TDD:n käytön nettovaikutusta tuottavuuteen onkin siis hyvin vaikea arvioida.

Myös psykologisia hyötyjä oli helppo huomata. TDD:n käyttö paransi huomattavasti luottamusta omaan koodiin ja siten vähensi stressiä. Koodia pystyi myös refaktoroimaan helpommin, sillä ei tarvinnut koko aikaa pelätä tekevänsä virheitä. TDD:n käyttö auttoi myöskin ymmärtämään ratkaistavan ongelman paremmin ja keskittymään paremmin juuri käsillä olevaan ongelmaan. Yleensäkin kehittäminen TDD:tä käyttäen oli paljon miellyttävämpää.

TDD:n käytön ja sen myötä syntyneiden yksikkötestien ansiosta myös jatkuvan integroinnin käyttämisestä oli selkeää hyötyä. Ohjelmisto pysyi enimmäkseen toimivassa tilassa ja ongelmat huomattiin helposti, ainakin TDD:llä kehitettyjen komponenttien osalta. Jatkuvan integroinnin ansiosta huomattiin esimerkiksi joitain ohjelmiston siirrettävyyteen liittyviä ongelmia, sillä integrointipalvelimen ohjelmistoympäristö ja esimerkiksi kääntäjä olivat hieman erilaiset kehittäjien normaaliin kehitysympäristöön verrattuna. Jatkuva integrointi mahdollisti myös esimerkiksi yksikkötestien määrän ja kattavuuden kehittymisen seuraamisen projektin aikana.

5.2 Ongelmat

Vaikka edellisessä kohdassa löytyikin monia TDD:n käyttöön liittyviä hyötyjä, löytyi myös monia ongelmia. Suurin TDD:n käyttöön liittyvä ongelma on menetelmän vaikeus. Vaikka se on idealtaan hyvinkin yksinkertainen, on sen käyttäminen tehokkaasti hyvin vaikeaa ja vaatii kehittäjältä paljon kokemusta ja harjoittelua. Kohdassa 2.1.1 kuvattiin hyvän yksikkötestin ominaisuuksia, käytännössä tällaisen hyvän yksikkötestin kirjoittaminen on kuitenkin erittäin vaikeaa. Huonoista yksikkötesteistä onkin enemmän haittaa kuin hyötyä, sillä aikaa tuhlaantuu sekä niiden kirjoittamiseen että erityisesti niiden ylläpitoon. Testien hyvä laatu on siis erittäin tärkeää. Erityisesti TDD:tä käytettäessä on myös tärkeää osata testata oikeita asioita ja vieläpä sopivassa järjestyksessä, jotta kehitys etenisi luontevasti ja tehokkaasti. Tämäkin vaatii paljon menetelmän harjoittelua ja erilaisten komponenttien kehittämistä TDD:tä käyttäen.

Yksi yleinen ongelma liittyen tuotettuihin yksikkötesteihin oli testien joustamattomuus. TDD:tä käytettäessä oli helppo kirjoittaa testejä, jotka määrittelivät toteutetun komponentin toiminnan liian tarkasti. Käytännössä testit siis pakottivat komponentin sisäisen toteutuksen tietynlaiseksi sen sijaan, että testit pakottaisivat komponentin ulkoisen käyttäytymisen tietynlaiseksi. Jos komponentin sisäistä

toteutusta muutettiin, piti myös testejä muuttaa. Tämä lisäsi testien ylläpitoon vaadittavaa työmäärää. Osasyyllinen ongelmaan oli projektissa käytetty CMock-ohjelmistokehys, jota käytettiin testisijaisten generointiin. CMockia käytettäessä luoduista testisijaisista tuli helposti liian tiukkoja, sillä ne tarkistivat testisijaisen rajapintaan tehtyjen funktiokutsujen oikean määrän, järjestyksen ja parametrien arvot. Monissa tapauksissa tarkistukset olivat aivan liian tarkkoja ja määrittelivät komponentin sisäistä toteutusta. Ongelman kiertämiseksi käytettiin paljon CMockin ignore- ja callback-liitännäisiä, joilla pystyttiin tekemään testisijaisista joustavampia. Erityisesti callback-liitännäisen avulla pystyttiin tekemään juuri tilanteeseen sopivia testisijaisia, mutta vaadittu työmääräkin oli suurempi. Tämänkin ongelman merkitys pienenee kehittäjän oppiessa paremmin käyttämään TDD:tä ja käytössä olevia työkaluja.

TDD:n soveltaminen sulautetun ohjelmiston kehittämisessä tuo mukanaan vielä omat ongelmansa. Kehitysympäristön ja kohdelaitteen väliset eroavaisuudet esimerkiksi prosessorin, kääntäjän, kirjastojen ja käytettävien otsikkotiedostojen osalta aiheuttivat ongelmia ja muutenkin hankaloittivat TDD:n käyttöä. Projektissa havaittiin esimerkiksi ristikääntäjän tekemiin optimointeihin liittyviä ongelmia, joiden jäljittäminen oli hyvin vaikeaa ja vaatii aikaavievää debuggausta kohdelaitteessa. Nämä ongelmat liittyivät kylläkin osittain vanhaan koodiin, jota ei oltu kehitetty TDD:tä käyttäen ja jolle ei myöskään ollut mitään yksikkötestejä. Muutenkin projektissa hyödynnetty vanha ja vaikeasti testattava koodi hankaloitti TDD:n käyttöä, sillä esimerkiksi monia vanhan koodin otsikkotiedostoja ei pystytty käyttämään lainkaan yksikkötesteissä niiden sisältämän kääntäjä- ja laitteistokohtaisen koodin takia. Ongelman kiertämiseksi jouduttiin tekemään erilaisia rajapintoja uuden koodin ja vanhan koodin välille. Tämä oli tietysti työlästä, mutta toisaalta hyödyllistäkin, sillä kääntäjä- ja laitteistoriippuvainen koodi saatiin paremmin eristettyä alustariippumattomasta koodista.

Käytettäessä TDD:tä sulautetun ohjelmiston kehittämiseen ongelmana on myös sopivien työkalujen heikompi saatavuus. Ohjelmistokehityksessä usein käytetyille oliopohjaisille kielille, kuten Java, C++ ja C#, on saatavilla monia laadukkaita, monipuolisia ja helppokäyttöisiä työkaluja, joilla TDD:n käyttö on helppo aloittaa. Projektissa käytetylle C-ohjelmointikielelle tarjonta on selvästi heikompaa, vaikka monia työkaluja onkin saatavilla. Ne eivät kuitenkaan ole yhtä monipuolisia ja helppokäyttöisiä, kuin vastaavat korkeamman tason kielten työkalut. Osittain työkalujen heikompi laatu ja rajoittuneisuus liittyvät C-ohjelmointikielen rajoituksiin, sillä esimerkiksi muissa kielissä usein käytettävä testisijaisten ajonaikainen generointi olisi hyvin vaikea toteuttaa. Toisaalta myös TDD:n käyttö sulautettujen ohjelmistojen kehityksessä on paljon vähäisempää, joten kiinnostus työkalujen kehittämiseenkin on pienempää.

5.3 Parannettavaa

TDD:n ja jatkuvan integroinnin soveltaminen projektissa ei ollut täydellistä, vaan sitä voisi vielä parantaa monella tavalla tulevissa projekteissa. Suurimmat hyödyt voitaisiin saada parantamalla ja laajentamalla TDD:n käyttöä. Nyt vain osa projektissa kehitetyistä komponenteista kehitettiin TDD:tä käyttäen, joten tulevaisuudessa yhä suurempi osa koodista voitaisiin kehittää TDD:n avulla. Aivan kaikkea ei tietenkään ole järkevää toteuttaa TDD:n avulla, mutta suurin osa komponenteista voitaisiin hyvinkin kehittää TDD:tä käyttäen. Nyt TDD:tä ei juurikaan käytetty tehtäessä muutoksia vanhaan koodiin, joten tältäkin osin TDD:n hyödyntämistä voisi laajentaa. Tehtäessä muutoksia vanhaan koodiin, voitaisiin muutokset tehdä TDD:tä käyttäen. Testien kirjoittaminen vanhalle koodille, jota ei ole suunniteltu testattavaksi, voi olla erittäin työlästä. Testikattavuus saataisiin kuitenkin nousemaan myös vanhan koodin osalta, mikä helpottaisi ylläpitoa ja esimerkiksi regressiotestausta.

TDD:n käytön harjoittelu olisi myös hyödyllistä. Projektissa huomattiin, että TDD:n käyttäminen tehokkaasti ja tuottavasti on vaikeaa. Kuitenkin menetelmää käytettäessä ja kokemuksen kertyessä, voi huomata tuottavuuden kasvavan ja myös tuotetun koodin ja testien laadun parantuvan. TDD:n käyttöä on luultavasti vaikea opettaa esimerkiksi koulutuksilla. Lyhyellä koulutuksella voidaan toki opettaa TDD:n perusideat, mutta TDD:n kunnollinen osaaminen vaatii kehittäjältä paljon harjoittelua TDD:n käytöstä oikeiden projektien parissa. TDD:n opetteluun ei siis ole mitään helppoa tapaa, vaan osaaminen tulee pitkäjänteisen työn tuloksena. Koska TDD:n käyttö on vaikeampaa sulautettuja ohjelmistoja kehitettäessä, TDD:n käytön harjoittelu kannattaa todennäköisesti aloittaa jollain helpommalla sovellusalueella.

Kohdassa 2.3.2 esiteltyä sulautettujen ohjelmistojen kehityssykliä hyödynnettiin projektissa, mutta ei kovin kattavasti. Kehityssyklistä oli käytössä vain osa siihen kuuluvista vaiheista. Esimerkiksi yksikkötestejä ajettiin vain kehitysympäristössä, kun niitä olisi kannattanut ajaa myös kohdelaitteella. Jatkossa kannattaisi siis pyrkiä ajamaan yksikkötestejä aina välillä myös kohdelaitteella, vaikka se onkin vähän hankalampaa kuin yksikkötestien ajaminen kehitysympäristössä. Myöskään minkäänlaisia automaattisia hyväksymistestejä ei tehty, ei kehitysympäristössä eikä kohdelaitteella ajettavia. Kehitysprosessia kannattaisi siis parantaa tältä osin hyödyntämällä kohdassa 2.2.1 kuvattua hyväksymistestivetoista kehitystä (ATDD). Automaattisia hyväksymistestejä voitaisiin ajaa joko kohdelaitteella tai kehitysympäristöön luodussa simulaatioympäristössä.

Vaikka jatkuvaa integrointia hyödynnettiinkin projektissa jo melko kattavasti, voisi myös sen käyttöä kehittää edelleen. Ensimmäisenä kannattaisi ainakin pyrkiä

tekemään myös ristikäännös automaattisesti osana jatkuvaa integrointia. Näin voitaisiin varmistua siitä, että ohjelmisto pystytään aina kääntämään myös ristikääntäjällä eikä vain kehitysympäristön kääntäjällä. Samalla saataisiin myös kohdelaitteelle ladattava käännetty ohjelmisto. Tämän parannuksen jälkeen voitaisiin ottaa edellisessä kappaleessa mainittuja parannuksia mukaan myös jatkuvaan integrointiin. Eli hyväksymistestejä tai kohdelaitteella ajettavia yksikkötestejä voitaisiin ajaa osana jatkuvaa integrointia. Jatkuvaa integrointia voitaisiin kehittää myös ottamalla mukaan esimerkiksi koodin staattista analyysiä.

5.4 Yhteenveto kokemuksista

TDD:n ja jatkuvan integroinnin käytön havaittiin tuovan monia erilaisia hyötyjä. Osa saavutetuista hyödyistä oli selkeitä ja helposti havaittavia, osa taas vaikutukseltaan pienempiä ja vaikeammin havaittavia. Hyötyjä saatiin myös monella eri ohjelmistokehityksen osa-alueella. Hyötyjä saatiin esimerkiksi ohjelmiston ulkoiseen laatuun, koodin sisäiseen laatuun ja testaukseen liittyen. Maininnan arvoisia olivat myös TDD:n positiiviset vaikutukset työn miellyttävyyteen. TDD:n käyttöön liittyviä ongelmiakin toki löytyi. Osa ongelmista liittyi yleisesti menetelmään ja erityisesti sen vaikeuteen, sillä menetelmä on nopea oppia, mutta sen hallitseminen hyvin vaatii paljon harjoittelua. Osa ongelmista taas liittyi TDD:n soveltamiseen sulautetun ohjelmiston kehittämiseen ja käytettyihin työkaluihin. Ongelmia aiheutti myös projektissa suuressa roolissa ollut vaikeasti testattavissa oleva vanha koodi. Monien ongelmien vaikutuksia voidaan kuitenkin vähentää ja saavutettuja etuja kasvattaa harjoittelemalla menetelmän käyttöä ja soveltamalla sitä laajemmin. TDD:n ja jatkuvan integroinnin käyttöä ohjelmistokehityksessä voidaankin siis vielä parantaa selvästi tutkittuun projektiin verrattuna. Yhteenvetona voidaan todeta TDD:n ja jatkuvan integroinnin käytön olleen erittäin hyödyllistä, vaikkei ongelmiltakaan vältytty.

6. JOHTOPÄÄTÖKSET

Tässä luvussa kuvataan ja arvioidaan tutkimuksella saavutettuja tuloksia. Luvun alussa kohdassa 6.1 esitellään tutkimuksen tärkeimmät tulokset ja verrataan niitä kohdassa 1.2 esitettyihin tutkimusongelmaan ja tutkimuskysymyksiin. Tämän jälkeen kohdassa 6.2 arvioidaan näitä tuloksia sekä tutkimusta ja työtä yleisesti. Lopuksi kohdassa 6.3 annetaan ehdotuksia jatkotutkimusaiheiksi.

6.1 Tutkimustulokset

Tutkimuksen päätavoitteena oli löytää vastaus kohdassa 1.2 asetettuun tutkimusongelmaan: miten TDD:n käyttöönotolla voidaan parantaa sulautettujen sovellusten ohjelmistokehitysprosessia. Tutkimusongelman laajuuden vuoksi tutkimusongelma jaettiin useaan pienempään tutkimuskysymykseen, joihin pyrittiin löytämään vastauksia. Vastauksia tutkimuskysymyksiin etsittiin tutkimuksen ensimmäisessä vaiheessa kirjallisuustutkimuksen avulla. Näin saatiin tutkimuskysymyksiin vastauksia aiempien tutkimusten ja kokemusten valossa. Tutkimuksen toisessa vaiheessa vastauksia etsittiin tapaustutkimuksen muodossa tutkimalla TDD:n käyttöä sulautetun tiedonkeruulaitteen ohjelmistoa kehitettäessä. Näin saatiin vastauksia myös käytännön kokemusten kautta ja pystyttiin varmistamaan ensimmäisessä vaiheessa löydettyjä vastauksia.

Ensimmäisessä tutkimuskysymyksessä käsiteltiin aiheeseen liittyviä ongelmia: mitä ongelmia on TDD:n käyttämisessä sulautettujen sovellusten kehittämisessä? Kohdassa 2.3.1 esiteltiin aiemmissa tutkimuksissa havaittuja ongelmia. Suurimmat ongelmat liittyivät testien ajamiseen kohdelaitteistossa, sillä se voi olla hankalaa ja hidasta. Näitä ongelmia voidaan pienentää kehittämällä ja ajamalla testejä ensin kohdelaitteen sijaan kehitysympäristössä. Tämä menetelmä tuo kuitenkin mukanaan omat ongelmansa, jotka liittyvät kohdelaitteen ja kehitysympäristön välisiin eroihin. Luvussa 5 käsitellään käytännön kokemuksien kautta havaittuja ongelmia, joita olivat esimerkiksi menetelmän vaikeus, testien ylläpitoon kuluva aika, vaikeasti testattavissa oleva vanha koodi ja käytettyihin työkaluihin liittyvät ongelmat.

Toisessa tutkimuskysymyksessä käsiteltiin vastaavasti TDD:llä saavutettavia hyötyjä: mitä hyötyjä saadaan TDD:n käyttämisestä sulautettujen sovellusten kehittämisessä? Myös hyötyjä käsiteltiin ensin aiempien tutkimusten pohjalta kohdissa 2.2.2 ja 2.3.5. Näiden tutkimusten mukaan TDD:n avulla on mahdollista saavuttaa

monenlaisia hyötyjä monilla eri ohjelmistokehityksen osa-alueilla. Hyödyt liittyivät esimerkiksi koodin ulkoiseen laatuun, koodin sisäiseen laatuun, testaukseen, tuottavuuteen, dokumentointiin sekä psykologisiin ja sosiaalisiin vaikutuksiin. Lisäksi osa hyödyistä liittyy erityisesti sulautettujen sovellusten kehittämiseen. Nämä hyödyt ovat lyhyesti ilmaistuna laitteistoon liittyvien riskien ja kulujen vähentyminen ja koodin parempi siirrettävyys. Esimerkiksi tarve koodin debuggaamiseen kohdelaitteessa vähenee selvästi. Hyötyjä käsiteltiin käytönnön kokemusten kautta luvussa 5 ja havaitut hyödyt olivat pääasiassa yhteneviä aiempien tutkimusten kanssa. Osa hyödyistä oli selkeämmin havaittavissa ja osa taas heikommin.

Kolmannessa tutkimuskysymyksessä käsiteltiin TDD:n käytön asettamia vaatimuksia projektin henkilöstölle: mitä TDD:n käyttäminen vaatii projektin henkilöstöltä? TDD:n käyttäminen vaatii tietysti menetelmän ja käytettävien työkalujen opettelua. TDD:n perusteiden lisäksi tutkimuksen ensimmäisessä vaiheessa löydettiin hyvin paljon asioita, joita kehittäjän olisi hyödyllistä ymmärtää. Tällaisia asioita ovat esimerkiksi testisijaisten käyttäminen ja toteuttaminen, erilaiset suunnitteluperiaatteet ja sulautettuihin ohjelmistoihin liittyvät erityispiirteet. Näitä asioita on käsitelty luvussa 2. Myös tutkimuksen toisessa vaiheessa havaittiin menetelmän olevan vaativa. Menetelmä on yksinkertaisen ideansa ansiosta nopea oppia, mutta sen käyttäminen tehokkaasti ja tuottavasti vaatii paljon kokemusta ja harjoittelua. Menetelmän käyttö on vielä hankalampaa sulautettuja ohjelmistoja kehitettäessä, joten TDD:n käytön harjoittelu kannattaakin todennäköisesti aloittaa jollain helpommalla sovellusalueella. Tutkimuksen toiseen vaiheeseen liittyviä kokemuksia löytyy luvusta 5. TDD:n käyttäminen vaatii siis henkilöstöltä motivaatiota ja aikaa uuden menetelmän opetteluun.

Neljännessä tutkimuskysymyksessä käsiteltiin arkkitehtuuria ja suunnitteluperiaatteita: mitä TDD:n käyttäminen vaatii sovelluksen arkkitehtuurilta ja suunnitteluperiaatteilta? TDD:n käytön asettamia vaatimuksia kehitettävän ohjelmiston arkkitehtuurille ja suunnitteluperiaatteille selvitettiin tutkimuksen ensimmäisessä vaiheessa. Arkkitehtuuria ja suunnitteluperiaatteita käsiteltiin kohdassa 2.3.4. Vaikka TDD:n käyttö ohjaakin sovelluksen arkkitehtuuria ja suunnittelua modulaariseen ja testattavaan suuntaan, kannattaa ohjelmiston suunnittelussa noudattaa muutamia hyväksi havaittuja periaatteita. Tällaisia periaatteita ovat esimerkiksi ohjelmiston jakaminen moniin pieniin komponentteihin, riippuvuuksien minimointi sekä alustariippuvaisen koodin eristäminen. Tutkimuksen toisessa vaiheessa pyrittiin noudattamaan näitä hyviä periaatteita toteutettavaa ohjelmistoa suunniteltaessa. Toteutetun ohjelmiston arkkitehtuuria käsiteltiin kohdassa 3.3.

Viidennessä tutkimuskysymyksessä käsiteltiin työkaluja: minkälaisia työkaluja TDD:n käyttäminen vaatii? Jotta TDD:n hyödyntäminen olisi tehokasta, pitää kohdissa 2.2.1 ja 2.3.2 esitellyn TDD:n kehityssyklin käyttö olla nopeaa ja vaivatonta.

Käytettävien työkalujen tärkein tehtävä on siis mahdollistaa kehityssyklin vaivaton käyttö. Hyvät työkalut helpottavat kohdassa 2.1.1 kuvattujen hyvien yksikkötestien kirjoittamista. Tutkimuksen toisessa vaiheessa käytettyjä työkaluja esiteltiin ja arvioitiin luvussa 4. Käytetyt työkalut osoittautuivat hyvin tehtäviinsä sopiviksi ja helpottivat huomattavasti TDD:n käyttöä.

Kuudennessa tutkimuskysymyksessä käsiteltiin parhaita käytäntöjä ja menetelmiä: mitkä ovat parhaat käytännöt ja menetelmät käytettäessä TDD:tä? Tähän kysymykseen on vaikea antaa tyhjentävää vastausta, sillä parhaita käytäntöjä ja menetelmiä on käsitelty lähes koko työssä ja niitä löytyi monia. Eräs maininnan arvoinen menetelmä on jatkuva integrointi. Vaikkei jatkuva integrointi aivan suoraan liitykään testivetoiseen kehitykseen, havaittiin se erittäin hyödylliseksi sitä tukevaksi menetelmäksi. Jatkuvaa integrointia käsiteltiin kohdissa 2.4, 3.5 ja 4.4

Yhdessä nämä tutkimuskysymyksiin löydetyt vastaukset muodostavat vastauksen kohdassa 1.2 asetettuun tutkimusongelmaan: miten TDD:n käyttöönotolla voidaan parantaa sulautettujen sovellusten ohjelmistokehitysprosessia. Tutkimuksen ensimmäisessä vaiheessa TDD:tä tutkittiin kirjallisuuden ja aiempien tutkimusten avulla. Näin saatiin tärkeää tietoa menetelmästä, jotta sen käyttöönotto tutkimuksen toisessa vaiheessa onnistuisi mahdollisimman hyvin. Tutkimuksen toisessa vaiheessa testivetoista kehitystä ja jatkuvaa integrointia hyödynnettiin onnistuneesti sulautetun tiedonkeruulaitteen ohjelmiston kehityksessä. Vaikka TDD:n käyttöön liittyen joitain ongelmia ilmenikin, oli kokemus kuitenkin kokonaisuutena erittäin positiivinen ja TDD todettiin hyödylliseksi kehitysmenetelmäksi.

6.2 Tulosten arviointi

Tutkimus onnistui kokonaisuutena erittäin hyvin, sillä tutkimusongelmaan ja kaikkiin tutkimuskysymyksiin löytyi hyvin vastauksia. Tieteelliseen tutkimukseen kuuluu kuitenkin olennaisena osana myös tulosten ja tutkimusprosessin arviointi. Tutkimuksen ensimmäinen osa oli luonteeltaan kirjallisuustutkimus, jossa lähdemateriaalin laatu, kattavuus ja tarkoituksenmukaisuus on suuressa roolissa. Käytettyjen lähteiden vakuuttavuuden takaamiseksi pyrittiin tutkimuksessa suosimaan mahdollisimman arvovaltaisia lähteitä. Käytännössä siis lähdemateriaalia valittaessa suosittiin kirjoja ja tieteellisissä julkaisuissa julkaistuja artikkeleita sekä vältettiin websivuja. Lähdemateriaalia pyrittiin valitsemaan kattavasti ja vältettiin tukeutumasta liikaa yksittäisiin lähteisiin. Yleisesti TDD:tä tutkittaessa tämä onnistuikin hyvin, sillä materiaalia löytyi hyvin paljon. Ongelmana olikin sopivimpien lähteiden valinta. Myöskään yksittäisten lähteiden tutkimiseen ei jäänyt aina kovinkaan paljoa aikaa. Toisaalta TDD:n soveltamista sulautettujen ohjelmistojen kehittämiseen on tutkittu selvästi vähemmän, joten siihen liittyvää hyvää lähdemateriaalia oli vaikeampi löytää. Sulautettujen ohjelmistojen osalta tutkimus perustuukin pääosin vain muu-

tamaan lähteeseen. Kokonaisuutena lähdemateriaali oli kuitenkin riittävän kattava ja monipuolinen.

Tutkimuksen toinen vaihe suoritettiin tapaustutkimuksena tutkimalla TDD:n käyttöä yhdessä ohjelmistokehitysprojektissa. Tapaustutkimuksen avulla pystyttiin vahvistamaan kirjallisuustutkimuksen aikana syntyneitä huomioita, joten se täydensi hyvin kirjallisuustutkimusta. Havainnot olivat pääasiassa yhteneviä kirjallisuustutkimuksen tulosten kanssa. Koska tapaustutkimus keskittyi vain yhden projektin tutkimiseen, ei sen tulosten pohjalta voida tehdä kovinkaan hyvin yleistyksiä. Tapaustutkimuksen osalta pitää myös muistaa, että tulokset ovat syntyneet hyvin pitkälle tutkijan havaintojen ja arvioiden pohjalta. Tutkijan mielipiteet ovat siis voineet vaikuttaa tuloksiin. Toinen huomioitava seikka on se, että tapaustutkimuksessa käytettiin vain kvalitatiivisia menetelmiä eikä kvantitatiivisia menetelmiä. Kvantitatiivisten menetelmien käyttö olisi vaatinut laajempaa aineistoa, esimerkiksi kyselyn tuloksia. Tämä ei kuitenkaan ollut käytännöllistä projektin pienen henkilömäärän takia. Tapaustutkimuksessa yhtenä ongelmana oli myös selkeän vertailukohdan puuttuminen, mikä hankaloitti TDD:n vaikutusten arviointia.

Tärkeää on myös arvioida tutkimuksessa saavutettujen tulosten merkitystä. Tässä tutkimuksessa ei juurikaan tuotettu täysin uutta tietoa, mutta tutkimuksessa pystyttiin vahvistamaan aiemmissa tutkimuksissa tehtyjä havaintoja. Toisaalta TDD:n soveltamista sulautettujen ohjelmistojen kehityksessä ei ole tutkittu vielä kovinkaan paljon, joten lisätutkimuksesta on hyötyä. Merkittävimmät hyödyt työstä on saanut varmasti tutkija itse karttuneen kokemuksen ja osaamisen myötä. Tutkimustuloksia voidaan hyödyntää myös kohdeyrityksessä levittämällä tietoa aiheesta ja ottamalla TDD käyttöön myös muissa sulautettujen ohjelmistojen kehitysprojekteissa. Tutkija järjestääkin tutkimuksen aiheesta koulutusta yrityksen muille työntekijöille.

6.3 Jatkotutkimusehdotukset

Vaikka kaikkiin tutkimuskysymyksiin saatiinkin vastauksia, jäi tutkimuksesta vielä joitain avoimia kysymyksiä. Esimerkiksi tutkimuksen toisessa vaiheessa eli tapaustutkimuksessa ei ehditty kokeilla kaikkia kirjallisuustutkimuksessa käsiteltyjä menetelmiä. Kokeilematta jääneitä menetelmiä olivat esimerkiksi kohdassa 2.2.1 esitelty hyväksymistestivetoinen kehitys (ATDD) ja kohdassa 2.2.3 esitelty Behaviour-Driven Development (BDD). Näistä jälkimmäisen ideoita kylläkin hyödynnettiin testien nimeämiskäytännöissä, mutta kattavammin menetelmään ei perehdytty. Tapaustutkimuksessa ei myöskään tutkittu yksikkötestien ajamista kohdelaitteella, vaan yksikkötestit ajettiin aina kehitysympäristössä.

Edellä mainittujen menetelmien lisäksi hyvä jatkotutkimuksen kohde olisi TDD:n soveltaminen tiedonkeruusovellusten lisäksi erilaisilla ja haastavammilla sovellusalueilla. Hyvä tutkimuksen aihe voisi olla esimerkiksi TDD:n soveltaminen turval-

lisuuskriittisten järjestelmien kehittämisessä. Myös TDD:tä ja sulautettujen ohjelmistojen kehitystä tukevien työkalujen arviointi, vertailu ja kehittäminen olisi hyödyllistä.

LÄHTEET

[1] Dave Astels. A New Look at Test-Driven Development, 2005.

[2] Martin R. Bakal, Jennifer Althouse, and Paridhi Verma. Continuous integration in agile development: How agile methods, continuous integration, and test-driven enhance design and development of complex systems. *DeveloperWorks, IBM*, 2012.

[3] Kent Beck. Aim, fire. *Software, IEEE*, 18(5):87 –89, sep/oct 2001.

[4] Kent Beck. *Test-Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[5] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 356–363, New York, NY, USA, 2006. ACM.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, Chichester, UK, 1996.

[7] Gerardo Canfora, Aniello Cimitile, Felix Garcia, Mario Piattini, and Corrado Aaron Visaggio. Evaluating advantages of test driven development: a controlled experiment with professionals. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 364–371, New York, NY, USA, 2006. ACM.

[8] D. Chelimsky, D. Astels, B. Helmkamp, Z. Dennis, D. North, and A. Hellesoy. *The RSpec Book: Behaviour Driven Development With RSpec, Cucumber, and Friends.* Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010.

[9] IEEE Computer Society. Test Technology Standards Committee, Institute of Electrical and Electronics Engineers, IEEE Standards Board, and IEEE Standards Association. *IEEE Std 1149.1-2001.* IEEE, 2001.

[10] P. Cordemans, S. Van Landschoot, and J. Boydens. *Test-Driven Development for Embedded Software: manual for industrial developers and academic instructors.* K.U.Leuven, 2011.

[11] L. Crispin. Driving Software Quality: How Test-Driven Development Impacts Software Quality. *Software, IEEE*, 23(6):70 –71, nov.-dec. 2006.

[12] Thomas Dohmke and Henrik Gollee. Test-Driven Development of a PID Controller. *Software, IEEE*, 24(3):44 –50, may-june 2007.

[13] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the Effectiveness of the Test-First Approach to Programming. *IEEE Trans. Softw. Eng.*, 31(3):226–237, March 2005.

[14] M. Fowler and M. Foemmel. Continuous integration, 2006. Viitattu 30.5.2012. Saatavissa: `http://www.martinfowler.com/articles/continuousIntegration.html`.

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[16] Martin Fowler. Mocks Aren't Stubs, 2007. Viitattu 28.3.2012. Saatavissa: `http://martinfowler.com/articles/mocksArentStubs.html`.

[17] S. Freeman and N. Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley signature series. Addison Wesley, 2009.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[19] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2004.

[20] J.W. Grenning. Applying test driven development to embedded software. *Instrumentation & Measurement Magazine, IEEE*, 10(6):20 –25, december 2007.

[21] J.W. Grenning. *Test-Driven Development for Embedded C*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2011.

[22] A. Gupta and P. Jalote. An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 285 –294, sept. 2007.

[23] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison Wesley Signature Series. Addison-Wesley, 2010.

[24] D.S. Janzen and H. Saiedian. On the Influence of Test-Driven Development on Software Design. In *Software Engineering Education and Training, 2006. Proceedings. 19th Conference on*, pages 141 –148, april 2006.

[25] D.S. Janzen and H. Saiedian. Does Test-Driven Development Really Improve Software Design Quality? *Software, IEEE*, 25(2):77 –84, march-april 2008.

[26] Ron Jeffries and Grigori Melnik. Guest Editors' Introduction: TDD–The Art of Fearless Programming. *Software, IEEE*, 24(3):24 –30, may-june 2007.

[27] Mike Karlesky, Mark VanderVoord, and Gred Williams. Ceedling dokumentaatio, 2010.

[28] Mike Karlesky, Mark VanderVoord, and Gred Williams. ThrowTheSwitch.org, 2011. Viitattu 10.7.2012. Saatavissa: `http://throwtheswitch.org/white-papers/`.

[29] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, New York, NY, USA, 1974. ACM.

[30] Kim Man Lui and Keith C.C. Chan. Test Driven Development and Software Process Improvement in China. In *Extreme Programming and Agile Processes in Software Engineering*, pages 219–222, 2004.

[31] Lech Madeyski and Lukasz Szala. The Impact of Test-Driven Development on Software Development Productivity - An Empirical Study. In Pekka Abrahamsson, Nathan Baddoo, Tiziana Margaria, and Richard Messnarz, editors, *Software Process Improvement*, volume 4764 of *Lecture Notes in Computer Science*, pages 200–211. Springer Berlin Heidelberg, 2007.

[32] Robert C. Martin. Professionalism and Test-Driven Development. *Software, IEEE*, 24(3):32 –36, may-june 2007.

[33] E.M. Maximilien and L. Williams. Assessing test-driven development at IBM. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564 – 569, may 2003.

[34] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[35] Roy Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[36] Roy Osherove. Unit Test - Definition, 2011. Viitattu 28.3.2012. Saatavissa: `http://artofunittesting.com/definition-of-a-unit-test/`.

[37] Kemppi Oy. DataGun tuotetiedot, 2012. Viitattu 24.7.2012. Saatavissa: http://www.kemppi.com/inet/kemppi/kit.nsf/0/4002ABB4C616FA00C22576460026917F?opendocument.

[38] Kemppi Oy. Tuoteluettelo 2012-2013, 2012.

[39] J.C. Sanchez, L. Williams, and E.M. Maximilien. On the Sustained Use of a Test-Driven Development Practice at IBM. In *AGILE 2007*, pages 5 –14, aug. 2007.

[40] Nancy Van Schooenderwoert and Ron Morsicato. Taming the Embedded Tiger - Agile Test Techniques for Embedded Software. In *Proceedings of the Agile Development Conference*, ADC '04, pages 120–126, Washington, DC, USA, 2004. IEEE Computer Society.

[41] M. Siniaalto and P. Abrahamsson. A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 275 –284, sept. 2007.

[42] Maria Siniaalto and Pekka Abrahamsson. Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study. In Bertrand Meyer, Jerzy R. Nawrocki, and Bartosz Walter, editors, *Balancing Agility and Formalism in Software Engineering*, pages 143–156. Springer-Verlag, Berlin, Heidelberg, 2008.

[43] John Ferguson Smart. *Jenkins - The Definitive Guide: Continuos Integration for the Masses: Also Covers Hudson.* O'Reilly, 2011.

[44] Mark VanderVoord. *Embedded Testing with Unity and CMock.* Throw-The-Switch Productions, 2010.

[45] Matti Vuori. Agile Development of Safety-Critical Software. Technical report, Tampere University of Technology, Department of Software Systems, 2011.

[46] S. Yenduri and L. A. Perkins. Impact of using test-driven development: a case study. *Proceedings of the 2006 International Conference on Software Engineering Research and Practice and Conference on Programming Languages and Compilers SERP'06*, 1, 2006.